Solving Number Series using Genetic Algorithms

Johannes Folger, Simone Schineller, Dominik Seuss

January 25, 2012

Contents

1 Introduction

The task of solving number series is often included in tests that aim to assess humans' intelligence or capabilities. Tested subjects are given a sequence of at least 5 numbers which they have to continue by one or more elements. The underlying idea is the assumption that solving number series is based on inductive reasoning: one has to find a certain rule that explains the seen sequence of numbers and has to apply this rule in order to continue the sequence. [2]

In our project we tried to model the human approach for solving number series by a computer program. While the strategies of humans vary for every individual, computers solve the problem by applying defined algorithms. We modeled the process of solving number series by using a genetic algorithm. The next part introduces genetic algorithms in general and describes our first approach of applying them to the number series problem. Section 3 gives an overview of the number series we wanted to test and documents their complexity. In the empirical study we tested several humans with our number series. Section 4 describes the findings of this study. Section 5 deals with the program's implementation and testing. The last two sections of this report summarize our results and compare the findings of the empirical study with our implementation's peformance.

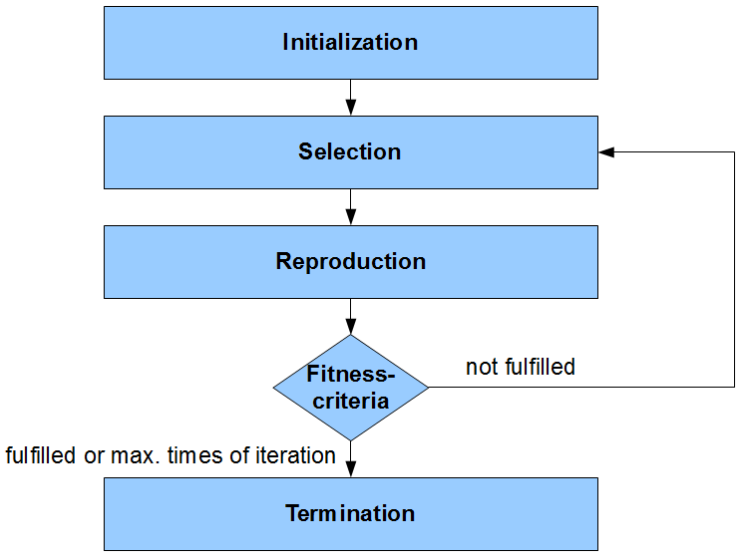


Figure 1: Phases of genetic algorithms

2 Genetic Algorithms

2.1 Genetic algorithms - Overview

Genetic Algorithms are parallel search techniques that use evolutionary principles to repeatedly modify structures by the application of genetic operations. The process starts with several different solutions that are simultaneously optimizied. This reduces the risk of converging to a local minimum. The initally generated solutions are improved by repeated application of selection, crossover and mutation. These steps are illustrated in fig 1.

Initialization In the first phase a great number n of different solutions is (randomly) generated. These solutions form the initial population.

Selection Selection can also be described as " survival of the fittest operator" since in this phase weak items are removed from the population while strong items are duplicated. A fitness function is used to decide whether an item is weak or whether its strong. Binary tournament selection is a widely applied selection method: Two solutions are randomly picked and compared. The fitter one stays in the population whereas the weaker solution is removed. This results in a population with n items as the process is repeated n times. These population is passed to the crossover stage.

Reproduction-Crossover & Mutation In the crossover phase $n/2$ pairs are formed by randomly picking items without replacement. From each pair-called parents- two offsprings are produced by cutting and splicing the parents at randomly selected crossover

points. At mutation stage each component of each item is changed according to a small predefined probability.

