UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS ASSIGNMENT No. 688

Evolving cache replacement policies using grammatical evolution

Mihael Miličević

Zagreb, June 2022

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 11 March 2022

BACHELOR THESIS ASSIGNMENT No. 688

Student:	**Mihael Miličević (0036521706)**
Study:	Electrical Engineering and Information Technology and Computing
Module:	Computing
Mentor:	prof. Domagoj Jakobović

Title: **Evolving cache replacement policies using grammatical evolution**

Description:

Describe the basic idea of evolutionary algorithms and the use of context-free grammars in computer science. Explore the paradigm of grammatical evolution as a machine learning method based on evolutionary algorithms. Define the page replacement strategy problem and describe popular heuristic page replacement strategies. Extend the existing framework for evolutionary computation with new representations in the form of grammatical evolution. Apply the realised grammatical evolution model to the page replacement strategy problem. Experimentally determine the efficiency of the implemented method with respect to the existing parameters and compare the obtained solutions with existing heuristic strategies. Include the source codes, the obtained results with necessary explanations and the used literature with the thesis.

Submission date: 10 June 2022

I would like to express my deepest gratitude to my mentor, prof. Jakobović, for his friendly and motivating mentorship, and his continued support, effort and help in writing this thesis. I would also like to thank my parents, for their support throughout my education.

CONTENTS

1. Introduction

Genetic programming is a branch of evolutionary computing whose primary concern is evolving computer programs, usually starting 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 [12]:

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 have enjoyed much popularity [9], and 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 policies. The rest of this thesis is organised as follows: chapter 2 covers the theoretical background involved in solving this problem, chapter 3 explains the problem of cache replacement policies and how it can be solved using grammatical evolution, chapter 4 covers the experiment design and the results analysis, and chapter 5 covers a conclusion of the thesis. Finally, in the appendix A, the software system constructed for the experiments covered in chapter 4 is described in detail.

2. Theoretical background

2.1. Context-free grammars

2.1.1. Formal definition

A formal grammar is a 4-tuple $G = (V, T, P, S)$ [11] where

- V is a finite set of nonterminal symbols.
- T is a finite set of terminal symbols, disjoint from V.
- P is a finite set of production rules that represent the recursive definition of a language. Each production has three components [5]:
  - 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 (if it consists of zero symbols), we denote that by writing ϵ.
- S is the start nonterminal symbol.

We can think of the terminal T and nonterminal V symbols as the building blocks of the 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 the elements of the T set, which belong to the alphabet of the language.

2.1.2. Formal grammar types

The most general grammars, defined only by the aforementioned rules and with no other restrictions, are called unrestricted grammars. For example, we can define one unrestricted grammar as $G = (\{A, B, C\}, \{a, b, c, d\}, P, A)$ 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 additional 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 in a context-free grammar, 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 its surrounding context in the intermediate string.

Context-free grammars have played a central role in the development of the compiler technology [5]. They are used to define the syntax rules of a language, as they are expressive enough to allow recursions, nesting and other concepts we expect from programming languages, yet they are simple enough to be parsed effectively.

2.2. Genetic algorithm

2.2.1. Basic principles

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 [4]. 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, its genotype is decoded onto its phenotype. Phenotype corresponds to the actual solution, and we determine how good of a solution one individual is by evaluating its phenotype using a fitness function.

2.2.2. Genetic algorithm pseudocode

The basic flow of the genetic algorithm is described as [4]:

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 the 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. Two common ways of choosing parents for the offspring are fitness-proportional selection, where the chance of each individual being chosen is proportional to its fitness, and tournament selection, where a random subset of the population is taken into account, and then the best individual or the best two individuals in this subpopulation are chosen as parents. One popular variant of the tournament selection is initialising the subpopulation of size three, choosing the two best individuals as parents, and replacing the third individual (the worst one) with the offspring of the chosen parents [12].

In line 5, we recombine the genotypes 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 its 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 its parents, but that isn't a significant problem, since we can also expect to 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 [4]. 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 the newly constructed 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 [4]. 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. The property of never loosing the best solutions between generations is called elitism [12].

2.3. Grammatical evolution

2.3.1. Definition

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. Authors of the algorithm have proposed using 8-bit codons in [9]. They've also ran some experiments with 12-bit codons and 16-bit codons, and the results have shown that increasing the codon size can result in a slower evolutionary search. The genotype-phenotype mapping is done using a context-free grammar which generates the language of the desired solutions.

The process of the 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 the 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 the lowest possible value.

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 [9].

2.3.2. Properties

Grammatical evolution has some interesting unique properties, namely the wrapping operator and code degeneracy [9].

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 [9].

Code degeneracy refers to the fact than the mapping process used in grammatical evolution is many-to-one. Many different codon configurations in the genotype can map onto 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 [9].

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 the genotype can cause drastic changes in the 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 [8]. To solve these problems, some extensions of the standard grammatical evolution have been proposed, like the structured grammatical evolution [7].

2.3.3. Demonstration of the genotype-phenotype mapping process

To demonstrate the process of the 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 the genotype to the 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 the symbols '<' and '>', and the first nonterminal symbol is the start symbol. Production heads are written 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:

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

We will also define one individual with the genotype:

$$[\,176,\ 49,\ 168,\ 253,\ 8,\ 65,\ 127,\ 26,\ 130,\ 99\,]$$

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

Each step of the mapping process will be followed by a figure showing the current state of the process visually. On the top of all figures is the unit's genotype shown as an array. All the fields in this array are colored ih blue, except for the one which the index used in mapping points to at this turn, as this field is colored in yellow. Beneath the genotype array will be the parsing tree built up to that point. The nonterminal symbol

which will be expanded at this turn is colored in red. The nonterminal symbols which have already been expanded are colored in green, and the nonterminal symbols which haven't yet been expanded and won't be at this turn, but sometime later in the future are colored in orange. Finally, the leaves of the tree, which are the terminal symbols, are colored in brown. These figures were built using the tool [3].

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 $<S> ::= a <A> b <S>$.

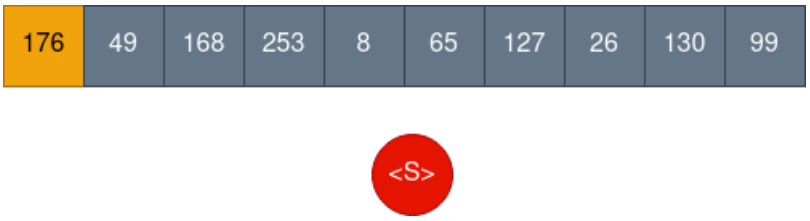

Figure 2.1: Genotype-phenotype mapping, step 1

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 $<A> ::= a \, b <C>$

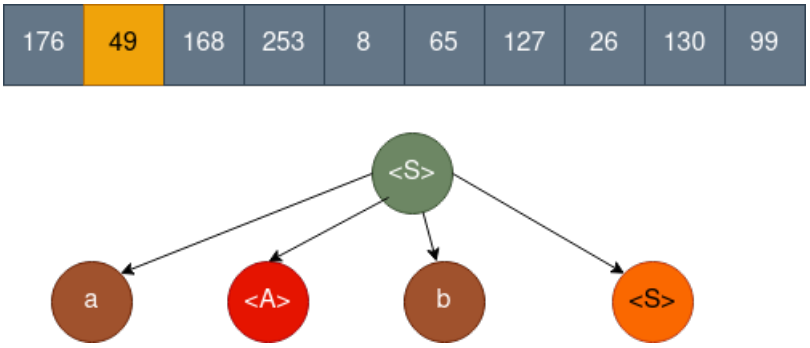

Figure 2.2: Genotype-phenotype mapping, step 2

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 $<C> \ ::= \ c \, <C>$.

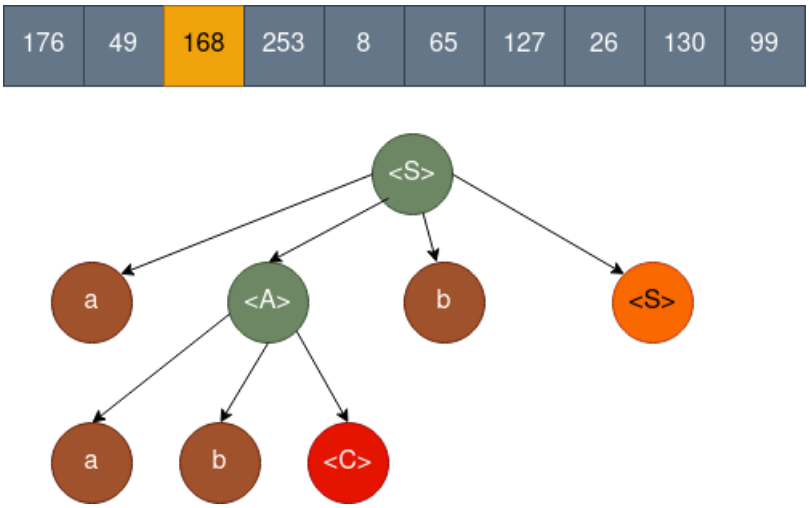

Figure 2.3: Genotype-phenotype mapping, step 3

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 $<C> \ ::= \ <C> \, d \, <C>$.

