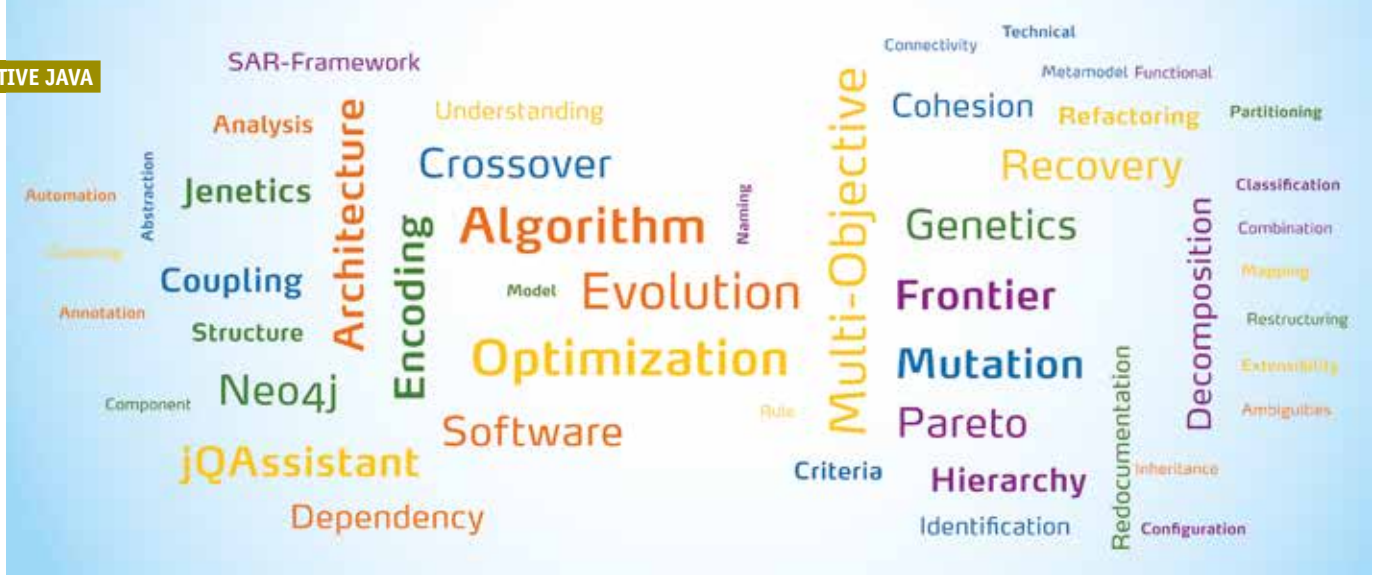

Viele Ziele

Die Evolution im Algorithmus – Teil 2: Multikriterielle Optimierung und Architekturerkennung

Stephan Pirnbaum

Genetische Algorithmen haben sich, wie Teil 1 gezeigt hat, als hilfreiches Mittel zur einfachen Umsetzung komplexer Optimierungsprobleme erwiesen. Dabei sind die Möglichkeiten, die die Evolutionstheorie der Informatik eröffnet, aber noch wesentlich größer und eigentlich nur von der Fantasie der Entwickler begrenzt. Dieser Artikel soll zeigen, wie genetische Algorithmen korrekt mit verschiedenen Optimierungszielen umgehen können und wie es sogar möglich ist, diese zur Erkennung von Softwarestrukturen einzusetzen.

Ein kurzer Rückblick

Der erste Teil in Heft 01/2018 [Pir18] hat gezeigt, wie genetische Algorithmen eingesetzt werden können, um Optimierungsprobleme mit wenig Aufwand zu lösen. Dabei haben wir gelernt, dass solche Algorithmen die Grundprinzipien der Evolution, nämlich Selektion, Vererbung und Mutation, implementieren. Beispielhaft wurde dafür das Ressourcenplanungsbeispiel eingeführt, welches das Ziel hat, Tasks unter Minimierung von Zeit und Kosten Ressourcen zuzuweisen. Wie sich herausstellte, stehen diese zwei Objectives allerdings in Konkurrenz zueinander. Die Single-Objective Optimization (SOO) kombinierte diese konfligierenden Objectives in einer Funktion. Zwar lieferte dieser Ansatz zufriedenstellende und sinnvolle Ergebnisse, wirft jedoch auch eine ganze Reihe von Problemen auf.

