kurz & bündig	
Inhalt	Genetische Algorithmen mit JGAP
Zusammenfassung	Wie die Mona Lisa aus Polygonen entsteht
Quellcode mit angeliefert	Ja

Genetische Algorithmen mit JGAP

Oder: Das Mona Lisa Malproblem

In Anlehnung an die Evolutionstheorie helfen Genetische Algorithmen beim Finden von Lösungen zu Problemen mit riesigem Suchraum. Der Artikel beschreibt die Umsetzung mit Hilfe des Open-Source Frameworks JGAP.

Von Klaus Meffert

Überblick

JGAP ist ein frei erhältliches, quelloffenes Framework für Java. JGAP steht für *Java Genetic Algorithms and Genetic Programming Package*. Während dieser Artikel die Arbeit mit JGAP zur Umsetzung von Genetischen Algorithmen (GA) beschreibt, ist die Genetische Programmierung (GP) eine komplexere Ausprägung Evolutionärer Algorithmen, die ebenfalls von JGAP unterstützt wird und nur kurz angerissen werden kann.

Die Verfahren GA und GP reihen sich zusammen mit klassischen Verfahren und Neuronalen Netzen in das Gebiet der Künstlichen Intelligenz ein. Evolutionäre Verfahren, die eine Untermenge der KI darstellen, beschäftigen sich mit dem Lösen von Problemen durch Maschinen nach dem Vorbild der biologischen Evolution. GAs lösen Probleme durch evolutionäres Hervorbringen von Chromosomen. Deren Erzeugung wird anhand einer Bewertungsregel, die Fitnessfunktion genannt wird, nach dem Darmwinschen Modell gesteuert. Die Ausprägung der Chromosomen wird dann je nach Problemstellung interpretiert. GAs operieren somit im Prinzip analog zu den vom Evolutionsforscher Charles Darwin entwickelten Grundsätzen. Die Genetik wird gemäß dem Vorbild aus der Natur im Sinne eines automatischen, von Zufall beeinflussten Regelkreises genutzt, um Probleme zu lösen. GAs basieren, wenn man es allgemein formulieren will, auf sich inhaltlich, aber nicht strukturell weiter entwickelnden Programmen. Die GP erlaubt darüber hinaus die strukturelle Weiterentwicklung. Dieser Vorteil wird durch den Nachteil der komplexeren Ausgestaltung und einer meist erhöhten Rechenzeit erkauft.

Es gibt zahlreiche Anwendungsgebiete für GAs. Dazu gehört das Finden von

- Funktionen/Formeln zu Wertetabellen,
- Routen zu gegebenen Wegmarken (Problem des Handlungsreisenden),
- Belegungen für gegebene Volumina (Rucksackproblem),
- geometrischen Anordnungen unter Berücksichtigung physikalischer Gesetze,
- Maximum- und Minimumsuchen, Parameteroptimierungen etc.

Ein Beispiel für einen praktischen Anwendungsfall ist das Ermitteln von Standorten für Mobilfunkantennen sowie die zu den Antennen einzustellende Konfiguration. Die zu findende Lösung sollte dabei so wenige Resourcen (Anzahl Antennen und Stromverbrauch der Konfiguration) wie möglich benötigen und gleichzeitig eine maximale Netzabdeckung gewährleisten. Für ein Gebiet wie die Bundesrepublik Deutschland kann man die nahezu unendliche Anzahl an Möglichkeiten erahnen, Antennen aufzustellen und zu konfigurieren. Durch bloßes Ausprobieren wird eine gute Lösung nur schwer zu finden ein. Hier bietet sich statt dessen ein evolutionäres Verfahren wie das der Genetischen Algorithmen an.

