ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Evolutionary Methods for Sequences

Bachelor's Thesis

Tung Nguyen

nguyetun@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Ard Kastrati
Prof. Dr. Roger Wattenhofer

July 31, 2021

Acknowledgements

I would like to thank my supervisor Ard Kastrati for supervising my thesis and his continuous support throughout the semester. Furthermore, I thank Prof. Roger Wattenhofer and his Distributed Computing Group at ETH Zürich for giving me the opportunity to do this exciting thesis. Lastly, I would like to thank my dear friend Dimitri von Rütte for helping me with process automation by introducing me to shell scripts. This saved me a lot of time during experimentation.

Abstract

Despite the fact that machine learning being a reliable learning and adaptive technology, these methods however fail at seemingly simple problems such as sequence prediction. We created a context-free grammar to build a program that generates polynomials based on this grammar. With a naive polynomial generation algorithm as our starting point, we implemented a genetic algorithm on top that would predict the polynomial that produces a sequence of integers that matches the given sequence. Experimental results show that the evolutionary method clearly outperforms the naive approach and is thus a superior method for sequence prediction.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 1

2 Background **3**

2.1 Context-Free Grammar 3

2.2 Combinator Expressions 4

3 Building the program **5**

3.1 Non-Evolutionary Generation 5

3.1.1 The Library 5

3.1.2 Generate Function 6

3.1.3 Finding the Sequence 9

3.2 Evolutionary Generation 9

3.2.1 Mutation and Crossover 10

4 Theoretical Approach **13**

4.1 Deriving the Probabilities 13

4.2 Computations 14

4.2.1 Length 0 polynomials 15

4.2.2 Length 1 polynomials 15

4.2.3 Length 2 polynomials 18

5 Experiments **23**

5.1 Non-Evolutionary vs Evolutionary 23

5.2 Hyperparameter tuning on short polynomials 27
5.3 Hyperparameter tuning on longer polynomials 28

6 Conclusions 32

Bibliography 33

Chapter 1

Introduction

1.1 Motivation

Consider a sequence of numbers starting with 2, followed by 4 and 6. What is the next number? The answer is not 8 but 14 and the formula to the solution is $x^3 - 6x^2 + 13x - 6$. But if this question is asked to anyone, everyone will answer 8. But why 8?

We know that Machine Learning is one of the technologies that has made the most impact in recent years. However, machine learning models such as the ones used in deep learning require a large amount of data and seemingly simple problems as above are already difficult for them.

Most people would answer that the fourth number followed by 6 is 8, because it is the most simple answer and if you were asked to give the formula to the solution, again most people would say $2x$ due to the fact that $2x$ is short and easy to write. Note that $2x+x-x$ would also be a valid solution, but no one will give such an answer.

Seeing and recognizing simple solutions by removing redundancies or simplifying the problem is an ability where we humans can perform with ease while machine learning models mostly struggle to recognize these patterns.

1.2 Contribution

In this thesis we will implement evolutionary methods to search for such sequences. The focus will only be on simple sequences that can be expressed as polynomials. We start by making observations supported by theoretical justifications to explore ways to search for these algorithms. However, the main focus of this thesis is to find general techniques and observations that can be potentially applied in the future for other types of sequences as well.

The goal of this thesis is to explore the power of evolutionary method as a form of meta learning. As stated above, we restrict ourselves to only work with polynomial integer sequences. We create a context-free grammar and build a program that generates arbitrary polynomials according to the predefined grammar. This program is then extended to improve the time and efficiency of finding the correct polynomial for the given sequence by implementing a genetic algorithm. We will make theoretical analysis and approximation on the runtime and performance and make experiments on various polynomials of different lengths taken from the **OEIS** database (On-Line Encyclopedia of Integer Sequences) [1], which is an online database of integer sequences. Based on the theoretical analysis, we will conduct experiments to compare the two methods and to find interesting behaviour of the evolutionary method.

Chapter 2

Background

In order to generate arbitrary polynomials we have to ensure that these expressions are well-typed (follow the usual mathematical notation) and that they can always be written in the form $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$. Factorized polynomials such as $(x+1) \cdot (x+2)$ are allowed but rational polynomials such as $\frac{2}{x+1}$ are not. For this reason, we define a context-free grammar for polynomials.

2.1 Context-Free Grammar

Recall that a context-free grammar is a formal grammar whose production rules are of the form $A \to \alpha$ where A is a nonterminal symbol and α a string of terminals and nonterminals. Note that α can also be empty.

General mathematical expressions can be expressed as a binary expression tree. Consider this simple expression $6 + 9$. Clearly, the operator $'+'$ requires two arguments and outputs a result of 15. Note that we can also express the number 9 as another expression such as $4 + 5$ and therefore rewrite our original expression as $6 + (4 + 5)$. We see that each argument of a binary operator can also consist of another subexpression. The same rules also apply to the operations $'-'$ and $'\cdot'$. We can recursively define each argument as another expression as long as we want or we can terminate the recursion by setting the arguments to a single variable x or a constant. However, we have to be more cautious with the power operation. In order to maintain the structure of well-defined polynomials, the exponent of a power expression cannot contain a variable. Therefore for the sake of simplicity we restrict the exponents to constants only although expressions such as $x^{(3+2)}$ are technically allowed but note that this can be rewritten as $x^3 \cdot x^2$.

From this observation we can derive the context-free grammar as follows. Let `NT = {Poly}` and `T = {Const, Var}` be the set of nonterminal and terminal symbols respectively. `Const` represents the set of natural numbers and `Var` is a symbol for the variable x.

Here are the production rules for this grammar:

$$\begin{aligned}
\texttt{Poly} &\rightarrow \texttt{(Poly)} + \texttt{(Poly)} \\
\texttt{Poly} &\rightarrow \texttt{(Poly)} - \texttt{(Poly)} \\
\texttt{Poly} &\rightarrow \texttt{(Poly)} \cdot \texttt{(Poly)} \\
\texttt{Poly} &\rightarrow \texttt{Poly}^{\texttt{Const}} \\
\texttt{Poly} &\rightarrow \texttt{Const} \\
\texttt{Poly} &\rightarrow \texttt{Var} \\
\texttt{Const} &\rightarrow \texttt{\{1,...,9\} Const2} \mid \texttt{\{0,...,9\}} \\
\texttt{Const2} &\rightarrow \texttt{\{0,...,9\} Const2} \mid \texttt{\{0,...,9\}}
\end{aligned} \tag{2.1}$$

2.2 Combinator Expressions

The main idea of this project is based on the principles presented in [2]. Briggs and O'Neill exploit the power of a strongly typed functional programming language in order to build a genetic program. Furthermore, they introduced combinator expressions as an alternative form of representation for the program. Programs written in functional languages such as Haskell can be simplified to λ-expressions [3] which correspond to combinator expressions.

Consider the function `Add` in a functional programming language such as Haskell. The syntax of this function is `(Add x y = x + y)`. In words, the `Add` function requires two arguments `x` and `y` and returns the sum of these numbers. As in all programming language, we can assign a type to each expression, namely `(Add:Int->Int->Int)`. However, we can also assign a type to the function `Add 2`, which now only requires one argument. So this function has the type `(Add 2:Int->Int)`. It takes an integer `y` and returns the value `2 + y`. We could also have a complete function of `Add 2 3` which we know has the type `(Add 2 3:Int)`.