Optimierung in Zeiten großer Ansprüche

Die Angepasstheit (Qualität) eines Individuums lässt sich meist nicht an nur einem Kriterium festmachen. Ein Auto soll vielleicht möglichst schnell, jedoch auch kraftstoffsparend sein. Oder, wie in unserem Beispiel, alle Tasks sollten in kürzest möglicher Zeit zu geringstmöglichen Kosten durchgeführt werden. Hier erfolgt ein Trade-off, welcher durch die Gewichtung der einzelnen Objectives in der Optimierungsfunktion gesteuert wird.

In dieser Gewichtung liegt das Problem begründet: Die Evolution kann sich nämlich in die falsche Richtung entwickeln. Das ist der Fall, wenn keine Parametrisierung gegeben ist und kein oder ein nichtlinearer Zusammenhang zwischen den Objectives vorliegt. Hierbei wird immer ein Objective dominieren. Außerdem ist der Einfluss von Objectives nicht per se festlegbar, wenn dieser entweder zu komplex zu erfassen ist (besonders bei mehr als zwei Objectives) oder schlicht unbekannt ist. Dadurch ist die Evolution beeinflusst, sie wird also den Suchraum in eine bestimmte Richtung durchsuchen. Im Gegenzug sollte SOO verwendet werden, wenn diese Faktoren eindeutig bestimmbar sind, da dies einer gezielten Suche entspricht. Was aber tun, wenn das nicht der Fall ist?

Die sogenannte *Multi-Objective Optimization* (MOO) oder auch Pareto-Optimierung [Wiki] ist eine multikriterielle Optimierungsmethode, welche zum Ziel hat, Objectives weder zu bevorzugen noch zu benachteiligen. Sie besagt für zwei Individuen A und B, dass wenn:

- A in mindestens einem Objective besser und in keinem schlechter als B ist, B von A dominiert wird. Damit stellt A eine bessere Lösung dar.
- A in mindestens einem Objective besser, aber auch in mindestens einem schlechter als B ist, A und B nicht paretovergleichbar sind.

Bei der Exploration des Suchraums ergibt sich dadurch eine Menge an optimalen Lösungen, welche das Hauptaugenmerk auf unterschiedliche Bereiche legt. Das ermöglicht eine gleichmäßige, breite Exploration der möglichen Lösungen. Die Menge der optimalen Lösungen wird als Pareto-Front bezeichnet.

Abbildung 1 stellt diese beispielhaft für die zwei Objectives Zeit-Optimalität und Kosten-Optimalität dar, welche umso höher sind, je niedriger Zeit beziehungsweise Kosten sind. Die Einträge

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

s1 bis s11 spiegeln Individuen wider, welche unterschiedliche Qualitäten aufweisen. So weisen s2 und s8 zwar die gleichen Kosten auf, jedoch benötigt s2 weniger Zeit. Dadurch wird s8 von s2 dominiert. Im Gegenzug weist s3 geringere Kosten auf, benötigt dafür aber auch mehr Zeit als s2. Dadurch sind diese beiden Individuen nicht paretovergleichbar. Die Pareto-Front ergibt sich aus der Menge der optimalen Lösungen s1 bis s7.

Eine Umsetzung multikriterieller genetischer Algorithmen stellt NSGA-II (Nondominated Sorting Genetic Algorithm II) [Deb02] dar. Die Implementierung der multikriteriellen Optimierung erfolgt ähnlich dem des in Teil 1 vorgestellten genetischen Algorithmus und ist in Listing 1 als Pseudocode dargestellt.

