Dorylus: An ant colony based tool for automated test case generation

Dan Bruce, Héctor D. Menéndez, and David Clark

University College London, UK

Abstract. Automated test generation to cover all branches within a program is a hard task. We present Dorylus, a test suite generation tool that uses ant colony optimisation, guided by coverage. Dorylus constructs a continuous domain over which it conducts independent, multiple objective search that employs a lightweight, dynamic, path-based input dependency analysis. We compare Dorylus with EvoSuite with respect to both coverage and speed using two corpora. The first benchmark contains string based programs, where our results demonstrate that Dorylus improves over EvoSuite on branch coverage and is 50% faster on average. The second benchmark consists of 936 Java programs from SF110 and suggests Dorylus generalises well as it achieves 79% coverage on average whereas the best performing of three EvoSuite algorithms reaches 89%.

Keywords: Search-based testing · Automated test case generation · Ant colony optimisation · Dorylus.

1 Introduction

Testing software can be a long and arduous task for developers. It can take up much of the development budget for what may feel like relatively little output. However, it is crucial for checking the correctness and reliability of programs. Automated test generation aims to reduce the burden on the developer by providing test suites that, according to some criterion, effectively test the program. An important part of creating tests for programs is ensuring that much of the program is exercised by the test suite. This has led to the common goal of coverage amongst manual test developers and automation tools alike [6].

Common goals for automatic test generation tools based on coverage are: prioritisation of branches in the generation process [6], infeasibility [2] and traversing branches that depend on complex conditions such as those based on sub-regions of strings. To deal with these problems we present Dorylus. Dorylus is a search-based optimisation tool capable of generating test data for Java programs. The goal of Dorylus is branch coverage for which it requires only the binary files of the software under test. It uses a two phase search-based optimisation algorithm which prioritises targets to maximise potential coverage. It then solves each target as a separate search problem. Rather than focusing on one target at a time, the target being solved changes continually according to its

probability of selection. This reduces wasting time on infeasible targets, whilst also allowing prerequisite branches to be covered early in the process. Moreover, only targets immediately reachable from current coverage are considered at any time, creating a dynamic vector of targets.

The contribution of this paper is the Dorylus tool. For any given target the search space of the problem is reduced through a novel lightweight path-based input dependency analysis. This combined with the construction of a probability distribution over numeric inputs and leveraging Levenshtein distance to define distances for string operations, allows rapid convergence on correct inputs. Unlike other techniques, Dorylus considers paths and aims to propagate many unique paths through the program to reach the targets. This path diversity unlocks rare probability predicates, due to many of them being only feasible from a specific rare probability state of the program.

A proof-of-concept qualitative analysis has been performed, comparing Dorylus with EvoSuite to demonstrate its suitability for the problem of test generation. Both tools are tested on 12 programs, 10 of which are taken from the literature, one of these is further modified and one program constructed from interesting code constructs [1]. The results show that on all programs Dorylus at least matched EvoSuite which, on average, covered 95.1% compared with Dorylus' 97.7%. Furthermore, Dorylus was twice as fast on average, reaching maximal coverage in 20.3 seconds whereas EvoSuite took 41.8 seconds on average. To test the generalisability of Dorylus, both tools are tested on 936 programs from the SF110 benchmark. For this corpus Dorylus was compared with three algorithms in EvoSuite; whole test suite (WTS), many objective sorting algorithm (MOSA) and many independent objectives (MIO). The results suggest that Dorylus' techniques generalise well as it reaches 79% branch coverage on the corpus, compared with that of EvoSuite who achieves 89%, 86% and 88% with WTS, MOSA and MIO respectively.

2 Dorylus

Dorylus' aim is branch coverage. In order to achieve its goal Dorylus needs two pieces of information from the program that it obtains by instrumenting it. Firstly, the basic blocks that were executed in a given execution, and secondly the values of the operands and the operator at each predicate. This information will guide Dorylus' search aiming for maximum coverage. With the operator's information from a given predicate, Dorylus uses branch distance, as defined by Korel [10], to calculate how far the given test inputs are from executing the unseen branch on the other side of the predicate. Branch distance specifies how far a given predicate is from switching outcomes. For example, given a conditional statement containing the expression $a == 100$, if the outcome is false then the branch distance is $100 - a$. Korel defined such distances for all boolean operations on numeric operands. Branch distance is used for all Java primitives as they are numeric. For string comparisons, Levenshtein distance is used to defined a measure for all core Java boolean string comparison functions. This

indicates the number of characters which must be changed (inserted, swapped or deleted) in order to change the outcome of the predicate. Furthermore, as Dorylus operates on bytecode any complex boolean expressions are broken down into atomic components so need not be considered.