Representing expressions in combinator form allows for much easier crossover between two or multiple expressions because we always maintain the correct type for these expressions. Consider this expression `Add (Add 2 3) (Add 6 9)`. We know that `Add 2` and `Add 6` have the same type and so do the integers `2, 3, 6` and `9`. We can therefore allow us to swap subexpressions with another of the same type. As an example we can swap `Add 2` with `Mult 4`, `9` with `Pow 2 3` and `Add` with `Sub`. We then get a new expression
`Sub (Mult 4 3) (Add 6 (Pow 2 3))` of the correct type.

CHAPTER 3

Building the program

3.1 Non-Evolutionary Generation

The idea is to have a basic set of mathematical operations, variable x and constants that the `generate` function takes from and builds arbitrary expressions. Prior to the generation, the length of the generated polynomial is defined. `generate` takes an object from the set of nonterminal and/or terminal symbols called *library* and recursively builds an polynomial with respect to the length. We will explicitly define the length and other helpful parameters in the following sections. The full code of this thesis can be found in the Git repository [4].

3.1.1 The Library

We have learned in the previous chapter that mathematical expressions can be represented as a binary expression tree. Since we are generating polynomials according to a grammar, we need a set of symbols that we define as `non_terminals` and `terminals`. Recall that nonterminal symbols are mathematical symbols that require parameters which can again be defined as a nonterminal or terminal symbols. In our case, the nonterminal symbols are the basic mathematical operations `Add, Sub, Mult` and `Pow`. The terminal symbols are `Var` and `Const`. In summary, the library consists of these two sets `non_terminals = {Add, Sub, Mult, Pow}` and `terminals = {Var, Const}`. Note that these sets remain unchanged throughout the process of generating polynomials. This will be different in the evolutionary generation since we are creating a library where the content of this set changes as we generate more polynomials.

In the actual implementation we used an additional nonterminal symbol `NConst` that allows us to generate multiple digit numbers by concatenating single digit numbers produced by the symbol `Const` with respect to the length of the polynomial.

Example 3.1. For example the number `187` is simply a concatenation of three separate digits generated by three `Const` symbols. In combinator form, we represent `187` as `NConst(1, NConst(8, 7))`.

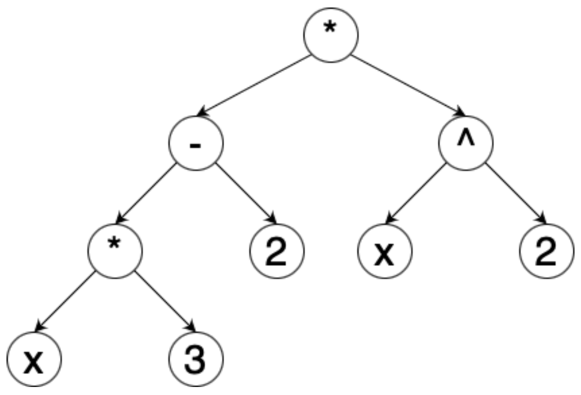

Figure 3.1: Expression tree for the polynomial `(x * 3 - 2) * x^2`

There are two reasons why we choose this way of implementation. Firstly, this allows us to generate any integer of arbitrary length. Secondly, we also want to account the number of digits to the length of the polynomial. The `non_terminals` set therefore contains five symbols. This is an implementation detail but necessary to justify the computations in Chapter 4.

3.1.2 Generate Function

In this section we will describe a naive algorithm to generate arbitrary polynomials. Note that this approach is a brute-force method and is not expected be very efficient.

In principle, we are building an expression tree in combinator form where each mathematical operation, variable and digit represents a node of this tree and nodes are connected if they are an argument of a parent node. Figure 3.1 shows an example of an expression tree that we are creating.

The idea is to create all possible polynomial expressions of a given length with respect to our grammar and then choose one polynomial uniformly at random. However, this will be computationally too expensive. We therefore need another approach that bypasses this problem.

Instead of generating all polynomials, we build only one polynomial by choosing components from the expression randomly. With this approach we avoid creating polynomials that we are not using and

We first will introduce some terminologies that will help understanding the implementation.

Definition 3.2 (Length)**.** The `length` of a polynomial is defined as the number of non-terminal symbols in the combinator representation of the polynomial.

Example 3.3. Consider the polynomial `2 * x + 1`. We can represent it in combinator form as `Add(Mult(2, Xx, 1)`. This polynomial has `length` 2. Multiple digit number such as `619` have `length` 2 because `619` is generated by `NConst(6, NConst(1, 9))`.

Example 3.4. The polynomial `x` has `length` 0, since the only symbol present is `Var`. Polynomials such as `6` or `9` also have `length` 0.

The process of generating an arbitrary polynomial can be roughly described in three steps:

1. Set the `length` by counting consecutive coin tosses landing on head.
2. Choose an item from `non_terminals` uniformly at random if `length` is not 0, otherwise choose from `terminals`.
3. recursively call `generate` for the arguments (if any) of the item chosen in step 2.

This is a high-level outline of the functionality of `generate` but we will discuss further in detail.

In the first step, we set the length by counting consecutive fair coin tosses landing on heads until the coin lands on tails, which we call the *prior* [5]2. This is equivalent to sampling a geometric distribution with probability $p = \frac{1}{2}$. By using a prior ensure that short polynomials are generated more frequently than longer polynomials. The probability of generating a polynomial diminishes exponentially with each increasing length.

We will now refer to taking a symbol from `non_terminals` and `terminals` as *growing* and *not growing* the tree respectively.

Definition 3.5 (Options)**.** In a binary expression tree, `options` of is defined as the total number arguments that do not have any item from `non_terminals` or `terminals` assigned to. In an empty tree, we start with `options = 1`.

We start growing our tree by picking an item from `non_terminals` uniformly at random if we have some length left and then we decrease the `length` parameter by 1. Otherwise the item is chosen from `terminals`. We then keep track of the number of options available and proceed growing in a preorder traversal order. As long as the `length` is not 0, we continuously grow the tree. If `length` reaches 0, we stop growing and only pick symbols from `terminals` when traversing through the remaining leaves. Finally, `generate` returns the finished polynomial.

However, there is a problem with this algorithm. Since we are traversing the tree in a preorder manner, we always choose to grow at the left argument of the node, resulting in an unbalanced tree. We want that each leaf is chosen to be grown with a uniform probability. The solution is, that each time we visit a leaf we choose to grow with probability $\frac{1}{\texttt{options}}$ and not to grow with probability $1 - \frac{1}{\texttt{options}}$.

Lemma 3.6. *By choosing each node to grow with probability* $\frac{1}{\texttt{options}}$ *we ensure that all nodes have equal chance to grow.*

Proof. Consider an arbitrary state of the function where `options = n`. Therefore we have n possible places to grow the tree, assuming we have some length left, i.e. `length > 0`.