Was auf den ersten Blick kompliziert aussieht, erweist sich in der Praxis als einfach umzusetzen. Aus den Individuen der Population werden nichtdominierte Fronten gebildet (für Abb. 1 existieren die drei Fronten {s1, s2, s3, s4, s5, s6, s7}, {s8, s9, s11} und {s10}). Da eine hohe Diversität gewünscht ist, sollten die Lösungen weit über den Suchraum verteilt sein. Dies erreicht man, indem man Individuen mit großem euklidischen Abstand selektiert. Dazu wird für jede Front der euklidische Abstand zwischen den Individuen berechnet und die Front zu den Survivors hinzugefügt, sollte die maximale Größe N nicht überschritten werden. Ansonsten wird die Front auf Basis der Distanz sortiert und die besten, also am weitesten auseinanderliegenden Elemente werden hinzugefügt. Zum Schluss erfolgt, wie bei genetischen Algorithmen üblich, die Selektion (auf Basis des Ranks, also der Optimalität der Front, sowie der Distanz), Crossover (also die Paarung der Individuen) sowie Mutation.

Multikriterielle Ressourcenplanung

Die vorgestellte Theorie soll nun in die Praxis umgesetzt werden. Dabei wird wie in Teil 1 die Bibliothek Jenetics [Jen] verwendet, welche seit Version 4.1.0 Unterstützung für MOO bietet. Dafür werden die Module *jenetics* und *jenetics.ext* benötigt, Letzteres beinhaltet den MOO-Support. Der Quellcode ist wieder unter [Git] zu finden.

Was soll umgesetzt werden? Das Ressourcenplanungsbeispiel aus Teil 1 hatte zum Ziel, einer Menge an Tasks eine Menge an Ressourcen mit geringstmöglichem Kosten- und Zeitaufwand zuzuweisen. Die zwei Objectives sind also die Minimierung der Kosten sowie die Minimierung der Zeit. Während diese beiden Ziele zuletzt noch in der Formel:

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

vereint wurden, sollen nun die Vorzüge des MOO genutzt werden.

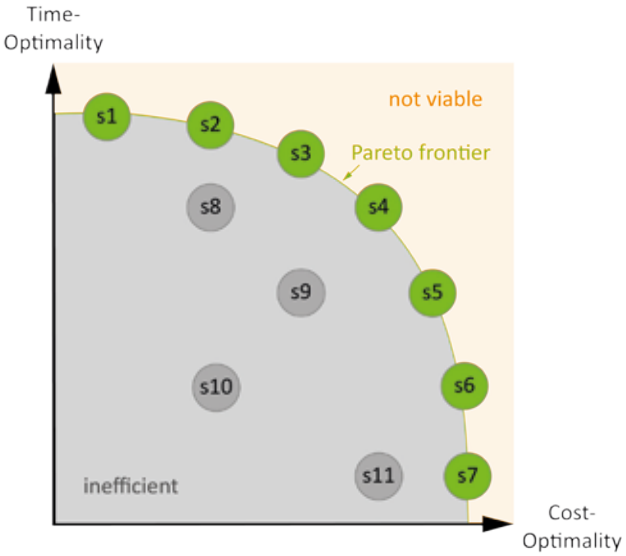

Abb. 1: Pareto-Front für zwei Objectives

Zunächst muss wieder die Codierung bekannt gegeben werden. Wie in Teil 1 verwenden wir dafür einen Integer-Genotypen, welcher die Zuweisung von Task zu Ressource vornimmt. Anschließend wird die *Engine* gebaut, welche die Konfiguration des Genetic Algorithm (GA) enthält. Listing 2 zeigt die Konfiguration für die MOO.

Im Vergleich zur Konfiguration für den SOO-Ansatz fallen einige Dinge auf. Der zweite Typparameter der Engine (der Datentyp der Fitness) ist nun nicht mehr ein einfacher Double, sondern ein `Vec`. Dieser repräsentiert einen Vektor, welcher in unserem Fall ein Array von `double`-Werten als zugrunde liegende Datenstruktur beherbergt. Dabei wird jedes Objective als eigener Eintrag in dem Array hinterlegt. Das sehen wir direkt bei dem Aufruf der `builder`-Methode, welche einen Lambda-Ausdruck übergeben bekommt, der einen `Genotype` auf einen Vektor der Länge zwei abbildet. Der erste Eintrag des Vektors stellt die benötigte Zeit, der zweite die verursachten Kosten dar. Die Konfiguration der Populationsgröße und der Crossover- und Mutationsoperatoren erfolgt analog dem SOO-Beispiel.

