International Journal of Engineering Trends and Technology
ISSN: 2231–5381 / https://doi.org/10.14445/22315381/IJETT-V73I1P131

Volume 73 Issue 1, 357-370, January 2025


*Original Article*

# Identification of Performance Regression Causing Code Modifications Using Memetic Algorithm

Brindha Subburaj[1], J. Uma Maheswari[2]

[1,2]*School of Computer Science and Engineering, Vellore Institute of Technology, Chennai, Tamil Nadu, India.*

[1]*Corresponding Author : brindha.s@vit.ac.in*

Received: 13 September 2024 Revised: 11 January 2025 Accepted: 16 January 2025 Published: 31 January 2025

***Abstract -*** *Regression testing in software development is a vital and inevitable process performed to ensure that the modifications made to the code do not affect the overall quality of the software. Conducting performance regression tests each and every time when we do some modifications to the code is costlier. Thus, it would be better if we could identify the code modifications that may lead to performance regression and apply regression tests only during such code modification instances. The multi-objective optimization problem formulated includes detecting the code modification that causes performance regression. In this paper, we propose a memetic algorithm named Memetic algorithm using NSGA-II and Local Search (MNSLS), where NSGA-II algorithm with controlled elitism technique is used for global search along with a new improved and controlled local search method. These global and local search techniques improve the exploration and exploitation properties of the algorithm and help to find fitter solutions. MNSLS is used to optimize the identification rules, which could characterize and identify the code modifications that pose a problem to the software quality by finding solutions with a better trade-off between the hit and dismiss rates as objectives. The performance of the proposed algorithm is evaluated using a set of around 8000 Git project commits. The multi-objective optimization results are compared with other evolutionary algorithms using the Hypervolume metric and Mann-Whitney U test. The proposed method is further compared with another evolutionary-based regression identification method called PRICE. The results of the above analysis show that the proposed MNSLS algorithm-based regression identification method is more efficient than other methods.*

***Keywords -*** *Performance Regression, Evolutionary Algorithm, Local Search, Memetic Algorithm, MNSLS.*

## 1. Introduction

In the software development field, performance regression testing is used to ensure that recent code modifications do not affect the functioning in terms of meeting the performance specifications of software developed and tested earlier. Deviation in response time and resource utilization are a few examples of performance regression [1]. One of the standard procedures to identify the possibility of performance regression occurrence is to execute performance benchmark tests during every code change [2]. The major issues with performing regression tests for every code modification are time and resource costs. Thus, the options available to perform regression tests are (a) to test after every code modification, (b) to test in intervals and (c) to delay the test till the end of an iteration or ignore the test. The first option, as already discussed, is costlier and delays the further development process to wait for the regression test results [3]. Chen et al. [1], in their work, have recommended testing for every code modification rather than delaying it to the future. The second option, performing regression tests in intervals [4], may reduce the cost of testing but introduces overhead in finding the code modification that created a performance regression if one such exists. The third option is also expensive as it delays the performance regression test till the end of a development cycle, and if any code modification that has introduced performance regression and is detected at a later point will demand rework [5]. Many research studies have evaluated the benefits of using performance regression tests only when the code changes are problematic. Identifying problematic code modifications, which create a performance regression, is challenging. One way to track such code changes is by analysing their patterns [6]. Huang et al. [3] have suggested a Performance Risk Analysis method (PRA) where code modifications are ranked based on how problematic they are in introducing performance regression. Further, performance regression tests are carried out for the code modifications with higher ranks. Using a performance detection model to identify the problematic code changes by training the model with a limited performance test, and once the model performance is satisfactory, it is used to predict the need for a performance regression test for a code modification [7]. Alocer et al. [8] used a horizontal profiling method to identify the code modifications that created performance regression. They compare the current code modification with

respect to benchmark profiles, assign a cost factor to detect deviations and make a decision about carrying out tests. Identifying problematic code changes that cause performance regression using the above methods is time-consuming and characterizing such changes will help in prioritizing and testing these changes easily. Recently, metrics have been used to characterize the code changes, which measures the structural effect of these changes on source code.

These static and dynamic metric-based performance regression identification using evolutionary algorithms is still in the early stage of research, which sets the motivation for conducting research in this field. Several problems related to software design and development are solved using evolutionary algorithms. In this research, the identification of code modifications requiring performance regression is formulated as a multi-objective optimization problem, and recent research related to this context is reviewed and presented. Regression testing is an optimization problem where value-based objectives are maximized, and cost-based objectives are minimized [9]. Evolutionary algorithms are used to solve the optimization problem in our research.

The challenge in using evolutionary methods includes utilizing appropriate structural metrics that characterize the code changes, fine-tuning the algorithm based on the problem type, and many other related factors that affect the performance of the above methods. Our primary objective is identifying performance regression and creating code modifications through optimal identification rules using an appropriate evolutionary algorithm. The multi-objective optimization problem includes two objectives: hit rate and dismiss rate [9], representing accuracy in identifying commits that create performance regression and accuracy and excluding the commits that do not create performance regression, respectively. The optimization algorithm tries to find feasible solutions that are the best trade-off between the objectives through the rules.

The proposed evolutionary algorithm, named Memetic algorithm using NSGA-II and Local Search (MNSLS), which is an NSGA-II with a controlled elitism-based evolutionary search algorithm, is further combined with an improved and controlled local search method, balancing the exploration and exploitation properties of the evolutionary algorithm. MNSLS execution begins with a set of input commits. It develops the identification rules to identify the code modifications (commits) that cause performance regression with the help of a set of dynamic metrics [10]. Along the evolution process, the identified rule is improved to identify the problem-causing commits and exclude non-problematic commits accurately. The research contributions specifically are,

- A Memetic algorithm using NSGA-II and Local Search (MNSLS) is proposed and applied to the performance regression identification problem.
- A new, improved, controlled local search method is proposed to exploit potential solutions.
- Experimentation using 8596 Git Project commits data taken from [10].
- Performance comparison of the proposed MNSLS algorithm with other evolutionary algorithms using hypervolume performance metric and Mann-Whitney $U$ test, a statistical test.
- Performance comparison with the price regression detection method is needed to strengthen the results further.

The proposed controlled local search technique and applying the memetic algorithm to the problem of identifying problematic code modifications is the novelty of this research work. The paper is structured as follows: Section 2 gives the related works, Section 3 details the proposed research methodology, Section 4 gives the experimentation, results, and discussion, followed by the conclusion.

## 2. Related Works

Characterizing the code modifications that introduce performance regression is now a widely considered option, as it is easier to make decisions about performance testing. It requires tracing the similarities and identical patterns along the code modifications. Code metrics, both static and dynamic nature metrics, are currently used to profile the code modifications. These metrics assess the structural impacts of the latest code modifications on source code, like lines of code, code complexity, etc. [1]. Oliveria et al. [6] have proposed the Perphecy approach to simplify the performance regression test selection process using static and dynamic metrics to identify the code modifications that create performance regression. Alshoaibi et al. [10] have presented an evolutionary search-based approach to detect problematic code modifications using static and dynamic indicators and have highlighted the effectiveness of dynamic metrics like count of added and deleted methods, highest percent static function length change, etc. Mkaouer et al. [11] have used around fifteen different quality metrics to evaluate the effect of refactoring.

Evolutionary Algorithms (EAs) are proposed using the Darwinian principle of evolution and survival of the fittest and are search-based optimization methods. They are widely used to solve real-world optimization problems. Fitness or objective functions are framed using the decision variables related to the area of optimization. EAs try to find the solution that best solves the function. EA starts its evolution with an initial set of parent populations chosen from the search space; the child population is generated using the recombination and mutation operation. Fitter solutions among the parent and child population are selected for further processing based on their fitness function values. Optimization algorithms are classified as single-objective, multi-objective and, recently, many-objective optimization algorithms based on the number

of objectives that are to be optimized. The present research work is based on a multi-objective optimization problem, and a few representative algorithms from this class are as follows. Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [12], Strength Pareto Evolutionary Algorithm 2 (SPEA2) [13], Fuzzy Adaptive Multi-Objective Differential Evolution with Diversity Control (FAMDE-DC) [14], self-adaptive multi-objective differential evolution-based trajectory optimization algorithm (STO), Non-dominated Sorting Moth Flame Optimization (NS-MFO) algorithm [16], Multi-Objective Evolutionary Algorithm based on Decomposition (MOEA/D) [17].

