PWiseGen: A Genetic Algorithms-based Framework for Pairwise Test Set Generation

Pedro Flores
University of Texas at El Paso, flores_barba@hotmail.com

Follow this and additional works at: https://digitalcommons.utep.edu/open_etd

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.utep.edu/open_etd/2484

This is brought to you for free and open access by DigitalCommons@UTEP. It has been accepted for inclusion in Open Access Theses & Dissertations by an authorized administrator of DigitalCommons@UTEP. For more information, please contact lweber@utep.edu.
PWisegen: A Genetic Algorithms-based Framework for Pairwise Test Set Generation

PEDRO FLORES

Department of Computer Science

APPROVED:

______________________________
Yoonsik Cheon, Chair, Ph.D.

______________________________
Olac Fuentes, Ph.D.

______________________________
Tzu-Liang (Bill) Tseng, Ph.D.

______________________________
Patricia D. Witherspoon, Ph.D.
Dean of the Graduate School
to my

WIFE

with love

by

PEDRO FLORES

THESIS
Presented to the Faculty of the Graduate School of
The University of Texas at El Paso
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Computer Science
THE UNIVERSITY OF TEXAS AT EL PASO
December 2010
I would like to thank all the people that contributed in making this research possible. Firstly, my thesis advisor, Dr. Yoonsik Cheon, who has given me guidance throughout the development of this thesis and also for being a motivator in pursuing the subject of this study. I also want to thank the committee members reviewing this thesis, Dr. Olac Fuentes and Dr. Bill Tseng, for their contributions and valuable time dedicated to this work.

I want to thank my brother Carlos as well, who has given me a hand with grammar and spelling, and in many cases proposing ideas to make a more fluid and interesting document to read. Likewise, everyone who in some way gave me courage to continue my graduate studies with dedication: my family and friends.

Lastly, I would like to thank my wife Thania, who has given me plenty of strength for my professional development, and for being a true pillar in my life. I am grateful for all the sacrifices and understanding for the time that I have not spent with her, just to be able to complete my graduate studies.
Abstract

Pairwise testing is a combinatorial testing approach that can be used to reduce the number of test cases dramatically for a software system. It specializes in the type of testing where the outcome of a system depends on the combination of several input parameters. This technique is widely used when there is not enough time to try out all possible combinations of the input parameters involved, because that would be very time consuming. The purpose of pairwise testing technique is to avoid testing all possible value combinations that a system can receive, and to preferably take a representative sample of test cases that captures all pairs of input values. This selection of test cases should be made carefully with some thought; for example, by combining input parameter values intelligently we may be able to parallelize executions and achieve a reduction of the test case set, which translates into a reduction of testing time.

Studies have shown that the generation of the least number of test cases for pairwise testing is an NP-complete problem. This is the reason why one of the problems that I wanted to address in this thesis is to find a way to generate the least amount of test cases possible for pairwise testing. Currently, there are several algorithms that can generate test cases for pairwise testing; however, a few have optimum results considering the number of test cases generated. In this research I propose PWiseGen – an approach based on genetic algorithms – that shows competitive results compared with other existing algorithms.

Although there is existing work on genetic algorithms to solve the problem of generating the test cases for pairwise testing, there are no tools or open-source code available. That’s why another of the problems that I wanted to address in this research is the lack of a tool based on genetic algorithms capable of generating test cases for pairwise testing, where one could adjust its configuration, extend its capabilities, or override some of its default behavior, given that genetic algorithms can have many different ways of performing its internal operations. PWiseGen is designed to serve as a framework, in which one can make
modifications and adapt to specific problems, or simply to serve as a tool to be used for further research. In order to accomplish this, I use object-oriented concepts to facilitate the reusability as well as the easiness of extending the tool.

Some of the prominent contributions made in this research include (a) an algorithm capable of generating test cases for pairwise testing, (b) a configurable open-source framework that can be used for experimenting on test case generation for pairwise testing, and (c) sample configurations and guidelines for using, adapting, and extending the framework.
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td>Abstract</td>
<td>v</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
</tbody>
</table>

## Chapter 1 Introduction

1.1 Background
   1.1.1 Pairwise Testing
   1.1.2 Genetic Algorithms

1.2 The Problems
   1.2.1 The Algorithm Problem
   1.2.2 The Tooling Problem

1.3 Contributions

1.4 Related Work
   1.4.1 In-Parameter-Order
   1.4.2 Orthogonal Arrays
   1.4.3 AETG
   1.4.4 Genetic Algorithms

1.5 Outline

## Chapter 2 Approach for the Algorithm Problem

2.1 Use of a Genetic Algorithm
   2.1.1 Chromosome Encoding
   2.1.2 Fitness Function
List of Tables

1.1 Example of a test case set for pairwise ........................................ 3
1.2 Example of pairs for pairwise testing ........................................... 10
1.3 Example of an exhaustive test case set for pairwise testing ............ 11
1.4 An example set of test cases for pairwise testing ............................ 12
1.5 Example of an orthogonal array .................................................. 16

4.1 Comparison of effectiveness for benchmark problems solved with existing
  approaches .................................................................................... 58
4.2 Comparison of smart mutation and random mutation ....................... 63
4.3 Comparison of different configurations - problem domain 2 ............. 66
4.4 Comparison of different configurations - problem domain 3 ............. 67
4.5 Comparison of different configurations - problem domain 4 ............. 67
4.6 Comparison of different configurations - problem domain 5 ............. 68
4.7 Comparison of different configurations - problem domain 6 ............. 69
# List of Figures

1.1 Biological model ......................................................... 4
1.2 Crossover genetic operator ............................................ 6
1.3 Mutation genetic operator ............................................. 6
1.4 Basic genetic algorithm workflow ..................................... 7

2.1 Chromosome encoding using an integer array ....................... 22
2.2 Different pairs fitness function ....................................... 24
2.3 All pairs that need to be captured by a chromosome ............... 25
2.4 Cumulative fitness array for roulette wheel selection ............ 26
2.5 Random mutation ....................................................... 29
2.6 Similarity mutation ..................................................... 31
2.7 Value occurrences mutation ........................................... 32
2.8 Pair occurrences mutation ............................................. 33

3.1 Genetic algorithm main class diagram ................................ 37
3.2 Crossover class diagram ............................................... 39
3.3 Fitness function class diagram ........................................ 40
3.4 Mutation class diagram ................................................ 41
3.5 Replacement class diagram ............................................. 42
3.6 Selection class diagram ................................................ 43
3.7 Population initialization class diagram ............................... 44
3.8 Utilities class diagram .................................................. 44
3.9 Information class diagram ............................................. 47
3.10 Configuration file `ga-config.xml` .................................... 50

4.1 Comparison with random algorithm - Experiment 1 ............... 55
4.2 Comparison with random algorithm - Experiment 2  .................................................. 56
4.3 Results of PWiseGen for benchmark problem 1  ......................................................... 58
4.4 Results of PWiseGen for benchmark problem 2  ......................................................... 59
4.5 Results of PWiseGen for benchmark problem 3  ......................................................... 60
4.6 Results of PWiseGen for benchmark problem 4  ......................................................... 60
4.7 Results of PWiseGen for benchmark problem 5  ......................................................... 61
4.8 Results of PWiseGen for benchmark problem 6  ......................................................... 61
4.9 Comparison of effectiveness for benchmark problems solved with PWiseGen and other existing approaches ................................................................. 62
4.10 Initial configuration as starting point ................................................................. 64
4.11 Modifying the MaxGenerations value ................................................................. 71
4.12 MyNewMutation UML class  ................................................................. 72
Chapter 1

Introduction

Testing is a very important part of a software development process. It is a way to ensure the quality of a software product being developed. In real life, software development corporations typically manage very tight schedules, in which their products have to be released to the market as soon as possible. This is why testers are faced with the challenge of testing a software system as much as possible to find most, if not all, of the errors contained in the system, with aggressive time constraints. This is why various software testing techniques have emerged to help diminish the amount of work without compromising the main objective, to find all the errors in a software system.

There are many parameters that determine how a software system behaves, and it is sometimes impossible, or there is no time for checking every possible combination. For example, if a system can receive an input of five parameters and each in turn can take ten different values, then we would be talking about $10^5$ possible combinations that would need to be tested. And if each test case takes one minute to execute, then it would take more than two months to finish testing all the combinations.

This is why I decided on making a contribution, finding an alternative for generating and reducing the number of test cases, by developing an algorithm that can be used to perform this task, and also turn this algorithm into a tool that can be used on future investigations, without the need of re-writing code that is already done. Therefore, I address two problems in this thesis: the algorithm problem and the tooling problem.

To solve the algorithm problem, I apply *genetic algorithms* to find a set of test cases that contains all the parameter combinations for pairwise testing (see Section 1.1 for detailed descriptions). To address the tooling problem, I turn the genetic algorithm-based test case
generator into a reusable framework – PWiseGen – by applying an object-oriented approach in a way that it could be easily configured, extended and reused.

In the remainder of this chapter, background information is presented with a brief explanation of what pairwise testing is; similarly, an overview of genetic algorithms is presented. Afterwards, problems that are addressed in this thesis can be seen. Following this, the contributions of my work are summarized; then related work is presented. Lastly, the structure and organization of this thesis can be found.

1.1 Background

1.1.1 Pairwise Testing

Pairwise testing is a combinatorial testing technique, where every pair of input parameters of a software system is tested [5]. The reasoning behind pairwise testing is that the majority of software errors are typically generated by a direct system input or due to some setting being used by that system. There also exists another category in which errors can appear frequently, caused by the combination of two input parameters.

Basically what pairwise testing suggests, is that we shouldn’t test all parameter combinations that a system could receive, but to focus on all the parameter pairings. In other words, as long as the group of test cases selected contains all the pairs, then we can say we found a set that is good for testing. Pairwise testing requires each parameter value to be captured at least by one test case. If we have a software system receiving three input parameters $x$, $y$ and $z$, and each parameter can have three different values, then there would be 27 different pairs: \(\{x_1, y_1\}, \{x_1, y_2\}, \{x_1, y_3\}, \{x_1, z_1\}, \{x_1, z_2\}, \{x_1, z_3\}, \{x_2, y_1\} \ldots \{y_3, z_3\}\). Therefore, a test case could be \(\langle x_1, y_3, z_2 \rangle\) which captures three of the 27 required pairs: \(\{x_1, y_3\}, \{x_1, z_2\}\) and \(\{y_3, z_2\}\). If we would like to perform exhaustive testing, trying out all possible combinations, then we would need to make 27 test cases. On the other hand, taking advantage of pairwise testing, we can considerably reduce the number of test cases;
Table 1.1: Example of a test case set capturing all 27 required pairs

<table>
<thead>
<tr>
<th>Test Case</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>x₀</td>
<td>y₁</td>
<td>z₀</td>
</tr>
<tr>
<td>TC2</td>
<td>x₂</td>
<td>y₂</td>
<td>z₂</td>
</tr>
<tr>
<td>TC3</td>
<td>x₀</td>
<td>y₀</td>
<td>z₂</td>
</tr>
<tr>
<td>TC4</td>
<td>x₀</td>
<td>y₂</td>
<td>z₁</td>
</tr>
<tr>
<td>TC5</td>
<td>x₁</td>
<td>y₂</td>
<td>z₀</td>
</tr>
<tr>
<td>TC6</td>
<td>x₁</td>
<td>y₀</td>
<td>z₁</td>
</tr>
<tr>
<td>TC7</td>
<td>x₁</td>
<td>y₁</td>
<td>z₂</td>
</tr>
<tr>
<td>TC8</td>
<td>x₂</td>
<td>y₀</td>
<td>z₀</td>
</tr>
<tr>
<td>TC9</td>
<td>x₂</td>
<td>y₁</td>
<td>z₁</td>
</tr>
</tbody>
</table>

for example, a set of nine test cases could capture all 27 required pairs (see Table 1.1).