We choose to grow at the node that we are visiting next with probability $\frac{1}{n}$, call it $node_n$. Assume that we do not choose to grow at this node, which happens with probability $1 - \frac{1}{n}$. `options` is decremented and we traverse to the next node $node_{n-1}$. Locally, we again choose to grow with probability $\frac{1}{n-1}$ and conversely not to grow with probability $1 - \frac{1}{n-1}$. However, in order to grow at $node_{n-1}$ we must have decided not to grow at $node_n$ prior. Let E_i be the events that we choose to grow at $node_i$ with $1 \leq i \leq n$. We know that $Pr[E_n] = \frac{1}{n}$. For all $i < n$ the probabilities E_i can be defined recursively as

$$\begin{aligned}
Pr[E_i] &= (1 - Pr[E_n]) \cdot (1 - Pr[E_{n-1}]) \cdot \ldots \cdot 1 - Pr[E_{i+1}] \cdot \frac{1}{i} \\
&= \prod_{k=i+1}^{n} (1 - Pr[E_k]) \cdot \frac{1}{i} \\
&= \frac{n-1}{n} \cdot \frac{n-2}{n-1} \cdot \ldots \cdot \frac{(i+1)-1}{i+1} \cdot \frac{1}{i} \\
&= \frac{1}{n}
\end{aligned}$$

We have shown that for any i the event E_i occurs with equal probability among n nodes, which is $\frac{1}{n}$. This concludes our proof. □

Here is the pseudocode for `generate`:

Algorithm 1: `generate(length, options)`

```
1  # Initialized as length=generate_coinflips, options=1
2  $coin$ ← throw biased coin with probability $\frac{1}{\texttt{options}}$ of landing heads
3  if $coin = heads$ OR length $\neq 0$ then
4      $item$ ← random element from non_terminals
5      length ← length − 1
6      options ← add the number of arguments to options − 1
7      $item.arg1$ ← generate(length, options)
8      $item.arg2$ ← generate(length, options)
9  else
10     $item$ ← random element from terminals
11     options ← options − 1
12 return $item$;
```

We will now show the process of generating a polynomial with an example.

Example 3.7. Let `length = 2`. Since no item has been chosen, we randomly pick an element from `non_terminals` say `Add` and we decrement `length` by 1. For the left argument

Algorithm 2: `generate_coinflips()`

```
1 coin ← make a coin flip
2 return 1+ generate_coinflips() if coin lands on heads else 0
```

of `Add`, we throw a biased coin with probability $\frac{1}{\texttt{options}}$ of landing heads. For this coin, the probability is $\frac{1}{2}$ since we have two empty arguments and thus `options = 2`. The coin lands on heads, so we choose another item randomly from `non_terminals`. The picked item is `Mult` and again the `length` is decremented. Since `length` is now 0, we are forced to always choose from `terminals` for all undefined arguments. The polynomial at the moment is defined as `Add(Mult(None, None), None)` where `None` represents empty arguments. We replace all empty arguments with a random item from `terminals`. We end up with a polynomial like `Add(Mult(x, 6), 9)` or `x * 6 + 9`

3.1.3 Finding the Sequence

Given a sequence of numbers, we can let the program try to find the polynomial that generates the exact sequence. This procedure can be described as follows:

1. We call `generate` which produces a random polynomial.
2. Check whether the sequence produced by this polynomial matches the given sequence.
3. Repeat the first step until the solution is found.

During this process we will keep track on how many trials `generate` needs until the exact solution is found to assess the performance of this algorithm.

Algorithm 3: find(sequence)

```
1 while solution is not found do
2     length ← generate_coinflips()
3     poly ← generate(length, options=1)
4     compare sequence generated by poly with the given sequence
5     if both sequence match then
6         solution found
7 end while
8 return poly
```

3.2 Evolutionary Generation

Now that we have the foundation of generating polynomials, we can extend the function `generate` to not only produce random polynomials but to learn from these and navigate towards the solution.

The basic idea of evolutionary generation is a genetic algorithm. We create a population of evolving *organisms* where each organism represents a possible solution and in our case, the organisms are polynomials. Among these possible solutions we select the *fittest* ones and mix some of these to create a new generation of population, in hope that the next generation is *fitter* and therefore closer to the actual solution.

We will use the parameters `population_size, tournament_size, mutation_rate` and `crossover_rate`. The genetic algorithm works as follows:

1. Generate polynomials and add them to the set *population* the set reaches the size `population_size` and evaluate the fitness of each polynomial in this set. We used *mean squared error* (MSE). If any of these organisms is a correct solution, we stop.
2. Take the `tournament_size` fittest polynomials from the population and store them in the tournament library `tournament_set`.
3. Create the next generation of population using the polynomials in `tournament_set` by applying crossover and mutation to the organisms and evaluate its fitness. Again, if any of these polynomials is a correct solution, we stop with the procedure.
4. Repeat from step 2 until the solution is found.

Note that since MSE is used as our fitness function, a correct solution will satisfy

$$\sum_i (y_{true}^{(i)} - y_{predict}^{(i)})^2 = 0$$

3.2.1 Mutation and Crossover

We will first provide a possible solution and then explain our approach on how we solve the problem that arise in the former method.

Step 3 of the algorithm requires the program to select polynomials among the ones in `tournament_set` and essentially form *offspring* [6]. In order to crossover we somehow need to be able to extract some parts of an already generated polynomial to combine the parts with another polynomial. A possible approach is to generate all possible subexpressions of the generated polynomial and store them in the tournament set and simply choose one subexpression uniformly at random when we crossover. However, we do not know the length of each subexpression in the tournament set and we do not want combine polynomials that will overshoot the desired length or even create very big polynomials. We therefore need a way to only produce polynomials of a given length despite allowing crossovers.

If a polynomial has k nodes in the tree then we will cut this polynomial into k subexpressions, where in each subexpression the root node of the subtree is node of the original tree. These subexpressions are then stored in `tournament_nt` and `tournament_t` according to the type of the root node of the polynomial.

Example 3.8. Consider the polynomial `Mult(x, Add(4, 2))`. This polynomial is then cut into the subexpressions `Mult(x, Add(4, 2))`, `Add(4, 2)`, `x`, `4` and `2`.

Now we have created a new library of already generated polynomials, divided into terminals and non-terminals. We can reuse Algorithm 1 with slight modifications. Every now and then, decided by `crossover_rate`, instead of only choosing an item from `non_terminals` or `terminals` we allow picking an item from the tournament sets `tournament_nt` and `tournament_t` respectively. However, we do not add the complete new subtree but only its root and 'ignore' the child nodes. If we recursively call `generate` for an argument that is already defined we now have the options to either keep this node or to mutate it. Mutating a node simply means to swap the current node with another node taken from the regular library uniformly at random according to the type (terminal or non-terminal). This also happens every now and then, decided by `mutation_rate`.

With this method we can simply stop growing if the desired length is achieved and crossover will not overshoot the length.

Here is the pseudocode of the evolutionary method:

Algorithm 4: `generate_evolutionary(length, options)`

```
# Initialized as length=generate_coinflips, options=1
coin ← throw biased coin with probability $\frac{1}{\texttt{options}}$ of landing heads
if coin = heads OR length ≠ 0 then
    if we decide to crossover then
        item ← random element from tournament_nt
        if we decide to mutate then
            change node with another of same type (terminal or non-terminal)
    else
        item ← random element from non_terminals
    length ← length − 1
    options ← add the number of arguments to options − 1
    item.arg1 ← generate(length, options)
    item.arg2 ← generate(length, options)
else
    if we decide to crossover then
        item ← random element from tournament_t
        if we decide to mutate then
            change node with another of same type (terminal or non-terminal)
    else
        item ← random element from terminals
    options ← options − 1
return item;
```

Algorithm 5: find_evolutionary(sequence, `population_size`, `tournament_size`, `mutation_rate`, `crossover_rate`)

```
1  while solution is not found do
2      for 1 to population_size do
3          length ← generate_coinflips()
4          poly ← generate_evolutionary(length, options=1)
5          compare sequence generated by poly with the given sequence
6          if both sequence match then
7              solution found
8              return poly
9          Compute fitness (MSE) of poly and insert it to population
10     end for
11     Update tournament_set by choosing tournament_size fittest polynomials from population
12 end while
```

Chapter 4

Theoretical Approach

In this chapter we will set the focus on analyzing the behaviour of non-evolutionary generation. We try to understand how close the generated polynomial is to the exact solution regarding the representation and whether we can derive and approximate the average number of trials needed until `generate` returns the exact solution. The procedure of computing the theoretical approximation of the trials needed is explained and important findings and observations are formulated and justified with mathematical computations.