Kommen wir aber zur Definition der Selektoren, müssen wir eine Unterscheidung treffen. Wie wir aus Listing 1 wissen, müssen für alle Individuen die Nondominated Fronts gebildet sowie die Crowding Distance berechnet werden. Auf dieser Basis erfolgt die Auswahl der Survivor, also derer, die sich weiterentwickeln dürfen. Diese Aufgabe übernimmt der vorkonfigurierte NSGA2-Selek-

```
pop = initial population (size: N)
while (true)
  fronts = nondominatedFronts(pop)
  crowdingDistance(f)
  for (f in fronts)
    if (|surv| + |f| <= N)
      surv += f
    else
      sort(f)
      surv += f[1: N - |surv|]
      break
  surv = selectFittest(surv)
  pop = crossover(surv)
  pop = mutate(pop)
}
```

Listing 1: NSGA-II-Optimierung

```
Genotype<IntegerGene> gt = // init
Engine<IntegerGene, Vec<double[]>>
  Engine = Engine
  .builder(i ->
    Vec.of(time(i), costs(i)), gt)
  .popuationSize(500)
  .survivorsSelector(
    NSGA2Selector.vec())
  .alterers(
    new SinglePointCrossover<>(1),
    new Mutator<>(0.01))
  .minimizing()
  .build();
```

Listing 2: Konfiguration der GA Engine

```
ISeq<Phenotype<IntegerGene, Vec<double[]>>> paretoFrontier =
  engine.stream()
    .limit(
      Limits.bySteadyFitness(500))
    .collect(MOEA.toParetoSet());
```

Listing 3: Ausführen der Evolution

tor, welcher über `NSGA2Selektor.vec()` erhalten werden kann. Mit `minimizing()` bringen wir zum Ausdruck, dass die Objectives minimiert werden sollen.

Die Evolution starten wir, indem wir den `EvolutionStream` wie in Listing 3 gezeigt ausführen. Als Ergebnis erhalten wir nun kein einzelnes Individuum mehr, sondern eine Menge optimaler Individuen durch den Aufruf von `MOEA.toParetoSet()`. Dabei wird die Pareto-Front auf 75 bis 100 Individuen eingeschränkt. Dieser Größenbereich stellt das Standardverhalten dar, welches nach Bedarf angepasst werden kann.

Die Ergebnismenge kann nun mit geeigneter Methodik weiter minimiert werden, beispielsweise über manuelle Abwägung der Lösungen oder auch programmatisch durch vordefinierte Regeln (geringste Kosten, geringste Zeit, zentrale Lösung, Kombination der Objectives...). Das ist aber nicht weiter problematisch, da die MOO zum Ziel hat, eine gleichmäßige, nicht-beeinflusste Suche zu ermöglichen und nicht ein konkretes Ergebnis zum Vorschein zu bringen.

Softwarearchitektur-Recovery mit genetischen Algorithmen

Mit den bis hierhin vorgestellten Grundlagen zu genetischen Algorithmen sowie der Bibliothek Jenetics ist es möglich, zahlreiche Optimierungsprobleme zu lösen. Dabei ist das Problem viel weniger die eigentliche Implementierung, sondern das Erkennen, dass sich das jeweilige Problem als Optimierungsproblem betrachten lässt. Es soll daher als Abschluss noch ein auf den ersten Blick weniger naheliegendes Problem betrachtet werden.

Eine klare Struktur von Softwaresystemen ist ein entscheidender Einflussfaktor auf den Aufwand und den Erfolg von Projekten. Umso mehr ist es erstaunlich, besonders bei großen, gewachsenen Systemen oftmals nur degenerierte Strukturen vorzufinden. In solchen Fällen kann eine Restrukturierung des Systems helfen, um den Entwicklungsprozess und die Stabilität in Zukunft zu verbessern. Manuell ist das aber bei Systemen mit mehreren Tausend Klassen und noch viel mehr Abhängigkeiten zwischen diesen ein sehr großer Aufwand. Wäre hier die Verwendung eines genetischen Algorithmus angebracht?

