Optimieren durch Selektion

Die Evolution im Algorithmus – Teil 1: Grundlagen

Stephan Pirnbaum

Bei der Lösung von Problemen lässt sich die Informatik häufig von anderen Wissenschaften inspirieren. Man denke nur an Neuronale Netze und die Möglichkeiten, die sich mit diesen eröffnen. Doch während diese aufwendig angelernt werden müssen, bietet Darwin mit seiner Evolutionstheorie eine interessante und leicht umsetzbare Lösung für eine Vielzahl von Optimierungsproblemen. Wie man sich das Ganze vorstellen kann, soll diese Kolumne näher beleuchten.

Jeder ist sicher schon einmal auf Optimierungsprobleme gestoßen und hat überlegt, wie man diese algorithmisch lösen könnte. Ein Beispiel hierfür ist Ressourcenplanung. Dabei sollen Tasks in zeitlicher Begrenzung eine Menge von Ressourcen zugewiesen werden, um diese unter Beachtung verschiedener Gesichtspunkte, beispielsweise den entstehenden Kosten oder der benötigten Zeit, abzuarbeiten. Dies ist ganz klar ein Optimierungsproblem. Doch sind ausgefeilte, hochgradig spezialisierte Algorithmen oder Brute-Force die einzige Möglichkeit zur Lösung solch komplexer Optimierungsprobleme?

Die Grundlagen der Evolutionstheorie

Bereits 1859 veröffentlichte Charles Darwin seine Evolutionstheorie [Wiki-a] und legte damit den Grundstein für die modernen Biowissenschaften. Auch die Informatik hat sich das Prinzip der Evolution auf verschiedenste Art und Weise zunutze gemacht.

Was besagt nun aber der eben genannte Darwinismus? In der Schöpfungstheorie existiert ein Gott, der die Welt mitsamt allen Tieren und Pflanzen erschaffen hat. Alle Lebewesen seien von Beginn an perfekt angepasst gewesen, sodass keine Art ausstarb oder neu entstand. Darwin widersprach dieser Theorie und stellte dafür drei Thesen auf:

- Alles Leben auf der Erde hat sich im Laufe der Zeit entwickelt.
- Die Weiterentwicklung von Arten erfolgt durch natürliche Selektion.
- Arten passen sich an die Umwelt an.

Hatte Darwin recht? Die Antwort kann in diesem Rahmen nicht gefunden werden, in der Tat entwickelten sich einige Arten weiter, während andere ausstarben. Die Evolution kann also als Optimierungsproblem angesehen werden, wobei das Ziel ist, bestmöglich an die Umgebung angepasst zu sein. Mit anderen Worten, der Erhalt einer Art kann nur gesichert werden, wenn sich die Individuen weiterentwickeln.

Evolution im Algorithmus

Soll ein genetischer Algorithmus zur Lösung eines Optimierungsproblems eingesetzt werden, stellen sich zunächst zwei Fragen:

- Wie können Probleme beziehungsweise deren Lösungen codiert werden?
- Welche Schritte gilt es zu implementieren?

Lassen wir uns dafür von der Natur inspirieren. Eine *Population* ist eine Menge von *Individuen* einer *Generation*. Ein Individuum bezeichnet in diesem Kontext eine Lösungsmöglichkeit für ein Problem. Das für uns interessante Erbmaterial wird als *Genotyp* bezeichnet und ist eine Menge von *Chromosomen*, welche wiederum eine Folge von *Genen* sind, deren Werte als *Allele* bezeichnet werden. Eine Array-Repräsentation der Lösungen ist daher naheliegend.

Stephan Pirnbaum ist Junior-Berater bei der buschmais GbR. Seine fachlichen Schwerpunkte liegen in der Konzeption und Entwicklung von Java-Applikationen im Unternehmensumfeld sowie der Analyse von Legacy-Anwendungen mit jQAssistant. E-Mail: stephan.pirnbaum@buschmais.com

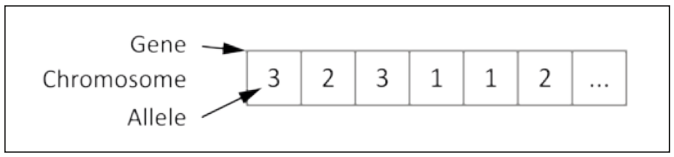

Abb. 1: Problemcodierung

Zwar sind andere Codierungen auch möglich, allerdings nicht so effizient. Abbildung 1 fasst die beschriebene Struktur noch einmal zusammen.