Grundlagen von Genetischen Algorithmen

GAs basieren im Wesentlichen auf den Elementen Population, Genotyp, Chromosom, Gen, Phyänotyp und Fitnessfunktion. Die Population ist die Menge aller Individuen, in der Natur würde man sagen: Lebewesen. Ein Individuum besitzt einen Genotyp, der die interne Repräsentation des Individuums darstellt. Der Genotyp wiederum besteht aus einer Menge von Chromosomen, mindestens also einem und maximal so vielen, wie für ein bestimmtes Problem benötigt werden. Ein einzelnes Chromosom enthält eine Menge an Genen. In einem Chromosom existiert mindestens ein Gen, maximal so viele wie benötigt bzw. vorgegeben. Ein Gen ist, wie in der Natur, quasi-atomarer Träger einer Erbinformation. Ein Chromosom fasst somit eine Sequenz von Erbinformationen an einem Ort zusammen. Der Genotyp ist der Container für alle in einem Individuum vorhandenen Erbinformationen. Um nun die Güte eines Individuums, also dessen Überlebensfähigkeit, zu beurteilen, wird eine so genannte Fitnessfunktion erstellt. In der Natur ist das Äquivalent zu dieser Funktion die Umgebung eines Lebewesens, die nicht simuliert werden muss, sondern an sich schon vorhanden ist. Im Computer existiert kein natürlicher Lebensraum, der qualitative Aussagen über die Güte einer Bitfolge erlauben würde. Daher muss eine Fitnessfunktion manuell vorgegeben werden, die aus der inneren Repräsentation, dem Genotyp, eine Darstellung, den Phänotyp, ableiten. Je nach Problemstellung fällt die Fitnessfunktion anders aus. Sie tut zwei Dinge: Erstens interpretiert sie einen gegebenen Genotyp, nämlich den des gerade untersuchten Individuums. Zweitens beurteilt sie das Ergebnis der Interpretation mit Hilfe eines Zahlenwertes. Der Zahlenwert kann entweder als Fehlerrate gedeutet werden, dann sind möglichst kleine Werte erstrebenswert. Andererseits kann der Fitnesswert auch als Erfolgsrate angesehen werden, welche einen großen Zahlenwert für eine hohe Güte bzw. Überlebensfähigkeit eines Individuums zurückliefert.

Damit nun eine Evolution stattfinden kann, wird nach der Bewertung der Individuen einer Generation eine Reihe von genetischen Operatoren durchgeführt. Diese Operatoren berücksichtigen meist die Fitness jedes Individuums, beziehen aber auch den Zufall mit ein. Sie korrespondieren mit den Vorbildern aus der Natur, nämlich Mutation, Crossing Over und Replikation. Nachdem die Operatoren die aktuelle Population verändert haben, werden Selektoren verwendet um die Spreu vom Weizen zu trennen. Das bedeutet normalerweise, mit einer hohen Wahrscheinlichkeit werden fitte Individuen in die nächste Generation gelassen und weniger fitte nicht. Das entspricht dem Überleben bzw. Sterben in der Natur. Die Evolution ist entweder nach einer vorgegebenen Anzahl von Generationen beendet oder sobald ein Individuum, also eine Lösung für das Problem, mit einer sehr guten Fitness vorliegt. Eine optimale Lösung für Probleme, die durch evolutionäre Verfahren wie GAs zu finden ist, gibt es meist nicht oder sie sind kaum aufzuspüren. Daher die Forderung nach einer „nur“ sehr guten Lösung.

JGAP – Ein Framework für Genetische Algorithmen

Die in einem GA verwendeten Elemente, nämlich Population, Genotyp, Chromosom, Gen und Fitnessfunktion sowie die genannten genetischen Operatoren und Selektoren, sind strukturell immer dieselben. Die Bereitstellung eines Frameworks zur Unterstützung des Entwicklers bietet sich also an. JGAP ist ein solches Framework, das die Implementierung von GAs deutlich vereinfacht. JGAP existiert als Open-Source Lösung bereits seit dem Jahr 2000 auf Sourceforge [1] und wird ständig weiter entwickelt. In Form einer dualen Lizenz wird auch für die Verwendung in kommerziellen Projekten eine kompatible Lizenz bereitgestellt.