We consider a polynomial to be found if the object returned by `generate` produces the same sequence as the exact solution. Note that the representation does not have to be congruent as $2x$ and $2x + x - x$ both produce the same sequence of numbers. In order to be able to estimate the number of trials needed for the given solution, we need to know the probability such that `generate` picks the right operations and symbols in the right order. Therefore computing the theoretical expected value requires knowing with what probability the searched object occurs.

Because `generate` randomly picks any arbitrary polynomial we can consider the process of finding the exact solution as a geometric random variable with probability p. In order to obtain the expected number of trials needed, we can simply exploit the property of a geometric random variable and compute $\frac{1}{p}$. We will now show how p can be derived and approximated for any polynomial.

4.1 Deriving the Probabilities

The probability of a given polynomial to be chosen is

$$p(\texttt{Poly}) = p(\texttt{length} = len(\texttt{Poly})) \cdot p(NT) \cdot p(T).$$

$p(NT)$ and $p(T)$ are the probabilities that the symbols from `non_terminals` and `terminals` contained in `Poly` are being chosen.

We first derive the probability $p(\texttt{length} = len(\texttt{Poly}))$. Given a polynomial, we can count the `length` of this expression by listing all `non_terminals` items. If we know how many `non_terminals` items are required, we know that `generate` must achieve this number of consecutive coin tosses landing on heads. Again, we can consider the consecutive coin tosses landing on heads as a geometric random variable. In order to achieve k coins landing on heads we must succeed k times and fail on the $k+1$-th try. Therefore, the probability such that `length = k` is $(1-\frac{1}{2})^k \cdot \frac{1}{2} = \frac{1}{2^{k+1}}$

To derive $p(NT)$ and $p(T)$ we will consider the Algorithm 1. Essentially, with slight abuse of notation we can say that

$$p(NT) = \prod_{item_{nt} \in \texttt{Poly}} p(item_{nt} | \texttt{length, options}).$$

To put it in words, $p(NT)$ is the product of the probabilities that the occurring non-terminal items $item_{nt}$ in `Poly` given the `length` and `options` parameters at a specific step of the algorithm. $p(T)$ is defined analogously.

In lines 4 and 10 we pick a random element from `non_terminals` and `terminals` respectively. The assignment in line 4 happens only if the coin thrown in line 2 lands on heads. This occurs with probability $\frac{1}{\texttt{options}}$. Subsequently, each item from `non_terminals` is chosen with probability $\frac{1}{5}$.

Analogously, the assignment in line 10 happens if the coin lands on tails which happens with probability $1 - \frac{1}{\texttt{options}}$. Again, each item from `terminals` is chosen with probability $\frac{1}{2}$.

Finally, the final probability can be obtained by multiplying the three separate probabilities.

Example 4.1. Consider the polynomial `(x + 1) + 2`. In combinator form, this polynomial can be expressed as `Add(Add(x, 1), 2)`. We see that this polynomial contains 2 items from `non_terminals`. So `length` must be 2 and is achieved by two successive coin tosses landing on heads and a subsequent failure, which happens with probability $\frac{1}{2^2} \cdot \frac{1}{2} = \frac{1}{8}$.

The outer most `Add` is chosen with probability $\frac{1}{5}$. This increases our `options` to 2 and decreases the `length` to 1. The inner `Add` of `x` and `1` is chosen with probability $\frac{1}{2} \cdot \frac{1}{5}$ and again this increases `options` to 3 and decreases `length` to 0. Because `length` is now 0 we always choose from `terminals` from this point and omit the coin tosses. Finally, `x` is chosen with probability $\frac{1}{2}$ and `1` and `2` are chosen with probability $\frac{1}{2} \cdot \frac{1}{10}$ each. To obtain the total probability of occurrence, we multiply everything together which results in

$$\frac{1}{8} \cdot \frac{1}{5} \cdot \frac{1}{2 \cdot 5} \cdot \frac{1}{2} \cdot \frac{1}{2 \cdot 10} \cdot \frac{1}{2 \cdot 10} = \frac{1}{320000}.$$

On average, we would need to do 320000 trials to find this polynomial.

4.2 Computations

In this section we will try to estimate the probability and the expected number of trials for some simple polynomials of different length and state important observations during the computations. For the following subsections we will fix the variable `length` and therefore omit the probability of the consecutive coin tosses to achieve the desired length unless we explicitly state the opposite. Note that any polynomial has infinitely many other polynomials that are equivalent. We fix the length in order to make the analysis easier for some polynomials. Since there are infinitely many solutions, we should expect that the theoretical expected number of trials needed will be greater or equal to the true expected value. The deviation however will not be large since we are considering cases with large prior. Polynomials of higher length have much lower prior which will significantly lower the overall probability. From the theoretical result we can have an idea on how long we should wait until the program finds a solution.

4.2.1 Length 0 polynomials

There are only two types of length 0 polynomials, namely `x` and numbers from `0` to `9`. We first consider the latter type. In order to generate a single digit number, we must choose the item `Const` when choosing uniformly at random from `terminals`. Note that because `length` is set to 0, `generate` is forced to take an item from `terminals` and is not required to toss a biased coin. Secondly, `Const` must then generate a digit from 0 to 9. Thus, each number has a probability of $\frac{1}{2} \cdot \frac{1}{10} = \frac{1}{20}$ and the expected number of trials needed until we find a specific single digit number is 20.

For `x`, `generate` must only choose the item `Var` from `terminals`, so `x` has a probability of $\frac{1}{2}$. Again, we need 2 trials on average to obtain the polynomial `x`.

4.2.2 Length 1 polynomials

We will make a detailed analysis on the single digit number polynomials `0,...,9, x` and some various simple polynomials of length 1. The expected number of trials can still be computed with high accuracy to the ground truth because we can consider all possible forms of representation of the sought polynomial. This however will be more difficult if we do not fix the `length` variable or consider polynomials of higher lengths. If we limit the length of the generated polynomials to 1 we allow ourselves to cover all possible forms because there are infinitely many solutions.

Consider again the polynomial `0`. We count these following forms of this polynomial:

$$
\begin{aligned}
\texttt{x - x:}\quad \texttt{Sub(x, x)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{2} = \mathbf{\frac{1}{20}}\\
\texttt{a - a:}\quad \texttt{Sub(a, a)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{10} = \mathbf{\frac{1}{200}}\\
\texttt{\_ * 0:}\quad \texttt{Mult(\_, 0)} &\Rightarrow \frac{1}{5}\cdot 1\cdot\frac{1}{2}\cdot\frac{1}{10} = \mathbf{\frac{1}{100}}\\
\texttt{0 * \_:}\quad \texttt{Mult(0, \_)} &\Rightarrow \frac{1}{5}\cdot 1\cdot\frac{9}{10}\cdot\frac{1}{2}\cdot\frac{1}{10} = \mathbf{\frac{9}{1000}}\\
\texttt{0\^{}a:}\quad \texttt{Pow(0, a)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{10}\cdot\frac{1}{9} = \mathbf{\frac{9}{1000}}\\
\texttt{0 + 0:}\quad \texttt{Add(0, 0)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{10}\cdot\frac{1}{2}\cdot\frac{1}{10} = \mathbf{\frac{1}{2000}}
\end{aligned}
$$

where `a` represents a number from `0,...,9` and `_` can be any terminal item. The probability of each case is computed by using the derivation from section 4.1. In order to obtain the total probability, all probabilities of each cases are added. We therefore get

$$\frac{1}{20}+\frac{1}{200}+\frac{1}{100}+\frac{9}{1000}+\frac{9}{1000}+\frac{1}{2000}=\frac{167}{2000}$$

