Programmierte Evolution

Evolutionäre Algorithmen helfen bei der Lösung hochkomplexer Probleme und orientieren sich an biologischen Grundprinzipien. Die Genetische Programmierung ist ein populärer Vertreter aus der Sparte der Evolutionären Algorithmen, der seine Praxistauglichkeit auch bei der NASA unter Beweis gestellt hat.

Klaus Meffert

Bestimmte Klassen von Problemen lassen sich mit herkömmlichen Verfahren nicht oder nur mit unverhältnismäßig hohem Ressourceneinsatz (Zeit, Geld) lösen. Evolutionäre Algorithmen bieten eine Möglichkeit, hochkomplexe Probleme nach dem Vorbild der Evolutionstheorie zu enträtseln. Das Konzept der Genetischen Programmierung (kurz: GP) ist ein populärer Vertreter der Evolutionären Algorithmen. Im Artikel werden wir die Funktionsweise der GP darstellen und anhand eines Beispiels veranschaulichen. Wir beginnen mit der Einordnung der GP in das Gebiet der Künstlichen Intelligenz. Danach folgen konkrete Hinweise und Darstellungen, die bei der Implementierung eines GP in der Sprache ihrer Wahl als Grundlage dienen können.

Allgemeine Einführung

Die GP ist ein außergewöhnliches und leistungsfähiges Verfahren zur Problemlösung mittels zufällig, aber evolutionär hervorgebrachter Programme. Die Verfahren GP und Genetische Algorithmen (kurz: GA) reihen sich zusammen mit Klassischen Verfahren und Neuronalen Netzen in das Gebiet der Künstlichen Intelligenz ein (kurz: KI). Oft verwendet man auch die englische Bezeichnung »Artificial Intelligence«. Die KI beinhaltet das Lösen von Problemen durch Maschinen. Evolutionäre Verfahren, die eine Untermenge der KI darstellen, beschäftigen sich mit dem Lösen von Problemen durch Maschinen nach dem Vorbild der biologischen Evolution. Die GP ist ein Verfahren, das Prinzipien der Evolution auf ein Programm anwendet, dessen Befehle in einer Baumstruktur abgelegt sind, da diese sich als praktikabel herausgestellt hat. Damit grenzt sich die GP von den GA ab. Im Rahmen eines GA wird nicht ein Programm hervorgebracht, sondern eine Population von Chromosomen, die in ihrer Größe normalerweise statisch sind. Jedes Chromosom wiederum besteht aus einem oder mehreren Genen, den Trägern der Erbinformationen. Die konkrete Ausprägung eines Gens heißt Allel. Ein Allel kann auf verschiedene Art und Weise auf dem Computer repräsentiert werden, so etwa als Zahl oder Zeichenkette. Ein GA besitzt im Normalfall Allele fester Länge und unterscheidet sich somit vom GP (siehe auch Abbildung 1). Denn ein GP-Programm in Form einer Baumstruktur kann theoretisch beliebig viele Knoten und Blätter besitzen. In der Realität schränkt man deren Anzahl ein, um die Rechenzeit klein zu halten. Schon mit weit weniger als tausend Knoten und Blättern lassen sich hochkomplexe Programme schreiben, die zum Hervorbringen sehr guter Näherungslösungen für viele Probleme geeignet sind. Die höhere Flexibilität eines GP-Programms gegenüber einer GA-Population von Chromosomen bedeutet eine verbesserte Fähigkeit zur Lösung einer Vielzahl von Problemen.