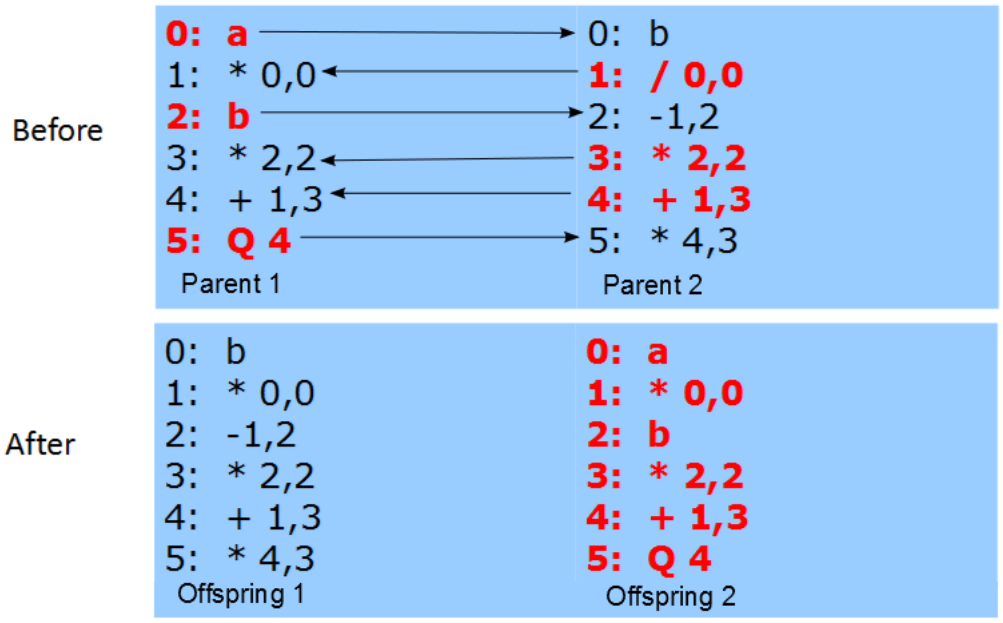
Fitness Assessment and Termination After changing the population by crossover and mutation each soluation is evaluated. If a solution's fitness reaches a predefined threshold or a defined number of iterations is reached, the algorithm terminates. Otherwise the whole process beginning with selection is repeated. [3]

2.2 Genetic Algorithms - Our approach

In the initialization phase the algorithm produces n equations that are possible solutions to the given number series. These equations are represented by Multiple Expression Genes. This kind of represenation eases the combination of equations in the crossover phase. The postfix expression $aa*bb*+Q$ for example, is represented in MEP as follows:

- **0:a** store a in 0
- **1:*0,0** multiply the value in 0 withe the value in 0 and store it to 1
- **2:b** store b in 2
- **3:*2,2** multiply the value in 2 with the value in 2 and store it to 3
- **4:+1,3** add the value of 1 and 3 and store it in 4
- **5:Q 4** take the squareroot of the value in 5 and store it in 5

After selecting the fittest items in the selection phase by the fitness function they are passed to the crossover stage. In the crossover phase the equations are crossbreeded to generate n offsprings. Figure 2.2 shows how two offspring are generated by swapping the lines of their parents. By applying this procedure n offsprings are created and passed to the mutation step. In the mutation phase a randomly chosen number of items is changed by randomly varying their operations. For example addition is replaced by multiplication. A fitness function is applied to determine the performance of these items. If the performance of at least one item reaches a certain threshold the algorithm terminates and the respective item is the soltuion for the given number series. The process also terminates if a certain numer of iterations is reached. Otherwise the process starts again with applying crossover to those items which had sufficient high fitness values.



3 Number Series

3.1 Our number series

We chose the following five number series:

- -2,5,-4,3,-6 $(+7, -9, +7 - 9....)$
- 16,22,34,58,106 $(n_{i+2} = n_{i+1} + 2^i * 6)$
- 1,1,2,3,5 $(f_n = f_{n-1} + f_{n-2}\ with\ n \geq 2,\ f_0 = 0\ and\ f_1 = 1)$
- 2,5,10,17,26 $(+3, +5 + 7 + 9...)$
- 2,3,10,15,26 $n_{i+2} = n_i + (i+1) * 4\ with\ i = 1\ for\ the\ first\ number\ of\ the\ sequence$

We decided to use these number series as each of them is generated using different operations, i.e. alternation, power of, multiplication and addition. [1]

3.2 Complexity of number series

We tried to find a formalism to determine the complexity of a number series. Unfortunately, this turned out to be very difficult as each person estimates the complexity of an operation in a different way. Table 1 shows our attempt to define the complexity of several operations.

