International Journal of Computer Applications (0975 - 8887)
Volume 187 - No.71, January 2026

Structural Test Data Generation using an Alterable Genetic Algorithm for Large Scale Branch Coverage

Djam Xaveria Youh, PhD
Department of Computer Science
University of Yaounde 1

Lol Abakar Adam
Department of Computer Science
University of Yaounde 1

ABSTRACT

Structural testing, also known as white-box testing, requires the generation of input data that ensures coverage of program structures such as statements, branches, and paths. The complexity of software development makes testing extremely challenging and demands novel approaches with significant advancement in the field. Manual testing by contrast, remains time-consuming and costly activity, accounting for almost 50% of software production costs. To address these challenges, several automated techniques have been explored, among which Genetic Algorithms (GAs) have emerged as one of the most widely adopted and studied approaches. GAs are a class of search-based optimization techniques inspired by natural selection, and they are particularly effective in solving complex search problems. However, the use of traditional GAs for structural test data generation suffers from premature convergence and population stagnation.
To address these limitations, we introduce the Alterable Genetic Algorithm (AGA), a novel approach in which a single genetic operator is applied per iteration. An adaptive alternation function dynamically selects the most appropriate operator for the current state of search, capitalizing on their respective strengths to guide the search more effectively. This paper investigates the extent to which AGA improves structural test data generation, particularly by achieving high branch coverage with fewer fitness evaluations.

General Terms

Software Engineering, Software Testing

Keywords

Structural testing, search-based software testing, genetic algorithms, test data generation, white-box testing

1. INTRODUCTION

Software testing is an essential phase of the software development life cycle (SDLC) witch consists of executing the program with the intent of finding errors [3]. Software development has become increasingly complex, making testing more difficult and requiring novel, more effective approaches.
Traditional testing methods are often not enough for large and complex systems. This has led to the rise of Search-Based Software Testing (SBST). Over the years, testing strategies and techniques have improved, supported by advances in technology and programming languages.
More broadly, Search-based Software Engineering (SBSE) is a field of software engineering research and practice that describes the application of search-based optimization techniques (metaheuristics) to software engineering problems.
The application of SBSE that concern software testing problems is called Search-based software testing (SBST). SBST was the first and most widely used sub-area of SBSE [11] [19].
Structural testing techniques focus on evaluating the internal structure of the software. Manual test data generation is costly and error-prone, motivating the need for automated approaches. SBST formulates test data generation as an optimization problem, where search algorithms like GAs can be employed effectively [11].

1.1 Problem Statement

Testing software systems, allow for early error detection, avoid error migration and it amplification in advanced SDLC phases, and consequently master the product quality. It then requires high quality test data, which is often difficult, expensive and time-consuming to generate manually.
Automated test data generation tools exist, but they often produce unrealistic test data and do not adequately cover all possible scenarios [14] [18]. This is extremely challenging for software testers who need to ensure that their test coverage is complete and efficient.
To address this problem, there is a need to develop more sophisticated and efficient methods for test data generation that accurately represents real-world scenarios and produces reliable test results.

Motivating Example

A simple Python function is considered that returns the middle value among three variables. Its implementation and corresponding control flow graph (CFG) are illustrated in Figures 1 and 2, respectively.
Testing the program below (figure 1) implies its execution with test cases. This process requires identifying with precision high quality test data that allows the test to go through all possible scenarios [21] [10]. That is to say, data that cover all the branches of the SUT, in order to detect possible hidden bugs.
Let B be the set of all branches in the *middle* function. For example, the **TRUE** branch from node 12 (Figure 2), denoted $b \in B$, must be executed with specific inputs to expose a potential fault

```
1  def middle(x, y, z):
2      if x < y:
3          if y < z:
4              return y
5          elif x < z:
6              return z
7          else:
8              return x
9      else:
10         if x < z:
11             return x
12         elif y > z:
13             return yy  #error
14         else:
15             return z
```

Fig. 1: Middle function code

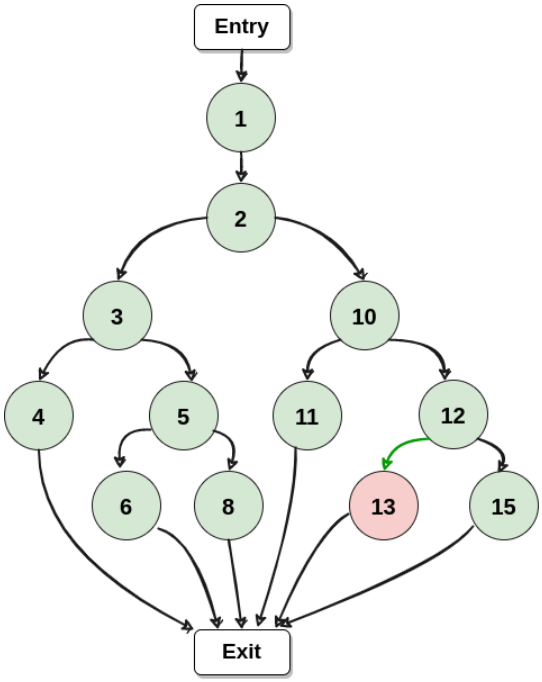

Fig. 2: Middle function code

in instruction 13 (Figure 1). Manually identifying such inputs is challenging and time-consuming, especially when considering all possible execution scenarios.
Let us extend our perspective beyond the middle function (Figure 1) to large-scale software systems comprising hundreds of thousands of lines of code. In such contexts, a central and enduring question in software engineering arises: **What is the computational effort required to systematically generate and evaluate test data for programs of this scale and complexity?**

Search-based test data generation using GA is a well-established approach, yet traditional GAs suffer from issues like **premature convergence** and **population stagnation**, caused by fixed operator rates and their simultaneous execution [8]. These limitations reduce their effectiveness in scenarios where the system under test does not exhibit Royal Road properties [1] [20].

The main objective of this research is to **Realize a novel approach based on Genetic Algorithm to automate structural test data generation for efficient branch coverage.**
The detailed objectives of this research in order to the aim achieve are as follows:

—Propose a novel variant of the Genetic Algorithm, referred to as the Alterable Genetic Algorithm (AGA);

—Decouple the application of genetic operators mutation and crossover in order to prevent their simultaneous execution, thereby enabling a more effective balance between exploration and exploitation through an **alternation function**;

—Evaluate the performance of AGA through a comparative empirical study with the standard Genetic Algorithm across various test programs.

In order to achieve the research aims and objectives based on the problems identified, it is therefore natural to ask the following primary question:

How efficient is search-based test data generation using Alterable Genetic Algorithm for given a set of real-world programs to achieve structural branch coverage?

Methodically address this main research question by distilling it down, which led to the following secondary research questions (**RQ1** and **RQ2**)

—**RQ1 :** How effective is the generation of test data based on an alterable Genetic Algorithm for branch coverage?

—**RQ2 :** What is the efficiency of structural test data generation based on Alterable Genetic Algorithm as opposed to the traditional Genetic Algorithm according to branch coverage criteria for real-world programs?

These questions will be points of reference for us, lines that will orient the research.

This paper focuses on the generation of structural test data transformed into optimization problems. The test criteria used to evaluate the optimazation techniques (**GAs** , **AGAs**) and the test data generated are also various and have different characteristics, but the one chosen in this study is the control flow criteria and specifically the branch coverage criteria.
The rest of this paper is structured as follows. In Section 2, the background and related works are reviewed. The proposed method is described in Section 3. Section 4 presents the experiments and results. Finally, Section 5 concludes the paper and outlines future research directions.

2. BACKGROUND & RELATED WORK

Software testing is verification plus validation, it is the process of verifying that a software system meets the specifications and requirements to ensure that it fulfills its purpose [3] [6].

2.1 Approaches to Software Verification and Validation

Multiple approaches have been used to verification and validation software, ringing static testing, dynamic testing to formal verification, however no single technique seems to be completely satisfactory [3]. Thus there is therefore a need for improving the methods for verification and validation based systems.

2.1.1 Static Testing. Static testing encompasses a category of software verification techniques aimed at identifying defects without executing the software under test (**SUT**). These techniques focus exclusively on the verification phase of the software testing process, analyzing artifacts such as code, documentation, and design to detect errors early [6]. A defining characteristic of all static testing methods is that they assess the SUT without requiring its runtime execution.
Static software testing techniques include:

—**Walk-throughs:** which is the process of inspecting the software by developer, development team, testers, users and customers to detect imperfections;
—**Code Inspection:** is characterized by going through the software source code to detect errors;
—**Requirements Analysis:** the process which involve reviewing the analysis document;
—**Analysis:** the process of reviewing the design document
—**Static Analysis:** Tools the process of using automated tools to analyze the software source code.