Uncovered branches become targets for Dorylus and this list is dynamically updated. A predicate can be observed as the guard of two branches, each labelled as $l_s \rightarrow l_d$ where l_s is the block containing the predicate and l_d is the destination block. Targets are defined as those branches where l_s is covered by the test suite but l_d is not. Therefore, as coverage changes the set of targets to be tackled is updated. When the program is executed, a trace of executed branches and the operands and operators of all executed predicates is given as output. The predicate information can be used to measure branch distance at a given location in the trace. When a test case's trace includes a previously unseen branch, it is added to the test suite.

Once the program is instrumented, some initial inputs are generated uniformly at random and ran on the instrumented program. The traces of these executions define the initial coverage and therefore provide Dorylus with a set of initial targets. Each target is a separate optimisation problem, where the fitness function is branch distance. It is important, however, that even though each search problem is separate, information should be shared between them. Therefore, all test cases that are generated are shared with all targets for which they pass through the guarding predicate. At each target, a novel lightweight dynamic path-based input dependency analysis is carried out to reduce the search space. This analysis identifies parts of the inputs whose mutation affects the outcome of the predicate, thereby drastically reducing the search space of inputs. This must be done on a path basis as depending on the path taken to the guard, different inputs may be included in the variables used in the predicate statement. For primitive numerical types we apply a search process called Ant Colony Optimisation for continuous domains ($\text{ACO}_{\mathbb{R}}$) [16]. It creates a probability distribution for each input at each target given the path taken to the guard. This is constructed by maintaining an archive of best performing test cases at each target, diversified by mandating that unique paths must be maintained and may not be removed. $\text{ACO}_{\mathbb{R}}$ controls the proximity to the target via Gaussian kernels [16]. When creating a new test case, the kernel is sampled to generate new primitive values. For String input values, rather than attempt to create a distribution, three mutations are defined based on the Levenshtein distance: insert, swap and delete. A guide test case is selected from the target's archive and the string is mutated uniformly at random a number of times according to the branch distance.

Dorylus not only works on simple programs requiring only primitives or strings, but also on more complicated real-world programs. Given a Java class to be tested, Dorylus identifies constructors and all public methods which can be called and labels them as entry points. It looks at the inputs for each method, and uses refection for those containing objects to find the object's constructor and its required inputs. This cycle is repeated until all objects have been deconstructed to primitive and string inputs and a sequence of method calls to instantiate an

object of the required type. Initial test cases call a constructor followed by a public method covering all combinations. During the search process, if for a target it appears that there are no inputs affecting the outcome, then the methods of any objects formed by the inputs are searched over. This process identifies state changing methods, such as setter methods, and then searches for the required input values. If it still appears that there are no inputs that affect the target, the process backtracks to previous targets. The aim being to find new paths to the guard predicate which can pass through the guard. When a test case is added to the test suite, the methods it calls and their inputs are sent to each target. Every target then adds a call to the method containing the target to the end of the sequence. This builds up complex sequences of method calls to explore all possible states of the program.

3 Experimental Setup

EvoSuite is a state-of-the-art test generation tool for Java programs [6]. Its aim is to generate unit tests to cover as much of a program as possible, using a genetic algorithm. EvoSuites search-based approach has featured many improvements over the years with techniques such as dynamic symbolic execution, hybrid search and testability transformations [8]. Furthermore, it has been the winner of a number of test case generation tool competitions [12].

This evaluation tests Dorylus and EvoSuite on a number of programs, and in all instances both were given 2 minutes to achieve as much coverage as possible. All experiments were carried out 10 times. The choice of 2 minutes is a tradeoff between coverage and time spent, which has been identified and used in a number of previous studies [7, 15]. Three different algorithms within EvoSuite were used; whole test suite (WTS) [6], Multiple Objective Sorting Algorithm (MOSA) [14] and Many Independent Objective (MIO) [2]. In all cases EvoSuite's only coverage criterion was branch coverage, as this is the only criterion included in Dorylus. All other setting of EvoSuite were left to their default values.

There are two corpora on which the tools have been compared. The first corpus consists of 12 programs which demonstrate constructs and programming styles using string inputs. This was selected due to Dorylus being specialised on primitive and string inputs. Furthermore, 10 of these programs have been used widely in the literature to test tools aiming to generate strings that must conform to some constraints, in order to reach parts of the program [1]. Some of the programs exhibit features which can be hard to handle for automated tools. Two programs have been added to the corpus in order to further test the tools. The first addition is to collapse multiple string inputs into a single input which is the split in the program. The other was made by combining different features of the programs to create a hard to cover program. Table I shows the complete list of all programs in the first corpus.