Ja! Dafür schauen wir uns zunächst an, worin der strukturelle Aspekt der Architektur besteht. Betrachten wir ein Softwaresystem in seiner Gänze, dann ist eine Dekomposition eine Zerlegung dessen in einzelne Komponenten, zum Beispiel Schichten oder Module. Im einfachsten Fall ist die definierte Paketstruktur bereits eine hierarchische Dekomposition. Dabei wäre ein Paket eine Komponente, welche wiederum Komponenten beinhalten und von anderen Komponenten abhängen kann. Die bestehende Paketstruktur ist aber natürlich nicht die einzig mögliche Zerlegung des Softwaresystems und meist auch nicht die beste.

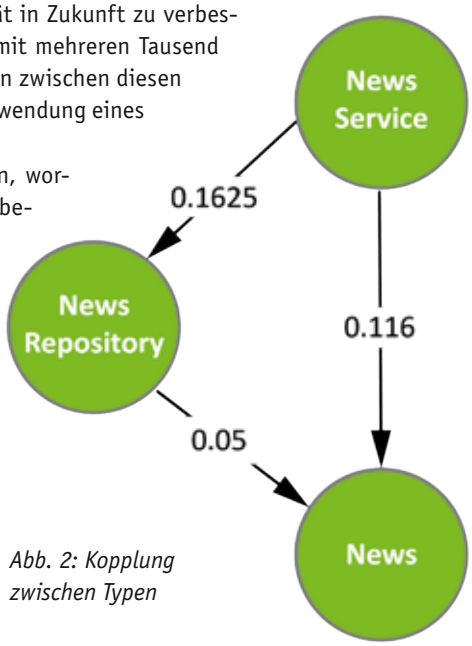

Abb. 2: Kopplung zwischen Typen

Ein fiktives System mit 5 Klassen kann auf 52 verschiedene Weisen zerlegt werden. Das ist eine überblickbare Menge, welche man per Brute Force durchsuchen kann. Bei 50 Typen ändert sich dies aber drastisch. Es existieren nun etwa 1,86E47 verschiedene Zerlegungen. Diese alle zu untersuchen, würde schlicht zu lange dauern. Wir können das Problem der optimalen Zerlegung aber als Suchproblem definieren, womit es zu der Klasse der Optimierungsprobleme gehört.

Ein großer Suchraum ist ein entscheidendes Kriterium, warum es sinnvoll ist, einen genetischen Algorithmus einzusetzen. Notwendig ist aber des Weiteren die Bewertbarkeit der Zerlegungen. Als Mensch können wir einschätzen, ob eine Zerlegung sinnvoll ist. Die Frage ist nur, was eine gute Dekomposition qualifiziert. Wenn wir ein System in Komponenten zerlegen, dann wollen wir, dass diese voneinander klar abgegrenzt sind. Die *Kopplung* bezeichnet, wie stark Komponenten voneinander abhängen. Unser erstes Objective ist es also, die Kopplung zwischen Komponenten zu minimieren.

Im Gegenzug dazu sollen aber auch die Bestandteile einer Komponente einen starken Zusammenhang haben. Die *Kohäsion* bezeichnet den inneren Zusammenhalt einer Komponente. Unser zweites Objective ist es also, die Kohäsion in den Komponenten zu maximieren.

Beide Werte können wir über die Anzahl und Stärke der Relationen (Methodenaufrufe, Vererbungsbeziehungen usw.) zwischen Komponenten/Typen berechnen. Abbildung 2 stellt drei Typen sowie deren Abhängigkeiten untereinander dar. Die Werte an den Relationen repräsentieren dabei jeweils deren Stärke zwischen 0 (keine Abhängigkeit) und 1 (größtmögliche Abhängigkeit).

Betrachten wir zwei Typen, so stellt die Relation zwischen diesen deren Kopplung dar. Wenn wir hingegen alle drei Typen als Bestandteil einer Komponente auffassen, so berechnet sich die Kohäsion im einfachsten Fall aus der Summe der Gewichte der Relationen.