Test data generation for SUT control flow coverage, using graph-based criteria without any execution, requires measurements based on symbolic execution techniques [5] [14].

2.1.2 Dynamic Testing. The dynamic testing is a type of software testing techniques in which the dynamic behavior of the software system is examined by compiling and executing its code.
Dynamic testing uses techniques as white box or structural testing, black box or functional testing and can be deployed at different levels. There are different software test levels as shown in Figure 3, They present the test abstraction scale, and according to the SDLC, the principles are distinguished as follows:

—**Unit Testing:** Testing individual modules or units which can be either methods or entire classes of the software independently of the rest of the program,
—**Integration Testing:** Individual units and modules are integrated into a larger subsystem and then tested to validate the operation of a set of modules and programs developed and tested independently at the unit test level,
—**System Testing:** When units and subsystems are fully integrated into a system that constitutes the software product, acceptance testing of the software prior to delivery is applied to verify a version of the system.

This research focuses only on the dynamic testing methods that are applied to the first two levels. However, when it comes to the third level of testing, the system testing, another category of testing approach is noted, which is defined exclusively for this level as the formal method.

2.1.3 Formal Method. Software testing is a critical activity to ensure system reliability and user satisfaction, yet it remains challenging due to the inherent ambiguity and imprecision of specifications written in natural language [18].
Formal methods are design practices that use rigorously specified mathematical models to build software or hardware, thus allowing the mathematical foundation to specify the system in a complete, precise, clear and unambiguous way.

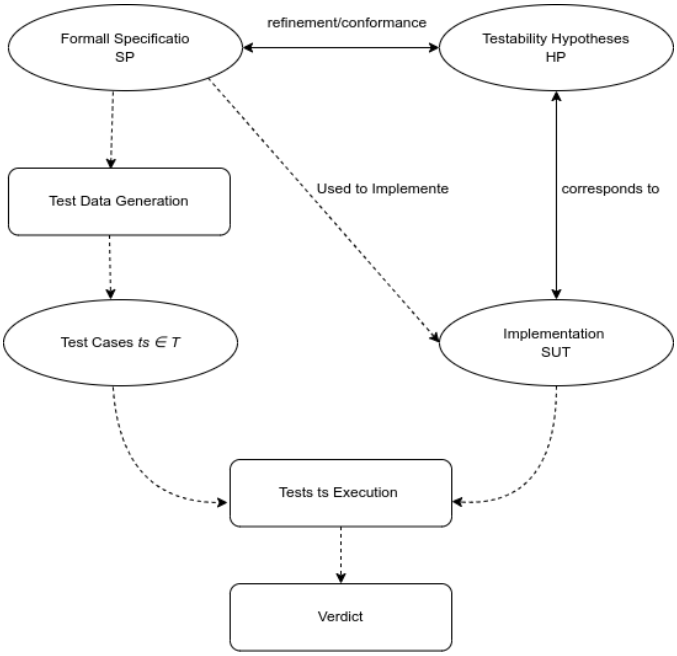

Fig. 4: Test process based on formal methods reproduced from [18, p. 7]

Formal methods are increasingly used in software engineering, particularly in software testing, where the following can be identified: **Z** based on the set theory and predicate logic, **CSP** (communicating sequential processes) based on concurrency theory or process algebra, **Promela** and **LOTOS** are both based on automaton theory etc.
In formal methods, a specification precisely defines the behavior of the SUT using a structured syntax and semantics to express properties such as preconditions, postconditions, and axioms. This allows for rigorous verification through formal proofs based on conformance or refinement relations between specifications.
However, since testing is performed on the implemented system rather than its specification as represented in Figure 4, a semantic gap emerges. To bridge this gap, testability hypotheses are introduced assumptions that connect the system's behavior to its formal specification, enabling meaningful testing within a formal verification framework [6] [18].

The figure (figure 4) above expresses the process of testing based on the formal specification, for a specification SP from which the SUT was developed. let HP be a set of testability hypotheses on the SUT, test cases ts belonging to the set of all possible tests T are generated from SP and then executed with the SUT, thanks to

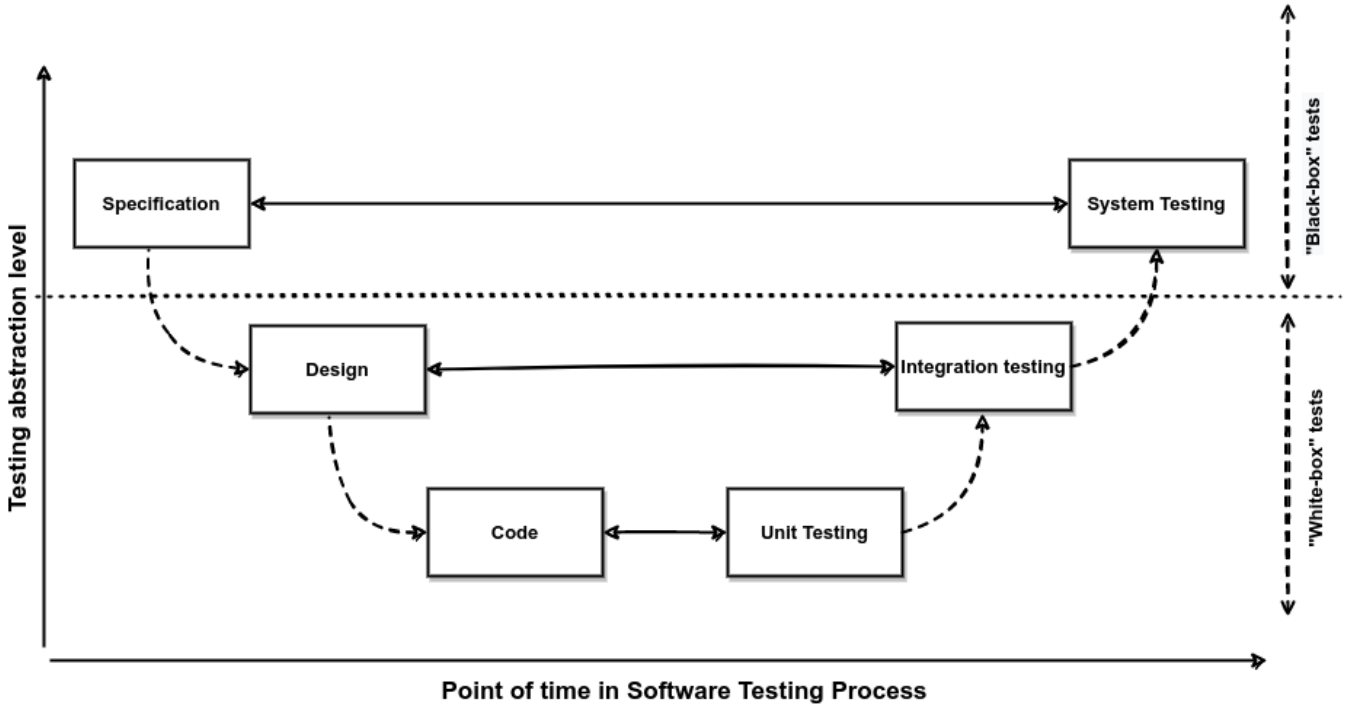

Fig. 3: Software testing levels

the inference system, proofs are formed and the test process can be guided, thanks to the conformance relation, a verdict is given as to whether the implementation of the SUT conforms to its formal specification and if so, the test of this implementation is successful, otherwise it is a failure and this is demonstrated by a counterexample [18].
Passing indicates that no evidence, counterexamples of non-conformance has been found on the test set T. The completeness of a test set T with respect to the specification SP, is the fact that the SUT passes all the tests of T if and only if it behaves as a conformance/refinement of SP, while taking into account the testability hyppothese on the SUT and which is summarised as follows :

$$\forall ts \in T,\ \ SUT\, pass\, ts \iff (HP(SP)\ conformance\ SP)$$

On the other hand, the failure of the test is explained by the existence of at least one counterexample to the mathematical reasoning presented above

Tretmans and Belinfante [18] used the **Promela** language to specify a labeled transition system used to implement a communication protocol for a conference system and developed a fully automated test tool **Torx** based on the formal specification in Promela that works in two modes: one manual, i.e., user-guided, and one automatic, i.e., guided by the formal deduction system, Using **Torx** they tested 28 SUTs of which there are 27 mutants where **Torx** was able to detect 25 of these 27 mutants tested, the other two mutants not detected by **Torx** is explained by the fact that the type of behaviour of the latter was not explicitly modelled in the Promela specification and therefore they conform to the specification, which brings us back to the fact mentioned above, Regarding formal methods, the focus is on testing a system rather than a specification.

