VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

CARTESIAN GENETIC PROGRAMMING IN PYTHON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE PETR DVOŘÁČEK
AUTHOR

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KARTÉZSKÉ GENETICKÉ PROGRAMOVÁNÍ V JAZYCE PYTHON

CARTESIAN GENETIC PROGRAMMING IN PYTHON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE PETR DVOŘÁČEK
AUTHOR

VEDOUCÍ PRÁCE Ing. ZDENĚK VAŠÍČEK, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Kartézske genetické programování (CGP) patří mezi evoluční algoritmy. Byl primárně vytvořen pro návrhu kombinačních obvodů. Dále může být použit k optimalizaci funkcí, v klasifikaci, evolučním umění atd. Tato práce se zabývá akceleračními technikami urychlující výpočet kandidátního řešení CGP v jazyce Python.

Abstract

Cartesian genetic programming (CGP) is one of the evolutionary methods. It was created for electronic circuit design. It can be used also in optimization of functions, classification, evolutionary art etc. This paper describes acceleration techniques to speed up the evaluation of candidate solution in CGP in Python.

Klíčová slova

Kartézské genetické programování, genetické programování, Python, akcelerace, kompilace za běhu programu, výpočet fitness funkce, symbolická regrese

Keywords

Cartesian genetic programming, genetic programming, Python, acceleration, just-in-time compilation, fitness evaluation, symbolic regression

Citace

Petr Dvořáček: Cartesian genetic programming in Python, bakalářská práce, Brno, FIT VUT v Brně, 2013

Cartesian genetic programming in Python

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Zdeňka Vašíčka.

....
Petr Dvořáček
January 9, 2017

Poděkování

Chtěl bych poděkovat Zdeňku Vašíčkovi za jeho konzultace a rady spojené s tvorbou této práce.

Contents

Chapter 1

Introduction

Genetic programming allows us to generate whole programs. It was designed by John Koza for solving symbolic regression which was represented by syntactic trees.

Cartesian Genetic Programming (acronym CGP) is a type of genetic algorithm. CGP was proposed by Julian Miller and Peter Thomson in 1999 for combinational circuit design for which the previous methods weren't effective. Moreover, we can utilize CGP in design of image filters, classification, optimization of equations or even in evolutionary art [7].

The candidate solution in CGP is represented as an acyclic oriented graph. However, the evaluation of candidate solution is computationally hard process and it can take a lot of time. Therefore the goal of this thesis is to accelerate the evaluation in circuit design and in symbolic regression. CGP algorithm is explained in chapter 2.

For the acceleration, I selected a compilation of the heuristic function which evaluates the candidate solutions. The heuristic function is the most frequently called function of the whole algorithm. The optimization method is so called Just-in-time (acronym JIT) compilation into bytecode of the interpreter (or it can be even compiled in operation code of the processor). This problematic is described in chapter 3.2.

Python language is the language with dynamic type system. This fact results that the algorithms in this language are usually slower than in the compiled language such as C. However, Cython language extends Python language. Cython contains static data types and it supports calling function from C language. With Cython language, we can easily develop effective modules for Python. In chapter 3.4 is described the Cython language.

In chapter 4 I present implementation of used acceleration methods and in chapter 5, we can see their results.

Chapter 2

Cartesian Genetic Programming

Cartesian Genetic Programming is classified as an evolutionary algorithm. These algorithms are based from Darwin's evolution theory or newer evolution theories. According to Ch. R. Darwin, the main force of evolution is natural selection of individuals which differ from each other. Natural selection favours the ones with better genes which increase the probability of survivable. The bad genes decrease the probability. The most adapted individuals breed and they pass their genetic information more frequently. [5]

2.1 Evolutionary Algorithms

Evolutionary algorithm works as follows. In the beginning, the initial population is created. The set contains the specific amount of candidate solutions. The initial population can be created randomly or we can utilize the known solution of the problem. The offspring set is created by application of crossover and mutation on selected individuals so called the parents set. The new population is created by selection of candidates from the previous population and from the offspring set. The selection proceeds as follows. Firstly, we evaluate each candidate solution by the heuristic function (fitness function). The fitness value gives us the quality of the solution. In the selection, we don't select the optimal solutions only. We may also add individuals of poor quality in order to not stuck in a local extreme. Algorithm is ended after finding the sufficient optimal individual [9].

The algorithm can look as follows:

Algoritmus 1: *Evolutionary Algorithm*

```
$t = 0$ {generation}
$P(t)$ = initialize_population()
evaluate $P(t)$
**while** found_the_sufficient == FALSE **do**
  $Q(t)$ = parent_selection($P(t)$)
  $Q'(t)$ = offspring_creation($Q(t)$)
  evaluate $Q'(t)$
  $P(t+1)$ = new_population_selection($P(t), Q'(t)$)
  $t = t + 1$
**end while**
```

2.1.1 Genetic Programming

Genetic programming (GP) is one of the variant of evolutionary algorithms. The main principle of genetic programming is the development of computational structures such as mathematical equations, digital circuits or computer programs. The computational structures are represented by abstract syntactic trees and these trees are called chromosomes[1]. These structures have variable length. Thus, it may overgrowth in some cases. This phenomenon is called *bloat* [7].

2.2 Principle of Cartesian Genetic Programming

Cartesian Genetic Programming tries to evade the bloat. Chromosomes in CGP are modelled as an acyclic oriented graph. Each node of the graph represents a specific function. In the beginning of the evolution, we determinate parameters of the graph and its nodes. We define the amount of primary inputs of the graph n_i, the amount of primary outputs of the graph n_o, the number of the graph columns n_c and the number of the graph rows n_r. Moreover, we have to specify the L-back parameter L which defines the rate of interconnection between columns of the graph. If $L = 1$, then the interconnection is minimal because the node can be connected to nodes from the neighbouring columns. If $L = n_c$, then the interconnection is maximal because the node can be connected to any node. Each node can be also connected to any primary input or any primary output. Furthermore, it is necessary to define the set of operations Γ. From this set, the node operation is assigned. Thus we have to also define the maximal arity of operation n_a (the maximal number of inputs to each node).

The interconnection is encoded by a chromosome of **constant length**. This is the main reason why bloat phenomenon doesn't occur in CGP. The length is given by equation:

$$G = n_c * n_r * (n_a + 1) + n_o$$

The principle of encoding is as follows. The index from interval $0, \ldots, i-1$ is assigned to each primary input. Then the index is assigned to each node in column by column. So the top-leftmost node has value i. The node below has number $i+1$. Then each node is encoded by $n+1$ numerical values. The first n values contains indexes of previous nodes. In other words, the first n values defines the connection to previous nodes. The last value of the tuple denotes the operation of the node. Then we add n_o indexes at the end of the chromosome. These indexes defines the nodes with the primary outputs.

Thus the redundancy of nodes arises because of the constant length of the chromosome. This means that some nodes do not have to be used in final solution at all. Moreover, the nodes do not have to use all own inputs. For example, let have $n_a \geq 2$, then the NOT gate utilize only one input. Also there can occur a situation when the group of nodes can be replaced by smaller group of the nodes. For example two nodes `NOT(AND(A, B))` can be replaced by one node `NAND(A, B)`. However this problem is solved by optimization.

2.2.1 Search algorithm

The goal of the evolutionary design is to find the chromosome which fulfil the user's specification. In case of evolutionary circuit design, the specification can be given by the truth

[1]Chromosome in genetics means something different. For genetic engineers, it is a structure in the cell nucleus, which contains DNA and protains. [5].

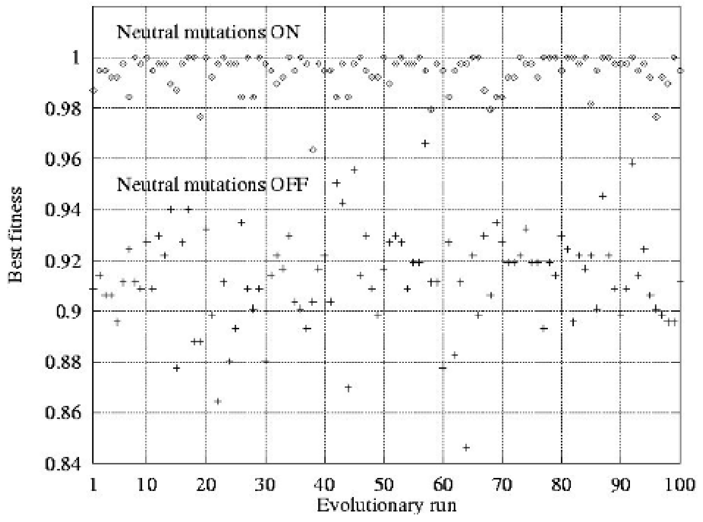


Figure 2.1: Neutral mutations in Cartesian Genetic Prgramming [7].
ON – The offspring of **the same or better fitness value** is selected as the parent.
OFF – The offspring of **the better fitness value** is selected as the parent.

table. The goal of the evolution is to find the circuit which works correctly.

The search algorithm is a variation of the evolutionary algorithm. Population consists of $1+\lambda$ individuals. The number 1 stands for one parent and λ is count of offspring. The parent selection slightly differs to evolutionary algorithms. In CGP we have only one parent thus, the most optimal individual is selected to be the parent. However, if there is an offspring with the same fitness value as parents have, then the offspring is selected as the parent. This means that we select individual in which neutral mutation happened. Therefore, we have more genetic diversity in term of generations. This leads to bigger probability to obtain the correct solution, this is pictured on figure 2.1. The evolution is ended by finding sufficient individual or by exhausting the number of the generations.

