Training a Machine-Learning Model for Optimal Fitness Function Selection with the Aim of Finding Bugs

Ivan Stranski
Supervisors: Mitchell Olsthoorn, Pouria Derakhshanfar, Annibale Panichella
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

ABSTRACT

Software testing is essential for a successful development process, however, it can be troublesome as manually writing tests can be time demanding and error-prone. EvoSuite is a test case generating tool developed to address this [18]. It can generate test cases for different test criteria - Line Coverage, Branch Coverage, Input Diversity, etc. Branch Coverage puts the focus on covering branches in the code, whilst Input Diversity puts the focus on the use of diverse inputs as parameters in the test cases. The downside is that the user needs to select the best suited test criteria, out of the many that EvoSuite provides, for the classes under test. It is not feasible for the user to manually find the optimal one for the classes under test. This paper aims to shine a light on the effectiveness of the combination of Input Diversity and Branch Coverage as a test criteria. This study presents a machine learning technique to automatically select the best combination of test generation objectives according to static metrics. The model we chose for this task is a decision tree as it directly provides a pattern. Said pattern is a combination of conditions that the static metrics need to hold for the chosen test criteria to be effective. The evaluation of the effectiveness was done one a benchmark of 346 classes taken from SF-110 Corpus of Classes [9] and the Appache Commons. To evaluate the effectiveness of Input Diversity in combination with Branch Coverage, we compared the test criteria to two other test criteria - Branch Coverage and the Default coverage criteria used in EvoSuite. The decision tree models created achieve an accuracy upwards of 90% in the best case and deem metrics such as *wmc*, *dit*, *fanin* and others to be crucial for the effectiveness of Input Diversity in combination with Branch Coverage.

1 INTRODUCTION

Software quality testing plays a key role in the software development process [7]. In the recent years, researchers proposed various automated test generation approaches and tools to help developers in the time taking testing tasks. Research shows that the automation of test generation significantly reduces testing costs [7]. Common approaches towards automation include search-based testing, random testing and symbolic execution [1]. All approaches aim to satisfy certain objectives. Fitness functions are different ways to measure how close a test case is to fulfilling an objective [18]. Search-based test generation relies on fitness functions to guide the optimization of a problem - in our case the generation of test cases [1]. The problem is presented as a search space consisting of combinations of test cases and the goal is to find a test case for each objective such that specified fitness functions or heuristics are satisfied [16]. EvoSuite [8] utilizes a multi-objective search-based technique and is a cutting-edge tool used for the automation of generating unit-level tests for Java [14]. The tool uses a genetic algorithm which is a part of a larger group of evolutionary algorithms [9]. As a unit test generation tool, it gets one class as class under test and also gets a set of testing objectives [8]. Then, it starts a genetic approach guided by fitness functions reflecting the given test objectives to guide the search process towards generating tests covering the given goals [8]. A study done by Almasi et al. showed that EvoSuite has an advantage in terms of fault detection capabilities on Randoop - a test case generating tool that utilizes random testing [1]. EvoSuite provides a multitude of different fitness functions. The user can freely specify which fitness function or combination of fitness functions is to be used when generating the test cases for the classes under test. Some of the provided fitness functions are Line Coverage, Branch Coverage, Input Diversity, etc. The Default fitness function that is used in EvoSuite is a combination of Line, Branch, Output, Weak Mutation, CBranch, Exception, Methods without Exceptions and Method Coverage. The problem that is going to be tackled in this research paper is namely the effectiveness of fitness functions of EvoSuite when it comes to guiding the search process towards finding bugs. EvoSuite has many different fitness functions, but it is unclear which is the best option available. Different classes under test pose different characteristics - static and dynamic metrics. Hence, a class with little to no branches may benefit little in comparison to a class with a lot of branches when Branch Coverage is used as a fitness function. Fitness functions have a big effect on performance and therefore it is beneficial to know under which circumstances we should enable which types of fitness functions. It is important to mention that we include Branch Coverage by default, as it provides us with the ability to meaningfully cover more code and therefore it greatly improves structural coverage [2].

EvoSuite still isn't automated to decide what fitness functions are appropriate for the classes under test and it is still up to the user to decide this through trial-and-error. Therefore this paper seeks to provide clarity on how effective the different state-of-the-art fitness functions are at guiding the search process towards finding bugs and higher structural coverage. More specifically, the goal of the paper is to investigate in which cases it is useful to have the combination of Input Diversity and Branch Coverage as a fitness function. We have selected Input Diversity as research shows that diversification of input yields better result than having similar input [2].

The question that will be answered in this paper is when and how does Input Diversity affect the number of bugs detected when combined with Branch Coverage. In favor of answering the question substantially, it will be broken down into three sub-questions:

- RQ 1: How does Input Diversity in combination with Branch Coverage affect structural coverage when compared to Branch Coverage and Default fitness function?
- RQ 2: How does Input Diversity in combination with Branch Coverage affect fault detection when compared to Branch Coverage and Default fitness function?
- RQ 3: How does the time budget affect structural coverage and fault detection when Input Diversity is used in combination with Branch Coverage?

The analysis was done on a benchmark of 346 classes taken from SF-110 Corpus of Classes [9] and the Apache Commons. To answer the first two questions we computed the static metrics of the classes under test. Then, we ran a Decision Tree Classifier with feature selection to find patterns between the static metrics and the branch coverage results and fault detection capabilities respectively. To

answer the third question we computed the box plots that were based on the distribution of results of each fitness function for each time budget respectively. Furthermore, we computed the averages of each fitness function for each time budget. Then we did a time-wise comparison of both box plots and averages time.

The decision tree models achieved in the best case accuracy upwards of 90%. The patterns present in the decision trees deem that the important metrics are *wmc*, *fanin*, *cbo*, *dit*, *totalMethodsQty*, *fanout*, *privateMethricsQty*, *protectedMethodsQty* and *cboModified*. We found that increasing the time budget from 60 to 180 seconds leads to a 4.74% increase in the average branch coverage achieved by Input Diversity with Branch Coverage, while increasing the time budget from 180 to 300 seconds leads to an increase of only 0.1%. Overall, on average for branch coverage Input Diversity with Branch Coverage works better than the Default fitness function, but not better than Branch Coverage. For mutation scores on average Input Diversity with Branch Coverage works better than Branch Coverage, but not better than the Default fitness function.

The rest of the paper is structured in the following way. Section 2 talks about background of the research and related works. Section 3 talks in depth about the approach taken towards answering the research questions. Section 4 outlines and discusses the results from the research and analysis done. Section 5 discusses alternative ways to approach the problem at hand and why certain decisions were made. Section 6 outlines what we have done to mitigate possible threats to validity. Section 7 discusses the ethical aspects and the reproducibility of the research. Section 8 concludes the paper and talks about future works.

2 BACKGROUND AND RELATED WORKS

EvoSuite. EvoSuite is a test case generating tool that use a genetic algorithm to implement a search-based approach towards the generation of test cases [2, 18]. A genetic algorithm is an evolutionary algorithm. Furthermore, it is a stochastic algorithm, meaning that it is non-deterministic and that there is some randomness involved, which could result in different solutions [12]. The difference between a coverage criteria and a fitness function is that the criteria is an objective we are aiming to fulfil and the fitness function is the measurement that shows how close we are to fulfilling this goal [18]. The algorithm starts with a random set of combinations of test cases and evaluates them based on a fitness function [2]. The selecting process discards the test cases with worse fitness function [2]. The set that is left is refactored through crossover and mutation based on the result from the selection and is used for the next round of evolution [2]. The process continues until either all of the objectives are satisfied or the given search budget is passed [2]. Selecting which solutions move on is done with a selection technique, usually roulette wheel, that mimics evolution in the real world [12]. After selecting the optimal sub-solutions the new set of solutions is created through a crossover operation, where parts of different solutions are interchanged within one another [12]. Lastly, a mutation operation can be performed depending on a predefined probability, where a part of the solution is changed [12]. Thus, the new set of solutions is created and re-inserted as the new population of solutions [12].

We will use EvoSuite to generate test cases for the classes under test. These test cases will grant us branch coverage and mutation score results that we will analyse in our study.

Fitness Functions. In our research we will use three different coverage criteria - Input Diversity with Branch Coverage, Branch Coverage and the Default fitness function. The fitness function of Branch Coverage measures the distance a test case is from covering a branch [17]. The fitness function of Input Diversity measures how diverse the input used for testing is [17].

Code Metrics. We need code metrics to have better insight about class under test in order to select the best objective. In our research we have chosen to use the Chidamber-Kemerer code metrics tool developed by Aniche [3]. It is an analytical tool that computes the code metrics for a chosen set of classes. Code metrics are a way of measuring the quality of the produced code on various criteria, thus, giving insight into what needs to refactored [5]. The CK tool provides the original CK code metrics - *wmc*, *dit*, *noc*, etc. - and is extended with other code metrics totaling 49 code metrics [3].