2.2 Test Adequacy Criteria

Structural or white-box testing relies on test adequacy criteria, which provide metrics to evaluate the effectiveness of test cases in detecting faults and improving software quality. These criteria, primarily based on coverage, are commonly classified into control flow and data flow coverage techniques [3] [7].
This research is based on the flow criterea, which can be summed up as follows:

2.2.1 Statement coverage. The nodes of the CFG for a statement coverage criteria, represent an individual statement connected to that which represent the next statement by an edge, so the statement coverage criteria is defined as follows: each statement in the SUT must be executed at least once during testing.

2.2.2 Branch coverage. A branch is an instruction or a set of instructions, that can be executed under one or more different conditions, for example an **if** statement or a **switch** or even a **loop**. Branch coverage is a measure of the number of branches or decisions of a module that have been executed during testing.
The branch coverage criteria specifies that each branch of a conditional decision statement in the SUT must be executed at least once during testing. Note that branch coverage already includes statement coverage, so 100% branch coverage also means 100% statement coverage [15].

2.2.3 Path coverage. In path coverage every possible path in the flow of the SUT, must be executed at least once during testing. A path is a set of branches linked by previous ones if they exist, pass-

ing through conditional nodes. The path always starts from the entry node. It is important to note that there are infeasible paths for which, there is no input data that crosses it.

One of the most persistent challenges in testing techniques based on path coverage is managing the exponential growth of execution paths in the system under test (SUT). Boonstoppel et al,; [2] proposed a method to reduce the number of paths explored by eliminating those that are guaranteed to have side effects identical to previously analyzed paths. Their approach, known as RWset analysis, tracks all read and write operations performed by the program. Using this information, it prunes paths early when it detects that their execution would be equivalent to one already explored. This technique directly addresses **the path explosion** problem by eliminating redundant execution paths.
The adequacy criteria are varied and generally depend on the design of the tests and the SUT Some are included, some are stronger [15] [9] [7]. However, for this research the branch coverage are used as criteria to evaluate and compare our techniques and tests.

2.3 Search-based Test Data Generation Techniques

The automation of test data generation is the critical point of the evolving software test automation problem [1]. The techniques used for automatic test data generation are always accompanied by an application of adequacy test criteria, as a metric. This metric used is called a fitness function or objective function. In this perspective of software test automation, several approaches have been proposed, among which **Search-based optimization techniques** or **meta-heuristics** [11] [1] [19] [9].
Combining human intelligence and machine computational power to solve problems, with minimum human effort possible, meta-heuristics techniques are defined as computational intelligent paradigms for sophisticated solving optimization problems. The meta-heuristics often used are classified into two groups the single solution meta-heuristics or simple type meta-heuristics which are **Hill Climbing (HC), Tabu Search (TS), Simulated Annealing (SA) etc** and the population based algorithms which are : **Genetic Algorithm (GA), Genetic Programming (GP), Evolution Programming (EP) etc**.

Genetic Algorithm

GAs are a class of search-based optimization techniques from the population based family inspired by natural selection, The GA proceeds by a selection that determines the set of potential solutions also called as poulation that are maintained for evolution [1] [9] [19] . The evolution of the population is possible thanks to the genetic operators or evolution operators, which are the crossover and the mutation. These operations (selection, crossover and mutation) are iterated until the defined stopping criteria are satisfied, as shown in the figure (figure 5). The set of individuals (population) for one iteration, is called a generation [12].

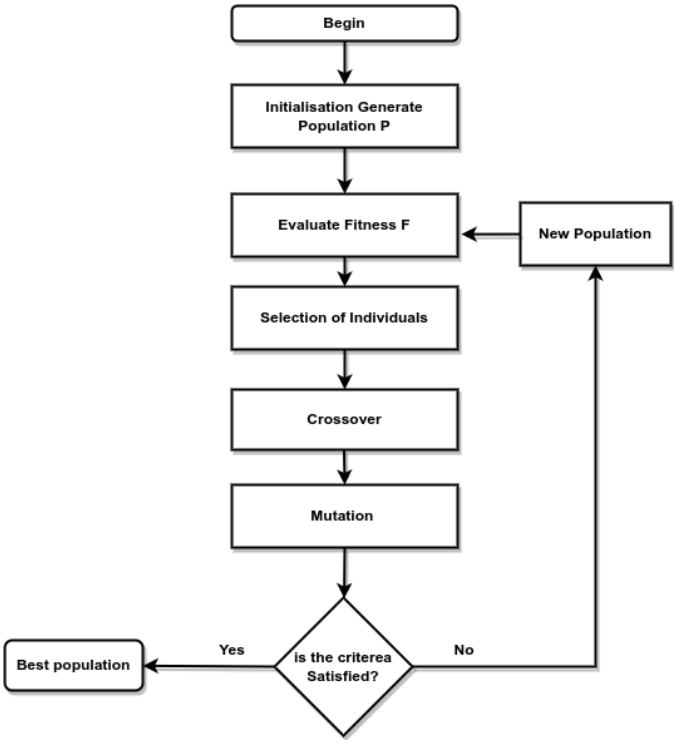

Fig. 5: Genetic Algorithm flow diagram reproduced from [1, p. 2]

Crossover. is a genetic operator designed to combine genetic material from two parent individuals at a selected point, generating offspring with the aim of inheriting beneficial traits from both parents. This mechanism enhances the exploitation of the current population by refining solutions through recombination, thus enabling the algorithm to explore the search space more effectively and improve its overall performance in successive generations.

Mutation. is a minor change in the genes of an individual. A form of mutation can be the replacement of one element of an individual's genetic code by another arbitrarily chosen element. It introduces diversity and promote exploration (searching new solution in new region).

Genetic Algorithm Limitations

GAs, as sophisticated as they are, have certain limitations. Among these limitations : **The premature convergence** problem, which represents a condition in which the algorithm prematurely converges to a sub-optimal or a local optimum solution, that the genetic operators (crossover and mutation) cannot produce offspring with better fitness afterwards. This phenomenon occurs when the population has lost its diversity.
Another GAs problem is the **population stagnation**, it refers to the situation in which the algorithm stagnates before converging to the optimal solution. One possible cause of this problem is the slowing down of the algorithm during evolutionary search, due to **inadequate operator design and their fixed rates** [8] [5].
These limitations are manifested as the relevant operators parameters are fixed in GA [8]. The joint execution of certain types of operators, constitutes a brake to the evolution of the solution's fitness.

A simple mutation operation after a crossover could, for example, make a potential solution unprintable by the implementation [1]. These problems are generally related to all evolutionary algorithms. Surprising as it is, the HC, a simple, less sophisticated type of algorithm would in some test generation scenarios outperform evolutionary algorithms and particularly the GA [12] This paradox is also explained by the fact that in some scenarios, crossover and mutation disrupts both the evolution of solution's fitness which is illustrated by Schema Theory [13].