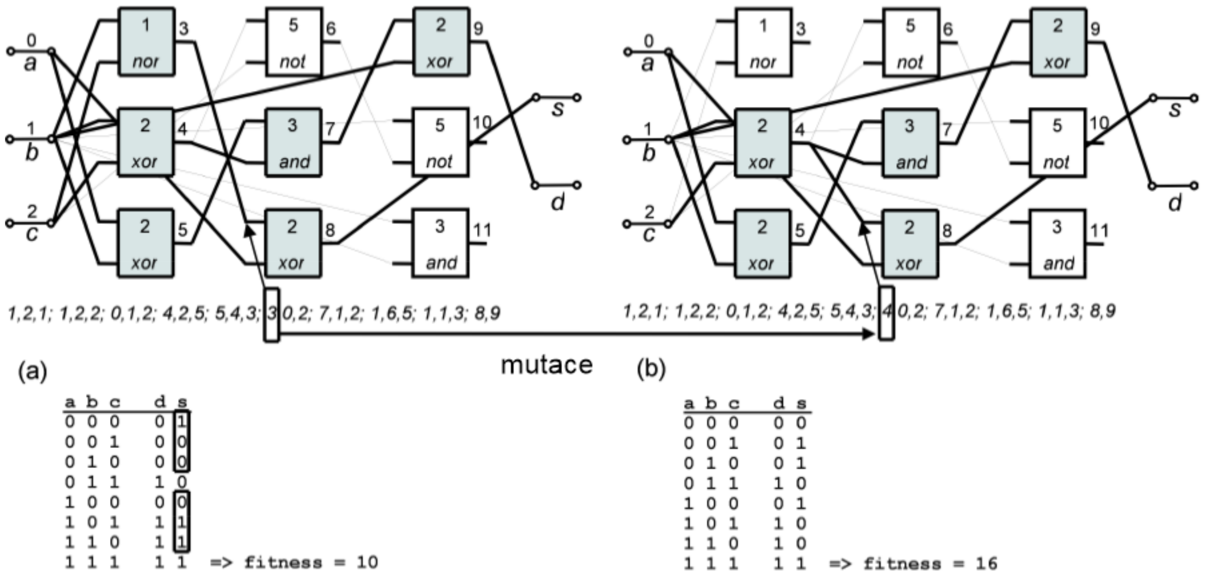
The evolutionary algorithm Cartesian Genetic Programming includes the following steps:

1. Initialization of population – creation of $1 + \lambda$ individuals.
2. Evaluation of the population by fitness function.
3. Selecting the parent – the individual with the best fitness value and checking its duplicity with the parent.
4. Cloning λ offspring and mutating them.
5. Go to step 2, if we haven't exhausted the number of generations yet.
6. The individual with the best fitness value is the result of the algorithm.

2.2.2 Implemenation of Mutation

During the evolution, the mutation changes the interconnectivity in the graph, the operations of the nodes or the primary inputs/outputs of the graph. By application of the mutation, the active node can be changed to inactive or vice versa. This situation is pictured on figure 2.2. The figure 2.2a shows the candidate solution and the chromosome before the mutation. The figure 2.2b shows the candidate solution and the chromosome before the mutation. We can see that the mutations have the significant effect to the final solution. Only a small change can lead to the solution with very high quality or even to very low quality. Mutation is neutral if it has no influence to fitness value. It may occur either if the mutation was applied to inactive nodes (white one in the image) or if we find the candidate solution with different phenotype (with the different connection of blue nudes) of same fitness value.

Common genetic programming also utilizes genetic operator called crossover. The crossover is not used in CGP. It leads to deceleration of the evolution and it also leads to worse solutions. [7]



1,2,1; 1,2,2; 0,1,2; 4,2,5; 5,4,3; 3 0,2; 7,1,2; 1,6,5; 1,1,3; 8,9

1,2,1; 1,2,2; 0,1,2; 4,2,5; 5,4,3; 4 0,2; 7,1,2; 1,6,5; 1,1,3; 8,9

```
a b c   d s
0 0 0   0 1
0 0 1   0 0
0 1 0   0 0
0 1 1   1 0
1 0 0   0 0
1 0 1   1 1
1 1 0   1 1
1 1 1   1 1  => fitness = 10
```

```
a b c   d s
0 0 0   0 0
0 0 1   0 1
0 1 0   0 1
0 1 1   1 0
1 0 0   0 1
1 0 1   1 0
1 1 0   1 0
1 1 1   1 1  => fitness = 16
```

Figure 2.2: Application of the mutation in CGP

2.2.3 Fitness Function

Fitness function controls the evolution algorithm. We have to assign fitness value to each candidate solution (chromosome). This value determines the difference between the candidate solution and the specification.

Digital Circuits Design

For combinational digital circuits, it means that fitness function evaluates the functionality of the circuit by testing all the possible input combinations. Fitness value of the specific candidate solution is either the number of correct responses to the specification or the number of incorrect responses to the specification. This depends on the implementation.

In the design of combinational digital circuits, we have to specify 2^k input vectors. Value k is the number of primary inputs of the circuit. Each candidate circuit is simulated to obtain its responses to each input vector. These responses are then compared with the specification. The fitness value corresponds to amount of the correct responses of the primary outputs of the cicuit for all of the input vectors. Let say, we have to design a circuit of $k = 4$ inputs and $l = 2$ outputs. Then the amount of input vectors is $2^k = 2^4 = 16$ and the maximal fitness function can be $l * 2^k = 2 * 16 = 32$.

Symbolic Regression

Symbolic regression tries to find such the mathematics function that it approximates the function f which is defined by the set of the input values in and by the set of expected responses. The fitness value is then calculated by formula

$$\sum_{i=0}^{m} |f(in_i) - out_i|$$

where m is the number of all imput combinations. Or we can use sum of quadratic deviation.

Evolutionary Art

In case of evolutionary art, the fitness function can be done by human. Man can subjectively select the best picture, therefore he or she gives the best fitness to the picture. In other cases of evolutionary art, the fitness function can be more complex [7].

2.2.4 Optimization

The optimization starts after the maximal fitness value is reached. In the other words, after the finding the correct solution. The most common optimization is reduction of the used nodes n_{used} in the graph. By reduction of the amount of used nodes, we can decrease the cost of the circuit and also we can decrease its power consumption. For the optimization, we used modified fitness function. Its formula is as follows:

$$fit = fit_{max} + n_r * n_c - n_{used}$$

However in some cases, it is better optimize hardware to its delay. The solution consisted by 42 gates can have delay of 10 gates and solution consisted by 39 gates can have larger delay of 12 gates. For these cases, we have to use multi-objective fitness function [9].

Chapter 3

Python and Acceleration of CGP

Python language [4] is one of the most known interpreted languages. The main advantage is the compatibility of the source code which is guaranteed by so called bytecode (or portable code). The another advantage of Python is its simple syntax, dynamic typing and easy JIT compilation.

Python has two development branches. The speed of algorithm depends on the interpreter of Python language. In some cases, Python2 is much faster than Python3. In section 3.3, I describe PyPy interpreter. It is an implementation of the interpreter that can be faster than the previously mentioned interpreters.

For programming modules, we can exploit Cython language. It is a language which contains static data types of variables and calling functions of C language.

3.1 Programming in Python

Python is easier to use than C language. Nevertheless, the final program can be slower. In this section, I focus on the effective coding in Python.

For Python, it is better to handle exceptions than the prevention. In the other words, `try: ... except: ...` is faster than the handling it via conditions [6]. Testing objects existence by `if object is None` is more effective than by `if object == None`. In the first case, the interpreter compares the addresses of the objects. In the second case, the interpreter compares their data [6]. String concatenation should be done by the method `join()` because it has linear time complexity. Operator `+` has quadratic time complexity. Furthermore, the interpreter has to check the data type of the objects [2]. In Python2, it is better to use function `xrange()` instead of `range()`. Function xrange creates an iterator / a generator. However `range()` allocates a list which can be used for the iteration. Thus, this function steals the computation resources. In Python3, the function `range()` creates the iterator (the generator) and the function `xrange()` was deleted [3]. Calling functions in Python is slow such as in other computer languages. It is better to roll up the functions in the loops.

3.2 The Acceleration Methods of CGP

By profiling, we find the slowest part of the algorithm. It is the evaluation of the candidate solutions in the case of CGP. For each solution, its simulation is necessary in order to obtain the response for all training data.

3.2.1 Parallel Simulation

In the case of digital circuits design, the time complexity has an exponential growth (as we add primary inputs k). It is needed to test the circuit response for each input combination in total 2^k cases. We can utilize so called parallel simulation where the input combinations are arranged into the vector of 32 or 64 bits. In the other words, the input combinations are arranged in integers. The bitwise operations are then used for the circuit simulation. Thus by one iteration, we can evaluate 32 or 64 cases of the fitness.

However in Python language, the integer has dynamic length. In the other words, Python can employ bitwise operation on the integers of 9000 bit length while using 32-bit computing machine. This feature has one disadvantage. The dynamic integer is allocated as an array and the bitwise operations can be slow.

Computation of the Difference Between Training Vector and Reference Vector

After the circuit simulation, we have to compare the response of candidate circuit with the target specification. This can be realized by calculation of Hamming distance between them. It can be realized by formula

$$fitness = zerocount(a \ \oplus \ b),$$

that a is the response of the candidate circuit and b is the target response. Exclusive disjunction $\oplus$ leads to zero on bit i if the bits in a and in b are equal on ith place. The result fitness is obtained by counting the bits via $zeroucount()$, which can be implemented in two ways.

In the first way, we create a lookup table. The table has two columns (key, value). The key has limited length of eight bits and the value is the number of zero bits for each key. It is necessary to utilize bitwise operation in order to compute Hamming distance for whole integer.

The second way is based on the bitwise operations as written in algorithm 2. This method consists only twelve operations and it has lesser demands on memory than the first method [1].

Algoritmus 2: *Population count*

```
    v = v˜- ((v˜>> 1) & 0x55555555);
    v = (v˜& 0x33333333) + ((v˜>> 2) & 0x33333333);
    c = ((v˜+ (v˜>> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
```

Utilize of SIMD coprocessors

Nowdays, the common processor unit has built in an SSE unit. SSE is acronym for Streaming SIMD Extensions where SIMD means Single Instruction Multiple Data. This unit has eight 128-bit registers. Each register contains either two 64-bit or four 32-bit floating point values. The operations are applied in parallel. Thus we can evaluate four arithmetical operations at one time. This is situated on picture 3.1.

