UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS ASSIGNMENT No. 688

Evolving cache replacement policies using grammatical evolution

Mihael Miličević

Zagreb, May 2022

CONTENTS

1. Introduction

Genetic programming is a branch of the evolutionary computing whose primary concern is evolving computer programs, starting mostly from random populations of programs. The evolution of these programs is inspired by the Darwinian theory of natural evolution, which lies on five basic principles(Čupić, 2019):

1. There are always more offspring than necessary.
2. The size of the population is approximately constant.
3. The quantities of the food resources are limited.
4. Species which sexually reproduce bear no identical offspring, instead there are always variations.
5. Most of these variations are hereditary.

One of the main tools in compiler design are context-free grammars. They provide an elegant, formal way of describing the structural rules of computer programs, and each valid computer program written in some arbitrary language satisfies the rules of that language's grammar. Grammar based approaches in the field of genetic programming has enjoyed much popularity (O'Neill and Ryan, 2003), and of the most popular and used technique is the grammatical evolution.

The topic of this thesis is using grammatical evolution in solving a popular benchmarking problem in the field of genetic programming, which is evolving cache replacement policy. The rest of this thesis is organised as follows: chapter 2 covers the theoretical background involved in this problem, chapter 3 explains the problem of cache replacement policies and explains how the problem can be solved using grammatical evolution, chapter 4 covers the experiment desing and results, and chapter 5 covers a conclusion of the thesis. Finally, in the appendix A, the computer system constructed for the experiments covered in chapter 4 is described in detail.

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;
9: end while
```

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(Čupić, 2019).

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 $1\,MOD2\ = 3\,MOD2\ = 5\,MOD2\ = 7\,MOD2\ =\ ...\ = 255\,MOD2\ = 1$. 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 '|'. 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 more nonterminal symbols in this string, the mapping process is finished, and this string is the final result of the mapping.

3. Evolving cache replacement policies

3.1. Cache replacement policies

Paging is a memory management scheme that breaks up physical memory into fixed-size blocks called frames, and breaks up logical memory into same size blocks called pages (Silberschatz et al., 2018). Physical memory refers to the real memory the operating system works with, and the logical memory refers to the memory space of each program being executed on the machine. Before a process is executed, it's pages are loaded into some frames.

Since the sum of logical spaces of all the programs being executed can exceed the physical space, sometimes a page must be kicked out of a frame, in order to make room for another page. This process is called swapping. The central question here is: which page should be kicked out? There are several strategies for this problem, and they are called cache replacement policies.

The FIFO (First In, First Out) strategy organizes the cache as a standard queue. The pages are kicked out in the order in which they were added.

The LRU (Least Recently Used) strategy kicks out the page which wasn't used for the longest time in the past. This strategy relies on the locality of reference, which means that if a page was recently accessed, it is likely that it will be accessed again in the near future.

The LFU (Least Frequently Used) strategy kicks out the page which was accessed the least number of times in some period of time.

The CLOCK strategy, also called the second chance strategy, traverses the frames in a round robin way. Each frame has a flag which can either have the value zero or one. When a page needs to be kicked out, the algorithm starts iterating over the frames, going to the first one once it reaches the end. If the current frame has the value of it's flag equal to zero, it's page is kicked out and the search is finished. If, on the other hand, it's flag is equal to one, the flag is set to zero, and the algorithm continues to the next frame, hence the 'second chance' name.

The general cache replacement policy looks as follows:

Algorithm 2 Cache replacement policy