Decision Tree. A decision trees is a type of supervised learning model. A decision tree is a chain of tests that check whether a numeric value fulfills certain conditions [11]. The model posses an irreplaceable advantage over black-box models - namely understandability, as it provides an easy to follow and logical representation of the inner workings of the model [11].

In our study we are going to use decision trees to find patterns in the static metrics of the classes under test that coincide with increase in either branch coverage or mutation score results. We are going to use specifically decision trees as they will provide us directly with the needed patterns. Said patterns will be embedded in the structure of the trees as conditional statements.

WSA. A whole suite approach is an approach that instead of optimising one test at a time, it optimises complete test suites at a time [16]. The Whole Suite with Archive approach utilizes an archive [19]. During test case generation, the algorithm takes note of test objectives that are covered and the tests that covered them [19]. Said objectives and tests are stored in an archive [19]. The process is then remodelled at the end of the current generation so that the next generation won't consider the objectives stored in the archive [19].

DynaMOSA. DynaMosa is a dynamic algorithm that considers the existing structural dependencies between objectives during optimisation of test suites [16]. Recent improvements in EvoSuite include the transition from the Whole Suite with Archive approach to the Many-Objective Sorting Algorithm with Dynamic target selection [16]. In a study done by the creators of DynaMOSA it is revealed that in a multitude of classes the approach manages to attain higher coverage than the previously used WSA approach [16].

Reinforcement Learning. In their paper Almulla et al. tried to address the fact that many coverage criteria do not have well defined fitness function or have fitness functions at all [2]. They used reinforcement learning to select and adapt during test suite generation the set of fitness functions that are used [2]. The fitness function selection algorithm described in their paper achieves up to 107.41% improvement when measuring exception detection and diversity of test suites [2]. Furthermore, it provides an improvement

of up to 259.90% when measuring fault detection for the goal of exception detection [2].

Research Gap. Many people have done research on Evosuite, but not everything has been covered. In their research, Rojas et al. took a look at the effectiveness of combining different coverage criteria that EvoSuite provides - Line Coverage, Branch Coverage, Exception Coverage, etc. [18]. However, they do not even mention the effectiveness of Input Diversity as a coverage criteria. This is the research gap that this paper aims to fill.

3 APPROACH

This section explains the approach taken and methods used to answer the research questions. Subsection 3.1 explains the data used in the research. Subsection 3.2 explains how the data was processed before it was used in the machine learning model. Subsection 3.3 and 3.4 explain how research questions 1 and 2, and research question 3 are going to be answered respectively.

3.1 Data

The research will be done on projects present in the SF-110 Corpus of Classes [9], provided by EvoSuite, and in the Apache Commons. From all available projects 346 classes were chosen as a benchmark. The classes used are the same as the ones used by Panichella et al. in their research on the performance of DynaMOSA [**?**]. Tests were generated through the use of EvoSuite for different fitness functions and different time budgets per fitness function. The main idea was to measure structural coverage and fault detection capabilities achieved by the test suites. The chosen time budgets are 60, 180 and 300 seconds and are based on previous research [15, 16]. For fault detection capabilities, we ran EvoSuite with the 60 seconds time budget only. The chosen fitness functions are Input Diversity in combination with Branch Coverage (IDBC), Branch Coverage (BC), and the Default fitness function (DFF) used by EvoSuite. To address the randomness in the search-based approach used in EvoSuite, we performed 10 runs with EvoSuite for each combination of fitness function and time budget. Furthermore, we computed the static metrics of the chosen classes with the CK tool created by Aniche [3]. Unfortunately, the tool was not able to generate metrics for all classes; to be exact metrics couldn't be generated for 6 classes. Therefore, we discarded those 6 classes from the data. Furthermore, for some classes, EvoSuite was not able to generate test suites as either there were no results for branch coverage or mutation score. There were classes as well for which EvoSuite wasn't able to generate 10 test suites, and therefore they had less than 10 results. In other to mitigate these two issues we filled the missing results with zeros, as not being able to create a test suite for a class technically means that 0% branch coverage or mutation score was achieved. We also discarded metrics that either produced null values or always produced a score of 0 - *tcc* and *lcc*.

3.2 Data Preprocessing

After computing the static metrics and obtaining the structural coverage and mutation score, our goal is to find patterns in static metrics that correlate to high structural coverage or high mutation score - in other words high fault detection capabilities. In favour of finding any valuable data we would have to filter out noise from the data. We performed a Shapiro-Wilk test per class. The test is given the 10 results per class and it computes computes a *p-value*. The *p-value* is then compared to 0.05 - the critical value for a 95% confidence interval. The null hypothesis states that the data is normally distributed. If the *p-value* is smaller than 0.05 then we reject the null hypothesis, which means that the data is non-normally distributed, and vice versa. Depending on the results of the Shapiro-Wilk test [20] we perform the Student's t-test [21] in the case of normality and a Wilcoxon rank sum test [24] in the case of non-normality. The test is performed per class - on the results of the 10 runs of each class. The tests check whether there is statistically significant difference when we compare the results of IDBC with the results of BC and DFF respectively. In other words, the test checks whether there is a statistically significant difference between the results of the pairs of fitness functions and returns a *p-value*. The null hypothesis is that the two sets of results come from the same distribution. If the *p-value* is smaller than 0.05 then the results do not come from the same distribution and there is significant difference between them, and vice versa. The classes for which insignificant differences were found were discarded from the data set. Further analysis was performed on the set of classes for which significant differences were found due to reasons explained in Section 5.

To get a better understanding of the results, we computed the Vargha and Delaney effect size measurement [22]. The measurement provided us with two results per class - value estimate and magnitude. The value estimate is a number between 0 and 1 that shows which of the two configurations being compared has better results. The threshold value is 0.5. A value below the threshold means that the second configuration has better results and vice versa. The magnitude is essentially how big the difference in the results are. Furthermore, any class-wise comparison that yielded a *negligible* magnitude according to the measurement was discarded from the set of significant classes.

3.3 RQ 1 and 2

Since we are looking at the circumstances that possibly render IDBC an effective fitness function, we utilize decision tree model. Previous studies show that they work well with conditions and are preferred when sophisticated problems are encountered [23]. We need a process that extracts the patterns from the data and presents them in an understandable way. This means that the only feasible model to use is indeed a decision tree, as no other model can present a pattern. Furthermore, sophisticated models such as a neural network would not suffice as there is no way we could understand what the patterns in the data are; not to mention that we are working with too little data to construct an efficient neural network. Moreover, in a research where the performance of a decision tree, logistic regression and a neural net are compared, Olson found that a decision tree is the most robust when data balancing is performed upon unbalanced data [13].

Using the previously computed Vargha and Delaney measurements, we prepared the data for training the model. Class-wise comparisons that provided an estimate strictly above 0.5 were labeled as *1* and the rest were labeled as *0*. In other words, we labeled classes in which we saw improvement in results when IDBC was

used as *1* and classes in which we saw no change or deterioration in results as *0*. The reason we mapped them to binary values is because we want to do classification.

After dividing the data with a 6:4 ratio for training and testing, we use a data balancing technique in the training set in order to deal with the data imbalance we performed SMOTE [6]. Performing SMOTE makes the most sense, but there is of course the downside that artificial data is created. However, using SMOTE instead of random oversampling, we overcome the risk of the model being overfitted [4]. This is a big upside, as in most cases one of the two labels is heavily underrepresented, which incurs a big risk of overfitting.

Due to the large dimensionality of the data - 49 features - and mainly the fact that not every code metrics provides useful insight in our research, we used a Random Forest Classifier and MRMR [1] to select the top 10 most significant features. One could argue that feature selection is not needed as in the worst case we would have to work with 346 data points, however, we still get the benefit of removing irrelevant or unimportant features, thus gaining efficiency. Furthermore, when the features in a data set are similar in size to the samples overfitting can occur [10]. This scenario is present in our data, hence, avoiding overfitting is another benefit of feature reduction. We chose to use the aforementioned methods, as they preserve labels and do not make combinations of the data in order to reduce the feature dimensionality. There is no specific reason behind choosing these two methods other than label preservation.

We decide to create different decision trees based on different number of features and take the best classifier. To find the best performing decision tree, we brute forced through all possible combinations of parameters and feature selection models. This means we created a decision tree using both feature selection models separately each time with a different number of features used - 5, 10, 15 and 20. For each decision tree we tuned the parameters of our decision tree by running a grid search on it, and we took the best estimator from the grid search.

3.4 RQ 3

For mutation score, we focused on the minimum time budget as the most significant differences between configurations can be observed in this time budget, therefore, we left out analysis of mutations score in the question. For the branch coverage results we computed the box plots per time budget for each fitness function and compared them to one another. It is important to mention that we used all data and not just the significant classes, as the insignificant classes are still actual data and part of the distribution. The box plots where created on the medians on the classes. Furthermore we looked at the averages of the classes and looked at how they change as the time budgets increase.

