Nature-Inspired Algorithms

by

Yves Beutler

Erstellt an der BFH Technik und Informatik
in teilweiser Erfüllung der Anforderungen an den Titel

Bachelor of Science BFH in Informatik

an der

BERNER FACHHOCHSCHULE

Juni 2019

Autor .....
Yves Beutler
BFH Technik und Informatik
10. Juni, 2019

Betreut durch .....
Dr. Mascha Kurpicz-Briki
Betreuende Professorin

Nature-Inspired Algorithms

by

Yves Beutler

Erstellt an der BFH Technik und Informatik
am 10. Juni, 2019, in teilweiser Erfüllung der
Anforderungen an den Titel
Bachelor of Science BFH in Informatik

Abstract

Nature-Inspired Algorithms können dort eingesetzt werden, wo traditionelle Problemlösungsmethoden nicht funktionieren. Einige dieser Algorithmen haben sich als äusserst performant und robust erwiesen und werden heute zur Problemlösung in den unterschiedlichsten Anwedungsgebieten eingesetzt. Zu der Gruppe der Nature-Inspired Algorithms gehören beispielsweise Evolutionäre-Algorithmen, welche starke Ähnlichkeiten zu Darwins Evolutionstheorie mit seinem bekannten Zitat von 'Survival of the fittest aufweisen'. Weiter werden Schwarm- und Neuronale-Algorithmen thematisiert, wobei aktuell die letzteren, durch die neusten Grafikkartengenerationen herbeigeführt, einen erneuten Aufschwung erleben dürfen. Nach einer kurzen Übersicht der einzelnen Gruppen wird jede behandelte Unterart der Nature-Inspired Algorithms durch einen ausgewählten Algorithmus im Detail veranschaulicht.

Ziel dieser Arbeit ist es, eine Einführung in die Thematik von Nature-Inspired Algorithms und ihren Einsatzmöglichkeiten zu geben und die bekanntesten Unterarten mit ausgesuchten Algorithmen im Detail zu erläutern.

Betreuende Professorin: Dr. Mascha Kurpicz-Briki

Danksagung

Special thanks to Dr. Mascha Kurpicz-Briki for supporting me during the semester while I was working on this term paper and Natalie Kuster for providing me with additional information about genetic algorithms and especially data science related topics I covered in the next few pages.

Another thank you to Kieran Willis for proofreading my term paper.

Inhaltsverzeichnis

List of Figures

List of Tables

Kapitel 1

Einführung

Seit dem Anbeginn unseres Planeten sah sich die Natur mit mehr oder weniger komplexen Problemstellungen konfrontiert, um das Überleben ihrer Bewohner sicherzustellen. Der Schlüssel zum Erfolg ist es, sich am besten auf die naturgegebenen Umstände anzupassen. Ohne den Einklang mit der Natur wären die Überlebenschancen einer jeden Spezies schwindend gering wenn nicht gar inexistent. Seit Jahrzehnten adaptieren Menschen die Techniken aus der Natur und ihren Geschöpfen um moderne Technologien auf ein neues Level zu befördern. Flugzeuge werden nach den Flügelcharakteristiken von Zugvögeln konstruiert und neue Klebstoffe werden durch das Studium von Geckofüssen mit ihren unglaublichen Hafteigenschaften entworfen. [Cro14]

Wir dürfen nicht ausser Acht lassen, dass die Natur Millionen von Jahre in die Forschung nach dem Schlüssel zum Überleben investierte und in den meisten Fällen scheiterte. Doch die Natur gab nicht einfach auf, sondern justierte bestimmte Parameter um über die Zeit zunehmend besser zu werden. Durch sich ständig ändernde Umstände sucht sie auch heute noch nach Lösungen für zukünftige Spezien.

Wieso sollte die Menschheit nicht selbst von diesen Millionen von Jahren an Forschung profitieren?

1.1 Anwendungsgebiete

Nature-Inspired Algorithms werden meist dort eingesetzt, wo traditionelle Algorithmen keine oder keine brauchbaren Ergebnisse liefern können. Folgende Gegebenheiten können Gründe sein, um auf Nature-Inspired Algorithms zurückzugreifen:

1. Die Anzahl möglicher Lösungen im Suchraum ist so immens gross, dass die Suche nach der bestmöglichen Lösung viel zu aufwändig wäre.

