FEDERAL UNIVERSITY OF RIO DE JANEIRO
INSTITUTE OF MATHEMATICS
BACHELOR DEGREE IN COMPUTER SCIENCE

LETICIA F. DE FIGUEIREDO

PyCoptimizer
A framework to optimize codes

RIO DE JANEIRO
2019

LETICIA F. DE FIGUEIREDO

PyCoptimizer
A framework to optimize codes

Final project of an undergraduate degree paper presented to the Computer Science Department of the Federal University of Rio de Janeiro as part of the requirements to obtain a Bachelor of Science degree in Computer Science.

Supervisor: Prof. João C. P. da Silva, D. Sc.
Co-supervisor: Daniel Hugo Cámpora Pérez, PhD
Co-supervisor: Fabrício Firmino de Faria, M. Sc.

RIO DE JANEIRO
2019

F475p

Figueiredo, Leticia Freire de
PyCoptimizer: a framework to optimize codes / Leticia Freire de Figueiredo. – 2019.

41 f.

Orientador: João Carlos Pereira da Silva.
Coorientador: Daniel Hugo Cámpora Pérez.
Coorientador: Fabrício Firmino de Faria.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2019.

1. Flags de otimização. 2. Algoritmo genético. 3. Algoritmos evolucionários. I. Silva, João Carlos Pereira da (Orient.). II. Cámpora Pérez, Daniel Hugo (Coorient.). III. Faria, Fabrício Firmino de (Coorient.). IV. Universidade Federal do Rio de Janeiro, Instituto de Matemática. V. Título.

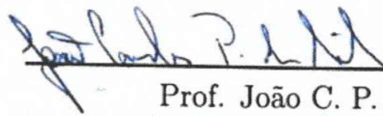
LETICIA F. DE FIGUEIREDO

PyCoptimizer
A framework to optimize codes

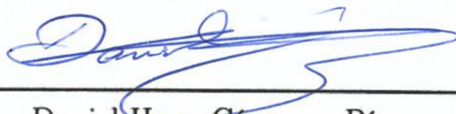
Final project of an undergraduate degree paper presented to the Computer Science Department of the Federal University of Rio de Janeiro as part of the requirements to obtain a Bachelor of Science degree in Computer Science.

Approved in December, 11th 2019

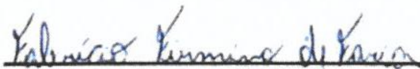
EXAMINATION COMMITTEE:



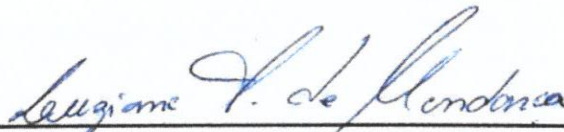
Prof. João C. P. da Silva
D. Sc. (UFRJ)



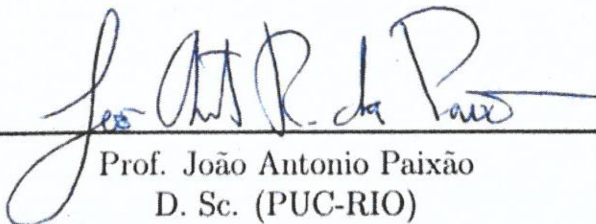
Daniel Hugo Cámpora Pérez
PhD (Universidad de Sevilla and CERN)



Fabrício Firmino de Faria
M. Sc. (UFRJ)



Prof. Luziane Ferreira de Mendonça
D. Sc. (UNICAMP)



Prof. João Antonio Paixão
D. Sc. (PUC-RIO)

RESUMO

Possuindo um código, desejamos otimizá-lo. Isso pode ser feito de diversas maneiras. Uma dessas maneiras é no momento da compilação, passar comandos, indicando como queremos compilar o código. Esses comandos incluem as flags de otimização. Mas não é fácil encontrar as flags que melhor otimizam o código. Uma solução para esse problema é usar o algoritmo genético, procurando o melhor conjunto de flags de otimização. No nosso trabalho, propomos um framework, chamado PyCoptimizer, no qual busca o melhor conjunto de flags para um determinado código. Comparamos nossos resultados com flags de otimização genéricos, como a flag -O1 em C ++, e mostramos que o framework alcança um resultado melhor.

Palavras-chave: flags de otimização. algoritmo genético. algoritmos evolucionários.

ABSTRACT

When having a code, we wish to optimize it. It can be done in many ways. One of these ways is in the compilation moment, passing commands, indicating how we want to compile the code. These commands include the optimization flags. But, finding the flags which best optimizes the code is not easy. One solution to this problem is to use the genetic algorithm, searching for the best set of optimization flags. In ou work, we propose a framework, called PyCoptimizer, in which searches the best set of flags for a given code. We compare our results with generic optimization flags, as -O1 flag in C++, and shown the framework reach a better result.

Keywords: optimization flags. genetic algorithm. evolutionary algorithm.

LIST OF FIGURES

Figure 1 – Example of the genetic algorithm 12
Figure 2 – Example of one individual 13
Figure 3 – Example of the tournament selection with q = 3 individuals 14
Figure 4 – Example of the exponential crossover 15
Figure 5 – Mutation example 15
Figure 6 – Bit inversion example 16
Figure 7 – Example of command to install PyCoptimizer 17
Figure 8 – Example of command to run the Python code 20
Figure 9 – PyCoptimizer schema 20
Figure 10 – Running with default parameters of genetic algorithms 24
Figure 11 – Experiments changing the genetic algorithm parameters 25
Figure 12 – Changing mutation rate using the same population 27
Figure 13 – Changing crossover rate using the same population 28
Figure 14 – Decreasing crossover rate with constant mutation rate 29
Figure 15 – Increasing crossover rate with a constant mutation rate 30
Figure 16 – Decreasing mutation rate with a constant crossover rate 31
Figure 17 – Increasing mutation rate with a constant crossover rate 32
Figure 18 – Results to Cross Kalman code using PyCoptimizer 32
Figure 19 – Frequency of 20 flags with mutation rate equals to 0.01 39
Figure 20 – Frequency of 20 flags with mutation rate equals to 0.001 39
Figure 21 – Frequency of 20 flags with crossover rate equals to 0.8 40
Figure 22 – Frequency of 20 flags with crossover rate equals to 0.6 40
Figure 23 – Frequency of 20 flags about the case study using 30 generations 41

LIST OF CODES

3.1 Python code example 19
4.1 Genetic algorithm of Pygmo taken from Pygmo - Simple Genetic Algorithm in 06/18/2019 26

LIST OF TABLES

Table 1 – Genetic algorithm parameters . . . 22
Table 2 – List of experiments . . . 23
Table 3 – Comparison time . . . 24
Table 4 – Results of each experiment . . . 26
Table 5 – Problemming flags . . . 29
Table 6 – Results to case study . . . 30
Table 7 – Duration of each running per experiment . . . 33

LIST OF SYMBOLS

$\forall$	For all
$\sum$	Sum
$\in$	Belongs to

TABLE OF CONTENTS

1 INTRODUCTION **10**

2 BASIC CONCEPTS **12**

2.1 Genetic Algorithm 12

2.1.1 Selection **13**

2.1.2 Crossover and Mutation **14**

3 IMPLEMENTATION **17**

4 EXPERIMENTS **22**

4.1 Experiment 1 - Minimizing Execution Time 22

4.2 Analysis of the resulting flags 27

4.3 Experiment 2 - Cross-Kalman 28

5 CONCLUSION **33**

REFERENCES **35**

APPENDIX A – FLAGS USED IN THE EXPERIMENTS **37**

APPENDIX B – FREQUENCY OF FLAGS **39**

APPENDIX C – FREQUENCY OF FLAGS - CASE STUDY . . . **40**

1 INTRODUCTION

A code is a sequence of instructions that need to be compiled to be interpreted by the computer. When compiling, the instructions are translated into a language called machine language, transforming code, written in a language better understood by humans, in machine language. For the compilation to be performed a compiler is used.

Many commands can be passed to the compilers when compiling. One possible command is the name we wish to give to our file after the compilation. Other commands which can be passed are the optimization flags. The optimization flags potentially make the generated executable more efficient. Different optimization flags have different objectives, being some of them: reduce the execution time in a compiled code and reduce the occupied memory space.

A code written in C/C++ allows many optimization flags being used. Each flag will apply a different optimization in the code. Optimization flags affect each other in an unpredictable manner that varies across compilers, compiler versions, and programs. Therefore, one possible way to identify the best set of optimization flags is to automatize the search process.

The problem to find the best set of optimization flags is a subject already discussed. In (HOSTE; EECKHOUT, 2008), the COLE: Compiler Optimization Level Exploration was presented which objective is to automate the search for optimal levels of compilation using multi-objective search. The compilation levels are sets of optimization flags and their optimal levels indicated whose flags of each level are active or not. This work uses the SPEA2 algorithm, which is an "elitist evolutionary algorithm that is inspired by genetic algorithms".

(PAN; EIGENMANN, 2006) also searches automatically the best combination of optimization techniques or optimization flags. To reach their objective, they construct an algorithm called Combined Elimination (CE).

In (COOPER; SCHIELKE; SUBRAMANIAN, 1999), the authors search the best set of optimization flags to reduce the space used when generating the executable program. They used a genetic algorithm to reach the solution but focusing only on reducing the used space by the code in the final of the compilation. In (ALMAGOR et al., 2004), the authors reached the conclusion that one sequence of optimization flags specifics to each code can optimize better in the compilation than a generic sequence. During the project, they aim to discover if it is effective to search for the best sequence of optimization flags for each case. They use some algorithms to find the solution and one of them is the genetic algorithm.

In (COOPER; SUBRAMANIAN; TORCZON, 2002), the authors constructed a system to find the best set of optimization flags. The flags are chosen with the intention

of minimizing one optimization goal, inherent in the code being used. An example of an optimization goal is the execution time of the program generated by the compilation. In this case, the system aims to find the flags that minimize the execution time. Another example is to minimize the code size produced by the compilation.

A way to automate this choice is by using a genetic algorithm. The genetic algorithm searches, in a field of feasible solutions, the best solution for the presented problem using the natural process of selection and generation of individuals. It can be used in many cases, where you want to find parameters that optimize the problem (TANG et al., 1996). This type of solution was already discussed by others researchers (ALMAGOR et al., 2004), (COOPER; SCHIELKE; SUBRAMANIAN, 1999), (HANEDA; KNIJNENBURG; WIJSHOFF, 2005).

In this work, a Python framework was built where, given the code in any language, a set of optimization flags and other parameters, using the genetic algorithm tries to achieve the best solution for the problem, inside the field of feasible solutions. The problem is defined by stating what you want to optimize in the resulting code. Two experiments will be made throughout the work, each one having a different purpose. These experiments will show us if the genetic algorithm really finds the best solution for each case.

This work is organized as follows: in chapter 2, we explain important concepts for the course of the project; in chapter 3, we give an overview of the framework and how it was constructed; in chapter 4, we show the experiments carried out and the reached results and in the chapters 5, we give our conclusion about the project described here and the next steps to improve the project.