Diese zwei Objectives stehen des Weiteren in Konkurrenz zueinander: Die Kohäsion sinkt, wenn kleinere Komponenten erstellt werden, wohingegen die Kopplung bei steigender Komponentenzahl steigt. Damit haben wir bereits zwei Objectives, welche in Konkurrenz zueinanderstehen und welche es uns ermöglichen, Dekompositionen zu bewerten. Das Open-Source-Projekt SAR-Framework [SAR] implementiert diese sowie drei weitere Objectives, welche hier aufgrund der Komplexität nicht weiter betrachtet werden sollen.

Das Software Architecture Recovery Framework (SAR) nutzt neben einer großen Anzahl an Objectives, welche mit Jenetics leicht spezifiziert werden können, aber auch einige andere, fortgeschrittene Techniken, die im Folgenden skizziert werden sollen, um die Mächtigkeit genetischer Algorithmen zu illustrieren.

Gesteuerte Mutation von Individuen

Bis zu diesem Punkt wurden zur Mutation der Individuen immer vorgefertigte Strategien verwendet. Diese sind zufallsbasiert in

der Auswahl der zu verändernden Gene sowie der Zielwerte. Aber ist es überhaupt sinnvoll, degenerierte Individuen während der Evolution zuzulassen?

Innerhalb einer Gesellschaft wäre diese Frage ethisch sehr umstritten. Aber in unserem Fall ist es nicht notwendig, Individuen evolvieren zu lassen, welche Komponenten definieren, die wiederum in Komponenten zerfallen können (schlechte Kohäsion). Deswegen implementiert das SAR-Framework eine eigene Mutationsoperation, welche die durch zuvor durchgeführte zufällige Mutationsoperatoren evolvierten Individuen auf ihre Gültigkeit überprüft und Komponenten gegebenenfalls teilt. Dies ist möglich, indem man das vordefinierte Interface `io.jenetics.Alterer` implementiert. Bei diesem handelt es sich um ein Functional Interface, welches die Methode `alter` definiert. Diese erhält als Parameter eine Population zum Mutieren sowie die Generationsnummer der neuen Generation übergeben. Als Ergebnis liefert sie ein `AltererResult` zurück.

Jenetics unterstützt grundsätzlich zwei Arten von Alterern. Zum einen sind dies `Recombinator`, welche beispielsweise die Umsetzung von Crossover ermöglichen und aus mindestens zwei Individuen ein neues generieren. Zum anderen gibt es die Ausprägung `Mutator`, welche auf jeweils einem Individuum die Ausprägung der Gene verändern.

Möchte man sich Arbeit ersparen, ist es sinnvoll, sich die jeweiligen Ausprägungen genau anzuschauen und zu bewerten, von welchem Typ abgeleitet werden sollte. Der im SAR-Framework implementierte `SplitMutator`, welcher nicht-zusammenhängende Komponenten zerteilt, erweitert beispielsweise direkt `Mutator`.

Listing 4 stellt beispielhaft die Implementierung eines sehr einfachen Mutators vor, welcher schlicht den Wert jedes Gens um eins erhöht beziehungsweise auf null setzt, wenn der vordefinierte Maximalwert überschritten werden würde. Es fällt auf, dass hier eine Methode mit dem Namen `mutate` überschrieben wird, welche ein Gen übergeben bekommt, und nicht die vorher besprochene `alter`-Methode. Das ist möglich, da die `Mutator`-Klasse diese bereits selbst implementiert und einzelne Methoden für die Mutation von Phänotyp, Genotyp, Chromosom und Gen zur Verfügung stellt. Dadurch ist es nicht mehr nötig, die gesamte Logik zu implementieren, sondern man kann auf einem spezifischen Niveau ansetzen. Diesen Mechanismus nutzt das SAR-Framework ebenfalls, um das Verschieben von Komponenten beziehungsweise von Typen in eine andere Komponente anhand der Kopplung zu steuern. Um den Zufall zu bewahren und trotz der gesteuerten Mutationsoperationen eine breite Suche zu ermöglichen, werden diese beiden Mutatoren als Zusatz zu zufällig agierenden Mutatoren von Jenetics eingesetzt.

```
class IncrementingMutator extends
    Mutator<IntegerGene, Integer> {
  @Override protected IntegerGene
  mutate(IntegerGene gene, Random random) {
    return gene.newInstance(
      (gene.intValue() + 1) % (gene.getMax() + 1));
  }
}
```