2. Die Komplexität der Problemstellung ist derart hoch, dass nur vereinfachte Modelle bei der Problemlösung zum Einsatz kommen und dadurch keine aussagekräftigen Lösungen entstehen.

3. Die Evaluierungsfunktion zur Bewertung einer möglichen Lösung variert mit der Zeit, so dass nicht nur eine sondern eine Vielzahl von Lösungen erforderlich ist.

[Bro11, Kap. 1.2]

1.1.1 Optimierungen

Häufige Anwendungsfälle dieser Algorithmen sind Optimierungen von Funktionen. Bei Optimierungen handelt es sich in der Regel um die Suche nach einer Parameterkombination für eine gegebene Funktion f um eine Kostenfunktion zu minimieren oder eine Wertefunktion zu maximieren.

1.1.2 Approximationen

Die Approximation beschreibt meist eine Funktion f, welche möglichst nahe an eine Zielfunktion angenähert werden möchte. Die approximierte Funktion f wird aus einem Set an Beobachtungen[1] generiert. Solche Approximationen werden häufig für Image Recognition verwendet und spielen ebenfalls eine wichtige Rolle bei der Klassifikation und dem Clustering von grossen Datenmengen.

[1] oftmals auch als 'training set' aus der Data Science bekannt

Kapitel 2

Evolutionäre Algorithmen

Diese Subkategorie von Algorithmen wurde durch den natürlichen Prozess der Evolution inspiriert. Charles Darwin begründete 1838 den Evolutionsprozess und das daraus resultierende Überleben der jeweils am besten angepassten Individuen mit der natürlichen Selektion. Diese stellt sicher, dass sich gut an den Lebensraum angepasst Lebewesen in der Natur durchsetzen können, währenddessen weniger gut angepasste Individuen aussterben. Zudem arbeitet die Natur nach dem 'Trial and Error' Prinzip. Dadurch werden Mutationen an Lebewesen ausprobiert und bei positivem Effekt auf das Überleben der Spezies beibehalten, bei einer Verminderung der Überlebenschancen jedoch wieder rückgängig gemacht. Dieser Prozess ist sehr zeitaufwändig, jedoch nachhaltig wirksam. Viele Algorithmen wie beispielsweise der Hill Climbing Algorithmus wenden ebenfalls dieses Prinzip an.

2.1 Anwendungsgebiete

hier anwendungsfälle definieren

2.2 Genetische Algorithmen

Die Genetischen Algorithmen sind die bekanntesten Vertreter aus der Gruppe der Evolutionären Algorithmen. Sie orientierten sich an der Populationsgenetik und ins-

besondere an den Mendelschen Regeln. [McC00] Bei Genetischen Algorithmen werden häufig Begriffe aus der Biologie verwendet. Die Anzahl an möglichen Lösungen nennt man Population. Eine einzelne Lösungsvariante wird als Chromosom bezeichnet, wohingegen diese wiederum aus einer Kombination an Variablen, in der Biologie Gene genannt, bestehen. Wie aus der Grafik 2-1 zu entnehmen ist, besteht ein Chromosom A_n aus mehreren Genen.