Memetic Algorithm (MA) [18] combines population-based global search algorithms like evolutionary algorithms and local search techniques like Tabu search, Hill climbing, etc. Several research papers published over previous years show the effectiveness of MA with high-performance results in solving optimization problems. A few recent studies based on MA have included the application of NSGA-II and Tabu search-based memetic algorithms to solve green job shop scheduling problems [19].

An MA based on Differential Evolution and a hill-climbing algorithm is used to optimize the clustering process in wireless sensor networks [20]. A Memetic algorithm using Teaching Learning based optimization and a Tabu search algorithm are used to optimize graph coloring problems [21]. When applied to such varied problem domains, the memetic algorithm further proves its effectiveness. Extending evolutionary algorithms to build predictive models using software metrics is analysed in detail and presented [22].

In [10], the NSGA-II algorithm is used to optimize the problem of identifying code changes that create performance regression. The performance is compared with other classic algorithms, showing a promising research approach in Evolutionary Algorithms. A multi-objective evolutionary algorithm named diversity based Genetic Algorithm (DIV-GA) was proposed by improving the diversity of the search population and is found to be efficient in optimizing the test case selection problem [23].

A wide range of Evolutionary algorithms were applied to minimize the test suite size by characterizing a multi-objective optimization problem with two objectives: to maximize the effectiveness and minimize the cost [24]. NSGA-II and MOEA/D algorithms were used to optimally choose a subset of the regression test suite to achieve a trade-off between cost and coverage [25]. Regression test case prioritization and selection are essential during a software maintenance phase; an improvement to the existing Ant Colony Optimization algorithm is made, and the resulting algorithm is named Enhanced ACO_TCSP, resulting in minimal runtime and maximum coverage [26]. A hybrid Spider Monkey method-based optimization algorithm was proposed and applied to optimize the regression test suite to find the minimal test cases needed to perform regression testing [27]. Tetrad optimization techniques based on evolutionary algorithms, such as an ant colony and bee colony, and genetic and greedy approach optimization methods were used for the test case selection and prioritization problem related to regression testing [28]. The related works focused on existing methods for performance regression causing code change identification are given in Tabe 1.

**Table 1. Studies on performance regression identification**

| Ref. No./Year | Year | Method |
|---|---|---|
| [10] | 2022 | • Performance regression causing code modification identification<br>• Metrics are used to characterize the structural code properties like lines of code<br>• Multi-objective evolutionary algorithms like NSGA-II, SPEA2, and IBEA are used<br>• Git project commits used for experimentation<br>• IBEA outperforms other algorithms |
| [29] | 2020 | • Machine Learning Model to identify performance regression<br>• Boosted decision tree, decision forest and SVM are used, and SVM attained best results<br>• Static and dynamic indicators to describe structural characteristics are used<br>• Solutions evaluated using hit-and-dismiss rates<br>• Git Project commits are taken for experimentation |
| [30] | 2024 | • Early detection of performance regressions<br>• Initially, component level performance deviation is identified and later mapped to the architecture level.<br>• Finally, system-level performance regressions are evaluated<br>• Experimentation conducted using Tea store and Train ticket open-source systems |
| [31] | 2022 | • Automated performance regression detection<br>• Random forest classifiers are used<br>• Metrics are estimated through aspects like synchronization, loop, external call, etc.<br>• Experiments conducted using systems like Hadoop, Cassandra and openJPA |

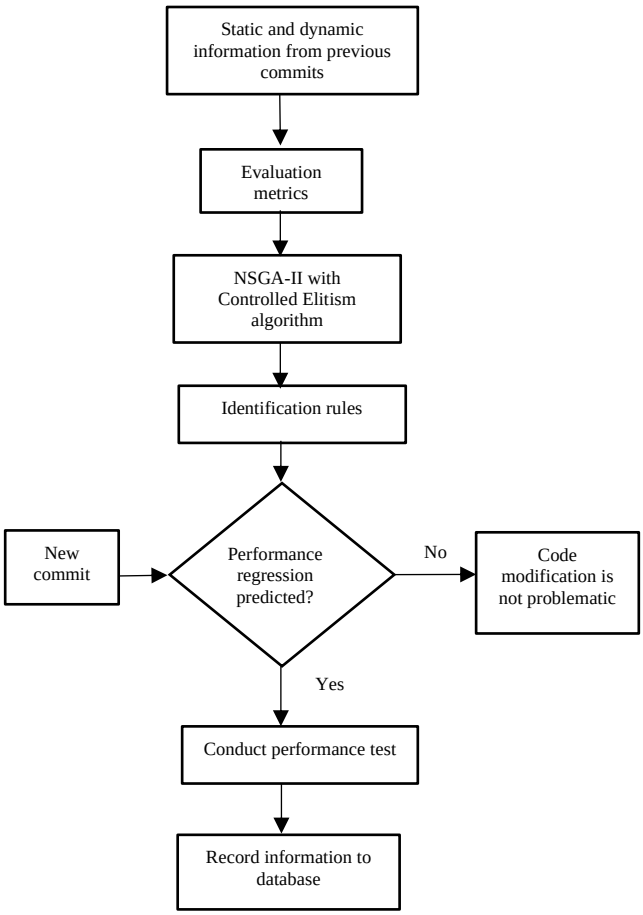



**Fig. 1 Flowchart MNSLS approach**

## 3. Proposed Method

The overall process of the research work is illustrated in Figure 1. Detailed discussion is provided in subsequent sections. From the database of previous historic commits (Git project), the static and dynamic information are extracted. This information is applied to the evaluation metrics, which are used in the formation of identification rules to identify the problematic code modifications/commits. The proposed MNSLS algorithm uses these evaluation metrics and generates the identification rules. For a new commit or code modification, this rule is used to identify whether it causes performance regression. If regression is identified, benchmarks will be used to assess the deviations. The updates are then stored back in the commits database. Through this process, the necessity to check for performance regression after every code modification could be avoided. The research problem is identifying the code modifications that cause performance regression instead of evaluating all the modified codes, saving time, effort and money. The problematic code modifications are identified using a set of static and dynamic metrics. These metrics measure the structural impact that code modifications create and are detailed in Section 3.1. An evolutionary algorithm-based approach to solving the above problem has been experimented and the steps are detailed in this section.

### *3.1. Evaluation Metrics Used for Profiling*

These metrics characterize and profile the code modifications that cause performance regression. There is a wide range of static and dynamic metrics available that are used to profile the code modifications [6][10]. The dynamic evaluation metrics used in the present research work are listed in Table 2. These evaluation metrics are indicators that predict the code modifications that cause performance regression. Static or Static and dynamic information are needed to derive the metric value.

The process of collecting static and dynamic information and applying it to evaluation metrics is performed as follows: first, the static and dynamic information is collected, and this step is detailed below. Next, the collected information is applied to the evaluation metrics listed in Table 1, and the prediction about the problematic code modifications is based on the hit and dismiss rate objective function values. The procedure to collect static and dynamic information [6] is briefed in this section and is illustrated in Figure 2. B1 and B2 represent sample benchmarks for a project. Dynamic information is collected whenever the benchmarks are executed against a commit. It is usually done when a performance change is predicted or execution happens at a prespecified interval.