Schema Theory. It is important to note that the aim of this section is not to present a comprehensive review of schema theory, but rather an intuitive introduction followed in each case by its generalization to software testing, as given by Harman and MacMinn [12].
GA's schema is a combination of a set of characters: $\{0, 1, *\}$, the $*$ is a wildcard which can be either a 1 or a 0, a schema can be considered as a model chromosome or a template. An instantiation is when a chromosome corresponds to a schema where the $*'s$ of the schema are replaced by the bit of the chromosome at the same position. A chromosome x is an instantiation of a schema s, is noted $x \in s$. The number of fixed positions (the number of bits except the $*$ or $|s - \{*\}|$) in a schema is called the schema order and is denoted $P(s)$.
It is almost impossible to determine the exact fitness value of a schema s because the GA selection operation will not identify a subset of solutions that will necessarily contain all possible instantiations of a schema, it is simply not designed for that.
On the other hand, at each generation i, the schema s will be able to be treated by determining an approximation of its fitness on the basis of its instantiations present in the generation $G(i)$.Thus one will be able to define a measure of the approximate aptitude, $\tilde{f}$(s,G(i)) , for a set of chromosomes of $G(i)$ from f(x,G(i)) function that gives the exact fitness of a chromosome x belonging to set that form the generation $G(i)$.

$$\tilde{f}(s, G(i)) = \frac{1}{|\{x|x \in s \wedge x \in G(i)|\}} \sum_{x \in s \wedge x \in G(i)} f(x, G(i)) \quad (1)$$

Returning to the SBST context, it is observed that the notions stated above are related to binary GAs. Thus to generate tests of real world programs it is useful to generalize them. To achieve this, a careful adaptation of the generalization provided by Harman and MacMinn [12] is applied for the schema to each formula.
Knowing that a schema s can be represented by a set of constraints c of the SUT and that depending on the SUT, the tests will not necessarily be represented in binary, let $h(x, c)$ be the function that determines whether a chromosome x satisfies c the constraint, then the generalization of the equation 1 gives us a formula like the following:

$$\tilde{f}(c, G(i)) = \frac{1}{|\{x|h(x, c) \wedge x \in G(i)|\}} \sum_{h(x,c)} f(x, G(i)) \quad (2)$$

The distance between the two most extreme fixed bit of the schema s, called the definition length. The definition length of s, will be denoted by $\delta(s)$.
Let the number of occurrences of s at generation i, be denoted by $N(s, i)$. The schema theory for binary GAs without the effect of mutation or crossover for a population of size L, given as :

$$N(s, i+1) \geq N(s, i) \frac{\tilde{f}(s, G(i))}{\frac{1}{L} \sum f(x, G(i))} \quad (3)$$

and its (equation 3) generalization for test data generation is given as follows: :

$$N(c, i+1) \geq N(c, i) \frac{\tilde{f}(c, G(i))}{\frac{1}{L} \sum f(x, G(i))} \quad (4)$$

Both mutation and crossover disrupt the schema, and create other patterns where the previous instances would not match. This can slow down the evolution of the solutions fitness. To take into account these effects, let us consider their respective perturbation effects as: Pm the probability that an individual bit is mutated and Pc the perturbation effect of the crossover operator. The size of a chromosome x is denoted by l. Thus the schema theory is extended from equation 4 and given as follows:

$$N(s, i+1) \geq N(s, i) \frac{\tilde{f}(s, G(i))}{\frac{1}{L} \sum f(x, G(i))} (1 - Pc \frac{\delta(s)}{l-1} - Pm.P(s)) \quad (5)$$

recall that $P(s)$ is the order of schema s. The generalization of the given schema theory for the generation of test data for real-world programs, while the order of the constraints in this case representing the order of the schema $P(s)$ and defined as the number of variables used to constitute the constraint predicate c designated by $P(c)$. For example, the constraint c constitutes by the following predicates: $(a + b \geq c)$ then the order of the constraint c will be equal to $3(P(c) = 3)$ because 3 variables a, b and c are used, so the generalization of the above formula (equation 5 is given as follows

$$N(c, i+1) \geq N(c, i) \frac{\tilde{f}(c, G(i))}{\frac{1}{L} \sum f(x, G(i))} (1 - Pc \frac{P(c)}{l-1} - Pm.P(c)) \quad (6)$$

However, genetic operators, especially the crossover remains essential in some scenarios. A **building block** is a set of fittest schemas which when recombined produce even more fittest schemas. **Royal Roads** are a simple function of separable building blocks. Used to test the building block hypothesis [20] [12] [4], which suggests that a GA will perform much better in the presence of building blocks because the crossover operation can assemble the best parts of two or more schemas, into one of the offspring's chromosomes, thus assembling lower order schemas of greater than or equal to average fitness could create higher order schemas that drive the search towards a convergence to the global optimum [20]. In the case of test data generation, to exhibit a royal road property, the best-fit schemas must be expressed as conjunctions of lower order schemas involving disjoint sets of input variables. When this property holds, the royal road theory predicts that the GA will perform well and that it will do so because of the presence of the crossover operator and the way in which the best-fitting schemas receive exponentially more leads than the worst-fitting schemas
Thus, to characterize the test data generation scenarios, It is noted that those favorable to GA are the ones adhering to Royal Road properties and non-favorable otherwise [20].

Numerous studies have applied evolutionary algorithms to software testing. This section aims to present some major works done in the field of search-based test data generation related to the GA.
Aljahdali et al,; [1] realized a review of some works done in the field of search-based test data generation and GA, a comparison of some techniques, are carried out and finally they have classified and presented some of the GAs **limitations in software testing**.
Harman and McMinn [12] conducted a theoretical and empirical study of Random Testing, HC and GAs, applied to structural test data generation, with various real world programs as SUT. Their results showed that evolutionary algorithms are suited in many situations, when it comes to generating input data for structural tests, while in some cases simpler techniques give surprising results, capable to overcome evolutionary algorithms including GA. They also proposed as solution, memetic algorithm that combines different local and global search techniques.
Khor and Gorgono [16] developed an automatic test generator, **genet**, that uses a GA and a Formal Concept Analysis used in the fitness function for branch coverage. The formal concept analysis uses data analysis concepts rather than the CFG, which avoids more human effort and keeps track of the test. The performance of **genet** has been tested and compared to a random test generator. The result showed, **genet** outperforms the random test in most of the stated SUTs and had almost the same level of efficiency when the density of solutions is high.
However, the good performance of a test generator does not depend essentially on its fitness function but on the GA itself and especially on the use of the genetic operators (mutation and crossover).
Mairhofer et al,; [17] developed **RuTeG**, a GA-based test data generator for dynamic languages that can handle complex object-oriented structures of the Ruby language. RuteG uses an analyzer, a generator to find domain-appropriate data, a test case executor, and finally a test case generator. The usability of RuTeG is compared to random testing. In the GA parameters, it is noted that the mutation operator is applied with a probability of 0.2. The results showed that RuTeG was able to achieve full code coverage in 11 out of 14 cases, where the lowest average code coverage was 88%. On the other hand, the random test case generator was only able to find test cases that covered all the code in 4 out of 14 cases.
However, GA limitations described by the Schema Theory, which present that both genetic operators disrupt schema is not addressed or at least it is insufficient to address it through low operator rates.

Han and Xiao [8], tried to address this problem with fluctuating operators' probability, the proposed called improved adaptive Genetic Algorithm, which involves the performance of conventional GA by dynamically determining the adaptive crossover probability and adaptive mutation probability, depending on the fitness of solutions. This allows a balance between exploiting the best solutions and exploring new search spaces. Thus, to avoid the local optimum. In this work, the proposed technique has found its application to the traveling salesman problem (TSP) where it has achieved good results compared to the traditional GA, but this does not eliminate any risk of perturbation caused by crossover and mutation because their execution remains joint.
Esnaashari and Damia [5] have also used a dynamic approach, to define the probabilities of crossover and mutation, then add a memetic step, in which the solutions founded, improve themselves using the reinforcement learning technique, Q-learning. This step added to GA in their approach consists in identifying the best solution or chromosome encoded in the form of a set of SUT test cases, then submit it to a memorization process in order to reinforce its aptitude by a sequence of checks and errors to modify the duplicate test cases encoded in this solution and replace one by another test case that improves the solution, before pass the solutions to the GA operators of crossover and mutation.

The method proposed by Esnaashari and Damia is therefore called **MAAT** for Memetic Algorithm for Automatic Test case generation. **MAAT** was compared to several recent meta-heuristic algorithms applied to the structural test data generation for path coverage, and showed better results than all the algorithms in terms of number of fitness evaluations and success rate. On the other hand, some of the test subjects in the experiments do not represent real-world programs, and so MAAT is also more considered as a specific approach to the path coverage criteria, since its memetic step depends especially on the paths that are not yet covered by the solution to be improved.
Previous work has shown promising results, but has also identified challenges related to premature convergence and population stagnation.

3. THE PROPOSED METHOD

As exhaustive testing is impossible, Test data generation could have near-optimal solutions. It is therefore necessary to formalize it mathematically into an optimization problem.

3.1 Problem Formulation

Structural test objectives include branch coverage can be encoded as a search goal in the fitness function.