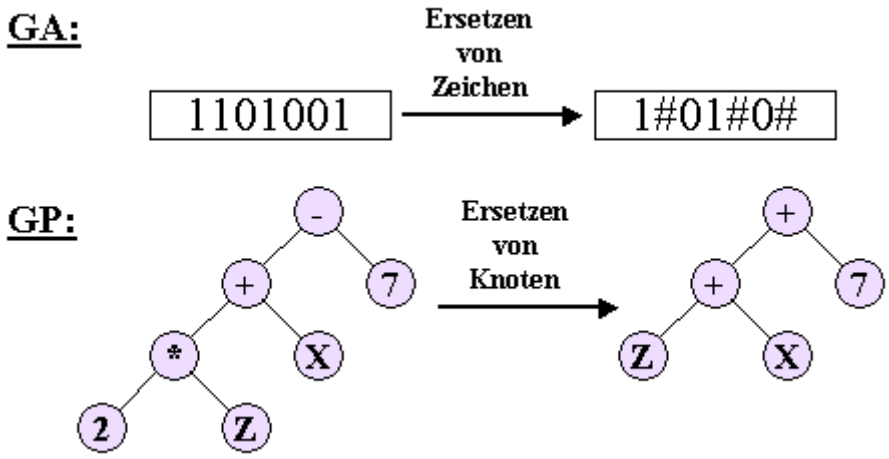

Abbildung 1: Differenzierung zwischen GA und GP

Eine naheliegende Frage lautet, welche Klassen oder Typen von Problemen geeignet sind, mit einem Algorithmus wie der GP bearbeitet zu werden. Eine eindeutige und gleichermaßen konkrete Antwort lässt sich auf diese allgemeine Frage nicht geben. Ganz generell formuliert, hilft die GP beim Lösen von Problemen, deren potentieller Lösungsraum sehr groß ist oder für deren Bearbeitung suboptimale Lösungen (Näherungslösungen) ausreichend sind. Voraussetzung für die Anwendung der GP ist es, ein Kriterium formulieren zu können, das bei der Beurteilung der Güte einer durch ein GP-Programm repräsentierten Lösung hilft. Dieses Gütekriterium wird oftmals auch als Fitnessfunktion bezeichnet. Diesem Grundprinzip der Evolution folgend, das von Charles Darwin ursprünglich populär wurde, fällt die GP in die Klasse von Algorithmen, die als „survival of the fittest“ (die Besten ihrer Rasse überleben) bezeichnet werden. Man kann den GP-Algorithmus gewissermaßen als automatischen Regelkreis bezeichnen, der auf stochastischen Prinzipien beruht, um Lösungen zu definierten Problemen zu finden.

Evolution auf dem Computer

Wir haben oben bereits einige Grundelemente von GA’s benannt: Chromosomen, Gene und Allele. Die GP führt diese Konstrukte fort und fügt darauf anwendbare Operationen hinzu. Diese Operationen heißen Replikation, Mutation und Crossing Over und dürften den meisten aus dem Biologieunterricht in der Schule vertraut sein. Anzumerken ist, dass die Grundelemente eines GA für die Implementierung eines GP verwendet werden können, ein GA also durch Einführung von strukturellen Erweiterungen in das GP-Paradigma überführt werden kann.

Im Prinzip kann man vereinfachend ein GP-Programm als Chromosom, die Elemente eines GP-Programms als Gene und deren konkrete Ausprägung als Allele bezeichnen. Eine Mehrzahl konkurrierender GP-Programme stellt dann die Population einer bestimmten Generation dar. Diese Analogie leuchtet ein, spricht man doch auch oft beim Menschen vom genetischen Code, von einem Programm, das in unserem Erbgut kodiert ist.

Der biologische Vorgang der Evolution muß nun auf den Computer übertragen werden. Dazu bedarf es einer geeigneten Repräsentation der biologischen Elemente und der darauf ausführbaren Operationen. Aus der Informatik bekannt und hilfreich ist der Begriff des abstrakten Datentyps (kurz: ADT). Ein ADT stellt ein Alphabet und eine Menge darauf ausführbarer Operationen dar. Der Begriff Alphabet ist hier in einer anderen Bedeutung als der umgangssprachlichen zu verstehen. Gemeint ist die mathematische Verwendung. Ein Alphabet repräsentiert demnach eine definierte Menge von Zeichen. Ein ADT eignet sich zur Abbildung eines Chromosoms und der darin enthaltenen Gene. Für das GP-Programm (Chromosom) und

die darin enthaltenen Befehle (Gene) wird also jeweils eine eigenständige objektorientierte Klasse vorgesehen. Für die Implementierung der Baumstruktur des GP-Programms bietet fast jede Programmiersprache entsprechende Konstrukte. Im schlimmsten Fall muß der Entwickler lediglich eine Liste von Listen verwalten, um eine dem Baum äquivalente Struktur zu erhalten. Die erste Liste beinhaltet nur das Wurzelelement (den zuoberst stehenden GP-Befehl). Sie zeigt auf ihre Kindknoten, die wiederum in einer eigenen, gemeinsamen Liste untergebracht sind.