Thus, dynamic information is not available for old commit2 with benchmark B2. Every new commit must be verified to determine whether it causes performance changes for the 2 benchmarks. To check whether the new commit affects the performance of benchmark B1, the static information from the new commit and dynamic information from old commit 2 are used.

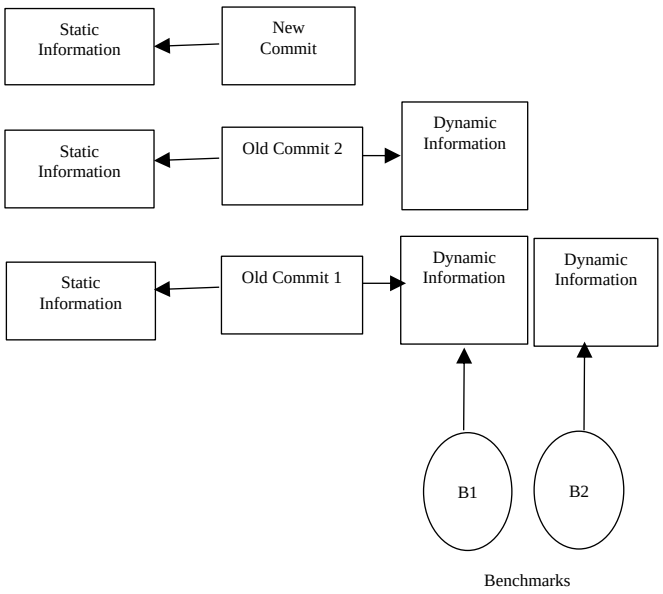



**Fig. 2 Static and dynamic information**

Similarly, to verify whether performance with respect to benchmark B2 changes due to the new commit, the static information of the new commit and static and dynamic information from old commit 1 are used. This is because dynamic information for old commit 2 with benchmark B2 is not available; thus, it is retrieved from the previous commit 1, for which the dynamic information of B2 is available. Thus, the static and dynamic information is collected through this process to predict performance change against a benchmark and decide to execute a benchmark test with the new commit.

### *3.2. Identification Rule*

The multi-objective optimization problem in this research work is the generation of identification rules that can successfully identify the code modifications that cause performance regression, and the problem is characterized by two objective functions, namely, hit and dismiss rates, respectively. The objective functions are detailed in Section 3.3.

Seven evaluation metrics detailed in the previous section form the search space with different associated values, subject to upper and lower bound limits and are represented as rules in the form of the tree. These identification rules are used to predict whether a new commit is problematic or not. If the commits are found to be problematic, a benchmark test is executed, and the static and dynamic information is observed to record the metric value, which is used to update the identification rule in future. An example rule is given below and the same is illustrated in Figure 3, in the form of a tree. The population of solutions used in the evolution process are given in the form of a tree [10]. ((((DM<18) OR (AM>38)) AND (PTCM>0.1%)) OR ((DMB<0.50) OR (PML>5%)))

### *3.3. Objective Functions*

The solutions attained using the MNSLS algorithm were evaluated using the following two objectives: hit rate and dismiss rate. These objective functions are described in this section.

**Table 2. Evaluation metrics**

| Metric | Description | Data |
|---|---|---|
| Number of methods that are deleted (DM) | Refactoring is the indicative reason for deleted methods, which may affect the performance. | Static |
| Number of methods that are added (AM) | The newly added functions may also affect the performance. | Static |
| Number of methods that are deleted and are reached in benchmark execution (DMB) | Refactoring is the indicative reason for deleted methods, and if they are reached in the benchmark execution process, it may affect the performance. | Static and Dynamic |
| The percentage of changed top-called methods (PTCM) | It implies that the percentage of altered top-level methods may affect performance. | Static and Dynamic |
| The percentage of changed top-called methods, by at least 10%, changes to its static instruction length (PTCM10) | It implies the percentage of top-called methods that are altered by 10% magnitude with respect to static instruction length, and these changes are of high risk. | Static and Dynamic |
| The percentage of static method length change (PML) | This metric implies the percentage of changes made to the static instruction length of a method, and if beyond the threshold value, are likely to degrade the performance | Static |
| The percentage of static method length change is called by benchmark (PMLB) | The description is the same as the above metric, and additionally, the function is called by the benchmark. | Static and Dynamic |