4 RESULTS

This section presents the results found during analysis of the data. Subsection 4.1 presents the results for research question 1. Subsection 4.2 presents the results for research question 2. Subsection 4.3 presents the results for research question 3.

[1] https://github.com/smazzanti/mrmr

4.1 RQ 1

Upon manual analysis of the classes, we noticed interesting results in the classes with significant difference from the comparison of IDBC against the DFF for the time budget of 300 seconds. There we found the following three aspects as prominent - moderate to high amount of branches in methods, most of which consecutive, class parameters and trivial methods. The manual analysis of classes for different configuration comparisons did not grant any interesting results.

Table 1 shows how many configuration-wise comparisons present differences that are statistically significant in terms of branch coverage. With the increase in the time budget we notice an increase in the significant differences. The increase seen is strict for IDBC vs BC, where as for IDBC vs DFF we that the significant differences decrease by 4 when the time budget is increased to 300 seconds. The number of significant differences shows that more classes have statistically significant difference for IDBC vs DFF than for IDBC vs BC. This hints that IDBC is to be preferred over DFF and not over BC. It is important to mention that we could not have more data points as the test generation process is time-taking.

	Time Budget	Significant	Insignificant
IDBC vs BC	60s	24	315
	180s	29	310
	300s	34	305
IDBC vs DFF	60s	65	274
	180s	74	265
	300s	71	268

Table 1: Number of Significant and Insignificant differences

The results for the top 5 positive correlations and top 5 negative correlations for each time budget for IDBC against BC and for IDBC agaisnt DFF are presented in Appendix A.1 and Appendix A.2 respectively. We notice that the correlation tables for IDBC vs BC present significantly higher correlations that the tables for IDBC vs DFF. Tables 5 through 7 show significant correlation - both positive and negative. Regarding the comparison of IDBC and BC, we notice that in table 5 we have particularly high negative correlations. With the increase in time budget the values of the negative correlations significantly drop; this can be seen in tables 6 and 7. This leads us to the conclusion that as the time budget is increased, singular static metrics have less negative effect on branch coverage results. In other words, the influence of negatively correlated metrics on branch coverage diminishes with increase in time budget. Furthermore, we notice that the highly correlated metrics are not constant, meaning that they vary. This leads us to the conclusion that different metrics are important for different time budgets and different configurations.

The results of the Vargha and Delaney effect size measurements are presented in Appendix B. In table 2 we see that in most cases IDBC wins against DFF. Furthermore, almost all classes win with a large difference - 55 classes for 60 seconds, 60 classes for 180 seconds, and 52 classes for 300 seconds. On the other hand, IDBC wins against BC in a small number of classes - 6 for 60 seconds, 9 for 180 seconds, and 8 for 300 seconds. The Vargha and Delaney effect

	Time budget	#Win				#Lose			
		Large	Medium	Small	Negligible	Large	Medium	Small	Negligible
IDBC vs BC	60s	4	2	0	0	16	2	0	0
	180s	8	1	0	0	17	3	0	0
	300s	7	1	0	0	22	4	0	0
IDBC vs DFF	60s	55	1	0	0	8	1	0	0
	180s	60	2	0	0	9	2	1	0
	300s	52	3	1	0	13	2	0	0

Table 2: Vargha and Delaney effect size measurement for branch coverage

size measurement further supports that IDBC is to be preferred over DFF and not over BC when it comes to branch coverage results. This observation supports our initial motivation to do this study, as the results show that the default configuration is not the best option all of the time.

Figures 3 through 5 and figures 6 through 8 are the best classifiers for IDBC vs BC and for IDBC vs DFF respectively for each time budget. The accuracy, feature reduction model and number of features used are as follows:

- Figure 3: 60%, Random Forest Classifier, 5 features
- Figure 4: 91.66%, Random Forest Classifier, 10 features
- Figure 5: 71.43%, Random Forest Classifier, 15 features
- Figure 6: 88.46%, Random Forest Classifier, 10 features
- Figure 7: 83.33%, MRMR, 15 features
- Figure 8: 68.97%, Random Forest Classifier, 5 features

In figure 3 we see that *fanin* plays a key role in determining whether IDBC will provide better results than BC for a class when the time budget is 60 seconds. In the case where *fanin <= 0.5*, the tree will chose IDBC as a better fitness function. To choose IDBC over BC *fanin* needs to be lower than or equal to 0.5. In figure 4 we see that *cbo* and *dit* determine whether IDBC will outperform BC when the time budget is 180 seconds. In the case where *cbo <= 3.47* and *dit <= 5.228*, the tree will choose IDBC as a better fitness function. In figure 5 we see that the key metrics are *dit* and *totalMethodsQty* when the time budget is 300 seconds. In the case where *dit > 6.597* and *totalMethodsQty <= 2.208*, the tree will choose IDBC as a better fitness function. We notice that *dit* is present in both decision trees for the time budgets of 180 and 300 seconds.

The following metrics determine whether IDBC will outperform DFF for each time budget respectively. In figure 6 the key metrics is *wmc* when the time budget is 60 seconds. In the case where *wmc > 21.5* or *wmc <= 11.5*, the tree will choose IDBC as the better fitness function. For the time budget of 180 seconds, we see in figure 7 that the key metrics are *fanout*, *privateMethricsQty* and *protectedMethodsQty*. In the case where textitfanout > 10.5 or *fanout <= 10.5 and privateMethodsQty > 23.5 and dit <= 41* or *fanout <= 10.5 and protectedMethodsQty <= 17.5*, the tree will choose IDBC as the better fitness function. For the time budget of 300 seconds, we see in figure 8 that the key metrics are *cbo*, *wmc*, *cboModified* and *fanin*. In the case where *cbo > 115 and cbo <= 377.499 and fanin <= 24.58* or *cbo <= 115 and wmc <= 15.757 and cboModified <= 27.459*, the tree will choose IDBC as the better fitness function. We notice that *dit* is presented in the decision trees for the time budgets 60 and 180 seconds. Both *wmc* and *cboModified* are present in the trees for the time budgets of 60 and 300 seconds.

When IDBC is compared to BC it significantly underperforms in 5.3% of the classes for the time budget of 60 seconds, in 5.9% of the classes for the time budget of 180 seconds, and in 7.7% of the classes for the time budget of 300 seconds. The static metrics cases in which IDBC is to be preferred over BC are: time budget 60 seconds - *fanin <= 0.5*, time budget 180 seconds - *cbo <= 3.47 and dit <= 5.228*, time budget 300 seconds - *dit > 6.597 and totalMethodsQty <= 2.208*. When IDBC is compared to DFF it achieves significantly better results in 16.5% of the classes for the time budget of 60 seconds, 18.23% of the classes for the time budget of 180 seconds, and in 16.22% of the classes for the time budget of 300 seconds. The static metrics cases in which IDBC is to be preferred over DFF are: time budget 60 seconds - *wmc <= 11.5* or *wmc > 21.5*, time budget 180 seconds - *fanout > 10.5* or *fanout <= 10.5 and privateMethodsQty > 23.5 and dit <= 41* or *fanout <= 10.5 and protectedMethodsQty <= 17.5*, time budget 300 seconds - *cbo > 115 and cbo <= 377.499 and fanin <= 24.58* or *cbo <= 115 and wmc <= 15.757 and cboModified <= 27.459*.

4.2 RQ 2

Table 3 shows how many configuration-wise comparisons present differences that are statistically significant in terms of mutation scores.

	Time Budget	Significant	Insignificant
IDBC vs BC	60s	66	273
IDBC vs DFF	60s	108	231

Table 3: Number of Significant and Insignificant differences

In the analysis of the mutation scores there are more significant differences to begin with - 61 for IDBC against BC and 101 for IDBC against the DFF. This gave us a lot more data to work with.

However, we found that only 12 class's scores of the IDBC were outperforming the class's scores of the DFF - all 12 differences had a large magnitude. On the other hand, we had 53 class's scores of IDBC that outperformed the class's scores of BC - 51 differences with large magnitude and 2 with medium magnitude. Talbe 4 shows the results of the Vargha and Delaney effect size measurement for mutation scores, which can also be found in Appendix E. This

	Time budget	#Win				#Lose			
		Large	Medium	Small	Negligible	Large	Medium	Small	Negligible
IDBC vs BC	60s	51	2	0	0	11	2	0	0
IDBC vs DFF	60s	12	0	0	0	94	2	0	0

Table 4: Vargha and Delaney effect size measurement for mutation score

higher number of significant difference is expected as when measuring structural coverage input diversity barely contributes unlike when measuring mutation scores. Having diverse input will enhance the fault detection capabilities due to a bigger amount of test cases. However covering a branch with different inputs counts as the branch being covered at most twice, as we are measuring if the branch is covered and nothing else - once when the branch condition is fulfilled and once when it is not.

The measurement shows that the winners achieve a large improvement in mutation score. Even though we have few winners when comparing IDBC against the DFF, all of them show large improvement in the mutation score.