The unit SSE was constructed for floating point numbers. However, it also has bitwise operations such as conjunction, disjunction, exclusive disjunction and negative conjunction. The realization of negation is done with negative conjunction where both operands contains the same register (variable). In case of evolutionary circuit design, we can evaluate 128 fitness cases in one iteration.

Moreover in SSE4.2 (and newer) instruction set, the instruction `POPCNT` was added. This instruction counts the sum of the bits which are set to number one. The principle of the calculation is similar to algorithm 2.

Technology Advanced Vector Extensions (AVX) extends 128-bit registers of SSE unit to 256-bits. The 256-bit block contains four 64-bit values or eight 32-bit values.

SSE and AVX instructions are primary used in the assembly languages or in structured languages such as C language. The usage of the SIMD coprocessor is problematic in Python because it is considered to be a scripting language.

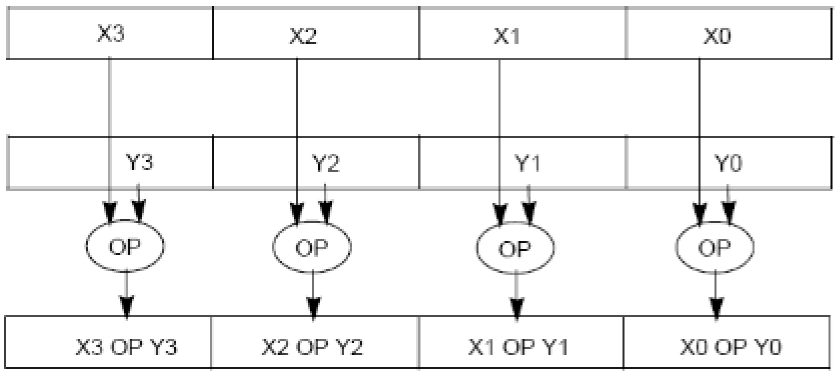


Figure 3.1: Execution of SIMD operation

Utilize of GPU

We can use graphic processor unit for the parallel evaluation of fitness value. GPU permits greater degree of parallelization because GPU contains more arithmetic units than SIMD coprocessors. The main disadvantage lies in the slow memory management. If GPU would realized the fitness calculation only and if the evolution algorithm would run on CPU, then the main problem was in the slow transmission of data between RAM and GPU memory. This method would be paid off if we would transmit large amount of data (megabytes). Still this problem can be solved if the whole algorithm would run on GPU. However, this solution give us another problem. Each condition stops all the threads in each block and each unaligned access to memory would lead to the time penalties.

3.2.2 Acceleration based on phenotype

We can speed up CGP algorithm if we will evaluate the phenotype of candidate solution. In the other words we will evaluate only the used nodes of the graph. We initiate the table with the nodes which are used as primary outputs. Then we pass backwards the chromosome. However this acceleration is not good if the phenotype uses almost all nodes of the graph.

3.2.3 Just in Time Compilation

Nowdays, the CPU utilizes instruction pipelining. The idea is based on the fact that each instruction processing can be splitted into phases:

- **Fetch** – Loading the instruction from the memory
- **Decode** – Decoding the instruction. Selecting the type of operation. Loading its operands.
- **Execution** – Execution of the instruction.
- **Write back** – Saving the results into the memory of the processed instruction.

All the instructions are completed in four clock cycles in the subscalar CPU. However in the CPU with instruction pipelining, only the first instruction is done in four clock cycles and the rest of them is completed in one clock cycle.

The problem occurs in the branching operations. The program may continue on another position in RAM. In that case, the unfinished instructions have to be stopped and flushed away.

The calculation of fitness value in CGP contains a lot of branches. In the function, there is several conditions `if()` in a main loop `for()`, see algorithm 3. Let say, we have k training vectors and we disable parallel simulation. We have to run this code for k times and the amount of processed branches is increased by k times.

Algoritmus 3: *Fitness value evaluation*

```
Input: Graph (the chromosome)
Output: Response of candidate solution
  Load one training vector into primary inputs responses.
  for each node $n$ in graph do
    $a$ = a node which is connected to the first input of $n$
    $b$ = a node which is connected to the second input of $n$
    operation = an operation of $n$
    if operation is $+$ then
      $n.response = a.response + b.response$
    else if operation is $-$ then
      $n.response = a.response - b.response$
    else if operation is $*$ then
      $n.response = a.response * b.response$
    else if operation is $sin$ then
      $n.response = sin(a.response)$
      ...
    end if
  end for
```

The branching can be decreased by just-in-time (acronym JIT) compilation of the graph during the run of algorithm. For each candidate solution, the JIT compilation is run only once. It is run right before the fitness value evaluation. The goal of the JIT compilation is to eliminate all the branches. In the result, we obtain an effective code which is then executed [11].

Unfortunately in Python, we can't eliminate all the branches because the code is interpreted and the interpreter uses branch operations. Nevertheless, we decrease the number of the interpreted operations. Python checks the data types of the relation operators in conditions which may lead to that the fitness function can run faster.

Utilizing function `compile()`

We can compile the Python code in Python with function `compile()`. The main parameter is a string with the source code. The result is a stream of bytecode. We execute this bytecode by functions `exec()` or `eval()`. The function `exec()` can contain bytecode with branches. However, the function `eval()` must not contain bytecode with branches, the usage of this function is only for arithmetical operations.

Function `compile()` is slow because the compilation employs the syntax control of the written code. Thus it is better to write own bytecode.

JIT Compilation into the Bytecode

For writing own bytecode, we will Pro vytvoření vlastního bytecode můžeme využít funkci `CodeType()` modulu `types` ze standardní knihovny Pythonu. This function accepts thirteen parameters in Python 3. In Python 2, it is twelve. The parameters are discribed in appendix or in Python documentation. The most important parameters are stack size, local variables, constants and the bytecode. The stack size refers to the size of the interpreter stack. Local variables and constants are saved in the `tuple()` data type. And the bytecode is saved as a bytearray [4].

By the reverse engineering, we can find the meaning of each byte of the bytecode. Function `compile()` returns object `Code` and its attribute `co_code` represents the bytecode. For example:

```
>>> c = compile("y = x/0","","exec");
>>> print(c.co_code)
0x6500006400001b5a010064010053
```

This bytecode can be decompiled by function `dis()` in `dis` module [4].

```
>>> dis(c.co_code)
  1       0 LOAD_NAME           0 (x)
          3 LOAD_CONST          0 (0)
          6 BINARY_TRUE_DIVIDE
          7 STORE_NAME          1 (y)
         10 LOAD_CONST          1 (None)
         13 RETURN_VALUE
```

The mostleft column stands for the line number of the compiled code. It is number 1 in our case. The second column with numbers (0, 3, 6, 7, 10, 13) is the index in the bytecode. The following column give us the name of the instructions. The semantic of the instructions is available in the documentation [4]. And the last column stands for the instruction arguments. The values of this column are indexes to tuple structure of either variables or constants. The name of variables or constants are given in the parenthesis.

Numerical values of instructions are given in the file `$PYTHON_LIB_DIR/opcode.py`. Nevertheless, they can be obtained directly in the Python if we use the values of the second column as the index to bytecode string. In example, we want to find the numerical value of divide instruction `BINARY_TRUE_DIVIDE`, then we write command `print(kod.co_code[6])`. The bytecode should be terminated with sequence of instructions `LOAD_CONST (None)`, `RETURN_VALUE`.

Thus in Cartesian Genetic Programming, it leads to generation of the bytecode with same length for each operating node. The bytecode of the graph is stored in the array with length of $n_r * n_c * l_n + 4$ where n_r is number of rows, n_c is number of columns and l_n is the maximal length of the bytecode for one node. Then we have to add constant 4 which denotes the termination of the bytecode.

We rewrite the bytecode of each node in order to speed up the graph compilation. However, if we compile only phenotype, then the graph has variable length. This problem is solved in chapter with implementation 4.1.3.

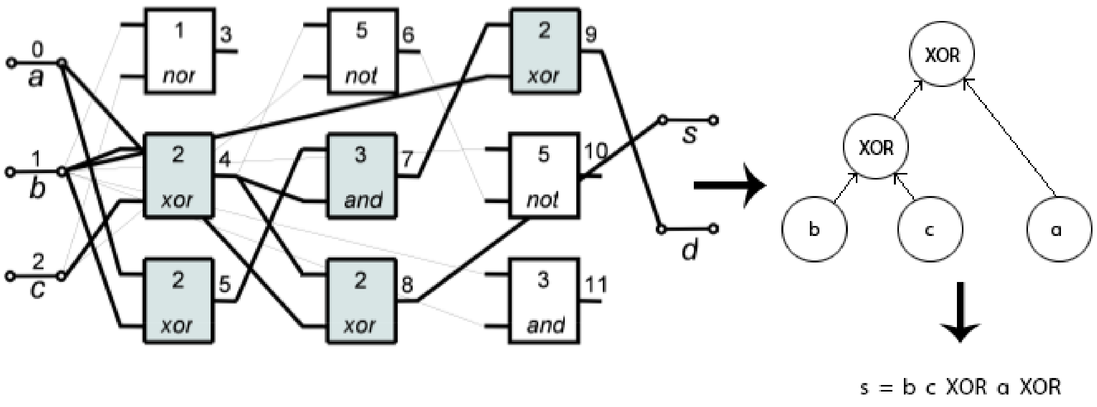


Figure 3.2: Postfix compilation of chromosome

JIT Compilation into the Bytecode in Prefix Notation

The interpreter of Python is based on the stack architecture similarly to Java Virtual Machine. The operations are executed on the top of the stack. This can be utilized in such way that the chromosome is compiled into postfix notation which is suitable for python interpreter. The code of phenotype looks like a tree and we can make a walk in which the children nodes are traversed before their parent nodes are traversed. This method is pictured on figure 3.2.