Für ein zu lösendes Problem wird ein GA mit JGAP durch Umsetzung der folgenden Schritte realisiert. Fundamental ist die Definition einer Konfiguration durch die Basisklasse *org.jgap.Configuration*. Hier werden genetische Operatoren und Selektoren definiert. Eine Standardimplementierung liegt mit Klasse *org.jgap.impl.DefaultConfiguration* bereit. Dort sind zwei Operatoren für Mutation und Crossing Over, ein Selektor sowie Standardwerte für Mutations- und Crossing Over-Rate vorbelegt. Die problemspezifischen Parameter müssen für den Einzelfall vorgegeben werden. Dazu gehören insbesondere die Größe der Population (Anzahl der Individuen pro Generation) und die Fitnessfunktion sowie der Aufbau der

Chromosomen. Letzteres bestimmt, wie viele Gene welchen Typs pro Chromosom zur Repräsentation eines Individuums verwendet werden. Wir erinnern uns: Alle Chromosomen eines Individuums repräsentieren eine mögliche Lösung für ein Problem, welche durch die Fitnessfunktion interpretiert und bewertet wird. Der Aufbau der Chromosomen wird in JGAP elegant durch beispielhafte Vorgabe deklariert. Der folgende Abschnitt demonstriert diesen und alle anderen notwendigen Schritte.

Das Mona Lisa Malproblem

Zur Veranschaulichung, wie man mit JGAP ein Problem definieren und eine Lösung finden kann, sei folgende Aufgabe gegeben. Ein Ende 2008 populär gewordener Anwendungsfall für die Verwendung Evolutionärer Algorithmen ist die Beschreibung und Abbildung eines Bildes mit Hilfe möglichst weniger Polygone (als *Mona Lisa Painting Problem* bezeichnet, siehe [2]). Das bedeutet im Idealfall das Finden einer stark komprimierten Beschreibung des Bildes durch die Menge der ermittelten Polygone. Zur Veranschaulichung wurde vom Initiator des Beispiels als Eingabebild die Mona Lisa von Leonardo da Vinci gewählt. Die JGAP-Gemeinschaft hat daraufhin das Mona Lisa Problem mit JGAP umgesetzt, und zwar sowohl mit Hilfe Genetischer Algorithmen als auch mittels Genetischer Programmierung. An dieser Stelle soll nur der erste Ansatz beschrieben werden. Der Ablauf zum Finden von Polygonen, die ein gegebenes Bild möglichst effizient abdecken, lautet:

0) Konfiguriere den GA für das Problem.

1) Erstelle eine zufällige Population von Polygonen mit Hilfe von Chromosomen.

2) Erzeuge aus den Polygonbeschreibungen pro Chromosom ein Bild.

3) Vergleiche das erzeugte Bild mit dem Ausgangsbild (Mona Lisa) und berechne die Abweichungen auf Pixelebene.

4) Selektiere die Chromosomen, die die Bilder mit den wenigsten Abweichungen erzeugen.

5) Wende genetische Operationen, wie Mutation, auf diesen Chromosomen an.

6) Eliminiere die schlechtesten Chromosomen und übernehme die anderen in die nächste Generation.

7) Springe zu Schritt 2 oder beende, wenn eine akzeptable Polygonbeschreibung gefunden wurde.

Die Quelltexte zum Beispiel finden sich in der JGAP-Distribution im Verzeichnis *examples/src/examples/monalisa*. Das Package *core* enthält die wesentlichen Klassen. Das Beispiel kann mittels der Klasse *GeneticDrawingApp* im Package *gui* gestartet werden.