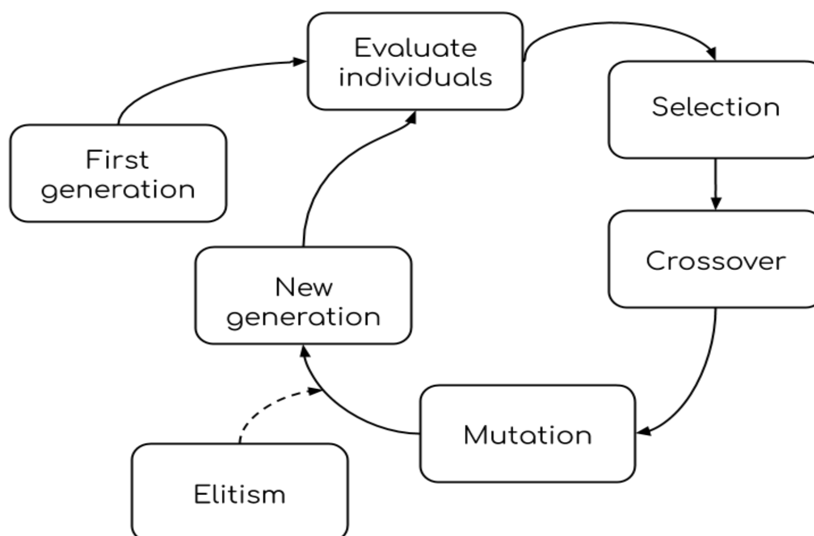
2 BASIC CONCEPTS

Optimization, according to Google [1], is "the creation of more favorable conditions for the development of something, or, the process by which the best value of a quantity is obtained." In computing, optimization is more similar to the second definition: given an optimization problem, we look for a solution that will generate the best possible result for the problem. An example of an optimization problem is the traveling salesman. In this problem, we want the traveling salesman to pass by various cities, passing by once by each of them and making the traveled distance the smallest possible. One of the ways to find the best solution for this problem is using evolutionary optimization (FOGEL, 1988), with evolutionary algorithms.

2.1 GENETIC ALGORITHM

Evolutionary algorithms use computational models based on the natural evolution with the finality to find the solution for the optimization problems. One type of evolutionary algorithm is the genetic algorithm (LINDEN, 2008) and it describes how the natural evolution works. In Figure 1, we have a schema about the genetic algorithm flow, with all its steps.

Figure 1 – Example of the genetic algorithm

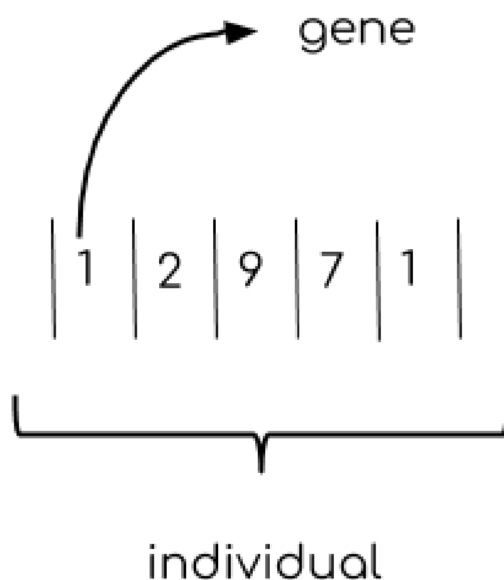


Given an optimization problem, we adapt it to one language where the genetic algorithm can understand. One example of the problem is the minimization of the sum of

[1] https://www.google.com/search?q=otimiza%C3%A7%C3%A3o+significadooq=otimiza%C3%A7%C3%A3o +significadoaqs=chrome..69i57j0l5.6253j1j9sourceid=chromeie=UTF-8. Accessed on 09/07/2019.

5 integer numbers between 1 and 10. In order for the genetic algorithm can understand this problem, we need to translate the given information for the structures that can be understood to the algorithm. One of these structures is the individual, which is modeled according to the possible solutions for the problem. One individual is a group of genes and their genes take this information. In the example, the genes will be values between 1 and 10 and the individual will be the set of these genes.

Figure 2 – Example of one individual



The initial population will be formed by a group of individuals and it will be part of the initial generation. After the formation of the population, each individual will be evaluated from an evaluation function within the environment in which it is inserted. This evaluation function is defined according to the problem in which we want the solution. In the problem we are looking for, the evaluation function will be the sum of all genes of an individual, as like in Equation 1:

$$f(individual) = \sum_{\forall\ gene\ \in\ individual} gene \tag{2.1}$$

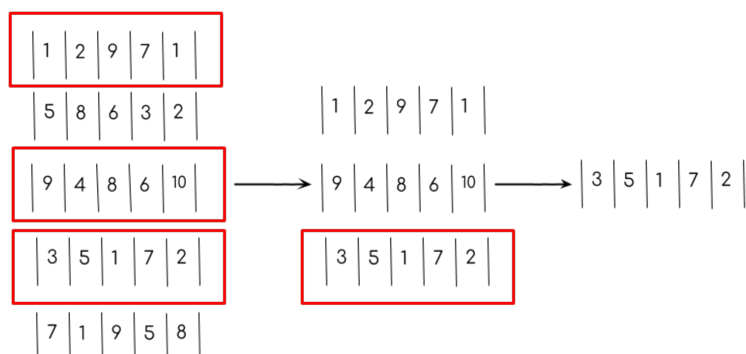
The function in Equation 1 receives one individual as a parameter and returns the sum of the genes of this individual. This is one way of evaluation of the individual. At the end of the evaluation of the individuals, will be made the selection for the next generation.

2.1.1 Selection

The selection aims to select the individuals that can have the best evaluation in the actual generation, to generate the individuals of the next generation population, after experience the crossover and mutation operations. In the example we are using, the prioritized individuals in the selection will be those who will have the smallest sums.

The selection uses a probabilistic method to choose the individuals (Back; Hammel; Schwefel, 1997). There are many methods implemented for the selection, being one of them the tournament selection. The tournament selection collects one random uniform sample of the individuals from the population. This sample should be size equals to q individuals, with q > 1. The best individual of this sample is selected for the next generation population. That is the individual with the best evaluation. This process is repeated until the next generation population is completed (Back; Hammel; Schwefel, 1997).

Figure 3 – Example of the tournament selection with q = 3 individuals



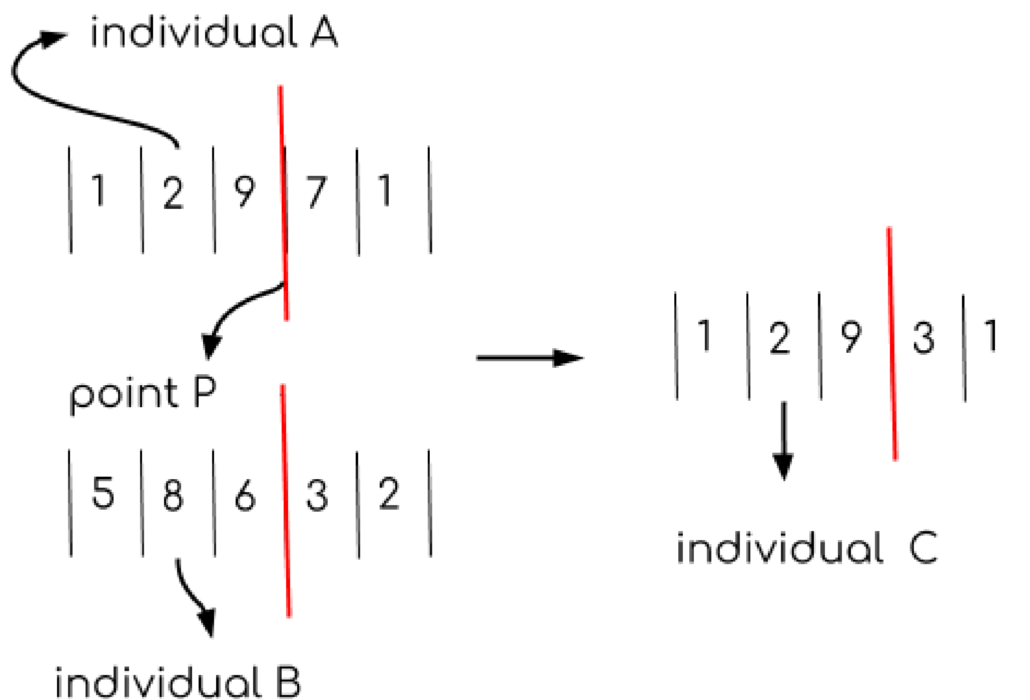
In figure 3, we have an example of tournament selection, using one sample with q=3 individuals. As the figure shows, within the current population, the individuals are chosen aleatory to compose one sample. The choices are made until the sample have a number of individual equal to q. After the choice, it is selected as the best individual within the selected sample.

2.1.2 Crossover and Mutation

After the selection, the crossover is the next step for the genetic algorithm. The crossover consists of constructing a new individual using parts of two individuals already existing in the population. One method used in the crossover is the exponential. In the exponential method, having two individuals A and B, an aleatory point P is chosen in the individual A. The new individual that is being constructed, we will call C, receives the genes present in the individual A. Since the point P, individual C is filled with the genes of the individual B following one probability p_c, which can be defined. The probability p_c is associated with the choice of each gene of B to be present in C. Generally, the crossover probability is high to the possibility of the creation of new individuals.

In figure 4, we have the individual A and B. The point P is chosen between the third and fourth genes in the individuals. The individual C receives the same genes from the individual A has until the point P. After this point, the genes of C can be filled with the genes of individual B or A, depending on the probability p_c.

Figure 4 – Example of the exponential crossover



Another step, after the crossover, is the mutation. In the mutation, the genes of some individuals can be modified, assuming the value of other possible genes within the solution. This mutation occurs according to a probability p_m. The mutation probability is indicated to be low, since there may be a loss of interesting characteristics in the population.

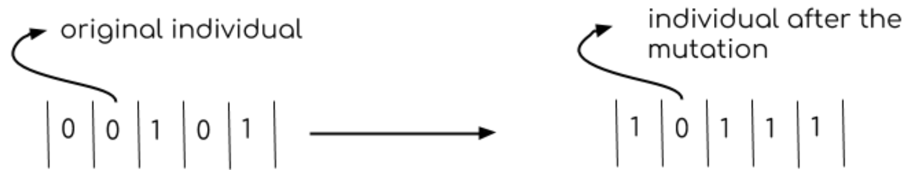
In our problem, after the individual is chosen, their genes are modified, assuming values between 1 and 10.

Figure 5 – Mutation example

Another very common way of mutation is the bits invesion. When the individual is constructed using only 0 and 1 in their genes, the mutation only converts their genes, following the probability rate.

When finalized the crossover and the mutation, in some cases, the genetic algorithm uses the elitism. The elitism passes the best individual of the current population to the next generation population. This ensures the best individual in one generation appe-

Figure 6 – Bit inversion example



ars in the following generation and the higher evaluation in the population in the next generations won't be smaller than in the current population (ASHLOCK, 2010).