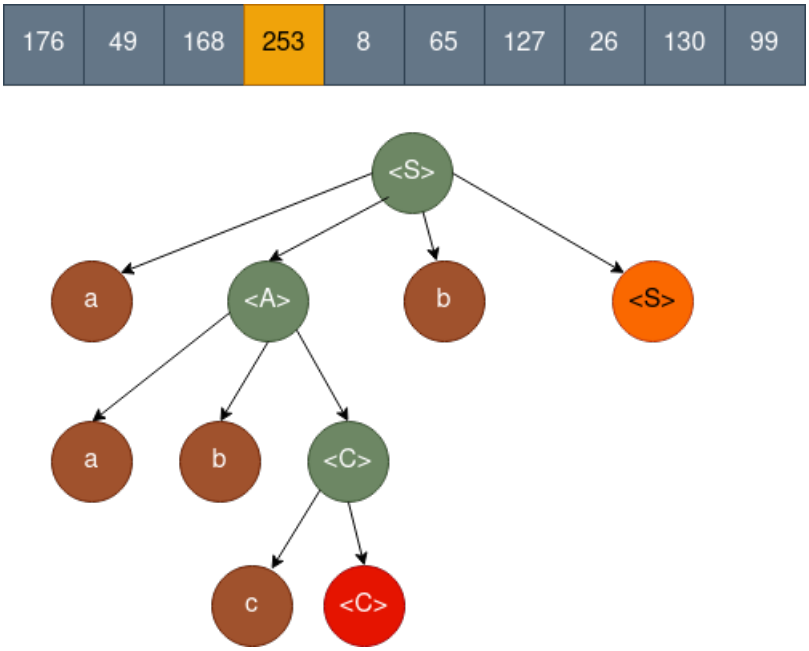

Figure 2.4: Genotype-phenotype mapping, step 4

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 $<C>\ ::=\ a$.

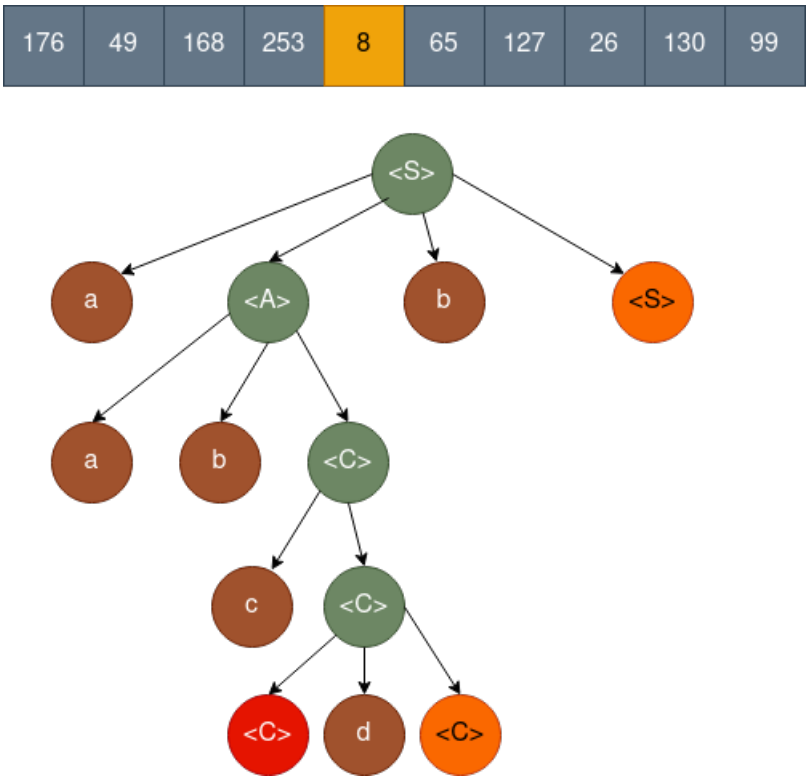

Figure 2.5: Genotype-phenotype mapping, step 5

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 $<C> ::= a$.

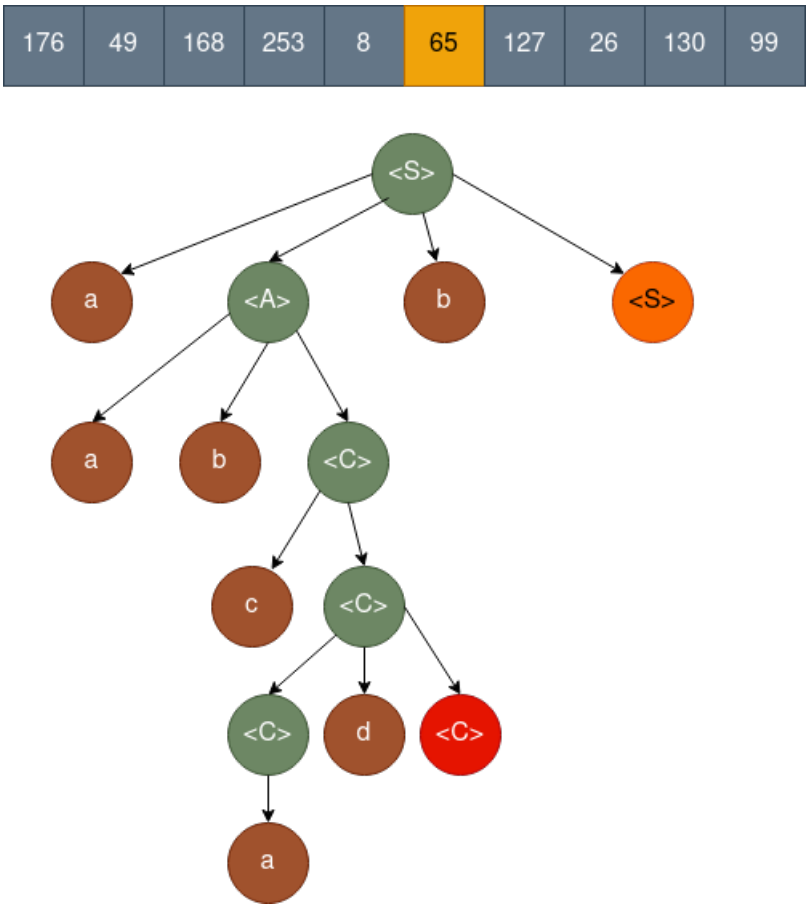

Figure 2.6: Genotype-phenotype mapping, step 6

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 $<S> \ ::= \ <C> \, d \, <A>$.

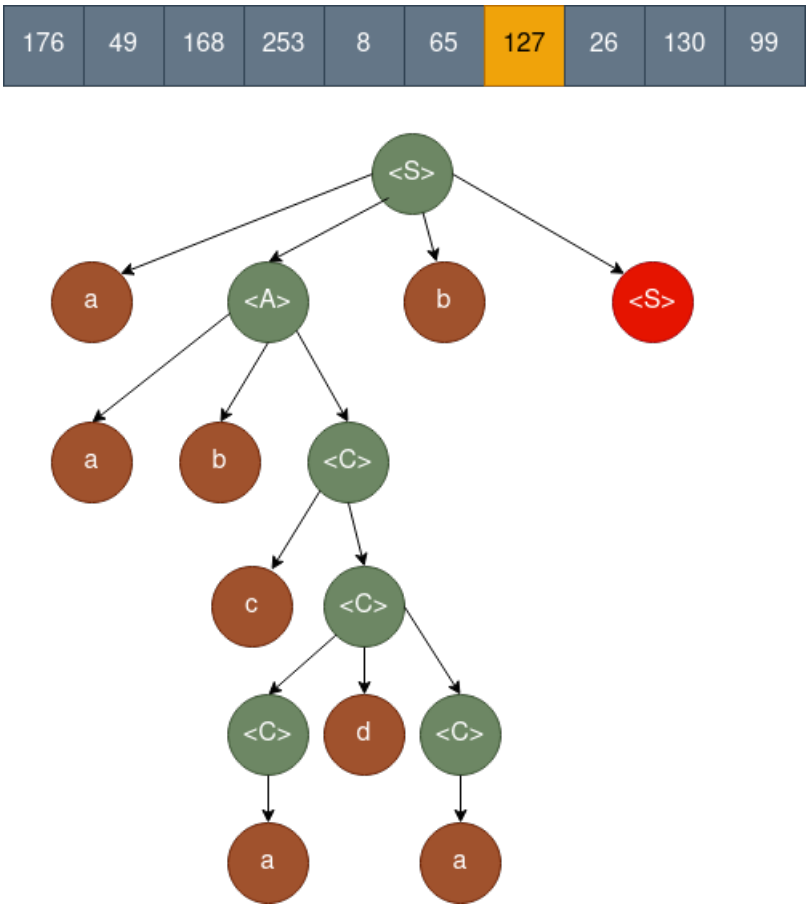

Figure 2.7: Genotype-phenotype mapping, step 7

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 $<C>\ ::=\ a$.