Jeder Knoten im Baum stellt einen Befehl dar. Jedes Blatt im Baum (also die untersten Knoten ohne weitere Kinder) repräsentiert entweder einen Befehl (eine Anweisung) ohne Parameter oder einen Eingabeparameter für einen Befehl. Die linke Seite unten in Abbildung 1 zeigt eine solche Programmrepräsentation. Intern arbeitet man natürlich nicht mit einer graphischen Repräsentation, sondern etwa mit einer Zeichenkettenrepräsentation. So kann das im unteren Teil von Abbildung 1 dargestellte Programm äquivalent mit

```
- + * 2 Z X 7
```

geschrieben werden (X und Z sind Eingabewerte). Man erhält diese Schreibweise, indem man, beim Wurzelknoten beginnend, alle linken Kinder abläuft und der Reihe nach notiert. Ist ein Blatt erreicht, schreitet man zurück, bis rechts ein Weg möglich ist und steigt diesen gleichermaßen herab. Ursprünglich stammt diese Schreibweise aus den Anfängen der GP-Algorithmen, die John Koza begründet hat. Hintergrund war die damals für die GP verwendete Sprache LISP, in der so genannte S-Ausdrücke existieren. Sie haben eine analoge Form, wie oben angegeben. In alltäglicher Notation würde man schreiben:

```
2Z + X - 7
```

Die Abarbeitungsreihenfolge von GP-Befehlen ergibt sich also gemäß dem eben dargestellten aus deren Position im Baum.

Programme dynamisch schreiben

Um nun dynamisch Programme schreiben zu können, müssen wir dem GP-Algorithmus eine Menge zulässiger Befehle zur Verfügung stellen. Das geschieht am einfachsten dadurch, dass wir ein Interface *IGPCommand* vorsehen (siehe Abbildung 3). Jede Klasse, die dieses Interface implementiert, fügt unserer Bibliothek verfügbarer Befehle einen weiteren hinzu. Weiterhin benötigen wir für viele Befehle Eingabeparameter. Ein Eingabeparameter kann entweder dynamisch berechnet werden und als Ergebnis einer Operation (also eines anderen GP-Befehls) zustande kommen. Oder es handelt sich um sogenannte Terminale. Das sind Unveränderliche, die einmal definiert und nie mehr modifiziert werden. Beispielsweise kann es sich um bedeutende Zahlen aus der Mathematik handeln, so etwa die Ludolfsche Zahl Pi (3,14159...), die eulersche Zahl (2,71828...) oder trivialerweise um die Zahl 2, die kleinste Primzahl und gerade Zahl.

Liefert jeder definierte GP-Befehl eine Ausgabe, die als Eingabe für jeden anderen GP-Befehl dienen kann, ist es einfach möglich, die syntaktische Korrektheit eines durch genetische Operationen neu entstandenen Programms festzustellen. Das ist wichtig, um unnötige Versuche zu vermeiden, die in größerer Anzahl ansonsten negativ auf die Zeitkomplexität wirkten.

GP-Programme setzen sich also aus einem vorher definierten Vorrat an Befehlen zusammen. Gemäß der von Darwin bekannten Prinzipien Replikation, Mutation und Crossing Over entstehen in jeder Generation neue Programme. Diese neuen Programme gehen demnach hervor aus deren Vorgängern durch Übernahme und Modifikation vorhandener Anweisungen und Anweisungszweige sowie durch zufälliges Einsetzen neuer Anweisungen und Zweige. Es leuchtet ein, dass ein Programm umso leistungsfähiger sein kann, desto größer der Befehlsvorrat ist. Je größer die Anzahl verfügbarer unterschiedlicher Befehle aber ist, umso komplexer und somit zeitaufwendiger wird die Suche nach einem für eine Problemstellung geeigneten Programm. Dank der rasanten Entwicklung der Prozessorleistung werden evolutionäre Verfahren wie GA oder GP für den praktischen Einsatz allerdings immer attraktiver.

