Philipps Universität Marburg

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik

Prof. Gabriele Taentzer
Stefan John

24. August 2020

Programmier-Praktikum

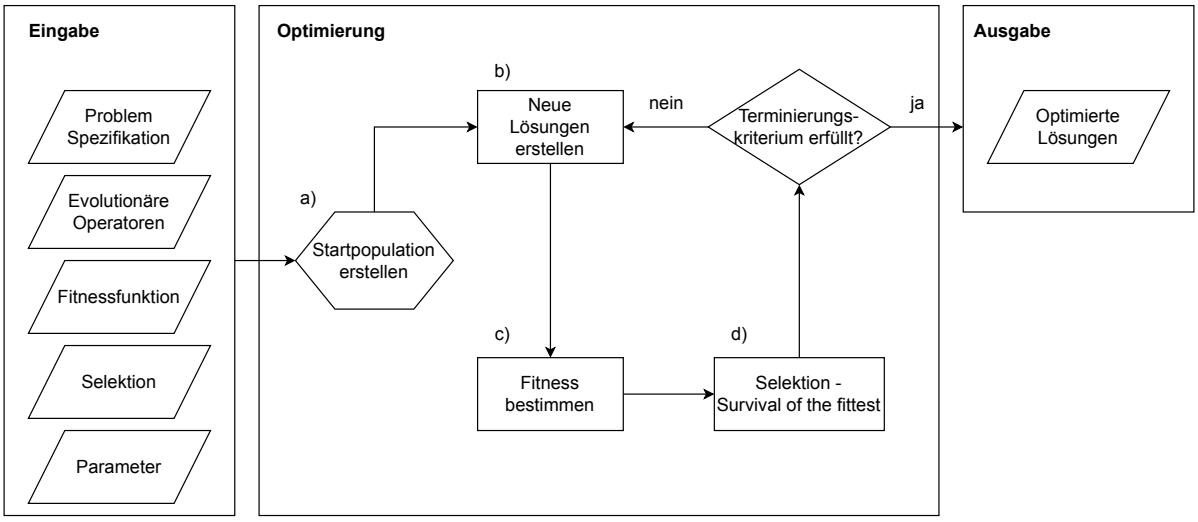

Abbildung 1: Schematische Darstellung des Ablaufs genetischer Algorithmen.

Vorstellung genetischer Algorithmen

In den nächsten zwei Tagen sollen Sie sich mit genetischen Algorithmen[1] beschäftigen. Genetische Algorithmen sind such-basierte heuristische Optimierungsverfahren. Sie werden in der Regel zur näherungsweisen Lösung von Problemen verwendet, für die eine exakte Lösung nur schwer oder unter hohem Zeitaufwand berechnet werden kann. Die Suche nach optimierten Lösungen ist dabei an die Konzepte natürlicher Evolution angelehnt.

Abbildung 1 stellt schematisch den Ablauf der durch einen genetischen Algorithmus durchgeführten Optimierung dar.

a) Genetische Algorithmen arbeiten üblicherweise auf einer *festgelegten Anzahl* möglicher Lösungen, der sogenannten Population. Zu einem gegebenen Problem wird daher zunächst eine Startpopulation (unoptimierter) Lösungen erstellt. Ausgehend von dieser wird iterativ nach besseren Lösungen gesucht.

b) Dazu werden, unter Verwendung evolutionärer Operatoren, aus den bestehenden Lösungen neue Lösungen abgeleitet. Dabei entsteht üblicherweise eine Menge von

[1] https://en.wikipedia.org/wiki/Genetic_algorithm

Lösungen, die sowohl alle Elemente der ursprünglichen Population, als auch die daraus abgeleiteten neuen Lösungen enthält.