Table 1: Complexity of operations

Operation	Complexity
+,-,*,/ a constant factor	1
power of	n
+,-,+,/ a changing factor	n^2
alternation	n^2
applying +,-,*,/ to i predecessors to obtain a successor	$i * n^2$

The formalism we chose is similar to the big O notation. However, there is an important difference with respect to the combination of different complexities. In the big O notation $O(n) + O(n^2)$ would result in $O(n^2)$ as the higher complexity always dominates. This seemed inappropriate for our problem as it is not differentiated enough. For this reason we decided to add up the different complexities for a series. Applying this approach results in these complexities:

- -2,5,-4,3,-6 : $1 + n^2$(alternation and +,-)
- 16,22,34,58,106 $2n^2$(+ a changing factor and power of)
- 1,1,2,3,5: $2 * n^2$ (applying + to 2 predecessors to obtain a successor)
- 2,5,10,17,26 n^2 (+ a changing factor)
- 2,3,10,15,26 $2n^2$ (+ a changing factor and alternation)

4 Empirical Study

We tested our number series with 5 subjects. Two persons were family members of ours. The remaining three were students. Although we encouraged all subjects to think aloud only three of them actually did.

- Subject 1(student)

Sequence	**Approach**
-2,5,-4,3,-6	calculating the differences between two subsequent numbers (solved)
16,22,34,58,106	calculating the differences between two subsequent numbers(solved)
1,1,2,3,	calculating the differences between two subsequent numbers; increasing lookahead; (not solved)
2,5,10,17,26	calculating the differences between two subsequent numbers(solved)
2,3,10,15,26	calculating the differences between two subsequent numbers(not solved)

- Subject 2(student)

Sequence	Approach
-2,5,-4,3,-6	calculating the differences between two subsequent numbers (solved)
16,22,34,58,106	calculating the differences between two subsequent numbers(solved)
1,1,2,3,	calculating the differences between two subsequent numbers; recognizing that this approach is misleading;increasing lookahead; remembering Fibonacci sequence(solved)
2,5,10,17,26	calculating the differences between two subsequent numbers; assuming that the differences are primes; dismissing this thought and replacing primes with odd numbers (solved)
2,3,10,15,26	calculating the differences between two subsequent numbers; assuming perumtation of odd numbers as differences(not solved)

- Subject 3(family member)

Sequence	Approach
-2,5,-4,3,-6	looking at the distance(not difference) between two numbers (not solved)
16,22,34,58,106	interpreting the last digit of each number(not solved)
1,1,2,3,	analysing divisors; idea of primes with exceptions(not solved)
2,5,10,17,26	interpreting the last digit of each number(not solved)
2,3,10,15,26	idea of primes with exceptions;(not solved)

- Subject 4(student)

Sequence	Result
-2,5,-4,3,-6	solved
16,22,34,58,106	solved
1,1,2,3,	not solved
2,5,10,17,26	not solved
2,3,10,15,26	not solved

- Subject 5(family member)

Sequence	Result
-2,5,-4,3,-6	solved
16,22,34,58,106	solved
1,1,2,3,	solved
2,5,10,17,26	not solved
2,3,10,15,26	not solved

5 Implementation of Genetic Algorithms Approach

5.1 System Realization

During our research we found an implementation of Genetic Algorithms for solving number series in C#. Therefore, we did not implement the program ourselves but adapted the implementation and changed it slightly. The original program did not allowed dynamic input via the terminal. The sequences it solved were hardcoded into the program. For this reason we implemented an interface that enables users to type in sequences into the terminal. The original program was implemented by Mike Gold an can be found at `http://www.c-sharpcorner.com/UploadFile/mgold/NumSeriesGenericAlgo08292005094027AM/NumSeriesGenericAlgo.aspx`. (see also References) The implementation consists of four classes and an interface called 'Genome'.