Generating the least number of test cases for pairwise testing has been proven to be an NP-complete problem [11]. This means that an efficient way to find the optimum solution is not known and that the time required to generate the test cases grows rapidly when the amount of parameters and possible values increase.

1.1.2 Genetic Algorithms

Genetic algorithm (GA) is a technique that simulates the natural process of evolution [7]. GA was discovered as a useful tool when dealing with searches and optimization related problems. GA constitutes an excellent option when it comes to finding solutions for problems with a huge search space and complexity.

GAs use biological models to emulate the process of evolution, where a population is made of a set of individuals. These individuals contain chromosomes, which are also constituted of genes, as shown in Figure 1.1. A gene represents a feature of an individual,
and the alleles are the possible values which can be taken by that gene. For example, the eye color feature of a person is the gene, and the alleles for that gene could be black, brown, blue, green, etc. The combination of genes within a chromosome is what defines an individual’s set of features.

For an algorithm to be considered “genetic”, it should at least have the following key elements:

- A way to represent a possible solution (chromosome encoding)
- Fitness function
- Genetic operators

**Chromosome Encoding**

In a typical GA, the individuals are generally constituted of a coded chromosome in a string of 0s and 1s; each string position represents a characteristic (gene) of the individual. This
coding can vary depending on the specific problem to be solved, given that the idea is for
each individual to represent a possible solution to the problem.

**Fitness Function**

The GA needs to have a means to measure each individual’s potential; here is where a fitness
function needs to be provided. This fitness function determines how good is an individual
amongst all the others, and its implementation logic is done depending on the problem
domain being solved. The fitness value – calculated by a fitness function – associated with
each individual is the element used to determine which individuals have more opportunities
to prevail in a population.

**Genetic Operators**

In a similar manner, the rules for evolution in the GA need to be defined. These rules are
the genetic operators which are going to be applied to the individuals, in order to facilitate
evolution. The most common genetic operators are:

**Selection:** This operator is used when selecting individuals for reproduction; later these
individuals are then referred to as parents. These parents are selected based on their
fitness values; the ones with better chances of reproducing are those with the best
fitness values.

**Crossover:** This operator works by copying parts of the genetic material of two parents
and generating two descendants with their combinations. The simplest is by selecting
a single crossover point to determine the segments being copied from both parents.
There are other ways to perform the crossover, for example, selecting more than one
point for the crossover, called multiple crossover. This operator can be seen in Figure
1.2.

**Mutation:** The mutation genetic operator modifies gene values randomly (see Figure 1.3);
usually this is applied to the descendants, based on a certain occurrence probability.
Replacement: This operator defines the rules of replacing existing individuals in the population with the newly created individuals, the descendants.

There exists other operators that are not very common, but they are still used in certain situations, such as the immigration operator, which inserts random-generated individuals every certain amount of time in order to increase the diversity in the population.

In general terms, a typical evolution process followed by a GA is basically to generate an initial random population and to find the best individuals inside it, in order to combine them and apply other genetic operators to generate new individuals. When making this process repeatedly, the population tends to evolve by showing better individuals which are closer to find a solution. The Figure 1.4 shows a basic GA workflow.
1.2 The Problems

1.2.1 The Algorithm Problem

Pairwise testing is a technique used to test software systems. For this technique, in order to effectively test a system made to receive a certain number of parameters, where each parameter is able to receive a certain amount of values, we need to ensure that all the value pairs are covered by the set of test cases. The easiest would be to generate a test case for each pair that we need to test, but this is overly impractical and not feasible for situations when each test case takes long for its execution. Therefore, the ideal would be to select test cases in a more intelligent way, so that we could parallelize the testing of the required pairs. This would help us to considerably reduce the number of test cases needed for pairwise testing. The challenge is to find a set with the least number of test cases containing all pairs.
Example

Suppose that we have been asked to test a Web based application, and the parameters that needed to be tested are:

- How does the application behave when using a certain browser
- How does the application look like when using a certain screen resolution
- How does the application behave depending on whether Javascript is enabled in the browser or not
- How does the application behave depending on whether cookies are enabled in the browser or not

For this, you are given a list of browsers and different screen resolutions. For the Javascript and cookies parameters, we already know that they can only be enabled or disabled. Suppose that they want to test the following browsers: Internet Explorer (IE), Firefox (FF), Opera, and Safari. Let’s also suppose that the following screen resolutions are to be tested: 800x600, 1024x768, and 1280x800. A sample test case would be ⟨Firefox, 800x600, Javascript enabled, Cookies disabled⟩.

Translating these requirements then we have:

**Browser**: IE, FF, Opera, Safari

**ScreenResolution**: 800x600, 1024x768, 1280x800

**Javascript**: JSEnabled, JSDisabled

**Cookies**: CkEnabled, CkDisabled

Knowing these parameters and their possible values, we can get the pairs that need to be captured in order to perform pairwise testing (see Table 1.2). If we would want to perform an exhaustive testing for pairwise testing, we would need to generate 44 test cases (see Table 1.3). However, we saw that this is very impractical and it would take longer to
complete. On the other hand, if we are able to capture all the pairs in a smaller set of test cases, then we could parallelize the tests needed for pairwise testing, by carefully choosing the values instead of using the “Any value” slots. Such is the example shown in Table 1.4, which shows 12 test cases containing all the 44 pairs, and where it can be observed how we make the most use of each test case, by testing other value combinations, which would have very likely been repeated for an exhaustive testing.

It is important to mention that we do have the means to check if a set of test cases is able to capture all pairs, but we do not have the certainty of knowing whether we have the smallest set or not, because a smaller set could exist, one that could capture all pairs as well.

The problem of generating a small set of test cases starts to get worse when the number of parameters and valid values increases, as it becomes more complex and less intuitive. Therefore, the first problem to be solved in this thesis is:

\begin{quote}
\textit{to develop an algorithm able to generate and minimize as much as possible the number of test cases that contain all parameter pairs of values that can be given as input to a software system, in order to perform pairwise testing.}
\end{quote}

1.2.2 The Tooling Problem

Another problem is that, there is no available tool for researchers that can be used to solve the algorithm problem using genetic algorithms. There is no open-source code for experimenting the application of genetic algorithms to pairwise testing, for example, by adjusting genetic algorithm parameters such as fitness function, genetic operators, etc. Given that genetic algorithms have a large number of configurable parameters, it is difficult to adapt them to a specific problem.

If a researcher is interested in exploring the uses of genetic algorithms, for generating and reducing the number of test cases, he or she would need to rewrite the code to be able to satisfy specific configuration needs. Also, given that genetic algorithms have many
Table 1.2: All 44 pairs that need to be captured by the test case set

<table>
<thead>
<tr>
<th>Pairs</th>
<th>Browser</th>
<th>Screen Resolution</th>
<th>Javascript</th>
<th>Cookies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IE</td>
<td>FF</td>
<td>Opera</td>
<td>Safari</td>
</tr>
<tr>
<td>Pair 1</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 2</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 3</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 4</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 5</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 6</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 7</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 8</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 9</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 10</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 11</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 12</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 13</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 14</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 15</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 16</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 17</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 18</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 19</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 20</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 21</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 22</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 23</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 24</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 25</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 26</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 27</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 28</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 29</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 30</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 31</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 32</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 33</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 34</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 35</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 36</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 37</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 38</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 39</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 40</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 41</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 42</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 43</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair 44</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Case</td>
<td>Browser</td>
<td>Screen Resolution</td>
<td>Javascript</td>
<td>Cookies</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>-------------------</td>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>TC₁</td>
<td>IE</td>
<td>800x600</td>
<td>Any value</td>
<td>Any value</td>
</tr>
<tr>
<td>TC₂</td>
<td>IE</td>
<td>1024x768</td>
<td>Any value</td>
<td>Any value</td>
</tr>
<tr>
<td>TC₃</td>
<td>IE</td>
<td>1280x800</td>
<td>Any value</td>
<td>Any value</td>
</tr>
<tr>
<td>TC₄</td>
<td>IE</td>
<td>Any value</td>
<td>JSEnabled</td>
<td>Any value</td>
</tr>
<tr>
<td>TC₅</td>
<td>IE</td>
<td>Any value</td>
<td>JSDisabled</td>
<td>Any value</td>
</tr>
<tr>
<td>TC₆</td>
<td>IE</td>
<td>Any value</td>
<td>Any value</td>
<td>CkEnabled</td>
</tr>
<tr>
<td>TC₇</td>
<td>IE</td>
<td>Any value</td>
<td>Any value</td>
<td>CkDisabled</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>TC₄₄</td>
<td>Any value</td>
<td>Any value</td>
<td>JSDisabled</td>
<td>CkDisabled</td>
</tr>
</tbody>
</table>

Table 1.3: Example of an exhaustive test case set for pairwise testing

variations in terms of the genetic operators and configurations that could be used, it would be necessary to have a mechanism that could enable an easy switch from one genetic operator to another and from one configuration to another.

Let us suppose that, we already have a genetic algorithm that solves the problem of generating and reducing test cases. But, we realize that for a certain problem domain – indicated by the number of input parameters and values of a system – our algorithm has difficulties or takes much longer time to find a solution. Therefore, we would like to modify the code to experiment with different genetic operators that could potentially improve the algorithm’s behavior for that particular problem domain. This means that we need to recompile our program to be able to use it and test our modifications to the genetic algorithm.

Therefore, the second research problem of this thesis is:

*to develop a tool that could serve as a framework in a way that it could make the configuration of the genetic algorithm easier, where one is able to add variations*
Table 1.4: An example set of test cases for pairwise testing

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Browser</th>
<th>Screen resolution</th>
<th>Javascript</th>
<th>Cookies</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC_1</td>
<td>IE</td>
<td>800x600</td>
<td>JSEnabled</td>
<td>CkEnabled</td>
</tr>
<tr>
<td>TC_2</td>
<td>IE</td>
<td>1024x768</td>
<td>JSDisabled</td>
<td>CkEnabled</td>
</tr>
<tr>
<td>TC_3</td>
<td>IE</td>
<td>1280x800</td>
<td>JSEnabled</td>
<td>CkDisabled</td>
</tr>
<tr>
<td>TC_4</td>
<td>FF</td>
<td>800x600</td>
<td>JSEnabled</td>
<td>CkDisabled</td>
</tr>
<tr>
<td>TC_5</td>
<td>FF</td>
<td>1024x768</td>
<td>JSDisabled</td>
<td>CkEnabled</td>
</tr>
<tr>
<td>TC_6</td>
<td>FF</td>
<td>1280x800</td>
<td>JSDisabled</td>
<td>CkDisabled</td>
</tr>
<tr>
<td>TC_7</td>
<td>Opera</td>
<td>800x600</td>
<td>JSDisabled</td>
<td>CkDisabled</td>
</tr>
<tr>
<td>TC_8</td>
<td>Opera</td>
<td>1024x768</td>
<td>JSEnabled</td>
<td>CkDisabled</td>
</tr>
<tr>
<td>TC_9</td>
<td>Opera</td>
<td>1280x800</td>
<td>JSDisabled</td>
<td>CkEnabled</td>
</tr>
<tr>
<td>TC_10</td>
<td>Safari</td>
<td>800x600</td>
<td>JSEnabled</td>
<td>CkEnabled</td>
</tr>
<tr>
<td>TC_11</td>
<td>Safari</td>
<td>1024x768</td>
<td>JSDisabled</td>
<td>CkDisabled</td>
</tr>
<tr>
<td>TC_12</td>
<td>Safari</td>
<td>1280x800</td>
<td>JSDisabled</td>
<td>CkDisabled</td>
</tr>
</tbody>
</table>
to the genetic operators or to modify the default behavior of a certain genetic operator without the need of recompiling the core of the algorithm.

