EVOTLBO: A TLBO based Method for Automatic Test Data Generation in EvoSuite

Mohammad Mehdi Dejam Shahabi
Software Engineering Lab., Department of Computer Engineering and Information Technology
Shiraz University of Technology
Shiraz, Iran

S. Ehsan Beheshtian
Software Engineering Lab., Department of Computer Engineering and Information Technology
Shiraz University of Technology
Shiraz, Iran

S. Parsa Badiei
Software Engineering Lab., Department of Computer Engineering and Information Technology
Shiraz University of Technology
Shiraz, Iran

Reza Akbari
Software Engineering Lab., Department of Computer Engineering and Information Technology
Shiraz University of Technology
Shiraz, Iran

S. Mohammad Reza Moosavi
Department of Computer Science, Engineering and Information Technology
Shiraz University
Shiraz, Iran

***Abstract*—Now-a-days software has a great impact on different aspects of human life. Software systems are responsible for safety of major critical tasks. To prevent catastrophic malfunctions, promising quality testing techniques should be used during software development. Software testing is an effective technique to catch defects, but it significantly increases the development cost. Therefore, automated testing is a major issue in software engineering. Search-Based Software Testing (SBST), specifically genetic algorithm, is the most popular technique in automated testing for achieving appropriate degree of software quality. In this paper TLBO, a swarm intelligence technique, is proposed for automatic test data generation as well as for evaluation of test results. The algorithm is implemented in EvoSuite, which is a reference tool for search-based software testing. Empirical studies have been carried out on the SF110 dataset which contains 110 java projects from the online code repository SourceForge and the results show that the TLBO provides competitive results in comparison with major genetic based methods.**

Keywords— EvoSuite; TLBO; test data generation

I. Introduction

In order to reduce software testing cost, automated test generation methods are used. These methods could be categorized into three classes based on the test data generation method used: random search algorithms, dynamic symbolic execution, and evolutionary optimization algorithms.

Dynamic Symbolic Execution (DSE) is the interpretation of programs using symbolic values for input arguments to explore code paths. A path is distinguished by logical conditions on the input values. A model for the condition is defined by a program input that follows the path described by the condition [1]. The drawback is path explosion which means that the number of feasible paths grows exponentially with an increase in program size.

Evolutionary algorithms are used to formulate the testing problem as an optimization problem. Search algorithms are used to find answers based on a cost function. These evolutionary algorithms, such as genetic and simulated annealing, try to find the best test suite that maximizes the coverage in the software under test. The commonly used evolutionary algorithm in the literature is the GA and its extensions (i.e., 73% of related papers). The mentioned reason is just the popularity of GA and its applications in various problems and fields [2]. There is no evidence to prove GA superiority in performance.

In our research, we applied other meta-heuristic algorithms and the proposed TLBO method is based on swarm intelligence for the evolutionary purpose of test data generation. Moreover according to the surveys on type of testing in software engineering, almost 75% of the researches done in this field discuss results on structural testing [2]. Despite what the majority of papers discuss, object oriented testing is used in this paper to evaluate the performance of our method. This is due to the recent trend in object oriented design, programming and object oriented testing in software engineering in the recent years.

Search-based techniques are appropriate for the automated generation of unit tests. There are search-based tools like AUSTIN for C programs [3] or EvoSuite for Java programs [4]. EvoSuite is a promising tool for automatic software testing that optimizes whole test suites towards satisfying a coverage criterion [5]. A coverage criterion represents a finite set of coverage goals (described in Section II-B).

The TLBO algorithm is implemented based on EvoSuite tool. The performance of the TLBO algorithm on the SF100 corpus of open source classes shows enhanced coverage in 4 coverage criterions in the generated test data.

The rest of the paper is organized as: In Section II, basic concepts have been described. Section III provides related works and the background for the proposed method. The TLBO algorithm is proposed in Section IV. In Section V the empirical studies for the proposed method is presented and finally in Section VI the paper is concluded and some ideas have been suggested to inspire future researches.

II. Basic Concepts

A. Test Data Generation

The objective of test data generation is to have a test suite that maximizes a coverage criterion [6]. A test suite contains a set of test cases each of which specifies the inputs, a sequence of statements and execution conditions to test different behaviors of the code under test, and the predicted results. Finding test input data is a challenging task. Constraint based techniques and search based methods are two promising methods in test data generation. Constraint based techniques use static and dynamic symbolic execution methods to generate appropriate input for test cases. The disadvantages of constraint based techniques include low scalability, inability to manage the dynamic aspects of a unit under test, and the type of constraints they can handle.

On the other hand, using search algorithms, an optimization problem is solved to generate test cases and suitable input for them. Search based methods can handle a variety of domains and are very scalable. However these methods get stuck in local optima and degrade when the search landscape offers insufficient guidance. Our approach for automatically generating test input data is a search based evolutionary algorithm, guided by a fitness function.

B. Coverage Criteria

Coverage criterions determine the goals to be covered for the search algorithm. Each test suite is optimized for performance in a certain criterion. There are many criterions in software testing (e.g., line, mutation, and exception). Based on the previous works in unit testing, four criterions have been used in this paper, namely: line coverage, branch coverage, method coverage and output coverage [7]. Line coverage presents the executed lines in the code. Branch coverage [8] is the number of branches covered by the test, like branches of conditional statements. Method coverage represents the methods invoked by the test case and Output coverage is a complementary coverage to the method coverage as it checks the output of the methods and tries to capture different outputs by changing the corresponding input [7].

C. Fitness Function

In search based software testing a fitness function determines how good a test suite is regarding the optimization objective, which is usually defined by a certain coverage criterion. In addition to checking whether a coverage goal is achieved, a fitness function also provides additional information to guide the search toward covering it.

Method coverage is among basic coverage criteria. This criterion requires the test suite to invoke every method in the class under test at least once. This can be done by direct calls in test cases which appears as a statement or by indirect calls. For regression test suites, it is important that each method is also invoked directly. For a set of goals in a particular coverage criterion, X, the search algorithm generates a test suite that maximizes the number of the covered goals. Fitness functions calculate a fitness value to guide the search toward a goal. Usually the approach level A and branch distance d are employed for this purpose.

The approach level *A(t,x)* for a given test *t* on a coverage goal $x \in X$ is defined as the minimal number of control dependent edges in the control dependency graph between the target goal and the control flow that is represented by the test case. The branch distance *d(t,x)* means how far a predicate in a branch *x* is from being evaluated as true [9].

In branch coverage criterion, the fitness function to minimize the approach level and branch distance between a test *t* and a branch coverage goal *x* is defined as:

$$f\,(t,x) = A(t,x) + \nu(d(t,x)) \qquad (1)$$

Where, ν is any normalizing function in the range (0, 1) [10].