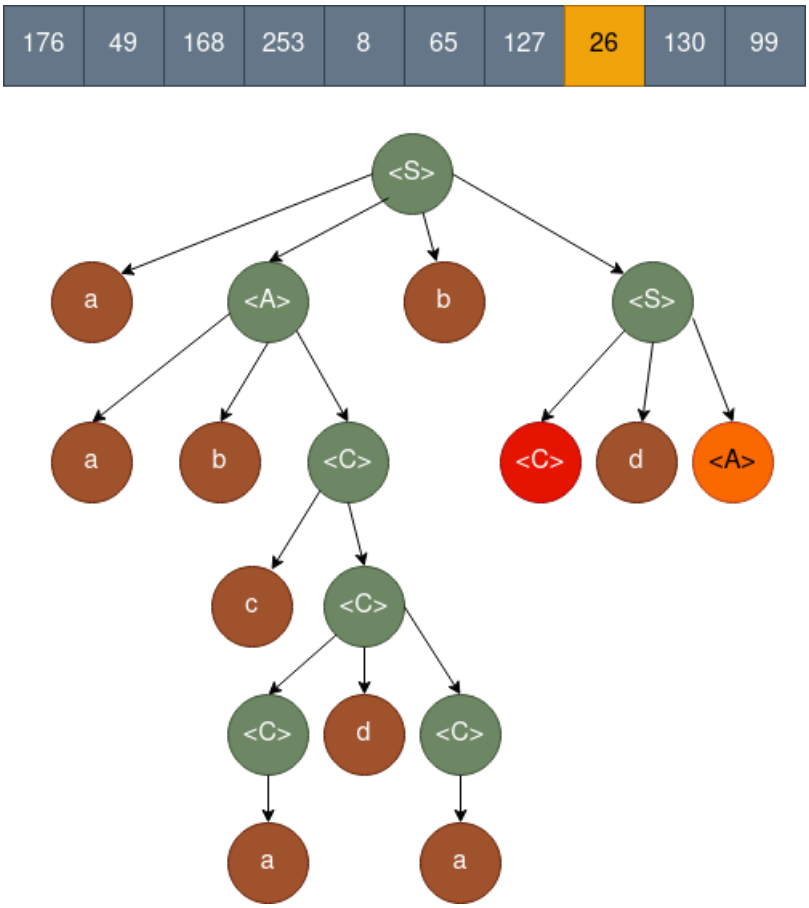

Figure 2.8: Genotype-phenotype mapping, step 8

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 $<A>\ ::=\ a\,b\,<C>$.

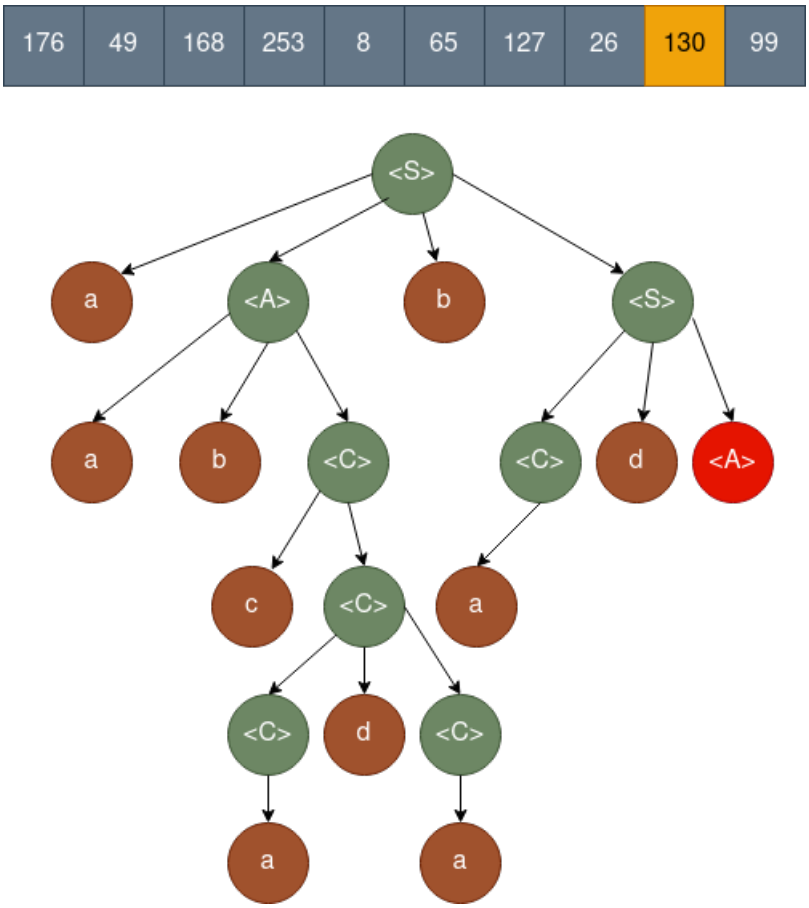

Figure 2.9: Genotype-phenotype mapping, step 9

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 99. The leftmost nonterminal symbol is $<C>$, which has 3 productions defined. So we calculate $99\ MOD\ 3\ =\ 0$, and apply the production at index 0, which is $<C>\ ::=\ c\ <C>$.

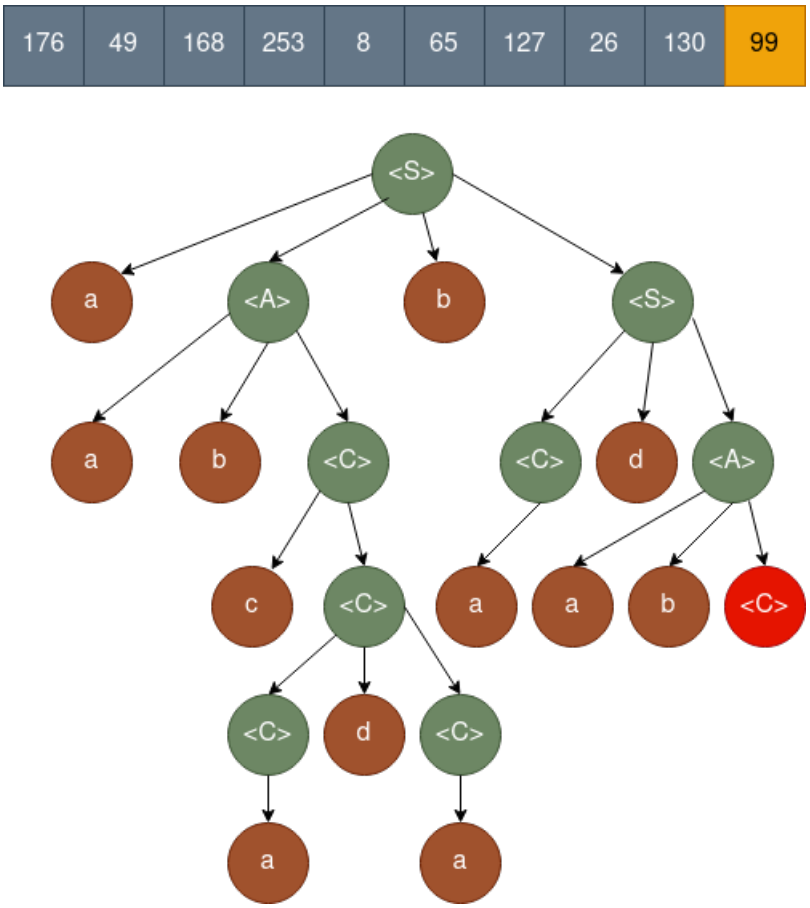

Figure 2.10: Genotype-phenotype mapping, step 10

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 $<C>\ ::=\ a$.