```
1:  for each page in the requests array do
2:      if page in cache
3:          do nothing;
4:      else if exists an empty frame
5:          put page in frame;
6:      else
7:          decide upon a frame;
8:          remove current page from frame;
9:          put page in frame;
10:     end if
11: end while
```

Lines 2 and 3 correspond to the situation where the currently required page is already in the cache. In that case, no action is needed.

Lines 4 and 5 correspond to the situation where the requested page isn't in the cache, but there exists an empty frame. In that case, we put the requested page in that empty frame.

Finally, lines 7, 8 and 9 correspond to the situation where teh requested page isn't in the cache, and all the frames already have some other page in them. In that case, first we must decide from which frame we will remove a page (line 7). Finally, we swap the old page from that frame with the requested page (lines 8 and 9).

An important note here is that all the cache replacement policies vary only in the line 7, that being the decision which frame will host the requested page. All other parts of the algorithm are equal in all strategies.

When evolving cache replacement policies, we only change the line 7. Since the topic of this thesis is evolving these policies using grammatical evolution, the first step in this process is defining a grammar. This is a crucial step and the next subchapter will describe the constructed grammar in detail.

3.2. Grammar for evolving cache replacement policies

Before listing and describing all the productions of our grammar, first we will go through some of the nonterminal symbols, the building blocks of our strategies. It is also important to note that the implementation of this software system was written in the C++ language, so the grammar was written in a way that satisfies the C++ language rules. The architecure of the built software system is described in detail in Appendix A.

Each strategy will keep track of some information about frames and pages, and this information will be available when choosing a frame in which the requested page will be stored. Each strategy will have four of these information arrays.

The first array is called last_accessed and it keeps track of when each frame was last accessed. We will also introduce functions last_accessed_min and last_accessed_max which will return the index of the frame which was accessed least recently and most recently, respectively.

The second array is called page_access_count and it keeps track of how many times each page was requested. We will introduce functions page_access_count_min and page_access_count_max which will return the index of the page that was accessed the least number of times and the most number of times, respectively.

The third array is called added_to_cache and it keeps track of when each page was added to cache. We will introduce functions added_to_cache_min and added_to_cache_max which will return the index of the page that was added to cache least recently and most recently, respectively.

Finally, the fourth array is called accessed. The first function that manages this array is called get_accessed. It takes one argument, the frame index, and returns the information stored about this frame. The second function is calles set_accessed. It takes two arguments, the first being the index of the frame, and the second being some value, and it stores that value at the index of the frame in the accessed array. Each time a frame is accessed, it's value is set to one. This array was introduced to make implementation of the CLOCK strategy possible, as will be explained in subchapter 3.3.

The start symbol is $<block>$ and it corresponds to zero or more simpler statements, which are encapsulated in the $<statement>$ symbol. Each statement corresponds to a programming construct like a loop or a simple one line statement.

During the process of writing and fine-tuning the grammar, one of the problems was the possibility of loops occuring inside loops. This would drastically slow down the process of choosing a frame, which is a decision that should be made relatively quickly. To solve this problem, we will introduce the $<block\_no\_loop>$ and $<statement\_no\_loop>$ symbols. These symbols are semantically equivalent to $<block>$ and $<statement>$, except they won't be able to contains loops, and by introducing these symbols we will ensure that loops can't be nested.

Symbols $<expression>$ and $<term>$ will be used to generate simple expressions. The $<modifiable>$ symbol will agregate all terms which can go on the left and the right side of the equality operator '=', while the $<non\_modifiable>$ symbol will aggregate all terms which can go only on the right side of the equality operator.

Symbols $<bool>$ and $<number>$ will be used to generate constants.

The productions for the symbol $<block>$ look as follows:

$<block>$ ::= $<statement>$ $<block>$

| ""

The first production allows recursive chaining of statements. The second production allows breaking the chaining, and without it, we would be stuck generating infinite statements one after another.

Similarly, productions for the $<block\_no\_loop>$ symbol are:

$<block\_no\_loop>$ ::= $<statement\_no\_loop>$ $<block\_no\_loop>$

| ""

Productions for the $<statement>$ symbol are:

$<statement>$ ::= if ($<expression>$) { $<block>$ } else { $<block>$ }

| if ($<expression>$) { $<block>$ }

| iterations = 0; while ($<expression>$ && iterations < frame_count) { $<block\_no\_loop>$ iterations++; }

| $<modifiable>$ = $<term>$;

| write ($<info\_field\_index>$, $<term>$, $<term>$) ;

| set_accessed ($<term>$,$<term>$)

The first production generates simple if-else branches, while the second production generates if branches without the else part. The third production generates loops. Loops are written as while loops, which initialize the counter variable iterations to zero, and then the loop is executed while some expression is evaluated as true and the iteration count is smaller than the number of frames in the cache. After each iteration, the iterations variable is increased by one. The fourth production generates instructions with equality operator. Finally, the fifth production generates calls of the 'write' function. Each strategy has two distinct arrays in which they can store informations about each frame. The write function takes three arguments, the first being the index of the array, the second being the index (or the frame number) in that array, and the third being the value that is being written at that index in that array.

Similarly, productions for the $<statement\_no\_loop>$ symbol are:

$<statement\_no\_loop>$ ::= if ($<expression>$) { $<block>$ } else { $<block>$ }

| if ($<expression>$) { $<block>$ }

| $<modifiable>$ = $<term>$;

|write($<info\_field\_index>$,$<term>$,$<term>$); |set_accessed($<term>$,$<term>$)

Productions for the $<expression>$ symbol are:

$<expression>$ ::= $<term>$

| $<term>$ == $<term>$

| $<term>$!= $<term>$

| $<term>$ > $<term>$

| $<term>$ >= $<term>$

| $<term>$ < $<term>$

| $<term>$ <= $<term>$

| $<term>$ + $<term>$

| $<term>$ - $<term>$

| $<term>$ * $<term>$

| division ($<term>$, $<term>$)

| remainder ($<term>$, $<term>$)

All of these productions generate simple arithmetic expressions. The division and remainder operators are written as functions, rather than just operators, because we will encapsulate these operations into functions that safely handle division by zero.

Productions for the $<term>$ symbol are:

$<term>$ ::= $<modifiable>$

| $<non\_modifiable>$

| $<expression>$

The $<term>$ symbol occurs inside expressions and it generates modifiable variables (which can be on the left and the right side of the equality operator), non-modifiable variables (which can only be on the right side of the equality operator), constants and nested expressions.

Productions for the $<modifiable>$ symbol are:

$<modifiable>$ ::= frame

| num1

| num2

| num3

We have already established that our strategies will only evolve the line 7 in the Algorithm 2, that being the decision in which frame should we put the page that isn't currently in the cache if all the frames are taken. For the sake of convenience, we will model this with functions that initialize the frame variable to zero and return that frame variable. The frame variable corresponds to the index of the chosen frame. We will want our strategies to modify this variable from it's initial value of zero, and that's why this variable will be modifiable. We will also give each strategy three free variables which they can change however they please, and these variables are called num1, num2 and num3.

Productions for the $<non\_modifiable>$ symbol are:

$<non\_modifiable>$ ::= time

| cache_size

| page_request

| find_min ($<info\_field\_index>$)

| find_max ($<info\_field\_index>$)

| read ($<info\_field\_index>$, $<term>$)

| page_access_count_min ( )

| page_access_count_max ( )

| last_accessed_min ( )

| last_accessed_max ( )

| added_to_cache_min ( )

| added_to_cache_max ( )

| get_accessed ($<term>$)

| $<number>$

| $<bool>$

The variable time refers to the index of the current request, and it gets increased by one for each new request. The variable cache_size refers to the number of frames in the cache, and the page_request variable refers to the page index of the page that is currently being requested. Function find_min takes one input argument, which is the index of strategy's arrays (and it can be one or two since each strategy has two info arrays), and it returns the index at which this array has the minimal vaue. The function find_max works in a similar way, except it returns the index at which the array has the maximal value. Function read takes two arguments, the first being the array index for this strategy, and the second being the index in that array, and this function reads the value in that array at that index and returns it.

Productions for the $<bool>$ symbol are:

$<bool>$ ::= true

| false

This symbol generates elementary Boolean logic constants, true and false.

Productions for the $<info\_field\_index>$ symbol are:

$<info\_field\_index>$ ::= 1

| 2

As it was explained previously, each strategy has two info arrays in which they can store arbitrary information about each frame. Addressing these info arrays is done using this symbol.

Finally, the symbols for generating integer constants are $<number>$, $<first\_digit>$, $<tail\_digits>$ and $<digit>$. Productions for generating numbers are recursive, and they can't start with the digit zero (unless the number is zero), since we want these numbers to be written in the decimal base, and if a number starts with 0 in the C++ language, it is interpreted as an octal number. Productions for these symbols are:

$<number>$ ::= $<first\_digit>$ $<tail\_digits>$ | 0

$<first\_digit>$ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

$<tail\_digits>$ ::= $<digit>$ $<tail\_digits>$ | ""

$<digit>$ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

3.3. Classic strategies written using the described grammar

One desired property of the grammar used in grammatical evolution is that it's able to generate all classic heuristic solutions, while also being able to generate some new solutions. In this subchapter, we will go through the classic stragies, which were explained earlier, and explain how they can be written using the grammar described in the subchapter 3.2.

3.3.1. FIFO (First In First Out)

The FIFO strategy can be written like:

```
$frame = added\_to\_cache\_min();$
```

3.3.2. CLOCK

The CLOCK strategy can be written like:

```
$iterations = 0;$
$while\ (get\_accessed(num1)\ ==\ 1\ \&\&\ iterations\ <\ frame\_count)\ \{$
    $set\_accessed(num1, 0);$
    $num1 = num1 + 1;$
```

```
        $if\ (num1 >= frame\_count)$ {
            $num1 = 0;$
    }
    $frame = num1;$
```

3.3.3. LRU (Least Recently Used)

The LRU strategy can be written like:

```
    $frame = last\_accessed\_min();$
```

3.3.4. LFU (Least Frequently Used)

The LFU strategy can be written like:

```
    $frame = page\_access\_count\_min();$
```

4. Experimental results

5. Conclusion

The goal of this thesis was to explore grammatical evolution as a grammar-based approach to genetic programming, study the problem of managing cache in modern computer systems, implement a computer system capable of evolving valid computer programs in an arbitrary language which solve an arbitrary problem, use the built computer system to generate and evolve cache replacement policies using real objective train data, test the generated strategies using real objective test data, and compare the generated strategies against classical heuristic approaches to solving the cache management problem on different cache sizes.

Grammatical evolution has proven able to generate cache replacement policies which can match classic heuristic approaches to solving this problem. For future work, it would be interesting to see how different derivatives and improved versions of this algorithm would compare against the simple version of the algorithm used in this thesis.

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.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 10th edition*. John Wiley and Sons, Inc., 2018.

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

Marko Čupić. *Umjetna inteligencija: Evolucijsko računarstvo*. 2019.

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.