Genetische Operationen

Besonders einzugehen ist auf die genetischen Operationen Crossing Over und Mutation. Die Replikation ist vorneweg schnell erklärt: Ein Zweig (Teilbaum) eines GP-Programms wird anhand eines Wurzelknotens ausgewählt. Dann wird eine passende Einfügestelle für diesen zufällig selektierten Zweig im Zielprogramm bestimmt und der Zweig an diese Stelle kopiert. Crossing Over ist etwas komplizierter und in Abbildung 2 schematisiert und in Abbildung 3 in eine Klassenstruktur eingeordnet. Beim Crossing Over bedarf es der Betrachtung zweier GP-Programme (in der Abbildung 2 mit A und B bezeichnet) und pro Programm je eines Teilbaums. Im Programm, das dem Crossing Over unterliegt, wird von diesem Teilbaum wieder ein Teilbaum ausgewählt und entfernt. An jener Stelle, wo der Teilbaum entfernt wurde, wird nun ein Teilbaum aus dem zweiten Programm eingefügt. Es entsteht ein neues Programm, in der Abbildung mit A' bezeichnet.

Im Rahmen der Mutation wird ähnlich verfahren. Jedoch wählt man nicht einen Teilbaum eines bestehenden GP-Programms als Substitut für einen Teilbaum eines anderen GP-Programms aus, sondern bestimmt zuerst zufällig eine Mutationsstelle in einem Programm. Als nächstes generiert der GP-Algorithmus wiederum zufällig einen Teilbaum. Er wird an der Mutationsstelle eingefügt. Der zufällig generierte Teilbaum kann entweder vollkommen neu erzeugt werden oder durch zufällige Modifikation eines bestehenden Teilbaums.

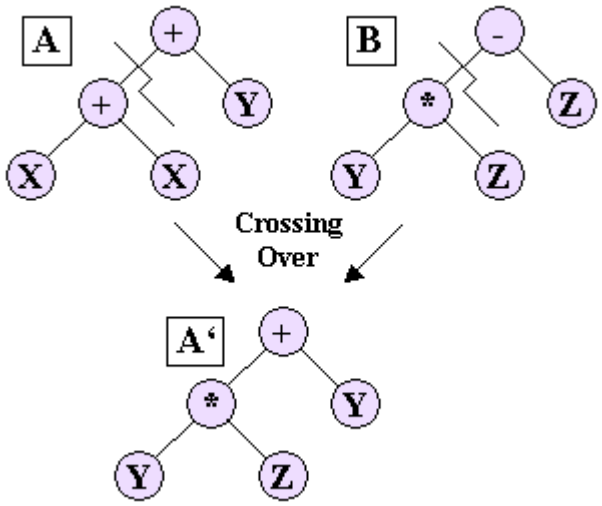

Abbildung 2: Crossing Over im Rahmen der GP

Die Fitnessfunktion

Die Fitnessfunktion ist der wohl wichtigste Bestandteil eines Evolutionären Algorithmus und soll hier näher betrachtet werden. Eine Fitnessfunktion kann wahlweise bevor oder nachdem der Grundstock an Befehlen und Terminalen definiert ist, aufgestellt werden (wie man möchte). Eine Fitnessfunktion ist ein Evaluator eines GP-Programms. Sie weist jedem Programm durch Vergabe eines Fitnesswertes einen Rang zu. Je besser der Rang, desto besser die Eignung des bewerteten Programms, das mit der Fitnessfunktion umschriebene Problem zu lösen. Es ist dem Programmierer überlassen, ob ein Fitnesswert von null das Optimum darstellt oder ob ein höherer Wert für bessere Ergebnisse steht. Die Wahl hängt mit vom vorliegenden Problem ab. Kennt man eine Formel, die Abweichung vom Optimum zu ermitteln, kann eine Fitness mit dem Wert null als Optimum angesetzt werden, indem sie als Abweichung vom besten Ergebnis interpretiert wird. Will man statt dessen den Fitnesswert verwenden, um beispielsweise den Wirkungsgrad eines elektrischen Filters zu beschreiben, eignet sich eine Fitnessfunktion, die höhere Werte für bessere Leistungen vergibt. Letztendlich ist die Fitnessfunktion eine Methode, die als Eingabe ein GP-Programm erhält, es mit Eingaben füttert, das Programm ablaufen lässt und die Ausgabe des Programms auswertet.