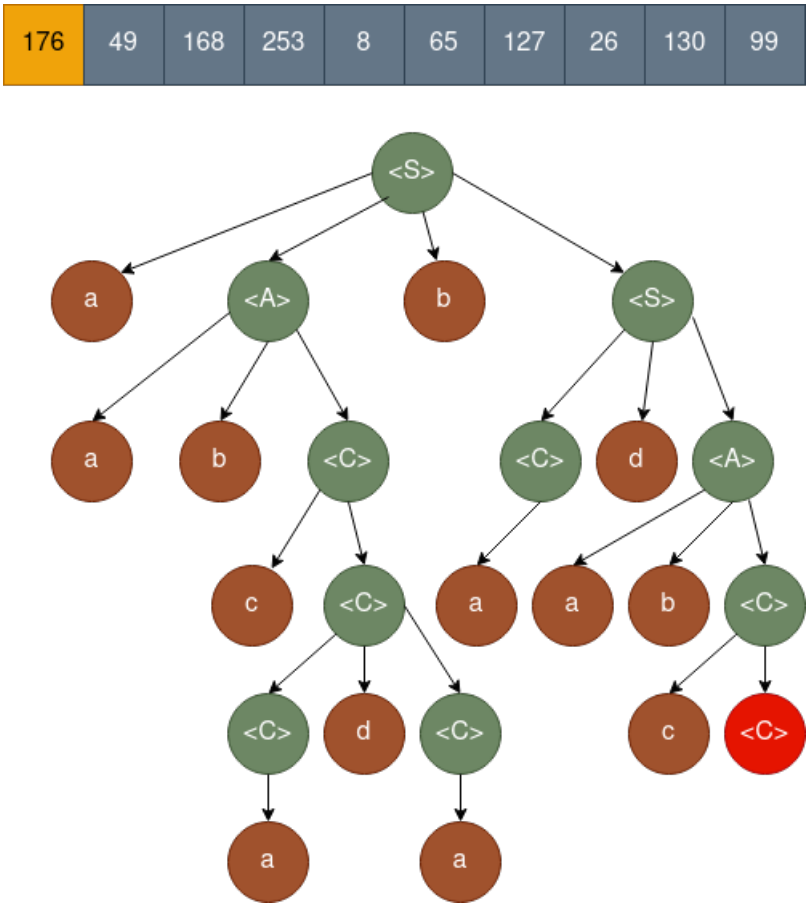

Figure 2.11: Genotype-phenotype mapping, step 11

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.

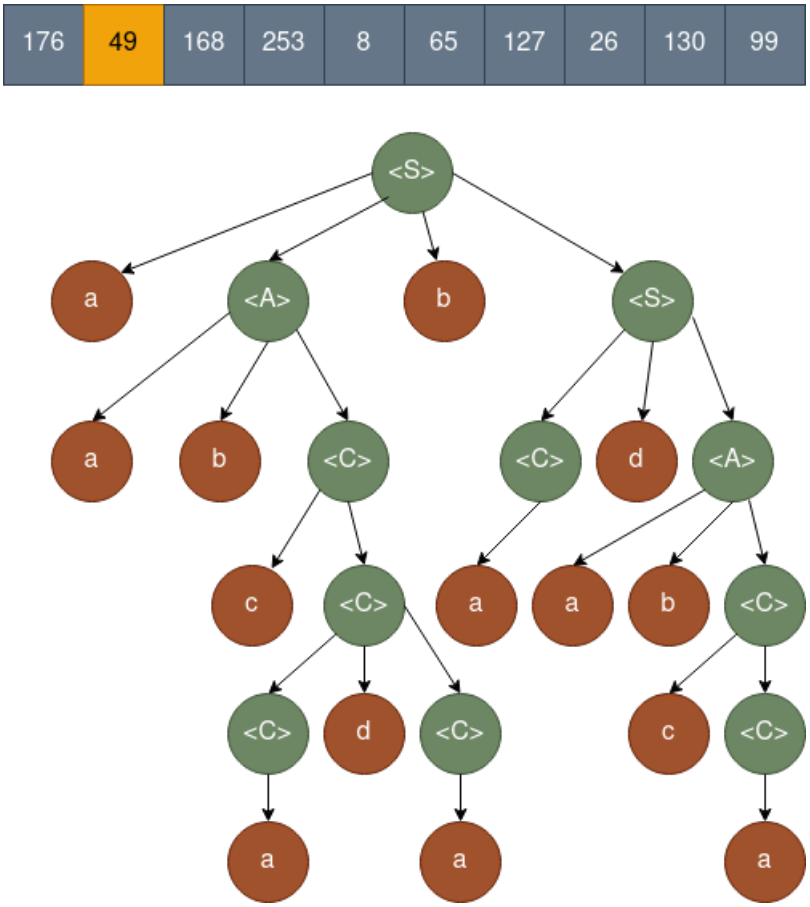

Figure 2.12: Genotype-phenotype mapping, step 12

3. Evolving cache replacement policies

3.1. Cache replacement policies

3.1.1. Problem definition

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 [10]. 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, its 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 removed from a frame, in order to make room for another page. This process is called swapping. The central question here is: which page should be removed? There are several strategies for this problem, and they are called cache replacement policies.

3.1.2. Popular heuristic strategies

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

The LRU (Least Recently Used) strategy removes the page which wasn't used for the longest time in the past. This strategy relies on the principle of 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 removes the page which was accessed the least number of times.

The CLOCK strategy, also called the second chance strategy, traverses the frames in a round robin way. Each frame has an access flag which can either have the value

zero or one. When a page needs to be removed, 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 its flag equal to zero, its page is removed and the search is finished. If, on the other hand, its flag is equal to one, the flag is set to zero, and the algorithm continues to the next frame, hence the 'second chance' name. Each time a frame is accessed, its flag is set to one.

The OPT (optimal) strategy removes the page that won't be used for the longest time in the future. This strategy isn't achievable because it requires the knowledge of the future page requests. It's significance is theoretical, because it can set an upper limit to what is possible to achieve.