1.3 Contributions

This thesis makes two different contributions. The first contribution is to give a solution to the problem of generating and reducing the number of test cases for pairwise testing by using genetic algorithms. The specific contributions include

(a) Heuristics were introduced in the genetic algorithm to aid in finding a solution faster. These heuristics help in producing individuals with better fitness, by making a chromosome analysis of the individuals, and thus make a more intelligent mutation than the random mutation.

(b) A series of experiments were conducted to evaluate different parameter combinations of the genetic algorithm, to be able to select the one that adapts more to the specific needs of a given problem.

The second contribution is to provide a genetic algorithm framework – PWiseGen – focused on generating and reducing the number of test cases for pairwise testing, which is reusable, configurable, and extendable. The specific contributions in this area include

(a) PWiseGen is reusable in that its source code doesn’t have to be modified when adapted to different problem domains.

(b) PWiseGen is capable of being extended in that if more variations for the genetic operators than the ones proposed in this research are desired, they can be introduced. Also, if a default behavior of a genetic operator needs to be overridden, it is possible to do so.
(c) PWiseGen uses a configuration file, which precisely serves to configure the algorithm’s behavior, so that changes can be made without the need of recompiling the source code.

(d) The framework was coded in Java [23], so that it could be executed in different platforms.

1.4 Related Work

Currently, there are some deterministic algorithms that are capable of generating test cases for pairwise testing. Some of the most common are mentioned below; it is not intended to explain how each one of them functions in detail, it is a brief description of how they work to give the reader an idea of the other existing approaches.

1.4.1 In-Parameter-Order

In-parameter-order [11] is a deterministic pairwise test set generator strategy, that basically works by starting the test set as if there were only two parameters, then extending by adding another parameter, and continues doing so for each additional parameter. For this, the strategy uses two additional steps when adding each parameter: horizontal growth, which enlarges the test cases to support the newly added parameter and assigns a valid parameter value; and – if it is needed – vertical growth, which adds new test cases when unable to capture all the pairs from the actual test set, assigning the missing values to generate all the pairs.

The advantage of this strategy is that the test cases can be reused for cases when you wish to add parameters that weren’t considered at the moment when the test set was generated. An example of this would be, when a new system version comes out that receives more parameters than the previous version, then we can reuse the test cases from the previous version and execute the algorithm just to produce the missing parameters.
The test cases can also be reused when possible values are added to a parameter.

### 1.4.2 Orthogonal Arrays

This strategy, also known as orthogonal latin squares [12], is another way of generating a pairwise test set. Orthogonal arrays are two-dimensional arrays of numbers which have a particular feature: when selecting two columns from the array, you will get an even distribution of the value combinations in the array. This means that there is a balance requirement; each pair is captured the same amount of times, which can cause the generation of unnecessary test cases. There are precalculated orthogonal array catalogs available on the web [22], where you only need to map your parameters with their possible values against factors and their levels respectively. Therefore, a suitable orthogonal array needs to be found, one that has the least amount of parameters and possible values required. When an ideal orthogonal array is not found, then you need to look for the smallest array that can handle the problem to be solved.

An example of an orthogonal array can be seen in Table 1.5, where there are four parameters that can have three different values.

### 1.4.3 AETG

AETG [13] is a commercial tool that uses another approach to generate test cases for pairwise testing. In general, it starts with an empty test set, then continues adding one test case at a time. Each time a test case is generated, this strategy produces a certain number of test cases as candidates based on a greedy algorithm, and selects the test case which has the largest number of pairs which have not been captured yet.

### 1.4.4 Genetic Algorithms

While searching for similar studies, two publications were found in which genetic algorithms were used to solve the algorithm problem of this thesis. One of them [10] is more focused
Table 1.5: Example of an orthogonal array with four parameters and three values each

<table>
<thead>
<tr>
<th>Run</th>
<th>Parameter1</th>
<th>Parameter2</th>
<th>Parameter3</th>
<th>Parameter4</th>
</tr>
</thead>
<tbody>
<tr>
<td>run1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>run2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>run3</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>run4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>run5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>run6</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>run7</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>run8</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>run9</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

on proving the feasibility of using genetic algorithms, but it doesn’t show all the genetic operators used or the actual flow of the algorithm; it doesn’t even show results obtained with the benchmark problems published at the pairwise testing official website [8]. The other paper written by McCaffrey [9] shows the genetic operators used, the algorithm flow in general terms and does show results obtained with the benchmark problems. However, it doesn’t provide the source code used to make experiments. In fact, some ideas from the mentioned paper were adopted to develop this research, such as the chromosome encoding and fitness function.

McCaffrey’s approach [9] uses standard genetic operators and does not make use of heuristics to reach to a solution faster. His approach also uses a fixed configuration, such as single crossover point, random mutation, a small mutation rate (0.001), etc. Compared with McCaffrey’s approach, as far as the number of test cases is concerned, PWiseGen shows better results in two of the six benchmark problems [8]; for the remaining problems, PWiseGen produced equal results for three problems, but was outperformed by McCaffrey’s approach for one problem.
Both publications didn’t mention any tool support or reusable framework.¹

1.5 Outline

The rest of this thesis is structured as follows:

In Chapter 2, I present my approach to solve the algorithm problem of this thesis, i.e., generating test cases for pairwise testing by applying genetic algorithms.

In Chapter 3, I present my approach to solve the tooling problem of this thesis, i.e., making PWiseGen a tool configurable and extendable that could serve as a framework.

In Chapter 4, the evaluation and results produced by PWiseGen can be found.

And lastly, in Chapter 5, I summarize my findings and conclusions, as well as future research directions.

¹I found through private communication with the author that there is no open-source code available for McCaffrey’s approach
Chapter 2

Approach for the Algorithm Problem

This chapter addresses the way to solve the algorithm problem – which consists in generating and reducing the number of test cases for pairwise testing by applying genetic algorithms. I present different parameters used in PWiseGen, such as chromosome encoding, fitness function, and genetic operators. For this, the same example introduced in Section 1.2.1 will be used throughout this chapter.

2.1 Use of a Genetic Algorithm

As it was mentioned in Chapter 1, generating the least number of test cases for pairwise testing is an NP-Complete problem. Given that there is no known algorithm for pairwise test case generation that provides optimum results, I decided to solve the algorithm problem by using genetic algorithms. The genetic algorithms specialize in maximization and minimization problems. In this particular case, what I want is:

\[
\text{to minimize the number of test cases and in turn capture all pairs from possible combinations of input parameters.}
\]

One of the challenges when attempting to make use of genetic algorithms is the large number of configurations that have to be defined for its proper use, and these depend on the problem we intend to solve. The solution that I propose is based on a standard genetic algorithm, with certain adjustments. To properly implement it with focus on the problem of generating the test cases for pairwise testing, the following tasks needed to be performed, which will be described in detail in this chapter:
• Encode test cases as chromosomes

• Define a fitness function

• Define genetic operators

• Define other configurations such as initialization strategy, population size and termination condition

2.1.1 Chromosome Encoding

This task refers to the way we are going to store information within an individual. We should remember that an individual from a given a population has to represent a possible solution to the problem. That is, if the objective of the algorithm is to produce a set of test cases for pairwise testing, then we should be able to store a set of test cases within an individual.

This is a crucial task, given that we have to select an efficient means to code and manipulate the information that we will be accessed by the genetic operators. The question here is, which chromosome encoding will provide us with more benefits? We have to consider if the chromosome is easy to manipulate and if the test cases are easy to reconstruct from the encoding. There are other implications that we need to consider, such as the space required to hold the chromosome information, the performance of manipulating the chromosome, etc.

I considered a bit sequence to be much more complex to handle, since we would need to somehow represent the different parameter values in small bit sequences of different lengths. Revisiting the example presented in Section 1.2.1, for the parameter browser we would have four possible values, which would need to be represented in two bits at least. For example, 00 would represent the value IE, 01 would represent the value FF, 10 would symbolize Opera and 11 would stand for Safari; for the parameter of screen resolution we would also need to use two bits: 00 would represent 800x600, 01 would be for 1024x768
and 10 for 1280x800; lastly, parameters for Javascript and cookies would only require one bit, where 1 would symbolize enabled and 0 would be for disabled.

Following this logic for encoding test cases, then a set of two test cases contained inside an individual could be [110010, 101011] which would be equivalent to [{Safari, 800x600, JsEnabled, CkDisabled}, {Opera, 1280x800, JsEnabled, CkEnabled}], where the two first bits of a test case indicate the browser, the following two for screen resolution, next to last bit for the value of Javascript and the last one for the value of cookies.

This type of encoding makes it difficult to implement the genetic operators for crossover and mutation, since the parameter values may have different bit lengths; we would need to be very careful to avoid generating “defective” test cases. For example, the test case {IE, 1280x800, JsEnabled, CkDisabled} represented by the bit sequence 000110, could suffer a mutation in the third bit, turning the sequence into 001110, which would provide an invalid value for the screen resolution position (since value 11 doesn’t exist). Something similar would happen when performing the crossover, we would need to be careful to avoid selecting an incorrect position that would produce defective test cases. These are potential problem sources when manipulating the chromosome; maybe we would gain time when performing the crossover and mutation with just using bitwise operations, but we would loose time validating and ensuring that an individual’s chromosome doesn’t result in an invalid sequence. Another disadvantage that we may face is that we would need a mechanism to rebuild test cases from the bit sequence, in order to be able to count the number of pairs contained in an individual.

A variation of the bit sequence that could be used for the chromosome encoding, is to take arbitrary fixed-length bit sequences, which would be mapped to one and only one test case. For example, let us suppose that the bit sequence 10110 is mapped to a certain test case, then this bit sequence would be used for obtaining or reconstructing the test cases representing the sequence in question. A chromosome would then be made from the concatenation of these bit sequences, to form a set of test cases. The downside to this approach is that it is also necessary to provide a mechanism to designate the mappings
between the bit sequences and the test cases in order to obtain the test cases. It should also be noted that it is not feasible to store all possible test cases for mapping them to a code, given that it would require a large amount of space. Therefore, it would be necessary to find an intelligent way to map them and save space. Also, another disadvantage found with this type of encoding is that we would still need to verify that the bit sequence of a test case does exist; it’s possible that after an individual’s chromosome is modified via mutation, the resulting bit sequence may not be mapped to a test case. As a closing comment, it is also necessary to take into account that, when using this type of encoding, modifying only one bit of a test case could lead us to a test case completely different to the original, which may not be a desired result, but to keep the same test case with a slight change.