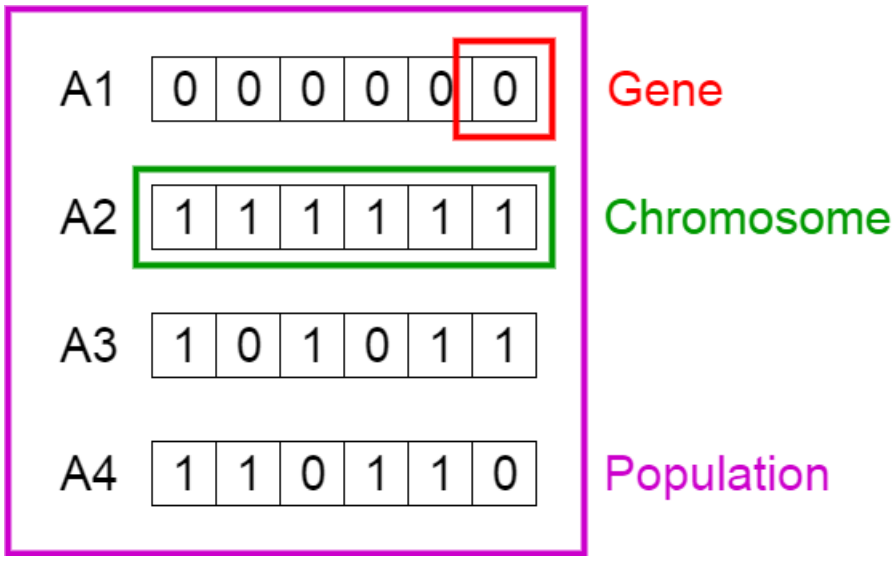

Figure 2-1: Population, Chromosome und Gene

Im folgenden werden die einzelnen Schritte erläutert, welche die Natur wie auch die Genetischen Algorithmen durchlaufen, um eine Anfangspopulation schrittweise weiterzuentwickeln. Die Natur sucht nach der erfolgsversprechendsten Lösung zum Überleben, wohingegen die Algorithmen optimale Lösungen für ganz unterschiedliche Problemstellungen suchen. [Mal17]

2.2.1 Initiale Population

Eine Population besteht aus mehreren unabhängigen Lösungen, im biologischen Kontext Chromosomen, um eine bestimmte Problemstellung zu lösen. Bei den meisten Implementationen von Genetischen Algorithmen wird ein Gen durch einen String repräsentiert. Bei Chromosomen eignen sich auch Binärwerte um das Vorhandensein

eines Gens auszuzeichnen.

2.2.2 Selektion

Anhand einer sogenannten Fitnessfunktion wird für jedes Individuum eine Punktzahl berechnet. Die Punktzahl gibt darüber Auskunft, wie kompetitiv sich ein Individuum gegenüber einem anderen verhält. Je höher dieser Wert, desto höher sind die Chancen, das besagtes Individuum von der Selektion für die Reproduktion verwendet wird. Durch die Selektion können nur die fittesten Individuen ihre Gene an die nächste Generation weitergeben und den schwächeren Individuen wird der Reproduktionszyklus verwehrt.

2.2.3 Kreuzung

Der womöglich zentralste Bestandteil eines Genetischen Algorithmus ist die Kreuzung. Hier werden aus den zuvor selektierten Individuen, einer Ansammlung der vielversprechendsten Lösungen, zwei Individuen zur Paarung ausgewählt. Die beiden Ausgewählten tauschen ihre Gene bis zu einem bestimmten Punkt, dem sogenannten Crossover Point, aus. Die restlichen Gene hinter diesem Punkt werden nicht verändert. Durch diesen Prozess entstehen gleich zwei neue Individuen.

2.2.4 Mutation

Die Natur nutzt Mutationen aus, um die Artenvielfalt innerhalb einer Population zu gewährleisten. Trotz der geringen Wahrscheinlichkeit kommt es vor, dass einzelne Gene, welche durch die Kreuzung eigentlich vorhanden wären, nicht mehr vorhanden sind und umgekehrt. Ein Genetischer Algorithmus flippt beispielsweise einzelne Variablen seiner Nachkommen.

2.2.5 Terminierung

Ein Genetischer Algorithmus terminiert, sobald sich die nächste Generation nicht mehr merklich von ihren Vorgängern unterscheidet. Man spricht oftmals auch von

Konvergenz. Es ist jedem Algorithmus selbst überlassen, wie lange ein Individuum in der Population überlebt. Häufig werden bei jedem Reproduktionszyklus ein oder mehrere Individuen mit den niedrigsten Fitnesswerten aus der Population gestrichen. In der Natur übernimmt der Tod diese Funktion.

2.2.6 Beispiel Implementation

Die einzelnen Schritte eines Genetischen Algorithmus sind in Listing 2.1 durch ein Python-Beispiel verdeutlicht:

```
1  # randomly initialize population
2  population = [a, b, c, d]
3  generation = 1
4
5  # run genetic algorithm
6  genetic_algorithm(population)
7
8  def genetic_algorithm(init_population=[]):
9    population = init_population
10   for individual in population:
11     print('hello')
12   return population
```

Listing 2.1: Genetischer Algorithmus (Python)

2.3 Weitere Algorithmen

Weitere Unterarten von Evolutionären Algorithmen sind beispielsweise: - Genetic programming - evolution strategy - neuroevolution (NEAT)

Another important motivation was the trend towards placing more of the burden of performance on the compiler. Many of the new architectures depend on an intelligent, optimizing compiler in order to realize anywhere near their peak performance [**?**, **?**, **?**]. In these cases, the compiler not only is responsible for faithfully generating native code to match the source language, but also must be aware of instruction latencies, delayed branches, pipeline stages, and a multitude of other factors in order to generate fast code [**?**].