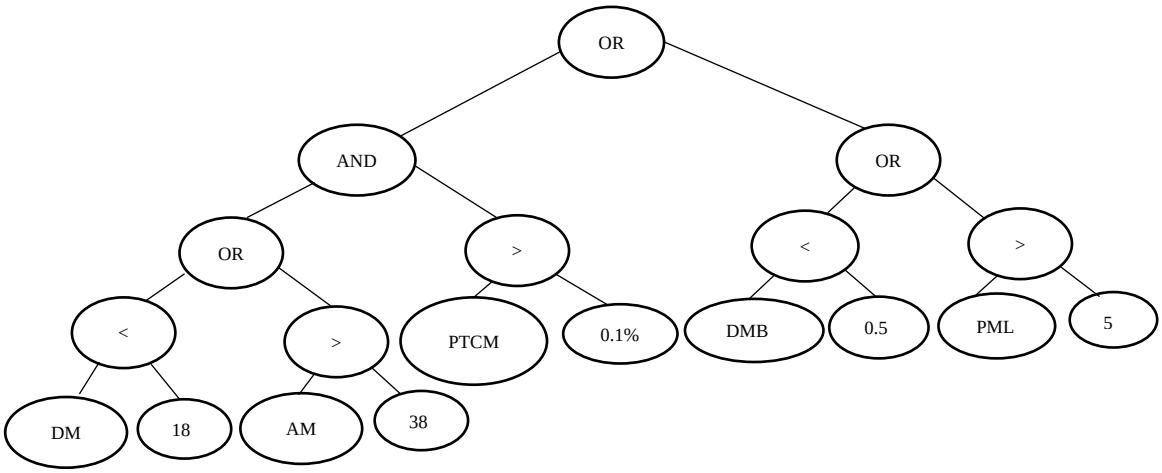



**Fig. 3 Identification rule - Tree representation**

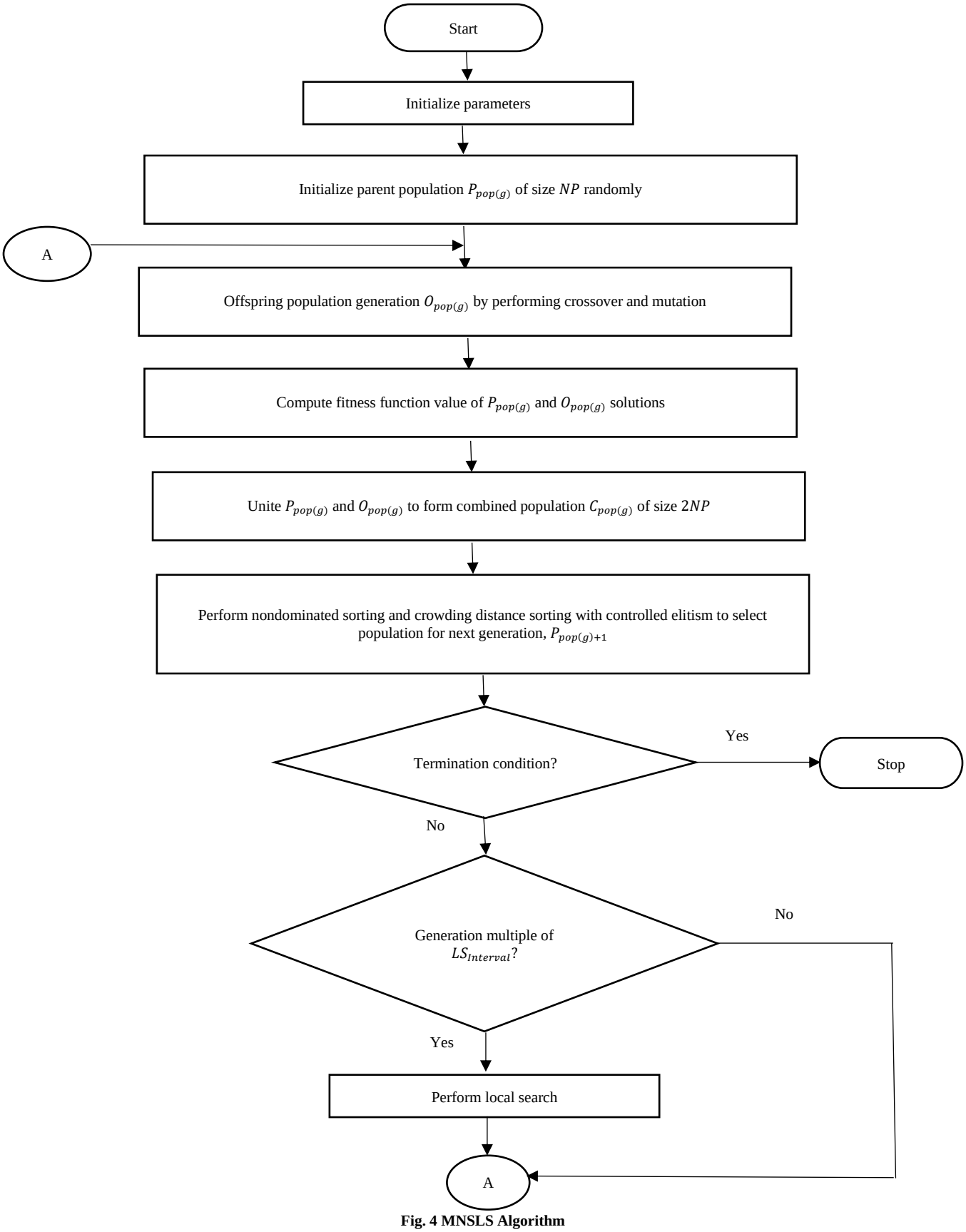



**Fig. 4 MNSLS Algorithm**

*3.3.1. Objective function 1 - Hit rate*

Let $H_T$ be the variable denoting the count of total commits, which causes performance regression. $H_D$ denotes the count of detected commits that cause performance regression. Hit rate gives the measurement value for the count of detected commits, which causes performance regression with respect to total commits, which causes performance regression. Equation 1 gives the hit rate calculation for a solution $sol$.

$$Hitrate(sol) = \frac{|H_D \cap H_T|}{H_T} \quad (1)$$

Hit values range between 0 and 1, and a hit value of 1 implies that all problematic commits are detected by the solution, which is attained through the evolutionary algorithm.

*3.3.2. Objective function 2 - Dismiss rate*

Let $D_T$ be the variable denoting the count of total commits that do not cause performance regression. $D_D$ denotes the count of detected commits that do not cause performance regression.Dismiss rate gives the measurement value for the count of detected non-problematic commits that do not cause performance regression with respect to total nonproblematic commits. Equation 2 gives the dismiss rate calculation for a solution $sol$.

$$Dismissrate(sol) = \frac{|D_D \cap D_T|}{D_T} \quad (2)$$

The dismiss rate value ranges between 0 to 1; a dismiss rate value of 1 implies that all nonproblematic commits are detected. An optimal solution will have both hit rate and dismiss rate values as 1, and finding one such solution or rule is difficult. Thus, the objective of the search process using the MNSLS evolutionary algorithm is to find solutions through simultaneous optimization of the conflicting objective functions, with a good trade-off between both hit rate and dismiss rates.

***3.4. Proposed MNSLS Method***

The flowchart of the proposed MNSLS method is illustrated in Figure 4.

*3.4.1. NSGA-II with Controlled Elitism*

Nondominated sorting genetic algorithm-II (NSGA-II) [12] is a modified and improved version of the NSGA algorithm with elitism preserving, nondominated-based ranking and crowding distance properties. The initial population of size $NP$ is randomly generated within the variable boundary conditions, and this parent population is represented as $P_{pop}$. The objective function value is calculated for all solutions. The population is sorted and arranged through their nondomination factor, and the solutions are assigned with a fitness value equal to their level of nondomination (fitness 1 represents the best level). The offspring population represented as $O_{pop}$ is generated through tournament selection, crossover and the mutation process. In the NSGA-II approach, parent and offspring populations $P_{pop}$ and $O_{pop}$ are combined together and are represented as $C_{pop}$ ($C_{pop} = P_{pop} + O_{pop}$) of size $2NP$. The new population for the next generation is chosen through nondominated sorting and crowding distance sorting methods. Though the NSGA-II algorithm is widely adapted to solve numerous optimization problems, there are a few factors that could be addressed to further leverage the search standards like, uncontrolled elitism could be restricted. Maintaining diversity lateral to the pareto front and accelerating the algorithm's convergence speed are a few additions that can contribute to a better search and optimize process. A diversity of solutions along and lateral to the non-dominated front is needed to have good convergence. The controlled Elitism technique [32] improves lateral diversity, as detailed below. As discussed above, maintaining lateral diversity with respect to the non-dominated front or pareto front is vital and helps overcome excessive exploitation. When this lateral diversity is lost, it affects the search process and leads to exploitation in the region of current promising solutions. This is depicted in Figure 5 [32]. The controlled elitism technique controls the rate of exploitation over exploration by adaptively restricting the count of solutions chosen from the best front (nondominated set of solutions). A geometric distribution restricts the number of solutions chosen from a front, ensuring solutions across all the fronts are selected, thus controlling elitism.

The geometric distribution is given by,

$$C_i = C\frac{1-r}{1-r^F}r^{i-1} \quad (3)$$

Variable $C_i$ represents the count of solutions to be selected from the $i^{th}$ front, where ($i = 1,2,..F$). Variable $r$ denotes the reduction rate, which is a user-specified value and is less than 1 ($r < 1$). This value range ensures solutions selected from the first front (best-nondominated front) are higher, and from the subsequent fronts, the count of solutions selected is exponentially reduced. $C$ represents the count of solutions to be selected in total.

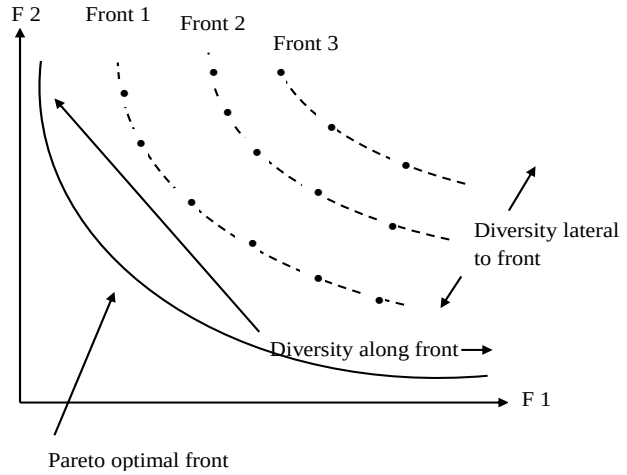



**Fig. 5 Controlled elitism**

Algorithm 1- Improved and Controlled Local Search

1 Initialize $LS\_Interval$
2 **if** $(Generation\%LS_{Interval} = 0)$ **then**
3 **for** $i \in \{1, 2, \ldots, F\}$
4 $CS \leftarrow select - random - solutions(i)$
5 $neighbour_{solution} = Local\_Search(CS)$
6 $Evaluate(neighbour_{solution})$
7 $FirstFront.update(neighbour_{solution})$
8 **return** $FirstFront$
9 $functioneval = functioneval + count\_of\_Localsearch\_Iteration$
10 **Endif**

**Fig. 6 Improved and controlled local search**

*3.4.2. Local Search*

To accelerate and improve the convergence speed of the algorithm, a local search is performed and is detailed subsequently. Local search techniques are known to improve the convergence properties by intensifying the search to find better solutions in the proximity region of promising solutions. These methods exploit the promising solutions attained so far, refine them, and optimize them further to improve fitness. The general practice is to choose solutions from the non-dominated front and to perform local search on these solutions to identify a better one. The proposed improved and controlled local search differs from the existing techniques by choosing candidate solutions from all available fronts to perform local search. The algorithm for local search is given in Figure 6.