To summarize, while evaluating the possibility of using a bit sequence encoding, I found that it would cause more problems in the long run. This is why I opted for a more intuitive encoding and easier to manipulate, an integer array encoding.

**Using an Integer Array Encoding**

By using an integer array encoding we can avoid many of the problems described earlier. The only disadvantage here is that we have to manipulate arrays instead of bit sequences, and as a result we would loose performance when handling such arrays. The idea of using arrays of integer values was suggested in McCaffrey’s work [9].

The basic idea for this encoding is to store test cases in groups of integer numbers. These integers are IDs which are assigned to each possible value. Going back to the example in Section 1.2.1, the values for parameter *browser* in this case would be: 0 for IE, 1 for FF, 2 for Opera, and 3 for Safari. Likewise, the values for parameter *screen resolution* would be: 4 for 800x600, 5 for 1024x768, and 6 for 1280x800. The values for *Javascript* would be 7 to indicate that Javascript is enabled and 8 when disabled. The same would apply for 9 and 10, they would be used to indicate whether *cookies* are enabled or disabled, respectively. Therefore, when using this encoding, an arbitrary test case would be ⟨2, 5, 8, 9⟩ which equals to ⟨Opera, 1024x768, JsDisabled, CkEnabled⟩.
Figure 2.1: Chromosome encoding using an integer array

The chromosome of an individual would then be formed by groups of integer numbers, and depending on the number of test cases stored in an individual and the number of parameters contained in each test case, this is the number of slots that we will have in the integer array (see Figure 2.1).

Each test case is implicitly limited in the array depending on the number of parameters that each test case contains. And, each slot of every test case represents a parameter, which can have only the valid values for that parameter.

After assessing advantages and tradeoffs, I considered that implementing this encoding in my algorithm was a good option, given that with this approach you can easily rebuild the test cases contained within an individual, in a way that it makes simple to count for all its pairs. With this encoding we don’t have to worry about selecting a crossover point that doesn’t generate defective test cases, since each value is stored independently inside a slot.
2.1.2 Fitness Function

A fitness function determines how good an individual is; an individual’s fitness is what gives the individual the possibilities to remain in the population and to be chosen for reproduction. This is one of the most important parts of a genetic algorithm because we need to have a way to measure each individual in a way that the strongest individuals are selected for producing newer and even stronger individuals to achieve an evolution in the population.

To decide on which fitness function would help my algorithm for yielding good results, I had to ask myself the following questions:

- What do we want to achieve? What is the objective of the genetic algorithm?
  
  As an answer, I concluded that we want to obtain a set of test cases which contains all pairs of parameter values.

- What features must an individual have to be considered strong?
  
  A good individual is one that contains many pairs, the more different pairs an individual has, the more we are close to our objective, which is to obtain all the parameter value pairs.

Based on these questions and answers, I decided to implement two fitness functions for PWiseGen: calculating different pairs and penalizing for repeated pairs.

Calculating Different Pairs

This function measures an individual’s fitness based on the number of different pairs included in its chromosome. This is a very straightforward fitness assessment, given that, the larger number of different pairs, the higher the possibilities of capturing all pairs. This function was also suggested by McCaffrey [9].

In order to calculate it, I use a 2-dimensional array [21] of boolean values, where the pairs contained in the chromosome of an individual are being annotated and each new pair
increases the fitness of that individual. What the fitness function would be doing is to put flags into the slots where each pair is cross referenced in the array (see Figure 2.2). With this approach we can check if a pair was already counted.

Given that the total number of pairs that should be captured is known and easy to calculate, as shown in Figure 2.3, we can easily verify if an individual contains a solution. An individual is a solution if the number of different pairs contained in its chromosome equals the total number of required pairs.

**Penalizing for Repeated Pairs**

This function is a variation of the *different pairs function*, where fitness is penalized when the same pair appears more than five times in an individual.

Basically, fitness is calculated in the same way as done by the different pairs function. We count the number of different pairs, but at the same time, we also count the number of repeated pairs after finding the same pair more than five times. The final fitness value
Figure 2.3: All pairs that need to be captured by a chromosome equals to \textit{numberOfDifferentPairs} – \textit{numberOfRepeatedPairs}.

This fitness function was implemented as part of initial experimentations to emphasize that we are interested in finding individuals that have different pairs, but not those that have many repeated pairs. This implementation remained as part of the tool’s variations for future research, as it didn’t show an improvement to the genetic algorithm.

2.1.3 Genetic Operators

This is another important task that has to be performed. Defining the genetic operators is like establishing the rules of evolution that will be applied to the population. In this research, I decided to implement the most common genetic operators: selection, crossover, replacement, immigration, and mutation. There was no particular reason to select these genetic operators, I just decided to give a try with the most common and those that have proven to be more beneficial in other problem domains.

When trying to adapt these genetic operators to the problem of generating and reducing the number of test cases, I also come up with some variations that would help me find a solution faster. An example is a smart mutation that will be explained later in this chapter.
Selection

The most common way to select parents for reproduction is using the roulette wheel selection method [18]. The probability of being selected is assigned to each individual of the population. This probability is calculated based on the fitness value of an individual. Individuals with higher fitness values have better chances of being selected.

I decided to implement a roulette wheel selection approach because it gives the opportunity for individuals with a low fitness value to participate in the reproduction, although having less probabilities of being selected. Even low fitness individuals could contain chromosomes with pairs that have not been captured by high fitness individuals. I implemented the roulette wheel selection by using a cumulative fitness array, as shown in Figure 2.4. The concept of “spinning the wheel” is basically to request a random number between zero and the population fitness – the sum of every individual’s fitness. This is used to determine in which pocket did the wheel fall, and thus obtain the selected individual.

Crossover

The crossover allows us to combine individuals that were selected for reproduction. The basic idea is to produce better individuals by combining the chromosomes of the selected parents. There are many ways to perform the crossover, and at first I didn’t find a logic to define what type of crossover could be most beneficial for the particular problem of generating test cases for pairwise testing. This is why I implemented four different types of crossover:
Single crossover point: In this method, a crossover in a single point is made, and the crossover point is the middle of the chromosome. This crossover point remains effective throughout the algorithm’s execution.

Single random crossover point: This method is used when a single crossover point in a variable position is desired. Each time a crossover is made, the position where the crossover happens is determined randomly.

Multiple crossover points: In this method, the crossover is made at various points of the chromosome. The number of crossover points can vary, and depending on the number of crossover points, the crossover positions will be divided evenly over the chromosome. Once determined, the crossover positions will remain the same throughout the whole execution of the algorithm.

Multiple random crossover points: This method also makes the crossover in several points of the chromosome, but unlike the previous, the positions where the crossover happens is always varying. Each time when two individuals are to be combined, random positions for the crossover are determined.

In Chapter 4, the assessments and evaluations of each crossover type will be presented. The idea of having these four options is to configure the algorithm so that it can adapt to the different problems as well as to serve as a reference for future investigations.

Replacement

When two parents are combined and a reproduction occurs, two new descendants are generated and become members of the population by substituting two existing members. For the replacement step, I implemented two ways of doing it.

Weaker individuals replacement: I considered this method to be the most convenient for my algorithm, given that we would have the certainty that a strong individual
would never be replaced. This method always selects two individuals with the lowest fitness values, to be replaced by the two new descendants.

**Selected parents complements:** This method was taken from McCaffrey’s work [9] and removes those individuals whose ranks are the complement of the parents ranks. For example, if we have a population of 30 individuals, and, if a rank 1 parent and a rank 5 parent were selected for reproduction, then the chosen to be replaced would be those having rank 30, complement of rank 1, and rank 26, complement of rank 5. The reason is that even the least fit individuals contain valuable information and should have some measure of protection from elimination.

In Chapter 4, an experimental evaluation of each of the types of replacement will be shown.

**Immigration**

This operation is used to introduce some randomness to the population during the execution of the algorithm. For P WiseGen, a configurable option indicates when immigration will happen. That is, for every $n$ number of generations, a new individual with a randomly generated chromosome is introduced to the population by replacing one of the existing individuals.

I decided to include this genetic operation to avoid stagnation in the population, where an improvement is not being achieved and when individuals are not able to obtain more pairs by using the genetic operators. The immigration genetic operator may improve the population by introducing some randomness to it and diversifying the individuals.

**Mutation**

Mutation plays a very important role for finding a set of test cases for pairwise testing. Through experiments I found that without mutation, the population shows improvement only in the first generations, reaching a stagnation in the subsequent generations. I also
learned that a high mutation rate makes the genetic algorithm find a solution faster. There are studies [16] stating that the optimum mutation rate strongly depends on the selection of the chromosome encoding and algorithms that don’t use bit sequences encodings could benefit from high mutation rates.

For PWiseGen, several mutation schemes were implemented. One of the implementations that I made is a random mutation that is done based on certain probability. This probability is given by the mutation rate. In random mutation, the value of one of the chromosome’s genes – gene position selected randomly – is switched to a new randomly selected value. This new value has to be valid for the slot where the mutation is taking place (see Figure 2.5). The other mutation schemes will be explained in the following section.

### 2.1.4 Smart Mutation

In this research, I introduced the notion of a *smart mutation*. As the name implies, this mutation tries to evolve populations in a more intelligent manner. This mutation was...
invented to find a solution faster, in terms of the number of iterations required.

Three different smart mutations were introduced: similarity mutation, value occurrences mutation and pair occurrences mutation. The behavior of these mutations can be controlled by the following parameters:

**Probability:** The mutation with this modality is the one that occurs based on a set probability, specified by a mutation rate.

**After $n$ generations:** This modality means that the mutation will always take place, but only after $n$ number of generations.

**Before $n$ generations:** This modality has the opposite effect of “After $n$ generations”, where the mutation takes place at the start of the algorithm until it reaches an $n$ number of generations.

**Every $n$ generations:** The mutation with this modality is configured in a way that the mutation occurs every $n$ generations.

**No improvement in $n$ generations:** When the fitness value of the best individual hasn’t changed for more than $n$ generations, then mutations begins to occur. If an improvement is detected in the fitness of the best individual, then mutation stops occurring.

**Similarity Mutation**

The objective of this mutation is to replace similar genes – i.e., the test cases. The logic behind it is that, if an individual has two very similar test cases, it is very probable that the individual’s fitness will improve if we replace one of these test cases with another. If we leave the similar test cases in an individual’s chromosome, we would be wasting space that could potentially capture pairs that the individual didn’t have.

This mutation requires a percentage of similarity threshold, which is configurable and basically determines whether two test cases are similar or not. For example, if a 75% is
specified as percentage, then it will replace the test cases that have at least 75% similar values.

In the example presented in Figure 2.6, the test case 2 is replaced because it’s similar to test case 1. The new test case can be generated randomly or generated using the values that have less occurrences in the chromosome, depending on the configuration provided.