3.1.3. Cache replacement policy pseudocode

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 for
```

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 he 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 of the 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

3.2.1. Introduction to the grammar design

Before listing and describing all the productions of our grammar, 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 architecture of the built software system is described in detail in the Appendix A.

3.2.2. Nonterminal symbols

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.

3.2.3. Information stored by each strategy

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 generated cache replacement policies will not manipulate these arrays directly, instead, each array will have some functions which are used to access it. In this sense, the functions used to access the information arrays can be considered as features of our learning problem.

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 frame which holds 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 frame which holds 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, its value is set to one. This array was introduced to make the implementation of the CLOCK strategy possible, as will be explained in the subchapter 3.3. The unique feature of this array is that the generated strategies can change its contents, unlike the other arrays which are only changed automatically and the generated stragies can only read their contents.

Each strategy will also be given some arrays and variables which aren't updated automatically, instead, the strategy can use them to store whatever information it chooses.

Each strategy will have two of these arrays, and they are accessed using the $read$ and $write$ functions. The $read$ function takes two arguments, the first being the index of the array, and the second being the index (the frame number) in that array, and this

function reads the value in that array at that index and returns it. The $write$ function takes three arguments, the first being the index of the array, the second being the index (the frame number) in that array, and the third being the value that is being written at that index in that array.

Next to these two arrays, each strategy will also have three free variables which can they can change however they please, and these variables are called $num1$, $num2$ and $num3$.

3.2.4. Symbols $<block>$, $<block\_no\_loop>$

Productions for the $<block>$ symbol are:

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

| ""

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:

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

| ""

3.2.5. Symbols $<statement>$, $<statement\_no\_loop>$

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 the equality operator. Finally, the fifth production generates calls of the $write$ function. This function is explained in the subchapter 3.2.3.

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

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

| if ($<expression>$) { $<block\_no\_loop>$ }

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

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

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

3.2.6. Symbol $<expression>$

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.

3.2.7. Symbol $<term>$

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.

3.2.8. Symbol $<modifiable>$

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 its initial value of zero, and that's why this variable will be modifiable. The $num1$, $num2$ and $num3$ variables are the free variables of the each strategy, and they are explained in the subchapter 3.2.3.

3.2.9. Symbol $<non\_modifiable>$

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. The rest of the productions are explained in the subchapters 3.2.2. and 3.2.3.

3.2.10. Symbol $<bool>$

Productions for the $<bool>$ symbol are:
$<bool>$ ::= true
| false

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

3.2.11. Symbol $<info\_field\_index>$

Productions for the $<info\_field\_index>$ symbol are:
$<info\_field\_index>$ ::= 1
| 2

As it was already explained, 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.

3.2.12. Symbols $<number>$, $<first\_digit>$, $<tail\_digits>$, $<digit>$

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

A 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 in the subchapter 3.1.2., and show 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:

$$\begin{aligned}
&iterations = 0;\\
&while\ (get_accessed(num1)\ ==\ 1\ \&\&\ iterations\ <\ frame_count)\ \{\\
&\quad set_accessed(num1, 0);\\
&\quad num1 = num1 + 1;\\
&\quad if\ (num1 >= frame_count)\ \{\\
&\quad\quad num1 = 0;\\
&\}\\
&frame = num1;
\end{aligned}$$

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

4.1. Experimental settings

The data used for generating and testing cache replacement policies was acquired from [2]. The program used to simulate cache replacement policies expects a linear array of integers as input data, where each integer represents a different page. So the data from the trace files was prepared in the following way: each page was assigned a number, starting from 0 and increasing by 1 for each first appearance of a page. Subsequent appearances of the same page were referenced with its original assigned number.

The fitness function which will be used in all the runs is simply the number of hits the strategy gets over a run.

The algorithm parameters are: selection operator, mutation rate, population size, codon count and maximum number of wrappings.

The only problem parameter is the cache size, which is the number of frames in the cache. This number will be varied to determine how well grammatical evolution performs on different cache sizes.

The two variants of the genetic algorithm that we will test, which correspond to the two types of the selection operator, are the steady-state algorithm with tournament selection and the generational algorithm with fitness-proportional selection and the elitism count of one. The number one was chosen for the elitism count since the goal of this parameter is only to preserve the best solution between generations. We still want to search the solution space as much as possible.

To avoid the combinatorial explosion of testing the Cartesian product of different values of the algorithm parameters, each parameter will be optimised one-dimensionally. This means that we will choose some average values for all the parameters, sort the algorithm parameters by our estimate of their importance from the most important to the least important, and then optimize them one by one. The problem parameter, which is the cache size, will also be set to an average, non-extreme value.

The parameters will be tested in the following order: selection operator (which is

expected to have a tremendous impact on the final results), mutation rate, population size, codon count and maximum number of wrappings (which is expected to have a minimal impact on the final results).

