Programmierte Evolution

Das Konzept der Genetischen Programmierung hilft bei der evolutionären Entwicklung ungewöhnlicher Lösungen. Selbst Organisationen wie die NASA greifen auf dieses Paradigma zurück, um Probleme auf außergewöhnliche Weise zu lösen.

Klaus Meffert

Verfahren der Künstlichen Intelligenz

Aufbauend auf den Artikel im vorigen Heft zum Thema Genetische Algorithmen (GA), knüpfen wir mit der Vorstellung der Genetischen Programmierung (GP) weiterführend an. Die GP ist ein außergewöhnliches und leistungsfähiges Verfahren zur Problemlösung mittels zufällig, aber evolutionär hervorgebrachter Programme. Die Verfahren GA und GP reihen sich zusammen mit Klassischen Verfahren und Neuronalen Netzen in das Gebiet der Künstlichen Intelligenz ein. Oft verwendet man auch die englische Bezeichnung »Artificial Intelligence« oder kurz AI. AI beinhaltet das Lösen von Problemen durch Maschinen. Evolutionäre Verfahren, die eine Untermenge der AI darstellen, beschäftigen sich mit dem Lösen von Problemen durch Maschinen nach dem Vorbild der biologischen Evolution. Die GP ist ein evolutionäres Verfahren, das Evolution auf ein Programm anwendet, dessen Befehle in einer Baumstruktur abgelegt sind.

Zur Erinnerung: GAs lösen Probleme durch evolutionäres Hervorbringen von Chromosomen. Deren Erzeugung wird anhand einer Bewertungsregel, die Fitnessfunktion genannt wird, nach dem biologischen Vorbild gesteuert. Die Ausprägung der Chromosomen wird dann je nach Problemstellung interpretiert. GAs operieren somit im Prinzip analog zu den vom Evolutionsforscher Charles Darwin entwickelten Theorien. Die Genetik wird gemäß dem biologischen Vorbild also im Sinne eines automatischen (gewissermaßen stochastischen) Regelkreises genutzt, um Probleme zu lösen. GAs basieren, wenn man es allgemein formulieren will, auf fixen, strukturell sich nicht weiter entwickelnden Programmen.

Genetische Programmierung

Das Konzept der GP geht einen Schritt weiter. Anstatt Chromosomen zu erzeugen, zu interpretieren und zu bewerten, werden dynamische Programme „geschrieben“. Diese Programme setzen sich 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 also 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.

Wie bei GAs gibt es auch in der GP eine Fitnessfunktion. Sie beurteilt die Eignung eines evolutionär entwickelten Programms zur Lösung eines vom Menschen vorgegebenen Problems. Deshalb ist ein Problem auf evolutionäre Weise nur lösbar, 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 besonders. 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.

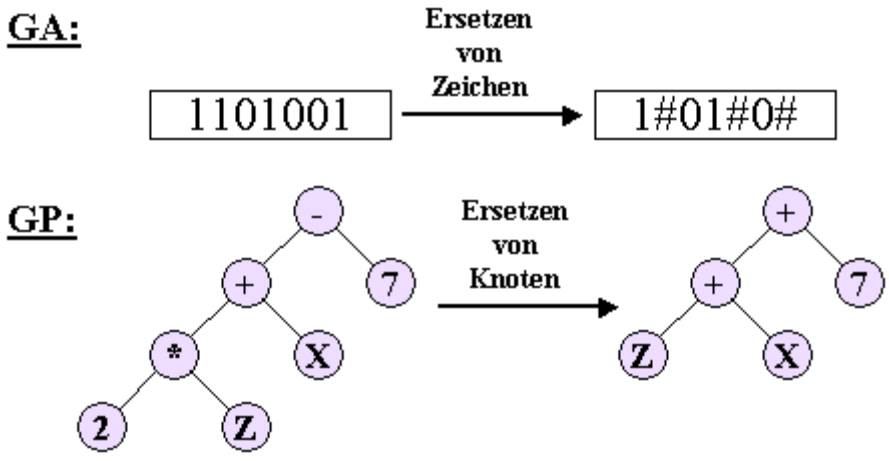
Besonders im Bereich der Elektrotechnik gibt es zahlreiche Problemstellungen, die sich mit der GP gut lösen lassen. Ein Grund ist die Verfügbarkeit leistungsfähiger Simulationswerkzeuge, wie etwa PSPICE für elektrotechnische Schaltungen. Dementsprechend konnten viele im Alltag brauchbare elektrotechnische Schaltungen durch Evolution hervorgebracht werden (darunter verschiedenste Filter, Controller, Verstärker und berechnende Schaltkreise).

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 URL 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.