**Value Occurrences Mutation**

This mutation focuses on finding values which are not present in any of the test cases that are contained within the chromosome of the individual to be mutated. It introduces such values to raise the number of pairs that the individual has. The way it introduces them is verifying which values have the most occurrences, so that they are replaced by the ones without any occurrences. The reasoning behind it is that, if an individual doesn’t possess a value within its chromosome, it means that it’s missing the opportunity of making pairs that it didn’t have. If we are able to introduce them, then the fitness would surely increase, or as a worst case scenario, it would stay the same, but it would never make it worse. An example case is shown in Figure 2.7.
Figure 2.7: Value occurrences mutation

Pair Occurrences Mutation

This mutation is a bit more complex than the value occurrences mutation but it has more or less the same principle, searching for pairs that are not included in any of the existing test cases of the individual. Overall, it has an added risk, given that a fitness increase cannot be guaranteed when replacing pairs, due that other pairs may be lost indirectly. This mutation requires that we provide a threshold to determine if a pair can be replaced by another. For example, if the threshold is set to be equal to 5, and if the number of occurrences of a pair is 6, then that pair is marked for replacement. On the other hand, if pairs stay within the threshold, they are considered to be safe from being replaced.

Figure 2.8 shows how this mutation works. Also, the risk mentioned earlier is visualized. For example, when modifying Test case 1 we would gain two new pairs, but we would also lose pairs that were generated with value 8, for example \{4, 8\} and \{8, 9\}, and we would generate a duplicated pair \{7, 9\} (in Test case 1 and in Test case 3). This is why sometimes an improvement can’t be reached with this mutation. The most likely situation where an improvement can be found is when there are excessive number of duplicates.

This mutation was implemented and left as a reference for future investigations, given that it could prove to be beneficial on certain situations.
Figure 2.8: Pair occurrences mutation
2.1.5 Other Parameters

These are several other parameters that could be configured to adjust the genetic algorithms. Examples of such parameters are population size, population initialization strategy and the termination condition.

**Population size:** Population size is the maximum number of individuals allowed in a population. To design this algorithm, it was decided to make this parameter configurable, so that it can be modified to adapt to specific problems.

**Population initialization strategy:** As default, the initial population is created randomly. Here, the chromosome for each individual is generated, and each slot of its genes is filled with a random generated value, which is within the allowed values for the position being filled.

**Termination condition:** As with the majority of genetic algorithms, the termination condition is met when a satisfactory solution is found or when the maximum number of generations is reached. For PWiseGen, the maximum number of generations is a configurable value.

2.1.6 The Algorithm

Now that we learned about all the parameters used in PWiseGen, it is time to learn about how the algorithm works. The following pseudocode shows the flow of activities performed by PWiseGen:

- get the parameters and their values
- calculate the total number of pairs possible
- create an initial population
- set generation to 1
- calculate the fitness of every individual
- sort the individuals based on their fitness values
while (generation ≤ max number of generations) and (solution not found) do

for number of reproductions required do

select two parents
produce two children combining parent’s chromosomes

if mutation requested then
mutate child or children
end if

select two individuals to replace
remove the selected individuals from the population
calculate fitness values and add the new children to the population
end for

if immigration requested then
remove an individual from the population
generate a new individual, i.e., the immigrant
calculate its fitness value and add it to the population
end if
end while
Chapter 3

Approach for the Tooling Problem

As it was mentioned in Section 1.2.2, one of the problems to be solved is to turn the PWiseGen in a tool that can serve as a genetic algorithm framework for pairwise test generation.

To solve the tooling problem, the following goals were defined:

• To be able to configure PWiseGen in such a way that there is no need for recompiling the source code when changes to the genetic algorithm parameters are desired.

• To be able to extend PWiseGen, so we can have the ability to add other variations of the genetic operators without the need of recompiling the original code.

• To be able to change the default behavior of the currently implemented genetic operators in PWiseGen, without having to change the original code.

• To make of PWiseGen a plug and playable component, in such a way that can be incorporated in any other solution written in the same programming language.

In order to achieve these goals I used an object-oriented approach, in which I applied design patterns, used interfaces, polymorphism, among others. I also used a configuration mechanism that was easy to access from PWiseGen, such is the case of an XML-based configuration file, which user defined parameters can be added without affecting the predefined configuration structure.

This chapter presents the approach I used to solve the tooling problem. First, I show the architecture of my solution, which includes UML class diagrams with a brief explanation for each design decision. And after that, I present details of the PWiseGen configurability.
3.1 PWiseGen Architecture

The architecture of PWiseGen was designed using object-oriented concepts to facilitate components reusability and to make it easy to extend. For this, the main UML class diagrams is presented to show some of the design decisions (see figure 3.1).

In this architecture, the genetic algorithm is composed of different interfaces, and also makes use of other classes for its operation.

The genetic operators were implemented as interfaces. This is to make the implementing classes responsible for providing the behavior of the genetic operator. The GeneticAlgorithm class is only interested in knowing that it can rely on these classes, and that they have the necessary methods to make the actual work. This approach makes PWiseGen easy to extend, variations to genetic operators can be created by just implementing the interfaces.

The abstract classes that implement the genetic operator interfaces, provide part of the interface’s methods in order to be reused by the concrete classes, which are discussed later.
in this chapter. The classes `GAInfo` and `DomainInfo` contains certain information that can
be used by the genetic operators, these classes were used as a way to encapsulate several
pieces of information which are needed in order to decide how the genetic operator will
behave. The Config class is used to retrieve the genetic algorithm parameters to be used.
The `FitnessComparator` class is the one responsible of comparing the individuals in order to
have them sorted in the data structure. The `PopulationInitializer` interface is responsible
of creating the initial population.

In the following sections, each part of the class design will be described in detail, to
ingcreasingly have a better understanding of the architecture being used.

### 3.1.1 Crossover Strategy

In order to provide different ways of performing the crossover, the template method design
pattern was used [19]. Using this design pattern we can define the general steps of per-
forming the crossover in an abstract class, but letting the subclasses to have the details of
where the crossover will take place.

`CrossoverStrategy` is an abstract class that implements the `Crossover` interface, in which
a default implementation of the main method `crossOver` is provided (see Figure 3.2). To
make the crossover, this method only has to know the position (or positions) in which the
crossover will take place; this is why the abstract method `getCrossoverPoints` was created,
so that the classes that extend from the `CrossoverStrategy` are the ones that provide the
implementation of the abstract method.

In a similar way, the abstract method `getCrossoverFlag` is used within the `crossOver`
method, so that the subclasses can implement a logic which determines whether a crossover
with the individuals will be made or not. An example of this is when you wish to make
the crossover based on a certain probability. Therefore, if the `getCrossoverFlag` method
returns a `false` value, then the crossover would not take place.

The classes `SingleCrossover`, `SingleRandomCrossover`, `MultipleCrossover` and `Multi-
pleRandomCrossover` are implementations that were created as part of PWiseGen (as it
was explained in Section 2.1.3). It is also worth mentioning that classes \textit{GAInfo} and \textit{DomainInfo} precisely contain information of the genetic algorithm and information of the domain respectively (see Section 3.1.8 for more information). These classes are sent from the main class \textit{GeneticAlgorithm} through the \textit{setExtraInfo} method, so that they are made available to all the subclasses of \textit{CrossoverStrategy}.

### 3.1.2 Fitness Function

\textit{FitnessFunction} is an abstract class that implements the \textit{Fitness} interface, which only provides implementation for the \textit{setExtraInfo} method. The method \textit{calculateFitness}, is invoked from within the main class \textit{GeneticAlgorithm}, but this implementation is not provided in the abstract class. Therefore, the classes extending from \textit{FitnessFunction} are forced to implement the \textit{calculateFitness} method, which returns an integer value indicating an individual’s fitness (see Figure 3.3).

The classes \textit{DifferentPairsFitness} and \textit{RepeatedPairsPenalizationFitness} are fitness func-
tion implementations used in PWiseGen (see Section 2.1.3).

In the same way, classes GAINfo and DomainInfo are sent from the main class GeneticAlgorithm (see Section 3.1.8), if additional information for fitness calculation is desired.

### 3.1.3 Mutation Strategy

The abstract class MutationStrategy implements the Mutation interface, and it provides a default implementation for the method mutate to perform the mutation, which is based on a random mutation (see Section 2.1.3). The template method design pattern [19] was also used in order to let the subclasses of MutationStrategy decide whether a mutation is made or not, by implementing the abstract method getMutationFlag (see Figure 3.4).

Classes NoMutation, SingleMutationRate and SmartMutation are mutation strategies implemented for PWiseGen, as it was explained in sections 2.1.3 and 2.1.4.

### 3.1.4 Replacement Strategy

The abstract class ReplacementStrategy implements the Replacement interface, which possesses methods which are used by the main class GeneticAlgorithm and by the subclasses
Figure 3.4: Mutation class diagram
of ReplacementStrategy. It contains utility methods for inserting individuals in a population, for removing them, replacing them, etc. (see Figure 3.5) but it only requires for the subclasses to implement the abstract method replaceIndividuals to provide the logic behind of which individuals are to be replaced.

Classes WeakerIndividualsReplacement and ParentsComplementReplacement are replacement strategies implemented for PWiseGen (see Section 2.1.3).

3.1.5 Parent Selector

The abstract class ParentSelector implements the Selection interface (see Figure 3.6), it only provides implementation for the setExtraInfo method, and it lets its subclasses to implement the method selectTwoParents which is used by the main class GeneticAlgorithm. The class RouletteWheelSelector provides the logic for selecting individuals for reproduction.
3.1.6 Population Initialization

The class `RandomPopulationInitializer` implements the `PopulationInitializer` interface (see Figure 3.7). The `GeneticAlgorithm` class makes use of the `createPopulation` method in order to initialize the population. A random initialization was implemented in PWiseGen as the default initializer.

3.1.7 Utility Classes

The GeneticAlgorithm class makes use of several utility classes (see Figure 3.8).

Config Class

This class is responsible for obtaining the configuration values that dictates the behavior of the algorithm. As shown in Figure 3.8, it has methods to obtain all types of values required by the algorithm. But, an important feature is that user defined values can also be obtained which weren’t considered in this solution, and these are obtained by using the method `getUserDefinedValue`. These configurable user defined values make PWiseGen
Figure 3.7: Population initialization class diagram

Figure 3.8: Utilities class diagram
extensible, so if another value is needed, it can be added to the configuration and retrieved from any other class which implements the genetic operators.

**ParametersFileReader Class**

This class is in charge of reading the input file, which is the one containing either the parameters and their possible values, or, only the amounts for each parameter and valid values (see Section 3.2.1 for more information).

**ResultsFileWriter Class**

This class is in charge of writing the algorithm’s progress in a text file, which is basically information about the current generation and the best fitness so far. This is made to obtain the necessary information for analysis and for plotting results in a chart.

**RandomGenerator Class**

This class is used for obtaining random values, either for integer values or double values.

### 3.1.8 GAInfo and DomainInfo

As we have been seeing in previous diagrams, classes *GAInfo* (which stands for Genetic Algorithm Information) and *DomainInfo* are propagated in such a way that they are made available to the classes implementing genetic operators. These classes contain useful information, ready to be used by whichever element requires it.