The decision trees for IDBC vs BC and IDBC vs DFF can be found in Appendix F. Both decision trees presented in the appendix concern the time budget of 60 seconds. In figure 9 we see that the key metrics that determine whether IDBC will be chosen over BC are *dit*, *fain* and *noc*. In the case where *dit <= 0.265 and fanin <= 12610.119 and noc <= 13.438*, the tree will choose IDBC as the better fitness function. Figure 10 shows that they key metrics when choosing between IDBC and DFF are *cboModified*, *cbo* and *wmc*. In the case where *cboModified > 8.5 and cboModified <= 28 and cbo <= 0.5 and wmc > 277.5* or *cboModified > 8.5 and cboModified <= 28 and cbo > 0.5*, the tree will choose IDBC as the better fitness function. Unlike the case for the decision trees of branch coverage, here we see no overlapping metrics in the decision trees.

The accuracy, feature reduction model and number of features used are as follows:

- Figure 9: 74.74%, MRMR, 10 features
- Figure 10: 81.82%, Random Forest Classifier, 5 features

When IDBC is compared to BC it achieves significantly better results in 15.6% of the classes. The static metrics cases in which IDBC is to be preferred over BC are: *dit <= 0.265 and fanin <= 12610.119 and noc <= 13.438*. When IDBC is compared to DFF it significantly underperforms in 28.3% of the classes. The static metrics cases in which IDBC is to be preferred over DFF are: *cboModified > 8.5 and cboModified <= 28 and cbo <= 0.5 and wmc > 277.5* or *cboModified > 8.5 and cboModified <= 28 and cbo > 0.5*.

4.3 RQ 3

In figures 1 it can be seen that as the time budget increases overall the distributions of the three fitness functions tend to extend towards the right. Change in the averages of branch coverage with respect to time can be seen in figure 12. Both figures are present in Appendix G. We observe an increase in the medians and upper quartiles when the time budget is increased from 60 to 180 seconds. However, when the time budget is increased to 300 seconds, we notice an increase in DFF's results, whilst the results of IDBC and BC remain almost the same. This is expected as branch coverage should converge to a value as we increase the time budget. There is a limited number of branches that can be covered and at some point increasing the time budget would not necessarily lead to an increase in results. This is because either all branches have been covered or for some reason branches that cannot be covered remain uncovered. This is further supported by the change in average branch coverage presented in figure 12. In the figure we see that an increase from 180 to 300 seconds does not result in increase in results for IDBC and DFF. The case for mutation scores, however, is going to be different. There is theoretically an infinite amount of ways to target and kill mutants. Therefore, mutation scores will benefit from increase in the time budget. Increase in time budget, will allow for more test cases to be created, hence, for more mutants to be killed. Whilst creating more test cases for branch coverage won't increase the results as at some point all branch should have been covered.

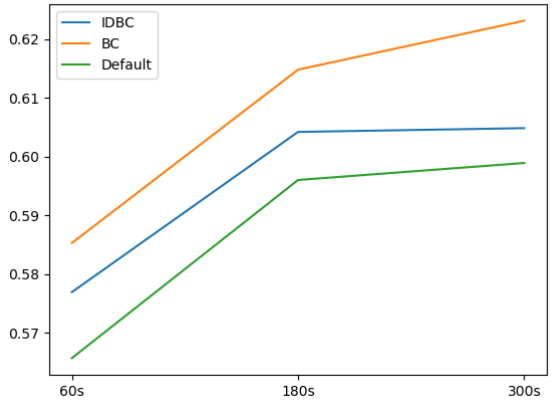

Figure 2: Change in branch coverage with respect to time

Increasing the time budget from 60 to 180 seconds leads to a 4.74% increase - from 57.69% to 60.42% - in the average branch coverage achieved by IDBC. Increasing the time budget from 180 to 300 seconds leads to a 0.1% increase - from 60.42% to 60.48% - in the average branch coverage achieved by IDBC. The percentage increase in the average branch coverage diminishes with the increase in time budget as the branch coverage results converge.

5 DISCUSSION

The first step we took towards finding patterns was computing a correlation matrix between the code metrics of the classes under

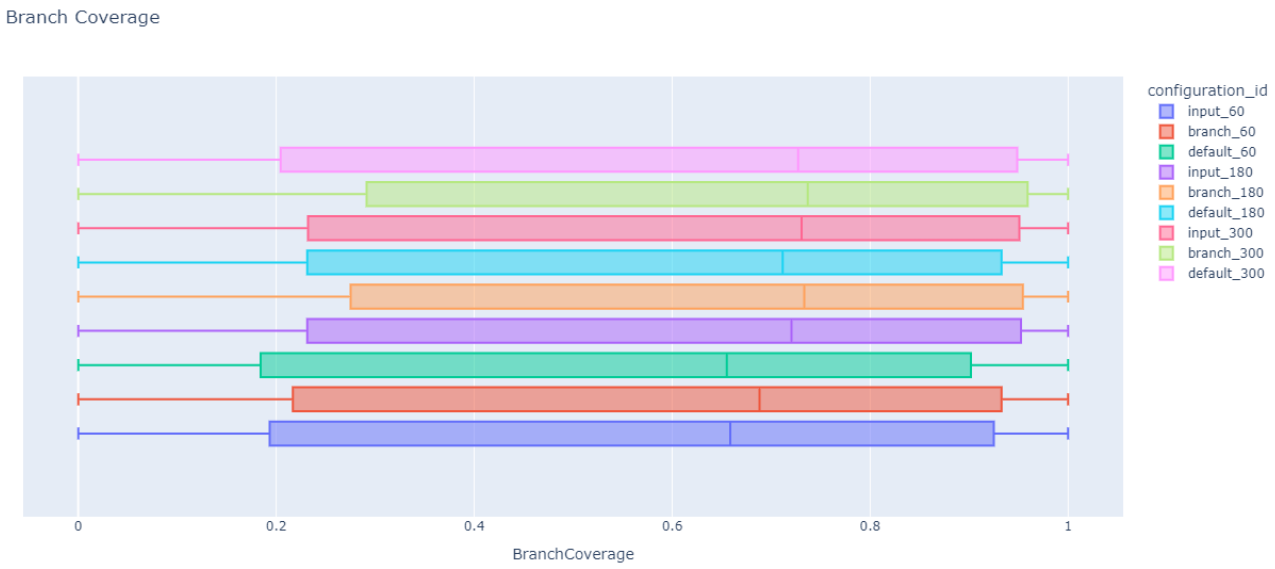

Figure 1: Data Distribution

analysis and the difference of the medians of their results. Both correlation matrices for structural coverage and mutation score showed significant correlation for some metrics. However, this was not enough to deduce patterns in the data. We decided to manually find patterns in the data by looking in the source code of the classes, however this was to no avail. Therefore, we decided to use machine learning to find patterns in the data.

Furthermore, there are several reasons we choose not to include the insignificant data in the data used to train the model. Firstly, there are a lot more insignificant data points than there are significant data points. This means that including the insignificant data points would create so much noise that any value that the significant data points provide will be essentially lost. As we didn't want to make any bold assumptions, we trained a model that included the insignificant data and it achieved 50% accuracy in best case scenario. We tried three data balancing techniques - oversampling, undersampling and class weights, and none of them helped. We have too much insignificant data, and oversampling created too much artificial data based on the significant data to match the number of insignificant data. Undersampling on the other hand, reduced the insignificant data to match the number of significant data points. However, we have to take into account that in the worst case scenario we had 12 significant data points that showed improvement. Therefore, after undersampling we were left with only 33 data points. Class weights did not work any better, as weights for some labels were too big - weight of 19 - and yet provided no improvement. Secondly, we mainly care about under which circumstances there will be improvement. As the insignificant data present little to no change in results, this means that miss-labeling a class where there would be no improvement as a class where there will be improvement, the consequences would be negligible. Therefore, we concluded that we can sacrifice potential miss-labeling for accuracy and discovery of more valuable patterns.

We used SMOTE and class weights to try to balance the data. There was no point in performing undersampling as this would decrease the data to very small amounts, hence any model of sorts would be very overfitted or inaccurate. We discarded class weights as an option as in some cases the class weights were too big - class weight of 8 for IDBC vs DFF for mutation score. Furthermore, the decision trees utilizing class weights underperformed the decision trees utilizing SMOTE. At best it achieved around 70% accuracy, however, this is worse in 3 out of 6 time budgets for branch coverage when compared to the trees utilizing SMOTE.

There are probably better suited methods for our data - PCA, SVD, etc. - but not for our purpose. Such methods might find better correlation between code metrics and branch coverage or mutation score by making combination of the features. As we previously specified, we believe that a combination of code metrics will show a higher correlation to branch coverage or mutation score than separate metrics on their own. Yet, we did not choose a method that would reduce dimensionality of data by creating combinations of code metrics as this would not preserve the labels, and we would not be able to infer a clear pattern from such combinations. Hence, we sacrificed more intricate and hidden correlations for label preservation, as label preservation is key for achieving our goal. Given the large number of features and the fact that most of them probably had little to no impact on the performance of fitness function, we thought that it would be most appropriate to decide how many features to preserve based on the values of the correlation matrix. However, few correlations were common between the different correlation matrices. Therefore, we decided that it would be better to

create different decision trees based on different number of features and take the best classifier.