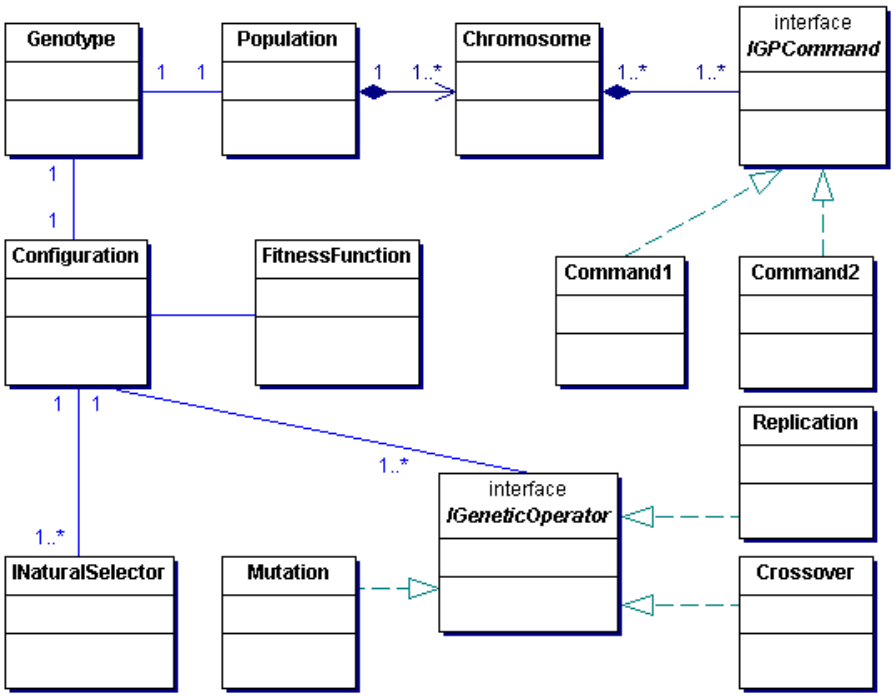

Abbildung 3: Klassendiagramm: Basisklassen einer GP-Implementierung

Gegeben die Fitnessfunktion und alles, was man zum kreieren eines GP-Programms benötigt, kann der Algorithmus ausgeführt werden. Kasten 1 zeigt die Arbeitsweise eines GP-Algorithmus im Pseudocode. Die Ähnlichkeit mit einem GA ist erkennbar. Die GP muss aber im Gegensatz zum GA ein Programm flexibler Struktur hervorbringen.

Weil die Fitnessfunktion essentieller Bestandteil eines Evolutionären Algorithmus wie der GP ist, kann ein Problem auf evolutionäre Weise nur gelöst werden, wenn sich eine geeignete Fitnessfunktion dafür aufstellen lässt. Insbesondere Probleme, die eine große Anzahl an potentiellen Lösungsmöglichkeiten bieten und deren Lösung schwer fassbar ist, eignen sich.

Zur Veranschaulichung zeigen wir im nächsten Absatz, welche Probleme sich konkret mit Hilfe der GP definieren und lösen lassen.

Patent, Patent

Genau wie für jeden Algorithmus und für jedes Konzept gibt es geeignetere und weniger geeignete Anwendungsgebiete für die GP. In der Praxis umgesetzte Resultate sind der beste Indikator für die Anwendbarkeit des Ansatzes. Darüber hinaus ist es in der Vergangenheit oft gelungen, zahlreiche Patente mit Hilfe der GP neu zu “erfinden”. Dabei muss man berücksichtigen, dass eine derartige Wieder-Entdeckung durch den Benutzer des GP-Ansatzes forciert wird und einen dementsprechend niedrigeren Stellenwert hat als die ursprüngliche Entdeckung eines patentierungswürdigen Sachverhalts durch einen menschlichen Erfinder. Statt von einem Patent oder einer Innovation spricht man im Rahmen einer Wiederentdeckung eines schon bestehenden Patents durch einen GP-Algorithmus eher von einer Deduktion.