Using this result, the expected number of trials is $\mathbf{\frac{2000}{167}} \approx \mathbf{11.976}$

We will also show the analysis for the polynomial `1`. Again, we look at these following cases:

$$
\begin{aligned}
\texttt{1\^{}a:}\quad \texttt{Pow(1, a)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{10}\cdot = \mathbf{\frac{1}{100}}\\
\texttt{a\^{}0:}\quad \texttt{Pow(a, 0)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{10}\cdot = \mathbf{\frac{1}{100}}\\
\texttt{x\^{}0:}\quad \texttt{Pow(x, 0)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{10}\cdot = \mathbf{\frac{1}{100}}\\
\texttt{a - b = 1, a > b:}\quad \texttt{Sub(a, b)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{10}\cdot\frac{1}{2}\cdot\frac{1}{10}\cdot 9 = \mathbf{\frac{9}{2000}}\\
\texttt{1 + 0:}\quad \texttt{Add(1, 0)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{10}\cdot\frac{1}{2}\cdot\frac{1}{10} = \mathbf{\frac{1}{2000}}\\
\texttt{0 + 1:}\quad \texttt{Add(0, 1)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{10}\cdot\frac{1}{2}\cdot\frac{1}{10} = \mathbf{\frac{1}{2000}}\\
\texttt{1 * 1:}\quad \texttt{Mult(1, 1)} &\Rightarrow \frac{1}{5}\cdot\frac{1}{2}\cdot\frac{1}{10}\cdot\frac{1}{2}\cdot\frac{1}{10} = \mathbf{\frac{1}{2000}}
\end{aligned}
$$

The expected number of trials is therefore $\mathbf{\frac{250}{9}} \approx \mathbf{27.778}$

This analysis can be done verbatim for any single digit number polynomial. However, we will show a table where the theoretical result is compared with the actual result produced by the program. Note that in order to obtain a reasonable average due to the large variance, we ran the program 1000 times (*rounds*) to compute the average.

Polynomial	Mean (theor.)	Mean (empirical)	Standard deviation
0	11.976	11.877	11.701
1	27.778	27.683	28.04
2	133.33	134.176	135.217

For the sake of completeness, we provide a table with the mean and standard deviation of the polynomials `3,...,9` of length 1. However, we will not show the calculation for each polynomial as it can be done verbatim.

Polynomial	Mean (theor.)	Mean(empirical)	Standard deviation
3	131.386	134.489	142.859
4	108.433	113.854	112.837
5	131.386	130.414	133.865
6	116.129	117.194	116.96
7	131.386	141.114	144.306
8	102.857	105.077	104.531
9	108.433	106.134	108.78

From the second table we observe that polynomials such as `5` or `7` have a higher average number of trials than `4` or `8`. The difference in expected number is reasonable as some numbers such as `4` have more divisors, which increases the chance of finding this object whereas prime numbers like `5` or `7` have less divisors.

Lastly, we provide a table with polynomials where the theoretical result is again compared to the result from the program. We will omit the derivation.

Polynomial	Mean (theor.)	Mean (empirical)	Standard deviation
x	27.692	27.579	28.423
x + 7	100	107.332	101.62702
9x	100	101.144	100.2327
x^7	100	100.577	101.4778

We see that the average closely matches the standard deviation. From this observation we can conclude that the probability p of finding this polynomial is very small. Because the process of finding a polynomial is equivalent to a geometric distribution, the standard deviation is $\sqrt{\frac{1-p}{p^2}}$. For small p we have $\sqrt{\frac{1-p}{p^2}} \approx \frac{1}{p}$, i.e. the standard deviation is almost equal to the mean.

4.2.3 Length 2 polynomials

We will step into a territory where the number of representations possible for a polynomial becomes very large and thus, we cannot cover all cases. However, we can limit ourselves to only consider forms that may occur frequent because they are easier to find. In this subsection, we let `length` to be variable and generated with the aforementioned method.

We will provide the analysis for the `0, 2 * x + 5` and `x**5 - x**2`.

For the generation of numbers, a verbatim analysis as for length 1 can be performed. With this analysis we will show how good of an approximation we can achieve.

The zero polynomial `0` of length 2 can occur in one of these forms:

```
      0 * (something)
      (something) * 0
        0 +- (zero)
        (zero) +- 0
         (zero)^a
   (zero) * (something)
   (something) * (zero)
```

where `a` $\in$ `{0,...,9}`, `zero` is a placeholder for an expression that evaluates to 0 such as `x - x` or `0^8` and `something` can be an arbitrary expression.

We just need to know the probabilities of `something` and `zero`. Since `something` can be any expression, the probability of this occurrence is 1. For `zero`, we can use the results from the analysis in subsection 4.2.2. Therefore, the probability such that an expression of length 1 evaluates to 0 is $\frac{167}{2000}$.

We can simply compute the probabilities for each case and sum them up to obtain the total probability. To summarize, we have

$$\begin{aligned}
\texttt{0 * (something)} &\Longrightarrow \mathbf{\frac{1}{100}} \\
\texttt{(something) * 0} &\Longrightarrow \mathbf{\frac{1}{100}} \\
\texttt{0 +- (zero)} &\Longrightarrow \mathbf{2} * \mathbf{\frac{159}{2000}} \\
\texttt{(zero) +- 0} &\Longrightarrow \mathbf{2} * \mathbf{\frac{159}{2000}} \\
\texttt{(zero)\^{}a} &\Longrightarrow \mathbf{\frac{1431}{200000}} \\
\texttt{(zero) * (something)} &\Longrightarrow \mathbf{\frac{159}{20000}} \\
\texttt{(something) * (zero)} &\Longrightarrow \mathbf{\frac{159}{20000}}
\end{aligned}$$

Again, we obtain the expected number of trials when the take the reciprocal value of the total probabilility and we get 22.399. Here is the comparison of theoretical computation with the programs result:

Mean (theor.)	Mean (empirical)	Standard deviation
22.399	21.806	21.975

As we see, it is a very decent approximation although we left out some edge cases.

Figure 4.1, 4.2 and 4.3 show the behaviour of finding the polynomials `0, 1` and `x` by fixing the `length` variable from 0 to 50. The x-axis shows the length and the y-axis the average number of trials. All runs were done with 1000 rounds.

It is interesting to see that there are more ways to represent `0` with length 1 than with length 0. Furthermore, for all graphs there exist a threshold where the result does not get worse by increasing the length. However, we are not able to justify this behaviour. Note that this graph shows the average number of trials without the prior. The prior will regularize the values and the graph would quickly converge to zero.