Konfiguration von JGAP

Bevor die Konfiguration von JGAP für das zu lösende Problem vorgenommen werden kann, muss das Problem genauer benannt werden. Wichtigstes Element in unserem Beispiel sind die Polyone. Ein Polygon wird beschrieben durch eine Farbe (Füllfarbe) sowie durch eine Menge von X/Y Koordinatenpaaren. Diese Festlegung wird JGAP durch Angabe eines Beispielchromosoms mitgeteilt, das aus mehreren Genen besteht (siehe Klasse *GAInitialChromosomeFactory*). Das Beispiel arbeitet mit dem HSB-Farbmodell (HSB = Hue, Saturation, Brightness = Farbton, Sättigung, Helligkeit) sowie einer Alpha-Komponente, die die Transparenz des Polygons bestimmt. Ein Beispielchromosom besteht also aus so vielen Genen, wie nötig sind, um sowohl die Informationen zur Farbe und zum Alphawert als auch zu den Punkten eines Polygons zu kodieren. Da nicht nur ein Polygon verwendet werden soll, muss im Gen Platz für so viele Polygone sein wie dies vom Entwickler für sinnvoll gehalten wird. Eine akzeptable Zahl sollte in etwa im Bereich von 80 Polygonen liegen, damit eine Chance besteht, das Ausgangsbild gut abzubilden. An dieser Stelle tritt ein Unterschied zwischen GA und GP zutage. Während für einen GA die Struktur bereits vorgedacht werden muss, ist dies in einem GP nicht der Fall. Bildlich äußert sich das beim Mona Lisa Problem so, dass bei der GA-Version bereits von Anfang an sehr viele Polygone gezeichnet werden, während bei der GP-Version ein fast stetiger Anstieg der Polygonzahl zu verzeichnen ist. Die Polygonanzahl der GA-Version

erscheint unter Umständen schwankend, weil einige Polygone unsichtbar sein können. Das ist insbesondere im Initialzustand der Fall oder wenn später alle Punkte eines bestimmten Polygons dieselben Koordinaten besitzen.

In der GA-Version wird also das Beispielchromosom wie folgt aufgebaut:

- Für den Farbwert eines Polygons vier Gene: Gene eins bis drei stellen je einen ganzzahligen Wert (Klasse *org.jgap.impl.IntegerGene*) dar für die Farbkomponenten der Füllfarbe im Bereich 0..255, Gen vier ist der Alphawert im Bereich 0..255
- Für jeden der fünf Punkte des Polygons je zwei Gene: Gen eins ist die X-Koordinate, Gen zwei die Y-Koordinate. Beide Gene sind ebenfalls vom Typ *IntegerGene*.

Das Framework kennt nun den Aufbau der Chromosomen. Es sorgt für eine zufällige Initialisierung dieser. Der zulässige Wertebereich pro Gen wird übrigens im Konstruktor des verwendeten Gentyps, also hier für das *IntegerGene*, übergeben. Weitere von JGAP bereit gestellte Gentypen sind: *BooleanGene* für Wahrheitswerte (true, false), *DoubleGene* für gebrochene Zahlen, *StringGene* für Zeichenfolgen sowie *CompositeGene* für Gene, die aus anderen Genen zusammen gesetzt sind. Insbesondere mit letzterem Gentypus ist es möglich, komplexere Strukturen für die Lösungssuche mit GAs zu definieren. Auf Basis der gegebenen Gentypen sowie der Basisklassen *BaseGene* und *NumberGene* ist es möglich, eigene Genvarianten zu programmieren.