Besonders im Bereich der Elektrotechnik gibt es zahlreiche Problemstellungen, die sich mit der GP gut lösen lassen. Ein Grund ist die Verfügbarkeit leistungsfähiger Simulationswerkzeuge, wie etwa PSPICE für elektrotechnische Schaltungen. Dementsprechend konnten viele im Alltag brauchbare elektrotechnische Schaltungen durch Evolution hervorgebracht werden (darunter verschiedenste Filter, Controller, Verstärker und berechnende Schaltkreise). Die NASA hat für den Weltraum eine Antennen mittels Evolutionärer Algorithmen hervorgebracht (siehe Link). Schaut man sich das Bild der Antenne an, wird

deutlich, dass ein menschlicher Erfinder sicher eine andere (wahrscheinlich weniger effiziente) Lösung hervorgebracht hätte.

Ein fiskalisch interessantes Beispiel für die Anwendbarkeit des Verfahrens ist die Vorhersage von Aktienkursen. Dieses Problem ist gewissermaßen atypisch, denn hier ist die Schwierigkeit nicht das Finden einer Fitnessfunktion. Vielmehr gilt es, alle in Frage kommenden Parameter zu identifizieren und quantitativ zu erfassen. Die Fitness wird einfach gemessen, indem ermittelte Istwerte mit schon bekannten Aktienkursen verglichen werden.

Die am Ende des Artikels angegebene Internetadresse zur Homepage von Koza eröffnet mannigfaltige Ressourcen zu Anwendungsfällen, die hauptsächlich der Elektrotechnik zuzuordnen sind. Die ebenfalls unten angeführte URL zum Thema Roboterfußball unterstreicht das breit gefächerte Einsatzspektrum evolutionärer Verfahren.

Im Fokus der Forschung

Neben den Grundkonzepten des GP-Paradigmas (Operatoren wie Replikation und Bausteine wie Chromosomen) sind durch intensive Forschungen, initiiert durch John Koza, weitergehende Konzepte zur optimierten Lösungssuche durch die GP entstanden. Ein häufig eingesetztes Verfahren ist die Anwendung von sogenannten „Automatisch Definierten Funktionen“ (kurz: ADF). Die Idee dahinter ist recht simpel: Wiederkehrende Befehlssequenzen, also Teilbäume eines Programmgraphen, werden durch eine Art Makro ersetzt. Statt also etwa fünf Befehle im Graphen zu platzieren, wird anstelle dessen eine ADF in Form eines Makros eingesetzt. Die ADF stellt weiterhin Plätze für benötigte Eingabeparameter bereit. Somit reduziert sich die Komplexität des Programms, weil dessen Anzahl an Anweisungen reduziert wird. Das ermöglicht schnellere Berechnungen und somit im Allgemeinen bessere Ergebnisse. Zudem helfen ADFs durch deren referenzierenden Charakter beim Aufbau wiederverwendbaren Strukturen. Gleichermaßen stellen Makros einen kompakten Befehlsblock dar, dessen Komposition sich evolutionär herausgebildet hat und dem ein entsprechend hoher Stellenwert zugeordnet werden kann.

Einen ähnlichen Charakter wie ADFs haben Mechanismen, die dem als Code Bloating bezeichneten Phänomen entgegenwirken sollen. Code Bloating ist das Aufblähen eines Programms durch unnütze Anweisungen. Das sind entweder Anweisungen ohne Auswirkungen oder solche, die nie erreicht werden. Die dem entgegenwirkenden Mechanismen sind komplex im Aufbau und sollen hier nicht weiter beschrieben werden.

Darüber hinaus, gibt es im Rahmen der GP weitere Operationen, die sich architekturverändernd auswirken:

Bei der *Duplikation von Subroutinen* wird eine Subroutine zufällig ausgewählt und an eine andere Stelle im Programmbaum kopiert. Falls ADFs in der Subroutine vorkommen, können diese im Klon zufällig modifiziert werden, um die ursprüngliche Subroutine leicht unterschiedlich vom Klon zu gestalten.

Duplikation von Argumenten: Innerhalb einer Routine wird ein Argument dieser Routine dupliziert, natürlich nur, wenn die maximale Anzahl zulässiger Argumente dabei nicht überschritten wird. Gleichzeitig sorgt der Algorithmus dafür, dass alle Aufrufe dieser Routine entsprechend angepasst werden.

Erzeugung von Subroutinen: Eine bestehende Subroutine wird an eine andere Stelle im Programmbaum kopiert. Dort werden dann weitere Elemente eingefügt, um die Programmsyntax regelkonform zu gestalten. Alternativ ist es auch möglich, die bestehende Subroutine nicht zu kopieren, sondern zu verschieben. In diesem Fall ist zusätzlich eine syntaktische Anpassung an der ursprüngliche Stelle der bestehenden Subroutine nötig.