If some node is used more than twice in the phenotype, then we can save its value into temporary field. Thus we eliminate reevaluation of the computed nodes.

This type of JIT compilation is more complex in the trade off the efficient code.

3.3 Accelerating Python Programs in PyPy

„If the implementation is hard to explain, it's bad idea, except PyPy.“

(Benjamin Peterson, hlavní vývojář PyPy) PyPy is an interpreter of Python written in Python. PyPy is based on JIT compilation of the code, it rolls up the loops and somehow it optimize the code [8].

The main problem is when we want to execute the compiled code in a loop. Then we have to wrap the compiled function into another function. Otherwise we get an inefficient code. We can compare the speed in the table 3.1.

`exec()` in the loop `for()`	`exec()` in function `myexec()` calling `myexec()` in the loop `for()`
1492	4357

Table 3.1: The number of evaluations per one second of the same bytecode in PyPy. Tested subject: evaluation of 4-bit adder

3.4 Cython

Cython is a generator of C source code from the Python that is enhanced by C data types. Furthermore, it contains calling C language functions. We can create effective code in this language. Cython language allows:

- to write a code in Python and to get its source code in C/C++
- to optimize Python code by utilizing static data types
- to utilize even dynamic data types from Python
- to create fast libraries/modules which can be run in Python
- to utilize the C/C++ libraries

The code of Cython is translated into C language and then the C code is compiled with GCC or G++. The final result is a library which can be used in Python.

3.4.1 Cython's Syntax

Static Data Typing

If we add `cdef` with the data type before the variable, then we get variable with static datatzpe. In example: `cdef int 42`. The key word `cdef` tells us that it is the type of C language. Data types of function results are defined by key word `def` for Pythonic ones or `cpdef` for Cythonic ones. Please note the p letter.

The Differences and Similarities between C and Cython

- There is no operator `->` in Cython. Instead of the arrow (i.e. `p->a`), we use a dot (i.e. `p.a`).
- Cython doesn't have unary operator of dereference `*`. We use `p[0]` instead of `*p`.
- In Cython, we can use reference operator `&` in the same way as in C language.
- We can utilize casting data types via `<type>var`, in example:
  ```
  cdef int * a, float * b
  a = <int *> b
  ```

Calling Functions from C Library

If we want to utilize a function from standard library of C language, we have to import the module via command `cdef extern from "lib_name":`. Then we have to write the name of each used function with parameters in its name space. The declaration must be written before the calling the function. For an example declaration of function `malloc()`.

```
cdef extern from "stdlib.h":
    void * malloc(int a)
```

Chapter 4

Implementation of Proposed Method

Each implementation corresponds to each optimization step. The first solution is inefficient version of CGP algorithm for the design of combinational circuits. The next solutions were optimized by parallel simulation (in case of circuit design) and Just-in-time compilation. Then the candidate solutions were translated into the bytecode in infix notation and into the bytecode in postfix notation. At the end of this work, the CGP module was written in Cython and it can be run in the Python interpreter. Each optimization step was checked in different interpreters - Python2, Python3 and PyPy.

4.1 CGP in Pyhton

The CGP can be divided into few functional blocks - modules. The core of algorithm is in the file `cgp_core.py`. The algorithm is based on the conception written in chapter 2. In the files `cgp_circuit.py` and `cgp_equation.py`, CGP is specialized on particular problem. These files were mostly modified during the optimization. Module `cgp.py` serves as a communicator between the CGP and Python interpreter.

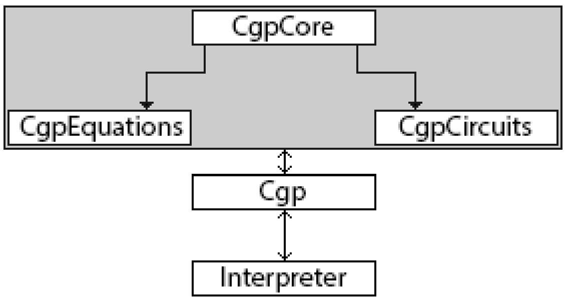


Figure 4.1: Communication diagram between the modules

Algorithm core

The core of algorithm (class `CgpCore`) is found in the file `cgp_core.py`. Function `runAscending()` runs CGP algorithm and the selection of the parent is based on maximization of fitness value.

This function is preferred for combinational circuits design.

The function `runDescending()` runs CGP algorithm as well. However, the parent selection is based on the minimization of fitness function. This fitness utilizes symbolic regression.

The function `_initCGPRun()` is used for initialization of CGP run. It calculates variables such as chromosome length, the amount of nodes in the graph. Furthermore, it sets pointers. The function `_initLevelBack()` allocates arrays of values which define the valid interconnections between the columns. These arrays are then used in the mutation `__mutation()` in order to prevent the creating incorrect graphs. The function `_initPopulation()` spawns population. The function `initOutputBuff()` creates an array which is used for the evaluation of each candidate solution. The next function `_initUsedNodes()` allocates an array of used nodes which is used for finding the phenotype by the function `_usedNodes()`.

Furthermore, this module contains two functions used for debugging `resultChrom()` and `showChrom()` which prints the individual (chromosome).

Communicator module

This module is used for the communication of the interpreter and the algorithm. It contains the class called `Cgp`, its constructor initiates the default values. The functions `file()` and `data()` sets training data of CGP. The next function `graph()` sets dimensions of the graph and graph interconnection. The method `run()` runs the CGP algorithm. This function selects the type of evolution (ascending or descending fitness) on the basis of the data type. The arguments of the functions are described in the appendix D.

The methods `showChrom()` and `resultChrom()` shows the chromosomes. The rest functions (such as `allLogicalOperations()`, `reedMuller()` or `symbolicRegression()`) are used for the setting of function set. User can define own function set too. The attributes `functionTable` and `functionTableOp` are the lists of strings representing the name of the function. The variable `functionArity` is the list of functiosn arity (the number of node inputs). The maximal funcion arity is limited to 2.

CGP specializatinos

In the files `cgp_circuit.py` and `cgp_equation.py` is located an implementation of two classes `CgpCircuit` a `CgpEquation`. Each implementation is particular to specific problem. The first is used for the evolutionary design of combinational circuits and the second solves the symbolic regression problem.

These files handles acceleration techniques which are described in the text below.

4.1.1 Naive implementation of evolutionary circuit design

The first fitness evaluation algorithm was written only for the design combinational circuits. The main advantage is in the simulation of the circuit which can be done only once per the candidate solution because the integer in python has not fixed length. Another advantage lies in the training data creation. The goal of this implementation was to measure the computational complexity of the solution in parallel simulation. The results are in the chapter 3.2.1.

However, this solution is not suitable for the following transcription into the Cython, because integers have fixed length in static typed languages (32 or 64 bits). The input/output data are not divided into the training vectors on such that length. Moreover, this evaluation of candidate solution is not suitable for symbolic regression.

In this solution, user can define his/her function set by the changing elements of the list `functionSet`. Each element is a pointer to the specific function.

4.1.2 Dataset Transfer to Training Vectors

The second implementation (M2) differs in its training dataset structure. In this case, the dataset is composed of array of static integers. The endeavour is to check if operations over static integers (*int type in Python) are faster than operations over dynamic integers (*long type in Python).

During initialization, the data are transformed into training vectors by `__convertData()` in a such way that each dimensions was in a row. This is pictured on figure 4.2. The structure was used for both training vectors - input data and output data. Function `__evalFitness()` then evaluates these training vectors.

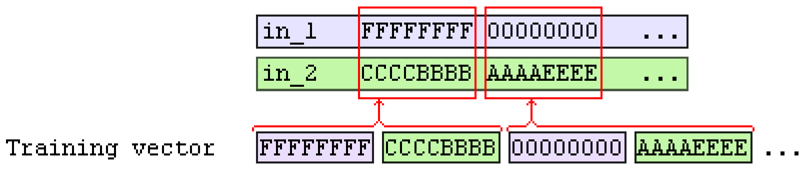


Figure 4.2: Dataset Transfer into Training Vectors

4.1.3 Just-in-time compilation by `compile()`

The principle of JIT acceleration technique was mentioned in the chapter 3.2.3. During initialization of `Cgp` object, the array of bytes is created by function `__initByteCodes()`. This array contains the bytecode in the running evolution.

There are a few ways how to compile the code. The third implementation (M3) used Just-in-time compilation by built in function `compile()`. The chromosome is firstly translated into string in `__getCode()`.

In order to creation of users operations, there is a list of functions `functionSetStr` containing elements of strings. Each string represents operation of the CGP node.

During evaluation of candidate solution, bytecode is executed by inbuilt function `exec()`.

4.1.4 Just-in-time Compilation by my Compilator

The next method of implementation (M4) is my own compilation, see function `__getByteCode()`. The main principle is that each node creates an array bytecode. The node function is changed by rewriting just parts of the bytecode.

4.1.5 Just-in-time compilation in the postfix notation

The next implementation (M5) is my own compilation into polish postfix notation, see `__getByteCodePostfix()`. The postfix notation is created by function `__getBc()`. There was one problem that occurred. If there is some node used twice or more in the expression syntax tree, then the program performed a few unnecessary calculations. I solved this with saving each interim result in the array `outputBuff`.

The creation of own functions in this implementation is the hardest because the user needs to know a bytecode. However, bytecode is (was) poorly documented and it was necessary to obtain it manually. See chapter 3.2.3. The bytecode is saved in the list `functionBCx` where `x` is a number from 1 to 5.

4.2 Cython

This section deals with the implementation techniques using Cython. The first method that used Cython (M6) was created by translating Python into C.

4.2.1 Static Typing of Variables

The second method (M7) used static typing of variables in order to speed up the algorithm. This section describes all the changes that had been done.