Die Fitnessfunktion interpretiert nun für jedes durch JGAP nach Regeln der Evolution generierte Individuum dessen Chromosom durch Auslesen der Werte der im Chromosom liegenden Gene. Korrespondierend dazu wird ein Bild aus diesen Polygonen gemalt und mit dem Mona Lisa-Bild verglichen (Schritte zwei und drei der obigen Liste). Da die Lage der Gene im Chromosom klar definiert ist, weiß die Fitnessfunktion auch, an welcher Stelle im Chromosom der Farbwert, wo der Alphawert und wo die Punktkoordinaten kodiert sind. Die Fitnessfunktion muss also nur diese Werte auslesen und entsprechend ihrer Bedeutung verwenden um Polygone zu malen. Danach wird jeder Punkt des Ursprungsbildes (Mona Lisa) mit dem durch den GA erzeugten Bild verglichen. Für den Vergleich sind mehrere Strategien möglich. Gewählt wurde die als Least-Mean-Square bezeichnete Methode zur Ermittlung der Unterschiede zwischen Ursprungs- und generiertem Bild. Dabei wird pro Pixel für jede Farbkomponente der Unterschied bestimmt und quadriert. Die quadrierten Differenzen aller drei Farbkomponenten (rot, grün, blau) werden addiert, um danach die Wurzel daraus zu ziehen. Das Quadrieren ist notwendig, um negative Werte zu vermeiden und Vorzeichen zu egalisieren. Da hier eine Differenz zwischen Soll und Ist gebildet wird, sind kleinere Rückgabewerte der Fitnessfunktion als besser anzusehen als höhere (optimal ist eine Differenz von null). Damit JGAP dies weiß, wird in der Konfiguration zum GA (Klasse *examples.monalisa.core.GAConfiguration*) der *DeltaFitnessEvaluator* verwendet. Das Gegenstück zu diesem Evaluator ist der *DefaultFitnessEvaluator*, der höhere Werte als besser ansieht. In Klasse *GAConfiguration* werden zudem weitere Basisparameter eingestellt, die von Problem zu Problem variieren können und spezifisch festzulegen sind. Dazu gehört insbesondere die Größe der Population, also die Anzahl der gleichzeitig in einer Generation vorhandenen Chromosomen, die mit einander ums Überleben konkurrieren. Im Beispiel wurde ein Wert von fünf gewählt, der ursprüngliche Vorschlag (siehe [2]) verwendet eine Populationsgröße von zwei.

Alle o.g. Schritte außer Nummer zwei und drei müssen nicht implementiert werden. Sie werden durch das JGAP-Framework bereits abgebildet und können über die Konfiguration (siehe Klasse *GAConfiguration*) beeinflusst werden. Zu Schritt fünf empfiehlt es sich allerdings, problemspezifische genetische Operatoren bereit zu stellen. Im Beispiel sind diese in den Klassen *SingleMutationOperator*, *PointMutationOperator*, *PolygonMutationOperator* und *ColorMutationOperator* gegeben (siehe Javadoc der jeweiligen Klasse für eine Beschreibung der Funktion).

Das JGAP-Beispiel enthält eine grafische Ausgabe (siehe Screenshot). Sie stellt den Fortschritt der Evolution als Kurve dar, die die Fehlerrate der bis dato besten Polygonlösung veranschaulicht. Für die Grafik wird die frei erhältliche Bibliothek JFreeChart verwendet, die in

JGAP integriert ist. Weiterhin wird das zur besten Lösung gehörende Polygonbild angezeigt, so dass man einen guten Eindruck erhält, wie die Evolution abläuft. Alle grafische Ausgaben betreffenden Klassen liegen im Package *gui*. Zu Versuchszwecken sind im JGAP-Beispiel weitere Bilder enthalten, u.a. ein bekanntes abstraktes Gemälde von Piet Mondrian, das eine Komposition aus Vierecken in verschiedenen Farben darstellt.

Start und Ablauf der Berechnung

Wenn JGAP konfiguriert ist, kann auf Grundlage dessen ein Genotyp aufgesetzt werden. Dies ist entweder durch Erzeugung einer initialen Population möglich, die dem Konstruktor der Klasse *org.jgap.impl.Genotype* mitgegeben wird. Alternativ und meist einfacher ist der Aufruf der statischen Methode *Genotype.randomInitialGenotype*. Letztere erzeugt die Population selbständig. Das ist immer dann möglich, wenn die Gene in den Chromosomen nicht von einander abhängig sind. Ein Beispiel für abhängige Gene ist das Problem des Handlungsreisenden (Travelling Salesperson Problem, kurz: TSP). Hier dürfen im klassischen Fall zwei Gene, die Städte repräsentieren, nicht denselben Wert annehmen. Denn auf der Rundreise des Handlungsreisenden soll jede Stadt und zwar genau einmal besucht werden.