After completing all these steps, we can say that one generation was completed. One new population was constructed to be initialized the next generation. The end of this loop - conclusion of one generation - will be defined according to the stop criteria. The stop criteria for the genetic algorithm can be defined in many ways, among them:

1. the genetic algorithm can finalize when a certain number of generation are reached or

2. the end can be indicated when the difference between the best individual of the generations (N-1) and N is very small.

The genetic algorithm is used in cases where the problem solution by classic ways is impracticable or when there are no deterministic solutions to the problem (Fogel, 1994). But the algorithm doesn't guarantee the best solution will be found. The genetic algorithm is a local search. Thus, if the global maximum or minimum is required, genetic algorithms don't guarantee it will be found. It is important to say the genetic algorithm doesn't have only one solution but a set of possible solutions (Back; Hammel; Schwefel, 1997).

3 IMPLEMENTATION

When compiling code, we can specify optimization flags. These flags have specifics purposes - some flags aim to decrease the used space by the executable program, others, decrease the total execution time - and, in the end, generate an optimized executable program. But, given the number of optimization flags (more than 150 possible flags), finding a set of flags that optimizes a given code can be very time consuming. In this chapter, we present the PyCoptimizer [2], a framework that used a genetic algorithm in order to find a set of flags that generates an optimized executable program.

PyCoptimizer uses the Pygmo [3] library which provides the genetic algorithm for our framework. Pygmo is a python version of the Pagmo library, written in C++. The library is constructed "around the idea of providing a unified interface to optimization algorithms and to optimization problems and to make their deployment in massively parallel environments easy".

After installing the framework - the example is in Figure 7 - the user must provide information about the code to be optimized and the genetic algorithm parameters in a file with .py extension, which will be used.

Figure 7 – Example of command to install PyCoptimizer

```
pip install PyCoptimizer
```

In this file, the user has to import the classes UserParameters and Coptimizer, provided by the PyCoptimizer framework and any other libraries he wishes. After, the user should define a class with the name he wishes and implements the abstract methods present in UserParameters. One of these methods is the `_init_`. The following parameters should be set: the crossover probability, the mutation probability, the number of individuals in the population, the number of genes in each individual and if the user wants to minimize or maximize your optimization goal. Additionally, the user must define the evaluation function (called `evaluation_function` in the framework). This method must return the value which will be optimized by the genetic algorithm.

In Code 3.1, we show an example of the file passed to COptimizer. The user made a class called Test, which inherits from the UserParameters abstract class. The `_init_` method has a dictionary with the parameters of genetic algorithm: number of generations (50 generations), number of individuals in the population (30 individuals), number of genes in each individuals (3 genes); crossover rate (30%) and mutation rate (10%). It's

[2] https://pypi.org/project/PyCoptimizer/. Accessed on 10/03/2019.
[3] https://esa.github.io/pagmo2/index.html. Accessed on 10/03/2019.

also defined the path to compile and clean the executable files from the goal code. In the example, both the `compile_path` and `clean_path` gets the current directory. The commands to compile and clean the code are defined as `make` and `make clean`, in the `compile_command` and `clean_command` variables, respectively.

The user must declare if he wants to minimize or maximize the evaluation function, using the `min_max` variable. When the variable is defined as 1, the objective is to minimize the evaluation function; if it's -1, the objective is to maximize. In our example, the user wants to minimize. Finally, the user defines the flags, using the `flags_list` variable, which will be used in the optimization.

In the `evaluation_function` function, the user implements how to evaluate the code. In the example, the function gets the execution time of the executable code after compiled and returns this number.

Code 3.1 – Python code example

```python
from PyCoptimizer.user_parameters import UserParameters
import os
import subprocess
import time
from PyCoptimizer.coptimizer import COptimizer

class Test(UserParameters):
    """ Class to test the abstract class UserParameters """

    def __init__(self):
        super(Test, self).__init__()
        self.dict_optimization = {"no_generations": 50,
                                  "no_pop": 30,
                                  "individual_size": 3,
                                  "crossover_rate": 0.3,
                                  "mutation_rate": 0.1}
        self.compile_path = os.getcwd()
        self.clean_path = os.getcwd()
        self.compile_command = "make"
        self.clean_command = "make clean"
        self.min_max = 1 #minimize
        self.flags_list = ["-falign-functions",
                           "-falign-jumps",
                           "-falign-labels",
                           "-falign-loops",
                           "-fauto-inc-dec", "-fbranch-probabilities",
                           "-fbranch-target-load-optimize", "-fbranch-target-load-
                               optimize2",
                           "-fbtr-bb-exclusive","-fcaller-saves"]

   def arguments_to_run_code(self):
        """ Implementing the method arguments_to_run_code.
            This method returns the command to run the executable file"""
        return"./nbody.gpp-3.gpp_run 50000000"

    def evaluation_function(self):
        """ Implementing the method evaluation_function.
            This method returns the value to be optimized by the framework.
            In this method, the user needs to run the executable file
            and get the value to pass to the framework."""
        def_path = os.getcwd()
        os.chdir(self.compile_path)
        my_env = os.environ.copy()
        my_env["PATH"] = "/usr/sbin:/sbin:" + my_env["PATH"]
        my_command = self.arguments_to_run_code()
        # counting time
        t0 = time.time()
        p = subprocess.Popen(my_command, shell=True,
                             stdout=subprocess.PIPE)
        out, err = p.communicate()
        # finalizing
        value = time.time() - t0
        os.chdir(def_path)
        return value

if __name__ == '__main__':
    COptimizer().main()
```

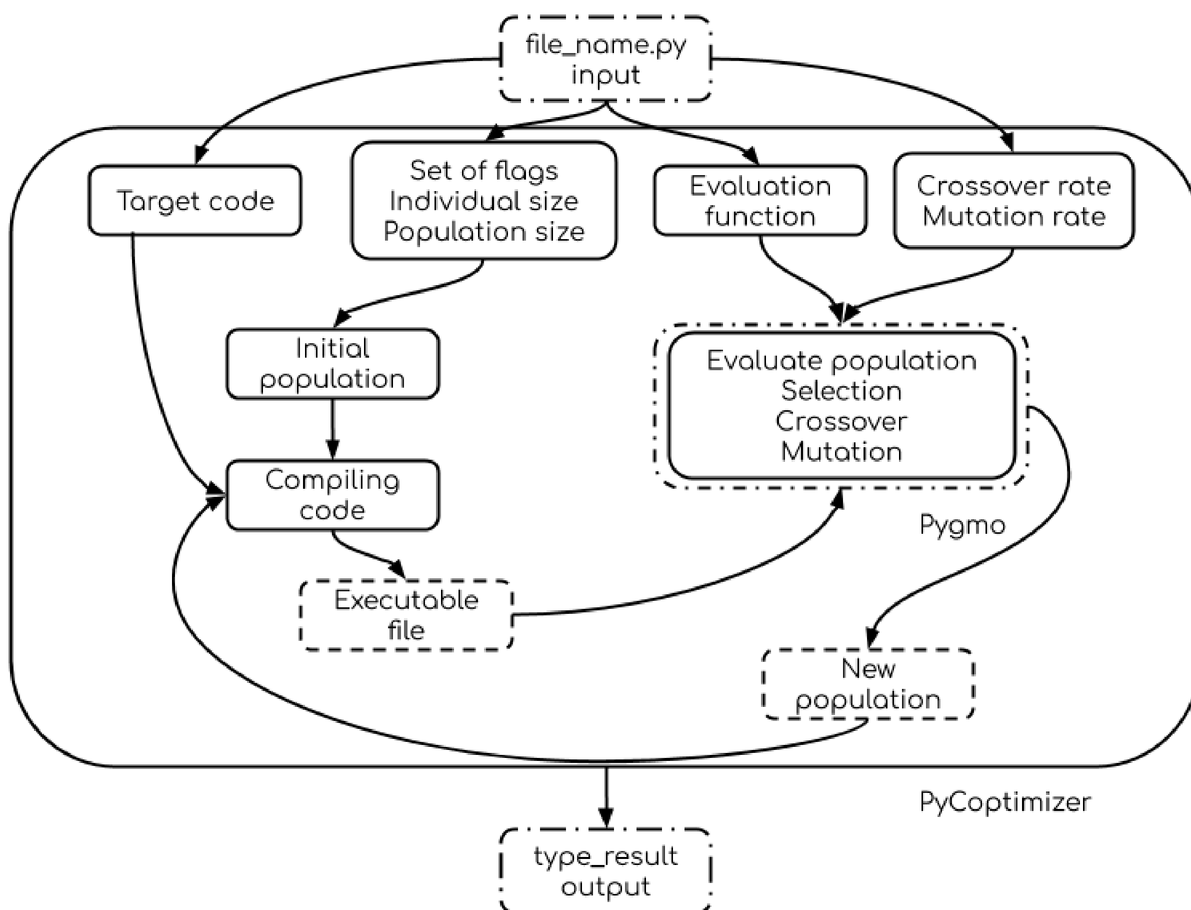
After the user builds the file teste.py, he can execute it using python teste.py type_result, where type_result can be graphic or log.

Figure 8 – Example of command to run the Python code

```
python teste.py type_result
```

This command will initialize the framework execution, collecting the present information in the teste.py file and use it in the class COptimizer. Initially, as shown in Figure 8, the framework uses the provided information (the set of flags, the individuals and the population sizes) to construct the initial population. The activation of the flags in the individuals of the initial population is made using the Python method random. The individuals use binary representation, where 1 means that the flag should be used and 0 means that the flag should not.

Figure 9 – PyCoptimizer schema



Let k be the number of generations we want to consider, $i = 0$ and P_0 be the initial population. While $i < k$,

1. For each individual `ind` in the population P_i:

    a) Construct the compile command referent to the individuals *ind*: whenever a gene in *ind* has value 1, the corresponding optimization flag will be integrated into the compilation command; otherwise, the flag will not be included.

    b) Compile the code using the constructed compile command.

    c) Run the compiled code to evaluate individual *ind* according to the defined evaluation function.

2. The evaluated population P_i is passed to the Pygmo library to apply the genetic operators (selection, crossover and mutation).

3. Let $i = i+1$ and set the next generation returned by Pygmo to P_i.

The reached results by the framework are returned in two possible ways:

1. in graphic, presenting the metric evolution which is being optimized over the generations

2. in a log, showing the best values in each generation.

In both cases, it is introduced in the terminal window, or where the PyCoptimizer is running, the final set of optimization flags found in the last generation by the genetic algorithm.

4 EXPERIMENTS

In the following section, we will present 2 experiments, using the developed framework to find the best set of flags in each case. In Section 4.1, we wish to find a set that minimizes the execution time of the compiled file. In Section 4.2, we used the PyCoptimizer to look for the flags which maximize the number of events processed by the code, in a second. In this last section, beyond the obtained results by our experiments, we will present the results of other experiments using this same code. We ran our experiments in a Linux Mint 64 bit with 24 cores Xeon of 3.4GHz with 64GB ram.

4.1 EXPERIMENT 1 - MINIMIZING EXECUTION TIME

The code used in our experiments came from The Computer Language Benchmarks Game site [4]. This site has a set of problems and different implementations to solve them, with some performance measurements such as execution time and amount of used cpu. It is possible to use the available code and reproduce the execution, using the same command to compile and execute.