The trace file used for parameter optimization was the 'swim.trace' file, and the data used for parameter optimization consisted of the first 20000 requests from the trace file. Each page index was modulated by 1000. The reason behind this decision was the goal to provide a unique setting for generating and testing our strategies, because it felt natural to train them and test them using the data with the same number of different pages. Retrospectively speaking, tampering with the trace file data in this way may have turned out to be a bad idea and may have foiled the parameter optimization process, so this decision was abandoned once the actual experiments began. Still, I believe that the parameter optimization wasn't completely futile, and that it still merited some valuable information about good algorithm parameter values.

Since the genetic algorithm is inherently stochastic, each experiment will be repeated 10 times, and the median value of these runs will be used as the final result. If the median value turns out to be a bad measure for comparing different parameter configurations (for example, if the median values of multiple configurations are the same), then the average value will be used instead.

Once all the parameters have been optimized, the algorithm will be run using these parameters on different cache sizes, and the generated strategies will be compared to the classical heuristic strategies for cache management.

All the diagrams included in this chapter were generated using the [1].

4.2. Parameter optimization

Before starting the experiments, some average non-extreme values for the parameters had to be chosen. The frame count was set to 100, the mutation rate was set to 1%, the population size was set to 30, the codon count was set to 20 and the maximum number of wrappings was set to 3. The starting number of generations for a generational algorithm is 50, while the number of generations for a steady-state algorithm is 500.

The fitness values represent the number of hits during the simulation of a strategy using a dataset which contains 20000 requests.

4.2.1. Selection operator

The first parameter that was optimized was the selection operator. The first selection operator that was considered was the fitness proportional roulette-wheel selection in a generational genetic algorithm, and the second selection operator that was considered was the tournament selection in a steady-state genetic algorithm.

The median fitness in both of these runs was 1907, so the average fitness was used for analysis.

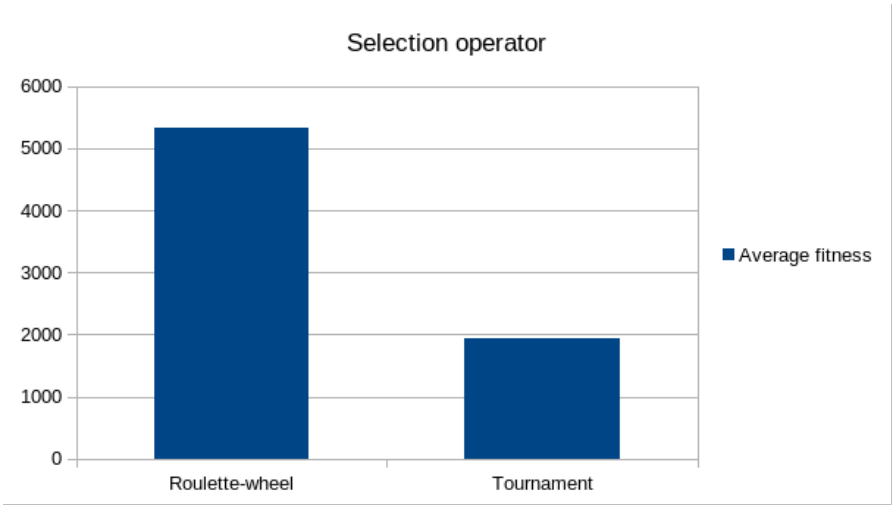

Figure 4.1: Results of optimizing the selection operator

The roulette-wheel selection had the average fitness of 5325.4, while the tournament selection had the average fitness of 1935.6. The roulette-wheel selection was chosen as the optimal selection operator.

Running 10 experiments using the roulette-wheel selection took about 25 minutes, while running 10 experiments with tournament selection tok about 60 minutes. I believe that both selection operators were given an equal chance, and the roulette-wheel selection cleary emerged as superior.

4.2.2. Mutation rate

The second parameter that was optimized was the mutation rate. Mutation rate was initially set to 1%, which was shown to be relatively small during the runs comparing the selection operators. So, the mutation rates tested were 1%, 2%, 5%, 10% and 20%.

All of these runs had the median fitness of 18999, so the average fitness was used for analysis.

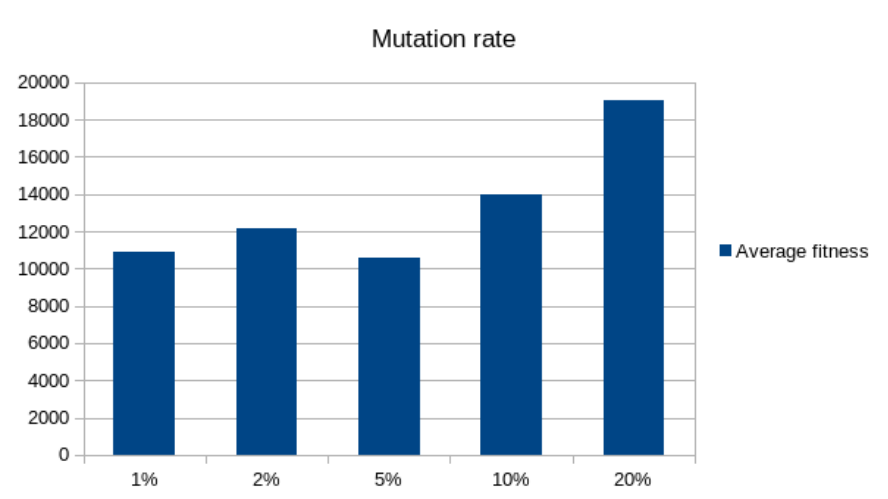

Figure 4.2: Results of optimizing the mutation rate

The mutation rate of 20% was chosen as optimal the mutation rate.

4.2.3. Population size

The third parameter that was optimized was the population size. The values that were compared during these runs were the sizes of 10, 30 and 50.

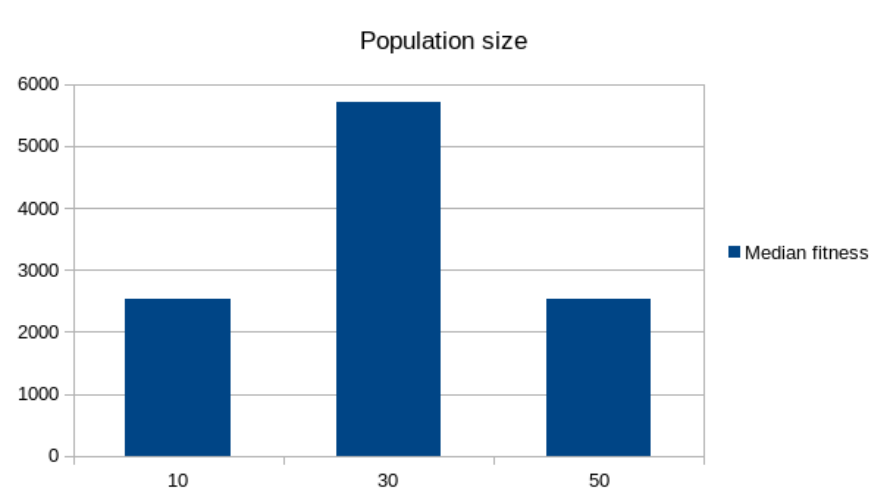

Figure 4.3: Results of optimizing the population size

The population size of 30 was chosen as the optimal population size.

4.2.4. Codon count

The fourth parameter that was optimized was the codon count. The values that were compared during these runs were the codon counts of 10, 20 and 30.

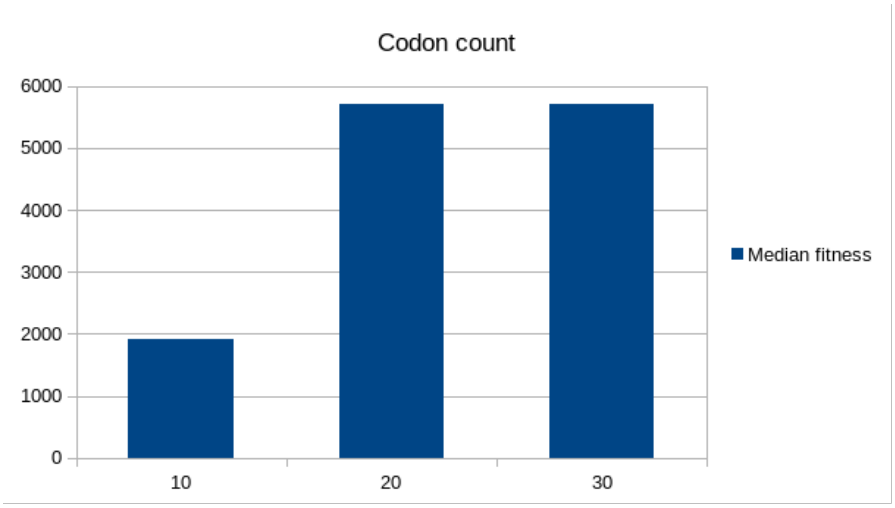

Figure 4.4: Results of optimizing the population size

Both the runs with the codon count set to 20 and 30 has the median fitness of 5713. The run where the codon count was set to 20 had the average fitness of 9642.3, and the run where the codon coun was set to 30 had the average fitness of 9243.7. Since the first run had higher average fitness, the value 20 was chosen as the optimal codon count. I suspect that the codon count has little significance to the result.

4.2.5. Maximum number of wrappings

The fifth and final parameter that was optimized was the maximum number of wrappings. The values that were compared during these runs were 0, 3 and 5.

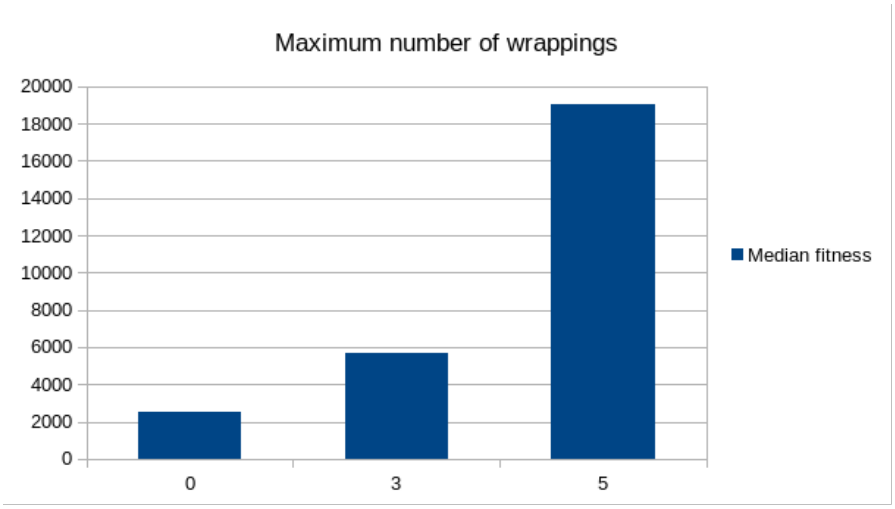

Figure 4.5: Results of optimizing the maximum number of wrappings

The value 5 was chosen as the optimal value for the maximum number of wrappings.

4.3. Results

Once the actual experiments began, several changes had to be made to the configurations of the runs, because the generated strategies rapidly overfitted to the train data.

First of all, a different trace file was chosen for the subsequent runs. The chosen trace file was the 'crafty_mem.trace'. The first 20000 requests were used as the train dataset, and the subsequent 200000 requests were used as the test dataset. The train dataset contained 1181 unique page requests, and the test dataset contained 2646 unique page requests. This trace file was chosen because it contained fewer unique page requests and more repetition that the 'swim.trace' file, so it seemed like a good fit for evolving cache replacement policies.

Secondly, some parameters of the runs were changes. Population size was increased to 200, and each run consisted of 200 generations. These changes were implemented to cultivate the exploration of the search space. Mutation rate was decreased from 20% to 5%, as the rate of 20% seemed very disruptive.

Another problem that emerged was that the strategies did not evolve at all, since the strategy in which they do nothing and the page from the frame at index 0 is always removed worked relatively well. This was the default strategy since the index of the frame from which the page would be removed was a program variable that was initialized to 0. There was a similar problem where the evolution would find clearly

nonsensical and unusable strategies. Two of the most popular strategies that fall into this category were the strategy to choose the frame that corresponds to the time variable and the strategy to choose the frame at the index which is equal to the index of the requested page. A solution to this problem was checking if the strategy doesn't set the frame variable, or if it sets it to the time variable or the requested page index variable, and if some of these cases were true, the strategy would be penalized with the fitness of zero.