Mittels der o.g. statischen Methode wird eine Genotyp-Instanz samt Population erzeugt. Aus der Genotyp-Instanz kann nun die Methode *evolve* aufgerufen werden, um die Evolution und somit die Lösungsfindung zu starten, indem eine einzelne Generation berechnet wird. Eine überladene Version von *evolve* erlaubt die Eingabe eines ganzzahligen Wertes größer null, der die Anzahl hintereinander ausführbarer Evolutionsschritte angibt. Der Aufruf von *evolve* steht typischerweise in einer Schleife. Alternativ kann die überladene Version der Methode mit einer hinter einander auszuführenden Anzahl an Evolutionsschritten als Eingabeparameter aufgerufen werden. Ein Vorteil des Einzelaufrufs ist etwa, dass individuelle Statistikdaten zur aktuell entwickelten Population ausgegeben werden können.

Darüber hinaus können Ereignisbehandler registriert werden, die bei jeder neuen Generation gefeuert werden. Dazu muss zuerst über Klasse *org.jgap.Configuration* der von JGAP bereits vorgegebene Ereignisbehandler über *getEventManager()* geholt werden. Nun kann diesem über *addEventListener* ein der Schnittstelle *org.jgap.event.GeneticEventListener* gehorchender Behandler übergeben werden.

Die Mona Lisa ist nach mehreren Tausend Evolutionsschritten in zufrieden stellendem, wenn auch nicht vollständig überdeckendem Maße mit Hilfe der Polygone abgebildet (abhängig von der konfigurierten Anzahl der Polygone und der Anzahl pro Punkte eines Polygons). Eine Verbesserung dieser hohen Anzahl an Iterationen kann durch die Verwendung flexiblerer Polygone und spezifischerer Operatoren erzielt werden. Im Beispiel werden der Einfachheit halber nur Polygone mit fünf Eckpunkten verwendet und keine Operatoren, die Punkte hinzufügen oder entfernen können. Erschwerend hinzu kommt die Transparenz der Polygone, gesteuert durch den Alpha-Anteil zur Füllfarbe. Dieser verkompliziert das Hinzufügen neuer Polygone, da diese darunter liegende Polygone (oft mehrere gleichzeitig) überlagern und sich vermischen und nicht volldeckend sind.

Der Reiz dieses Anwendungsfalls liegt sicherlich darin, dass die statt findende Evolution plastisch dargestellt wird und zudem populär an einem allbekannten Bild wie der Mona Lisa. Die Komplexität des Problems, Polygone zur Beschreibung eines beliebigen Bildes finden, erlaubt es, Ineffizienzen im verwendeten GA bzw. der zugrunde liegenden Konfiguration aufzudecken und so eine Steigerung der Effizienz zu ermöglichen. Die Popularität des Beispiels wird auch daran deutlich, dass es mittlerweile eine ganze Reihe von Umsetzungen gibt, sowohl in Java mit und ohne Framework-Verwendung als auch in Javascript oder Adobe Flash (siehe [2]). Einige Beispiele verwenden die GA, andere die GP als Lösungskonzept.