For the tests in this section, we chosen codes made in C++, instead of other languages because C++ has an extensive list of optimization flags [5], making possible more combinations. The site shows 10 problems [6] with codes in C++ and we selected a code that creates binary trees [7] using the minimum number of allocations. Our goal is to minimize the execution time.

Table 1 – Genetic algorithm parameters

Parameter	Value
Individual size	190 genes
Individual representation	Binary
Population size	65 individuals
Number of generations	50 generations
Crossover method	Exponential
Mutation method	Polynomial
Crossover rate	0.7
Mutation rate	0.005
Elitism	Yes

The parameters shown in Table 1 were used in the genetic algorithm. In the experiments, we modify these parameters in order to determine which are the best combination

[4] https://benchmarksgame-team.pages.debian.net/benchmarksgame/. Accessed on 10/20/2019.
[5] https://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_3.html#SEC13. Accessed on 10/20/2019
[6] https://benchmarksgame-team.pages.debian.net/benchmarksgame/measurements/gpp.html. Accessed on 10/20/2019.
[7] https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/binarytrees-gpp-2.html. Accessed on 10/20/2019.

to be used.

One of the parameters for the genetic algorithm is the amount of genes present in the individual, corresponding to the number of flags considered. The population size should be a value that allows the generation of a big diversity of individuals to guarantee the solution and, at the same time, it does not take a long time to find the solution at the final of the generations (ASHLOCK, 2010), (LINDEN, 2008). In this case, as the individual size is fixed and equals 190 genes, it has the possibility of 2^{190} different individuals. The reasonable population size must be between [50, 100] individuals (Fogel, 1994) because the execution time of GA increases as the population size grows - and if this number is small, we do not have great genetic variability in our population (Fogel, 1994). So, we set the population size equals to 65 individuals for the base experiment.

Another important parameter to the genetic algorithm is the crossover rate. The best rates for this parameters lies between [0.6, 0.95] (Fogel, 1994), (Back; Hammel; Schwefel, 1997). A low crossover rate is only recommended for a high population size; when the population size is equal to 200 individuals, usually the best performance is achieved with low crossover and mutation rates (SCHAFFER et al., 1989). For this experiment, we set the rate equals to 0.7.

Unlike the crossover rate, the mutation rate cannot be high, because the genes can lose interesting characteristics. The most recommended values to mutation rate lies between [0.001, 0.01] (Fogel, 1994), (Back; Hammel; Schwefel, 1997). We defined the mutation value equals to 0.005 to be the default rate in the experiments.

In Table 2, we present all the experiments we did in this section. In each one of the four experiments, only one parameter of the genetic algorithm was changed.

Table 2 – List of experiments

Experiment	Sub-Experiment	Mutation rate	Crossover rate	Population size	Number of generations
#1	#1.1	increase rate to 0.01	x	x	x
	#1.2	decrease rate to 0.001	x	x	x
#2	#2.1	x	increase rate to 0.8	x	x
	#2.2	x	decrease rate to 0.6	x	x
#3	#3.1	x	x	increase size to 80 individuals	x
	#3.2	x	x	decrease size to 50 individuals	x
#4	#4.1	x	x	x	increase to 70 generations
	#4.2	x	x	x	decrease to 30 generations

The idea is to modify the parameters presented in Table 2 to check the impact of each one of them in the results. For each test, we ran 4 executions using different initial populations. The list of flags used in all strategies can be seen in Appendix A.

In the experiment, our goal is to minimize the execution time in the generated executable program. Table 3 shows the running time of the compiled code without any kind of optimization and with the -O1 optimization flag. The goal of the flag -O1 is to reduce the compiled code size and the execution time, but these optimizations are made in a way to not spend a lot of time in the compilation process [8].

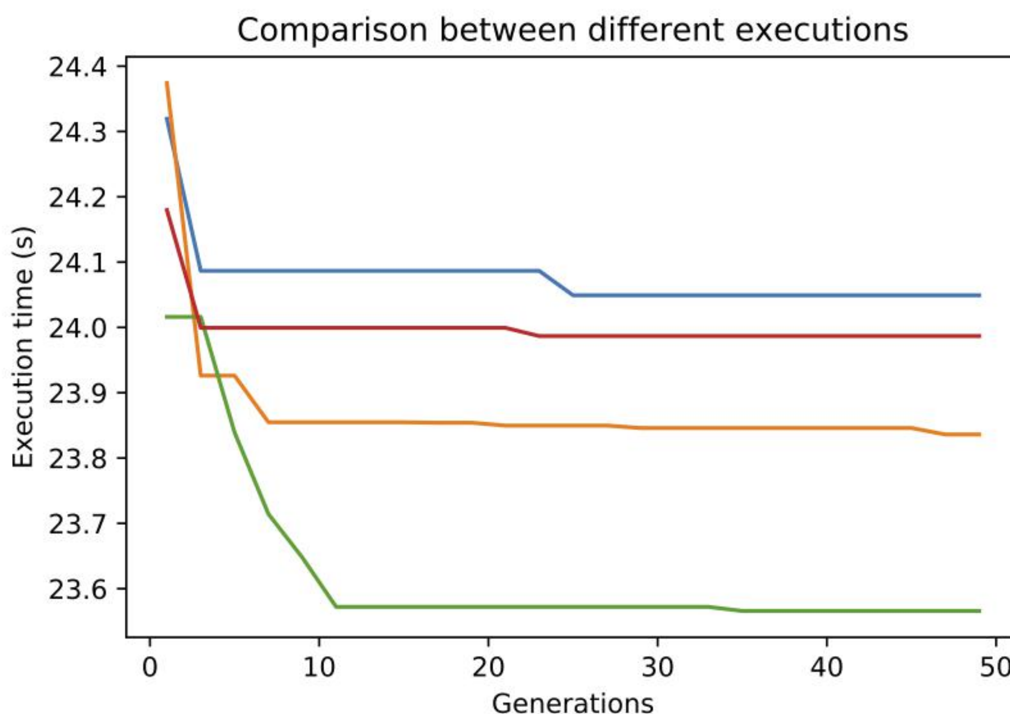
[8] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed on 10/20/2019

Table 3 – Comparison time

Time without optimization	31.574 seconds
Time optimizing with -O1 flag	28.402 seconds

Using the parameters defined in Table 1, we can see in Figure 10 that the genetic algorithm provided better performance early on comparing with the compiling code without optimization and with optimization with -O1 flags (between 14.23% and 15.21%). But these performances did not improve much in the final population (between 15.34% and 16.65%).

Figure 10 – Running with default parameters of genetic algorithms



In the graphic, each color corresponds to one different experiment.

In Figure 11, each chart represents the result of an experiment, where we executed PyCoptimizer with the same input source code and four different initial populations.

In all these experiments, the results are similar. As we can see in Table 4, the execution time in the initial and final populations in all experiments is a bit different but not too much. Comparing with the compilation without optimization or with a generic optimization, it's clear that the genetic algorithm gives better solutions. Even the worst final execution times in the experiments are better than the presented execution times in Table 3. The most likely reason for this is that optimization flags like this in Table 3 do not consider what code is being optimized neither what kind of machine is being used, unlike when it's used optimization flags that fit the code and the machine. These flags

Figure 11 – Experiments changing the genetic algorithm parameters

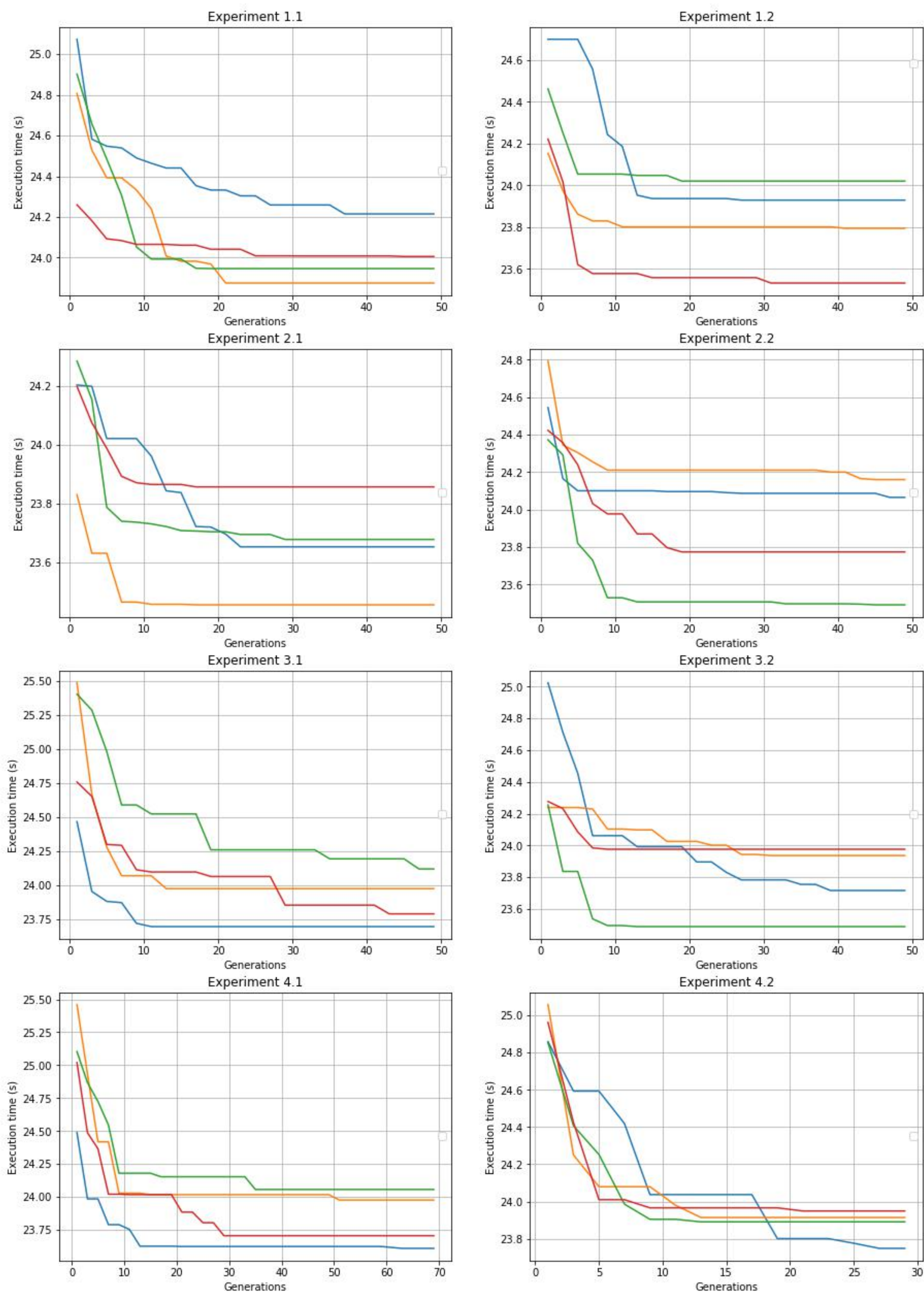


Table 4 – Results of each experiment

Experiments	Start time		Finish time	
	mean (s)	standard deviation (s)	mean (s)	standard deviation (s)
#1.1	24.7605	0.3043	24.0108	0.1267
#1.2	24.3843	0.2149	23.8196	0.1839
#2.1	24.1288	0.1760	23.6599	0.1423
#2.2	24.5336	0.1624	23.8734	0.2619
#3.1	25.0273	0.4292	23.8953	0.1636
#3.2	24.4481	0.3322	23.7798	0.1945
#4.1	25.0176	0.3475	23.8368	0.1848
#4.2	24.9322	0.0841	23.8761	0.0763

-O1, -O2, -O3 are generic flags to optimize and not necessarily make the best optimization in all the codes.