Whereas that moduleless Python has only List and dictionary for array representation, then the time performance is decreased due to allocation of new lists or new elements. In the case of large arrays, we can import the module `array` in Python. It contains C arrays and it has limited functionality (we can save only numbers), however it is faster than Pythonic arrays.

In Cython, the manipulation with this datatype is problematic. The structure `array` is created by Python and it works with Python allocated memory. We can not change the data else the compiler yields an error. Thus, we have to create the new arrays with function `malloc` from standard library from C language.

Cython uses static classes which are more similar to C++ class than to Python class. Moreover, Python is dynamic typed language. In Cython, types were converted into static ones in order to achieve a speed up. However, this leaded to drop the original object design which was described on the beginning of this chapter. Specifically, module `CgpCore` was removed. User can choose a function set from the sets in the file `cgp_functions.pyx` or user can define . Furthermore, there was a small change in training vectors. In the previous implementations, the algorithm had two training vectors (one array for input, other for output). In this implementation, these arrays are interleaved (input vector, output vector, input vector, output vector...).

4.2.2 Disabling User to Define Functions

Utilizing OOP concept causes speed penalty because the user functions may be saved in the array as a pointers to their callings. This leads to the time penalization – too much functions calling (running or something like that). This can be fixed by the direct function calling (running). Thus the code:

```
out[j] = self.functionSet[chrom[idx]](a, b)
```

can be rewritten to:

```
op = chrom[idx]
if   op == 0: c = a + b
elif op == 1: c = a - b
elif ...
```

This technique was used in the method (M8).

Chapter 5

Experimental Evaluation

This chapter contains experimental evaluation of each proposed implementation which were described in the chapter 4. The performance evaluation was based on the number of candidate solution simulations per one second.

The method of verification of the function is described firstly and then I write about each acceleration technique. In the section 5.2, I compare parallel simulations. The comparison of JIT compilations is in the section 5.3. In the next section 5.4, Cython implementations are evaluated.

5.1 Verification of the Fitness Function

The correctness of fitness value evaluation was verified in this way. CGP was initiated by a chromosome encoding the fully functional solution (circuit or equation). Its fitness value should be the maximal. Furthermore in the verification of the algorithms, all of the CGP parameters have the same values in order to obtain the same fitnesses. For symbolic regression, the function for one dimensional logarithm was used this chromosome:

```
[0,0,7, 1,0,4, 1,0,8, 2,0,7, 4,3,5, 5,5,4, 2,3,8, 6,2,8, 8,4,7, 2,4,5, 7,9,0,
 9,8,6, 12,10,6, 13,13,4, 10,0,2, 13]
```

Its maximal fitness value is around 123 for data in the file `logx.txt` and its functional set is $\{0.25, 0.50, 1.00, id, add, sub, mul, div, sin\}$.

For the verification of combinational circuit design, I utilized 4-bit adder (note that it has 8 inputs) which has chromosome:

```
[2,6,7,   7,3,5,   9,8,6,   5,1,3,   2,6,7,   2,8,1,   13,4,0,   0,4,7,   8,9,3,
 11,5,2,  4,4,7,   15,17,7, 10,13,6, 2,14,1,  15,3,3,  3,7,3,    13,14,1, 2,4,2,
 25,16,3, 5,26,1,  11,20,5, 28,19,7, 0,4,1,   23,2,0,  25,2,7,   20,11,7, 30,27,0,
 18,29,1, 35,0,4,  10,0,0,  18,16,1, 30,10,6, 28,37,2, 32,31,1,  31,29,5, 28,19,7,
 18,33,1, 29,15,6, 45,34,2, 18,46,1, 47,35,44,38,23]
```

It has got maximal fitness value around 1280 and the minimal used nodes of 23 for the set of functions: $\{id, and, or, xor, not, nand, nor, nxor\}$.

5.2 The Influence of Parallel Simulation

By profiling of CGP algorithm, we can find out that CGP spends a lot of time in the evaluation of the fitness. The evaluation of candidate solution must be run for each training vector. In the ideal conditions, we would run the simulation of candidate solution only once if we utilized parallel simulation. Nevertheless, Python has dynamic integers (long datatype in Python2) and it can approximately give us „ideal“ conditions by itself (see method M1). However, if we use static integers (int datatype in Python2) we can run in parallel up to 64 simulations on 64-bit machine in one run of the simulation (method M2). This leads to an increase of the computational time. See figure 5.1.

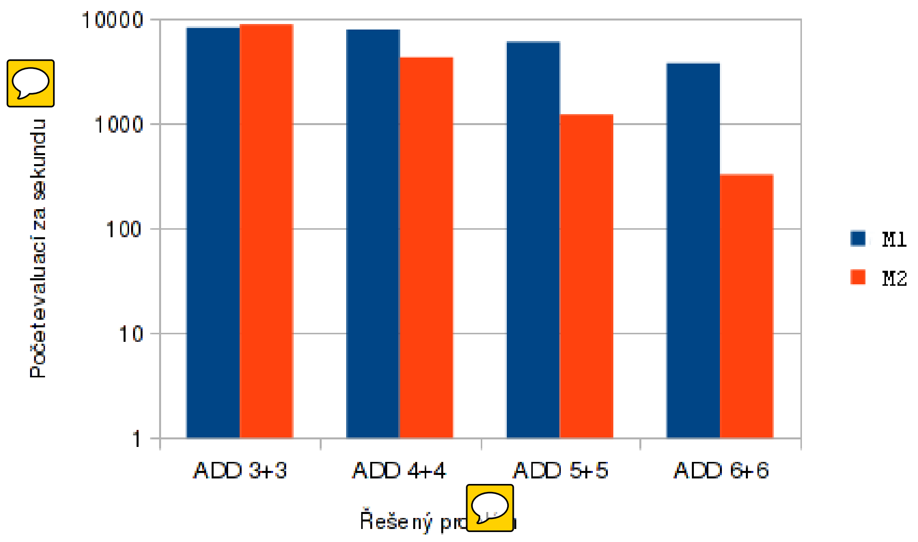


Figure 5.1: The performance of the implementation utilizing parallel simulation (M1) and semi-parallel implementation (M2). X-axis - the solved problem. Y-axis - the number of evaluations per second.

The interpreter is also important, see figure 5.2. PyPy seems to be faster for smaller problems and Python3 may be used for the larger problems.

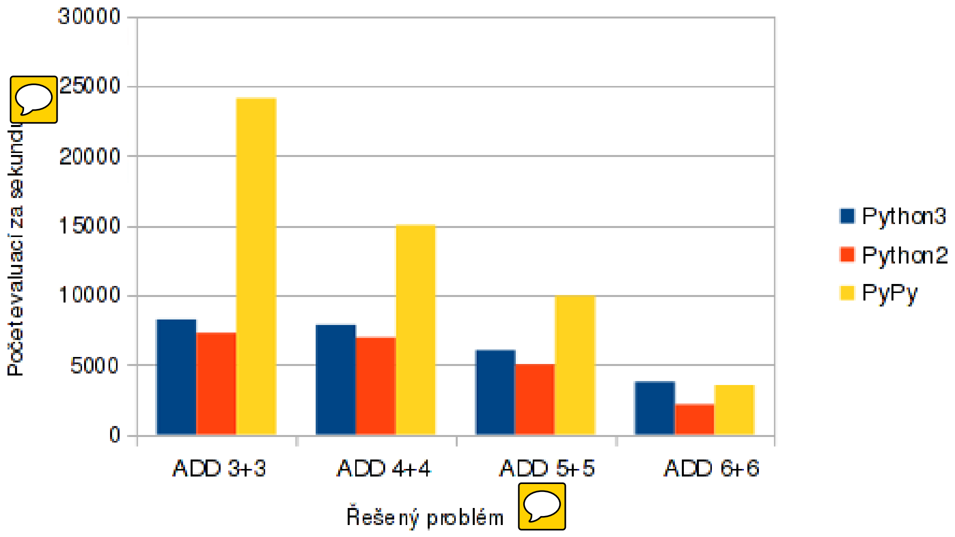


Figure 5.2: The performance of interpreters in the implementation utilizing parallel simulation (M1). X-axis - the solved problem. Y-axis - the number of evaluations per second.

5.3 Experimental Evaluation of the Implementations Using JIT Compilations of the Chromosome

In chpaters 3.2.3 and 4.1.3, JIT compilations were presented. The results of this experiment show us that the effective compilation of the fast code is also important. Method M3 contains inbuilt function `compile()` for the JIT compilation. Method M4 has my own JIT compiler and the last method M5 utilizes also my own JIT compiler but into postfix notation.

On figures 5.3, 5.4, we can see that the last mentioned method (M5 - own compiler into postfix notation) has the best results for larger amount of training vectors.

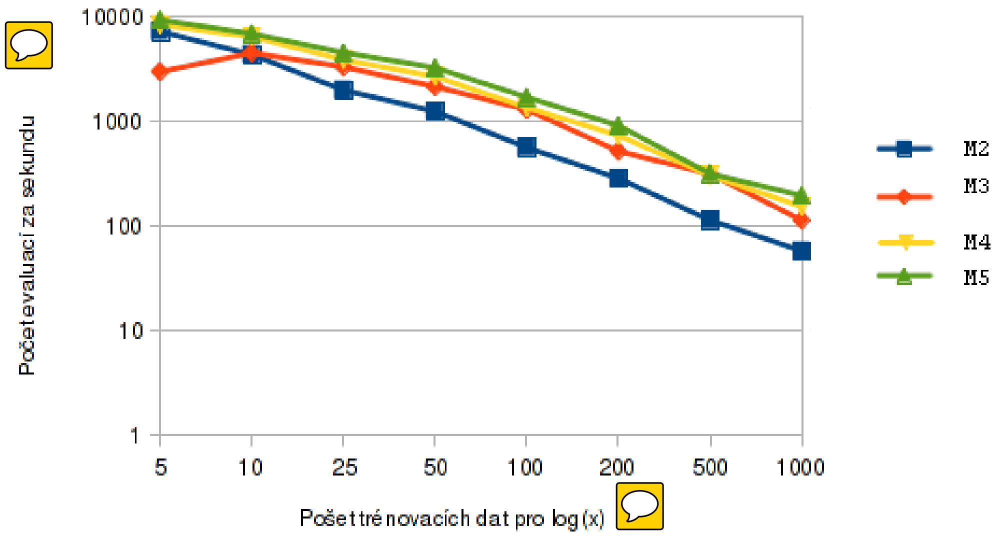


Figure 5.3: The performance of **Python2** for the original implementation (M2), the JIT compiler containing inbuilt function compile inbuilt `compile()` (M3), my own JIT compiler in infix notation (M4) and own JIT compiler in postfix notation (M5). X-axis - the number of training vectors for $log(x)$ function. Y-axis - the number of evaluations per second.