S a set of all possible solutions (test for the SUT),

$$S : \{S_1 \dots S_i \dots S_J\} \qquad \text{where } i, J \in \mathbb{N}$$

I set of Y consecutive SUT inputs that form a single solution (test case) S_i,

$$I : \{I_1 \dots I_y \dots I_Y\} \qquad \text{where } Y \in \mathbb{N} \text{ and } y \in [1 \dots Y]$$

B a set of all Z branches for a given SUT,

$$B : \{b_1 \dots b_z \dots b_Z\} \qquad \text{where } Z \in \mathbb{N} \text{ and } z \in [1 \dots Z]$$

a function $F : S \to B$ which associates for each single solution S_i the set of branches:

$$\{BS_1 \dots BS_i \dots BS_J\} \qquad \text{where } BS_i \subseteq B$$

that it has been covered by. The F function given as:

$$F(S_i) \;=\; BS_i$$

another function $G : S' \to B$ that takes a subset of solutions $S' \subset S$
returns a set of branches covered by this set $BN \subset B$, $S_i \in S' \Longrightarrow BS_i \subset BN$

$$G(S') \;=\; BN$$

A final function $O : S \times C \to B$ for maximum coverage of a given SUT branches:

$$O(S) \;=\; \begin{cases} S', & \text{if } \exists\, S' \subset S \text{ where } |G(S')| \;=\; |B| \\ Max(G(S')), & \text{otherwise} \end{cases}$$

3.2 Search Space Representation

A Chromosome is represented in two distinct but related forms: **genotype** and **phenotype**.

—The **genotype** encodes the sequence of consecutive inputs applied to the SUT.

—The **phenotype** corresponds to the set of program branches covered by executing these inputs.

Formally, let:

—$\{I_1, \ldots, I_y, \ldots, I_Y\}$ denote the sequence of inputs that make up the genotype of a solution S_i, where $Y \in \mathbb{N}$ and $y \in [1, Y]$.

—$\{BS_1, \ldots, BS_i, \ldots, BS_J\}$ denote the set of branches covered by S_i, which forms its phenotype, where $J \in \mathbb{N}$ and $i \in [1, J]$.

To store and manage candidate solutions during the search process, each solution is modeled using the following.

—a *list of inputs* representing the genotype,

—a *list of covered branches* representing the phenotype,

—a *fitness value* assigned via a fitness evaluation function.

This data structure enables an efficient representation of the input-coverage relationship, supports genetic operations, and facilitates fitness-based selection.

Figure 6 illustrates an example of a solution S_i consisting of 7 inputs, covering 8 branches, with an associated fitness score.

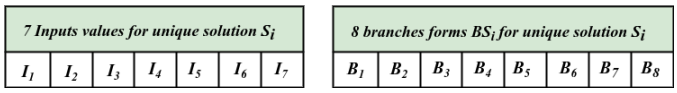

Fig. 6: Solutions Coding Example

3.3 Fitness Function

The fitness function evaluates how well a test input satisfies a given test objective. For example, branch distance and approach level are common metrics used to guide the search toward branch coverage [12].

Considering the function $T_{SUT}(x, s)$ which determines at the execution of the SUT if the target branch x is covered by the inputs of this solution, this function is given as follows:

$$T_{SUT}(x, s) = \begin{cases} 1, & \text{if } x \quad \text{is covered} \\ 0, & \text{otherwise} \end{cases}$$

Where $x \in B$, x belongs to the set of SUT branches and s is a solution to be evaluated. The fitness function used in this research to evaluate a solution individually is given by:

$$F(s) = \quad W_1 \cdot T_{SUT}(x, s) \quad + \quad W_2 \cdot \frac{|covered\ branches|}{|total\ branch|}$$

Where W_1 and W_2 represent respectively the weight given to exploration and exploitation, the constraint related to the weights for a good balance is given by: $W_1 + W_2 1$.
The following table (table **??**) represents the different ranges in solution exploration and exploitation.

3.4 Alterable Genetic Algorithm Design

The AGA aims to avoid the joint execution of its operators. For an iteration, it uses only one of the two, and thus alternates between them during the search, in order to find the optimum. AGA looks for the right time to execute the right operator through its alternation function.
AGA first initializes a set of possible solutions, (test case), evaluates each solution, running them as inputs to the SUT, and then, through the operation selection, selects according to its strategy the best solutions for crossover OR the mutation.

The alternation function uses statistical data on the set of solutions phenotypes (covered branches), to determine the operator that will be activated at this iteration (this generation). The crossover or mutation produces offspring until the satisfaction of the stoping criteria or the resources are exhausted. The structure of an AGA is given below in Algorithm

3.4.1 Chromosome Encoding. The test inputs are encoded as chromosomes. Depending on the type of SUT input, binary or real-valued encoding can be used.

Alternation Function

The alternation function is the main part of the AGA. It dynamically coordinates the most relevant operators of the algorithm. The mutation rates are often very low in most techniques [17], to avoid undesirable effects of genetic operators. However, AGA does not need to reduce the mutation rate strongly, as the alternation function allows to activate the right operator (Algorithm 1) at the right time to face these problems and reduce its effects.

Algorithm 1 Alternation Function

Input: $P, CurentOperator$
Output: *DesignatedOperator*
function Alternation Function (P, $CurentOperator$)
1: $DesignatedOperator \leftarrow Current\ Operator$
2: Compare the Phenotype of the population P;
3: Gets the Similarity Ratio S ;
4: **if** ($CurentOperator == Crossover$) **and** ($S \geq 65\%$) **then**
5: $DesignatedOperator \leftarrow Mutation$;
6: **else if** ($CurentOperator == Mutation$) **then**
7: $DesignatedOperator \leftarrow Crossover$;
8: **end if**
9: **return** $DesignatedOperator$;

Selection, Crossover, and Mutation

Roulette wheel selection, uniform single-point crossover, and the random reset mutation technique are used. These operators ensure diversity and convergence of the population.

Weight Ranges	Outcome
$W_1 = W_2$	Same balance between exploitation and exploration
$W_1 < W_2$	Favoring exploitation over exploration
$W_1 > W_2$	Favoring exploration over exploitation
$0 \leq W_1 \leq 0.3$	Low level of exploration implies high level of exploitation
$0.3 < W_1 \leq 0.7$	Medium level of exploration implies medium level of exploitation
$0.7 < W_1 \leq 1.0$	High level of exploration implies low level of exploitation

Table 1. : Exploitation and exploration ranges representation

Termination Criteria

The algorithm stop after a fixed number of generations or when full coverage is achieved. The coverage level allows us to obtain the progression of the branches coverage found by the following formula:

$$levelofcoverage = \frac{coveredbranches}{totalbranch} \times 100$$

Algorithm 2 Alterable Genetic Algorithm (AGA)

Input: SUT, Max_i, $Population_{size}$
Output: Test Suite
1: Generate random initial population P
2: $i \leftarrow 0$
3: $CurrentOperator \leftarrow$ Crossover
4: **repeat**
5: Evaluate fitness $f(x)$ for each $x \in P$
6: Select parent from P according to the selection method
7: AlternationFunction$(P, CurrentOperator)$
8: Perform designated operator
9: Construct new population P'
10: $P \leftarrow P'$
11: $i \leftarrow i + 1$
12: **until** All branches covered **or** $i \geq Max_i$

Mathematical Model: The AGAs Schema Theory

From GAs previous schema theory 6. AGA's schema theory used for test data generation could be deduced. Knowing AGA unlike traditional GA executes only one genetic operator at each iteration, this performed by the Alternation Function. Therefore, if the crossover operator is active then logically the mutation would be deactivated, then the probability of mutation would be zero ($Pm = 0$). In the opposite case it is the probability of the crossover be zero ($Pc = 0$). Thus, the mathematical model representing the schema of an AGA would be as follows:

$$N(c,i+1) \geq \begin{cases} N(c,i)\ \frac{\tilde{f}(c,G(i))}{\frac{1}{L}\sum f(x,G(i))}(1 - Pc\frac{P(c)}{l-1}), & \text{if i+1 activate crossover} \\ N(c,i)\ \frac{\tilde{f}(c,G(i))}{\frac{1}{L}\sum f(x,G(i))}(1 - Pm.P(c)), & \text{if i+1 activate mutation} \end{cases} \quad (7)$$

4. EXPERIMENTS & RESULTS

4.1 Setup

An empirical study were conducted on on five (5) SUTs that were proposed by third parties to avoid bias. The results are shown in the figure below (Figure: 7). The proposed AGA result is compared to the traditional GA in terms of branch coverage.
The parameters are defined, in order to carry out this study fairly and without bias, these parameters as described in the table below (table 2), presents according to the used techniques the size, the scope of experimentation and also the different types of operator used.
These test subjects were selected after a proposal of a group of third persons, they are python implementations of the following algorithms: Euclid, Find, Triangle ISBN10 checker and ISBN13 checker.