Grundlagen der GP

Zur Veranschaulichung der fundamentalen biologischen Grundprinzipien der GP sei auf den vorigen Artikel dieser Reihe verwiesen. Wir beschränken uns auf die Darstellung der bei der GP neu hinzu gekommenen Konzepte im Vergleich mit GAs. Ein Grundverständnis von Replikation, Crossing Over und Mutation setzen wir voraus. 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. In Abbildung 1 ist prinzipiell der Unterschied zwischen einem GA und der GP dargestellt.



Es hat sich als geeignet herausgestellt, bei der GP ein Programm in einer Baumstruktur abzulegen. 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
```

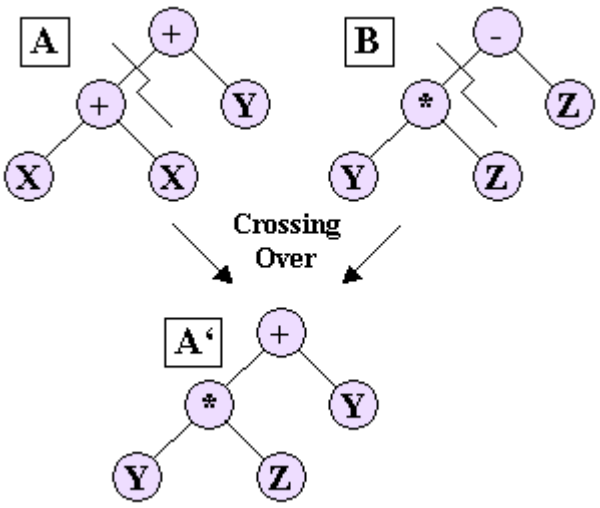
geschrieben werden (X und Z sind Eingabewerte). Man erhält diese Schreibweise, indem man, beim Wurzelknoten beginnend, alle linken Kinder abläuft und der Reihe nach notiert. Ist ein Blatt erreicht, schreitet man zurück, bis rechts ein Weg möglich ist und steigt diesen gleichermaßen herab. Ursprünglich 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
```

Ein GP-Programm besteht lediglich aus zwei Typen von Symbolen: Aus einem Set von Funktionen und aus einem Set von Terminalen. Funktionen sind arithmetische Operationen, Verzweigungsoperatoren und auch alles, was aus einer Eingabe eine Ausgabe erzeugt. Terminale sind externe Eingaben (unabhängige Variablen) sowie numerische Konstanten.

Das Generieren eines Programms geschieht dadurch, dass aus einem Vorrat möglicher Symbole (Funktionen, Terminale) zufällig welche ausgewählt und zusammengestellt werden. Die Auswahl und Zusammenstellung dieser Befehle unterliegt jedoch nicht vollends dem Zufall, sondern auch gewissen Beschränkungen und Optimierungen. Das Anfügen von Befehlen an vorhandene Sequenzen findet beispielsweise so statt, dass nur sinnvolle und mögliche Befehle berücksichtigt werden. Manche Befehle wollen zudem mit Parameterobjekten versorgt werden. Je nach Anzahl und Art dieser Eingabeparameter wählt der Algorithmus zulässige Parameter aus. Dies können, genau wie beim GA, etwa Wahrheitswerte, Zahlen oder Zeichenketten sein. Die Interpretation eines solchen Eingabeparameters obliegt dabei alleine dem zugrunde liegenden Befehl. Sie ist letztendlich Sache des Programmierers, der die Befehlsabarbeitung implementieren muss.

Besonders einzugehen ist gleich auf die genetischen Operationen Crossing Over und Mutation. Die Replikation ist dagegen 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.



Beim Crossing Over bedarf es der Betrachtung zweier GP-Programme (in der Grafik 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.

Weitergehende Konzepte

Neben den eben geschilderten Grundkonzepten 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.

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.

ECJ – Ein GP Framework

Genau wie für einen GA 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. Der wohl populärste Vertreter von GP-Frameworks ist ECJ von Sean Luke. Der Link auf die Homepage des Autors ist am Ende des Artikels angegeben. ECJ ist Open Source und somit frei erhältlich. Die spezielle Lizenzvereinbarung erlaubt es, ECJ auch ohne Freilegen des eigenen Quelltextes zu verwenden. Es handelt sich also dankenswerterweise nicht um eine sogenannte „virale Lizenz“. Eine virale Lizenz beinhaltet nämlich das Gebot, selbst erstellten Quelltext, der das Lizenzprodukt verwendet, ebenfalls unter dieser viralen Lizenz öffentlich zugänglich zu machen.