Fazit

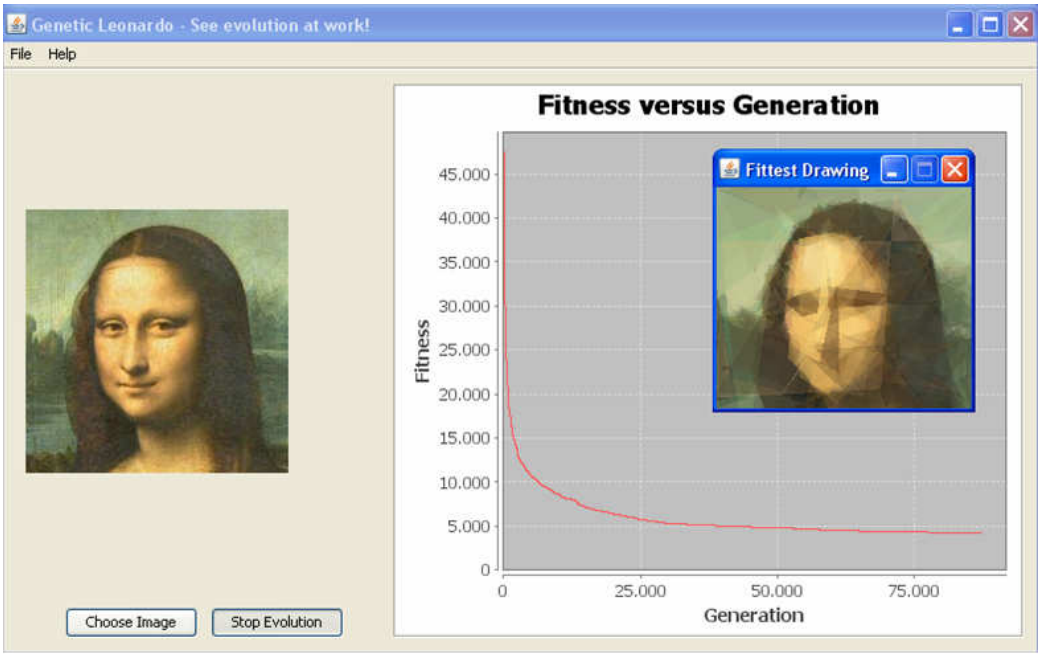
Das Problem, die Mona Lisa mit Polygonen nachzuzeichnen, veranschaulicht die Arbeitsweise von GAs. Was zunächst spielerisch wirkt, kann in einer ernsthaften Anwendung

mündet, denkt man an die Bildkompression. Es wird deutlich, dass die Wahl der Parameter für den GA entscheidend ist. Dazu gehören die Anzahl der Polygone, die Anzahl der Punkte im Polygon, die Größe der Population sowie Mutations- und Crossing Over-Rate. Diese sind bis auf die Polygonanzahl nicht als kritisch zu beurteilen. Denn die Anzahl der Polygonpunkte kann durch entsprechende Operatoren dynamisiert werden. Die Raten der genetischen Operatoren sollten in einem Normbereich liegen, der recht gering ist (etwa 5 bis 15 Prozent). Die Größe der Population wird kompensiert durch die Anzahl der erforderlichen Generationen: Je kleiner die Population, desto größer die notwendigen Durchläufe. Die Notwendigkeit, einige Strukturen und Parameter vorzudenken, kann zudem durch die GP eliminiert werden (siehe das GP-Beispiel von JGAP).

Regelmäßig werden neue Anwendungsgebiete für GAs und GPs entdeckt und somit in der Praxis sonst nicht erreichbare Lösungen produziert. Wer einen spielerischen Einstieg sucht, kann neben dem genannten Beispiel das Projekt RobocodeJGAP (siehe [1]) analysieren. Dort kämpfen auf einer virtuellen Plattform Roboter, die durch Java-Code repräsentiert werden, gegeneinander. JGAP wird hier verwendet, um vollwertigen Java-Code zu schreiben.

Die genannten Anwendungsfälle haben dazu beigetragen, dass JGAP optimiert und funktionell erweitert werden konnte. So bleibt zu hoffen, dass die Gemeinschaft auch in Zukunft rege zu Open-Source Projekten wie diesem beiträgt. Anregungen sind immer willkommen.

Klaus Meffert ist Diplom-Informatiker und als freiberuflicher Berater und Entwickler im SAP- und Java-Bereich tätig. Seine berufsgeleitende Doktorarbeit zum Thema Software-Entwicklung hat er abgeschlossen. Weiterhin engagiert er sich im Open-Source Bereich sowie als Fach- und Buchautor. www.klaus-meffert.de

Links & Literatur

- [1] JGAP: http://jgap.sf.net/
- [2] Mona Lisa Malproblem: http://www.klaus-meffert.de/doc/monalisa.html