For instance, if there are $F$ non-dominated fronts available in a generation as an outcome of the NSGA-II algorithm, one solution from each front is selected to perform local search in order to identify a better neighbourhood, the candidate solutions selected to perform local search are stored in set $CS$. Next, a direct search-based local search method called Nelder-Mead [33] is used to search the proximity of the neighbourhood regions of these candidate solutions to identify a fitter solution. A solution $neighbour_{solution}$ that is identified through local search is evaluated and compared with all the solutions in the first non-dominated front, and if it dominates any solution in this front, then $neighbour_{solution}$ is added, and the dominant solution is removed. The $neighbour_{solution}$ is also added to the first non-dominated front if it is not dominated by any solution in this front. In either case, a fitter solution is included in the first/best non-dominated front, thus exploiting the potential solutions and further improving the search process. Since the local search method increases the search cost, and to avoid exploitation at a high rate, local search is performed in regular intervals ($LS\_Interval$), which makes it a controlled local search method. This improved and controlled local search method contributes to enhancing the rate of convergence of the proposed MNSLS algorithm.

## 4. Experimentation and Results

The parameter settings, performance indicators and the algorithms chosen for comparison are all detailed in this section. The open-source Git project is used for performance evaluation. Performance test bookmarks exist for the same, spanning across different projects. The database of around 8596 Git project commits, as listed in the PRICE approach [10], is used for evaluation. All the commits are taken for experimentation.

### *4.1. Parameter settings*

The parameters in the proposed MNSLS algorithm and their associated values are listed in Table 3.

**Table 3. Algorithm parameters and their value**

| Parameter | Value |
|---|---|
| Population size ($NP$) | 100 |
| Mutation rate | 0.2 |
| Crossover rate | 0.8 |
| Number of iterations | 10000 |
| Runs | 30 |
| Local search Interval ($LS\_Interval$) | 10 |
| Local search number of iterations ($LS_{Itcount}$) | 100 |

The threshold limits of the evaluation metrics, which form the variables of the search algorithm used to find the optimal identification rule, are set at the value as given in [10] and are listed in Table 4.

**Table 4. Evaluation metrics and their threshold value**

| Evaluation metric | Threshold value |
|---|---|
| Number of methods that are deleted (DM) | 20 |
| Number of methods that are added (AM) | 44 |
| Number of methods that are deleted and are reached in benchmark execution (DMB) | 0.553 |
| The percentage of changed top-called methods (PTCM) | 0.597% |
| The percentage of changed top-called methods, by at least 10%, changes to its static instruction length (PTCM10) | 30% |
| The percentage of static method length change (PML) | 500% |
| The percentage of static method length change is called by benchmark (PMLB) | 14% |

### *4.2. Performance Metric- Hypervolume*

Hypervolume (HV) [34, 35] is a performance metric used to analyze and compare the performance of the algorithms. HV metric assesses the convergence and diversity properties of the attained non-dominated solution set and does not require a true front for evaluation.

HV gives the area between the attained non-dominated front and a reference point. The reference points used include the worst objective function value from the attained solution set for each objective. The higher the HV value, the better the solution quality.

### *4.3. Nonparametric Statistical Test- Mann-Whitney U Test*

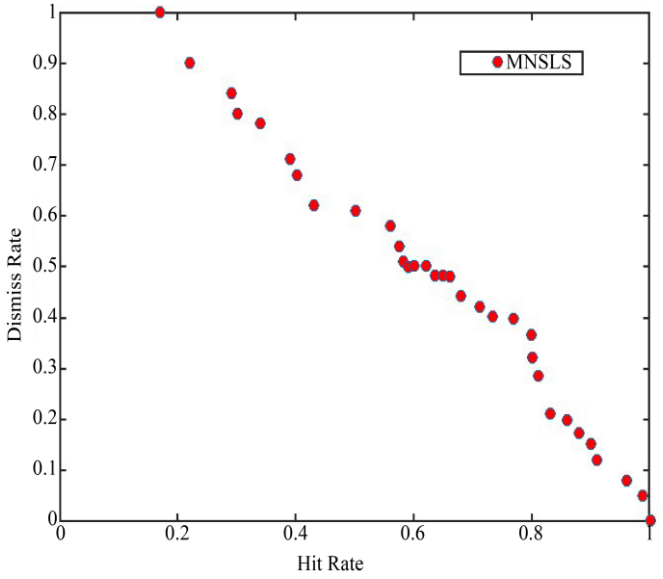
To further strengthen the findings, a statistical test is conducted between the algorithms taken for comparison. Mann-Whitney *U* test [36] is used for this purpose and is also called the Wilcoxon Rank Sum Test. It is a pair pairwise comparison test and interprets if there is any significant difference between the population group. The null hypothesis is that there is no significant difference between the two populations, and they are equal.

The $p$-value less than 0.05 indicates a significant difference between the algorithms to rule out the null hypothesis. The MNSLS algorithm is compared with three other state-of-the-art algorithms, NSGA-II [12], SPEA2 [12], PESA-II [15] and IBEA [34] algorithms. A 10-fold cross-validation is used, with nine folds for training and one-fold for testing, making a total of 300 runs per algorithm.

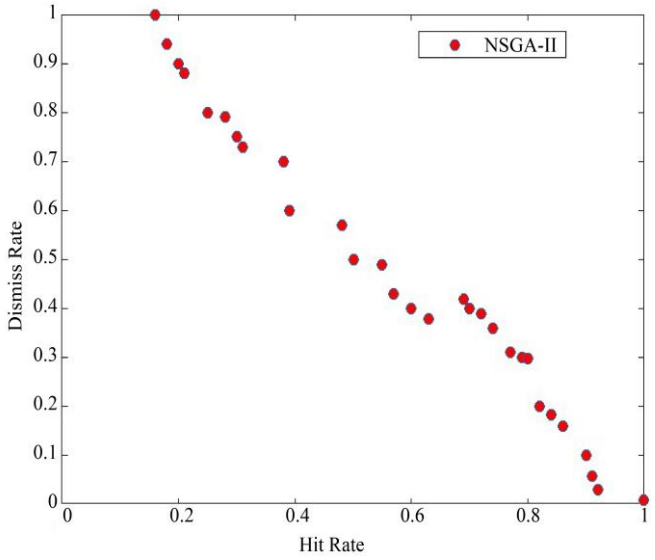
### *4.4. Results*

The pareto fronts obtained by MNSLS and the other algorithms taken for comparison are given in Figure 7. The attained front using the proposed MNSLS method, as shown in Figure 7(a), is promising and evident through the following statistical test results. The statistical test results are given in Table 5, which gives the obtained $p$-value using the Mann-Whitney *U* test.