6 THREATS TO VALIDITY

To address the small number of data points used - 346 - we have included a variety of different classes from different projects. This ensures that the research was done on a diverse and non-trivial code. We made sure to use the same data for both branch coverage results and fault detection capabilities. Furthermore, the benchmark used was taken from previous research regarding EvoSuite [16]. To address the randomness of the search-based approach utilized by EvoSuite, we performed 10 runs per class, thus, creating a distribution of the results for each class. To make sure that our classifier works with well represented data, we used a data balancing technique to mitigate data imbalance. Analysis of the data was backed up by statistical methods - Vargha and Delaney effect size, Wilcoxon rank sum test, and Student's t-test.

7 RESPONSIBLE RESEARCH

The paper does not raise any ethical concerns nor violate any copy right laws since the source codes and information used are openly available online. The CK tool used for code metrics generation is purely deterministic and therefore the static metrics discussed can easily be reproduced. Even test generation can be reproduced if we pass the same random seed despite it being stochastic. Furthermore, the methodology of the paper does not rely on our own implementations of the used analytical methods, but rather on libraries and already existing implementations that are free to use, hence, the approach could be easily reproduced. Furthermore, the source code used for this research is available online [2].

8 CONCLUSION AND FUTURE WORK

The goal of the paper was to shine a light on the effectiveness of IDBC compared to BC and DFF respectively. We wanted to see whether IDBC can be a better option than DFF and in which cases this hold true. To this end we created a Decision Tree model to find relevant patterns in the data that suggest IDBC is superior and analysed how the different time budgets affect the results achieved by IDBC. The analysis was conducted on a benchmark of 346 classes taken from the SF-110 Corpus of Classes and the Appache Commons. Future work include testing the created models with new data outside of the used benchmark. It may be beneficial to compare IDBC against other fitness functions as well. We plan to further analyse the effects of time budgets through statistical significance analysis.

REFERENCES

[1] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27

[2] Gay G. Almulla, H. 2022. Learning how to search: Generating effective test cases through adaptive fitness function selection.. In Empir Software Eng 27, 38 (2022). https://doi.org/10.1007/s10664-021-10048-8

[3] Maurício Aniche. 2015. Java code metrics calculator (CK). Available in https://github.com/mauricioaniche/ck/.

[4] Gustavo E. A. P. A. Batista, Ronaldo C. Prati, and Maria Carolina Monard. 2004. A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data. SIGKDD Explor. Newsl. 6, 1 (jun 2004), 20–29. https://doi.org/10.1145/1007730.1007735

[5] E Chandra and P Edith Linda. 2010. Class break point determination using CK metrics thresholds. Global journal of computer science and technology (2010).

[6] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. Journal of artificial intelligence research 16 (2002), 321–357.

[7] Sebastiano Panichella et al. 2016. The impact of test case summaries on bug fixing performance: An empirical investigation.. In Proceedings of the 38th International Conference on Software Engineering. 2016. https://doi.org/10.1145/2884781.2884847

[8] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (Szeged, Hungary) (ESEC/FSE '11). Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[9] Gordon Fraser and Andrea Arcuri. 2014. A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite. ACM Transactions on Software Engineering and Methodology (TOSEM) 24, 2 (2014), 8.

[10] A. Jović, K. Brkić, and N. Bogunović. 2015. A review of feature selection methods with applications. In 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). 1200–1205. https://doi.org/10.1109/MIPRO.2015.7160458

[11] Sotiris B Kotsiantis. 2013. Decision trees: a recent overview. Artificial Intelligence Review 39, 4 (2013), 261–283.

[12] Seyedali Mirjalili. 2019. Genetic Algorithm. Springer International Publishing, Cham, 43–55. https://doi.org/10.1007/978-3-319-93025-1_4

[13] David L. Olson. 2005. Data Set Balancing. In Data Mining and Knowledge Management, Yong Shi, Weixuan Xu, and Zhengxin Chen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 71–80.

[14] Mitchell Olsthoorn, Pouria Derakhshanfar, and Annibale Panichella. 2021. Hybrid Multi-level Crossover for Unit Test Case Generation. In Search-Based Software Engineering, Una-May O'Reilly and Xavier Devroey (Eds.). Springer International Publishing, Cham, 72–86.

[15] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing. In 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). 1224–1228.

[16] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. IEEE Transactions on Software Engineering 44, 2 (2018), 122–158. https://doi.org/10.1109/TSE.2017.2663435

[17] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Incremental Control Dependency Frontier Exploration for Many-Criteria Test Case Generation. In Search-Based Software Engineering, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer International Publishing, Cham, 309–324.

[18] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In Search-Based Software Engineering, Márcio Barros and Yvan Labiche (Eds.). Springer International Publishing, Cham, 93–108.

[19] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A Detailed Investigation of the Effectiveness of Whole Test Suite Generation. Empirical Softw. Engg. 22, 2 (apr 2017), 852–893. https://doi.org/10.1007/s10664-015-9424-2

[20] S. S. SHAPIRO and M. B. WILK. 1965. An analysis of variance test for normality (complete samples)†. Biometrika 52, 3-4 (12 1965), 591–611. https://doi.org/10.1093/biomet/52.3-4.591 arXiv:https://academic.oup.com/biomet/article-pdf/52/3-4/591/962907/52-3-4-591.pdf

[21] Student. 1908. The probable error of a mean. Biometrika (1908), 1–25.

[22] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics 25, 2 (2000), 101–132. http://www.jstor.org/stable/1165329

[23] Sholom M. Weiss and Ioannis Kapouleas. 1989. An Empirical Comparison of Pattern Recognition, Neural Nets, and Machine Learning Classification Methods. In Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1 (Detroit, Michigan) (IJCAI'89). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 781–787.

[24] F Wilcoxon. 1945. Individual comparisons by ranking methods. Biom. Bull., 1, 80–83.

[2] https://github.com/IvanStranski/CSE3000_Research_Project

A CORRELATIONS BRANCH COVERAGE

A.1 IDBC vs BC

60	Positive	anonymousClassesQty	0.4233
		parenthesizedExpsQty	0.422
		defaultFieldsQty	0.1928
		numbersQty	0.1618
		publicFieldsQty	0.0788
	Negative	wmc	-0.5836
		totalMethodsQty	-0.5672
		cbo	-0.5594
		fanout	-0.5594
		returnQty	-0.5545

Table 5: Correlation for 60 seconds time budget

180	Positive	maxNestedBlocksQty	0.4654
		anonymousClassesQty	0.1727
		staticMethodsQty	0.1335
		staticFieldsQty	0.1066
		synchronizedMethodsQty	0.0674
	Negative	totalFieldsQty	-0.3708
		privateFieldsQty	-0.3579
		protectedMethodsQty	-0.3457
		uniqueWordsQty	-0.3386
		returnQty	-0.3199

Table 6: Correlation for 180 seconds time budget

300	Positive	innerClassesQty	0.4366
		cboModified	0.4166
		fanin	0.4057
		defaultMethodsQty	0.3348
		privateMeethodsQty	0.2992
	Negative	lambdasQty	-0.2903
		lcom*	-0.2034
		maxNestedBlocksQty	-0.1215
		loopQty	-0.1139
		protectedMethodsQty	-0.1014

Table 7: Correlation for 300 seconds time budget

A.2 IDBC vs DFF

60	Positive	publicFieldsQty	0.2072
		assignmentsQty	0.1948
		modifiers	0.1734
		mathOperationsQty	0.1704
		logStatementsQty	0.1615
	Negative	fanin	-0.2181
		privateFieldsQty	-0.1735
		nosi	-0.1371
		anonymousClassesQty	-0.1257
		cboModified	-0.1153

Table 8: Correlation for 60 seconds time budget

180	Positive	lcom	0.1557
		defaultMethodsQty	0.1383
		publicFieldsQty	0.1256
		uniqueWordsQty	0.0992
		totalMethodsQty	0.0911
	Negative	synchronizedMethodsQty	-0.2827
		privateFieldsQty	-0.1712
		fanin	-0.1359
		staticMethodsQty	-0.1096
		modifiers	-0.1

Table 9: Correlation for 180 seconds time budget

300	Positive	defaultMethodsQty	0.1853
		loopQty	0.1650
		nosi	0.1443
		publicFieldsQty	0.1406
		mathOperationsQty	0.1377
	Negative	tryCatchQty	-0.1387
		lcom*	-0.1263
		modifiers	-0.1076
		noc	-0.0831
		anonymousClassesQty	-0.0747