Another basic coverage criterion is line coverage which will satisfy by executing all the lines in the class under test [7]. For this purpose, a fitness function for the line coverage criterion uses branch distance to estimate how far a predicate is from evaluating to the expected outcome. For example, given a predicate *x==1*0 and an execution with value 5, the branch distance for the expected outcome being true would be |10-5|=5. Branch distances can be calculated by applying a set of standard rules [8], [11]. To optimizing a test suite (rather than a single test case) toward satisfying line coverage criterion, the fitness function needs to calculate the branch distance for all branches. The line coverage fitness value of a test suite can be calculated by executing all test cases, and for each executed statement calculating the minimum branch distance among all of the branches that are control dependencies to that statement. Hence, the line coverage fitness function is defined as:

$$Fitness_{line}(test\ suite) = \nu(\,|total\ lines - covered\ lines|) + \sum_{b\in B} \nu(d_{\min}(b, suite)) \qquad (2)$$

Where, ν is any normalizing function, $d_{min}(b,suite)$ is the minimum distance and *B* is the set of control dependent branches.

For some methods, method coverage, line and branch have similar fitness values. In this case, unit tests are written to cover not only the input values of methods but also the output (returned) values. This criterion can help to improve fault detection capability [12]. To determine output criterion coverage goals, the following function maps methods' return type to abstract values.

$$O(type) = \begin{cases} \{true, flase\} & if\ type\ \equiv boolean \\ \{-,0,+\} & if\ type\ \equiv number \\ \{alphabet, digit,*\} & if\ type \equiv char \\ \{null\ , \neq null\} & other\ wise \end{cases} \quad (3)$$

To satisfy this criterion for each abstract value *V∈O(type)*, a test suite should contain at least one test case which when executed calls a method that returns a value that is characterized by V.

D. Problem representation

Evolutionary algorithms, employing global search methods, are used for optimization of test data generation problem. The representation used in our proposed method is the same problem representation used in EvoSuite. Test suites and test cases are both formulated as chromosomes containing genes. A test suite chromosome consists of test cases that test a class in a specific criterion. A test case respectively includes statements that cover a goal or set of goals in that criterion. Statements are categorized into five groups: method calls; primitive statements that declare a variable; constructor statements that create classes; field statements that access public members of a class and assignment statements which assign a value to a variable.

E. Mutation

Mutation is the occasional random alteration of a gene in a chromosome which alters some features with unpredictable effect on coverage. In a test-suite level, mutation is done by randomly generating test cases and adding them to the set. This random generation is similar to the initial population generation in an evolutionary algorithm. In test-case level, mutation is done by adding or removing or changing the statements in the test case [13], [14].

For method call statements this is done by adding extra method calls or removing the existing ones. The change is completed by calling a method with a different value for its arguments. For constructor statements either a different object is created or another constructor of the class is used or the input value for the constructor is changed. For primitive statements mutation can be done by changing the type of the variable or declaring new ones. Mutation on field statements can be done by accessing a different member of the class with the same or different type. In mutating an assignment statement, the assigned value can be changed.

III. Related Works

Automatic test data generation has been proposed to both increase the precision of software testing and decrease the cost of software testing. Various tools are available based on the proposed methods. In a survey presented by Ali, *et al*. 450 articles have been reviewed and almost 75% of them have carried out their research on unit testing [2]. They mentioned that 73% of the papers used genetic algorithms and 14% of the papers used simulated annealing algorithms. Although genetic algorithms perform better than local search algorithms, but there is no evidence to show that they perform better than global search algorithms. On top of all the reasons mentioned, there are lots of ready tools that adopt GA and are easily accessible for everyone.

In another survey by Harman, *et al*. the history of test data generation and automatic test data generation using evolutionary algorithms has been reviewed [15]. Harman have some recommendations in the paper: using search algorithms on generating test data for testing non-functional features in a software; using search algorithms on establishing the test strategy; using multiple-goal algorithms on generating test data to optimize multiple features in a software.

The literature review of automatic test data generation can be categorized under three subsections of random test data generation, dynamic symbolic execution and search based software testing. However we focus on the search based software testing. One of the major issues of test data generation is the generation of the initial population. The initial population has an influence on both the final solution and the number of generations [16]. In the paper presented by Pachauri and Srivastava [17], three methods were introduced to sort branches to be chosen as goals for coverage.

The work presented by Fraser and Arcuri [5], shows that the whole test suite approach achieves up to 18 times the coverage than the traditional approach which would target coverage goals individually. This method also generates test suites that are up to 44% smaller due to the prevention of the search redundancy and overlapped coverage of goals. In traditional methods for selecting one goal at a time, it is assumed that all the importance of goals is equal and the goals are independent. In contrary the whole test suite generation method targets a coverage criterion rather than a coverage goal. This solves several issues including the collateral coverage problem (i.e., the accidental coverage of the remaining targets [18]), and the effect of selecting goals in a specific order is inevitable in the traditional method.

In the work done by Suresh and Rath [19], a method was proposed to extract basic paths from Control Flow Graph (CFG) by genetic algorithms. In this method after identifying the basic paths, test data is generated to cover them. In the work presented by Bueno, *et al.* [20], a new method was proposed to generate the test cases as different from each other as possible. In this method a cost function that determines the difference between test cases tries to maximize this difference. In addition to solving this function with their own proposed algorithm, it has also solved with genetic and simulated annealing algorithms and the results have been compared.

In another work by Hermadi, *et al.* [21], a new stopping condition has been introduced. This method stops the search if there are paths in the software and there is no test case that can reach them. These conditions have been tested in 20 software data sets and the results are compared with other stopping conditions. In the work done by Pachauri, *et al.* [22], a parallel algorithm has been proposed based on master-slave model and genetic algorithm to generate test data. In this method master selects a path for each slave based on "Path prefix" strategy. Slaves then generate test data to cover that path using genetic algorithm. The results on two software show high precision in the generated data. It is noticeable that this method uses distributed techniques to generate test data.

Another paper on optimizing meta-heuristic algorithms has been presented by YueMing, *et al.* [23]. In this method that is based on particle swarm algorithms, the particles are divided into two groups, each having its own search method. This method has shown better performance both in execution time and in the quality of generated test cases. In the proposed method by Hoseini, *et al.* [24], the sequence diagram has been used as the input instead of the control flow graph. This method identifies basic paths in the software and generates test data to cover them using genetic algorithm. One key feature of this method is that is carries out the test before the development phase.

Reference [25] has used genetic algorithm to generate a sequence of method and constructor invocations of a class to test it. Then using a multiple-goal approach optimizes the length and the number of instructions in the test cases. Another idea in this article is to use previously generated test data as the initial population for the genetic algorithm to optimize them further. Results show a better performance than the manual method and some of other automatic methods. Change analysis test is technique that puts bugs in a software deliberately to realize if the generated test cases can detect it. If not, existing test cases should be modified or further test cases are required.

Zeller [26] have proposed a method to generate test data for detecting changes in object oriented classes. In this method test data are optimized for finding the most bugs rather than having the most coverage. In this work 'NTEST' has been introduced as way of generating test data for change analysis test, based on object oriented programs. Using change analysis test rather than structural testing, not only the place in code that needs testing is acquired but also what should be tested there is specified.