It can be seen from the results in Figure 11 that the convergence in all experiments occurred around generation 30. So, we decided to repeat the experiments with 3 modifications: (i) use the same initial population in all experiments; (ii) decreasing the number of generations (from 50 to 30) and (iii) increasing the number of execution of the PyCoptimizer (from 4 to 10).

In the next analysis, we considered the execution time in the first (start time) and in the last (final time) generation of each execution. Even starting with the same initial population, in the next graphics, it is possible to observe that the start time is not the same. This occurs because Pygmo gets the initial population and already applies the genetic algorithm, as we can see in Code 5.1. The mean of the start time is 27.2477 seconds.

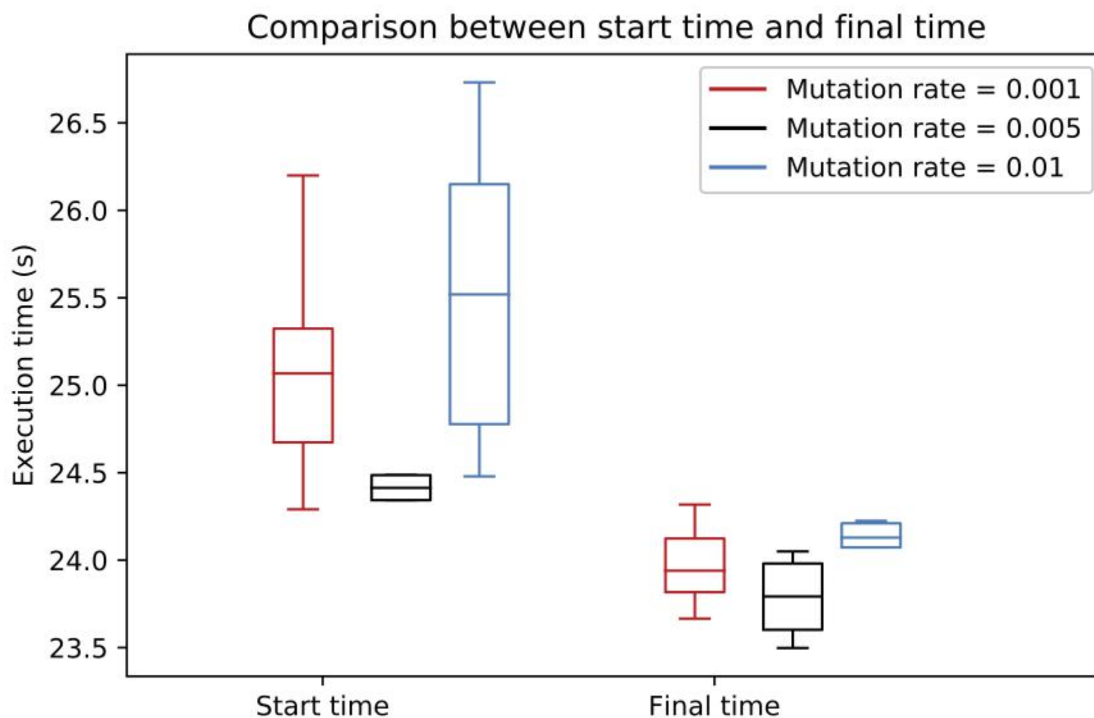
Code 4.1 – Genetic algorithm of Pygmo taken from Pygmo - Simple Genetic Algorithm in 06/18/2019

```
Start from a population (pop) of dimension N
while i < gen
    Selection: create a new population (pop2) with N individuals
selected from pop (with repetition allowed)
    Crossover: create a new population (pop3) with N individuals
obtained applying crossover to pop2
    Mutation:  create a new population (pop4) with N individuals
obtained applying mutation to pop3
    Evaluate all new chromosomes in pop4
    Reinsertion: set pop to contain the best N individuals taken
from pop and pop4
```

In Figure 12, we can see that varying the mutation rate does not interfere in the results since almost all execution times remain in the interval correspondent to 0.001 mutation rate. In Figure 13, it is possible to see that all crossover rates impact for the results, changing the means and the variance. The crossover with rate equals to 0.6 starts and finishes with the higher mean but obtain the best gain - 8.086%. With the smaller crossover rate, we save 3.65% of the time, while with crossover rate equals to 0.7, we save

2.55%. Still in the crossover rate equals to 0.6, the final time shows the worst mean, with more variance in the times. This agrees with what we already discussed previously: lower crossover rates are only recommended for a population with big size (SCHAFFER et al., 1989).

Figure 12 – Changing mutation rate using the same population

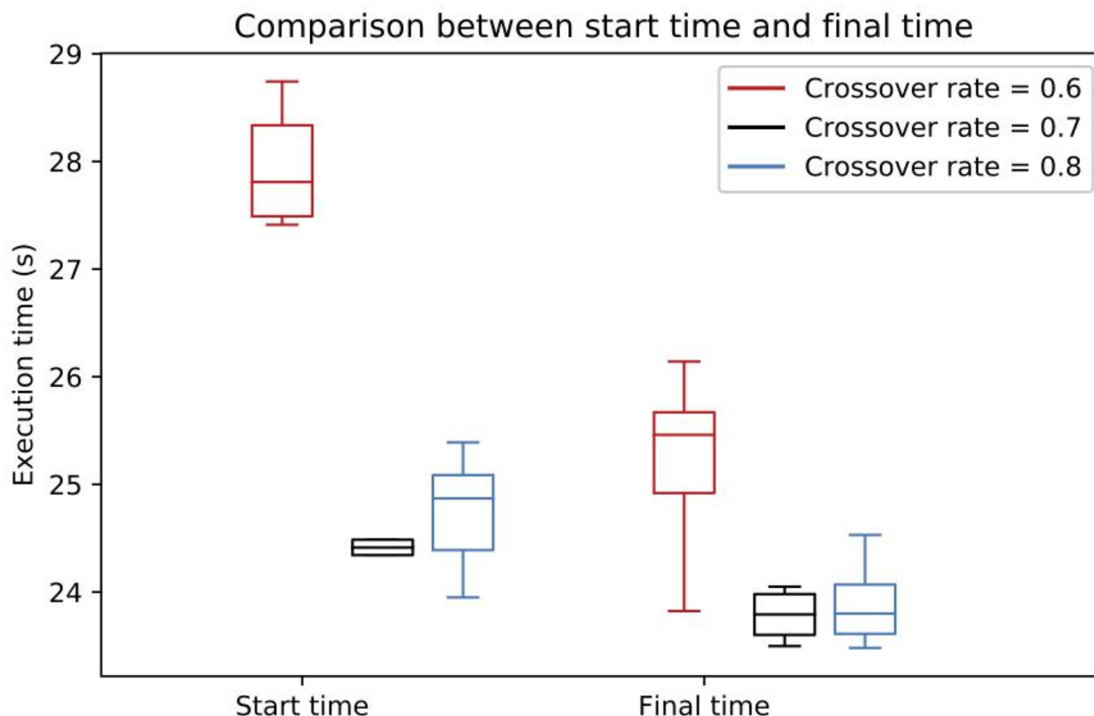


4.2 ANALYSIS OF THE RESULTING FLAGS

At the end of each genetic algorithm execution, we generated a list of optimization flags corresponding to the best individual in the last generation. We can observe that the number of resulting flags in each running is 94 flags on average, with a standard deviation of 5 flags, which were not affected by changing crossover and mutation rates.

Comparing all resulting flags in each experiment, we can note that no optimization flags always appears in all runnings. The -Os flag appears more frequently in the runnings - in 37 of 40 executions. This flag optimizes the execution time and tries to reduce the compiled code, without look to the wasted time in the compilation [8]. In Appendix B, we have all flags that were part of the population with the best performance at the end of each execution. In each one of the flags, we calculate the frequency which appears in each one of the runnings.

Figure 13 – Changing crossover rate using the same population



4.3 EXPERIMENT 2 - CROSS-KALMAN

CERN (the Conseil Européen pour la Recherche Nucléaire) has the Large Hadron Collider (LHC) which "is the world's largest and most powerful particle accelerator"[9]. Two high-energy particle beams travel inside the LHC close to the speed of the light and collide in four locations, corresponding to the four particle detectors [10].

One of the detectors is the LHCb - Large Hadron Collider beauty - which investigates the difference between matter and antimatter, studying particles called beauty quark or b quark [11]. The particles leave signals in the detector and it is necessary to make a reconstruction of the particles. For this reconstruction, the LHCb has trigger levels. In the first level, the Kalman filter algorithm, one of the algorithms to the reconstruction of the particles, takes 60% of the time of the reconstruction (CAMPORA-PEREZ, 2016).

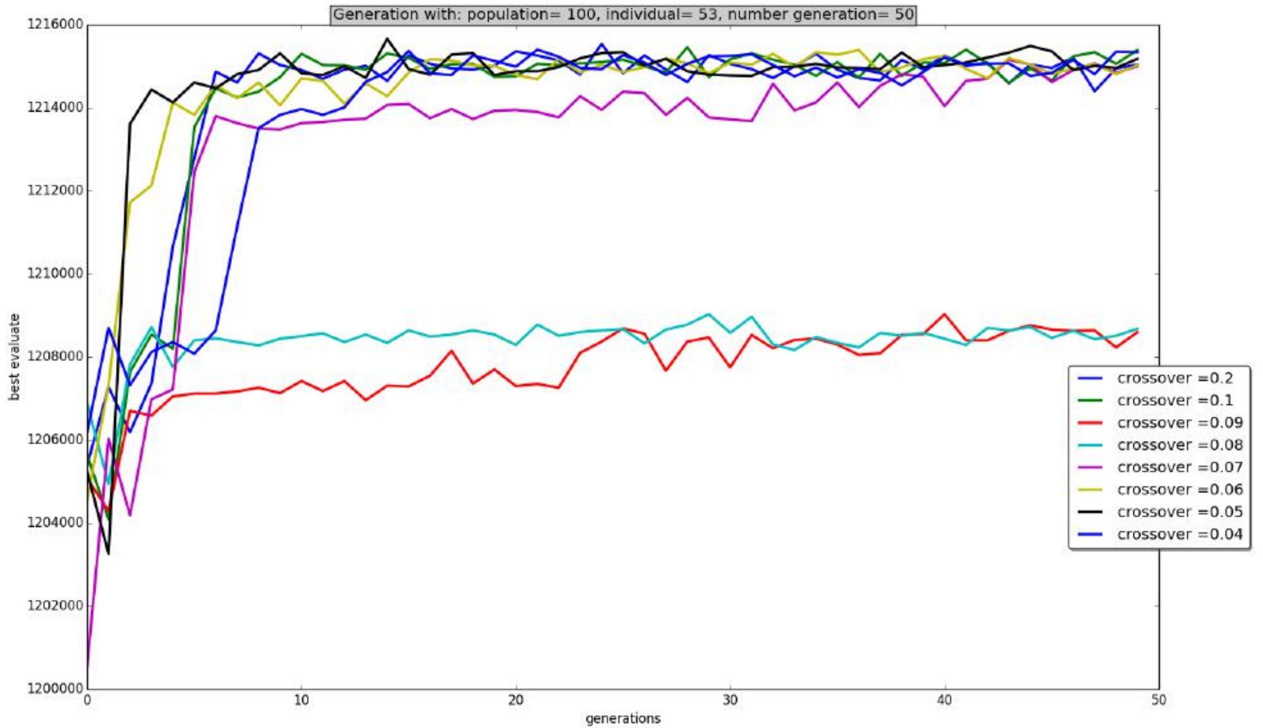
To maximize the throughput (which means getting more reconstructions in a feasible time), the idea is to find the best set of optimization flags using the genetic algorithm to reach this objective (FIGUEIREDO, 2017). The achieved results (FIGUEIREDO, 2017) are shown from Figure 14 until Figure 17. The genetic algorithm was executed with individuals composed of 53 genes, represented in the same way we represent the genes in this project, a population with 100 individuals and 50 generations. In the set of flags we are using in the experiments, 47 of the optimization flags used to get the results are