Table 10: Correlation for 300 seconds time budget

B VARGHA AND DELANEY EFFECT SIZE FOR BRANCH COVERAGE

	Time budget	#Win				#Lose			
		Large	Medium	Small	Negligible	Large	Medium	Small	Negligible
IDBC vs BC	60s	4	2	0	0	16	2	0	0
	180s	8	1	0	0	17	3	0	0
	300s	7	1	0	0	22	4	0	0
IDBC vs DFF	60s	55	1	0	0	8	1	0	0
	180s	60	2	0	0	9	2	1	0
	300s	52	3	1	0	13	2	0	0

Table 11: Vargha and Delaney effect size measurement

C DECISION TREES FOR BRANCH COVERAGE

C.1 IDBC vs BC

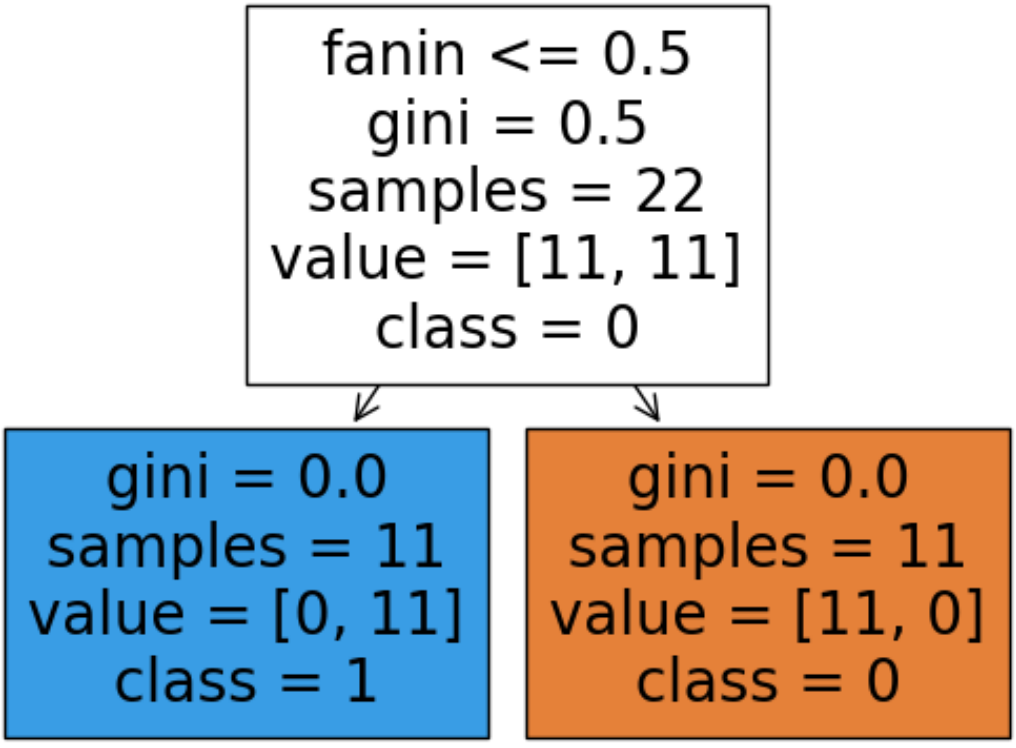

Figure 3: Input Diversity with Branch Coverage against Branch Coverage for 60 seconds

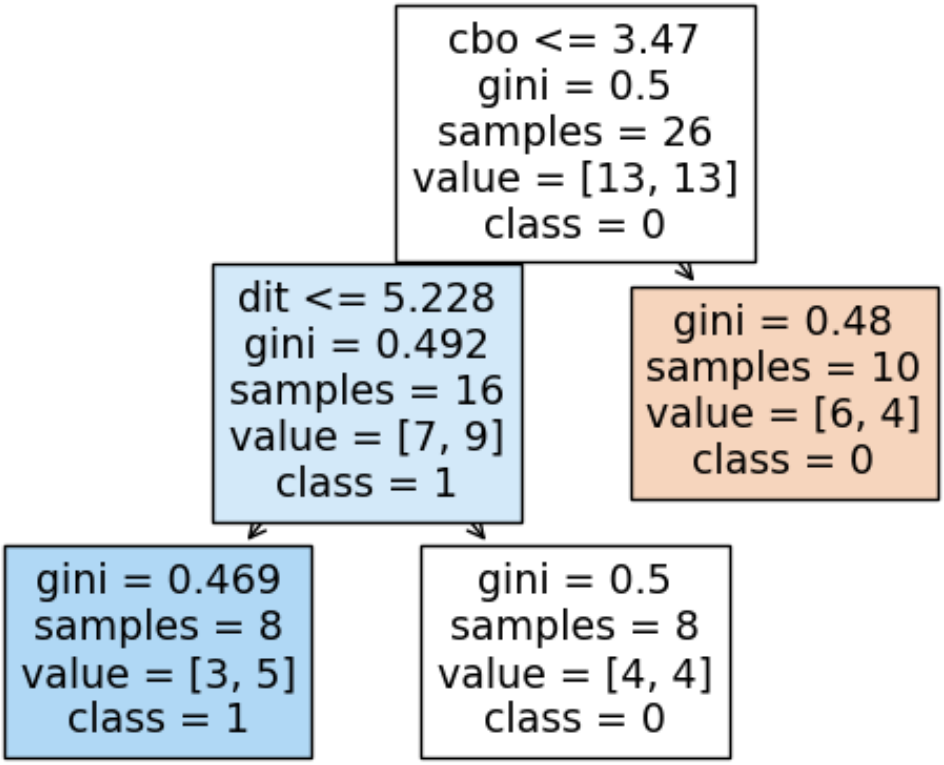

Figure 4: Input Diversity with Branch Coverage against Branch Coverage for 180 seconds

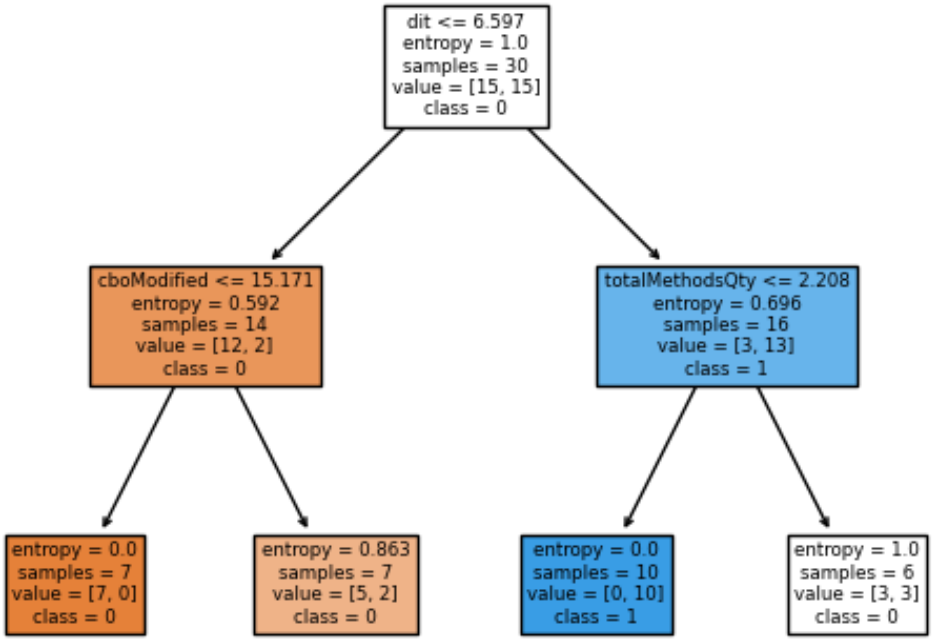

Figure 5: Input Diversity with Branch Coverage against Branch Coverage for 300 seconds