Listing 4: Nutzerspezifische Mutatoren

Eine automatische Dekomposition

Zum Schluss soll noch kurz das SAR-Framework im Einsatz betrachtet werden. Für eine detailliertere Darstellung sei der Blog-Beitrag *„Dein System: Das unbekannte Wesen"* [Bus], welcher das Tool aus Anwendersicht beleuchtet, empfohlen. Das SAR-Framework ermöglicht es, ein mittels jQAssistant [jQA] eingelesenes Softwaresystem zu analysieren. Dabei ist es zunächst möglich, über Artefakt-, Klassen- und Paketnamen festzulegen, welche Klassen betrachtet werden sollen. Das SAR-Framework startet dann mit der Paketstruktur als initialer Population die Evolution und optimiert die Zerlegung auf der Basis der fünf Objectives.

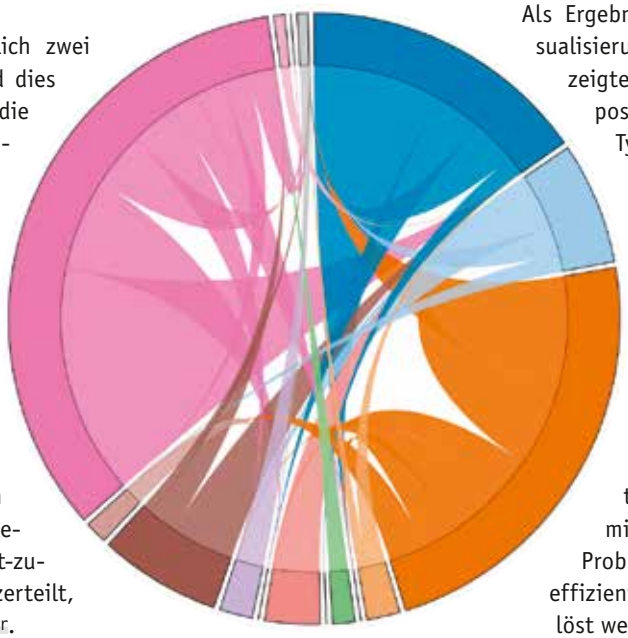

Abb. 3: Ergebnisse des SAR-Frameworks

Als Ergebnis erhält man eine interaktive Visualisierung ähnlich der in Abbildung 3 gezeigten, welche die hierarchische Dekomposition mitsamt deren Komponenten, Typen und Abhängigkeiten darstellt. Damit ist es leicht möglich, konkret Abhängigkeiten zwischen Typen zu finden und mögliche Hotspots zu entdecken.

Schlussbetrachtung

In den beiden Teilen über die Evolution im Algorithmus haben wir eingehend die Theorie hinter genetischen Algorithmen beleuchtet und mittels Beispielen gezeigt, wie man Probleme so formulieren kann, dass sie effizient mit genetischen Algorithmen gelöst werden können. Auch haben wir untersucht, wie man zur Effizienzsteigerung eine gesteuerte Mutation erreicht, ohne aber den Zufall zu verlieren. Der Inhalt wurde schlussendlich von einer Betrachtung des aktuell in Entwicklung befindlichen SAR-Frameworks abgerundet, welches es ermöglicht, Strukturen in Softwaresystemen zu erkennen.

Literatur und Links

[Bus] Buschmais GbR, Dein System: Das unbekannte Wesen?, http://www.buschmais.de/2018/03/07/dein-system-das-unbekannte-wesen/

[Deb02] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, in: IEEE Trans. Evol. Comput., 2002

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

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

[jQA] jQAssistant, Startseite, https://jqassistant.org/

[Pir18] St. Pirnbaum, Die Evolution im Algorithmus – Teil 1: Grundlagen, in: JavaSPEKTRUM, 1/2018

[SAR] SAR-Framework, https://github.com/buschmais/sar-framework

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