4.2 Results

It is important to note that, the approaches used in this study were executed 15 times each, and for each test subject. The results presented in the table below (table **??**), are the collected average of these 15 repeated executions.

The table **??** contains a first column, that indicates the SUT, it also contains two (2) columns (AGA's level of coverage and GA's level of coverage) that represent respectively the average coverage level reached by the AGA and GA approach. Two (2) others columns (AGA's number of iteration and GA's number of iteration), also representing respectively the average number of iterations exhausted by the AGA and GA to reach their average coverage level.

The empirical study addresses these two research questions presented in the prvious section:

RQ1 : How effective is the generation of test data based on an alterable Genetic Algorithm for branch coverage?

The primary objective of this research is to address the problem of test data generation. Although the proposed approach may not guarantee an absolute optimal solution in all scenarios, but a near-optimal solution, it aims to provide effective and practical results in most cases.

RQ2 : What is the efficiency of structural test data generation based on Alterable Genetic Algorithm as opposed to the traditional Genetic Algorithm according to branch coverage criteria for real-world programs?

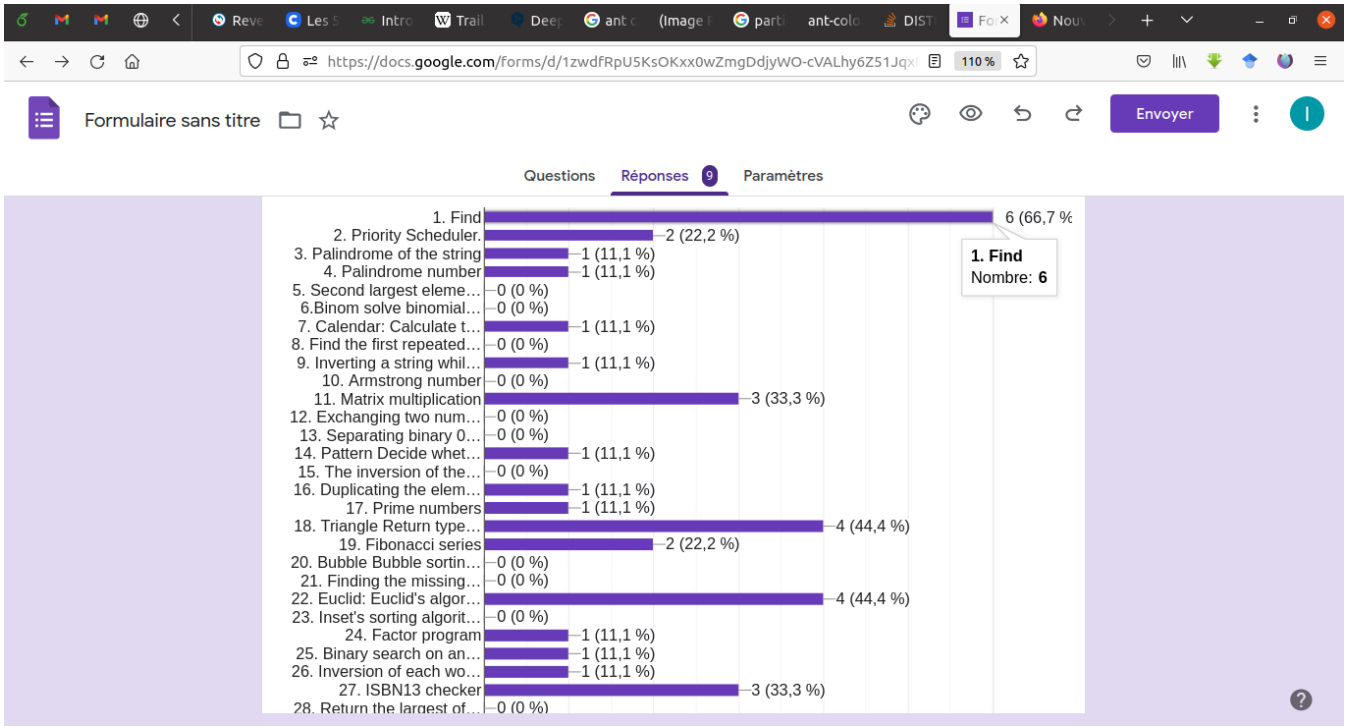

Fig. 7: SUTs solicitation by tiers

Parameter	Setting	Method	Comment
Population size	50	GA, AGA	
Number of generations	1200	GA, AGA	Maximum number of iterations (Max_i)
Selection operator	Roulette wheel	GA, AGA	Fitter individuals have a higher selection probability
Mutation operator	Random reset	GA, AGA	Genes are randomly modified
Mutation rate	0.2	GA	Frequency of gene mutation
Crossover operator	Uniform single-point	GA, AGA	Recombination from a single crossover point
Mutation rate	Dynamic	AGA	Frequency of single-gene mutation
Initial population	Random	GA, AGA	Random initialization of the first generation

Table 2. : Parameter settings for the empirical study

SUT	AGA coverage	AGA iterations	GA coverage	GA iterations
Euclid	100%	8.4	100%	8.533
Find	100%	9	100%	9
Triangle	100%	25.867	100%	27.867
ISBN10 Checker	100%	133.2	98.461%	328.133
ISBN13 Checker	95.833%	1200	94.444%	1200

Table 3. : Results based on achieved branch coverage and number of iterations

To validate the hypothesis, that the AGA outperforms the GA, where the latter (AGA) is truly an improvement. The AGA is compared to the traditional Genetic Algorithm, with respect to the chosen branch coverage criteria. So the efficiency and the improvement that the AGA brings, is determined through the number of branches of the test subjects, that the technique will be able to cover. This search question allows us in this axis to determine what is the efficiency of the AGA compared to the GA for automatic test data generation?

Answer to RQ1

The proposed AGA provides an effective and practical solution to the test data generation problem, often yielding near-optimal out-

comes. As evidenced in Table **??**, the approach achieves full coverage in four (4) out of five (5) systems under test (SUTs).
In the **Euclid** case, the AGA achieved 100% branch coverage with an average of 8.4 generations. For the **Find** program, complete coverage was reached after approximately 9 iterations. In the **Triangle checker** scenario, all branches were successfully explored within an average of 25.867 generations. Finally, in the **ISBN10 Checker** case, the algorithm attained full branch coverage in 133.2 iterations on average, demonstrating consistent effectiveness across diverse systems under test (SUTs).
In the case of the **ISBN13 Checker**, the AGA achieved a near-optimal solution, reaching 95.833% branch coverage. This slight reduction in coverage, compared to other SUTs, may be attributed to the increased structural complexity of the ISBN13 validation logic and the more intricate conditions governing access to bibliographic information based on the ISBN identifier.

Answer to RQ2

To study AGA contribution in front of the GA, Investigate the efficiency of the AGA's alternation function on real world programs. The results are summarized for each SUT in the 2 figures (figure 8 and 9) below:
Figures 9 illustrates the resources exhausted by the two methods AGA and GA, for each SUT. The resources used up are translated by the number of generation.
The number of iterations and the coverage of the branches obtained are illustrated in the figures in the form of graphs, typical of other experiments carried out in the field. The average branch coverage and the corresponding number of iterations obtained by the AGA is designated by the blue bars, as indicated in the legends, on the other hand, that of the GA is designated by the orange bars.
The different SUTs of the experiment are placed on the x-axis and the average coverage of the branches obtained by the methods, is displayed on the y-axis in figure 8. In figure 9 the SUTs are placed on the y-axis and the average iteration numbers of the methods are placed on the x-axis.