C.2 IDBC vs DFF

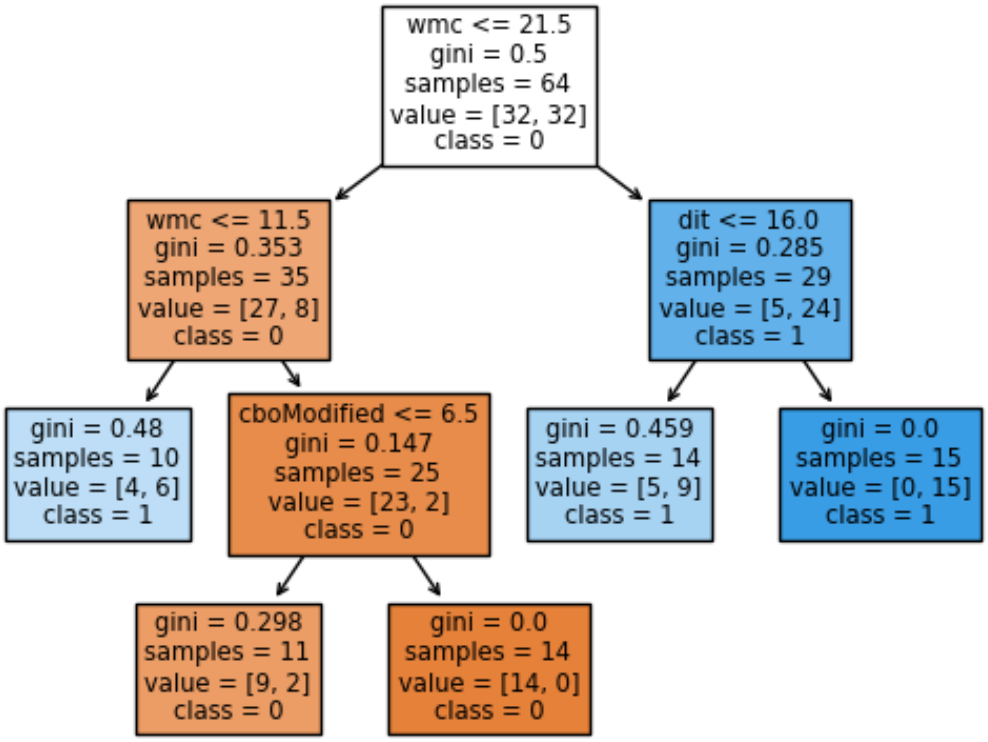

Figure 6: Input Diversity with Branch Coverage against Default fitness function for 60 seconds

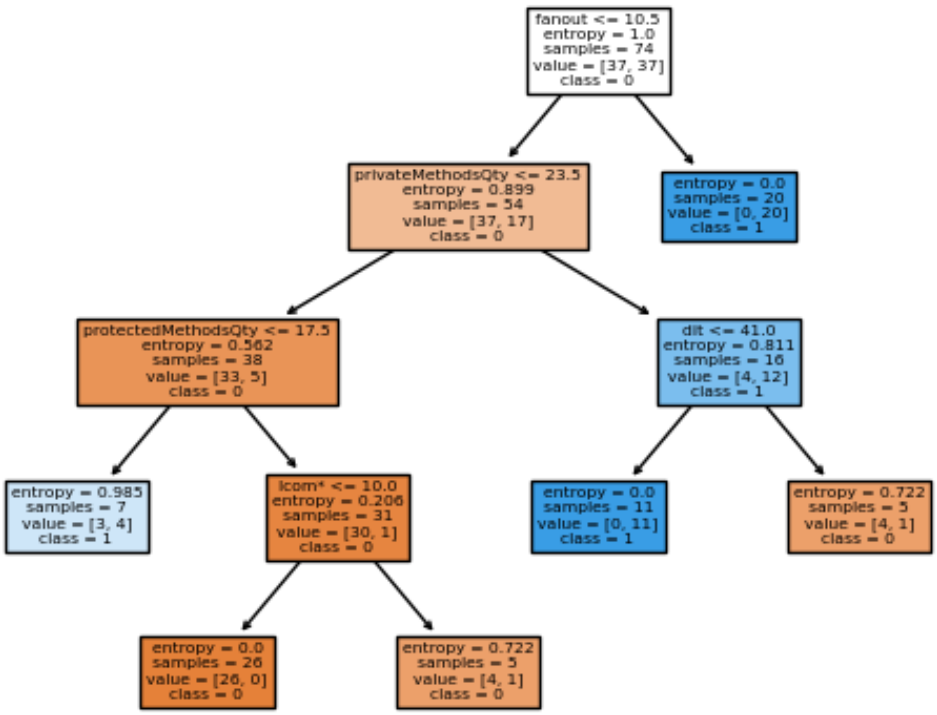

Figure 7: Input Diversity with Branch Coverage against Default fitness function for 180 seconds

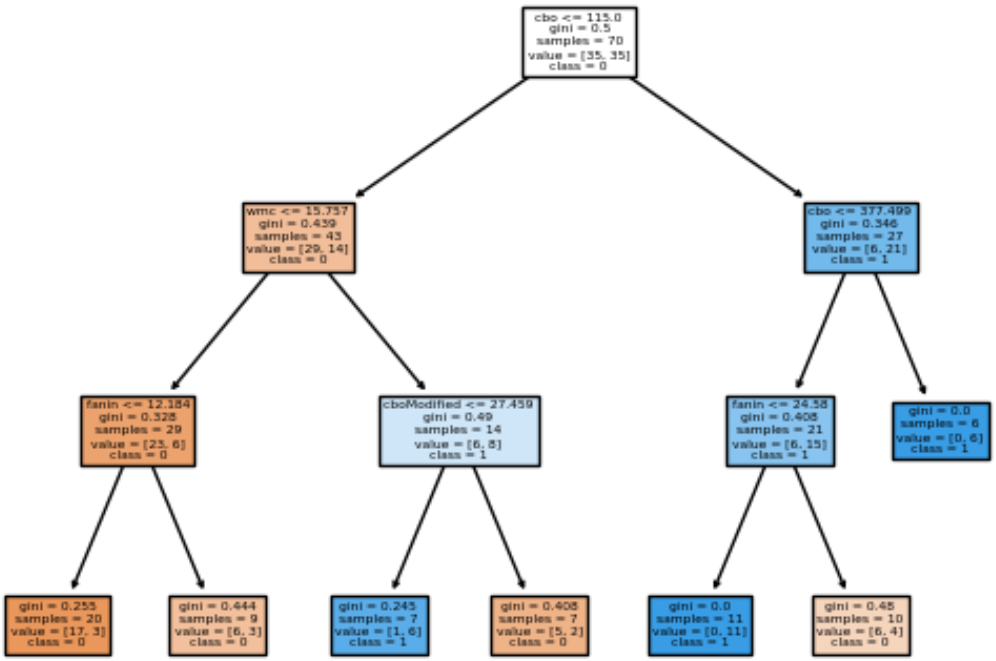

Figure 8: Input Diversity with Branch Coverage against Default fitness function for 300 seconds

D CORRELATIONS MUTATION SCORE

D.1 IDBC vs BC

60	Positive	maxNestedBlocksQty	0.3177
		loc	0.2808
		privateMethodsQty	0.2753
		assignmentsQty	0.2637
		wmc	0.2567
	Negative	modifiers	-0.1329
		finalMethodsQty	-0.059
		abstractMethodsQty	-0.0514
		noc	-0.0018

Table 12: Correlation for 60 seconds time budget

D.2 IDBC vs DFF

60	Positive	lcom*	0.3094
		finalMethodsQty	0.2489
		staticMethodsQty	0.2237
		uniqueWordsQty	0.2049
		publicFieldsQty	0.1935
	Negative	abstractMethodsQty	-0.3392
		modifiers	-0.3115
		stringLiteralsQty	-0.2881
		mathOperationsQty	-0.2627
		lcom	-0.2362

Table 13: Correlation for 60 seconds time budget

E VARGHA AND DELANEY EFFECT SIZE MEASUREMENT MUTAION SCORE

	Time budget	#Win				#Lose			
		Large	Medium	Small	Negligible	Large	Medium	Small	Negligible
IDBC vs BC	60s	51	2	0	0	11	2	0	0
IDBC vs DFF	60s	12	0	0	0	94	2	0	0

Table 14: Vargha and Delaney effect size measurement

F DECISION TREES MUTATION SCORE

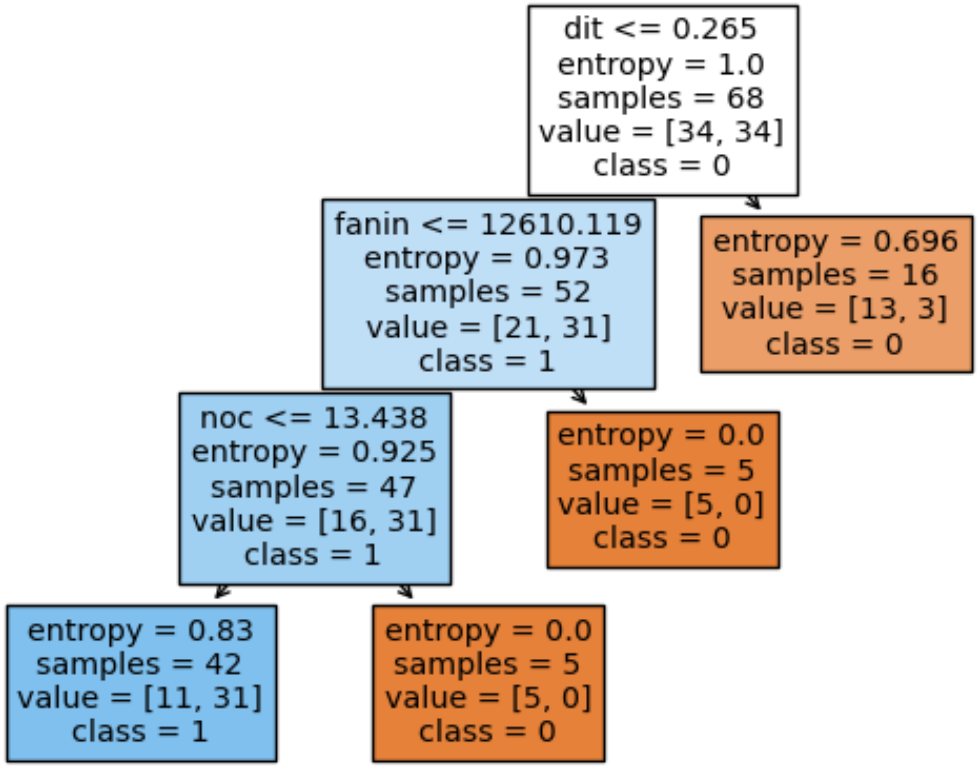

Figure 9: Input Diversity with Branch Coverage against Branch Coverage for 60 seconds

