SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 000

Evolving cache replacement policies using grammatical evolution

Mihael Miličević

Zagreb, svibanj 2022.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

CONTENTS

1. Introduction

2. Theoretical background

2.1. Context-free grammars

A formal grammar is a 4-tuple $G = (V, T, P, S)$ (Sipser, 2013) where

- V is a finite set of nonterminal symbols.
- T is finite set of terminal symbols, disjoint from V.
- P is finite set of production rules that represent the recursive definition of a language. Each production has three components (Hopcroft et al., 2007):
  - head, which consists of one or more terminal and nonterminal symbols.
  - production symbol $\rightarrow$.
  - body, which consists of zero or more terminal and nonterminal symbols. If the body is empty (consists of zero symbols), we denote that by writing ϵ.
- S is a start nonterminal symbol.

We can think of the terminal T and nonterminal V symbols as the building blocks of a grammar. Productions P define the rules by which these building blocks substitute each other in the process of building a string. This string belongs to the language which is defined by the grammar. The start symbol S represents the starting building block. The process of building a string starts by applying a production over the start symbol, and this process of applying productions over intermediate strings continues until no more productions can be applied. At that point, the result string is built, and it consists only of elements of the T set, which belong to the alphabet of the language.

The most general grammars, defined only with the aforementioned rules and with no other restrictions, are also called unrestricted grammars. For example, we can define one unrestricted grammar like $G = (\{A, B, C\}, \{a, b, c, d\}, P, A)$ where P contains productions such as:

- $AB \rightarrow c$
- $aBdC \rightarrow \epsilon$
- $a \rightarrow ABdCAB$
- $abba \rightarrow ABBA$

Context-free grammars are a subset of unrestricted grammars which impose one restriction, that the head of a production is exactly one symbol from the nonterminal symbols V. With this restriction, none of the previously written productions can be valid, but we can write productions such as:

- $A \rightarrow \epsilon$
- $B \rightarrow a$
- $A \rightarrow abCdB$
- $C \rightarrow abba$

Context-free grammars get their name from their property that every nonterminal symbol A from V can be substituted with a string of terminals and nonterminals S if there exists a production rule in P such that $A \rightarrow S$, no matter it's surrounding context in the intermediate string.

Context-free grammars have played a central role in the compiler technology (Hopcroft et al., 2007). They are used to define syntax rules of a language, as they are expressive enough to allow recursions and other concepts we expect from programming languages, yet they are simple enough to be parsed effectively.

2.2. Genetic algorithm

Genetic algorithm is a metahueristic inspired by the Darwinian theory of evolution. The main idea is that, given a population in an environment with limited resources, competition for those resources causes natural selection (Eiben and Smith, 2015). This principle is sometimes called 'survival of the fittest'.

One of the first design choices in solving an optimisation problem using a genetic algorithm is individual representation. We can distinguish two different terms - genotype and phenotype. Genotype corresponds to the encoding of an individual. Variation operators (recombination and mutation) are applied on the genotype of an individual.

In the process of evaluating an individual, it's genotype is decoded onto it's phenotype. Phenotype corresponds to the actual solution, and we determine how good of a solution one individual is by evaluating it's phenotype using a fitness function.

The basic idea flow of a genetic algorithm is described as (Eiben and Smith, 2015):

Algorithm 1 Genetic algorithm

```
1: initialise population;
2: evaluate each solution;
3: repeat while (terminal condition not satisfied)
4:     select parents;
5:     recombine pairs of parents;
6:     mutate acquired children;
7:     evaluate children;
8:     select individuals for next generation;
```

In line 1, we initialise the population. We can either initialise it with random solutions, or, if we know some good solutions which can be a good starting point for the evolutionary search process, we can seed it with those.

In line 2, we evaluate each solution using a fitness function. Fitness function is applied over the individual's phenotype, and it returns a measure of how good a solution is. In that case, the goal of the genetic algorithm is to find the solutions which maximize this measure. When solving some problems, it is much easier to determine how bad of a solution one individual is, rather than how good of a solution it is. In that case, fitness function returns a measure of how bad a solution is, and the goal of the genetic algorithm is to minimize this measure.