As seen in Figure 3.9, the class *GAInfo* has information, such as:

- **Population**: This is the array of individuals in the population.
- **Generation**: This holds the current generation.
- **Population Fitness**: This is the sum of every individual’s fitness.
- **Solution**: This is the individual containing a solution.
Last Best Fitness: The fitness value of the best individual from the previous generation.

No Improvement Count: The generation counter which increments when no improvement is registered for the best individual’s fitness.

Last Two Selected Parent Ranks: The ranks of the last two parents selected for reproduction.

These data was being added to the class GAInfo according to when they were required for genetic operator implementations. These are some required and some extra information, which may help when deciding the behavior of a genetic operator.

The class DomainInfo basically contains the Parameter and Value collections. This is in fact the starting set of information to be used for generating the test cases. It also contains the total number of pairs which should be captured by an individual, to be considered as a solution. The classes Individual, Parameter and Value complement the information contained in GAInfo and DomainInfo.

Individual: This is a class representing a possible solution, which contains a chromosome, indicated by the integer array genes, where the GeneticAlgorithm class is going to apply the genetic operators.

Parameter: This class represents a parameter that we want to test, such as the browser, screen resolution, javascript and cookies.

Value: This is a class that represents a possible value of a parameter, such as IE, FF, Opera and Safari.

3.1.9 Summary

This architecture allowed me to achieve three of the four goals defined at the beginning of this chapter. The remaining goal will be explained in next section.
Figure 3.9: Information class diagram
With this approach, PWiseGen is able to be extended, more variations can be added to
the tool by just subclassing any of the abstract classes that implements the genetic operator
interfaces and implementing the required methods, or also by implementing directly the
interface and providing all the methods defined in the interface. With this, the GenetigAl-
gorithm class, will be able to use the new genetic operator variation without any change
or recompilation of the original source code.

The goal regarding the ability to change the default behavior, is achieved thanks to the
polymorphism. Since default behavior was implemented in PWiseGen for genetic operators,
we can replace this default behavior by extending from the abstract classes and overriding
the implemented methods.

Finally, PWiseGen can be used as a plug and playable component, since it can be
incorporated in any other solution by providing the implementation of the genetic operator
interfaces. Let’s suppose that there is another tool which have a class that can be used as
one of the genetic operators, they only need to implement the desired interface, and provide
the required method implementations, so this other tool can now call the PWiseGen engine
using their own classes.

3.2 Configurability

In order to achieve configurability in PWiseGen, an XML-based [20] file was used. XML is
a widely used format, and it is fully compatible with most of the programming languages.
It also provides flexibility when adding information to the XML file, which was one of the
important features to achieve the extensibility when adding user defined parameters to the
configuration.

PWiseGen also uses an input file with certain specific format, which will be discussed
later. This input file is used to determine the parameters and values that will be used to
generate the pairwise test cases.
3.2.1 The Input File

The input needed for executing PWiseGen is basically an XML configuration file *ga-config.xml*, which will be thoroughly explained in Section 3.2.2. It would also require a second file (*input.txt*) which could have two different formats:

- Parameters Format

  When the file is written in this format, it means that all parameters and their possible values will be specified in the text file. For example:

  Browser: IE, FF, Opera, Safari
  ScreenResolution: 800x600, 1024x768, 1280x800
  Javascript: JSEnabled, JSDisabled
  Cookies: CkEnabled, CkDisabled

- Counts Format

  This format indicates that the file only includes the amount of parameters and the quantity of possible values. This format was implemented for experimenting with the benchmark problems published in the Pairwise Testing’s page [8]. For example:

  15:4
  17:3
  29:2

  Which indicates that we want to generate the test cases for 15 parameters that can have 4 different values each, 17 parameters with 3 possible values, and 29 parameters with 2 possible values. When the algorithm reads this format, what it does is generate names for all parameters and for all values. For example, it would generate the parameter *aaa* with all its 4 possible values *aaa0, aaa1, aaa2, aaa3*; also, the parameter *aab* with its 4 possible values *aab0, aab1, aab2, aab3*; and so on.
3.2.2 The Configuration File

The configuration is stored inside the file *ga-config.xml* in XML format, which contains the attributes and necessary values, so that the algorithm knows how to behave. A configuration example is shown in the Figure 3.10, where:

**IsParamsFile**: specifies the type of format of the file *input.txt*, be it parameters and specific values, or, only quantities of parameters and values (as explained in Section 3.2.1).

**PrintEveryX**: specifies that current progress of the algorithm will be sent to the file every X generations.

**PopulationSize**: is the maximum number of individuals allowed in the population.

**TestSetSize**: is used to define the algorithm’s goal. It is the number of test cases that each individual will contain.
MaxGenerations: is to indicate the termination condition if a solution is not found.

FitnessFunction: is where you specify the fully qualified name of the class to be used as a fitness function. Note that the class must implement the Fitness interface or extend from the abstract class FitnessFunction.

CrossoverStrategy: is the fully qualified name of the class to be used as a crossover strategy. The class must implement the Crossover interface or be extended from the abstract class CrossoverStrategy.

ParentSelector: is the fully qualified name of the class to be used as a parent selector. The class must implement the Selection interface or be extended from the abstract class ParentSelector.

ReplacementSelector: is the fully qualified name of the class to be used as a replacement selector. The class must implement the Replacement interface or be extended from the abstract class ReplacementSelector.

MutationStrategy: is the fully qualified name of the class to be used as a mutation strategy. The class must implement the Mutation interface or be extended from the abstract class MutationStrategy.

PopulationInitializer: is the fully qualified name of the class to be used as the population initializer. The class must implement the PopulationInitializer interface.

MutationRate: is the probability for a mutation to occur used in the random mutation, having 1 as the maximum value. Other examples could be 0.5, 0.1, 0.001, etc.

NumberReproductions: is the number of reproductions that will take place in each generation. Typically, the number of reproductions will be equal to the half of the population size.
**ImmigrantEveryX:** indicates the amount of generations where an individual will be introduced to the population with random values.

**NumberCrossoverPoints:** is the value used when we make use of the crossover strategies *MultipleCrossover* and *MultipleRandomCrossover*.

**UseSimilarityMutation:** *true* applies if you wish to use the smart mutation *Similarity Mutation*, otherwise *false*.

**SimilarityMutationType:** Values could be *PROBABILITY*, *AFTER_X*, *BEFORE_X*, *EVERY_X* and *NO_IMPROVEMENT_X*, as explained in Section 2.1.4.

**SimilarityMutationValue:** is the value to be used, depending on the type. If *PROBABILITY* was selected, then the value for this attribute will be taken as the probability for a mutation to occur.

**ThresholdSimilarity:** is the similarity percentage (see Section 2.1.4).

**ReplaceWithSmartTestCase:** *true* if you wish to replace test cases with one that has the least number of occurrences in its values, or *false* for replacing test cases randomly.

**UseValueOccurrenceMutation:** *true* if you wish to use the smart mutation *Value Occurrences Mutation*, otherwise *false*.

**ValueOccurrenceMutationType:** uses the same logic and values as the attribute *SimilarityMutationType*.

**ValueOccurrenceMutationValue:** uses the same logic as the attribute *SimilarityMutationValue*.

**BalanceValueOccurences:** This attribute was left as part of the experimentations, which could serve on future research. This configuration indicates that, when a value is not found in the individual’s chromosome, then it tries to introduce it, but at the same
time it balances the number of occurrences with the value which appears the most. This only happens when this attribute is set to true.

**UsePairOccurrenceMutation**: *true* if you wish to use the smart mutation *Pair Occurrences Mutation*, otherwise *false*.

**PairOccurrenceMutationType**: uses the same logic and values as the attribute *SimilarityMutationType*.

**PairOccurrenceMutationValue**: uses the same logic as the attribute *SimilarityMutationValue*.

**ThresholdPairOcurrences**: Value used in the smart mutation *Pair Occurrences Mutation* (see Section 3.2.1).

If there is a need to add more attributes to the configuration file, it is only a matter of selecting an attribute name, and assigning a desired value. This will make the value available from any class without the need to re-compile the core of the genetic algorithm. For example:

```xml
<entry key="MyAttribute">MyValue</entry>
```

And to obtain this value, it is only a matter of invoking the static method *getUserDefinedValue* of the class *Config*. For example:

```java
String myValue = Config.getUserDefinedValue("MyAttribute");
```

### 3.2.3 Summary

With the use of the XML file *ga-config.xml* and the *input.txt* file, I achieved the remaining goal of having a way to configure PWiseGen without modifying the source code, therefore there is no need to re-compile whenever we want to change the parameters of the genetic algorithm.
4.1 Evaluation of PWiseGen for Solving the Algorithm Problem

For evaluating the genetic algorithm, a series of experiments were conducted to prove its effectiveness. These experiments are summarized in four categories:

1. Comparison of PWiseGen against a purely random test set generation in terms of the number of generations or iterations needed to find a solution.

2. Comparison of effectiveness of PWiseGen against other approaches. This comparison was measured by the number of test cases generated after the algorithms ended their execution.

3. Comparison of smart mutation and random mutation, in terms of the number of generations needed to find a solution.

4. Comparison of different configurations such as single crossover point, multiple crossover point, smart mutations, etc., also measured in terms of the number of generations needed to find a solution.

4.1.1 Comparison Against Purely Random Test Set Generation

The objective of this evaluation is to show that PWiseGen can find a solution at least faster than the purely random test set generation. The speed was not measured in time, but in terms of the number of iterations (or generations) required to find a solution.
In order to be able to perform this test, an algorithm that could generate populations of individuals randomly was created. The way this random algorithm works is that in each generation a new population is generated randomly, and each population's individual is inspected to check whether it contains a solution. This continues successively until the termination condition is met, or, until a solution is found.

Two experiments were executed to see if PWiseGen is able to outperform the algorithm which generates populations randomly. In these experiments, each algorithm was executed ten times in order to get an average. The results are shown below.

**Experiment 1**

In this experiment, both algorithms were executed to find the set of test cases which captures all the necessary pairs for an input of 4 parameters with 3 possible values for each, which in turn would be 54 pairs in total. The test set size that was given to both algorithms was of 9 test cases, which means generating all pairs in 9 test cases.

**Result.** As shown in Figure 4.1, the random algorithm was not able to find the result after 100,000 generations; on the other hand, PWiseGen found the result in generation #322.
Experiment 2

In this experiment, both algorithms were given an input of 1 parameter with 4 possible values, 39 parameters with 3 possible values and 35 parameters with 2 possible values, which would be 17,987 pairs. The test set size provided was 30.

Result. As shown in Figure 4.2, the random algorithm was again unable to find a solution after 10,000 generations; on the other hand, PWiseGen found a solution in 2035 generation.

With these two experiments we can conclude that PWiseGen outperforms the strategy of generating test cases randomly.