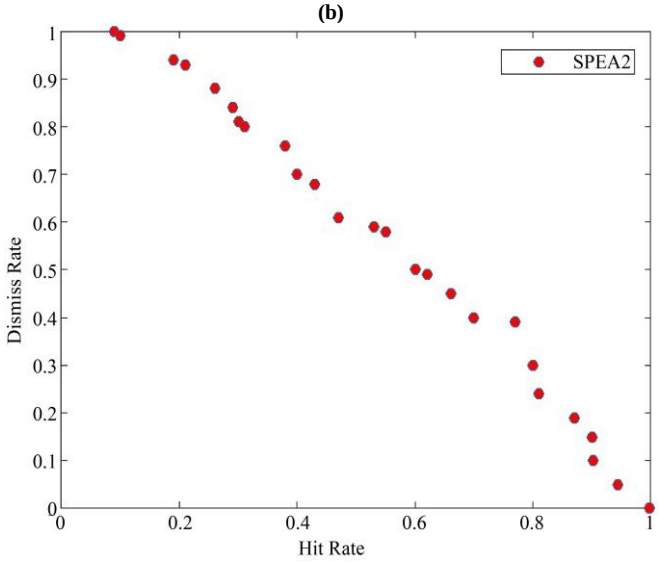
From the statistical test results, it can be observed that the $p$-value of MNSLS, when compared to all three other algorithms, is less than 0.05, which shows there is a significant difference between the algorithms. Even the results of other algorithms have significant differences except for SPEA2 and PESA-II algorithms, as there is no significant difference in performance between them as the $p$-value is 0.06, which is greater than the threshold value of 0.05.



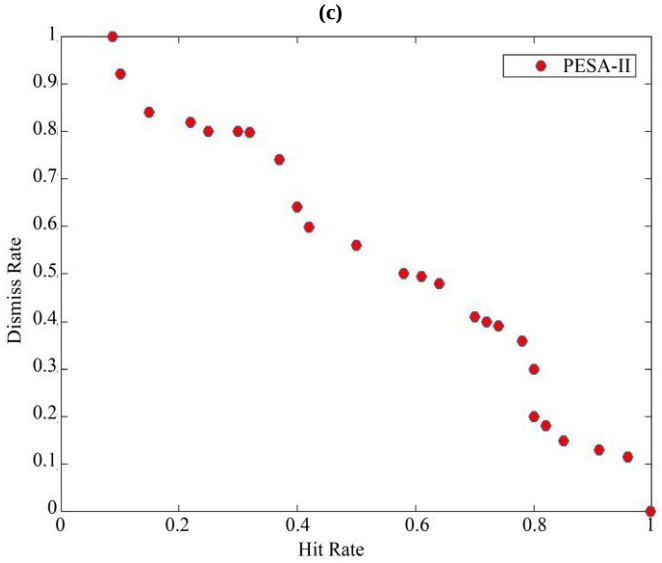

**(a)**



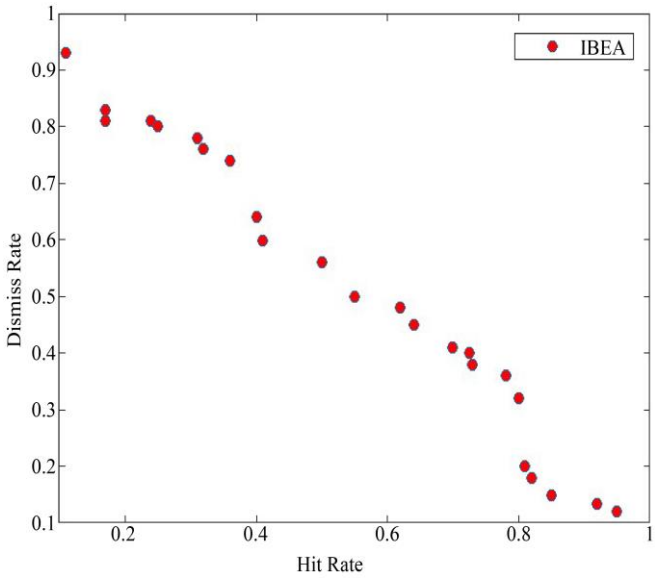

**(b)**




**(c)**




**(d)**




**(e)**

**Fig. 7 Obtained pareto fronts (a) MNSLS, (b) NSGA-II, (c) SPEA2, (d) PESA-II, (e) IBEA.**

**Table 5. Mann-Whitney *U* statistical test results using HV metric**

| | **Algorithm B** | | | |
|---|---|---|---|---|
| **Algorithm A** | **NSGA-II** | **SPEA2** | **PESA-II** | **IBEA** |
| **IMP-NSGA-II** | 0.021 | 5.98E -09 | 7.8E -15 | 2.17E-16 |
| **NSGA-II** | | 2.59E -07 | 2.3E -16 | 2.19E-16 |
| **SPEA2** | | | 0.06 | 2.23E-16 |
| **PESA-II** | | | | 2.18E-16 |

**Table 6. Evaluation metric values attained vs. actual**

| | **PMLB** | **PML** | **PTCM10** | **PTCM** |
|---|---|---|---|---|
| **Identification rule value** | <1.1% | <0.25% | >28 | <0.1 |
| **Actual value for commit** | 0.063% | -0.024% | 115% | -0.015% |

The evaluation metric values attained using the MNSLS algorithm are analysed to further strengthen the results. Commit 0719f3ee is taken for this purpose, the obtained evaluation metric value through MNSLS and the actual Commit metric value [10] are presented in Table 6, and it can be seen that the predicted value for all the evaluation metrics is appropriate and shows that the code modifications are problematic.

### *4.5. Comparison with Other Regression Identification Techniques*

It is evident from the results discussed in Section 3.4 that the proposed MNSLS algorithm best solves the multi-objective optimization problem, which includes hit rate and dismiss rate as objectives used to detect the code modification that causes performance regression. The proposed method of regression identification using MNSLS is further compared with another evolutionary optimization-based regression identification approach called PRICE [10]. This helps to strengthen the results of the proposed approach. PRICE approach represented using IBEA as the evolutionary algorithm is taken for comparison. As discussed in the previous section, 10-fold cross-validation is used, where nine-fold is used for training, and one-fold is used for testing. The rule generated using the training dataset is used for evaluation during testing. The approaches were compared using the hit rate and dismiss rates.

The comparison results with respect to hit rate and dismiss rate between the approaches are presented in Figure 8a and Figure 8b, respectively. The hit rate of the proposed MNSLS algorithm-based approach ranges from 60% to 100%, whereas for PRICE, it is about 48% to 100%. Thus, the hit rate is higher with the proposed approach. The dismiss rate of the proposed approach varies from 15% to 98%, and for the Price approach, it is about 17% to 97%; on average, the dismiss rate of the proposed MNSLS algorithm-based approach is better.

To further strengthen the claim, the precision, recall, F1 and AUC scores are calculated using hit and dismiss rates. The results of the above metrics using the proposed MNSLS and PRICE [10] approach are given in Table 7.