In line 3, we start the evolutionary loop. Each new iteration of this loop corresponds to a different generation of the population. This loop is stopped when some stopping criteria is satisfied. The simplest stopping criteria is reaching the predetermined number of iterations. If we know how good of a solution we need, we can also stop this loop once we find one such solution.

In line 4, we select the parents which we will later recombine to get their offspring, which will be new candidate solutions. When choosing which individuals will be parents, we want to keep their fitness in mind. We want to choose the better individuals more often, in order to guide the evolutionary search towards better solutions. If we don't discriminate individuals over their fitness when selecting parents for future solutions, our evolutionary search can degrade into random search. On the other hand, we want to be able to choose the bad solutions to be parents as well, albeit with a low

chance. If we chose only the good solutions to be parents, our search could become too greedy, and get stuck in a local optimum.

In line 5, we recombine the genotype of selected parents to get the genotype of their offspring. Recombination process can yield one or more offspring. The idea behind recombination is - if we chose a good genotype-phenotype mapping, we can expect that an individual's fitness will be coded into it's genotype. Combining good solutions should yield even better solutions, if we choose the good parts of the first parent and good parts of the second parent, and then combine these good parts into a new solution. Of course, since the recombination process is random, we can also sometimes choose the bad part of the first parent and the bad part of the second parent to get an individual that is worse than it's parents, but that isn't a significant problem, since we can also get good solutions which can guide the evolutionary search.

In line 6, we apply the mutation operator on newly constructed individuals, which are the result of the recombination process. The main idea behind mutation is that it should increase diversity of the population. Mutation operator should not be guided by some rule, instead, it should be random and unbiased (Eiben and Smith, 2015). If we don't include the mutation operator, or if we set the mutation rate to be too low, our search can get stuck in a local optimum. On the other hand, we don't want the mutation rate to be too large either. Since the new solutions are constructed from good solutions (with a high chance), and we expect that good solutions have their fitness coded into their genotypes, if the mutation operator is too disruptive, it can nullify the good results of the recombination, and degrade our evolutionary search into random search.

In line 7, we evalute newly construced individuals. These individuals are being evaluated with the same fitness function used in line 2.

In line 8, we choose which individuals carry on to the next generation. Unlike the parent selection, which is stochastic, this process is usually deterministic. Two common strategies are choosing the best individuals from both parents and offspring, and the age-biased approach, which chooses only from the offspring (Eiben and Smith, 2015). If we choose the latter approach, a problem we might encounter is loosing good solutions, if these good solutions have no children better than themselves. To counter this problem, we can choose to carry over a few of the best solutions into the next generation without changing them, and this approach is called elitism.

2.3. Grammatical evolution

Grammatical evolution is a evolutionary computation technique used to evolve computer programs which have a high fitness in regards to a fitness function, or in other words, which do some certain task well. The phenotypes of the individuals are executable computer programs. The genotypes of the individuals are arrays of 8-bit numbers, called codons. The genotype-phenotype mapping is done using a context-free grammar which generates the language of desired solutions.

The process of genotype-phenotype mapping works as follows: we start at the first codon in individual's genotpye, and we traverse the parsing tree using depth-first search, until we find a nonterminal symbol. Once we find a nonterminal symbol, we check all productions of our specified grammar in which this nonterminal symbol is the left side (head). We calculate which production to apply on this nonterminal symbol using the formula

$$Production = (Codon\ integer\ value)$$

$$MOD$$

$$(Number\ of\ productions\ for\ the\ current\ nonterminal\ symbol)$$

Once we apply the chosen production, we move on to the next codon. If we have reached the end of codon array, we move to the first codon, and this process is called wrapping. In practice, the maximum number of wrappings is specified, in order to avoid infinite recursions, and if the process exceeds this maximum number of wrappings, the mapping fails and the individual's fitness is set to zero.

This process of traversing the parsing tree and applying grammar productions continues until there are no more nonterminal symbols in the parsing tree, and at that point the mapping is complete (O'Neill and Ryan, 2003).

Grammatical evolution has some interesting unique properties, the wrapping operator and code degeneracy (O'Neill and Ryan, 2003).

During the genotype-phenotype mapping process, an individual could run out of codons. In that situation, the wrapping operator is applied, which means that the mapping continues from the first codon. This technique draws inspiration from a phenomen exhibited by bacteria, viruses and mitochondria which allows them to reuse the same genetic material for the expression of different genes (O'Neill and Ryan, 2003).