4.1.2 Comparison of Effectiveness Against Other Approaches

The objective of this evaluation is to compare results generated by PWiseGen against other existing approaches, which are published in the pairwise testing website [8]. This website shows effectiveness results when executing algorithms with 6 benchmark problems, which are the same tests that I made to try out with PWiseGen. The effectiveness is measured by the number of test cases generated after the algorithm ends its execution. That is, the less the number of test cases generated, the more effective the algorithm.
CPU time required to reach a result is out of the scope of this thesis, therefore is
not considered in this evaluation, because it is difficult to make comparisons with other
approaches, due to differences in platforms, CPU speed, number of CPU cores, amount of
RAM, etc. It would be difficult to simulate the exact same environments as the ones used
for the other algorithms. Also, this information is not provided by any of the algorithm
developers; they just make emphasis on their respective effectiveness. However, I also
included the CPU time that it took to reach my results, using a computer with a Pentium
dual core 1.60 GHz and 1 Gb of RAM. Therefore, CPU information shown in my results is
only for the record and should be taken in only as reference information.

The Table 4.1 shows some of the tools with their effectiveness results. These tools were
selected because there is more information about them and papers that cover details of how
the algorithms were implemented. The algorithm AETG [13] was explained in general in
Chapter 1, IPO [11] uses the in-parameter-order algorithm (see Chapter 1), TConfig [14]
and CTS [15] are algorithms which use orthogonal arrays, and lastly, GAPTS [9] makes
use of a genetic algorithm.

Parameter sizes shown in Table 4.1 are a way of representing the number of parameters
and possible values. For example, $4^{15} \times 3^{17} \times 2^{29}$, means that there are 15 parameters of 4
values on each, plus 17 parameters of 3 values each, and 29 parameters of 2 values each.

**Benchmark Problem 1**

This experiment was made giving PWiseGen an input of 4 parameters with 3 values each;
the required goal is to generate a test set able to capture 54 pairs. For this, a test set size
of 9 test cases was used.

**Result.** As shown in Figure 4.3, PWiseGen was able to find the result after 322 generations
and it found a solution in less than a second.
Table 4.1: Comparison of effectiveness for benchmark problems solved with existing approaches

<table>
<thead>
<tr>
<th>Problem</th>
<th>Parameter Sizes</th>
<th>AETG</th>
<th>IPO</th>
<th>TConfig</th>
<th>CTS</th>
<th>GAPTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$3^4$</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>$3^{13}$</td>
<td>15</td>
<td>17</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>$4^{15} \cdot 3^{17} \cdot 2^{29}$</td>
<td>41</td>
<td>34</td>
<td>40</td>
<td>39</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>$4^1 \cdot 3^{39} \cdot 2^{35}$</td>
<td>28</td>
<td>26</td>
<td>30</td>
<td>29</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>$2^{100}$</td>
<td>10</td>
<td>15</td>
<td>14</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>$10^{20}$</td>
<td>180</td>
<td>212</td>
<td>231</td>
<td>210</td>
<td>196</td>
</tr>
</tbody>
</table>

Figure 4.3: Results of PWiseGen for benchmark problem 1
Benchmark Problem 2

This experiment was made by giving PWiseGen an input of 13 parameters of 3 values each; and the result being sought is to generate a test set which captures 702 pairs. For this effect, a test set size of 15 test cases was used.

Result. As shown in Figure 4.4, PWiseGen was able to find a solution in the generation #958,878; and it took approximately half an hour to find this solution.

Benchmark Problem 3

This experiment was made by giving PWiseGen an input of 15 parameters with 4 values each, 17 parameters of 3 values each, and 29 parameters with 2 values each; therefore it is required to generate a test set which can capture 14,026 pairs. For this, a test set size of 34 test cases was used.

Result. As shown in Figure 4.5, PWiseGen was able to find a solution in the generation #34,005. It took approximately half an hour to find this solution.

Benchmark Problem 4

This experiment was made by giving PWiseGen an input of 1 parameter of 4 values, 39 parameters of 3 values each, and 35 parameters of 2 values each; it is necessary to generate
a test set which is able to capture 17,987 pairs. For this, a test set size of 26 test cases was used.

**Result.** As shown in Figure 4.6, PWiseGen was able to find a solution in the generation #9,102. It took approximately 10 minutes to find this solution.

**Benchmark Problem 5**

This experiment was done by giving PWiseGen an input of 100 parameters of 2 values each; it is required to generate a test set able to capture 19,800 pairs. For this, a test set size of 10 test cases was used.

**Result.** As shown in Figure 4.7, PWiseGen was able to find a solution in the generation
#512,249 and taking approximately 6 hours in doing so.

**Benchmark Problem 6**

This test was made by giving PWiseGen an input of 20 parameters of 10 values each; it is required to find a test case able to capture 19,000 pairs. A test set size of 220 test cases was used.

**Result.** As shown in Figure 4.8, PWiseGen was able to find a solution in the generation #298,949. It took approximately 4 hours to find this solution.
Summary

Taking into account the results obtained with PWiseGen and comparing them with results published for the other existing approaches, it can be observed (see Figure 4.9) that PWiseGen shows competitive results, only being outperformed on the benchmark problem #6. But when compared against the rest, PWiseGen shows better or equal results.

4.1.3 Comparing Smart Mutation and Random Mutation

This evaluation was made with the intention of proving the effectiveness of a smart mutation when compared with random mutation. For this, ten trial runs were conducted, with each of these mutation strategies. With the results gathered, an average of the generation (where the result was found) was calculated.
Table 4.2: Comparison of smart mutation and random mutation

<table>
<thead>
<tr>
<th>Run</th>
<th>Smart Mutation</th>
<th>Random Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>49832</td>
<td>32539</td>
</tr>
<tr>
<td>2</td>
<td>12944</td>
<td>42395</td>
</tr>
<tr>
<td>3</td>
<td>51267</td>
<td>39412</td>
</tr>
<tr>
<td>4</td>
<td>45236</td>
<td>88907</td>
</tr>
<tr>
<td>5</td>
<td>33882</td>
<td>20545</td>
</tr>
<tr>
<td>6</td>
<td>30436</td>
<td>40692</td>
</tr>
<tr>
<td>7</td>
<td>26657</td>
<td>70925</td>
</tr>
<tr>
<td>8</td>
<td>67996</td>
<td>82352</td>
</tr>
<tr>
<td>9</td>
<td>19225</td>
<td>30883</td>
</tr>
<tr>
<td>10</td>
<td>41484</td>
<td>62790</td>
</tr>
<tr>
<td></td>
<td><strong>Average</strong></td>
<td><strong>51144</strong></td>
</tr>
</tbody>
</table>

For this, an input of 13 parameters with 3 values each was provided, using a test set size of 18.

The results of the trial runs are shown in Table 4.2, which in average show that the use of smart mutations is 1.35 times faster than random mutation.

4.1.4 Comparing Different Configurations of PWiseGen

This evaluation was made due to the large number of configurations that could be made to PWiseGen, which could become confusing when selecting an appropriate configuration for a given problem. I am presenting the results of several trial runs here, with different sample configurations focused on solving the benchmark problems, which I named as problem domains in this section.

For these trials, an initial configuration was taken as a starting point (see Figure 4.10),
in which several trial runs were made for each problem domain, so that an average could be obtained and thus be able to have a point of comparison when using configurations different than the initial. It is worth mentioning that for the execution of the mentioned trial runs, a best (known) test set size was not used, instead a larger test set size was used to be able to make more trials in less time.

Figure 4.10: Initial configuration as starting point

Nine different configurations were selected, which only have one variation when compared with the initial configuration:

1. Using SingleRandomCrossover
2. Using MultipleCrossover with 2 crossover points
3. Using MultipleCrossover with 5 crossover points
4. Using MultipleRandomCrossover with 3 crossover points
5. Using MultipleRandomCrossover with 5 crossover points
6. Using ParentsComplementReplacement

7. Using 0.1 as Similarity Mutation rate

8. Using 0.1 as Value Occurrence Mutation rate

9. Using 0.1 as Pair Occurrence Mutation rate

For each problem domain, the same test with the same configuration was executed ten times, to later obtain an average of the number of generations needed to find a solution.

**Problem Domain 1**

This problem domain refers to the type of problems where the input is similar to the one in benchmark problem 1, in which we have 4 parameters with 3 values each. These tests were omitted due the fact that with the initially proposed configuration is more than enough to find a solution in less than a second.

**Problem Domain 2**

This problem domain refers to the type of problems where the input is similar to the one in benchmark problem 2, where we have 13 parameters with 3 values each. For these trials, a test set size of 18 was used.

**Result.** As shown in Table 4.3, there are some considerable differences between configurations, for example the configuration #6 is not effective for this problem domain. Therefore, when using the initial configuration, it is enough for attacking this type of problems, however the use of 2 crossover points (as used in configuration #2) shows a little bit more improvement.

**Problem Domain 3**

This problem domain refers to the type of problems where the input is similar to the benchmark problem 3, in which we have 15 parameters with 4 values each, 17 parameters
Table 4.3: Comparison of different configurations - problem domain 2

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result found in generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>174</td>
</tr>
<tr>
<td>1</td>
<td>223</td>
</tr>
<tr>
<td>2</td>
<td>142</td>
</tr>
<tr>
<td>3</td>
<td>181</td>
</tr>
<tr>
<td>4</td>
<td>276</td>
</tr>
<tr>
<td>5</td>
<td>154</td>
</tr>
<tr>
<td>6</td>
<td>1932</td>
</tr>
<tr>
<td>7</td>
<td>280</td>
</tr>
<tr>
<td>8</td>
<td>154</td>
</tr>
<tr>
<td>9</td>
<td>213</td>
</tr>
</tbody>
</table>

with 3 values each and 29 parameters with 2 values each. For these tests a test set size of
40 was used.

Result. As it can be seen in Table 4.4, the use of multiple random crossover slightly
improves the speed when finding the result, as it was applied in configurations #4
and #5.

Problem Domain 4

This problem domain refers to the type of problems where the input is similar to the one
in benchmark problem 4, where we have 1 parameter with 4 values, 39 parameters with 3
values each and 35 parameters with 2 values each. For these trials, a test set size of 30 was
used.

Result. As it can be seen in Table 4.5, there is no substantial difference between using one
configuration or another except with configuration #6, but it continues to be better
Table 4.4: Comparison of different configurations - problem domain 3

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result found in generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>432</td>
</tr>
<tr>
<td>1</td>
<td>766</td>
</tr>
<tr>
<td>2</td>
<td>525</td>
</tr>
<tr>
<td>3</td>
<td>531</td>
</tr>
<tr>
<td>4</td>
<td>418</td>
</tr>
<tr>
<td>5</td>
<td>376</td>
</tr>
<tr>
<td>6</td>
<td>3248</td>
</tr>
<tr>
<td>7</td>
<td>515</td>
</tr>
<tr>
<td>8</td>
<td>540</td>
</tr>
<tr>
<td>9</td>
<td>846</td>
</tr>
</tbody>
</table>

Table 4.5: Comparison of different configurations - problem domain 4

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result found in generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>253</td>
</tr>
<tr>
<td>1</td>
<td>217</td>
</tr>
<tr>
<td>2</td>
<td>260</td>
</tr>
<tr>
<td>3</td>
<td>231</td>
</tr>
<tr>
<td>4</td>
<td>187</td>
</tr>
<tr>
<td>5</td>
<td>171</td>
</tr>
<tr>
<td>6</td>
<td>&gt;1000</td>
</tr>
<tr>
<td>7</td>
<td>232</td>
</tr>
<tr>
<td>8</td>
<td>225</td>
</tr>
<tr>
<td>9</td>
<td>207</td>
</tr>
</tbody>
</table>
Table 4.6: Comparison of different configurations - problem domain 5

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result found in generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>256</td>
</tr>
<tr>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>268</td>
</tr>
<tr>
<td>3</td>
<td>253</td>
</tr>
<tr>
<td>4</td>
<td>182</td>
</tr>
<tr>
<td>5</td>
<td>172</td>
</tr>
<tr>
<td>6</td>
<td>&gt;1000</td>
</tr>
<tr>
<td>7</td>
<td>261</td>
</tr>
<tr>
<td>8</td>
<td>268</td>
</tr>
<tr>
<td>9</td>
<td>245</td>
</tr>
</tbody>
</table>

to use multiple random crossover points, as it was done with configurations #4 and #5.