ECJ beinhaltet Klassen, die die von Koza geprägten Verfahren und Ideen direkt bereitstellen. Die allgemeinen GP-Konzepte (Populationsbildung, Mutation etc.) sind selbstverständlicher Bestandteil des Paketes.

Das ECJ Package enthält bereits einige Beispiele mit Quelltext. Auf der Homepage von ECJ ist ergänzend eine umfangreiche Linkliste zu anderen GP-Frameworks und –Systemen zu finden. Wir wollen eines der Beispiele näher betrachten. Gleichzeitig zeigen wir damit die Konzepte von ECJ auf und erläutern die Verwendung des Frameworks für den Entwickler.

Der Rasenmähermann

Zur Veranschaulichung, wie man mit ECJ ein Problem definieren und mit Hilfe des GP-Konzeptes eine Lösung finden kann, sei folgende Aufgabe gegeben: Eine vorgegebene Rasenfläche soll mit so wenigen Operationen wie möglich gemäht werden. Die für den Rasenmäher zur Verfügung stehenden atomaren Tätigkeiten beinhalten Kommandos wie *mähe* oder *drehe dich nach links*. Zur Reduzierung der Komplexität des Problems kann der Rasenmäher an einem Ende des Rasens (oben, unten, links, rechts) herausfahren und landet auf der gegenüber liegenden Seite. Eine realitätsnahe Definition des Problems ist ohne weiteres ebenfalls mit der GP lösbar, verringert aber die Anschaulichkeit des Beispielprogramms.

Unser exemplarisches Problem ist in Package *ec.app.lawnmower* definiert. In diesem Package sind alle problemspezifischen Java-Klassen abgelegt. Auch die problemspezifische Konfigurationsdatei ist Teil des Packages. Wir gehen im Folgenden auf die Bestandteile des so genannten Rasenmäher-Projekts ein und schildern danach den Ablauf des GP-Algorithmus an diesem Beispiel.

Problemklasse

Als Problemklasse wird die Klasse bezeichnet, die im wesentlichen das Setup vornimmt und die Fitnessfunktion bereitstellt. Die Fitnessfunktion bewertet die Güte eines Individuums. Ein Individuum ist hier ein komplettes GP-Programm (ein Programmbaum also), das aus den durch den Entwickler bereit gestellten Funktionen besteht. Für unser Beispiel ist die Problemklasse *ec.app.lawnmower.LawnMower*. Sie ist eine Unterklasse von *ec.gp.GPProblem*. Jede Klasse, die ein GP-Problem lösen soll, erbt von dieser Klasse. Gleichzeitig implementiert *ec.app.lawnmower.LawnMower* das Interface *ec.simple.SimpleProblemForm*. Dieses Interface ist für alle Probleme geeignet, die bei ihrer Bewertung nur einzelne Individuen betrachten und nicht Gruppen von abhängigen Individuen.

GP-Funktionen

Die für den GP-Algorithmus verwendbaren Befehle sind im Package *ec.app.lawnmower.func* abgelegt. Alle dort befindlichen Klassen sind (direkt oder indirekt) von *ec.gp.GPNode* abgeleitet. Eine *GPNode* ist ein abstrakter Knoten in einem GP-Programmbaum. In der Konfigurationsdatei werden die eben genannten GP-Funktionen deklariert. Das GP-

Framework weiß somit, welche Funktionen für die GP-Bearbeitung zur Verfügung stehen. Alternativ hätte man die Möglichkeit vorsehen können, ein Java-Package anzugeben, in dem alle verfügbaren Funktionen deklariert sind. Das hätte aber den Nachteil gehabt, bestimmte Sets von GP-Funktionen nicht dynamisch (je nach Problemstellung) zusammenstellen zu können. In der ADF-Variante (*adf.params*) sieht man zudem den Vorteil des deklarativen Vorgehens: Man kann eine ADF explizit definieren, indem man der ADF alle in ihr enthaltenen *GPNode*s zuweist, die als Subroutinen eines Unterprogramms fungieren.

Konfigurations- bzw. Parameterdatei

Die Konfigurations- oder Parameterdatei enthält alle für das Problem relevanten Parameter. Für den Rasenmähermann beinhaltet ECJ zwei exemplarische Parameterdateien. In der Datei *src/ec/app/lawnmower/noadf.params* werden die für den Rasenmähermann verfügbaren Kommandos (*mähe*, *drehe dich nach links* etc.) in simpler Form deklariert. Dahingegen deklariert die im selben Verzeichnis liegende Datei *adf.params* zusätzlich zwei ADFs. Somit ist es möglich, bei der Lösungsfindung im Vorfeld manuell einzugreifen. Denn eine so definierte und verfügbare gemachte ADF stellt in Form einer Subroutine ein (hoffentlich) intelligentes Verhalten bereit. Sie fungiert als Makro, das sich ein Mensch erdacht hat.