To combine the two methods of test data generation, search based algorithms and constrained based algorithms, a hybrid solution is proposed by Fraser [27] that works based on genetic algorithm. The algorithm evolves a set of answers chosen by the fitness function toward gaining the most coverage. To speed up the algorithm and avoid the search being confined to local optimizations, a mutation operator was introduced to be added to the GA. What this mutation does is the dynamic execution based on limitations. Instead of random alternations in the chromosomes genes (bytes) or blindly changing the input for methods in the generated test cases, the mutation is done based on the execution path's properties of the chromosome. By doing this a new path is formed in the search space and as a result increases the coverage. Results show a 28% improvement compared to search based methods and a 15% improvement to the limitation based methods. In the work done by Koleejan, *et al.* [28], a method is presented based on genetic and particle swarm algorithms. The main goal of this paper is to optimize the performance of the previous methods by generating multiple test cases in every iteration. Results show that the implemented algorithms perform better than the previous methods.

Arcuri and Fraser have shown the challenges of applying EvoSuite to randomly selected open source projects from SourceForge [29]. This research is of importance because many similar tools are tested with just a few hand selected cases and as a result they are optimized for those specific classes and are not to be generalized. Working with automated search based software testing tools in a real and industrial level project is the ultimate goal of software testing, which is achieved by EvoSuite, however there are challenges that require the testers' attention. The everlasting problems like seeding, tuning and bloat control have been fairly addressed in EvoSuite due to its years of development and surprised encounters with unexpected behaviors the developers had to deal with. Moreover for an industrial scale software regression testing is vital. This is achieved by generating test cases with assertions which capture the current behavior of the software. In addition to that test cases need to be readable by users, because no matter how good a test case is in finding failures, a user needs to check the test cases to ensure that failures found are caused by real faults and not because of the violation of a precondition and also to check the assertions to make sure that the captured behavior is correct. This readability is achieved by several methods. For example In case of variables with large values, EvoSuite tries to make them smaller using a binary method. Moreover naming the variables with proper understanding names or dedicating individual lines to them are also deliberated to make the generated test case as clear and readable as possible. To make analyzing the data easier, test data coverage results are in the form of CSV (comma separated values) files. Every column represents a coverage criterion and every row represents a class in the project.

In recent years many successful applications of swarm intelligence based methods have been reported by researchers. It seems that these methods have the potential to be applied in a broad range of software engineering problems such as software testing. Based on our knowledge there are a few swarm intelligence based methods applied for test data generation in EvoSuite. Hence, this work is aimed to design a swarm intelligence based method for automatic test data generation in EvoSuite.

IV. The Proposed Method

In this section, the proposed EvoTLBO algorithm is described in details. The pseudo code of the proposed algorithm is represented in Fig. 1. The proposed EvoTLBO algorithm is based on standard TLBO which is known as a swarm intelligence algorithm [30]. TLBO has been presented to optimize continues problems. Hence, we need to adapt it for discrete search space. In other words, the movement operator of TLBO is changed to suit moving of individuals in a discrete space. The algorithm has three phases: initialization, update, and termination.

Solution representation plays an important role in success of a population based method. Here, as mentioned before in Section II-D, the same representation which is presented by EvoSuite is used. As can be seen from Fig. 2, every individual is represented as a chromosome and attributes of each individual is determined by its genes. In terms of test data generation, test cases and test suites are both represented as chromosomes. On the test suite level, a chromosome's genes correspond to test cases. On the test case level, genes are statements in a test case. Statements can be method call, constructor, primitive statement, filed, assignments, etc.

```
Initialize number of students, termination condition
While (termination condition not met)
    Calculate the mean of decision variables
    Identify the best solution as teacher        //in our case the best test case or test suite based on the criterion
    Identify the movement percentage based on the average and a random number        //sets the movement parameter
    Modify solution based on best solution        //moving towards the teacher
    $X_{new} = X_{old} + r(X_{teacher} - (T_F)Mean)$        //movement formula based on the movement percentage
    If the new solution better than existing
        Accept the solution        //continues to move toward a student
        Mutate the solution
    Else
        Reject the solution        //doesn't change the solution
    End If
    Select two solutions randomly $X_i$ and $X_j$
    If $X_i$ better than $X_j$
        $X_{new} = X_{old} + r(X_i - X_j)$        //move toward a better student or solution
    Else
        $X_{new} = X_{old} + r(X_j - X_i)$        //move away from a worse student or solution
    End If
    If the new solution better than existing
        Accept the solution
        Mutate the solution
    Else
        Reject the solution
    End If
End While
Return best solution
```

Fig. 1. Pseudo code of the proposed EvoTLBO algorithm.

A. Initialization

The algorithm receives number of individuals and termination condition as inputs. The process starts with a randomly generated initial population. For this purpose, the initial solutions generated by the EvoSuite are used.

B. Update (teaching phase)

The algorithm has two main phases of teaching and learning that simulates the teaching and learning in a classroom. The teacher is the best student of the class. The whole class works together to reach the best level of knowledge (best answer).

This means that social knowledge is shared between individuals through best solution ever found. In the teaching phase, every student moves toward the teacher. For this purpose, the average of decision variable is computed and each individual is updated using the following equation:

$$X_{new} = X_{old} + r(X_{teacher} - (T_F)Mean) \quad (4)$$

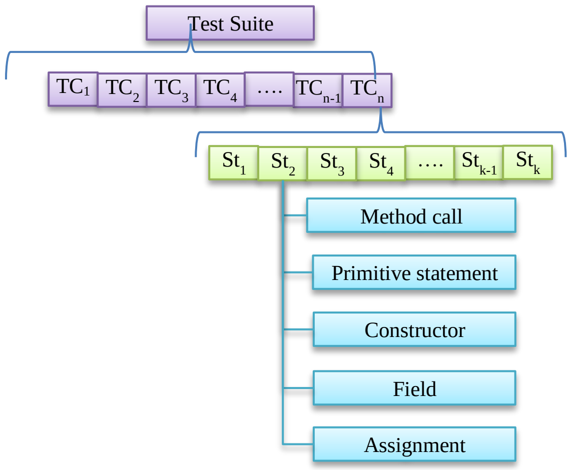

Fig. 2. Solution representation

Where, X_{new} and X_{old} are the new and old position of the individual, r is a random number, $X_{teacher}$ is the position of the teacher, and $(T_F)Mean$ is the mean of decision variables. This parameter shows that the knowledge of all the individuals are used to update solutions. Using social knowledge in appropriate way (as used in EvoTLBO) can help the algorithm perform better in search space.

The movement operator in EvoTLBO is changed in a way that makes it applicable to a discrete search space of the test data generation problem. The proposed movement strategy changes each individual's attributes with regards to another member to make one look similar to the other. This change is done by obtaining attributes of one individual and adding a portion (set as a parameter) of them to the other one.

The general model for movement considers that individual i wants to move towards individual j. Each individual represents a test suite which is consisted of an array of test cases. The number of test cases in an individual is considered as its position in the search space. For the sake of simplicity, an example is presented in Fig. 3.