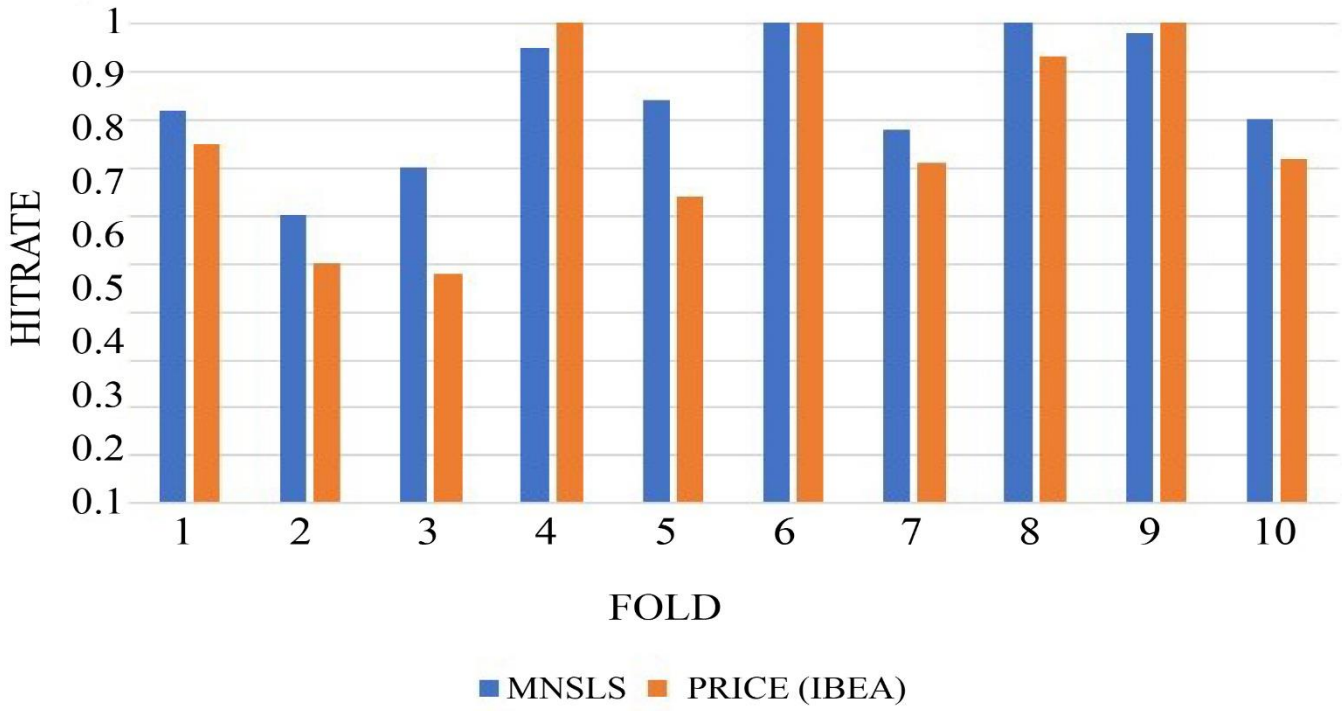



**Fig. 8 (a) Hit rate of MNSLS vs. PRICE**

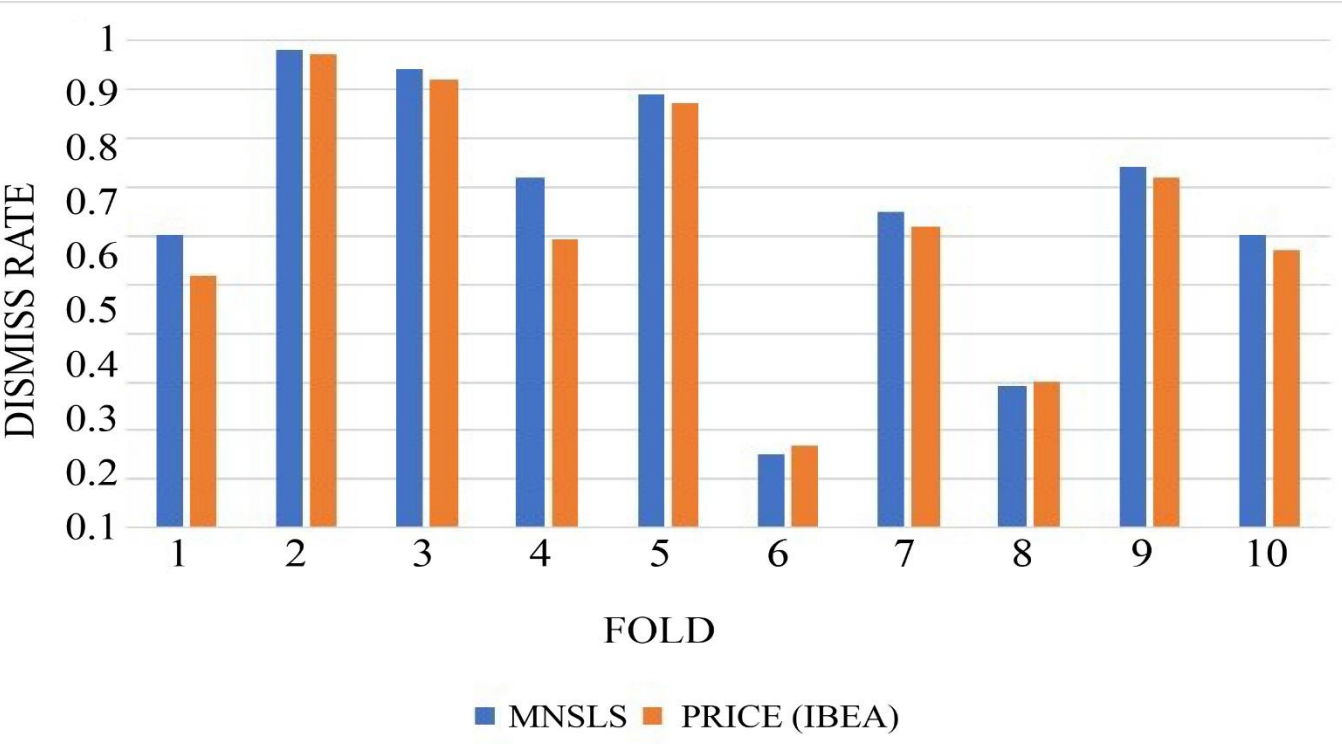



**Fig. 8 (b) Dismiss rate of MNSLS vs. PRICE**

**Table 7. Results of MNSLS approach with different metrics, comparing with PRICE**

| | Pre. | Recall | F1 | AUC |
|---|---|---|---|---|
| **MNSLS** | 0.89 | 0.84 | 0.90 | 0.87 |
| **PRICE** | 0.81 | 0.76 | 0.79 | 0.78 |

From the results, it is evident that the proposed MNSLS approach identifies the problematic codes better when compared to the other approach taken for comparison. The results using the hypervolume metric and statistical tests prove the same as well. The proposed memetic algorithm using the controlled local search procedure aids in better exploitation by finding better solutions in the proximity of promising solution regions at regular intervals, whereas the other existing evolutionary algorithm-based methods applied to regression identification problems are based on a global search method or algorithms which is not adapted to this problem. The proposed MNSLS algorithm based on both global and local search approaches attains better solutions, resulting in better identification.

### *4.6. Limitations*

The proposed method identifies code changes that occur in functions. However, considering the code changes in statements will make the detection model more efficient; the same is the scope for future work. The generalization aspect of the proposed work has practical implications to be considered, and the same is true of the scope of future work. Thus, applying the proposed method to other projects will give better results concerning real-world problems.

## 5. Conclusion

In this research work, identifying the code modifications that are problematic and cause performance regression using the evolutionary algorithm is presented. Evaluation metrics that characterize the code modifications act as the input variables for the algorithm. The identification rule that predicts the problematic code modifications is defined as the search problem with two objectives: hit rate and dismiss rate.

Deriving this identification rule through an evolutionary algorithm is the major research objective. Memetic algorithm using NSGA-II and Local Search (MNSLS) is proposed. In MNSLS, the NSGA-II algorithm with controlled elitism is used as a global search algorithm. New, improved and controlled local search techniques are used to enhance the search performance and attain pareto optimal solutions that form the best trade-off between the two objectives.

Commits from the Git project are used as the dataset for experimentation. MNSLS algorithm is compared with three other state-of-the-art evolutionary algorithms. Hypervolume metric is used as the performance indicator to evaluate the quality of the solutions obtained and further, used to compare the performance between the algorithms. To strengthen the findings, the Mann-Whitney *U* test, which is a nonparametric statistical test, is conducted.

To further investigate the performance of the overall approach for performance regression identification using the MNSLS algorithm, a comparative study is conducted between the MNSLS-based regression identification approach and the IBEA algorithm-based PRICE approach for regression identification. Through the results obtained using the above experimentation, it is evident that the proposed technique results are better than those of the other evolutionary algorithms and regression detection approach, and the identification rules that are attained appropriately classify and identify the problematic code modifications.

This research could further be expanded to analyse the effectiveness of evaluation metrics in identifying problematic commits. This work will be performed in the future by including several other metrics and studying the contribution of various metrics. Another extension is to analyse the performance of the proposed evolutionary algorithm-based regression identification with other machine learning-based models used for regression identification. Further, extending and conducting experimentation with other projects improves the generalization aspect of the identification model.