Die Evolution setzt sich aus drei Schritten zusammen. Im ersten Schritt wird bestimmt, welche Individuen einer Generation geeignet sind, um sich fortzupflanzen. In der Natur erfolgt diese Bewertung anhand des Anpassungsgrades eines Individuums an seine Umwelt. Im Algorithmus kommt es hier auf eine sinnvolle Skalarisierung an, was als *Fitnessfunktion* bezeichnet wird. Beispiele dafür werden in den kommenden Abschnitten gegeben.

Zur eigentlichen Individuenauswahl [Wiki-b] gibt es verschiedene Strategien, beispielsweise *Fitness Proportional Selection*, bei welcher zwar alle Individuen selektiert werden können, die Wahrscheinlichkeit der Selektion aber von dessen Fitness abhängt, oder *Tournament Selection*, bei welcher mehrere Wettkämpfe unter Teilmengen der Population durchgeführt und deren Sieger selektiert werden. Zwar wäre es am einfachsten, die Individuen mit der höchsten Fitness auszuwählen, jedoch läuft man hier Gefahr, die Vielfalt zu stark einzuschränken.

```
pop = initial population
while (true) {
  survivors = selectFittest(pop)
  pop = crossover(survivors)
  pop = mutate(pop)
}
```

Listing 1: Basisalgorithmus

Da nun eine Menge geeigneter Individuen bestimmt wurde, kommt es im zweiten Schritt zur eigentlichen Fortpflanzung (Crossover). Dabei wird das Erbmaterial jeweils zweier Individuen kombiniert. Abbildung 2 zeigt das *Single-Point Crossover*, bei dem das Erbmaterial an jeweils einer Stelle geteilt wird. Alternativ dazu steht *Multi-Point Crossover*. Hier teilt man das Erbmaterial an mehreren Stellen.

Um für Varianz zu sorgen und ein Kollabieren des explorierten Lösungsraums zu vermeiden, kommt es im letzten Schritt zur Muta-

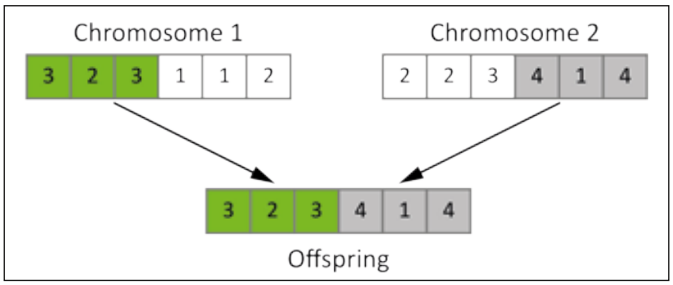

Abb. 2: Single-Point Crossover

tion. Dabei werden Gene zufällig ausgewählt, welche dann ihren Wert zu einem anderen wechseln.

Listing 1 stellt den soeben beschriebenen Algorithmus in Pseudocode dar. Eine detaillierte Einführung in genetische Algorithmen kann unter [Nat] gefunden werden.

Affen schreiben Shakespeare

Als Einstiegsbeispiel soll das Infinite-Monkey-Theorem [Tec] umgesetzt werden. Es besagt, dass es einem Affen in unendlicher Zeit möglich ist, alle Texte von Shakespeare an einer Schreibmaschine zu replizieren.

Man könnte zur Überprüfung dieses Theorems einen Brute-Force-Algorithmus implementieren, welcher so lange Zeichenkombinationen produziert, bis ein Werk von Shakespeare entstanden ist. Nimmt man sich das berühmte Zitat: „to be or not to be" und begrenzt die Eingaben auf Kleinbuchstaben sowie Leerzeichen (die Allele), gibt es 27^{18} mögliche Zeichenketten. Bei einem sehr schnell arbeitenden Affen, welcher eine Variante pro Sekunde schafft, würde es im ungünstigsten Fall $1{,}8*10^{18}$ Jahre dauern, bis nur dieser kurze Abschnitt entstanden ist. Zum Vergleich: Das Alter des Universums wird auf $13{,}8*10^{9}$ Jahre geschätzt. Wie kann nun aber ein genetischer Algorithmus helfen?