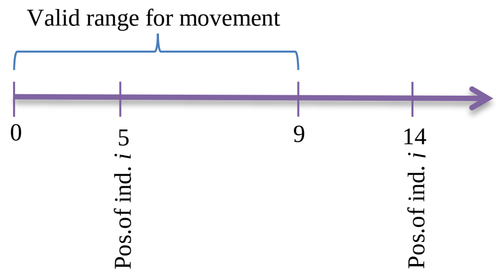

Fig. 3. An example of movement pattern.

Assume that individual i contains 5 test cases and individual j contains 14 test cases where both of them have two test cases in common. The positions of individuals i and j are 5 and 14 in the search space respectively. The difference between these two individuals is 7 because they have 2 common test cases. Based on this assumption, any movement pattern such as (4), (5), and (6) can be used.

What happens when moving one member closer to the other one is that some of the destination's attributes (i.e., test cases or statements) are copied and added to the source, leaving the destination as it is. Adding test cases or statements from one individual to the other is done by copying genes between chromosomes.

This movement works on both test suite and test case levels. On the test suite level, to decide which test cases are added to a test suite, they are prioritized based on their coverage. Test cases with exclusive goal coverage have higher priority. However, on test case level there is no prioritization.

After movement, the updated solutions are mutated using the same scenario given in Section II-E. The mutation helps the method to explore more regions to find better solutions.

C. Update (learning phase)

The teaching phase is followed by the learning phase in which the students tutor each other. In the teaching phase, all the members move toward the teacher. In the learning phase, a classmate is chosen randomly for each individual, and then they are compared in terms of their fitness. Actually, in case of the teaching phase, the individual is paired with the teacher in the movement operator meaning that it would get more like the teacher in that phase. If the random classmate's score (fitness value) is higher, then the individual moves toward it using following equation:

$$X_{new} = X_{old} + r(X_i - X_j) \tag{5}$$

In contrary if the randomly selected classmate has a lesser score, the individual gets further from it and closer to the teacher as:

$$X_{new} = X_{old} + r(X_j - X_i) \tag{6}$$

The unified random selection of the classmates results in searching a wider range of the search area, because not all the students move particularly toward the best-known member. Also, as in every movement operator, during movement the two individuals involved are maintained and no new members are generated.

After movement, the updated solutions are mutated using the same scenario given in Section II-E.

D. Termination

At the end of every iteration, the whole population is evaluated and if the minimum requirement (i.e., specific coverage percentage) is found in a member, the algorithm ends. On the other hand if there is no such member, the algorithm chooses the best individual as the teacher and continues into its evolutionary iterations. As these iterations can go on continually, in addition to the members' qualification, there are other stopping conditions like the number of iterations or time limit.

V. Performance Study

The performance of the proposed movement strategy is studied in this section. As shown in the results, there have been improvements in four criteria.

A. Tool Selection

The automation of software testing is done by various tools. The performance study of the proposed method is done by implementing and integrating it into the "EvoSuite" platform. Three of the well-known tools are briefly introduced here. One of the tools that also works on java programs is "Randoop" which generates the test cases mostly random and making assertions based on the feedback it gets from the execution of the test cases. Another random based tool is "T3". The basis of this tool is on random generation of test data sequences these sequences are saved and can be used for regression testing. In addition to that, T3 also performs "pair-wise" testing. "JTExpert" is another tool for java programs which uses search algorithms to generate a test suite. The drawback of this tool is that it only generates tests for branch coverage criterion and also has a lower overall performance in comparison to EvoSuite.

Regarding EvoSuite's performance among other similar tools, it has participated in the "9th International Workshop on

Search-Based Software Testing" and has achieved the highest overall score on the benchmark classes among the other tools [31]. It is noticeable that EvoSuite has close coverage to the manual method but at a fraction of time in generating them.

B. EvoSuite

EvoSuite is the tool of focus in this research. This tool generates test cases for codes written in java by using assertions to examine the integrity of the code [4].

To achieve this, EvoSuite has a hybrid method to generate test suites and optimizes them through an evolutionary process to satisfy a coverage criterion. EvoSuite suggests oracles for the generated test suites in the form of assertions. These small but effective assertions capture the behavior of the software to help the developer detect potential deviation. EvoSuite works on byte code which means that it doesn't need the source code. Test cases are evolved using evolutionary algorithms like Genetic and TLBO. One of the advantages of EvoSuite to other competitors is that it uses a whole test suite approach in which the evolutionary process tries to satisfy multiple coverage goals at the same time. This method and other challenges of using this tool in the real world are explained further, later in this section.

EvoSuite works on a master-slave architecture which enables parallel processing. This feature means that for example, calculating fitness value for a population can be done on different cores of a system or even on separate systems. This feature can help the performance of this tool effectively specially in large projects. In this architecture, a main process starts multiple sub-processes that do the actual search for the best test data. The communication between these processes is done by TCP, which makes EvoSuite independent from the signals of the operating system it is running on.

C. Dataset

Given that proving the performance of evolutionary algorithms is mathematically almost impossible, the performance in these cases is measured by empirical studies. There are challenges in using empirical methods. One of the important ones is to make sure that a technique which performs well under certain circumstances in the laboratory can also perform as well in real world problems. In literature, most of the works don't use a systematic method to choose the data set. In the matter of test data generation, there are many open source software available online. In [32] SF110 a set of 110 java projects were randomly selected from SourceForge code repository for automatic test data generation studies. This data set is also used by the EvoSuite development team. Since studying all the 22 thousand classes of this data set could take up to 1000 days, 50 random classes from SF110 is randomly selected to study the performance of the proposed EvoTLBO algorithm. The selected classes are shown in the table below. Classes are numbered in order to compare the coverage results.

TABLE. I. 50 CLASSES USED FOR TEST DATA GENERATION

Class #	Class Name
1	geo.google.mapping.AddressToUsAddressFunctor
2	com.werken.saxpath.XPathLexer
3	httpanalyzer.ScreenInputFilter
4	corina.formats.TRML
5	corina.map.SiteListPanel
6	lotus.core.phases.Phase
7	org.dom4j.tree.CloneHelper
8	org.dom4j.util.PerThreadSingleton
9	macaw.presentationLayer.CategoryStateEditor
10	org.fixsuite.message.view.ListView
11	com.browsersoft.openhre.hl7.impl.config.HL7SegmentMapImpl
12	com.lts.caloriecount.ui.budget.BudgetWin
13	com.lts.io.ArchiveScanner
14	com.lts.swing.table.dragndrop.test.RecordingEvent
15	com.lts.swing.thread.BlockThread
16	de.outstare.fortbattleplayer.gui.battlefield.BattlefieldCell
17	org.sourceforge.ifx.framework.complextype.RecChkOrdInqRs_Type
18	org.sourceforge.ifx.framework.complextype.PassbkItemInqRs_Type
19	umd.cs.shop.JSListSubstitution
20	jigl.image.utils.LocalDifferentialGeometry
21	org.sourceforge.ifx.framework.element.Fee
22	com.lts.xml.MapElement
23	weka.gui.beans.TrainTestSplitMaker
24	weka.filters.unsupervised.attribute.RandomProjection
25	com.lts.swing.table.rowmodel.tablemodel.RowModelTableModel
26	net.sourceforge.squirrel_sql.fw.datasetviewer.ColumnDisplayDefinition
27	org.gudy.azureus2.core3.util.ShellUtilityFinder
28	org.gudy.azureus2.core3.torrentdownloader.impl.TorrentDownloaderManager
29	jcmdline.UsageFormatter
30	net.sourceforge.squirrel_sql.fw.sql.ISQLExecutionCallback
31	br.com.jnfe.base.ICMSST