Taking these ideas one step further, it seems that the floating point operations that are normally single, large instructions can be further broken down into smaller, simpler, faster instructions, with more control in the compiler and less in the hardware. This is the idea behind a micro-optimizing FPU; break the floating point instructions down into their basic components and use a small, fast implementation, with a large part of the burden of hardware allocation and optimization shifted towards compile-time.

Along with the hardware speedups possible by using a μFPU, there are also optimizations that the compiler can perform on the code that is generated. In a normal sequence of floating point operations, there are many hidden redundancies that can be eliminated by allowing the compiler to control the floating point operations down to their lowest level. These optimizations are described in detail in section 2.4.

2.4 Description of micro-optimization

In order to perform a sequence of floating point operations, a normal FPU performs many redundant internal shifts and normalizations in the process of performing a sequence of operations. However, if a compiler can decompose the floating point operations it needs down to the lowest level, it then can optimize away many of these redundant operations.

If there is some additional hardware support specifically for micro-optimization, there are additional optimizations that can be performed. This hardware support entails extra "guard bits" on the standard floating point formats, to allow several

unnormalized operations to be performed in a row without the loss information[1]. A discussion of the mathematics behind unnormalized arithmetic is in appendix **??**.

The optimizations that the compiler can perform fall into several categories:

2.4.1 Post Multiply Normalization

When more than two multiplications are performed in a row, the intermediate normalization of the results between multiplications can be eliminated. This is because with each multiplication, the mantissa can become denormalized by at most one bit. If there are guard bits on the mantissas to prevent bits from "falling off" the end during multiplications, the normalization can be postponed until after a sequence of several multiplies[2].

As you can see, the intermediate results can be multiplied together, with no need for intermediate normalizations due to the guard bit. It is only at the end of the operation that the normalization must be performed, in order to get it into a format suitable for storing in memory[3].

2.4.2 Block Exponent

In a unoptimized sequence of additions, the sequence of operations is as follows for each pair of numbers (m_1,e_1) and (m_2,e_2).

1. Compare e_1 and e_2.
2. Shift the mantissa associated with the smaller exponent $|e_1 - e_2|$ places to the right.
3. Add m_1 and m_2.
4. Find the first one in the resulting mantissa.

[1] A description of the floating point format used is shown in figures **??** and **??**.

[2] Using unnormalized numbers for math is not a new idea; a good example of it is the Control Data CDC 6600, designed by Seymour Cray. [**?**] The CDC 6600 had all of its instructions performing unnormalized arithmetic, with a separate `NORMALIZE` instruction.

[3] Note that for purposed of clarity, the pipeline delays were considered to be 0, and the branches were not delayed.

5. Shift the resulting mantissa so that normalized

6. Adjust the exponent accordingly.

Out of 6 steps, only one is the actual addition, and the rest are involved in aligning the mantissas prior to the add, and then normalizing the result afterward. In the block exponent optimization, the largest mantissa is found to start with, and all the mantissa's shifted before any additions take place. Once the mantissas have been shifted, the additions can take place one after another[4]. An example of the Block Exponent optimization on the expression X = A + B + C is given in figure **??**.

2.5 Integer optimizations

As well as the floating point optimizations described above, there are also integer optimizations that can be used in the μFPU. In concert with the floating point optimizations, these can provide a significant speedup.

2.5.1 Conversion to fixed point

Integer operations are much faster than floating point operations; if it is possible to replace floating point operations with fixed point operations, this would provide a significant increase in speed.

This conversion can either take place automatically or or based on a specific request from the programmer. To do this automatically, the compiler must either be very smart, or play fast and loose with the accuracy and precision of the programmer's variables. To be "smart", the computer must track the ranges of all the floating point variables through the program, and then see if there are any potential candidates for conversion to floating point. This technique is discussed further in section **??**, where it was implemented.