## References

[1] Jinfu Chen, and Weiyi Shang, "An Exploratory Study of Performance Regression Introducing Code Changes," *2017 IEEE International Conference Software Maintenance and Evolution (ICSME)*, Shanghai, China, pp. 341-352, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[2] King Chun Foo et al., "Mining Performance Regression Testing Repositories for Automated Performance Analysis," *Proceedings of the IEEE 10th International Conference on Quality Software*, Zhangjiajie, China, pp. 32-41, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[3] Peng Huang et al., "Performance Regression Testing Target Prioritiza-Tion Via Performance Risk Analysis," *Proceedings of the ACM 36th International Conference on Software Engineering*, New York, United States, pp. 60-71, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[4] Michael Pradel, Markus Huggler, and Thomas R. Gross, "Performance Regression Testing of Concurrent Classes," *Proceedings of the International Symposium on Software Testing and Analysis*, New York, United States, pp. 13-25, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[5] Shadi Ghaith et al., "Profile-Based, Load-Independent Anomaly Detection and Analysis in Performance Regression Testing of Software Systems," *2013 17th European Conference on Software Maintenance and Reengineering*, Genova, Italy, pp. 379-383, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[6] Augusto Born De Oliveira et al., "Perphecy: Performance Regression Test Selection Made Simple but Effective," *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, Japan, pp. 103-113, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[7] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund, "Accurate Modeling of Performance Histories for Evolving Software Systems," *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, CA, USA, pp. 640-652, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[8] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente, "Prioritizing Versions for Performance Regression Testing: The Pharo Case," *Science of Computer Programming*, vol. 191, pp. 1-25, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[9] M. Harman, "Making the Case for MORTO: Multi Objective Regression Test Optimization," *IEEE Fourth International Conference on Software Testing*, Berlin, Germany, pp. 111-114, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[10] Deema Alshoaibi et al., "Search-Based Detection of Code Changes Introducing Performance Regression," *Swarm Evolutionary Computation*, vol. 73, pp. 1-19, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[11] Mohamed Wiem Mkaouer et al., "High Dimensional Search-Based Software Engineering: Finding Tradeoffs Among 15 Objectives for Automating Software Refactoring using NSGA-III," *Proceedings of Annual Conference on Genetic and Evolutionary Computation*, New York, United States, pp. 1263-1270, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[12] K. Deb et al., "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182-197, 2002. [CrossRef] [Google Scholar] [Publisher Link]

[13] Eckart Zitzler, MarcoLaumanns, and Lothar Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm," *ETH Zurich, Computer Engineering and Networks Laboratory*, vol. 103, pp. 1-22, 2001. [CrossRef] [Google Scholar] [Publisher Link]

[14] S. Brindha, and S. Miruna Joe Amali, "A Robust and Adaptive Fuzzy Logic Based Differential Evolution Algorithm using Population Diversity Tuning for Multi-Objective Optimization," *Engineering Applications of Artificial Intelligence*, vol. 102, pp. 1-14, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[15] David W. Corne et al., "PESA-II: Region-Based Selection in Evolutionary Multiobjective Optimization," *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, San Francisco, CA, United States, pp. 283-290, 2001. [Google Scholar] [Publisher Link]

[16] Vimal Savsani, and Mohamed A. Tawhid, "Non-Dominated Sorting Moth Flame Optimization (NS-MFO) for Multi-Objective Problems," *Engineering Applications of Artificial Intelligence*, vol. 63, pp. 20-32, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[17] Qingfu Zhang, and Hui Li, "MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712-731, 2007. [CrossRef] [Google Scholar] [Publisher Link]

[18] W.E. Hart, N. Krasnogor, and J.E. Smith, *Memetic Evolutionary Algorithms*, Recent Advances in Memetic Algorithms, pp. 3-27, 2005. [CrossRef] [Google Scholar] [Publisher Link]

[19] Sezin Afsar et al., "Multi-Objective Enhanced Memetic Algorithm for Green Job Shop Scheduling with Uncertain Times," *Swarm and Evolutionary Computation*, vol. 68, pp. 1-14, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[20] M. Manikandan, S. Sakthivel, and V. Vivekanandhan, "Efficient Clustering using Memetic Adaptive Hill Climbing Algorithm in WSN," *Intelligent Automation and Soft Computing*, vol. 35, pp. 3169-3185, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[21] Tansel Dokeroglu, and Ender Sevinc, "Memetic Teaching-Learning-Based Optimization Algorithms for Large Graph Coloring Problems," *Engineering Applications of Artificial Intelligence*, vol. 102, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[22] Ruchika Malhotra, Megha Khanna, and Rajeev R. Raje, "On the Application of Search-Based Techniques for Software Engineering Predictive Modeling: A Systematic Review and Future Directions," *Swarm and Evolutionary Computation*, vol. 32, pp. 85-109, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[23] Annibale Panichella et al., "Improving Multi-Objective Test Case Selection by Injecting Diversity in Genetic Algorithms," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 358-383, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[24] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb, "Cost-Effective Test Suite Minimization in Product Lines Using Search Techniques," *Journal of Systems and Software*, vol. 103, pp. 370-391, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[25] Wei Zheng et al., "Multi-Objective Optimisation for Regression Testing," *Information Sciences*, vol. 334-335, pp. 1-16, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[26] Shweta Singhal et al., "Multi-Objective Fault-Coverage Based Regression Test Selection and Prioritization Using Enhanced ACO_TCSP," *Mathematics*, vol. 11, no. 13, pp. 1-21, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[27] Arun Prakash Agrawal, Ankur Choudhary, and Parma Nand, "An Efficient Regression Test Suite Optimization Approach Using Hybrid Spider Monkey Optimization Algorithm," *International Journal of Swarm Intelligence Research*, vol. 12, no. 4, pp. 57-80, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[28] Shweta Singhal et al., "Empirical Evaluation of Tetrad Optimization Methods for Test Case Selection and Prioritization," *Indian Journal of Science and Technology*, vol. 16, pp. 1083-1044, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[29] Max Mendelson, "*Identifying Performance Regression from The Commit Phase Utilizing Machine Learning Techniques*," Master Thesis, Rochester Institute of Technology, Rochester, USA, 2020. [Google Scholar] [Publisher Link]

[30] Lizhi Liao et al., "Early Detection of Performance Regressions by Bridging Local Performance Data and Architecture Models," *Arxiv*, pp. 1-13, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[31] Jinfu Chen, Weiyi Shang, and Emad Shihab, "PerfJIT: Test-Level Just-in-Time Prediction for Performance Regression Introducing Commits," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1529-1544, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[32] Kalyanmoy Deb, *Salient Issues of Multi-Objective Evolutionary Algorithms*, Multi-Objective Optimization using Evolutionary Algorithms, Wiley, pp. 414-417, 2001. [Google Scholar]

[33] J.A. Nelder, and R. Mead, "A Simple Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308-313, 1965. [CrossRef] [Google Scholar] [Publisher Link]

[34] Eckart Zitzler, and Simon Künzli, "Indicator-Based Selection in Multiobjective Search," *Parallel Problem Solving from Nature-PPSNVIII*, pp. 832-842, 2004. [CrossRef] [Google Scholar] [Publisher Link]

[35] Johannes Bader, and Eckart Zitzler, "HypE: An Algorithm for Fast Hypervolume-Based Many-Objective Optimization," *Evolutionary Computation*, vol. 19, pp. 45-76, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[36] G.W. Corder, and D.I. Foreman, *Comparing Two Unrelated Samples: The Mann-Whitney U-Test and the Kolmogorov-Smirnov Two-Sample Test*, Nonparametric Statistics: A Step-by-Step Approach, Wiley, pp. 71-80, 2014. [Google Scholar]