Class #	Class Name
32	glengineer.agents.setters.FunctionsOnSequentialGroupAndElement
33	org.sourceforge.ifx.framework.element.ForExDealStatusInqRq
34	org.sourceforge.ifx.framework.element.BankAcctTrnImgRevRs
35	com.aelitis.azureus.core.download.DownloadManagerEnhancer
36	corina.browser.Row
37	corina.graph.DensityPlot
38	com.browsersoft.openhre.hl7.impl.parser.HL7CheckerStateImpl
39	org.bouncycastle.asn1.DERUTCTime
40	module.RuleSet
41	net.kencochrane.a4j.DAO.Cart
42	org.petsoar.security.Address
43	org.sourceforge.ifx.framework.pain001.simpletype.DocumentType1Code
44	corina.prefs.components.BoolPrefComponent
45	jaw.gui.ProcessarEntidades
46	org.jcvi.jillion.fasta.pos.PositionFastaRecord
47	de.huxhorn.lilith.data.access.AccessEvent
48	com.sap.netweaver.porta.mon.StopCommand
49	org.sourceforge.ifx.framework.element.DevDepType
50	org.sourceforge.ifx.framework.complextype.DepAcctStmtRevRs_Type

D. *Algorithm Configurations*

All of the algorithms start with an initial population of size 50 which is generated with the random method mentioned in the literature. The algorithms have 2 minutes to run each time. In addition to timeout, a certain coverage percentage (i.e., 100%) is also a stopping condition. Each of the classes have been processed in 10 iterations to ensure reliable results.

The total time required for runs is calculated as follows:

$$Iterations * Coverage\ criterion * Classes$$
$$* Execution\ time = 10 * 4 * 50 * 2$$
$$= 66\ hours$$

In addition to the common settings, each algorithm has its own specific configurations which are set as follows: for the genetic algorithm, selection is rank and crossover is single point. In the proposed EvoTLBO method, the teaching factor is selected as 0<random<2.0.

E. *Experimental Results*

Standard GA and Monotonic GA which are built into the EvoSuite tool by its developing team are used for comparing the results of the proposed EvoTLBO algorithm in 4 coverage criterions of Branch, Line, Method and Output. Two factors have been used for performance comparison, the number of classes which the algorithm has achieved the highest coverage in and the percentage of the total number of covered goals. These two factors are shown in the last row of the table for each algorithm. For every table the first column is the class number correspondent to Table 1. For every algorithm the first column is the coverage percentage in that class and the second column is the ratio of the covered goals to the total number of goals for that class in the specified criterion.

Branch coverage: The results of applying the EvoTLBO algorithm with the suggested movement operator in it are presented in Table 2. The table shows the results in branch coverage criterion. Standard GA has the highest score in terms of covering the most number of classes. This algorithm has the highest coverage in 36 classes in 12 of which achieving exclusive coverage that no other algorithm has. The monotonic GA algorithm gets the second rank by achieving highest coverage in 28 classes and exclusive coverage in only 3 cases. The proposed EvoTLBO algorithm does not show a good performance compared to the two genetic algorithms, it achieves the highest coverage in 24 classes alongside with the other two and it has exclusive coverage in only two classes. Regarding the goal coverage independent of which class they are in, all three algorithms have close performance. There are a total of 2101 goals in this criterion in all of the classes combined. Although the ranking stays the same, but the 62.67% of standard GA at first is close to 59.21% of the EvoTLBO at last.

TABLE. II. BRANCH COVERAGE RESULTS

#	Standard GA		Monotonic GA		EvoTLBO	
1	20.00%	(6/30)	20.00%	(6/30)	20.00%	(6/30)
2	85.99%	(416.2/484)	86.84%	(420.3/484)	85.12%	(412/484)
3	100.00%	(5/5)	94.00%	(4.7/5)	98.00%	(4.9/5)
4	24.78%	(22.3/90)	21.89%	(19.7/90)	19.56%	(17.6/90)
5	0.65%	(1/153)	0.65%	(1/153)	0.59%	(0.9/153)
6	72.86%	(20.4/28)	100.00%	(28/28)	100.00%	(28/28)
7	100.00%	(4/4)	100.00%	(4/4)	100.00%	(4/4)
8	85.71%	(6/7)	85.71%	(6/7)	85.71%	(6/7)
9	8.33%	(1/12)	7.50%	(0.9/12)	7.50%	(0.9/12)
10	5.26%	(4/76)	5.26%	(4/76)	4.21%	(3.2/76)
11	100.00%	(12/12)	100.00%	(12/12)	100.00%	(12/12)

#	Standard GA		Monotonic GA		EvoTLBO	
12	12.50%	(4/32)	12.50%	(4/32)	10.63%	(3.4/32)
13	71.33%	(32.1/45)	69.78%	(31.4/45)	57.78%	(26/45)
14	99.67%	(29.9/30)	100.00%	(30/30)	97.67%	(29.3/30)
15	98.89%	(8.9/9)	97.78%	(8.8/9)	98.89%	(8.9/9)
16	82.54%	(58.6/71)	81.83%	(58.1/71)	65.07%	(46.2/71)
17	100.00%	(34/34)	100.00%	(34/34)	100.00%	(34/34)
18	100.00%	(28/28)	100.00%	(28/28)	100.00%	(28/28)
19	92.86%	(6.5/7)	88.57%	(6.2/7)	90.00%	(6.3/7)
20	91.56	(181.2/198)	88.28	(174.7/198)	95.70	(189.4/198)
21	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
22	100.00%	(1/1)	100.00%	(1/1)	100.00%	(1/1)
23	66.23%	(70.2/106)	64.62%	(68.5/106)	54.72%	(58/106)
24	41.71%	(65.9/158)	41.39%	(65.4/158)	37.34%	(59/158)
25	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
26	90.41%	(44.3/49)	90.82%	(44.5/49)	91.02%	(44.6/49)
27	80.00%	(7.2/9)	88.89%	(8/9)	88.89%	(8/9)
28	78.82%	(40.2/51)	74.31%	(37.9/51)	71.18%	(36.3/51)
29	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
30	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
31	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
32	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
33	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
34	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
35	8.53%	(9.3/109)	3.85%	(4.2/109)	7.34%	(8/109)
36	57.59%	(33.4/58)	58.79%	(34.1/58)	50.17%	(29.1/58)
37	17.65%	(3/17)	17.65%	(3/17)	17.65%	(3/17)
38	92.86%	(97.5/105)	91.24%	(95.8/105)	80.86%	(84.9/105)
39	99.71%	(33.9/34)	99.12%	(33.7/34)	98.53%	(33.5/34)
40	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
41	23.08%	(9/39)	23.08%	(9/39)	23.08%	(9/39)
42	100.00%	(11/11)	100.00%	(11/11)	100.00%	(11/11)
43	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
44	0.00%	(0/6)	0.00%	(0/6)	0.00%	(0/6)
45	100.00%	(1/1)	100.00%	(1/1)	100.00%	(1/1)
46	100.00%	(18/18)	100.00%	(18/18)	100.00%	(18/18)
47	100.00%	(134/134)	100.00%	(134/134)	92.54%	(124/134)
48	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
49	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
50	100.00%	(26/26)	100.00%	(26/26)	100.00%	(26/26)
	36	62.67%	28	62.55%	24	59.21%