Ein gängiger evolutionärer Operator ist die sogenannte *Mutation*. Bei dieser entsteht, analog zum Vorbild aus der Natur, eine neue Lösung durch eine kleine Veränderung einer bestehenden.

c) Nach Anwendung der evolutionären Operatoren wird, über eine sogenannte Fitnessfunktion, die Qualität aller nun vorhandenen Lösungen ermittelt.

d) Ein Selektionsprozess wählt abschließend aus, welche Lösungen die Population der nächsten Iteration bilden. Hierbei wird meist das Prinzip des *Survival of the fittest* umgesetzt, d.h. qualitativ bessere Lösungen werden bevorzugt.

Die Schritte b), c) und d) werden solange zyklisch durchlaufen, bis ein vorher *festgelegtes Kriterium zur Termination* des genetischen Algorithmus (bspw. das Erreichen einer festgelegten Anzahl von Iterationen) erfüllt ist.

Vorgegebene Schnittstelle

In den folgenden Aufgaben sollen Sie zum einen ein Framework für genetische Algorithmen entwickeln und zum anderen ein Optimierungsproblem im Kontext dieses Frameworks modellieren, implementieren und durch Implementierung passender Operatoren lösen.

Ihre Implementierung soll dabei auf einer vorgegebenen Schnittstelle basieren, die als Java-Modul[2] im Sinne des Java Platform Module Systems (JPMS) implementiert wurde. Sie finden diese in Ilias[3] als IntelliJ-Projekt. Erstellen Sie zunächst selbst ein neues *leeres* Projekt *genetic-algorithm*. Importieren Sie anschließend in diesem das bereitgestellte Projekt als IntelliJ-Modul und verwenden Sie dieses Modul als Grundlage für die weiteren Arbeitsschritte.

Durch die vorgegebene Schnittstelle lassen sich Aufgabe 1 und 2 unabhängig voneinander bearbeiten und lösen.

[2] https://www.oracle.com/corporate/features/understanding-java-9-modules.html
[3] https://ilias.uni-marburg.de/goto.php?target=file_2057789_download&client_id=UNIMR

1 Implementierung des Frameworks

Implementieren Sie im Paket *ga.framework* eine Klasse *GeneticAlgorithm*, die den oben beschriebenen Ablauf eines genetischen Algorithmus mithilfe der gegebenen Interfaces und Klassen umsetzt.

1.1 Grundimplementierung (Spezifikation) [5 Punkte]

Um ein Optimierungsproblem mithilfe der Klasse *GeneticAlgorithm* lösen zu lassen, müssen von einem Nutzer folgende Parameter angegeben werden:

- das Problem, welches gelöst werden soll (Klasse *Problem*),
- die Größe der zu verwendenden Population,
- eine Liste evolutionärer Operatoren, die zur Suche neuer Lösungen verwendet werden sollen (Interface *EvolutionaryOperator*),
- die Fitnessfunktion, die zur Bestimmung der Qualität einer Lösung herangezogen werden soll (Interface *FitnessEvaluator*),
- der Selektionsmechanismus (Interface *SelectionOperator*) und
- als Terminierungskriterium die Anzahl der durchzuführenden Iterationen/Evolutionsschritte.

Über eine Methode *runOptimization()* soll der Nutzer die Optimierung durchführen können. Der Ablauf soll dabei wie folgt aussehen:

- Entsprechend der festgelegten Populationsgröße wird zunächst eine Startpopulation erstellt. Die Erstellung einzelner Lösungen wird dabei durch die Methode *createNewSolution()* der konkreten *Problem*-Implementierung übernommen.
- Aus der vom Nutzer angegebenen Liste evolutionärer Operatoren wird in jeder Iteration zufällig *einer* gewählt und mit *evolvePopulation()* auf die aktuelle Population angewandt.
- Anschließend wird, über die Methode *evaluate()* des vom Nutzer festgelegten *FitnessEvaluators*, für alle Lösungen die Fitness bestimmt.
- Der vom Nutzer angegebene *SelectionOperator* wählt abschließend mit der Methode *selectPopulation()* die Population für die nächste Iteration aus.