Python3 has much slower `compile()` function if we compare M3 on the plot 5.3 between M3 on the plot 5.4. JIT compilation into postfix notation M5 has the best results for the larger amount of data.

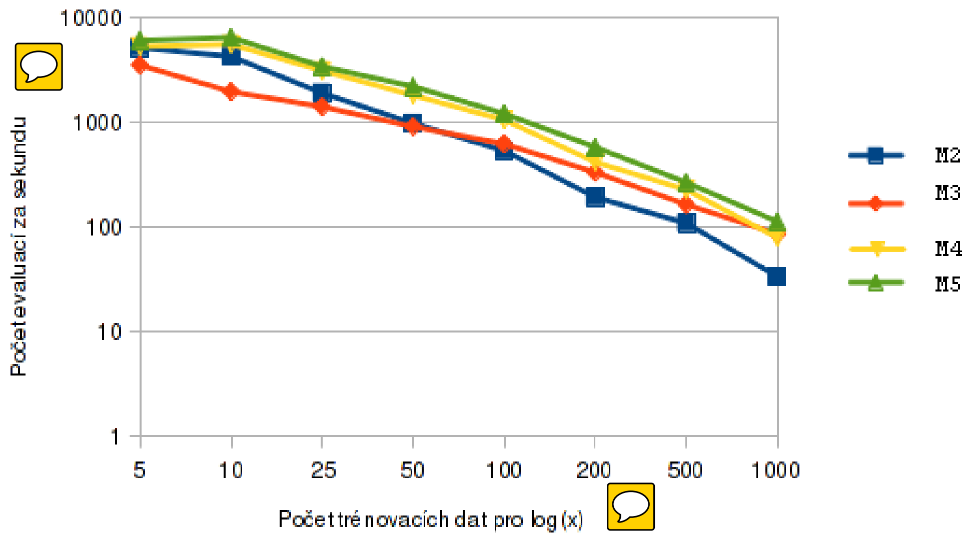


Figure 5.4: The performance of **Python3** for the original implementation (M2), the JIT compiler containing inbuilt function compile inbuilt `compile()` (M3), my own JIT compiler in infix notation (M4) and own JIT compiler in postfix notation (M5). X-axis - the number of training vectors for $log(x)$ function. Y-axis - the number of evaluations per second.

The worst interpreter for JIT compilation is PyPy. On the plot 5.5, we can see that compilation slowed everything. The reason may be that PyPy uses JIT-compiler itself, see chapter 3.3. Moreover, method M5 is the worst because the postfix transformation contains recursion callings of a function.

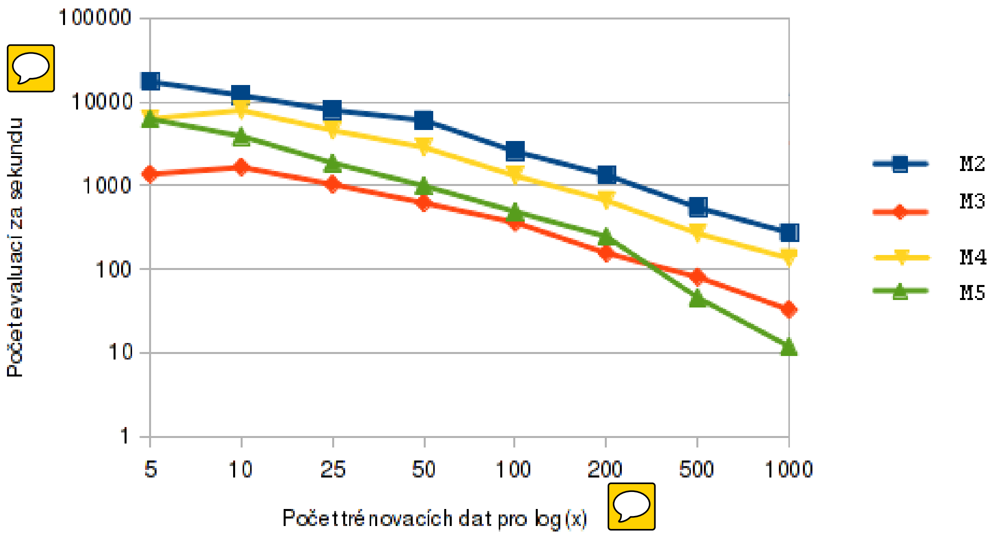


Figure 5.5: The performance of **PyPy** for the original implementation (M2), the JIT compiler containing inbuilt function compile inbuilt `compile()` (M3), my own JIT compiler in infix notation (M4) and own JIT compiler in postfix notation (M5). X-axis - the number of training vectors for $log(x)$ function. Y-axis - the number of evaluations per second.

We can compare interpreters on the figure 5.6.

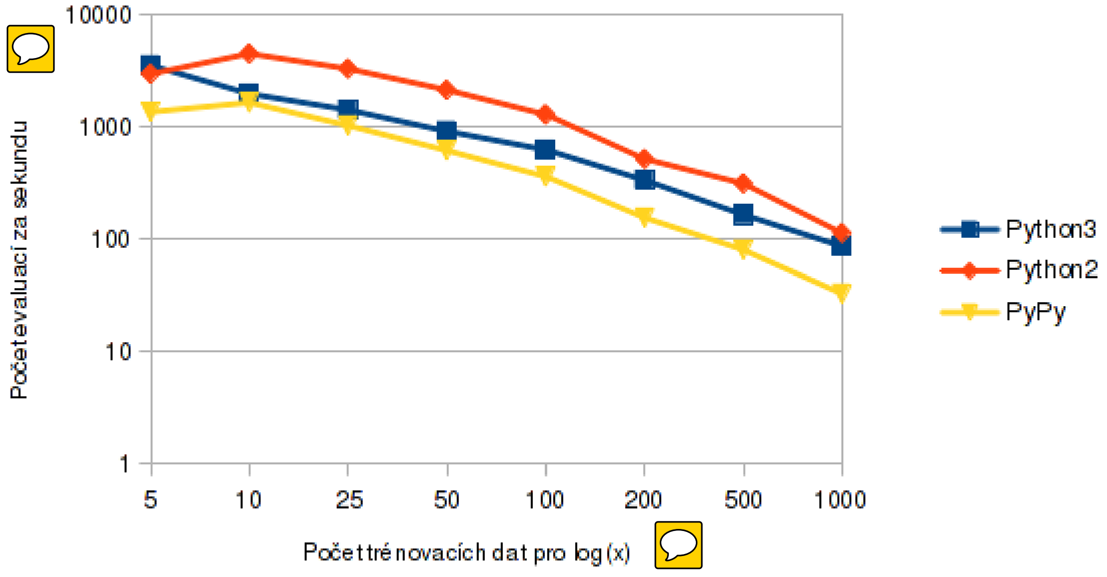


Figure 5.6: Porovnání výkonnosti kompilace pomocí funkce `compile()` pro jednotlivé interprety

On the table 5.1, we can see that method M5 is nearly four times faster than the method without any acceleration. Functions $cos(x)$, $sin(x)$, $log(x)$ a $xxyz(x)$ were used as a benchmark with a thousand training vectors.

	cos(x)	sin(x)	log(x)	xxyz(x)
Without JIT (M2)	40	43	58	52
With JIT into postfix (M5)	176	188	196	193

Table 5.1: The number of evaluations per second on the benchmark containing a thousand training vectors. Interpreter - Python2

5.4 Experimental Evaluation of the Implementations in Cython

This section compares three different implementations in Cython which were described in chapter 4.2. The first implementation was just translation of Python into C language (M6). Method M7 utilizes static typing. Method M8 extends M7 with the rolling up the function set. We can compare each method on the figure 5.7.

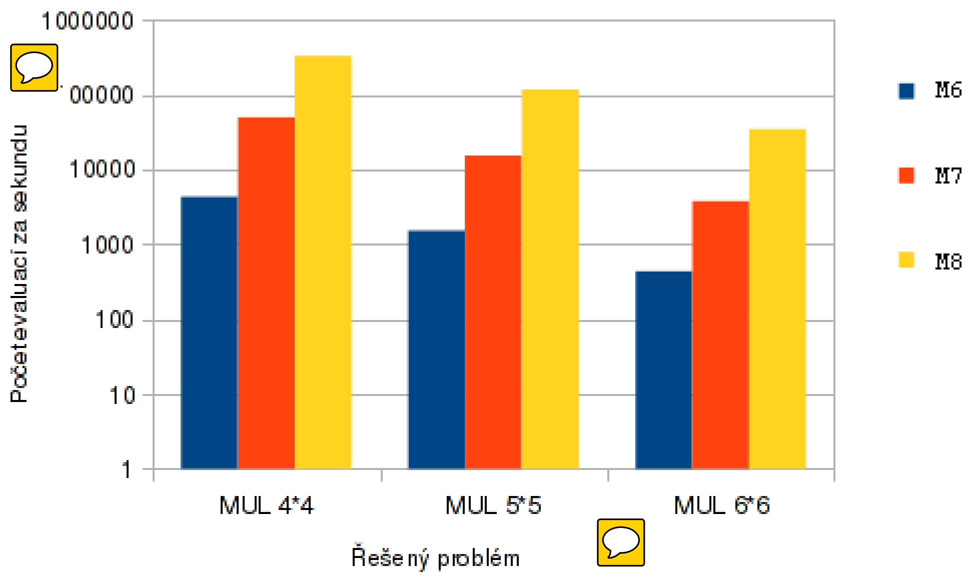


Figure 5.7: The performance of **Cython** implementations. M6 - from the pure Python source code. M7 - Python source with static types. M8 - Python source with static types and with rolled up functions. X-axis - the solved problem. Y-axis - the number of evaluations per second.