Weitere mögliche Operation sind beispielsweise *Löschen von Subroutinen* und *Löschen von Argumenten*. Sie sind analog zu *Erzeugung von Subroutinen* und *Duplikation von Argumenten*. Prinzipiell ist es der Phantasie des Entwicklers überlassen, welche Operationen er dem GP-Algorithmus zur Hand gibt. Man muss wie bei jedem Programm nur darauf achten, nicht übermäßig kreativ zu sein.

Eine häufig angewandte Praxis zur Selektion der fittesten Individuen einer Population ist die so genannte Tournierselektion (engl.: *Tournament Selection*). Normalerweise werden die besten Individuen durch direkten Vergleich mit allen anderen ermittelt. Die Tournierselektion bildet willkürliche Gruppen meist gleicher Größe und lässt – bildlich gesprochen – die in einer Gruppe befindlichen Individuen im gegeneinander kämpfen. Der Sieger geht mit den Siegern der anderen Tourniere in die nächste Runde, wird also für die Bildung der folgenden Generation berücksichtigt.

Das Konzept der sogenannten Kultur stellt einen fortgeschrittenen Ansatz zur generationsübergreifenden Übermittlung von Informationen dar. Kultur, oder Culture, wie Experten das Konzept im Englischen nennen, ist ein globaler Speicher. Er erinnert an ein Langzeitgedächtnis, welches das nur eine Generation bestehende Kurzzeitgedächtnis ergänzt. Vorteil dieses quasi persistenten Informationsträgers ist die Weitergabe von Erfahrungswerten an neue Generationen.

Anleitung zur Implementierung eines GP

Der in Kasten 1 dargestellte Algorithmus dient als Grundlage für jede GP-Implementierung. Das dort beschriebene Abbruchkriterium wird erreicht, wenn entweder eine willkürlich festgelegte Anzahl von Generationen überschritten wurde oder wenn die errechnete Fitness des besten Individuums den Ansprüchen genügt. Die Fitnessfunktion ist stark problemabhängig. Oft reicht es, eine fixe Tabelle von Wertepaaren (festgelegter Eingabewert, erwarteter Ausgabewert) zu definieren und in einer Schleife alle Eingaben in das GP-Programm zu stecken, dessen Fitness evaluiert werden soll. Die Soll-Ausgabewerte werden mit den Ist-Ausgabewerten verglichen und dann wird die Abweichung über die Schleife summiert. Je geringer die Abweichung, desto besser. Wie oben beschrieben, gibt es auch Probleme, für die ein Wirkungsgrad maximiert werden soll. Hier muss eine entsprechende Formel implementiert werden, etwa eine, die aus der Elektrotechnik bekannt ist, handelt es sich um ein Schaltungsproblem.

Kasten 1: Pseudocode Genetische Programmierung

```
Bestimme initiale Zufallsbelegung für Population von GP-Programmen.
Führe folgende Schritte wiederholt aus, bis Abbruchkriterium erreicht:
          • Bewerte die Programme der Population (berechne Fitness).
          • Mittels Reproduktion, Crossing Over und Mutation, erstelle neue
            Population von Programmen aus der aktuell vorliegenden.
          • Wende evtl. spezielle Verfahren zur Optimierung der entstandenen
            GP-Programme an, um etwa Code Bloating zu reduzieren oder
            unzulässige Programme zu vermeiden.
Das Programm mit dem höchsten Fitnesswert stellt die augenblicklich beste Lösung
für das vorliegende Problem dar. GP bringt dementsprechend hauptsächlich sehr
gute Näherungslösungen hervor (suboptimale Lösungen).
```

Die wichtigsten zu implementierenden Klassen können Abbildung 3 entnommen werden und sollen im folgenden erklärt werden. Die zentrale Klasse in der Graphik ist *Genotype*. Ein Genotyp besteht aus einer Menge von Chromosomen (Klasse *Chromosome*), die zusammen eine Population (Klasse *Population*) bilden. Die Population ist Bestandteil des Genotyps, der den internen Zustand eines Individuums darstellt. Pro Chromosom gibt es mindestens einen, maximal beliebig viele Kommandos (Interface *IGPCommand*). Der Entwickler muss problemspezifisch entsprechende GP-Kommandos implementieren (in der Abbildung mit *Command1* und *Command2* angedeutet).