- Es werden solange neue Iterationen durchgeführt, bis das vom Nutzer angegebene Limit erreicht ist.
- Tritt ein Fehler auf, wird die Optimierung abgebrochen und eine passende Fehlermeldung ausgegeben.
- Konnte die Optimierung ohne Auftreten von Fehlern durchgeführt werden, soll von der Methode *runOptimization()* die in der letzten Iteration entstandene Population zurückgegeben werden.

1.2 Selektionsoperator [2 Punkte]

Implementieren Sie einen einfachen Selektionsoperator *TopKSelection*. Dieser soll, aus der übergebenen Liste von Lösungen, die besten Lösungen für die nächste Population auswählen. D.h. aus der übergebenen Liste von Lösungen werden die besten k Lösungen übernommen, wobei k der Populationsgröße entspricht.

1.3 Erweiterung um Fluent-API [4 Punkte]

Verändern Sie Ihre Implementierung der Klasse *GeneticAlgorithm* so, dass diese ein Fluent-Interface[4] zur Spezifikation der in Aufgabe 1.1 genannten Parameter anbietet. Im Folgenden ist beispielhaft ein entsprechender Aufruf einer Optimierung dargestellt.

```
1 GeneticAlgorithm ga = new GeneticAlgorithm();
2 List<Solution> result = ga.solve(yourProblem)
3         .withPopulationOfSize(10)
4         .evolvingSolutionsWith(yourEvolutionaryOperator)
5         .evolvingSolutionsWith(yourEvolutionaryOperator)
6         .evaluatingSolutionsBy(yourFitnessEvaluator)
7         .performingSelectionWith(yourSelectionOperator)
8         .stoppingAtEvolution(100)
9         .runOptimization();
```

Beachten Sie, dass Ihr *Fluent-Interface* die im Beispiel dargestellte Reihenfolge der Parameter erzwingt. Zudem soll eine Optimierung erst gestartet werden können, wenn alle

[4] https://de.wikipedia.org/wiki/Fluent_Interface

nötigen Parameter festgelegt wurden. Bedenken Sie auch, dass die Angabe mehrerer evolutionärer Operatoren möglich sein muss.

Tipp: Verwenden Sie innere Klassen zur Umsetzung des Fluent-Interfaces.

2 Das Rucksack-Problem

Ein beliebtes Beispiel für Optimierungsprobleme ist das sogenannte Rucksack-Problem[5]. Für dieses sollen Sie nun eine zum Framework aus Aufgabe 1 passende Implementierung schreiben, die es ermöglicht, konkrete Instanzen des Rucksackproblems mithilfe des Frameworks zu lösen.

Die Implementierung von Optimierungsproblemen soll dabei in einem separaten IntelliJ-Modul erfolgen. Zudem soll dieses, genau wie das Framework, als Modul im Sinne von JPMS entwickelt werden. Es ist dabei wichtig zu beachten, dass JPMS und das IntelliJ-Modulsystem zwei unabhängige Systeme zur Verwaltung von Modulen darstellen. Um ein korrektes Zusammenspiel zwischen Framework- und Problemimplementierung zu ermöglichen, müssen Abhängigkeiten in beiden Systemen korrekt spezifiziert werden!

Wer sich noch nicht mit Java-Modulen beschäftigt hat, sollte sich die auf Seite 2 verlinkte Erläuterung aus dem Hause Oracle durchlesen. Zudem könnte eine kurze Einführung zum Umgang mit JPMS-Modulen in IntelliJ[6] interessant sein.

Fügen Sie Ihrem Projekt für die Implementierung des Rucksackproblems zunächst ein neues IntelliJ-Java-Modul *ga.problems* hinzu. Legen Sie in diesem Modul ein Paket *ga.problems.knapsack* an, welches später alle Klassen des Problems enthalten soll. Die im Folgenden genannten Klassen sollen jeweils die entsprechenden Interfaces bzw. abstrakten Klassen der vorgegebenen Schnittstelle implementieren.