The results showed the contrasts between the AGA and the GA, for the test data generation on the same SUTs. It can be seen from the table**??** that the AGA achieved complete coverage (100%) in 4 of the 5 scenarios, while the GA scored 3 out of 5. Let's have a closer look at the results for each test subject through figure 8 and 9.
The experimental comparison across five case studies demonstrates the consistent efficiency of the AGA over the traditional GA. In simpler programs such as Euclid, Find, and Triangle, both methods achieved 100% branch coverage, with AGA requiring slightly fewer iterations, indicating a modest yet consistent improvement in convergence speed. In the more structurally complex ISBN10 Checker, AGA successfully achieved full coverage, while GA stagnated at 98.461%, highlighting GA's vulnerability to local optima. Finally, in the ISBN13 Checker case, where neither method attained full coverage, AGA still outperformed GA with a higher average coverage (95.833% vs. 94.444%), reinforcing its robustness and superior adaptability in challenging scenarios.
This section aims to calculate the time complexity, of the traditional GA and compare it with the one of the proposed approach. The time complexity of meta-heuristic algorithms in general depends on the problem to be solved, from which the fitness function is defined and the stopping condition is determined. In general the AGA and traditional GA have a time complexity in the best case $O(c)$, where c is a constant, because it is quite possible, that the search can converge to the global optimum solution from the first iteration and in the worst case, their complexity will be $O(\infty)$, because it is also obvious, there is a possibility that the two algorithms can not find the expected optimal solution and therefore they enter a kind of infinite loop.
On the other hand, in this project the second scenario has been solved through the definition of a maximum number of iterations, as Max_i, which marks a determining constraint, when the algorithm does not find the optimal solution until this maximum number of iterations, this one is forced to stop and return the best solution, that it has been able to find until then.

THE COMPLEXITY OF TRADITIONAL GA

The complexity of the GA. with a defined maximum number of iterations is given as follows, but first, let us recall the main steps that compose the algorithm:

(1) generation of the initial population with size defined as P,
(2) the fitness function to evaluate the aptitude of all individuals (solution) in the population,
(3) select the individuals according to the roulette wheel method, for the next generation,
(4) according to the crossover probability, the selected individuals are crossed and then according to the mutation probability the newly generated individuals are mutated to generate a new population,
(5) evaluate the fitness of the new members, reinsert the new individuals and select the next population,
(6) steps $2, 3, 4$ and 5 are repeated until the algorithm reaches the maximum number of iterations or it has found the expected solution, that is, its fitness value reaches the norm set.

For the first step **(1)** of the GA the operation consists in generating randomly P individuals of size Y, giving a complexity $O(P \cdot Y)$ for this step. From the second step **(2)** the algorithm enters a loop conditioned by a maximum number of iterations Max_i or convergence to the expected optimal solution. An iteration that executes the second step consisting in evaluating one by one the P individuals of the population by executing each one with the SUT. This gives a complexity that strongly depends, on the complexity of the program chosen to be the SUT. Let us consider the complexity of the SUT as C_{SUT}. Seeing that it is executed for each individual of the population it will thus cost $O(P \cdot C_{SUT})$ for one iteration of this step. At the third step **(3)** the operation selects among the P individuals and according to a probability proportional to their abilities. Thus the complexity of the roulette selection method is $O(P \cdot logP)$ given mainly by the cost of sorting. Step four **(4)** or, the joint operation of crossing and mutation is performed according to their respective probability. Let Pc be the crossover probability and Pm the mutation probability, the complexity of this step is defined as $O(Max(Pc \cdot P, Pm \cdot P))$ which is equal to $O(P \cdot Pc)$ because $Pc > Pm$. At the next step **(5)** the operation will consist in reevaluating the new individuals created by the crossing and the mutation by executing the SUT, then adding them to the population in order to select the new population or the next generation, this step has a complexity of $O(Max(Pc \cdot P \cdot C_{SUT}, Pm \cdot P \cdot C_{SUT}))$ which also gives us $O(P \cdot Pc \cdot C_{SUT})$.

The steps **2,3,4** and **5** are thus iterated Max_i times in the worst case, then, the complexity of all the steps iterated in the loop will be multiplied by Max_i Thus, the complexity of the

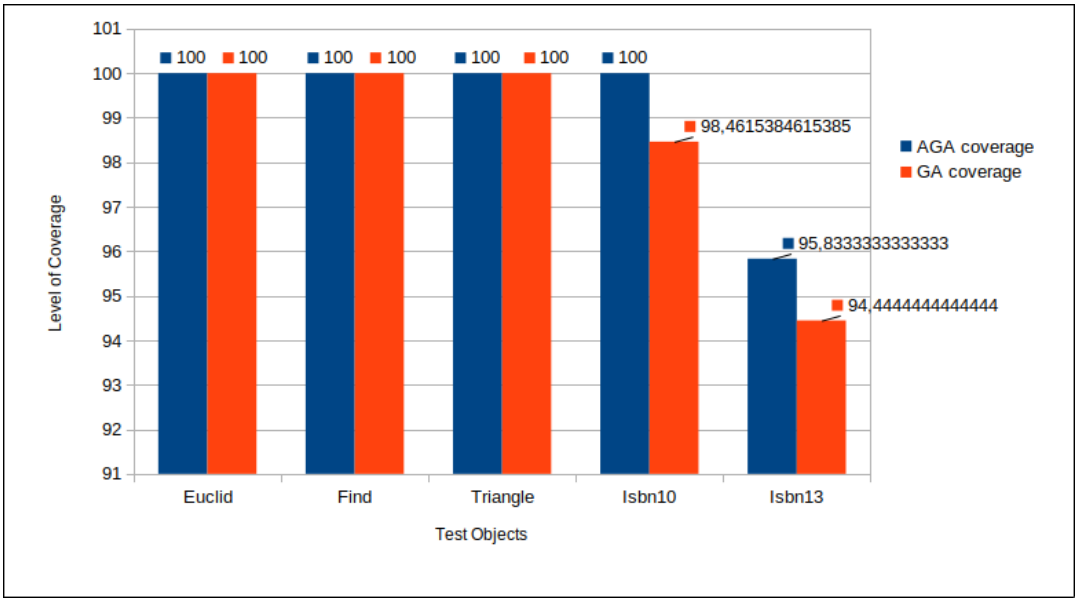

Fig. 8: Level of Coverage AGA vs GA

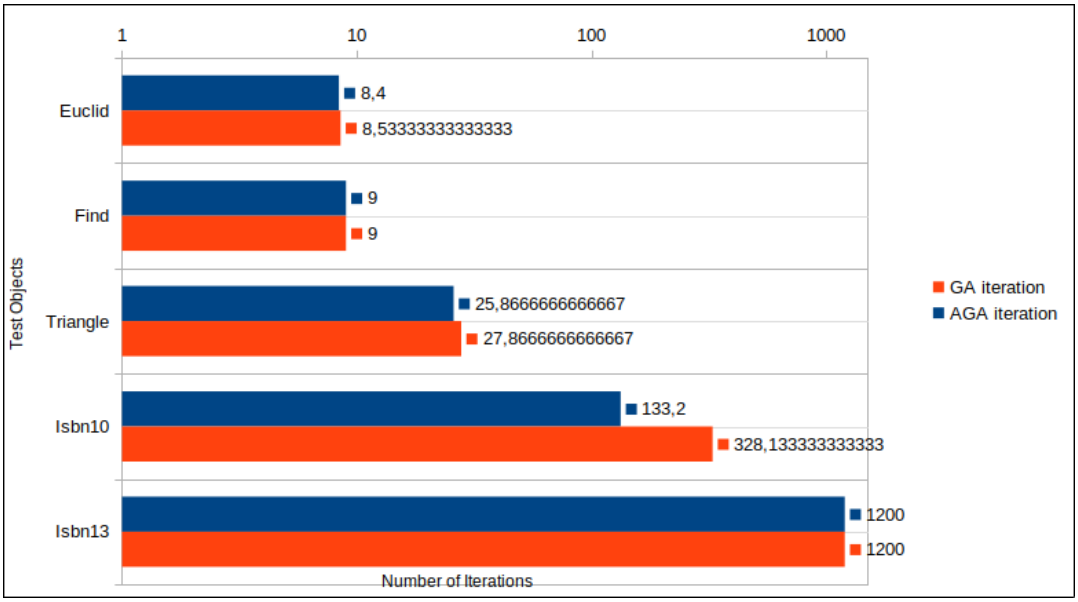

Fig. 9: Iterations exhausted by AGA vs GA

set forming the instruction block of the GA steps is collected as follows:

—**Step 1** = $O(P \cdot Y)$
—**Step 2** = $O(P \cdot C_{SUT} \cdot Max_i)$
—**Step 3** = $O(P \cdot logP \cdot Max_i)$
—**Step 4** = $O(P \cdot Pc \cdot Max_i)$
—**Step 5** = $O(P \cdot Pc \cdot C_{SUT} \cdot Max_i)$