Der Genotype besitzt weiterhin eine Referenz auf ein Konfigurationsobjekt (Klasse *Configuration*). Dieses enthält alle notwendigen Informationen, um eine Generation zu berechnen. Insbesondere gehören dazu die Selektoren (Interface *INaturalSelector*). Sie bestimmen, welche Chromosomen aus der aktuellen in die nächste Population übernommen werden. Normalerweise wählt der Selektor diejenigen mit der besten Fitness aus, beispielsweise die besten 10% der gesamten Population. Auf die gewählten Kandidaten werden dann

genetische Operationen angewandt (Interface *IGeneticOperator*), so etwa die bekannten Replikation, Mutation und Crossing Over (siehe Abbildung für entsprechende Klassen). Die Operationen werden so oft ausgeführt, bis eine festgelegte Anzahl von Chromosomen für die nächste Population zustande gekommen ist. Als Sammelobjekt für Referenzen besitzt die Konfiguration auch einen Verweis auf die Fitnessfunktion (siehe Abbildung 3). Somit ist der GP-Algorithmus in der Lage, mit Hilfe der Genotyp-Klasse alle nötigen Berechnungen durchführen zu können.

Genau wie für viele andere Verfahren bietet es sich auch bei der GP an, auf bereits vorhandene Frameworks zurückzugreifen. Das erleichtert die Realisierung und Implementierung einer Problemlösung mittels der GP ganz erheblich. Am Ende dieses Artikels sind diverse Links angeführt, darunter auch Verweise auf Frameworks für die Umsetzung eigener Problemlösungsstrategien mittels der GP.

Fazit

In diesem Artikel haben wir Charakteristik und Arbeitsweise der GP dargestellt. Mit Hilfe der vermittelten Grundlagen und den dargestellten Grundbestandteilen einer GP-Implementierung steht einer Implementierung in einer Sprache ihrer Wahl nichts mehr im Wege.

Nach eingehender Betrachtung des Paradigmas der GP und dem Vergleich mit der GA, beide Verfahren aus dem Gebiet der Künstlichen Intelligenz und im speziellen Vertreter der Evolutionären Algorithmen, ist es angebracht, ein gemeinsames Fazit zu ziehen. Im Vergleich schneidet die GP gegenüber den GA's insofern besser ab, als dass sie praktisch die Verarbeitung komplexerer Probleme erlaubt und somit ein größeres Anwendungsgebiet abdeckt. Beide Verfahren orientieren sich an denselben Grundkonzepten, nämlich denen der Evolution. Dementsprechend lassen sich nur bestimmte Arten von Problemen mit GA's und GP's lösen. Gelingt es jedoch, eine geeignete Fitnessfunktion für ein gegebenes Problem aufzustellen, bringen die Verfahren mit hoher Wahrscheinlichkeit und fast schon wie auf wundersame Weise eine oft nichttriviale und außergewöhnliche Lösung hervor. Hierin liegt die Stärke Evolutionärer Algorithmen. Die aktuelle Forschung beschäftigt sich damit, Antworten auf spannende Fragen zu finden, wie etwa: „Wie kann man zu einem gegebenen Problem(kreis) automatisch eine Fitnessfunktion entstehen lassen?". Die Antwort hierauf wurde noch nicht abschließend gefunden. Aktuelle Arbeiten lassen jedoch erahnen, dass der Mensch - genau wie die Natur dies mit den Naturgesetzen vollbracht hat - eine gültige Antwort konkretisieren kann.

Links & Literatur

- JGAP, ein Framework für Java: http://jgap.sourceforge.net/
- Genetic Programming Engine (Delphi, C++): http://sourceforge.net/projects/gpengine/
- Homepage von John Koza: http://www.genetic-programming.com/
- Roboter spielen Fußball: http://www.teambots.org/
- NASA: Antenne für den Weltraum: http://www.spaceref.com/news/viewpr.html?pid=14394
- Klaus Meffert: Genetische Algorithmen mit Java – Auf Darwins Spuren, in *Java Magazin* 08/2004