Method M8 is approximately fast as the algorithm written in pure C language, see table 5.2.

	Cython (M8)	C[10]
`5x5 MUL`	117644	119165
`6x6 MUL`	34780	40295

Table 5.2: The number of the evaluation per second between the implementation written in C and Cython language.

On figure 5.8, we compare Cython implementations with interpreters.

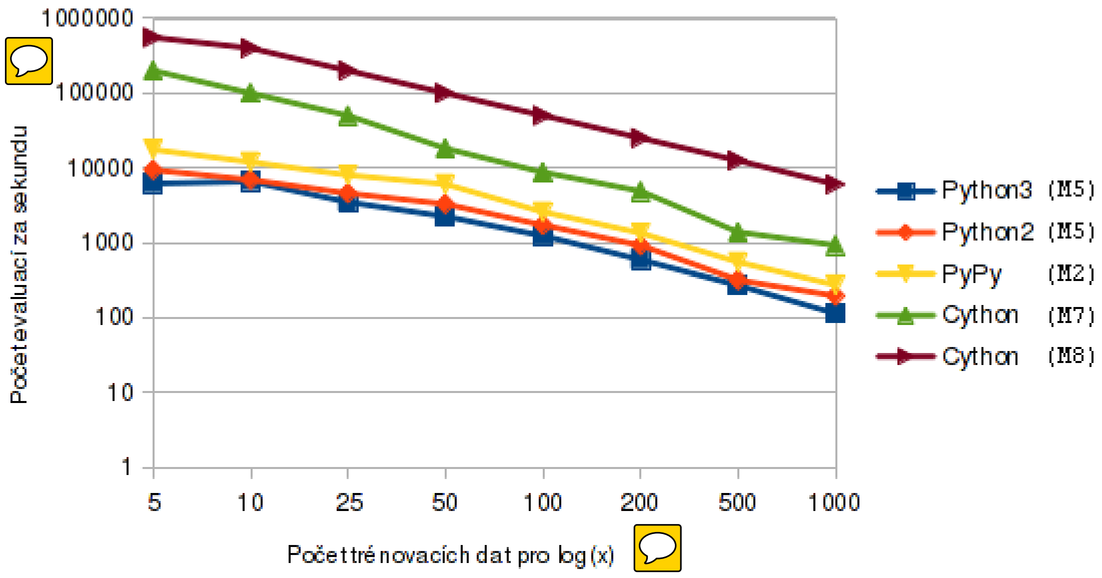


Figure 5.8: The comparison of the best acceleration techniques for each interpreter and Cython language. X-axis - the number of training vectors for $log(x)$ function. Y-axis - the number of evaluations per second.

Chapter 6

Conclusion

The goal of this thesis was to speed up Cartesian Genetic Programming in Python laguage. Several accelerations were used for that: parallel simulation, JIT compilation and utilizing Cython. Each acceleration technique was tested in the each interpreter – Python2, Python3 and PyPy.

Experimental evaluation gave us that JIT compilation can increase the time performance of the algorithm. Furthermore, the technique of the compilation is also important. Built-in function `compile()` checks the syntax analysis and it leads to slowing up the algorithm. However if we use JIT compilation into postfix bytecode, we can speed up the algorithm more than four times for a thousands training vectors. Parallel simulation seems to be the best acceleration technique, however it depends on the architecture of the processor where CGP is run. The most effective acceleration utilizes a module created by Cython language.

In further work in Cython module, GUI extension can be created or a new functional sets can be added. It can be extended for another problems such as image filter design or evolutionary art. Another way of the further work is to utilize SIMD coprocessors for the JIT compilation or utilizing reconfigurable circuit in FPGAs (Field-programmable gate arrays) in order to do parallel simulations.

Bibliography

[1] Sean Eron Anderson. *Bit Twiddling Hacks - Count Bits Set Parallel.* Last visit 5. 5. 2013.
http://graphics.stanford.edu/~seander/bithacks.html.

[2] Oliver Crow. *Efficient String Concatenation in Python.* Last visit 5. 5. 2013.

[3] Justin Dailey. *Python: range() vs. xrange().* Last visit 5. 5. 2013.
http://justindailey.blogspot.cz/2011/09/python-range-vs-xrange.html.

[4] Python Software Foundation. *Python v3.1.1 documentation.* 2013.
http://docs.python.org/3/.

[5] Jan Jelínek and Vladimír Zicháček. *Biologie pro gymnázia.* Publisher in Olomouc, 2003.

[6] Alex Martelli, Anna Ravenscroft, and David Ascher. *Python Cookbook.* O'Reilly Media, 2005.

[7] Julian Miller. *Cartesian Genetic Programming.* Springer, 2011.

[8] The PyPy Project. *PyPy.* Last visit 5. 5. 2013. http://pypy.org/.

[9] Lukáš Sekanina, Zdeněk Vašíček, Michal Bidlo, Richard Růžička, Jiří Jároš, and Petr Švenda. *Evoluční hardware.* Academia, 2009.

[10] Zdeněk Vašíček. Cartesian genetic programming, 2013.
http://www.fit.vutbr.cz/~vasicek/cgp/.

[11] Zdeněk Vašíček and Karel Slaný. Efficient phenotype evaluation in cartesian genetic programming. In Alberto Moraglio, Sara Silva, Krzysztof Krawiec, Penousal Machado, and Carlos Cotta, editors, *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012*, volume 7244 of *LNCS*, pages 266–278, Malaga, Spain, 11-13 April 2012. Springer Verlag.

Appendix A

CD content

- /**tex** – LATEXsource codes
- /**cgp** – result Cython module
- /**src** – source codes
  - /**cython** – Cython source codes
    - /**v0.1** – M6 dynamic typing
    - /**v0.2** – M7 static typing
    - /**v0.3** – M8 static typing with rolled up functions
  - /**data** – benchmarks
  - /**v0.1** – M0 without fitness calculation
  - /**v0.2** – M1 parallel simulation
  - /**v0.3** – M2 semi-parallel simulation (64-bit)
  - /**v0.4** – M3 JIT compilation via `compile()`
  - /**v0.5** – M4 JIT compilation into infix notation
  - /**v0.6** – M5 JIT compilation into postfix notation
- /**viewer** – chromosome viewer

Appendix B

Bytecode

This part of the appendix is about function `CodeType` in the module `types` with that was created bytecode . However function `CodeType()` is (was) documented poorly in [4] and numerical values of bytecode wasn't documented at all. The list of bytecode instruction can be find on the next page.

```
>>> print(types.CodeType.__doc__)
code(argcount, kwonlyargcount, nlocals, stacksize, flags, codestring,
      constants, names, varnames, filename, name, firstlineno,
      lnotab[, freevars[, cellvars]])

Create a code object.  Not for the faint of heart.
```

- **argcount** – arguments count; structure type `int`
- **kwonlyargcount** – keywords arguments count; structure type `int` [1]
- **nlocals** – number of local variables; structure type `int`
- **stacksize** – interpreter stack size; structure type `int`
- **flags** – bitarray of informations about the code; structure type `int`
- **codestring** – bytecode array; structure type `bytes`
- **constants** – constants of the code; structure type `tuple`
- **names** – names of variables; structure type `tuple`
- **varnames** – names of the argument of function; structure type `tuple`
- **filename** – the name of file, where the bytecode can be saved; structure type `string`
- **name** – function name of the compiled bytecode; structure type `string`
- **firstlineno** – the number of the first line of the code (similar to program counter in the processor); structure type `int`
- **lnotab** – tabbing of the functions in the bytecode; structure type `tuple`

[1]not implemented in Python2

Bytecode	Instruction name	Semantics	Operation
1	POP_TOP	Copy and remove data from the top of the stack.	
9	NOP	No operation.	
15	UNARY_INVERT	Inverts the top of the stack.	`TOS = ~TOS.`
20	BINARY_MULTIPLY	Multiplication.	`TOS = TOS1 * TOS`
23	BINARY_ADD	Addition.	`TOS = TOS1 + TOS`
24	BINARY_SUBSTRACT	Subtraction.	`TOS = TOS1 - TOS`
27	BINARY_TRUE_DIVIDE	Division.	`TOS = TOS1 / TOS`
64	BINARY_AND	Logical Conjunction.	`TOS = TOS1 & TOS`
65	BINARY_XOR	Logical Exclusive Disjunction.	`TOS = TOS1 ^ TOS`
66	BINARY_OR	Logical Disjunction.	`TOS = TOS1 | TOS`
25	BINARY_SUBSCR	Load the value from the array.	`TOS = TOS1[TOS]`
60	STORE_SUBSCR	Save the value from the array.	`TOS1[TOS] = TOS2`
90	LOAD_NAME	Save the value from the top of the stack into a variable.	`name = TOS`
100	LOAD_CONST	Load constant.	`push(co_constants[arg])`
101	LOAD_NAME	Load variable.	`push(co_names[arg])`
83	RETURN_VALUE	Returns top of stack and leaves the function.	`return TOS`

Table B.1: Used Bytecode Instructions

Appendix C

Results

In this part of appendix, we can see obtained results for each benchmark.
CGP Parameters for circuit design were: rows=1, cols=40, levelback=40, functions=(id, not, and, or, xor, nand, nor, xnor)
CGP Parameters for symbolic regression: rows=1, cols=15, levelback=15, functions=(id, 1.0, 0.5, 0.25, add, sub, mul, div, sin)
CPU params: Intel Core i3-2310M (3MB Cache, 2.10 GHz) OS: Arch Linux 64bit