Finally, while loops were removed from the grammar for these runs. The reason behind this decision was that the strategies found it very hard to use them in a smart way (like the CLOCK strategy use a while loop), and they drastically increased the simulation times, to the point where it would be impossible to run all the simulations in an acceptable timespan. The most common problem with while loops was that the strategies would leave their bodies empty or fill them with nonsensical instructions, and this would be calculated for every frame (since the maximum number of iterations is equal to frame count), for every page request in the data set, for every strategy that used while loops, for every generation, for every run of the experiment.

Four different experiments were run, each with a different cache size. For each experiment, a bar plot will be included, showing the hit counts of the strategies OPT, FIFO, CLOCK, LRU, LFU, as well as three chosen strategies generated by the grammatical evolution, and they will be marked as GE1, GE2, GE3. Each experiment was run 10 times, and each run had population of 200, so for every experiment, 2000 strategies from the last generations were tested using the test data. There was a lot of repetition in the generated strategies and the hit counts, so the values for the strategies GE1, GE2 and GE3 were chosen as the three best unique scored hit counts.

4.3.1. Experiment 1: frame count 100

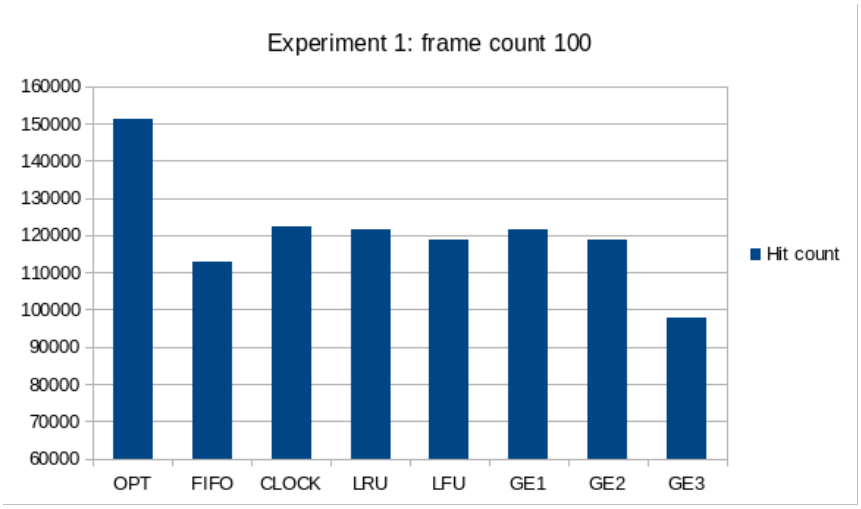

Figure 4.6: Results of experiment with the frame count set to 100

4.3.2. Experiment 2: frame count 200

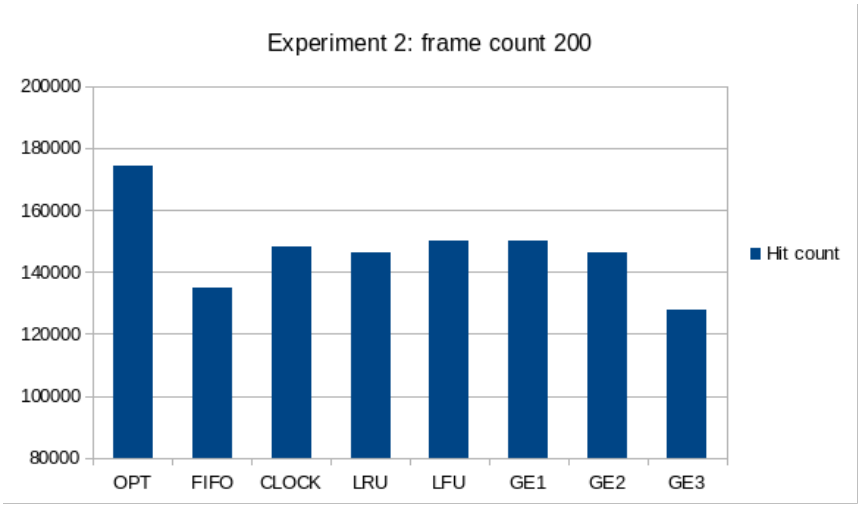

Figure 4.7: Results of experiment with the frame count set to 200

4.3.3. Experiment 3: frame count 300

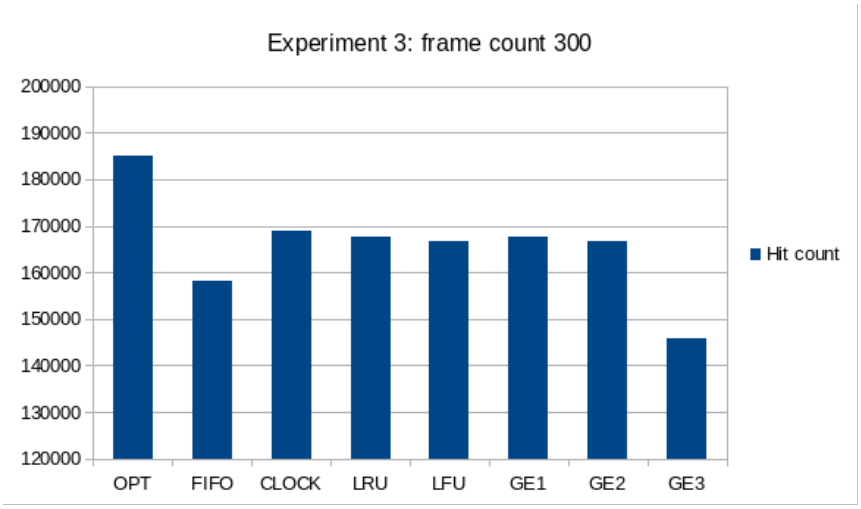

Figure 4.8: Results of experiment with the frame count set to 300

4.3.4. Experiment 4: frame count 500

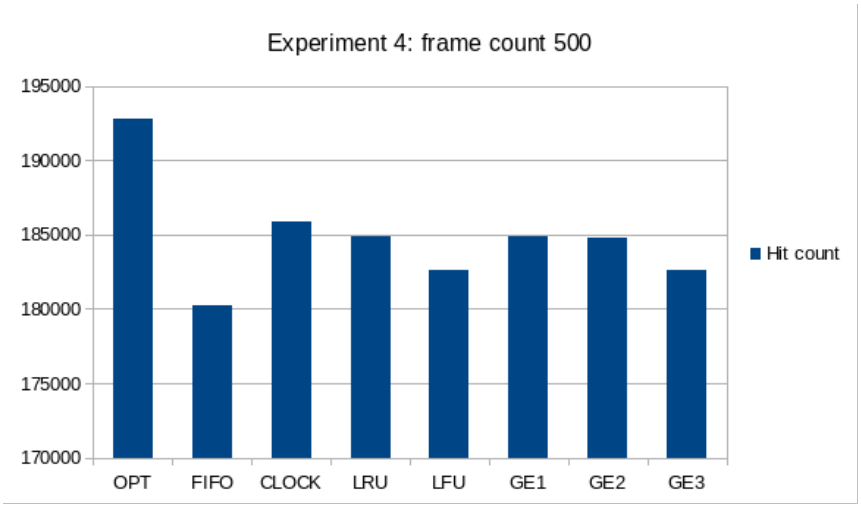

Figure 4.9: Results of experiment with the frame count set to 500

4.3.5. Results analysis

During all the runs, the best strategies that the evolutionary search managed to find were equivalent to one of the heuristic strategies, namely the LRU and LFU strategies. Unfortunately, the generated strategies never managed to learn how to use the info arrays or develop any complex behaviour, so it seems that expecting the genetic

algorithm to develop and learn such behaviours may have been too ambitious. Nevertheless, the algorithm always converged and resulted in strategies that can compare to classic heuristic approaches.

One interesing property that the computer programs written by humans usually possess, and the computer programs generated by genetic programming techniques usually don't possess, is the property of parsimony. If something is parsimonius, it is as simple as it can be while performing its function, and this principle is often called Occam's razor. Like the genome of living beings, programs generated by genetic programming are rarely minimal structures for performing their tasks, instead, they are packed with unused substructures that usually reflect their evolutionary history, not their functionality [6].

Programs generated by genetic programming usually contain parts which aren't used for anything, or expressions which are much more complicated than they need to be. As humans, we prefer computer programs which are easy to read and not more complex than they need to be, so if genetic programming is used to generate programs for a certain task, it would be a good idea to have a human manually inspect the generated programs, or to have them inspected by a code analysis software.

4.3.6. Examples of generated strategies

The following strategy is a good example of non-parsimonious code. This strategy is equivalent to the LFU strategy, but it sets the $accessed$ info array, even though it never uses that information. The strategy would be functionally identical if we were to remove that unnecessary code.

$frame = page\_access\_count\_min();$
$set\_accessed(num1! = num3, frame);$

Another interesting example is the following generated strategy. This strategy is also equivalent to the LFU strategy, even though the $num3$ variable is being added to the $frame$ variable, because the $num3$ variable is initialized to zero and never changed, so the strategy would be functionally the same if there was no addition.

$frame = page\_access\_count\_min() + num3;$

The next strategy is really interesting. It is almost identical to to the LRU strategy, except it uses the information about the least recently used frame that was accurate one request ago, and now it's slightly outdated. This strategy was 0.03% worse than the LRU strategy during the experiments with the frame count of 500.

$$frame = num2;$$
$$num2 = last\_accessed\_min();$$

The generated strategies were sometimes really long and unreadable. An example of this is the following strategy.

$$write(1, page\_access\_count\_min(), num3 <= num1);$$
$$if(find\_max(2) * page\_access\_count\_min() < time)$$
$$\{if(num1 <= frame == time)\{\}\}else\{\}set\_accessed(num1, frame);$$
$$if(frame)\{write(1, frame, page\_access\_count\_min());\}$$
$$else\{set\_accessed(num3, find\_max(2)*page\_access\_count\_min() < time >= frame); num1 = frame == time;\}$$

The next strategy is the final strategy we're going to discuss, and it's interesting for two reasons. First, we can notice that it has an expression which looks like $x - false - y$, which is functionally equivalent to $x - y$. Secondly, we can notice that the whole expression is actually a comparison using the $>$ operator. The whole expression can be evaluated either to 1 if it's true, or to 0 if it's false, so this strategy can remove pages only from its first two frames.

$$frame = division(frame <= page\_access\_count\_min() - false - num1 >= num2 == last\_accessed\_max() - frame > frame >= frame, time) > page\_access\_count\_min();$$

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 software system capable of evolving valid computer programs in an arbitrary language which solve an arbitrary problem, use the built software 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 the classic 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 for 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

[1] LibreOffice Calc. https://www.libreoffice.org/discover/calc/. Diagrams generated: 2022-06-07.

[2] Oberlin College Computer Science, Course Computer Organization, Homework Assignment 6. https://occs.oberlin.edu/~ctaylor/classes/210SP13/cache.html. Accessed: 2022-06-05.

[3] diagrams.net. https://app.diagrams.net/. Accessed: 2022-06-03.

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

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

[6] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[7] Nuno Lourenço, Filipe Assunção, Francisco B. Pereira, Ernesto Costa, and Penousal Machado. Structured Grammatical Evolution: A Dynamic Approach. 2018.

[8] Jessica Mégane, Nuno Lourenço, and Penousal Machado. Co-evolutionary Probabilistic Structured Grammatical Evolution. 2022.

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

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

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

[12] Marko Čupić. *Umjetna inteligencija: Evolucijsko računarstvo*. 2019.

List of Figures

List of Algorithms

Appendix A
Software architecture