Line coverage: In Table 3 the results are based upon the line coverage criterion. The monotonic GA has achieved the highest coverage in 30 cases 7 of which were exclusive to this algorithm. The standard GA holds the second place closely with just 1 less class. Although EvoTLBO algorithm has the third place but it still has competitive results as it achieves the highest coverage in 25 classes along with others and has exclusive coverage in 4 classes. The same ranking goes for goal coverage percentage. The two genetic algorithms have close scores with less than 1 percent difference and the EvoTLBO algorithm has coverage more than 2 percent lower.

TABLE. III. LINE COVERAGE RESULTS

#	Standard GA		Monotonic GA		EvoTLBO	
1	27.03%	(10/37)	27.03%	(10/37)	26.76%	(9.9/37)
2	86.74%	(308.8/356)	87.67%	(312.1/356)	87.08%	(310/356)
3	98.18%	(10.8/11)	97.27%	(10.7/11)	98.18%	(10.8/11)
4	44.87%	(70/156)	48.53%	(75.7/156)	31.41%	(49/156)
5	0.43%	(1/232)	0.43%	(1/232)	0.39%	(0.9/232)
6	100.00%	(20/20)	100.00%	(20/20)	100.00%	(20/20)
7	38.46%	(5/13)	38.46%	(5/13)	38.46%	(5/13)
8	79.13%	(18.2/23)	80.00%	(18.4/23)	78.26%	(18/23)
9	0.00%	(0/46)	0.00%	(0/46)	0.00%	(0/46)
10	3.64%	(8/220)	3.64%	(8/220)	3.64%	(8/220)
11	100.00%	(26/26)	100.00%	(26/26)	100.00%	(26/26)
12	26.92%	(28/104)	26.92%	(28/104)	26.92%	(28/104)
13	81.36%	(53.7/66)	81.97%	(54.1/66)	62.88%	(41.5/66)
14	94.19%	(69.7/74)	94.05%	(69.6/74)	96.89%	(71.7/74)
15	72.69%	(18.9/26)	73.08%	(19/26)	72.69%	(18.9/26)
16	78.85%	(102.5/130)	87.08%	(113.2/130)	72.54%	(94.3/130)
17	100.00%	(51/51)	100.00%	(51/51)	100.00%	(51/51)

#	Standard GA		Monotonic GA		EvoTLBO	
18	100.00%	(42/42)	100.00%	(42/42)	100.00%	(42/42)
19	93.00%	(9.3/10)	90.00%	(9/10)	93.00%	(9.3/10)
20	87.66	(326/372)	87.25	(324.5/372)	92.47	(343.9/372)
21	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
22	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
23	70.38%	(112.6/160)	67.94%	(108.7/160)	65.81%	(105.3/160)
24	52.68%	(125.9/239)	53.35%	(127.5/239)	50.54%	(120.8/239)
25	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
26	95.80%	(95.8/100)	95.80%	(95.8/100)	96.00%	(96/100)
27	93.75%	(7.5/8)	100.00%	(8/8)	100.00%	(8/8)
28	90.20%	(45.1/50)	89.00%	(44.5/50)	88.80%	(44.4/50)
29	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
30	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
31	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
32	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
33	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
34	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
35	10.16%	(18.6/183)	5.79%	(10.6/183)	13.28%	(24.3/183)
36	66.63%	(57.3/86)	65.93%	(56.7/86)	58.95%	(50.7/86)
37	8.51%	(4/47)	8.51%	(4/47)	8.30%	(3.9/47)
38	86.54%	(165.3/191)	85.39%	(163.1/191)	77.38%	(147.8/191)
39	100.00%	(55/55)	98.91%	(54.4/55)	98.91%	(54.4/55)
40	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
41	39.06%	(50/128)	39.06%	(50/128)	39.06%	(50/128)
42	100.00%	(15/15)	100.00%	(15/15)	100.00%	(15/15)
43	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
44	21.43%	(3/14)	21.43%	(3/14)	21.43%	(3/14)
45	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
46	100.00%	(26/26)	100.00%	(26/26)	100.00%	(26/26)
47	100.00%	(86/86)	100.00%	(86/86)	100.00%	(86/86)
48	12.50%	(1/8)	12.50%	(1/8)	10.00%	(0.8/8)
49	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
50	100.00%	(39/39)	100.00%	(39/39)	100.00%	(39/39)
	29	57.32%	30	57.52%	25	55.04%

Output coverage: The results of output coverage are presented in Table 4. On the number of classes with the highest coverage, standard GA scores 29 cases with 7 exclusive highest coverage. Monotonic GA achieves highest coverage in 24 classes with exclusive coverage in 3 cases. EvoTLBO ranked at last scores 21 on highest coverage with only 2 exclusively covered classes. Regarding the total goals in this criterion, of all the 1056 goals, standard GA being at the top covers 49.57% of them. EvoTLBO with the least score, covers 47.91% of the goals.

TABLE. IV. OUTPUT COVERAGE RESULTS

#	Standard GA		Monotonic GA		EvoTLBO	
1	50.00%	(2/4)	50.00%	(2/4)	50.00%	(2/4)
2	39.01%	(55.4/142)	38.59%	(54.8/142)	38.59%	(54.8/142)
3	66.67%	(2/3)	66.67%	(2/3)	66.67%	(2/3)
4	6.06%	(2/33)	6.06%	(2/33)	6.06%	(2/33)
5	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
6	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
7	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
8	50.00%	(1/2)	50.00%	(1/2)	50.00%	(1/2)
9	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
10	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
11	63.64%	(7/11)	63.64%	(7/11)	63.64%	(7/11)
12	0.00%	(0/77)	0.00%	(0/77)	0.00%	(0/77)
13	80.00%	(4/5)	78.00%	(3.9/5)	80.00%	(4/5)
14	48.67%	(14.6/30)	33.00%	(9.9/30)	50.67%	(15.2/30)
15	0	(0/0)	0.00%	(0/0)	0.00%	(0/0)
16	83.33%	(10/12)	83.33%	(10/12)	83.33%	(10/12)
17	100.00%	(40/40)	100.00%	(40/40)	99.00%	(39.6/40)
18	100.00%	(33/33)	100.00%	(33/33)	99.70%	(32.9/33)
19	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
20	48.90	(93.8/192)	46.30	(88.8/192)	50.31	(96.5/192)
21	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
22	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
23	37.78%	(17/45)	38.89%	(17.5/45)	35.78%	(16.1/45)
24	32.03%	(25.3/79)	33.16%	(26.2/79)	28.99%	(22.9/79)