Code degeneracy refers to the fact than the mapping process used in grammatical evolution is many-to-one. Many different codon configurations in genotype can map into the same phenotype program. For example, during the mapping process, if the

current nonterminal symbol has two different productions as specified by the grammar, then the first production would be chosen if the current codon value is even, and the second production would be chosen if the current codon value is odd. This is because $0\,MOD\,2\,=2\,MOD\,2\,=4\,MOD\,2\,=6\,MOD\,2\,=\,...\,=254\,MOD\,2\,=0$, and $1\,MOD\,2\,=3\,MOD\,2\,=5\,MOD\,2\,=7\,MOD\,2\,=\,...\,=255\,MOD\,2\,=1$. The values are shown up to 254 and 255 because these are the largest even and odd numbers, respectively, that fit into the 8 bits of one codon. This property of the genotype-phenotype mapping which enables it to have many different genotypes which all map to the same phenotpye cultivates the genetic diversity of the population (O'Neill and Ryan, 2003).

Grammatical evolution has some drawbacks, namely low locality and high redundancy. Low locality refers to the property of the genotype-phenotype mapping that small changes in genotype can cause drastic changes in phenotype, or even completely different phenotypes. High redundancy refers to the fact that the genotype-phenotype mapping is many-to-one, which means that many different genotypes can map to a single phenotype. Because of these two properties, the evolutionary search process in grammatical evolution can sometimes behave like random search (Mégane et al., 2022). To solve these problems, some extensions of the standard grammatical evolution have been proposed, like the structured grammatical evolution (Lourenço et al., 2018).

To demonstrate the process of genotype-phenotype mapping in the grammatical evolution, we will define a context-free grammar and one individual's genotype, and then show a step by step mapping from genotype to phenotype. Instead of defining a grammar using the (V, T, P, S) 4-tuple, we will define it using the BNF notation. Terminal symbols are written as lowercase symbols, nonterminal symbols are written as uppercase symbols surrounded by symbols '<' and '>', and the first nonterminal symbol is the start symbol. Production heads are on the left side of the '::=' string, and on the right side of this string are bodies of productions which share their left side, separated by the symbol 'I'. We will define our grammar as:

$<S>\ ::=\ a\ <A>\ b\ <S>\ |\ <C>\ d\ <A>$

$<A>\ ::=\ c\ <B>\ <A>\ c\ |\ a\ b\ <C>\ |\ d$

$<B>\ ::=\ <S>\ a\ <S>\ |\ <C>\ d\ <A>\ |\ b\ d$

$<C>\ ::=\ c\ <C>\ |\ <C>\ d\ <C>\ |\ a$

We will also define one individual with genotype:

$$[\,176,\ 49,\ 168,\ 253,\ 8,\ 65,\ 127,\ 26,\ 130,\ 100\,]$$

All the intermediate strings will be displayed like linear strings, which can be constructed by traversing the parsing tree in depth.

The mapping process starts with the string $<S>$, and the index $i = 0$ in the genotype array. Since $i = 0$, the codon value we will use at this step is 176. The first nonterminal symbol in the string is $<S>$, and it has 2 productions defined in the grammar. So we calculate $176\ \ MOD\ \ 2\ =\ 0$, and apply the production at index 0, which is $<S>\ ::=\ a\ <A>\ b\ <S>$.

Our current string is $a\ <A>\ b\ <S>$, the index value is $i = 1$, and the codon value is 49. The leftmost nonterminal symbol is $<A>$, which has 3 productions defined. So we calculate $49\ \ MOD\ \ 3\ =\ 1$, and apply the production at index 1, which is $<A>\ ::=\ a\ b\ <C>$

Our current string is $aab<C>b<S>$, the index value is $i = 2$, and the codon value is 168. The leftmost nonterminal symbol is $<C>$, which has 3 productions defined. So we calculate $168\ \ MOD\ \ 3\ =\ 0$, and apply the production at index 0, which is $<C>\ ::=\ c\ <C>$.