Let us calculate the complexity of the set by performing the maximum operation on the complexity of the instructions as follows:

$$O(Max(P \cdot Y, \quad P \cdot C_{SUT} \cdot Max_i, \quad P \cdot logP \cdot Max_i, \quad P \cdot Pc \cdot Max_i, \quad P \cdot Pc \cdot C_{SUT} \cdot Max_i))$$

Given that the maximum of these five (5) parameters: $P \cdot Y$, $P \cdot C_{SUT} \cdot Max_i$, $P \cdot logP \cdot Max_i$, $P \cdot Pc \cdot Max_i$, and $P \cdot Pc \cdot C_{SUT} \cdot Max_i$ considered is $P \cdot C_{SUT} \cdot Max_i$ then the time complexity of the traditional Genetic Algorithm for the structural test data generation of a defined SUT would be:

$$O(P \cdot C_{SUT} \cdot Max_i)$$

THE COMPLEXITY OF AGA

The proposed algorithm includes all the steps mentioned above, with one difference : the execution of the crossover and mutation operators is not joint but alternated, thanks to the addition of the alternation function constituting a new step. It can be stated that the AGA has one more step than the GA. .
The step of the alternation function is in charge of determining the unique operator to be executed, between the crossover and the mutation during an iteration. This one consists in comparing each covered branch or phenotype of a solution, with the set of branches covered by the $P-1$ other solutions of the current population. The comparison of the set of branches gives a percentage of similarity determining for the choice of the operator, and will thus cost, in term of temporal complexity $O(P^2)$ or $O(P \cdot P)$. It is important to note also that this step is also iterated Max_i times in the algorithm. Thus the alternation function step costs $O(P \cdot P \cdot Max_i)$ and therefore the complexity of the AGA is calculated as follows Knowing that $O(P \cdot C_{SUT} \cdot Max_i)$ is the complexity of the rest of the steps given in the section above are presented as follow:

$$O(Max(P \cdot P \cdot Max_i,\ P \cdot C_{SUT} \cdot Max_i))$$

At this level it is not obvious to determine the maximum of these two parameters: $P \cdot P \cdot Max_i$ and $P \cdot C_{SUT} \cdot Max_i$ because the SUT can vary and its complexity varies too. But the worst case would be a maximum C_{SUT} complexity, where the C_{SUT} would be higher than P ($C_{SUT} > P$). then the maximum is $P \cdot C_{SUT} \cdot Max_i$ and finally the complexity of the AGA for test data generation to a given SUT, would be :

$$O(P \cdot C_{SUT} \cdot Max_i)$$

Compared to the traditional GA, for each iteration, the proposed algorithm has one more step that costs $O(P \cdot P)$, but in the worst case they have similar complexity.

DISCUSSION

The experiments showed an improvement of the solution AGA over the GA for the test scenarios as SUTs. On the other hand the results validated the expected ability of the AGA with its alternation function to guide the solutions, over the premature convergence and stagnation problems described by the Schema Theory [12] and reduce the effect of perturbation, of the genetic operators which resulted in the AGA outperforming the GA in almost all the scenarios of this experimentation.
However, challenges remain in handling complex input constraints and Design AGA's fitness function in order to achieve path coverage.

Comparison of AGA with various recent approaches

After implementing the proposed approach, a comparative analysis was carried out with other recent state-of-the-art techniques.
The table **??** clearly shows that, AGA approach is superior in terms of applicability, scalability and computational cost required but the MAAT approach based on the genetic algorithm and reinforcement learning [5] is the most efficient with complete coverage. However, the test criteria used in experiments for this approach are path coverage and some test subjects do not accurately represent real-world programs.

5. CONCLUSION AND FUTURE WORK

This paper made some contributions in the field of SBST, a solution applied to test data generation named Alterable Genetic Algorithm (AGA). AGA is a technique from the family of population-based evolutionary algorithms such as Genetic Algorithm.
The addition at the AGA level, is an alternation function that rhythms the execution of the genetic operators and especially alternates between the two for each iteration. This function separates them (mutation and crossover) and allows to activate the right operator, at the right time of the search. A mathematical model representing the AGA Schema deduced from the one generalized for the SBST. Deduced model has theoretically demonstrated the improvement that AGA has brought over GA, in terms of undesirable effects related to crossover or mutation
An empirical study was conducted to evaluate the performance of the Alterable Genetic Algorithm (AGA) in comparison to the conventional Genetic Algorithm (GA), grounded in the theoretical foundations of Schema Theory. The results empirically confirmed that AGA consistently outperformed the traditional GA across most scenarios, highlighting its efficiency and improved search capability.
Future directions include a deeper analysis of the alternation function to optimize the trade-off between computational cost and algorithmic sophistication, as well as extending the approach to accommodate complex data structures and support path coverage criteria.

6. REFERENCES

[1] Sultan H Aljahdali, Ahmed S Ghiduk, and Mohammed El-Telbany. The limitations of genetic algorithms in software testing. In *ACS/IEEE International Conference on Computer Systems and Applications-AICCSA 2010*, pages 1–7. IEEE, 2010.

[2] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 351–366. Springer, 2008.

[3] Rajiv Chopra. *Software testing: a self-teaching introduction*. Mercury Learning and Information, 2018.

[4] Philippe Collard and Alessio Gaspar. " royal-road" landscapes for a dual genetic algorithm'. In *ECAI*, volume 96, page 12th. PITMAN, 1996.

[5] Mehdi Esnaashari and Amir Hossein Damia. Automation of software test data generation using genetic algorithm and reinforcement learning. *Expert Systems with Applications*, 183:115446, 2021.

[6] Marie-Claude Gaudel. Formal methods for software testing. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–3. IEEE, 2017.

[7] Robert Gold. Control flow graphs and code coverage. *International Journal of Applied Mathematics and Computer Science*, 20(4):739–749, 2010.

[8] Shifen Han and Li Xiao. An improved adaptive genetic algorithm. In *SHS Web of Conferences*, volume 140, page 01044. EDP Sciences, 2022.

Approach	Efficiency	Applicability	Scalability	Computational Cost
AGA	– highest average was a complete coverage – lowest average was 95.833%	High	Yes	Low
GA for Dynamic Programming language RuTeG [17]	– highest average was a complete coverage – lowest average was 88%	Medium	Yes	Low
Improved adaptive GA [8]	– Unknown	Low	Yes	Medium
MAAT: GA & Reinforcement learning [5]	– the average coverage was 100% (complete coverage)	Low	No	High

Table 4. : Comparison of AGA with state-of-the-art Approaches

[9] Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE'07)*, pages 342–357. IEEE, 2007.

[10] Mark Harman. Open problems in testability transformation. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 196–209. IEEE, 2008.

[11] Mark Harman and Afshin Mansouri. Search based software engineering: Introduction to the special issue of the ieee transactions on software engineering. *IEEE Transactions on Software Engineering*, 36(6):737, 2010.

[12] Mark Harman and Phil McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 73–83, 2007.

[13] Frederick Hayes-Roth. Review of" adaptation in natural and artificial systems by john h. holland", the u. of michigan press, 1975. *ACM SIGART Bulletin*, (53):15–15, 1975.

[14] Weigang He, Jianqi Shi, Ting Su, Zeyu Lu, Li Hao, and Yanhong Huang. Automated test generation for iec 61131-3 st programs via dynamic symbolic execution. *Science of Computer Programming*, 206:102608, 2021.

[15] René Just. On effective and efficient mutation analysis for unit and integration testing. 2016.

[16] Susan Khor and Peter Grogono. Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 346–349. IEEE, 2004.

[17] Stefan Mairhofer, Robert Feldt, and Richard Torkar. Search-based software testing and test data generation for a dynamic programming language. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1859–1866, 2011.

[18] Gerrit Jan Tretmans and Axel Belinfante. *Automatic testing with formal methods*. Centre for Telematics and Information Technology, University of Twente, 1999.

[19] Sapna Varshney and Monica Mehrotra. Search based software test data generation for structural testing: a perspective. *ACM SIGSOFT Software Engineering Notes*, 38(4):1–6, 2013.

[20] Richard A Watson and Thomas Jansen. A building-block royal road where crossover is provably essential. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1452–1459, 2007.

[21] Dalin Zhang, Jianwei Sui, and Yunzhan Gong. Large scale software test data generation based on collective constraint and weighted combination method. *Technical Gazette/Tehnički Vjesnik*, 24(4), 2017.