Wenn wir die zwei Individuen: „tx bu or noo to by" und "ta fcorv nkanto be" betrachten, sehen wir, dass die erste Variante wesentlich näher an der gesuchten Lösung ist als die zweite. Das Wissen kann sich die Evolution zunutze machen, in dem durch Crossover der besten Individuen zunächst eine Nachfolgergeneration gebildet und anschließend eine geringe Anzahl an Mutationsoperationen durchgeführt wird. Die Codierung für dieses Problem ist trivial, da sich eine Zeichenkette einfach auf ein Chromosom abbilden lässt.

Ziel ist es, den Wortlaut „to be or not to be" zu erhalten. Die Fitness einer Lösung kann als Abstand zum Zieltext ausgedrückt werden. Im einfachsten Fall ist das die Anzahl abweichender Stellen. Als Selektionsmethode wird die *Fitness Proportional Selection* gewählt, da so schlechtere Lösungen nicht von vornherein ausgeschlossen werden und somit eine größere Varianz herrscht. Single-Point Crossover mit dem mittleren Index als Crossover-Point wird zur Fortpflanzung verwendet.

Unter [Git] lässt sich der Quellcode zu diesem Beispiel in drei verschiedenen Varianten finden. Als Brute-Force-Variante, als selbstimplementierter genetischer Algorithmus sowie auf Basis der Bibliothek Jenetics [Jen-a]. Letztere wird im nächsten Beispiel eingeführt und dient hier nur der Vollständigkeit.

Im Folgenden soll zunächst der Einfluss der verschiedenen Parameter auf die Ausführungsgeschwindigkeit erläutert werden. Prinzipiell gilt es, zwei Parameter zu optimieren. Diese sind die Größe der Population und die Wahrscheinlichkeit der Mutation. Für die in den Tabellen 1 und 2 gezeigten Auswertungen wurde der selbstimplementierte Algorithmus verwendet.

Wie in Tabelle 1 zu erkennen ist, sinkt mit steigender Populationsgröße die Anzahl benötigter Generationen stark. Bei kleinen Populationen ist die Qualität der initialen Population von größter Wichtigkeit, da hier pro Generation nur eine sehr begrenzte Anzahl verschiedener Individuen überprüft werden kann. Jedoch hilft die unbegrenzte Steigerung der Populationsgröße nicht, da dies dem Grundgedanken eines genetischen Algorithmus entgegengewirkt wird die Ausführungszeit wieder ansteigt. Als zweiter Parameter kann die Mutationsrate angepasst werden, welche für den letzten Lauf auf 1 Prozent gesetzt wurde. Tabelle 2 präsentiert die Ergebnisse für eine Populationsgröße von 500.

Bei einer Mutationsrate von 0 Prozent muss aus der initialen Population die Lösung unmittelbar entstehen können, ansonsten kann nie eine Lösung gefunden werden. Ist die Mutationsrate hingegen zu hoch, ändern sich zu viele Gene der Lösungen und der genetische Algorithmus nähert sich dem Brute-Force-Ansatz. Im Allgemeinen gilt, je länger die Chromosomen sind, desto höher kann auch die Mutationsrate sein. In Experimenten wurde eine Mutationsrate von $0{,}004\log_2 l$ als optimal für derartige Codierungen befunden [Dov99].

Auch die Fitnessfunktion ist optimierbar. Hierbei ist es wichtig zu beachten, dass sich eine leichte Verbesserung bereits stark auf den Fitness Value auswirkt. Während aktuell die Anzahl korrekter Buchstaben durch die Anzahl aller geteilt wird, erwirkt eine Quadrierung von Dividend und Divisor eine stärkere Auswirkung einer Verbesserung. Dies führt für eine Populationsgröße von 500 und einer Mutationsrate von 1 Prozent zu 202 benötigten Generation in 305 ms.