[9] https://home.cern/science/accelerators/large-hadron-collider. Accessed on 10/21/2019
[10] https://home.cern/about. Accessed on 10/21/2019.
[11] https://home.cern/science/experiments/lhcb. Accessed on 10/21/2019.

included.

In Figure 14 and Figure 15, the experiment was testing different values for the crossover rate while the mutation rate was constant and equals to 0.1. In Figure 16 and Figure 17, the experiment varied the mutation rate with the crossover rate equals 0.3. As we can see, the experiments get the best value in 10 generations, maintaining the same value for the next generations.

Figure 14 – Decreasing crossover rate with constant mutation rate



We did it using a code provided in the GitLab repository [12]. We will use the flags in Appendix A but eliminating the following flags in Table 5. These flags were eliminated because it caused problems when compiling the code and produced corrupted executable files.

Table 5 – Problemming flags

-flto
-fsingle-precision-constant
-fwhole-program

Table 6 shows the values when compiling without optimization and with the -O2 flags.

[12] https://gitlab.cern.ch/dcampora/cross_kalman/tree/master. Accessed on 10/21/2019.

Figure 15 – Increasing crossover rate with a constant mutation rate

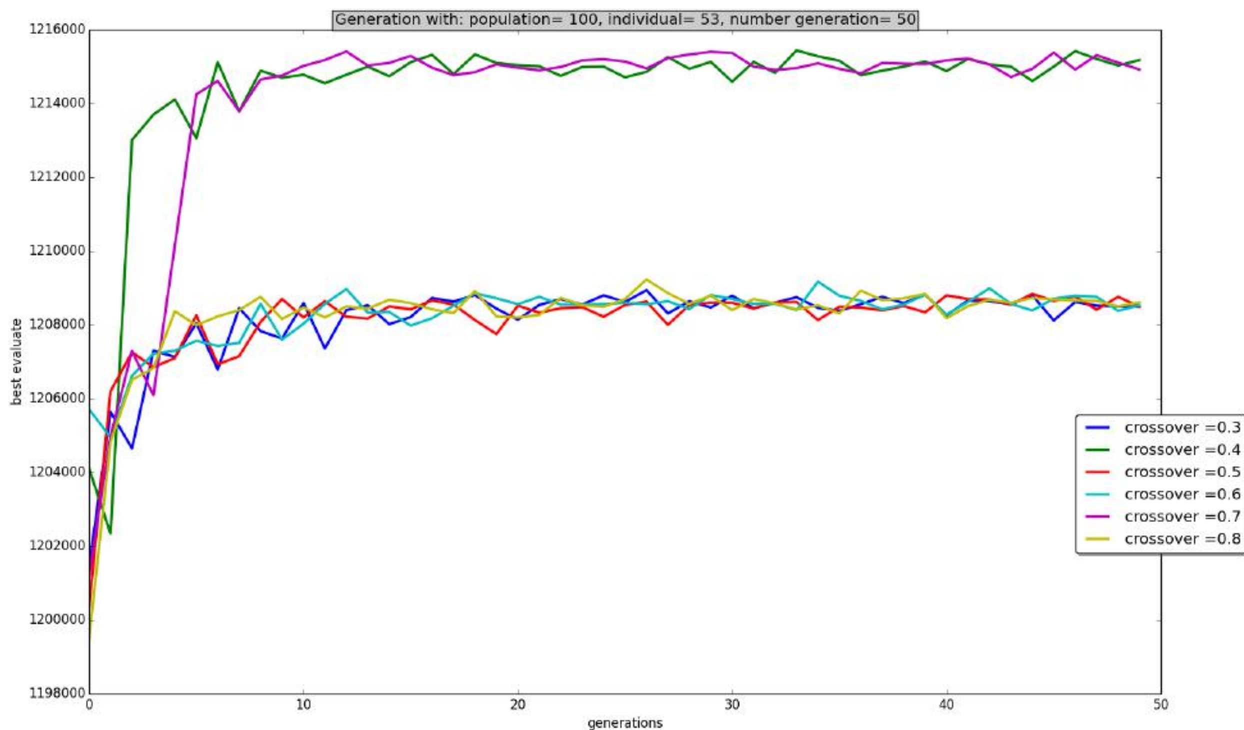


Table 6 – Results to case study

Throughput without optimization	54076.5
Optimization with -O2 flag	1.78958e+06

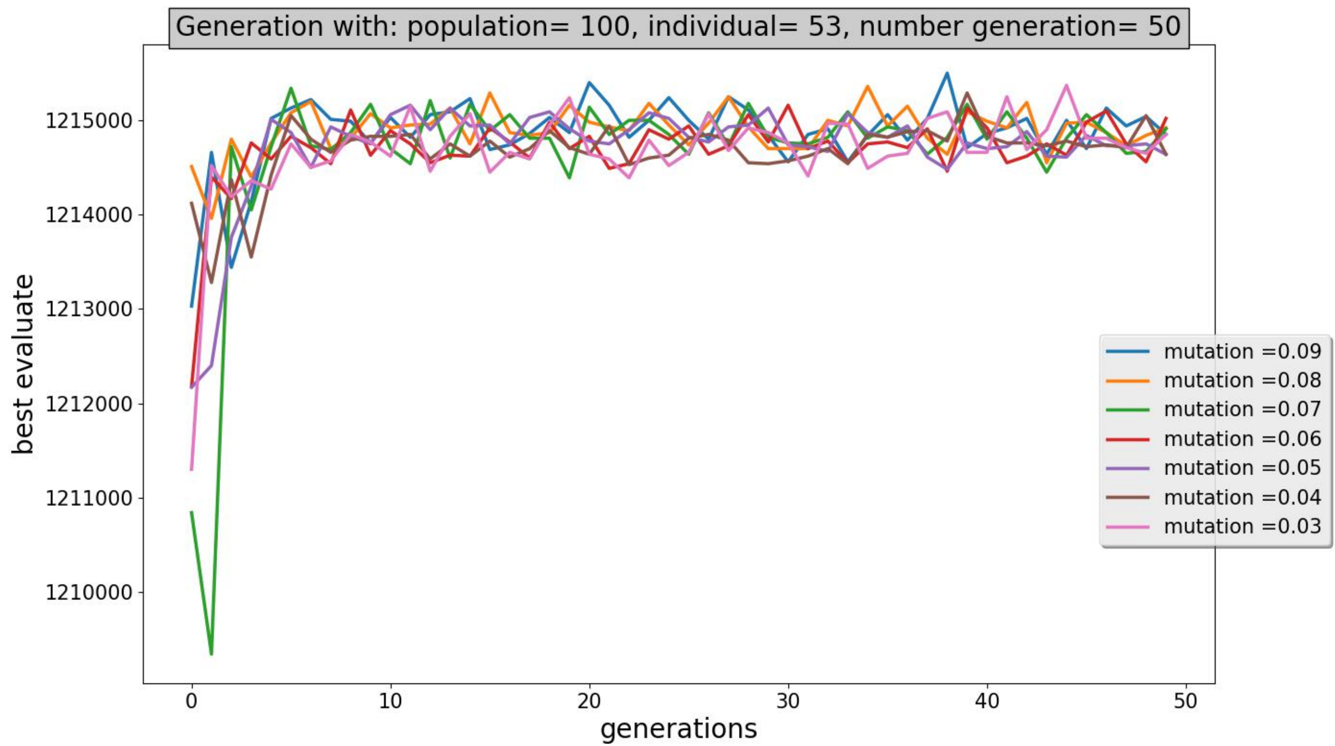
We ran the PyCoptimizer for the code, using the presented parameters in Table 1. As we saw previously, in Figure 10, the resultant values converge when they reached 30 generations. We did the same for this code, even being a different code and we ran for 10 executions.

The found throughput mean value in the initial population is 2,067,806.0 (standard deviation equals 71,703.00) and in the final population was 2,243,975.0 (standard deviation of 24,028.98), representing an improvement of 8.52%. As we can see, both obtained throughput mean values are larger than the one obtained when we use only the -O2 flag. Note that the variance decreases, indicating the throughput tends to converge in 30 generations.

Analyzing all the executions, we can observe in the last generation of each running, in mean, 90 flags were used, with a standard deviation of 6 flags. No flag appears in all executions and only 47 flags appear more frequently (50% of executions). The complete results can be seen in Appendix 3.

In comparison with the results in Figure 18, the results reached by PyCoptimizer were

Figure 16 – Decreasing mutation rate with a constant crossover rate



better. This is because there was a much larger solution space - the PyCoptimizer used 187 flags of optimization - while the report used 53 flags. Nine of 47 flags more frequent in our executions are also present in the report (FIGUEIREDO, 2017).