#	Standard GA		Monotonic GA		EvoTLBO	
25	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
26	85.21%	(60.5/71)	83.38%	(59.2/71)	82.68%	(58.7/71)
27	44.44%	(4/9)	44.44%	(4/9)	44.44%	(4/9)
28	10.69%	(9.3/87)	10.11%	(8.8/87)	9.89%	(8.6/87)
29	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
30	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
31	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
32	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
33	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
34	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
35	3.78%	(3.1/82)	1.22%	(1/82)	2.68%	(2.2/82)
36	47.78%	(12.9/27)	44.07%	(11.9/27)	41.85%	(11.3/27)
37	80.00%	(4/5)	80.00%	(4/5)	80.00%	(4/5)
38	94.63%	(51.1/54)	91.48%	(49.4/54)	81.11%	(43.8/54)
39	68.42%	(13/19)	67.89%	(12.9/19)	65.79%	(12.5/19)
40	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
41	50.00%	(6/12)	50.00%	(6/12)	50.00%	(6/12)
42	100.00%	(15/15)	100.00%	(15/15)	100.00%	(15/15)
43	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
44	0.00%	(0/0)	0.00%	(0/0)	0.00%	(0/0)
45	52.76%	(15.3/29)	53.45%	(15.5/29)	49.66%	(14.4/29)
46	60.00%	(9/15)	60.00%	(9/15)	59.33%	(8.9/15)
47	93.24%	(69/74)	93.24%	(69/74)	93.24%	(69/74)
48	0.00%	(0/3)	0.00%	(0/3)	0.00%	(0/3)
49	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
50	100.00%	(26/26)	100.00%	(26/26)	100.00%	(26/26)
	29	49.57%	24	48.58%	21	47.91%

Method coverage: The coverage of methods is the last criterion used for comparison. The results of method coverage are presented in Table 5. In this criterion EvoTLBO performs better than the other two by achieving the highest coverage in 44 classes and exclusively covering 11 classes with the highest coverage percentage. Standard GA and monotonic GA both cover 35 classes with highest coverage alongside each other and EvoTLBO. The only case which standard GA has exclusive coverage is number 35. Monotonic GA doesn't cover any classes exclusively. Regarding the total goal coverage, EvoTLBO again has the first rank with 90.08% coverage. The two genetic algorithms have very close coverage percentage.

TABLE. V. METHOD COVERAGE RESULTS

#	Standard GA		Monotonic GA		EvoTLBO	
1	100.00%	(3/3)	100.00%	(3/3)	100.00%	(3/3)
2	100.00%	(44/44)	100.00%	(44/44)	100.00%	(44/44)
3	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
4	72.00%	(3.6/5)	80.00%	(4/5)	82.00%	(4.1/5)
5	50.00%	(1/2)	50.00%	(1/2)	50.00%	(1/2)
6	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
7	25.00%	(1/4)	25.00%	(1/4)	100.00%	(4/4)
8	100.00%	(4/4)	100.00%	(4/4)	100.00%	(4/4)
9	20.00%	(1/5)	20.00%	(1/5)	20.00%	(1/5)
10	33.33%	(1/3)	33.33%	(1/3)	23.33%	(0.7/3)
11	100.00%	(8/8)	100.00%	(8/8)	100.00%	(8/8)
12	11.11%	(1/9)	20.00%	(1.8/9)	28.89%	(2.6/9)
13	50.00%	(4/8)	50.00%	(4/8)	78.75%	(6.3/8)
14	100.00%	(16/16)	100.00%	(16/16)	100.00%	(16/16)
15	76.67%	(4.6/6)	81.67%	(4.9/6)	83.33%	(5/6)
16	100.00%	(8/8)	100.00%	(8/8)	100.00%	(8/8)
17	100.00%	(34/34)	100.00%	(34/34)	100.00%	(34/34)
18	100.00%	(28/28)	100.00%	(28/28)	100.00%	(28/28)
19	100.00%	(3/3)	100.00%	(3/3)	100.00%	(3/3)
20	100.00%	(19/19)	100	(19/19)	100	(19/19)
21	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
22	100.00%	(1/1)	100.00%	(1/1)	100.00%	(1/1)
23	100.00%	(23/23)	100.00%	(23/23)	100.00%	(23/23)
24	75.00%	(24/32)	76.88%	(24.6/32)	86.25%	(27.6/32)
25	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
26	93.75%	(30/32)	93.75%	(30/32)	99.69%	(31.9/32)
27	100.00%	(4/4)	100.00%	(4/4)	100.00%	(4/4)
28	100.00%	(10/10)	100.00%	(10/10)	100.00%	(10/10)
29	100.00%	(4/4)	100.00%	(4/4)	100.00%	(4/4)
30	0.00%	(0/1)	0.00%	(0/1)	0.00%	(0/1)

#	**Standard GA**		**Monotonic GA**		**EvoTLBO**	
31	0.00%	(0/6)	0.00%	(0/6)	0.00%	(0/6)
32	0.00%	(0/8)	0.00%	(0/8)	0.00%	(0/8)
33	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
34	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
35	33.08%	(4.3/13)	22.31%	(2.9/13)	26.15%	(3.4/13)
36	90.00%	(9/10)	90.00%	(9/10)	95.00%	(9.5/10)
37	75.00%	(3/4)	75.00%	(3/4)	82.50%	(3.3/4)
38	100.00%	(42/42)	100.00%	(42/42)	100.00%	(42/42)
39	99.23%	(12.9/13)	100.00%	(13/13)	100.00%	(13/13)
40	100.00%	(1/1)	100.00%	(1/1)	100.00%	(1/1)
41	100.00%	(7/7)	100.00%	(7/7)	100.00%	(7/7)
42	100.00%	(11/11)	100.00%	(11/11)	100.00%	(11/11)
43	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
44	33.33%	(1/3)	33.33%	(1/3)	30.00%	(0.9/3)
45	90.63%	(14.5/16)	87.50%	(14/16)	92.50%	(14.8/16)
46	100.00%	(8/8)	100.00%	(8/8)	100.00%	(8/8)
47	100.00%	(34/34)	100.00%	(34/34)	100.00%	(34/34)
48	50.00%	(1/2)	50.00%	(1/2)	80.00%	(1.6/2)
49	100.00%	(2/2)	100.00%	(2/2)	100.00%	(2/2)
50	100.00%	(26/26)	100.00%	(26/26)	100.00%	(26/26)
	35	87.41%	35	87.47%	44	90.08%

VI. Conclusion And Future Works

In this work, a method based on TLBO has been proposed to generate test data automatically. The proposed method was applied on 50 randomly selected classes in EvoSuite. The performance of EvoTLBO method was compared with the two methods of standard GA and monotonic GA. The results showed that EvoTLBO is efficient and provides competitive results in comparison with the other methods.