Anwendungsszenarien genetischer Algorithmen

Das vorhergehende Problem mag als Einstiegsbeispiel genügen, hat allerdings mit der Praxis wenig zu tun. Denn wer die Lösung bereits kennt, muss keinen Algorithmus implementieren, um das Gleiche erneut zu erhalten. In Fällen hingegen, in denen zwar das Ergebnis noch nicht bekannt ist, sich allerdings die Qualität der Lösung bewerten lässt, können diese helfen, schnell zu einer guten Lösung zu kommen.

Im Folgenden soll an einem komplexeren Beispiel gezeigt werden, was passiert, wenn konkurrierende Qualitätskriterien zur Bewertung einer Lösung notwendig werden und auf welche Art man Probleme codieren kann. Das erfolgt unter Zuhilfenahme der Bibliothek Jenetics [Jen-a], welche sich in der Praxis als variabel und robust, aber dennoch als einfach zu verwenden erwiesen hat.

Es folgt ein kurzer Ausflug in die Welt des Scheduling. Dabei soll jeder Task jeweils eine Ressource zeitlich begrenzt zur Verfügung gestellt werden. Ziel ist es, in kürzest möglicher Zeit alle Tasks abzuarbeiten. Dabei gibt es die Besonderheit unterschiedlich leistungsstarker Ressourcen, deren Betriebskosten überlinear mit der Anzahl gefertigter Gegenstände steigen (siehe Abb. 3). Das Ziel

Populationsgröße	Benötigte Generationen	Terminierung
1	Unendlich?	Nie?
100	5164	1409 ms
500	388	475 ms
1000	250	519 ms
10000	119	1535 ms

Tabelle 1: Abhängigkeit der explorierten Generationen von der Populationsgröße

Mutationsrate	Generationen	Terminierung
0 %	1 oder unendlich	<1 ms oder nie
0,1 %	1365	884 ms
1 %	388	475 ms
2 %	859	679 ms
10 %	Nie?	Nie?

Tabelle 2: Abhängigkeit der explorierten Generationen von der Mutationsrate

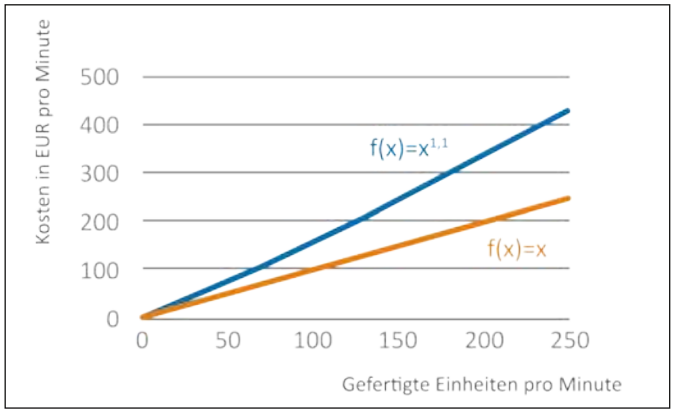

Abb. 3: Verhältnis von Fertigungszeit zu -kosten

ist wie bereits angedeutet die Minimierung der Ausführungszeit sowie der Kosten. Diese zwei sogenannten *Objectives* stehen aber in Konkurrenz zueinander. Eine Verringerung der Zeit führt zu einer Erhöhung der Kosten bei optimaler Ressourcenbelegung (daher das überlineare Wachstum).

Implementierung der Ressourcenplanung

Jenetics ist eine Bibliothek zur Implementierung genetischer Algorithmen. Dabei macht sich Jenetics das Java-Stream-API zunutze, indem die gesamte Evolution durch einen *EvolutionStream* umgesetzt wird. Die Bibliothek bietet zahlreiche Varianten von Genen, Chromosomen, Genotypen usw. in Abhängigkeit vom Datentyp (bspw. char, int und double) sowie diverse Selektions- und Mutationsoperatoren. Sollte dies nicht genügen, ist Jenetics durch klar definierte Interfaces leicht erweiterbar. Zur tiefer gehenden Einarbeitung sei an dieser Stelle die exzellente Dokumentation [Jen-b] empfohlen.