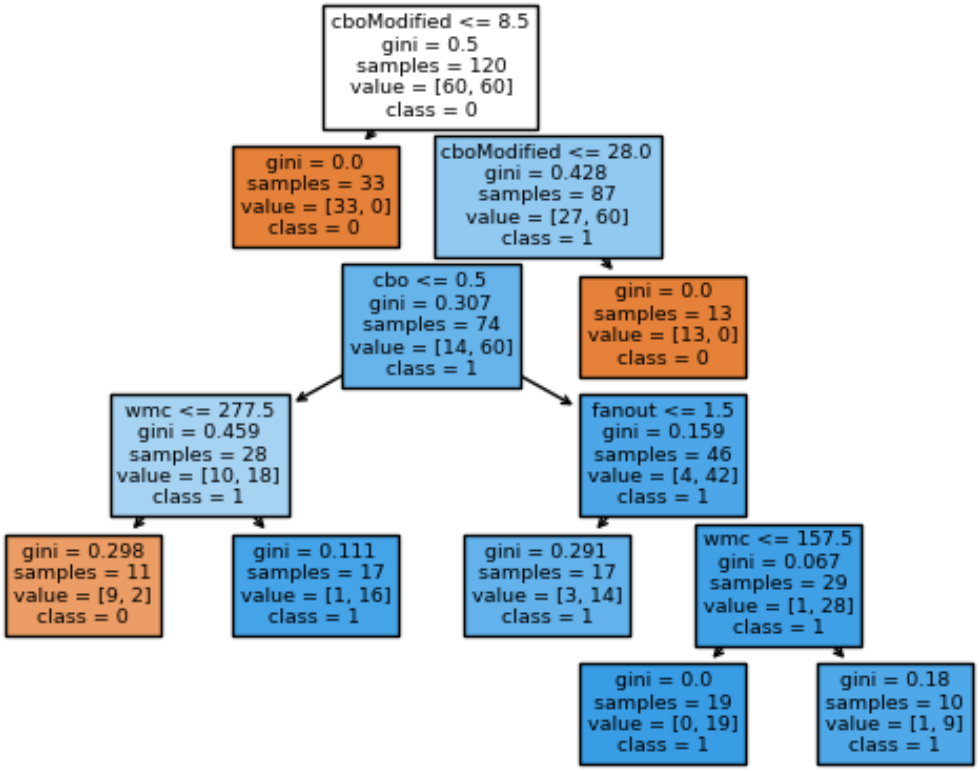

Figure 10: Input Diversity with Branch Coverage against Default fitness function for 60 seconds

G TIME ANALYSIS

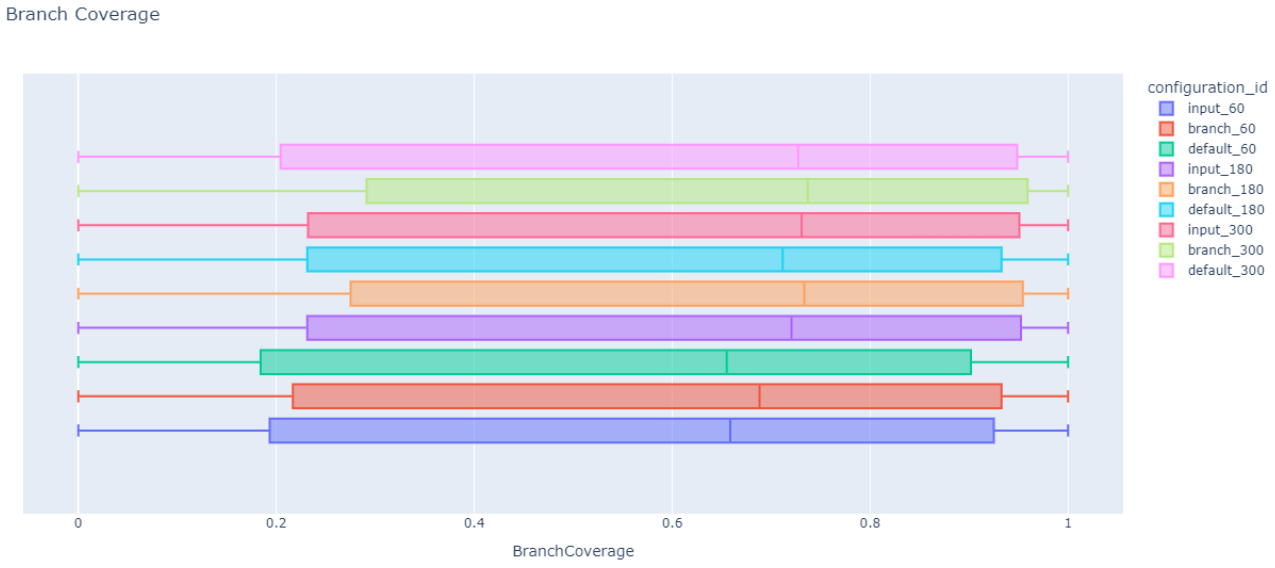

Figure 11: Data Distribution

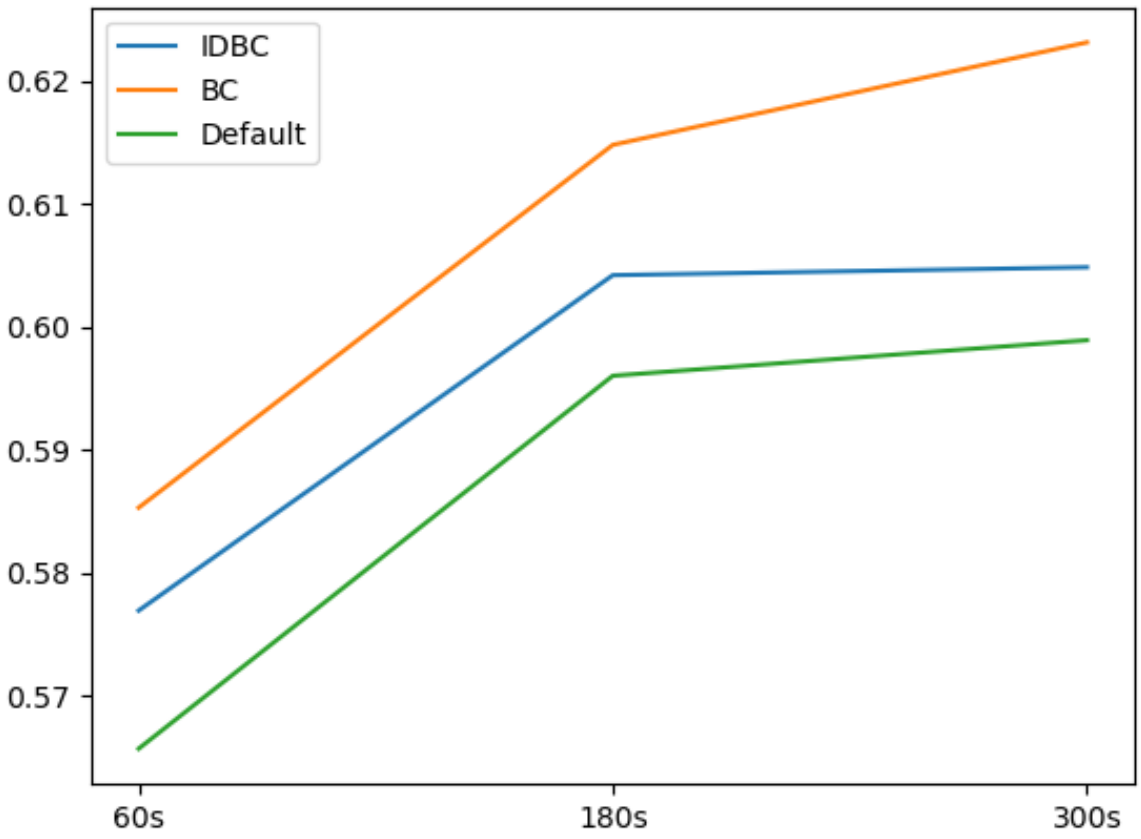

Figure 12: Change in branch coverage with respect to time