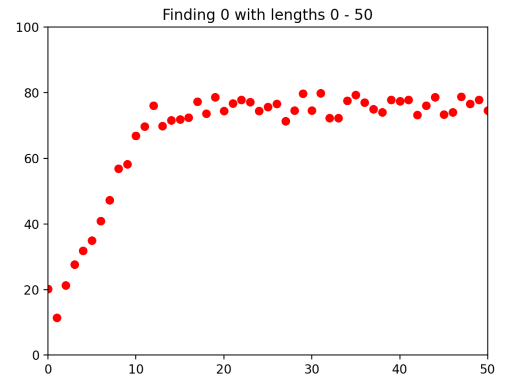

Figure 4.1:

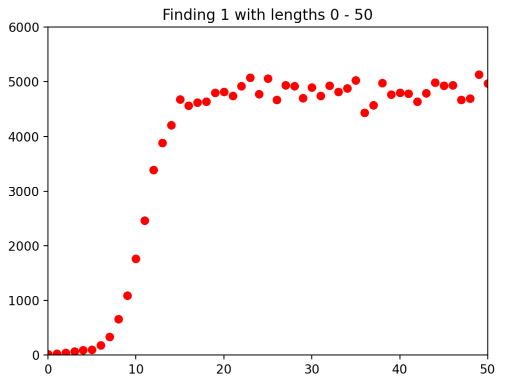

Figure 4.2:

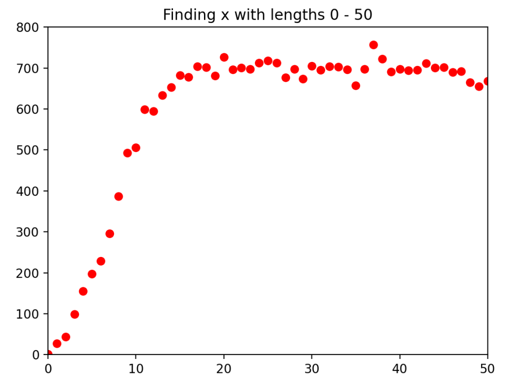

Figure 4.3:

We will now perform the analysis for the polynomial `2 * x + 5` which has a length of 2. Here we note possible forms of this polynomial:

```
  2 * x + 5
  x * 2 + 5
  5 + x * 2
  5 + 2 * x
 (x + x) + 5
 x + (x + 5)
 (x + 5) + x
 x + (5 + x)
 (5 + x) + x
 5 + (x + x)
```

Note that some forms may look the the same but are different objects due to associativity. With the derived probabilities, we can simply compute the probability of occurrence for each form. It is easy to verify that the first four forms $p($`2 * x + 5`$), \dots, p($`5 + 2 * x`$)$ have a $\frac{1}{160000}$ chance each and the last six forms $p($`(x + x) + 5`$), \dots, p($`5 + (x + x)`$)$ a chance of $\frac{1}{16000}$. In total, we get a probability of $4 \cdot \frac{1}{320000} \cdot 6 \cdot \frac{1}{32000} = \frac{1}{5000}$. Although `2 * x + 5` and `x + x + 5` have the same length, the latter form has a higher chance of being found since generating a specific number reduces the overall probability by a factor of 10. We note that the algorithm tends to find polynomials with less numbers more frequently

Again, we will compare the theoretical mean with the results from the program:

Mean (theor.)	Mean (empirical)	Standard deviation
5000	3997.465	3924.3487

We see that the true expected number of trials lies at around 4000. There is a large deviation in the numbers since we have not covered all possible cases but only the most significant ones. Nonetheless, this is still a very close approximation and gives a good lower bound.

Finally, we will look at the polynomial `x**5 - x**2`. First, notice that this polynomial in this exact form is not very likely to be found since it has length 2 and two numbers occur in this form. This would add an additional factor of $\frac{1}{2}$ for the length and $\frac{1}{2^2} \cdot \frac{1}{10^2}$ for the two numbers. We could try to reformulate this expression. With simple algebra we can reformulate `x**5 - x**2` to `x * (x**4 - x)`, `(x**4 - x) * x` and `x**5 - (x * x)`. We see that we only need to generate one number in each form. In fact, if we run the program we see that these two forms occur the most frequently. We will now see how good the approximation is by only considering these two forms.

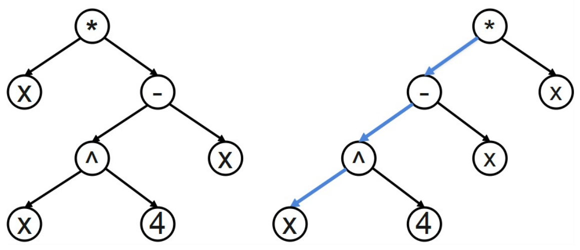

Figure 4.4: Expression tree `x * (x**4 - x)` and `(x**4 - x) * x`

We first consider the first two forms. Note that these two forms are their respective commutative counterpart. However, computations show that the probabilities of occurrence do not match, i.e. $p($`x * (x**4 - x)`$) \neq p($`(x**4 - x) * x`$)$. In fact, we have $p($`x * (x**4 - x)`$) = \frac{1}{640000}$ and $p($`(x**4 - x) * x`$) = \frac{1}{960000}$. Note that the expression tree for `(x**4 - x) * x` is 'heavy' in the left subtree, meaning that the tree mostly grows from the left argument. The combinator form of `(x**4 - x) * x` is `Mult(Sub(Pow(x, 4), x), x)` whereas the combinator form of `x * (x**4 - x)` is `Mult(x, Sub(Pow(x, 4), x))`. The reason why this form has a lower probability is because this expression tree has more consecutive `non_terminals` nodes than its commutative counterpart. This leads to multiplying the factors $\frac{1}{2}, \frac{1}{3}, \frac{1}{4}$ and so forth because program tries to avoid this imbalance by throwing a biased coin each time we grow the tree, which is the procedure explained and proven in Lemma 3.6. Figure 4.4 shows the difference in the tree structure. The blue arrows this chain of `non_terminals`.

Lastly, we can compute the probability $p($`x**5 - (x * x)`$)$ which is $\frac{1}{960000}$. By adding the probabilities together and computing the mean using $\frac{1}{p}$ we obtain

$$\left(\frac{1}{640000} + \frac{2}{960000}\right)^{-1} \approx 274'285.714.$$

Here is a table where we compare the theoretical result with the empirical:

Mean (theor.)	Mean (empirical)	Standard deviation
274'285.714	219'107.145	231'596.939

Despite only considering very few cases, we see that the theoretical mean is a very close approximation to the empirical mean. This is because of the prior. Longer polynomials have an exponentially small probability with each increasing length.

Chapter 5

Experiments

In this section we examine how well our evolutionary method for sequences works by presenting results from three experiments. The first and most extensive experiment compares the performance of non-evolutionary versus evolutionary method on 18 polynomials. This experiment will show whether the non-evolutionary method will outperform the naive algorithm and where the limitations of each method are.

The other experiments are on hyperparameter tuning of the evolutionary method on various polynomials. This part mainly shows the speedup achievable by optimal parameter choice.

All experiments were done on the *Euler* Cluster of ETH Zürich [7].

5.1 Non-Evolutionary vs Evolutionary

Let us begin with the comparison of the two methods. We know that the naive approach randomly samples polynomials from the whole search space until the right solution is found. Since this is equivalent to a geometric distribution, the chances of finding a solution diminishes exponentially by increasing the length of the solution. The evolutionary method tries to combat this problem by making the geometric sampling less 'brainless'. For the runs of the evolutionary method, we used fixed parameters with the values `population_size=1000` `tournament_size=5, mutation_rate=0.1` and `crossover_rate=0.3`. Tables 5.1 and 5.2 show the side by side comparison of the performance on 18 polynomials. Polynomials marked in bold font are taken from the **OEIS** database. For all runs we set an upper bound for the rounds of 500 but we will write how many rounds a run has finished within 24h. Clearly, the more rounds a run has completed the more accurate the results will be.

In Table 5.1 we see that there are many missing entries. This is because the naive algorithm failed to even complete one round of finding the solution within 24 hours. The measured time to generate the polynomial and evaluate the sequence that this polynomial produces with given sequence takes about 0.1 seconds. So even if 10^7 does not seem large for computers we still need to wait 277 hours on average. However, we did not try to implement the program more efficient since this would go beyond the scope of this thesis.