Figure 17 – Increasing mutation rate with a constant crossover rate

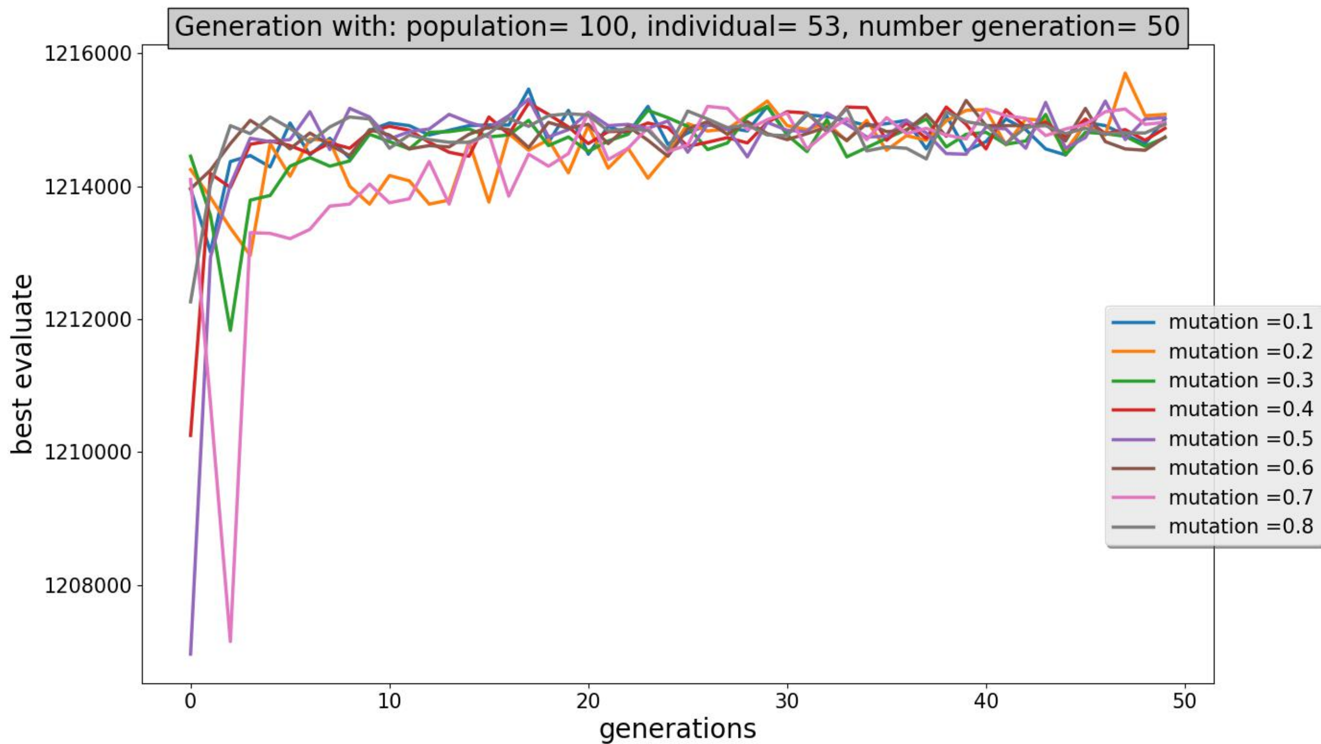
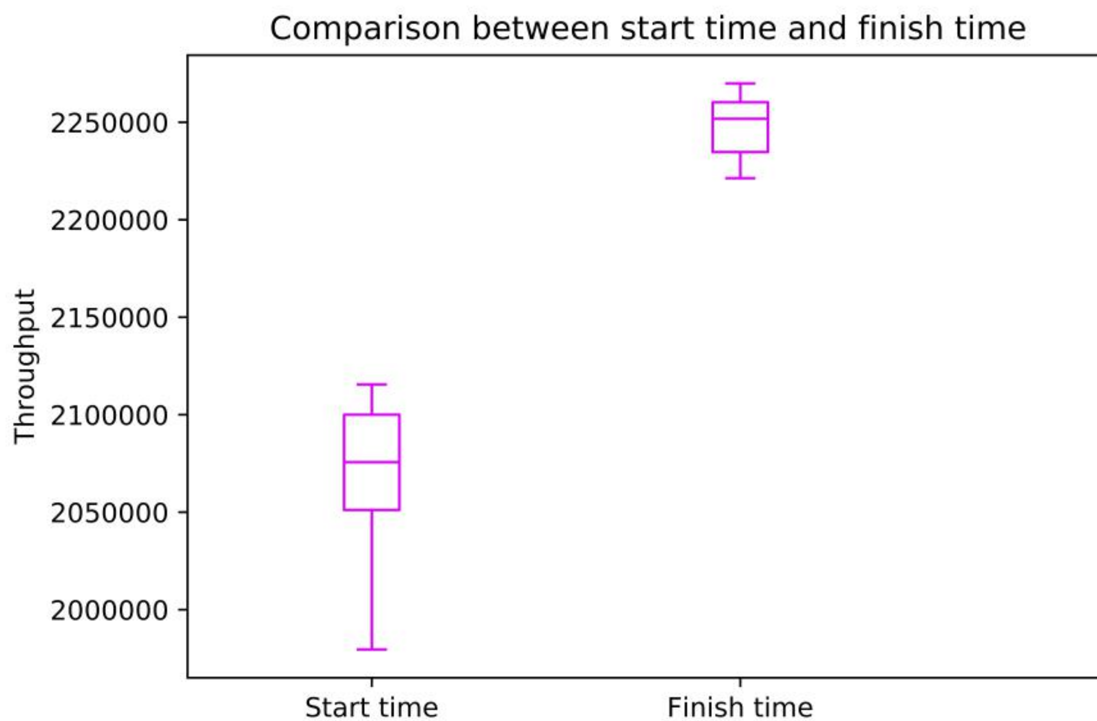


Figure 18 – Results to Cross Kalman code using PyCoptimizer



5 CONCLUSION

In this work, we build a framework based on genetic algorithm called PyCoptimizer that, given source code, finds a set of flags that generate an optimized compiled code.

After built the framework, we did 2 experiments. In the first, we wished to decrease the execution time of the executable file. In the second experiment, we search for increasing the throughput of the file. In both of them, the PyCoptimizer gave a set of optimization flags which performed a better optimization in the code, compared with the baseline results.

We used the Pygmo library to program the genetic algorithm. We face several library limitations, such as the impossibility of knowing the state of the population over the generations. In the experiments, we also had problems. Some optimization flags generated a corrupted file, not allowing its execution, as presented throughout the work. Therefore, we had to look for which flags could be used and which not. Each experiment took a long time to execute and finalize. In Table 7, we can see the duration of each running in each experiment. The spent time was long because, for all populations, the PyCoptimizer needed to compile and execute the codes, to evaluate the individuals.

Table 7 – Duration of each running per experiment

Experiments	Number of generations	
	30 generations	50 generations
Experiment 1	∼3 days/running	∼5 days/running
Experiment 2	∼one week/running	-

At the end of our project, we suggest as future works the development of new methods for genetic operators. We only developed one method for each operator, when many methods exist. For this reason, new methods can give more possibilities for the user. We also suggest the parallelization of the genetic algorithm, in the compilation moment of the code. This allows a faster genetic algorithm because this is the part that takes more time during the framework execution. We recommend being parallelized only in the compilation moment. When running the executable programs in a parallelized way, the programs use the machine resources at the same time. These resources are shared in the execution of the files. The generated result, at the final of the execution of each file, does not be trustworthy because the programs did not use all the capacity of the machine only for their executions.

Another possible future work is the implementation of other methods, besides the genetic algorithm, to search the solution. The availability of new methods allows the user has more than one option of choice for the operation of the framework. Besides

the implementation of new methods, the framework must be changed for the user can be capable to choose the method he wants to use.

REFERENCES

ALMAGOR, L. et al. Finding effective compilation sequences. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 39, n. 7, p. 231–239, jun. 2004. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/998300.997196>.

ASHLOCK, D. **Evolutionary Computation for Modeling and Optimization**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441919694, 9781441919694.

Back, T.; Hammel, U.; Schwefel, H. . Evolutionary computation: comments on the history and current state. **IEEE Transactions on Evolutionary Computation**, v. 1, n. 1, p. 3–17, April 1997. ISSN 1089-778X.

CAMPORA-PEREZ, D. H. LHCb Kalman Filter cross architectures studies. Oct 2016. Disponível em: <https://cds.cern.ch/record/2229971>.

COOPER, K. D.; SCHIELKE, P. J.; SUBRAMANIAN, D. Optimizing for reduced code space using genetic algorithms. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 34, n. 7, p. 1–9, maio 1999. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/315253.314414>.

COOPER, K. D.; SUBRAMANIAN, D.; TORCZON, L. Adaptive optimizing compilers for the 21st century. **J. Supercomput.**, Kluwer Academic Publishers, Norwell, MA, USA, v. 23, n. 1, p. 7–22, ago. 2002. ISSN 0920-8542. Disponível em: <https://doi.org/10.1023/A:1015729001611>.

FIGUEIREDO, L. F. d. **Evolutionary Optimization Of LHCb Software Optimization**. Geneva: CERN, 2017. (Technical Report).

FOGEL, D. B. An evolutionary approach to the traveling salesman problem. **Biological Cybernetics**, v. 60, n. 2, p. 139–144, Dec 1988. ISSN 1432-0770. Disponível em: <https://doi.org/10.1007/BF00202901>.

Fogel, D. B. An introduction to simulated evolutionary optimization. **IEEE Transactions on Neural Networks**, v. 5, n. 1, p. 3–14, Jan 1994. ISSN 1045-9227.

HANEDA, M.; KNIJNENBURG, P. M. W.; WIJSHOFF, H. A. G. Automatic selection of compiler options using non-parametric inferential statistics. In: **Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques**. Washington, DC, USA: IEEE Computer Society, 2005. (PACT '05), p. 123–132. ISBN 0-7695-2429-X. Disponível em: <http://dx.doi.org/10.1109/PACT.2005.9>.

HOSTE, K.; EECKHOUT, L. Cole: Compiler optimization level exploration. In: **Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization**. New York, NY, USA: ACM, 2008. (CGO '08), p. 165–174. ISBN 978-1-59593-978-4. Disponível em: <http://doi.acm.org/10.1145/1356058.1356080>.

LINDEN, R. **Algoritmos Genéticos (2a edição)**. BRASPORT, 2008. ISBN 9788574523736. Disponível em: <https://books.google.com.br/books?id=it0kv6UsEMEC>.

PAN, Z.; EIGENMANN, R. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: **Proceedings of the International Symposium on Code Generation and Optimization**. Washington, DC, USA: IEEE Computer Society, 2006. (CGO '06), p. 319–332. ISBN 0-7695-2499-0. Disponível em: <http://dx.doi.org/10.1109/CGO.2006.38>.

SCHAFFER, J. D. et al. A study of control parameters affecting online performance of genetic algorithms for function optimization. In: **Proceedings of the Third International Conference on Genetic Algorithms**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989. p. 51–60. ISBN 1-55860-006-3. Disponível em: <http://dl.acm.org/citation.cfm?id=93126.93145>.

TANG, W. K. et al. Genetic algorithms and their applications. **Signal Processing Magazine, IEEE**, v. 13, p. 22 – 37, 12 1996.