Our current string is $aabc<C>b<S>$, the index value is $i = 3$, and the codon value is 253. The leftmost nonterminal symbol is $<C>$, which has 3 productions defined. So we calculate $253\ \ MOD\ \ 3\ =\ 1$, and apply the production at index 1, which is $<C>\ ::=\ <C>\ d\ <C>$.

Our current string is $a\ a\ b\ c\ <C>\ d\ <C>\ b\ <S>$, the index value is $i = 4$, and the codon value is 8. The leftmost nonterminal symbol is $<C>$, which has 3 productions defined. So we calculate $8\ \ MOD\ \ 3\ =\ 2$, and apply the production at index 2, which is $<C>\ ::=\ a$.

Our current string is $a\ a\ b\ c\ a\ d\ <C>\ b\ <S>$, the index value is $i = 5$, and the codon value is 65. The leftmost nonterminal symbol is $<C>$, which has 3 productions defined. So we calculate $65\ \ MOD\ 3\ =\ 2$, and apply the production at index 2, which is $<C>\ ::=\ a$.

Our current string is $aabcadab<S>$, the index value is $i = 6$, and the codon value is 127. The leftmost nonterminal symbol is $<S>$, which has 2 productions defined. So we calculate $127\ \ MOD\ \ 2\ =\ 1$, and apply the production at index 1, which is $<S>\ ::=\ <C>\ d\ <A>$.

Our current string is $a\ a\ b\ c\ a\ d\ a\ b\ <C>\ d\ <A>$, the index value is $i = 7$, and the codon value is 26. The leftmost nonterminal symbol is $<C>$, which has 3 productions

defined. So we calculate $26\ MOD\ 3\ =\ 2$, and apply the production at index 2, which is $<C>\ ::=\ a$.

Our current string is $a\,a\,b\,c\,a\,d\,a\,b\,a\,d\,<A>$, the index value is $i = 8$, and the codon value is 130. The leftmost nonterminal symbol is $<A>$, which has 3 productions defined. So we calculate $130\ MOD\ 3\ =\ 1$, and apply the production at index 1, which is $<A>\ ::=\ a\,b\,<C>$.

Our current string is $a\,a\,b\,c\,a\,d\,a\,b\,a\,d\,a\,b\,<C>$, the index value is $i = 9$, and the codon value is 100. The leftmost nonterminal symbol is $<C>$, which has 3 productions defined. So we calculate $100\ MOD\ 3\ =\ 1$, and apply the production at index 1, which is $<C>\ ::=\ c\,<C>$.

Our current string is $a\,a\,b\,c\,a\,d\,a\,b\,a\,d\,a\,b\,c\,<C>$, the index value is $i = 10$. Since our codon array has only 10 elements, index $i = 10$ is out of bounds, but the mapping process is incomplete, so we apply the wrapping operator and set $i = 0$. The codon value is 176. The leftmost nonterminal symbol is $<C>$, which has 3 productions defined. So we calculate $176\ MOD\ 3\ =\ 2$, and apply the production at index 2, which is $<C>\ ::=\ a$.

Our current string is $a\,a\,b\,c\,a\,d\,a\,b\,a\,d\,a\,b\,c\,a$. Since there are no nonterminal symbols in this string, the mapping process is finished, and this string is the final result.

3. Evolving cache replacement policies

3.1. Cache replacement policies

3.2. Grammar

4. Experimental results

5. Conclusion

BIBLIOGRAPHY

A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing, 2nd edition*. Springer, 2015.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Automata Theory, Langugages and Computation, 3rd edition*. Pearson Education, Inc, 2007.

Nuno Lourenço, Filipe Assunção, Francisco B. Pereira, Ernesto Costa, and Penousal Machado. Structured grammatical evolution: A dynamic approach. 2018.

Jessica Mégane, Nuno Lourenço, and Penousal Machado. Co-evolutionary probabilistic structured grammatical evolution. 2022.

Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Springer, 2003.

Michael Sipser. *Introduction to the Theory of Computation, 3rd edition*. Cengage Learning, 2013.

Appendix A
Sofware architecture

Appendix A Body

Evolving cache replacement policies using grammatical evolution

Sažetak

Sažetak na hrvatskom jeziku.

Ključne riječi: Ključne riječi, odvojene zarezima.

Title

Abstract

Abstract.

Keywords: Keywords.