Wo man die Parameterdatei ablegt, spielt keine Rolle. Aus organisatorischen Gründen bietet es sich jedoch an, diese im Pfad abzulegen, wo die übrigen Ressourcen zum GP-Problem gespeichert sind. Die Konfigurationsdatei übernimmt alle wesentlichen Parameter für einen GP-Algorithmus aus der Datei *src/ec/gp/koza/koza.params*. Dort sind beispielsweise Parameter wie allgemeine Logik für die Fitnessverwaltung, eine Initialisierungsklasse oder Crossing Over Parameter definiert.

Start und Ablauf der Berechnung

Nachdem man problemspezifische Implementierungen vorgenommen und eine dazu gehörige Konfigurationsdatei angelegt hat, ist es möglich, die Berechnung zu starten.

Es gibt eine ausgezeichnete Klasse in ECJ, die *ec.Evolve* heißt. Sie dient zum eigentlichen Start der GP-Berechnung. Sie wird nicht in einer Parameterdatei deklariert. Vielmehr wird sie über die Kommandozeile beim Start des Java-Programms aufgerufen. Ihr wird dabei die zum Problem gehörige Parameterdatei übergeben. Dies ist möglich und sinnvoll, da in der Parameterdatei alle relevanten Informationen zum Problem deklariert sind.

Nach dem Start der Anwendung übernimmt ECJ alles weitere. Hierin liegt der eigentliche Nutzen dieses Frameworks. Nicht nur, dass ECJ die zuvor verwendeten Basisklassen bereitstellt. Gleichzeitig kümmert sich das Framework darum, den Rahmen der GP-Anwendung zu definieren. Dazu gehören insbesondere: Einlesen von Parametern, Ablauflogiken bereitstellen, Fehlerbehandlung und Ausgabesteuerung. Während der Berechnung gibt ECJ auf der Java-Konsole Statusmeldungen aus. Das ist deswegen nützlich, weil eine Berechnung auch einmal länger dauern kann, trotz der heutigen Leistungsfähigkeit von Prozessoren.

Ergebnis der Berechnung

Die Standardausgabedatei namens *ou.stat* ist der Container für Statusausgaben. In ihr wird die pro Generation beste Lösung festgeschrieben. Nach Beendigung des Durchlaufs – entweder nach Finden einer ausreichend guten Lösung oder nach Abbruch wegen Überschreitung der maximal festgesetzten Berechnungsschritte – notiert ECJ dort das Ergebnis der insgesamt besten Generation.

Die Art und Formatierung der Ausgabe wird beeinflusst durch die in der Konfigurationsdatei angegebene Statistikklasse; Der zugehörige Parameter in dieser Datei heißt *stat*. Für das Rasenmäherproblem wurde eine eigene Statistikklasse

ec.app.lawnmower.LawnmowerStatistics implementiert. Der Grund ist die pseudographische Ausgabe des „Spielfeldes“ in Form einer einfachen Textgraphik. Das Spielfeld ist im Rasenmäher-Projekt der zu mähende Rasen. Es sei angemerkt, dass die ECJ beiliegende beispielhafte Implementierung des Rasenmäher-Problems die Statistikausgabe allerdings nicht in der Statistikklasse hält. Vielmehr ruft die Statistikklasse eine Methode – nämlich die Methode *describe(...)* – der Problemklasse auf. Dieses Vorgehen ist willkürlich und nicht Gesetz.

Die Ausgabe zum Rasenmäherproblem zeigt ein rechteckiges Feld, das durch ein Gitternetz in Plätze aufgeteilt ist. Jeder Platz enthält eine Nummer, die angibt, nach wie vielen Operationen der Platz vom Rasenmäher abgemäht wurde. Manchmal fehlen Nummern. Das liegt daran, dass auf einem Platz mehrere Operationen durchgeführt werden können (Drehung, Mähvorgang), pro Platz aber nur eine Nummer ausgegeben wird.

Fazit

Nach eingehender Betrachtung der beiden Verfahren GA und GP aus dem Gebiet der Künstlichen Intelligenz, ist es angebracht, ein gemeinsames Fazit zu ziehen. Im Vergleich schneidet die GP gegenüber den GAs 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 GAs und GPs 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

- ECJ: http://cs.gmu.edu/~eclab/projects/ecj/
- Homepage von John Koza: http://www.genetic-programming.com/
- Roboter spielen Fussball: http://www.teambots.org/
- Klaus Meffert: Genetische Algorithmen mit Java – Auf Darwins Spuren, in *Java Magazin* 08/2004

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).
```