Table 5.1: Performance of Non-Evolutionary Method

Polynomial	**Length**	**Rounds**	**Mean (theor.)**[1]	**Mean (emp.)**	**Std deviation**	**Runtime (h)**[2]
2 * x + 5	2	500	4’705.882	4’052.994	4’083.584	0.249
x**5 - x**2	3	500	144’000	219’870.258	239’891.549	13.143
(x + 1)(x + 2)(x + 3)	5	—	3.07e+10	—	—	—
x**3 - 2*x + 5	4	19	5.9e+07	10M	8.8M	—
(x + 1)**50	3	78	3.8M	2.5M	2.6M	—
69 * (x + 12)	4	—	1.3e+09	—	—	—
42069	4	—	1.8e+09	—	—	—
(x + 6)**9 * (x - 6)**9	5	—	6.99e+09	—	—	—
x**10 - 1	3	500	8.6M	1.6M	1.5M	22.634
(3 * x + 5) * (2 * x - 6)	5	—	3.8e+11	—	—	—
6 * (x - 2)**5	3	5	4e+08	13M	10M	—
x2 + 1**	2	250	14’545	10’408.336	9’882.808	0.316
x2 - x + 1**	3	250	314’181	109’855	114’915	3.264
2 * x * (2 * x - 1)	4	163	3.1M	1.2M	1.3M	—
x * (3 * x - 2)	3	150	3.1M	798’278	917’123	14.420
x + x(x - 1)(x - 2)(x - 3)	7	—	3.2e+11	—	—	—
(3 * x + 1)(3 * x + 2)	5	—	4.8e+08	—	—	—
6 * x * (x - 1) + 1	4	—	1.3e+09	—	—	—

[1] Lower bound if program does not terminate in reasonable time
[2] No entry indicates that the run was killed after 24h

Table 5.2: Performance of Evolutionary Method

Polynomial	**Length**	**Rounds**	**Mean (empirical)**	**Std deviation**	**Runtime (h)**[1]
2 * x + 5	2	500	1860.392	1’510.417	0.097
x**5 - x**2	3	500	12’227.15	14’485.1	0.734
(x + 1)(x + 2)(x + 3)	5	500	62’253.216	194’144.732	3.961
x**3 - 2*x + 5	4	500	11613.614	7’423.291	0.751
(x + 1)**50	3	60	1.8M	2M	—
69 * (x + 12)	4	250	513’863.805	809’991.307	—
42069	4	500	103’475.2	142’460.178	3.541
(x + 6)**9 * (x - 6)**9	5	—	—	—	—
x**10 - 1	3	500	12’823.614	11’236.189	0.705
(3 * x + 5) * (2 * x - 6)	5	100	1’006’117.12	2’683’920	14.442
6 * (x - 2)**5	3	—	—	—	—
x2 + 1**	2	250	2’276.788	1’558.962	0.096
x2 - x + 1**	3	250	5’211.724	3’943.510	0.206
2 * x * (2 * x - 1)	4	250	6’605.176	6’015.75	0.194
x * (3 * x - 2)	3	250	8’722.952	9’939.225	0.274
x + x(x - 1)(x - 2)(x - 3)	7	100	697’872.140	1’622’513	11.676
(3 * x + 1) * (3 * x + 2)	5	100	24’723.460	18’201.35	0.327
6 * x * (x - 1) + 1	4	250	34’695.596	29’490.43	1.052

[1] No entry indicates that the run was killed after 24h

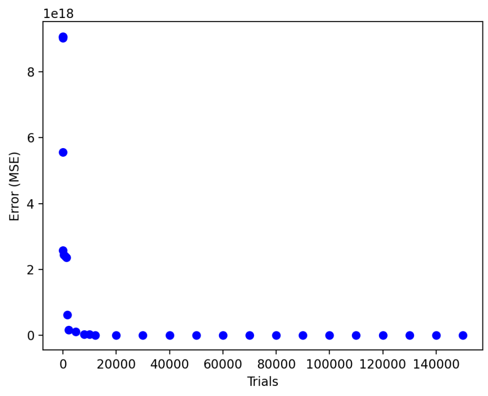

Figure 5.1: Convergence of the polynomial `6 * (x - 2)**5` with non-evolutionary method

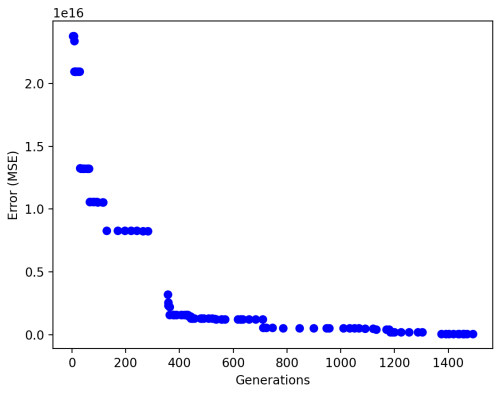

Figure 5.2: Convergence of the polynomial `6 * (x - 2)**5` with evolutionary method

Note that the naive algorithm found no solution for polynomials of length 5 and longer. Therefore this length 5 must be the threshold where the brute-forcing fails to succeed. However, due to beauty of the method with the length we can approximate the average number of trials by using the techniques from Chapter 4.

Figures 5.1 and 5.2 show the error convergence of the polynomial `6 * (x - 2)**5` for one round. The dots show the smallest error achieved up to that point. Note that the x-axis of Figure 5.2 shows the number of generations, where each generation is equivalent to 1000 trials because we set `population_size=1000`.

We see that the error of the non-evolutionary method becomes stationary from the 200'000-th trial. On the other hand, the error of evolutionary method faster but still at a very slow rate. The error remains almost stationary at the 4'000-th generation and moves by only a little.

In Table 5.2 we now see more results. And if we compare the results from the two tables, it is easy to see that the evolutionary method clearly outperforms the naive approach. Astonishingly, polynomials that the non-evolutionary method fails to find are no challenge to the evolutionary method. For example the polynomial `x + x(x - 1)(x - 2)(x - 3)` needed approximately 700'000 trials until one solution was found. This is a speedup of six orders of magnitude when comparing with the theoretical mean. However, we have two polynomials where even the evolutionary method fails to complete even one round. One assumption why these two fail is because of the high degree and length together that makes it hard to find. High degree polynomials have many local optima and may get stuck there. Therefore even a mutating the expression tree is not significant enough to 'escape' from these local optima.

5.2 Hyperparameter tuning on short polynomials

In this experiment we try to find how mutation and crossover affects the performance on short polynomials of fixed lengths. The polynomials that were tested are the zero polynomial `0` of fixed lengths 0 and 1 and `1` and `2` of fixed length 1. All runs in this experiment were done in 1000 rounds in order to obtain accurate results with low variance.

Tables 5.3, 5.4, 5.5 and 5.6 show the results of changing the mutation- and the crossover rate. For these experiments we fixed the population and the tournament size to 15 and 10 respectively. Note that these population and tournament size are intentionally set to be low because these polynomials already have a low mean of around 20 and in order to see the effect of crossover and mutation, the algorithm needs to create the whole population before crossover and mutation takes place. Otherwise we would simply stop when the solution is found.

From these three tables we see that adding mutation and crossover worsens the performance. The entry where mutation and crossover rates are 0 delivers the best results. This might be because these polynomials already have a high chance of being generated and adding mutation and crossover is equivalent to adding noise.

Note that the last row in table 5.3 has no result. In not all but most of the cases, the program simply becomes stuck due to the low diversity in the tournament sets. Therefore we should avoid a crossover rate of 1.0.