Bei der Implementierung des Beispiels ist zunächst über die Codierung nachzudenken. Es kann hier eine Zerlegung in Stammdaten (Ressourcen und Tasks) sowie Bewegt-Daten (Zuweisung von Ressourcen) erfolgen. Die letztere Kategorie ist während der Evolution zu manipulieren. Es wäre möglich, Allele als Objektstrukturen zu repräsentieren, welche Taskobjekte in den Ressourcenobjekten referenzieren, doch schadet dies sowohl der Handhabbarkeit als auch der Geschwindigkeit. Besser ist es, weiterhin auf einem Array primitiver Daten zu arbeiten. Abbildung 4 stellt dies dar.

Die Ressourcen sowie Tasks werden jeweils in einem eigenen Array gepflegt. Für beide existiert eine eigene Klasse, welche die Kosten und gefertigten Items pro Minute beziehungsweise den Workload (also die Anzahl beauftragter Items) abspeichern. Das Chromosom besitzt nun so viele Gene, wie Tasks existieren. Die Zuordnung von Gen zu Task wird dann über Indexgleichheit hergestellt. Da die Zuordnung von Ressourcen optimiert werden soll, werden diese die Allele darstellen. Im Beispiel bedeutet das, dass der Task an Index 1 (T2) der Ressource an Index 2 (R3) zugeordnet wird. Da Indexzugriffe günstig sind, ist diese Codierung einer Codierung komplexer Objekte als Allele vorzuziehen.

Jenetics muss wissen, welche Struktur das Erbmaterial hat. Dies geschieht wie in Listing 2 gezeigt. Ein Chromosom hat die gleiche

```
Resource[] res = // init resources
Task[] tasks =   // init tasks
Genotype<IntegerGene> gt = Genotype.of(
  IntegerChromosome.of(0,
    res.length - 1, tasks.length));
```

Listing 2: Strukturdefinition

```
Engine<IntegerGene, Double> engine = Engine
  .builder(this::fitness, gt)
  .populationSize(500)
  .alterers(
    new SinglePointCrossover<>(1),
    new Mutator<>(0.01))
  .selector(
    new RouletteWheelSelector<>())
  .build();
```

Listing 3: Konfiguration der GA Engine

```
EvolutionResult<IntegerGene, Double> result = engine
  .stream()
  .limit(
    Limits.bySteadyFitness(500))
  .collect(EvolutionResult
    .toBestEvolutionResult());
```

Listing 4: Ausführen des genetischen Algorithmus

Anzahl Gene wie Tasks vorhanden sind (3. Parameter), wobei die Anzahl der Allele der Anzahl Ressourcen gleicht (Parameter 1 und 2).

Als Nächstes muss die Genetic Algorithm (GA) Engine konfiguriert werden. Zur Konfiguration zählen die Fitnessfunktion, die Populationsgröße und die Art der Crossover- und Mutationsoperationen mitsamt deren Wahrscheinlichkeiten. Listing 3 zeigt eine passende Konfiguration für dieses Beispiel.

Eine Unbekannte ist momentan noch die Fitnessfunktion (`this::fitness`), welche später betrachtet wird. Zunächst einmal ist es wichtig, die Evolution zu starten. Dafür erlaubt die Engine die Erstellung eines EvolutionStream wie in Listing 4 gezeigt. Zu bemerken sind hierbei die Definition des Abbruchkriteriums sowie die Reduktion auf den besten gefundenen *Phänotypen* (das ist das Erscheinungsbild des Individuums).

In dieser Konfiguration wird die Evolution solange ausgeführt, bis innerhalb der letzten 500 Generationen kein besseres Individuum gefunden wurde. Es existieren allerdings noch weitere Abbruchkriterien, beispielsweise für das Erreichen einer bestimmten Fitness oder einer bestimmten Generationenzahl.