APÊNDICES

APPENDIX A – FLAGS USED IN THE EXPERIMENTS

-falign-functions	-falign-jumps	-falign-labels
falign-loops	-fauto-inc-dec	-fbranch-probabilities
-fbranch-target-load-optimize	-fbranch-target-load-optimize2	-fbtr-bb-exclusive
-fcaller-saves	-fcombine-stack-adjustments	-fconserve-stack
-fcompare-elim	-fcprop-registers	-fcrossjumping
-fcse-follow-jumps	-fcse-skip-blocks	-fcx-fortran-rules
-fcx-limited-range	-fdata-sections	-fdce
-fdelayed-branch	fdelete-null-pointer-checks	-fdevirtualize
-fdevirtualize-speculatively	-fdevirtualize-at-ltrans	-fdse
-fearly-inlining	-fipa-sra"	-fexpensive-optimizations
-ffat-lto-objects	-ffast-math	-ffinite-math-only
fforward-propagate	-ffunction-sections	-fgcse
-fgcse-after-reload	-fgcse-las	-fgcse-lm
-fgraphite-identity	-fgcse-sm	-fhoist-adjacent-loads
-fif-conversion	-fif-conversion2	-findirect-inlining
-finline-functions	-finline-functions-called-once	-finline-small-functions
-fipa-cp	-fipa-cp-clone	-fipa-pta
-fipa-profile	-fipa-pure-const	-fipa-reference
-fipa-icf	-fira-hoist-pressure	-fira-loop-pressure
-fno-ira-share-save-slots	-fno-ira-share-spill-slots	-fisolate-erroneous-paths-dereference
-fisolate-erroneous-paths-attribute	-fivopts	-fkeep-inline-functions
-fkeep-static-consts	-flive-range-shrinkage	-floop-block
-floop-interchange	-floop-strip-mine	-floop-unroll-and-jam
-floop-nest-optimize	-floop-parallelize-all	-flra-remat
-flto	-fmerge-all-constants	-fmerge-constants
-fmodulo-sched	-fmodulo-sched-allow-regmoves	-fmove-loop-invariants
-fno-branch-count-reg	-fno-defer-pop	-fno-function-cse
-fno-guess-branch-probability	-fno-inline	-fno-math-errno
-fno-peephole	-fno-peephole2	-fno-sched-interblock
-fno-sched-spec	-fno-signed-zeros	-fno-toplevel-reorder
fno-trapping-math	-fno-zero-initialized-in-bss	-fomit-frame-pointer
-foptimize-sibling-calls	-fpartial-inlining	fpeel-loops
-fpredictive-commoning	-fprofile-correction	-fprofile-use
-fprofile-values	-fprofile-reorder-functions	-frename-registers
-freorder-blocks	-freorder-blocks-and-partition	-freorder-functions
-frerun-cse-after-loop	-freschedule-modulo-scheduled-loops	-frounding-math
-fsched2-use-superblocks	-fsched-pressure	-fsched-spec-load
-fsched-spec-load-dangerous	-fsched-stalled-insns-dep	-fsched-stalled-insns
-fsched-group-heuristic	-fsched-critical-path-heuristic	-fsched-spec-insn-heuristic
-fsched-rank-heuristic	-fsched-last-insn-heuristic	-fsched-dep-count-heuristic
-fschedule-fusion	-fschedule-insns	-fselective-scheduling
-fselective-scheduling2	-fsel-sched-pipelining	-fsel-sched-pipelining-outer-loops
-fsemantic-interposition	-fshrink-wrap	-fsignaling-nans
-fsingle-precision-constant	-fsplit-ivs-in-unroller	-fsplit-wide-types
-fssa-phiopt	-fstdarg-opt	-fstrict-aliasing

-fthread-jumps	-ftracer	-ftree-bit-ccp
-ftree-builtin-call-dce	-ftree-ccp	-ftree-ch
ftree-coalesce-vars	-ftree-copy-prop	-ftree-dce
-ftree-dominator-opts	-ftree-dse	-ftree-forwprop
-ftree-fre	-ftree-loop-if-convert	-ftree-loop-im
ftree-phiprop	-ftree-loop-distribution	-ftree-loop-distribute-patterns
-ftree-loop-ivcanon	-ftree-loop-linear	-ftree-loop-optimize
-ftree-loop-vectorize	-ftree-pre	-ftree-partial-pre
-ftree-pta	-ftree-reassoc	-ftree-sink
-ftree-slsr	-ftree-sra	-ftree-switch-conversion
-ftree-tail-merge	-ftree-ter	-ftree-vectorize
-ftree-vrp	-funit-at-a-time	-funroll-all-loops
-funroll-loops	-funsafe-math-optimizations	-funswitch-loops
-fipa-ra	-fvariable-expansion-in-unroller	-fvect-cost-model
-fvpt	-fweb	-fwhole-program
-fuse-linker-plugin	-O	-O0
-O1	-O2	-O3
-Os	-Ofast	-Og

APPENDIX B – FREQUENCY OF FLAGS

Figure 19 – Frequency of 20 flags with mutation rate equals to 0.01

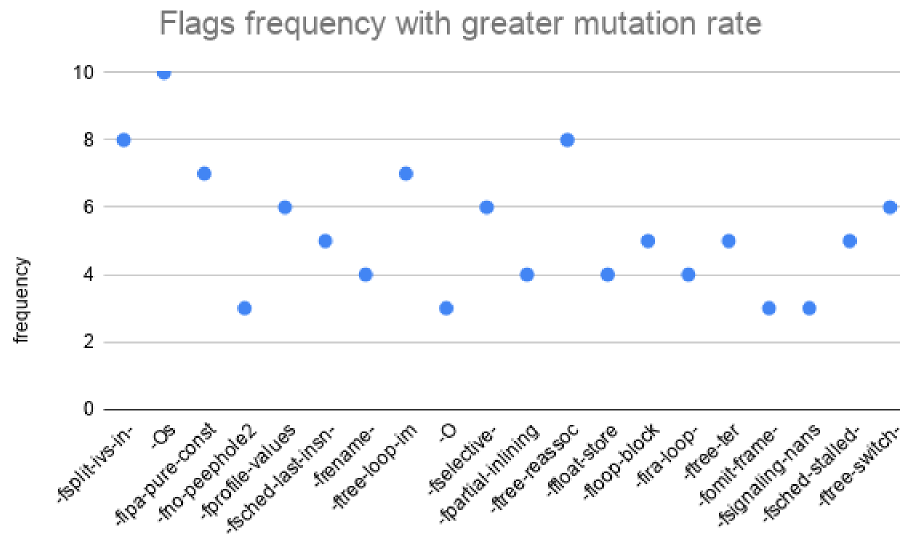


Figure 20 – Frequency of 20 flags with mutation rate equals to 0.001

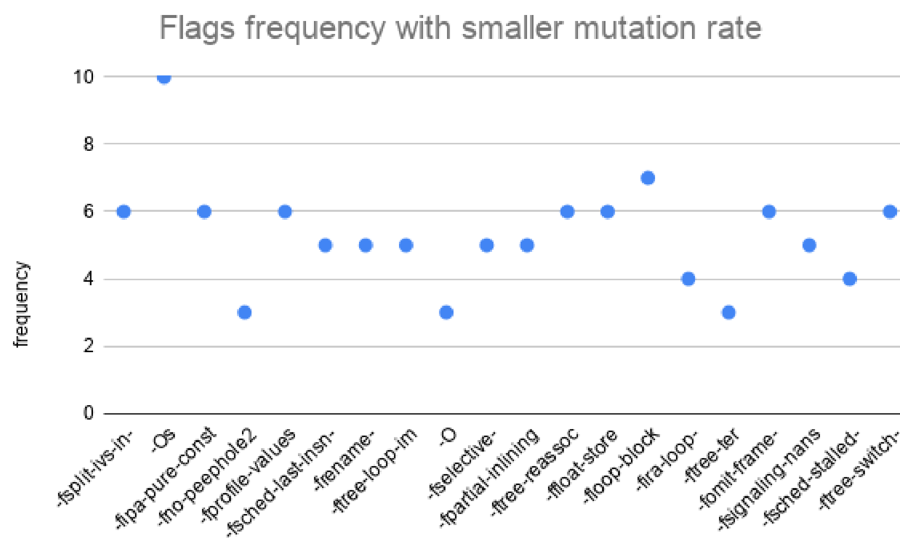


Figure 21 – Frequency of 20 flags with crossover rate equals to 0.8

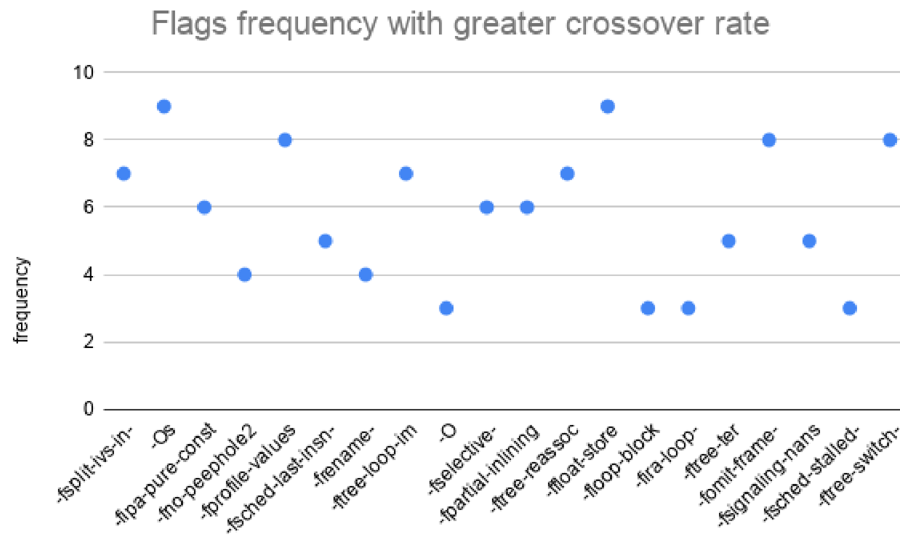
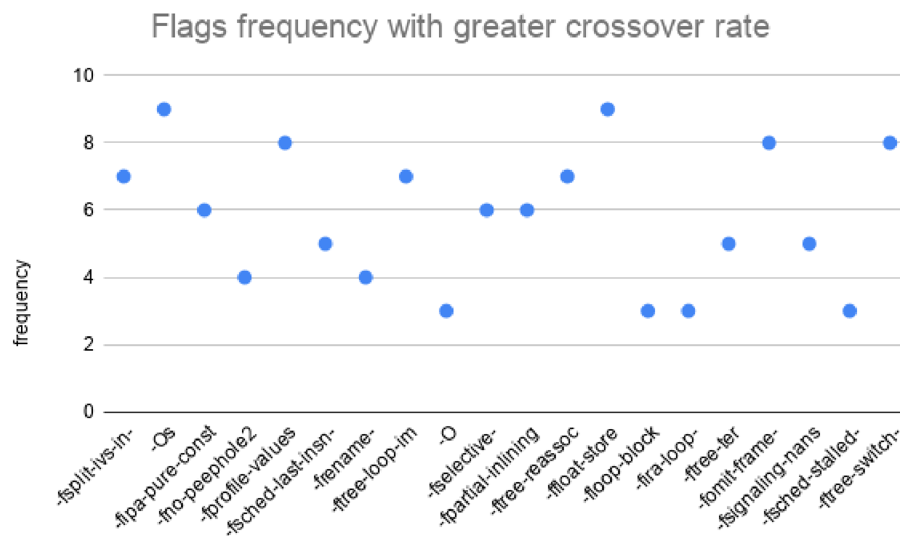


Figure 22 – Frequency of 20 flags with crossover rate equals to 0.6



APPENDIX C – FREQUENCY OF FLAGS - CASE STUDY

Figure 23 – Frequency of 20 flags about the case study using 30 generations