The experience gained on working with different evolutionary algorithms has given us a wider perspective on this matter. Knowing the challenges of software testing and software quality validation, suggestions to improve the results further are made. Given the performance of swarm intelligence algorithm, more empirical studies using a larger number of classes are suggested. Extending EvoTLBO with new movement patterns and social models may result better performance. Analyzing other swarm intelligence paradigm algorithms like bats in generating test data is suggested. Other optimization paradigm algorithms for test data generation could be studied. Proposing a movement method in the search space of swarm intelligence algorithms for solving object oriented test problems is of importance. It is recommended to present a method to change the discrete space of the algorithm to a continuous form to implement the movement. Since evolutionary algorithms are dependent on their initial parameters values, empirical studies on tuning these parameters by comparing the execution results of different values is recommended. Utilizing multiple goal optimization algorithms in generating test data, to approach all the goals at the same time. Using fitness functions to generate tests for non-functional properties of software is a need. Further analysis of other tools like Randoop is recommended.

References

[1] J. P. Galeotti, G. Fraser, and A. Arcuri, “Improving search-based test suite generation with dynamic symbolic execution,” 2013 IEEE 24th Int. Symp. Softw. Reliab. Eng. ISSRE 2013, pp. 360–369, 2013.

[2] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, “A systematic review of the application and empirical investigation of search-based test case generation,” IEEE Trans. Softw. Eng., vol. 36, no. 6, pp. 742–762, 2010.

[3] K. Lakhotia, M. Harman, and H. Gross, “AUSTIN: A Tool for Search Based Software Testing for the C Language and Its Evaluation on Deployed Automotive Systems,” in 2nd International Symposium on Search Based Software Engineering, 2010, pp. 101–110.

[4] G. Fraser and A. Arcuri, “EvoSuite : Automatic Test Suite Generation for Object-Oriented Software,” Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng., pp. 416–419, 2011.

[5] G. Fraser and A. Arcuri, “Evolutionary Generation of Whole Test Suites,” 2011 11th Int. Conf. Qual. Softw., pp. 31–40, 2011.

[6] P. McMinn, “Search-Based Software Testing: Past, Present and Future,” in Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011, pp. 153–163.

[7] J. M. R. J. C. M. V. G. F. A. Arcuri, “Combining Multiple Coverage Criteria in Search-Based Unit Test Generation,” 2015.

[8] P. McMinn, “Search-based software test data generation: A survey,” Softw. Test. Verif. Reliab., vol. 14, no. 2, pp. 105–156, 2004.

[9] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, A detailed investigation of the effectiveness of whole test suite generation. 2016.

[10] M. Barros and Y. Labiche, “Search-Based Software Engineering: 7th International Symposium, SSBSE 2015 Bergamo, Italy, September 5-7, 2015 Proceedings,” Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 9275, pp. 77–92, 2015.

[11] B. Korel, “Automated software test data generation,” IEEE Trans. Softw. Eng., vol. 16, no. 8, pp. 870–879, Aug. 1990.

[12] N. Alshahwan and M. Harman, “Coverage and Fault Detection of the Output-uniqueness Test Selection Criteria,” in Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 181–192.

[13] H. Sthamer and P. Morgannwg, “The Automatic Generation of Software Test Data Using Genetic Algorithms,” no. November, 1995.

[14] P. Tonella, “Evolutionary Testing of Classes,” pp. 119–128.

[15] M. Harman, Y. Jia, and Y. Zhang, “Achievements , open problems and challenges for search based software testing,” 8th IEEE Int. Conf. Softw. Testing, Verif. Valid., no. Icst, 2015.

[16] H. Maaranen, K. Miettinen, and M. M. Mäkelä, “Quasi-random initial population for genetic algorithms,” Comput. Math. with Appl., vol. 47, no. 12, pp. 1885–1895, 2004.

[17] A. Pachauri and G. Srivastava, “Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism,” J. Syst. Softw., vol. 86, no. 5, pp. 1191–1208, 2013.

[18] G. Fraser and A. Arcuri, “Whole test suite generation,” IEEE Trans. Softw. Eng., vol. 39, no. 2, pp. 276–291, 2013.

[19] Y. Suresh and S. Rath, “A Genetic Algorithm based Approach for Test Data Generation in Basis Path Testing,” Int. J. Soft Comput. Softw. Eng., vol. 3, no. 3, pp. 326–332, 2014.

[20] P. M. S. Bueno, M. Jino, and W. E. Wong, “Diversity oriented test data generation using metaheuristic search techniques,” Inf. Sci. (Ny)., vol. 259, pp. 490–509, 2014.

[21] I. Hermadi, C. Lokan, and R. Sarker, “Dynamic stopping criteria for search-based test data generation for path testing,” Inf. Softw. Technol., vol. 56, no. 4, pp. 395–407, 2014.

[22] A. Pachauri, “Towards a parallel approach for test data generation for branch coverage with genetic algorithm using the extended path prefix strategy Towards a Parallel Approach for Test Data Generation for Branch Coverage with Genetic Algorithm using the Extended Path,” no. March, 2016.

[23] D. YueMing, W. YiTing, and W. DingHui, “Particle swarm optimization algorithm for test case automatic generation based on clustering thought,” in Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), 2015 IEEE International Conference on, 2015, pp. 1479–1485.

[24] B. Hoseini and S. Jalili, “Automatic test path generation from sequence diagram using genetic algorithm,” Telecommun. (IST), 2014 7th Int. Symp., pp. 106–111, 2014.

[25] G. Fraser and A. Zeller, “Mutation-driven Generation of Unit Tests and Oracles,” 2010.

[26] G. Fraser and A. Zeller, “Exploiting common object usage in test case generation,” Proc. - 4th IEEE Int. Conf. Softw. Testing, Verif. Validation, ICST 2011, pp. 80–89, 2011.

[27] J. Malburg and G. Fraser, “Combining Search-based and Constraint-based Testing.”

[28] C. Koleejan, B. Xue, and M. Zhang, “Code Coverage Optimisation in Genetic Algorithms and Particle Swarm Optimisation for Automatic Software Test Data Generation,” pp. 1204–1211, 2015.

[29] G. Fraser and A. Arcuri, “txtUnknown - Unknown - EvoSuite On The Challenges of Test Case Generation in the Real World.txt.pdf.”

[30] R. V. Rao, V. J. Savsani, and D. P. Vakharia, “Teaching–learning-based optimization: A novel method for constrained mechanical design optimization problems,” Comput. Des., vol. 43, no. 3, pp. 303–315, 2011.

[31] G. Fraser, “EvoSuite at the SBST 2016 Tool Competition,” pp. 10–13, 2016.

[32] G. Fraser and C. Science, “A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite,” ACM Trans. Softw. Eng. Methodol., vol. 24, no. 2, p. 8, 2014.