- Class1
- MultiPopulationController
- Population
- EquationGenome

In the following we will briefly describe their operations and functionality. The elements of a population- called item, solution or equation so far- are called "genome" as this corresponds to their name used in the program.

Class1 It's the entry point of the application and provides a console implementation and functions for handling multithreading. On demand you can start more than one population at a time.

MultiPopulationController This class provides the functionality to control a large number of populations and also an implementation for multithreading. It writes status messages to the command line. Examples for status messages are information about what the best genome is at the moment and in which population it resides. It calculates the best gene from all populations and injects the genomes with the highest fitness into all the other populations.

Population This class mainly implements all the genetic functions. It provides the code for mutation and for crossovers (using the genes) and decides which genomes can die and which ones are allowed to reproduce. All these operations are based on the fitness function which is defined in the next class Equation Genome.

EquationGenome The class provides the code for the genomes and their parts (the genes). Each genome realizes our idea of multiple expressions. We chose predecessor, position, addition, substraction, multiplication, division, modulo and as constants 0 and 1 for the genes. It calculates the fitness of a genome via adding the error terms while applying the genome at the given number series.

Table 2: Performance of GA implementation for our number series

Sequence	**Result**
-2,5,-4,3,-6	solved after one run
16,22,34,58,106	solved after one run
1,1,2,3,	solved after two runs
2,5,10,17,26	solved after one run
2,3,10,15,26	not solved

Table 3: Comparison of human and GA performance

sequence	number of persons who solved	performance of GA
-2,5,-4,3,-6	4	solved after one run
16,22,34,58,106	4	solved after one run
1,1,2,3,	2	solved after two runs
2,5,10,17,26	2	solved after one run
2,3,10,15,26	0	not solved

5.2 Testing and results

We tested the programm by giving it the number series presented in section 3.1. The performances for the sequences are listed in table 2. The performance values in table **??** makes it that clear the program had problems with solving alternating number series since it did not solve the last sequence. For all other sequences a solution was found after one run, except for the Fibonacci sequence. For this problem the algorithm needed two runs.

6 A cognitive model for solving number series

With respect to the results in table 3 one could say that the implementation we used can serve as a cognitive model for the humans' behaviour when solving number series. This assumption is based on the fact that algorithm and humans produced similar results. For the last series neither humans nor the algorithm could find a solution. According to the way how the program works we would propose that humans use a strategy that is similar to the algorithm's functionality. Humans find one possible solution, apply and evaluate it. The solution is changed or replaced if it fails to solve the problem. This theory is also supported by the the three subjects whose thoughts we recorded. Genetic algorithms start with many items that are evaluated by a fitness function. They select only the best solutions and crossbreed them to generate better offsprings. The algorithm terminates if a solution with high fitness is found or if a certain number of steps is reached. The termination criterion may also resemble human performance: persons stop solving a number series if they found one solution or if they are not able to find a solution after

a certain number of trials.

7 Discussion

Based on our data we propose the functionality of GAs as a cognitive model for the human strategy to solve number series. This proposal is based on two observations. Firstly, the results of the empirical study are similar to the performance of the program we used. Secondly, the procedure applied by humans resembles the steps a GA has. However, we only tested five number series with five subjects. One should repeat the study with more series and more persons in order to get reliable results.

References

[1] Gold, Mike: *Evolving Numeric Series using Genetic Algorithms in C#*. Not published, 2004 http://www.c-sharpcorner.com/UploadFile/mgold/NumSeriesGenericAlgo08292005094027AM/NumSeriesGenericAlgo.aspx

[2] Korossy, Klaus: Solvability and Uniqueness of Linear-Recursive Number Sequence Tasks. In: *Methods of Psychological Research — Online* 3 (1998), Nr. 1

[3] Mahfoud, Sam ; Mani, Ganesh: Financial forecasting using genetic algorithms. In: *Applied Artificial Intelligence* 10 (1996), Nr. 6, S. 543–566