2.1 Problem und Lösung [4 Punkte]

Implementieren Sie die Klassen *KnapsackProblem* und *KnapsackSolution*. Die *Problem*-Klasse soll generelle Informationen über das Rucksackproblem (z.B. Kapazität des Rucksacks) halten und über *createNewSolution()* neue *KnapsackSolutions* anlegen können. Beim Erstellen einer neuen Lösung soll dabei solange ein zufälliger Gegenstand in den Rucksack gepackt werden, bis keiner der noch übrigen Gegenstände mehr hinein passt. Für den Fall, dass alle im Problem spezifizierten Gegenstände zu schwer für den Rucksack sind, also kein einziger Gegenstand hineingepackt werden kann, soll eine *NoSolutionException* geworfen und die Optimierung abgebrochen werden.

[5] https://de.wikipedia.org/wiki/Rucksackproblem
[6] https://www.jetbrains.com/help/idea/getting-started-with-java-9-module-system.html

Die *Solution*-Klasse soll Lösungsspezifische Informationen halten (z.B. welche Gegenstände sich im Rucksack befinden). Gegebenenfalls macht es Sinn, zusätzliche (redundante) Informationen zu halten, um die Performanz des Programms zu verbessern.

Selbstverständlich können Sie auch weitere Klassen anlegen, falls Sie diese zur Modellierung von Problem und Lösung benötigen.

2.2 Fitness [2 Punkte]

Implementieren Sie die Klasse *KnapsackFitnessEvaluator*, welche die Fitness aller Lösungen einer Population berechnet. Nutzen Sie zur Berechnung der Fitness einer Lösung Java-Streams und Lambda-Ausdrücke!

2.3 Mutationsoperator [5 Punkte]

Implementieren Sie für das Rucksackproblem *einen* evolutionären Operator *KnapsackMutation*, der zu jeder bestehenden Lösung einer Population zufällig auf eine der folgenden Arten eine neue Lösung erzeugt und diese zur Population hinzufügt:

- Ein zufälliger Gegenstand wird aus dem Rucksack entfernt.
- Aus den Gegenständen, die noch nicht gewählt wurden und noch in den Rucksack passen, wird ein zufälliger gewählt und in den Rucksack gelegt.

Wichtig ist dabei, dass alle bestehenden Lösungen auch nach Anwendung des Operators noch Teil der Population sind. Veränderungen dürfen also nur auf Kopien der bestehenden Lösungen durchgeführt werden. Verwenden Sie zur Erzeugung von Kopien einer Lösung einen konkreten Copy-Konstruktor für *KnapsackSolutions*.

Achten Sie auch darauf, dass zur Erstellung einer neuen Lösung immer exakt *eine* der genannten Veränderungen durchgeführt wird. Sollte keine der beiden Varianten anwendbar sein, ist dies als Fehler (*EvolutionException*) zu behandeln und die Optimierung insgesamt abzubrechen.

Es soll nur *ein* evolutionärer Operator entstehen, der beide Varianten integriert. Verwenden Sie daher innere Klassen, um die unterschiedlichen Varianten der Mutation zu implementieren.

Hinweis

Die Teilaufgaben sind weitestgehend unabhängig voneinander lösbar. Für die Erstellung der Fitnessfunktion und des Mutationsoperators benötigt man jedoch zumindest eine minimale Implementierung (Stub) der *KnapsackSolution*.

3 Ausführung einer Optimierung [2 Punkte]

Erstellen Sie eine Klasse *ConcreteProblem*, die eine Optimierung anhand eines konkreten *KnapsackProblems* demonstriert. Das Beispielproblem soll dabei aus vier Gegenständen g1(Gewicht:5, Wert:10), g2(Gewicht:3, Wert:8), g3(Gewicht:3, Wert:5) und g4(Gewicht:2, Wert:6) sowie einem Rucksack mit einer Kapazität von 11 Gewichtseinheiten bestehen.

Initialisieren Sie in der Klasse das Problem, führen Sie, mit einer Population der Größe 4, eine Optimierung über 10 Iterationen durch und geben Sie auf der Konsole für jede Lösung der finalen Population aus, welche Gegenstände im Rucksack enthalten sind.