The results from table 5.6 deviate from the other. Here the opposite shows to be true. We see that a mutation and crossover rate of 1.0 and 0.8 respectively delivers the best result. We can assume that between a mean of 20 and 130 lies the threshold where mutation and crossover start to become useful.

Table 5.3: Mutation/Crossover tuning for `0` with `length=0` fixed*

Mutation	Crossover	Mean	Standard deviation	Runtime (s)
0.0	**0.0**	**19.535**	**18.535**	**1.267**
0.1	0.2	21.999	23.143	1.457
0.1	0.4	25.93	31.123	1.672
0.1	0.6	32.814	41.989	2.077
0.1	0.8	58.203	89.452	3.746
0.1	1.0	N/A	N/A	N/A

* `population_size=15, tournament_size=10` fixed

Table 5.4: Mutation/Crossover tuning for `0` with `length=1` fixed*

Mutation	Crossover	Mean	Standard deviation	Runtime (s)
0.0	**0.0**	**11.882**	**12.173**	**1.707**
0.1	0.2	13.381	14.129	3.064
0.1	0.4	14.165	17.53	3.043
0.1	0.6	16.767	22.682	3.543
0.1	0.8	19.671	29.61	3.424

* `population_size=15, tournament_size=10` fixed

5.3 Hyperparameter tuning on longer polynomials

This experiment is similar to the previous one however the length of the polynomial is not fixed. We will examine the effect of crossover and mutation for the polynomials `2 * x + 5` and `x**5 - x**2`. For these experiments we fixed the population- and tournament size to 100 and 10 respectively.

Table 5.7 shows the result of a grid-search of the parameters. We see that the best result is achieved with a mutation rate of 0.4 and a crossover rate of 0.6. Whereas the best result from table 5.8 is with a mutation- and crossover rate of 0.1 and 0.6 respectively.

Furthermore, we see that even a little amount of mutation can drastically improve the

Table 5.5: Mutation/Crossover tuning for `1` with `length=1` fixed*

Mutation	**Crossover**	**Mean**	**Standard deviation**	**Runtime (s)**
0.0	**0.0**	**28.551**	**28.016**	**4.57**
0.1	0.2	30.834	29.856	6.609
0.1	0.4	33.227	36.127	7.13
0.1	0.6	42.012	45.586	9.721
0.1	0.8	68.552	106.874	13.617

* `population_size=15, tournament_size=10` fixed

Table 5.6: Mutation/Crossover tuning for `2` with `length=1` fixed*

Mutation	**Crossover**	**Mean**	**Standard deviation**	**Runtime (s)**
0.0	0.0	131.656	132.756	18.123
0.2	0.6	103.897	94.173	19.276
0.2	0.8	195.76	331.323	32.461
0.4	0.6	83.523	69.932	14.851
0.4	0.8	123.012	268.39	20.616
0.6	0.6	73.099	59.58	14.124
0.6	0.8	69.329	67.327	13.683
0.8	0.6	69.973	56.024	12.281
0.8	0.8	64.053	68.824	13.011
1.0	0.6	69.839	56.929	13.295
1.0	**0.8**	**55.8**	**50.228**	**12.413**

* `population_size=15, tournament_size=10` fixed

performance and runtime. This difference can be seen by comparing the two entries with parameters [0.01, 0.4] and [0.0, 0.4]. The runtime is cut down to one fifth and the empirical mean in the worse run is at 64’000 compared to 19’000 in the run with little mutation.

From this and the previous experiment we can conclude that the parameter choice for mutation on crossover depends on the polynomial itself. In other words, there exist no set of values that is optimal for all polynomials. We could further investigate what type of polynomials tend to work better with a specific set of parameters but this would go beyond the scope of this thesis.

Table 5.7: Mutation/Crossover tuning for `2 * x + 5`[1]

Mutation	Crossover	Mean	Standard deviation	Runtime (s)
0.0	0.0	3’550.947	3’575.001	542.442
0.0	0.2	2’096.403	1’986.131	311.3
0.0	0.4	2’004.055	2’097.1	322.183
0.0	0.6	2’601.0	2’798.910	369.932
0.0	0.8	4’733.892	6’497.883	670.234
0.2	0.2	1’608.093	1’520.338	243.161
0.2	0.4	1’190.585	1’089.567	186.565
0.2	0.6	1’208.137	1’119.043	188.838
0.2	0.8	1’469.635	1’637.172	223.641
0.4	0.2	1’695.405	1’690.377	295.340
0.4	0.4	983.574	888.563	156.242
0.4	**0.6**	**948.9**	**894.386**	**180.262**
0.4	0.8	1077.198	1069.447	198.743

[*] `population_size=100, tournament_size=10` fixed

Table 5.8: Mutation/Crossover tuning for `x**5 - x**2`*

Mutation	Crossover	Mean	Standard deviation	Runtime (h)
0.0	0.0	172’489.066	172’073.831	18.174
0.005	0.2	26’748.498	36’632.662	3.643
0.01	0.2	23’681.492	35’259.138	3.152
0.05	0.2	15’580.208	18’400.505	1.879
0.1	0.2	13’816.043	15’338.775	5990.072
0.005	0.4	26’130.472	50’121.839	1.663
0.01	0.4	18’769.775	36’252.052	2.863
0.05	0.4	11’709.647	16’266.382	1.548
0.1	0.4	10’206.849	13’488.606	1.351
0.005	0.6	24’481.012	53’003.249	4.818
0.01	0.6	20’034.162	42’828.928	3.662
0.05	0.6	12’334.297	17’357.220	1.819
0.1	**0.6**	**9’603.815**	**11’911.830**	**1.351**
0.4	0.6	10601.647	13737.221	1.469
0.005	0.8	46’117.016	163’361.233	13.26
0.01	0.8	27’623.236	77’914.140	5.753
0.05	0.8	14’086.166	22’359.067	2.240
0.1	0.8	11’546.916	14’937.217	1.72
0.0	0.2	54’603.291	88’639.632	8.146
0.0	0.4	64’728.996	119’760.309	15.121

* `population_size=1000, tournament_size=1` fixed

Chapter 6

Conclusions

In this thesis we have build a fast evolutionary method for predicting sequences. We described how the naive approach works and how we implemented a genetic algorithm on top to improve the performance and runtime. Furthermore, we analyzed and predicted the runtime of the brute-force method based on probabilities. We performed experiments where we compared the non-evolutionary method with evolutionary. From the results of test on various polynomials we can conclude that evolutionary method is a far superior method for sequence prediction. Additionally, the results of experiments on hyperparameter tuning shows that this algorithm can be fine-tuned to further improve perfomance, however the optimal parameter choice varies for each polynomial.

In the future, further experiments can be done on more polynomials to examine further strengths and limitation of the evolutionary method. Moreover, with this thesis we showed that we can implement a fast method to predict sequence. An interesting idea for further future work could be to try to apply this method for different types of sequences besides polynomials such as recursively defined integer sequences.

Bibliography

[1] N. J. A. Sloane, "The oeis: A fingerprint file for mathematics," May 2021.

[2] F. Briggs and M. O'Neill, "Functional genetic programming and exhaustive program search with combinator expressions," Aug. 2007.

[3] S. P. Jones, "The implementation of functional programming languages," 1987.

[4] "Gitlab," https://gitlab.ethz.ch/disco-students/fs21/sequence-evolution.git.

[5] J. Schmidhuber, "Optimal ordered problem solver," 2004.

[6] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," 1994.

[7] "scicomp," https://scicomp.ethz.ch/wiki/Euler, Aug. 2014, last accessed 26 July 2021.