Problem Domain 5

This problem domain refers to the type of problems where the input is similar to the one in benchmark problem 5, where we have 100 parameters with 2 values each. For these trials, a test set size of 12 was used.

Result. As it can be appreciated in Table 4.6, once again, it becomes clear that it is better to use multiple random crossover point for this problem domain, as it was done in configurations #4 and #5.
Table 4.7: Comparison of different configurations - problem domain 6

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result found in generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>3692</td>
</tr>
<tr>
<td>1</td>
<td>4315</td>
</tr>
<tr>
<td>2</td>
<td>4684</td>
</tr>
<tr>
<td>3</td>
<td>3610</td>
</tr>
<tr>
<td>4</td>
<td>4670</td>
</tr>
<tr>
<td>5</td>
<td>3034</td>
</tr>
<tr>
<td>6</td>
<td>&gt;10000</td>
</tr>
<tr>
<td>7</td>
<td>4068</td>
</tr>
<tr>
<td>8</td>
<td>4677</td>
</tr>
<tr>
<td>9</td>
<td>4645</td>
</tr>
</tbody>
</table>

**Problem Domain 6**

This problem domain refers to the type of problems where the input is similar to the one in benchmark problem 6, where we have 20 parameters with 10 values each. For these trial runs, a test set size of 250 was used.

**Result.** For this and the other problem domains, results leaned towards the multiple random crossover point (in this case with 5 crossover points), as a beneficial operator for finding a solution faster, as shown in table 4.7.

**Summary**

The results show that the use of the configuration #5 always outperforms the other configurations in the problem domains 3, 4, 5 and 6, where the number of parameters and possible values start to increase. When a small number of parameters and values is given, like in the case of problem domain 2, the use of configuration #2 shows the best results.
4.2 Evaluation of PWiseGen for Solving the Tooling Problem

The need to create a framework presented itself when I started to modify the source code, each time that I needed to experiment with a variation of a genetic operator. Having to adjust the source code each time, and then return it to its original state after the experimentation, it started to become tedious, also being a potential cause of possible errors of wrongly set configurations. Therefore, for this evaluation the following goals were defined.

4.2.1 Goals

1. To be able to modify the genetic algorithm configuration without having to alter the original source code.

2. To have the ability to extend PWiseGen without having to alter the original source code.

3. To have the chance of modifying default behavior of a given genetic operator without having to alter the original source code.

4. To be able to re-use the genetic algorithm from another Java project.

The following evaluations were performed in order to test whether or not these goals were achieved.

4.2.2 Modifying the Maximum Number of Generations

Supposing that we want to modify the maximum number of generations, so that the algorithm stops when the 10th generation is reached. Then, we would only need to modify the variable called \textit{MaxGenerations} in the configuration file \textit{ga-config.xml}, as shown in Figure 4.11.
4.2.3 Creating a New Mutation Operator

Let us suppose that we want to add a new genetic operator for mutation, where we wish that the mutation doesn’t occur based on probability, but every other round, once yes and not in the next, indefinitely.

For this we need to create a class that extends from `MutationStrategy`, as shown in Figure 4.12 \(^1\). Being extended from `MutationStrategy`, we have to provide an implementation of the method `getMutationFlag` in our new class, given that it is an abstract method. Therefore,

\(^1\)If we are going to work over a separate project, then we have to include the JAR file of PWiseGen in the project’s build path. We also have to put the configuration file `ga-config.xml` and the input file `input.txt` in the project’s root directory, so that they are positioned in the “user.dir” path which is the user’s current working directory
the code would remain more or less like the following:

```java
package mypackage;

import edu.utep.pw.ga.mutation.MutationStrategy;

public class MyNewMutation extends MutationStrategy {

    boolean flag;

    @Override
    protected boolean getMutationFlag() {
        flag = !flag;
        return flag;
    }
}
```

To make use of this new class, the only thing we need to do is:

- To modify the configuration file to use our new genetic operator for mutations.

  ```xml
  <entry key="MutationStrategy">mypackage.MyNewMutation</entry>
  ```

- Execute the algorithm to start, as shown below:
package mypackage;

import edu.utep.pw.ga.GeneticAlgorithm;

public class Main {

    public static void main(String[] args) {

        GeneticAlgorithm ga = new GeneticAlgorithm();
        ga.start();
    }
}

With this example, goals #2 and #4 are achieved (see Section 4.2.1), given that we extended the tool when adding a new genetic operator for mutation without needing to change the original source code. Also, we made use of the genetic algorithm from a different Java project.

4.2.4 Overriding Default Mutation Behavior

Now, let’s suppose that we don’t want a random mutation as it is done by default with the method `mutate` of the class `MutationStrategy`. This time we want that the mutation to always modify the first slot or gene of the chromosome, and to place any arbitrary value for that specific slot.

Then, re-using the code from the previous example. Now we override the method `mutate`, and provide the logic for the mutation, remaining something similar to:
public class MyNewMutation extends MutationStrategy {

    boolean flag;

    @Override
    protected boolean getMutationFlag() {
        flag = ! flag;
        return flag;
    }

    @Override
    public void mutate(Individual individual) {
        if (this.getMutationFlag() == false)
            return;

        int gene = 0;
        int valueX = 0; // get some valid value
        individual.getGenes()[gene] = valueX;
    }
}

With this example, goals #3 and #4 are met (see Section 4.2.1), given that we modified the default behavior of the mutation genetic operator without the need to alter the original source code. Also, we made use of the genetic algorithm from a different Java project.

### 4.2.5 Summary

All the goals defined in Section 4.2.1 where achieved, the results of this evaluation show that PWiseGen can be configured without changing and re-compiling the source code, it can be extended so we can add variations to the genetic operators, and we can also override the default behavior of the genetic operators without having to modify or re-compile the original source code. Besides, it can be used from another Java project as a plug and
playable component.
Chapter 5

Concluding Remarks

As a conclusion, I can say that the results of this research were favorable, both problems presented in this thesis could be solved with the approaches proposed herein. The algorithm problem was solved via a genetic algorithm; also the tooling problem, which was solved by making that algorithm work as a framework.

The advantage of solving the algorithm problem with a genetic algorithm is that one can define the goal, in this case the number of test cases that one is aiming to obtain, and the maximum time that one is willing to wait for a solution to be found, which is something that deterministic algorithms lack. The genetic algorithm will try to find a favorable solution, if a solution does exist, it is possible to find it with a good use of different genetic operators.

In the evaluation stage, it was observed that PWiseGen is slow when trying to find the optimal known results. On the other hand, PWiseGen is fast when a certain flexibility margin is given, when it comes to the number of test cases.

There could be testing scenarios in which it’s better to sacrifice time to obtain test cases, instead of the time taken to execute the test cases themselves. For example, if it is considered to execute tests cases that take three days each one to finish, then it should be worthwhile to take time to obtain a small set of test cases. In other words, it is better to wait ten hours, for example, to obtain a small set of test cases, instead of executing many test cases, each of them taking three days to complete.

There may be other testing scenarios where it is not so important to generate the least number of test cases, given that each test case doesn’t take that much time to be executed. For these cases, PWiseGen could also be used, giving it a not-so-small test size, but reasonable for the parameters and values that need to be tested. In these situations,
it is obvious that the algorithm will obtain a solution much faster.

The advantage that PWiseGen has over the other existent proposal [9] (based on genetic algorithms), is that heuristics are used in my approach, which tries to attack this problem domain more directly. The other proposal is based on standard genetic operators, but it makes the process slower when finding a solution.

Speaking of the genetic algorithm itself, it could be observed that the genetic operator crossover only plays an important part in the first generations produced by the algorithm. On the other hand, the mutation operator plays an important part throughout the whole algorithm’s execution.

A very important aspect that I also wanted to solve with this thesis was the tooling problem, given that it’s difficult to find tools that can facilitate the modification of the algorithm’s behavior. With the approach of turning PWiseGen into a framework, now it’s easier to make experimentations without the need of re-writing code that was already made.

In fact, the tooling problem emerged by itself when trying to improve on my genetic algorithm; it was very difficult to be modifying it to implement some new ideas, and when these ideas were not favorable, I had to return my code to its original state. Making PWiseGen a framework allowed me to work faster on my own research.

5.1 Future Work

As future work, I present some research directions here:

- In this investigation only nine different configurations were evaluated. However, PWiseGen can support a large number of configurations and one which performs better than the ones evaluated in this thesis work may exist.

- The chromosome encoding could be subject for future research. A more efficient encoding could be invented, which could improve the algorithm’s performance.
• In this research, some of the variations for the genetic operators were provided, but other operators can still be explored which could improve the algorithm’s effectiveness.

• Given the nature of genetic algorithms, some of the algorithm’s internal processing can be parallelized, which could result in an improved performance.

• Given that the CPU time measuring of the algorithm was not considered as part of the scope of this research, all available algorithms could be collected and a testing environment could be prepared for making CPU time comparisons among them.

• As a way of speeding up the test with the different configurations and variations of the genetic operators, a good idea to implement would be that the algorithm is made able to support being run by a batch process, in which one could select the different inputs and configurations. This way, one leave the process running, without the need of setting up and monitoring each run.

• Another good feature to incorporate in PWiseGen would be to provide a friendly user interface, so that it wouldn’t be necessary to make the configurations directly in the XML file, or that one wouldn’t require to use the command line to execute the algorithm, or that the GUI would be made to show the statistics of several trial runs.
References


Curriculum Vitae

Pedro Flores was born in November 11th, 1981 in México. He finished his high school studies in the Spring of 1999 in the City of Cuauhtémoc, Chihuahua, México. Pedro later changed his residence to study for a degree in Juárez, Chihuahua, México, where he graduated from the Universidad Autónoma de Ciudad Juárez (UACJ) and obtained a Bachelor’s degree in Computer Systems Engineering. Just after graduation, he participated in a Summer scientific research program in Puebla, México where he collaborated in a research project at the National Institute of Astrophysics, Optics and Electronics (INAOE) for three months. In the Fall of 2004, he started working professionally as a Software Engineer at the UACJ and did so for a year and a half. In the Spring of 2006, he worked for almost 4 years in Electronic Data Systems (now HP) as a Systems Analyst; at this time he started with his graduate studies at The University of Texas at El Paso, pursuing a master’s degree in Computer Science.

Permanent address: Ciudad Juárez, Chihuahua, México