Problem	TV[1]	M0	M1	M2	M3	M4	M5
3x3 ADD	1	12398	8264	8857	1432	6529	5057
4x4 ADD	4	11290	7865	4290	1361	4616	3877
5x5 ADD	16	8381	6053	1229	787	1814	1704
6x6 ADD	64	4026	3788	327	337	544	551
4x4 MUL	4	10439	7137	3479	1017	3638	2953
5x5 MUL	16	7541	5099	1075	732	1458	1366
6x6 MUL	64	3540	2400	236	238	359	368
PARITY	8	12508	8665	4876	3876	6833	7095
LOG(x)	5	14246	-	5228	3562	5390	6162
LOG(x)	10	12545	-	4314	1990	5637	6559
LOG(x)	25	9279	-	1928	1427	3155	3465
LOG(x)	50	6363	-	991	921	1836	2231
LOG(x)	100	3984	-	543	630	1069	1226
LOG(x)	200	2268	-	195	337	422	587
LOG(x)	500	998	-	110	166	230	270
LOG(x)	1000	506	-	34	87	80	114
COS(x)	1000	519	-	63	82	130	143
SIN(x)	1000	519	-	58	83	125	141
LOG(x)	1000	506	-	34	87	80	114
XXYZ(x)	1000	524	-	68	127	140	144

Table C.1: The number of evaluations per second in Python3. M0 - no fitness calculation

Problem	TV	M0	M1	M2	M3	M4	M5
3x3 ADD	1	13694	7297	9426	1700	7342	5150
4x4 ADD	4	12611	6952	4306	1559	4961	3965
5x5 ADD	16	10108	5021	1221	907	1939	1783
6x6 ADD	64	5246	2173	355	442	583	595
4x4 MUL	4	11580	6193	3709	1349	4127	3152
5x5 MUL	16	8929	3877	954	719	1424	1252
6x6 MUL	64	4514	1472	239	276	374	376
PARITY	8	13829	6678	5471	4911	7779	7987
LOG(x)	5	18149	-	7250	2991	8406	9347
LOG(x)	10	16480	-	4354	4525	6469	6915
LOG(x)	25	12595	-	1993	3327	3889	4555
LOG(x)	50	8928	-	1249	2156	2695	3269
LOG(x)	100	5686	-	564	1300	1358	1708
LOG(x)	200	3319	-	286	521	737	915
LOG(x)	500	1472	-	114	315	304	317
LOG(x)	1000	758	-	58	114	154	196
COS(x)	1000	762	-	40	132	115	176
SIN(x)	1000	762	-	43	132	122	188
LOG(x)	1000	758	-	58	114	154	196
XXYZ(x)	1000	758	-	52	130	141	193

Table C.2: The number of evaluations per second in Python2

Problem	TV	M0	M1	M2	M3	M4	M5
3x3 ADD	1	20456	24105	12130	858	8429	6502
4x4 ADD	4	32458	15016	8564	763	4357	4173
5x5 ADD	16	42089	9958	2825	402	1306	1380
6x6 ADD	64	17794	3562	803	184	349	368
4x4 MUL	4	51083	16538	9315	677	4447	4297
5x5 MUL	16	37742	7810	2152	341	1121	1168
6x6 MUL	64	17275	2262	511	137	277	295
PARITY	8	61059	17996	22319	2301	4450	4338
LOG(x)	5	39520	-	17521	1370	6277	6253
LOG(x)	10	41085	-	11941	1660	7955	3902
LOG(x)	25	40903	-	7982	1037	4570	1869
LOG(x)	50	25739	-	6048	621	2873	998
LOG(x)	100	14389	-	2563	364	1311	490
LOG(x)	200	7714	-	1339	156	666	246
LOG(x)	500	3091	-	547	81	268	46
LOG(x)	1000	1674	-	274	33	136	12
COS(x)	1000	1625	-	158	37	100	20
SIN(x)	1000	1617	-	175	36	107	5
LOG(x)	1000	1674	-	274	33	136	11
XXYZ(x)	1000	1623	-	224	36	123	8

Table C.3: The number of evaluations per second in PyPy

Problem	TV	M6	M7	M8
3x3 ADD	1	9033	200000	1250000
4x4 ADD	4	4957	66666	606060
5x5 ADD	16	1732	18180	188676
6x6 ADD	64	481	4876	45556
4x4 MUL	4	4327	50000	333332
5x5 MUL	16	1523	15384	117644
6x6 MUL	64	434	3772	34780
PARITY	8	4493	200000	666642
LOG(x)	5	5569	200000	555552
LOG(x)	10	3334	100000	400000
LOG(x)	25	1554	50000	202020
LOG(x)	50	809	18180	101052
LOG(x)	100	413	8692	50540
LOG(x)	200	210	4876	25242
LOG(x)	500	84	1376	12500
LOG(x)	1000	43	920	6000
COS(x)	1000	42	1500	12000
SIN(x)	1000	43	2400	12000
LOG(x)	1000	43	920	6000
XXYZ(x)	1000	80	1712	12000

Table C.4: The number of evaluations per one second in Cython

Legenda

M0 – without fitness calculation
M1 – parallel simulation
M2 – semi-parallel simulation (64-bit)
M3 – JIT compilation via `compile()`
M4 – JIT compilation into infix notation
M5 – JIT compilation into postfix notation
M6 – Cython with dynamic typing
M7 – Cython with static typing
M8 – Cython with static typing with rolled up functions
TV – Training Vectors.

Appendix D

User Manual

After decompression of the archive, the module can be compiled with `make py2` for Python2. In case of Python3, run it with `make py3` or `make`.

Import the module in the interpreter `import Cgp`. This module contains a class `Cgp`. For show this help, you can type `print(Cgp.__doc__)` in the interpreter.

Syntax and sematic of the methods should be same for all of the versions (M5, M6, M7). However, M7 differs in one thing. User can not create his own function set directly in the Python (see 4.1).

D.1 Constructor

```
__init__(rows, cols, lback, in, out)
```

Constructor creates an object of the class. Its parameters are not rquired: Number of graph rows (default 1), number of graph cols (default 40), levelback (default 40), input data, output data. Input and output data are represented as a list of integers in the case of circuit design, for example `input = [in1, in2, in3]`, where each element of the list `in1`, `in2`, `in3` is an integer represented the input vectors of genetic programming. In the case of symbolic regression, data are represented as a list of dimensions, where each dimension contains a list of real numbers. For example `input = [in1, in2, in3]`, where `in1`, `in2`, `in3` are lists of doubles (training data) for each dimension. The number of training vectors should be same for each dimension. In the other words, $len(in_1) = len(in_2) = len(in_3) = \ldots = len(in_i)$ where i is the amount of dimensions.

D.2 Graph change

```
graph(rows, cols, lback)
```

This method changes the graph of CGP. It accepts two required parameters: row count and columns count. Third parameter is not required and it gives us the number of interconnection between the columns (levelback). If it is not specified then levelback is equal to number of columns (maximal interconnection).

D.3 Reading input data from the file

`file(filename)`

With this method, we can read a file with training data for genetic programming. The syntax should be:

- the comment begins with #
- empty lines are ignored
- in the case of symbolic regression dimensions are divided by a space
- in the case of circuit design spaces can be ignored
- input and output training vectors are divided by a colon character `inputs : outputs`
- example for symbolic regression
  ```
  0.1 6.5 : 1.0
  1.5 1.6 : 0.0
  6.5 1.6 : 1.0
  7.5 6.5 : 0.5
  ```
- example for circuit design
  ```
  00 : 0
  01 : 0
  10 : 0
  11 : 1
  #  yes it is an conjunction
  ```

D.4 Data change

`data(input, output)`

This method changes training data. The arguments are input data, output data in the format described in the constructor

D.5 Running the algorithm for symbolic regression

`run(generation, population, mutations, acc, runs)`

Arguemnts are: The number of generations, Population size, the number of mutations, the accuracy of symbolic regression. The number of runs is not required, default is one run.

D.6 Running the algorithm for circuit design

`run(generation, population, mutations, runs)`

Arguemnts are: The number of generations, Population size, the number of mutations, the number of runs (not required), default is one run.

D.7 Function set change

The object of the class `Cgp` contains an attribute `function`. This variable has a function set class. User can change each function in the function set. For circuit design, we can use these inbuilt function sets:

- `allLogicalOperations()` – (id, and, or, xor, not, nand, nor, nxor)
- `booleanAlgebra()` – (id, and, or, not)
- `reedMuller()` – (id, or, xor, not)
- `moje()` – (id, or, xor, and)
- `nandOnly()` – (id, nand)

For symbolic regression:

- `symbolicRegression()` – (0.25, 0.50, 1.00, id, add, sub, mul, div)
- `symbolicRegressionWithSin()` – (0.25, 0.50, 1.00, id, add, sub, mul, div, sin)

For example with command `cgp.function.booleanAlgebra()`, we select Boolean algebra as a working function set. In default, the `allLogicalOperations()` is selected in case of circuit design or `symbolicRegressionWithSin()` in case of symbolic regression.

User can create also his own function set objects, however he need to add these attributes:

- `type` – type of functions set (symbolic regression or circuit design)
- `set` – list of the functions
- `arity` – arity of the functions
- `table` – function names which are used in printing of chromosome

Note that user can not create function with bigger arity than two.

D.8 Prining Results of CGP

- `resultChrom()` – returns a chromosome in the syntax for CGPTools
- `showChrom()` – returns a chromosome in ASCII syntax
- `bestFitness` – the best found fitness value
- `chrom` – returns a chromosome as described in the beginning of the work
- `elapsed` – time elapsed
- `evalspersec` – the number of evaluations per one second