The other way to do this is to rely on specific hints from the programmer that a certain value will only assume a specific range, and that only a specific precision is

[4]This requires that for n consecutive additions, there are $\log_2 n$ high guard bits to prevent overflow. In the μFPU, there are 3 guard bits, making up to 8 consecutive additions possible.

desired. This is somewhat more taxing on the programmer, in that he has to know the ranges that his values will take at declaration time (something normally abstracted away), but it does provide the opportunity for fine-tuning already working code.

Potential applications of this would be simulation programs, where the variable represents some physical quantity; the constraints of the physical system may provide bounds on the range the variable can take.

2.5.2 Small Constant Multiplications

One other class of optimizations that can be done is to replace multiplications by small integer constants into some combination of additions and shifts. Addition and shifting can be significantly faster than multiplication. This is done by using some combination of

$$\begin{aligned} a_i &= a_j + a_k \\ a_i &= 2a_j + a_k \\ a_i &= 4a_j + a_k \\ a_i &= 8a_j + a_k \\ a_i &= a_j - a_k \\ a_i &= a_j \ll m\text{shift} \end{aligned}$$

instead of the multiplication. For example, to multiply s by 10 and store the result in r, you could use:

$$\begin{aligned} r &= 4s + s \\ r &= r + r \end{aligned}$$

Or by 59:

$$t = 2s + s$$

$$r = 2t + s$$
$$r = 8r + t$$

Similar combinations can be found for almost all of the smaller integers[5]. [?]

2.6 Other optimizations

2.6.1 Low-level parallelism

The current trend is towards duplicating hardware at the lowest level to provide parallelism[6]

Conceptually, it is easy to take advantage to low-level parallelism in the instruction stream by simply adding more functional units to the μFPU, widening the instruction word to control them, and then scheduling as many operations to take place at one time as possible.

However, simply adding more functional units can only be done so many times; there is only a limited amount of parallelism directly available in the instruction stream, and without it, much of the extra resources will go to waste. One process used to make more instructions potentially schedulable at any given time is "trace scheduling". This technique originated in the Bulldog compiler for the original VLIW machine, the ELI-512. [?, ?] In trace scheduling, code can be scheduled through many basic blocks at one time, following a single potential "trace" of program execution. In this way, instructions that *might* be executed depending on a conditional branch further down in the instruction stream are scheduled, allowing an increase in the potential parallelism. To account for the cases where the expected branch wasn't taken, correction code is inserted after the branches to undo the effects of any prematurely

[5]This optimization is only an "optimization", of course, when the amount of time spent on the shifts and adds is less than the time that would be spent doing the multiplication. Since the time costs of these operations are known to the compiler in order for it to do scheduling, it is easy for the compiler to determine when this optimization is worth using.

[6]This can been seen in the i860; floating point additions and multiplications can proceed at the same time, and the RISC core be moving data in and out of the floating point registers and providing flow control at the same time the floating point units are active. [?]

executed instructions.

2.6.2 Pipeline optimizations

In addition to having operations going on in parallel across functional units, it is also typical to have several operations in various stages of completion in each unit. This pipelining allows the throughput of the functional units to be increased, with no increase in latency.

There are several ways pipelined operations can be optimized. On the hardware side, support can be added to allow data to be recirculated back into the beginning of the pipeline from the end, saving a trip through the registers. On the software side, the compiler can utilize several tricks to try to fill up as many of the pipeline delay slots as possible, as seendescribed by Gibbons. [**?**]

Bibliography

[Bro11] Jason Brownlee. *Clever Algorithms: Nature-Inspired Programming Recipes*. LuLu, January 2011. http://www.cleveralgorithms.com.

[Cro14] Alfred Crosby. Geckskin - Gecko-Like Adhesives, 2014. University of Massachusetts Amherst, https://geckskin.umass.edu (Zugriff am 20.04.2019).

[Mal17] Vijini Mallawaarachchi. Introduction to Genetic Algorithms, July 2017. https://towardsdatascience.com/introduction-to-genetic-algorithms-e396e98d8bf3 (Zugriff am 04.05.2019).

[McC00] Phillip McClean. Mendel's First Law of Genetics (Law of Segregation), 2000. https://www.ndsu.edu/pubweb/~mcclean/plsc431/mendel/mendel1.htm (Zugriff am 16.05.2019).