Umsetzung komplexer Zielfunktionen

Die Ressourcenplanung hat gezeigt, dass sich die Bewertung der Lösungsqualität nicht immer nur auf eine Zielfunktion beschränkt. Bei solch einer Situation spricht man von *Single-Objective Optimization*. Hier haben wir es allerdings mit zwei in Konkurrenz stehenden Objectives zu tun (Multi-Objective Optimization). Man hat nun zwei Möglichkeiten. Einerseits die folgende Kombination der Objectives:

- fitness (x) = -c * x.costs - t * x.time

Die negativen Vorzeichen kommen daher, dass die Fitness maximiert werden soll. Die Anpassung an besondere Situationen (nahende Deadlines) ist durch entsprechende Parametrisierung möglich. Tabelle 3 zeigt Ergebnisse des Beispielcodes für unterschiedliche Parametrisierungen. Dabei wurden 20 Ressourcen mit einer Gesamtleistung von 800 Items pro Minute und 100 Tasks mit einem Gesamtworkload von 97.000 Items angelegt. Eine Lösung wird nach etwa 3000 Generation und 15 Sekunden gefunden.

	Kosten	Zeit
c = 0; t = 1	163.000	548 min
c = 0,3; t = 0,7	132.660	3318 min
c = 0,7; t = 0,3	122.200	9085 min
c = 1; t = 0	122.150	9470 min

Tabelle 3: Entwicklung der Kosten und Zeit

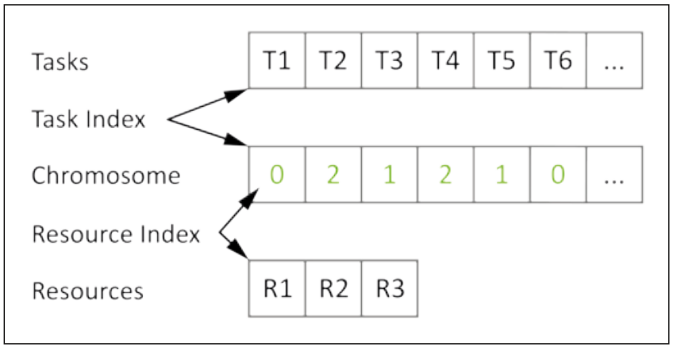

Abb. 4: Codierung komplexer Probleme

Der nichtlineare Zusammenhang von Zeit und Kosten ist hier eindeutig erkennbar, ebenso die leichte Anpassbarkeit des genetischen Algorithmus. Allerdings gibt es hier und insbesondere bei Problemen mit mehr als zwei Objectives eine große Hürde: Es erfolgt eine Bewertung über den Einfluss der Objectives. So kommt es, dass eine Lösung mit x.cost = 100 und x.time = 200 gleichgestellt wird mit einer Lösung mit x.costs = 200 und x.time = 100. Dies ist nicht immer gewollt und teilweise sogar falsch. Was also tun?

Multi-Objective Optimierung und Ausblick

Das Stichwort lautet Pareto-Optimierung [Wik-c], bei welcher die Objectives als gleich wichtig angesehen werden und die somit das eben gezeigte Problem zumindest während der Evolution vermeidet.

Wie man dies umsetzt, soll im zweiten Teil des Artikels anhand des Ressourcenplanungsbeispiels erklärt werden. Anschließend soll gezeigt werden, wie genetische Algorithmen die Struktur in großen, verwachsenen Softwaresystemen erkennen können und so eine Refaktorisierung und Redokumentation unterstützen können. Es bleibt also spannend.

Literatur und Links

[Dov99] D. Doval, S. Mancoridis, B. S. Mitchell, Automatic Clustering of Software Systems using a Genetic Algorithm, in: Softw. Technol. Eng. Pract., 1999

[Git] GitHub, Beispielcode, https://github.com/buschmais/GeneticAlgorithms

[Jen-a] Jenetics, Startseite, http://jenetics.io/

[Jen-b] Jenetics, Dokumentation, http://jenetics.io/manual/manual-4.0.0.pdf

[Nat] D. Shiffman, The Nature of Code, Kapitel 9: Genetische Algorithmen, http://natureofcode.com/book/chapter-9-the-evolution-of-code/

[Tec] TechTarget, Infinite Monkey Theorem, http://whatis.techtarget.com/definition/Infinite-Monkey-Theorem

[Wiki-a] Wikipedia, Evolutionstheorie, https://de.wikipedia.org/wiki/Evolutionstheorie

[Wiki-b] Wikipedia, Selektion, https://en.wikipedia.org/wiki/Selection_(genetic_algorithm)

[Wiki-c] Wikipedia, Pareto-Optimierung, https://de.wikipedia.org/wiki/Pareto-Optimierung