For the purposes of writing this thesis, an implementation of the grammatical evolution algorithm was developed using the object-oriented paradigm in the C++ language. This system can be used to generate and evolve programs in any arbitrary language, used to solve any arbitrary problem.

The classes written for this generic part of the implementation are:

- **GrammaticalEvolution** - the top class in the hierarchy, used to encapsulate the process of mapping a unit's genotype onto its phenotype
- **Grammar** - the class used for reading and parsing the .bnf files, and also storing information about a grammar and its productions
- **Crossover** - an implementation of the crossover operator
- **Mutation** - an implementation of the mutation operator
- **Unit** - the class used for storing information about a unit
- **Node** - the process of the genotype-phenotype mapping is implemented using linked lists, and this class encapsulates nodes in this list
- **Symbol** - a class used in the genotype-phenotype mapping, stores information about the contents of each node in the linked list
- **GrammarParsingState** - the process of reading and parsing the .bnf file is implemented as a final automata, and this enumeration stores all the states of that automata
- **SymbolType** - an enumeration which stores all types which an instance of the class Symbol can have, and the types are terminal and nonterminal

- **DecodeException** - a class used for handling all potential problems during the process of parsing a .bnf file

The other part of the implementation is the classes used for the purpose of evolving cache replacement policies. These classes are:

- **Strategy** - the base class for a cache replacement strategy
- **GEStrategy** - the class used for the strategies generated by the GrammaticalEvolution
- **GeneratedStrategies** - the class which stores all generated strategies in one place; this class has to be recompiled during each iteration of the genetic algorithm
- **FIFO** - an implementation of the FIFO strategy
- **LRU** - an implementation of the LRU strategy
- **LFU** - an implementation of the LFU strategy
- **CLOCK** - an implementation of the CLOCK strategy
- **OPT** - an implementation of the OPT strategy

For the sake of simplicity and the adherence to the 'separation of concerns' principle, the main function was split into six files, each with its own main function. These files are:

- **init** - initializes a random population and stores each unit's genotype in a separate file inside the /solutions directory
- **decode** - reads the genotypes of every unit inside the current population and decodes their genotypes to their phenotypes, which are then stored inside the GeneratedStrategies
- **run** - tests all strategies from the GeneratedStrategies using the train data, and stores their scores inside the /results directory
- **evolution** - reads each strategy's score and performs the evolutionary process of creating a new generation, which is then stored inside the /solutions directory

- **test_generated_strategy** - simulates a generated strategy using the test data and outputs its hit count
- **test_heuristic_strategies** - simulates the heuristic stragies using the test data and outputs their hit counts

During each iteration of the evolutionary process, new strategies are generated and stored as C++ files, which need to be compiled before they can be run. The class responsible for decoding the genotype of each unit, which is an array of 8-bit numbers, to its corresponding phenotype, which is a computer program, is the GrammaticalEvolution class. This class has a $decode$ method which takes as input some Unit and returns its phenotype representation as a string. This string needs to be properly placed into a specific file, so that it can be correctly compiled and run. This is the responsibility of the main loop of the program. This main loop isn't written as a C++ file, instead it is a bash script. This bash script manages and calls the six files which do a specific part of the evolutionary process, recompiles the parts of the code that change during each iteration, and assures the persistency of the generated strategies between each run. Each strategy has a genotype, phenotype and score, and each of these is stored inside a different textual file.

Figure A.1 shows a class diagram of the complete project. The classes which belong to the core part of the project and which can be applied to any problem are colored in blue. The classes which are specifically built for the problem of evolving cache replacement strategies are colored in green. These classes are shown without their member variables and functions since these are unimportant in viewing the whole project and the connections between its parts. Finally, the six C++ files which are being compiled to executable programs and subsequently run by the main bash script are colored in yellow. These aren't classes so they technically don't belong in a class diagram, but they were included for the sake of showing all the built components and their connections. The diagram was built using the tool [3].

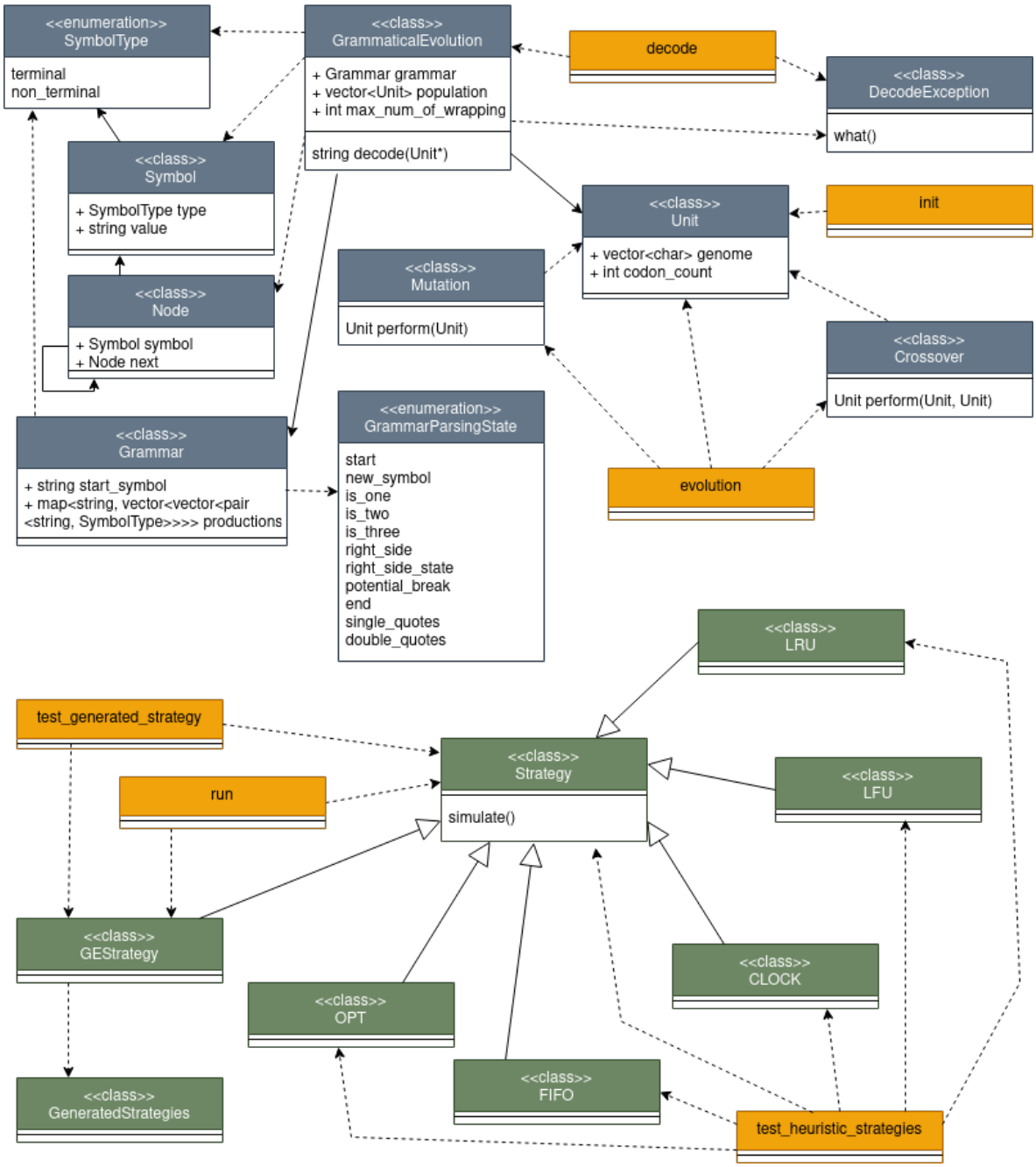

Figure A.1: Class diagram

Evolving cache replacement policies using grammatical evolution

Abstract

Genetic programming is a branch of evolutionary computing in which the population consists of computer programs. Grammatical evolution is a popular grammar-based approach to genetic programming, capable of evolving programs in an arbitrary language, using fixed-length arrays of integers as genotypes, a context-free grammar for the definition of the language, and a genotype-phenotype mapping which uses the specified grammar to map integer arrays to computer programs. Evolving cache replacement policies is a popular benchmark problem in the field of genetic programming. The topic of this thesis is using grammatical evolution to evolve cache replacement policies. Generated strategies are compared to classic heuristic approaches to cache management, including FIFO, LRU and CLOCK algorithms.

Keywords: evolutionary computing, genetic programming, genetic algorithm, grammatical evolution, cache replacement policies, optimization, formal grammars, ECF

Evolucija strategija zamjene stranica korištenjem gramatičke evolucije

Sažetak

Genetsko programiranje je grana evolucijskog računarstva u kojoj populaciju čine računalni programi. Gramatička evolucija je popularan pristup genetskom programiranju koji se bazira na formalnim gramatikama, sposoban evoluirati programe u proizvoljnom jeziku, koristeći polja cijelih brojeva fiksne duljine za genotipe, kontekstno neovisnu gramatiku za definiciju jezika, i mapiranje s genotipa na fenotip koje koristi definiranu gramatiku za preslikavanje s polja cijelih brojeva na računalne programe. Evolucija strategija zamjene stranica je popularan problem u području genetskog programiranja. Tema ovog rada je korištenje gramatičke evolucije za evoluiranje strategija zamjene stranica. Generirane strategije su uspoređene s klasičnim heurističkim pristupima za upravljanje memorijom, uključujući algoritme FIFO, LRU i CLOCK.

Ključne riječi: evolucijsko računarstvo, genetsko programmiranje, genetski algoritam, gramatička evolucija, strategije zamjene stranica, optimizacija, formalne gramatike, ECF