The second corpus was selected from the SF110[1] benchmark. SF110 contains 110 Java projects from SourceForge, the first 100 were selected to be statistically

[1] http://www.evosuite.org/experimental-data/sf110/

representative, the final 10 were the 10 most popular Java projects on SourceForge at the time [7]. SF110 consists of 23,886 programs, of this Dorylus can handle 8,398, of which 1,000 were chosen uniformly at random to judge the performance of Dorylus and test how well it generalises. If any of the 10 repetitions of a program failed due to a crash in one of the tools, we reran it up to three times. If after these reruns there were still repetitions which had failed, the program was removed from the corpus. Of the 1000 selected programs there were 64 that caused such issues and were therefore removed, leaving a final corpus of 936 classes with an average of 7.6 branches.

4 Results

Table I presents the first corpus on which the tools where tested, along with each tools average time and coverage for each program. Dorylus can be seen to be competitive on coverage with EvoSuite, matching coverage on all programs and improving coverage on two. EvoSuite covers an average of 95.1% reaching 100% on all but three programs, whereas Dorylus achieves 100% on all but one program, with an average of 97.7% branch coverage. The programs on which EvoSuite fails to reach 100% coverage are those which require optimisation of sub-regions of inputs. For example, FileSuffix takes two string input parameters, the first being the type of file and the second a file name. The file extension is then taken to be the substring after the last period in the file name. The program then checks the extension against the file type. Coverage of this program is conditional upon being able to generate an input with at least one period, and being able to find the correct substring to be placed after this period (the extension). The correct extension is dependent upon first providing a valid file type, and then matching the given extension with the one defined in the program. Dorylus covers the entire program in under 10 seconds on average, whereas EvoSuite only reaches 100% coverage on one of the ten repetitions, on average only covering 84.8%. TestProgram is similarly conditional on substring optimisation in addition to having a branch that is only feasible when a certain path is taken through the program. On this class, EvoSuite consistently reached 84.6% coverage and no higher, never covering the branch that required a specific path to be followed. This program emphasises the need for path diversity.

In terms of performance, EvoSuite is far quicker on the simpler programs without nested conditional statements and with many literals present for seeding, for example, in the case of DateParse. As programs get more complex, Dorylus' performance overtakes that of EvoSuite. On the corpus, EvoSuite reaches maximum coverage in a mean time of 41.8 seconds with a median of 14.5 seconds, compared with Dorylus' mean time of 20.3 seconds and median of 7.3 seconds. As 100% coverage is not attained in three instances for EvoSuite there are three cases of reaching the maximum time of 120 seconds, Dorylus only reaches the timeout once. An interesting program to mention is DateParse1V, which is DateParse modified so that instead of two string inputs there is only one. This input is then split so that the first three characters become the original input two, and the

Program	Branches	EvoSuite WTS		Dorylus	
		Time(s)	Coverage	Time(s)	Coverage
Calc	12	1.1	100%	6.5	100%
Cookie	13	16.6	100%	3.7	100%
Costfuns	20	1.5	100%	3.9	100%
DateParse	39	1.1	100%	31.7	100%
FileSuffix	23	119.6	84.8%	9.6	100%
NotyPevar	8	2.4	100%	1.6	100%
Ordered4	29	37.8	100%	2.1	100%
Pat	39	120	72%	120	72%
Text2Txt	23	10.5	100%	8.0	100%
Title	43	12.4	100%	11.3	100%
DateParse1V	39	58.6	100%	39.4	100%
TestProgram	13	120	84.6%	5.7	100%
Mean	25	41.8	95.1%	20.3	97.7%
Median	23	14.5	100%	7.3	100%

Table 1. Results on the first corpus for both EvoSuite and Dorylus. It shows the mean time and coverage for each tool over 10 repetitions on each program.

remaining characters the original input one. The control flow of the program is left unaffected by this change. Despite a small change to the representation, the time taken for EvoSuite to cover the program jumps from 1.1 seconds up to 58.6 seconds. Whereas Dorylus only increases from 31.7 to 39.4 seconds. DateParse1V demonstrates that Dorylus can be more resilient to representation changes than EvoSuite and is quicker to handle substring optimisation.

In the second corpus there are 936 real-world Java classes upon which the tools were tested, the results for which can be seen in Table II. All three algorithms within EvoSuite performed similarly, with WTS reaching 89.5%, MOSA 85.7% and MIO 87.6%. Dorylus reached 79.1% in 30.2 seconds on average. This suggests that the techniques used in Dorylus generalise well, given that EvoSuite is a mature state-of-the-art tool. Interesting to note is the number of programs on which each algorithm achieved higher coverage than the other three. Dorylus outperforms all EvoSuite configurations on 29 programs, suggesting that there is a set of programs on which its techniques are more effective.

However, many of the classes within SF110 are small, with many having no branches at all. As such results for the largest 100 classes of the 936 are also presented. These classes, are substantial with an average of 42 branches and minimum and maximum of 24 and 273 branches respectively. Coverage was much lower, Dorylus reaches 31.29% coverage and EvoSuite 54.56%. However, it can be seen that Dorylus is much quicker on these larger programs. This suggests that it is either failing or stopping the search process early and as such future work will include more randomness in the search in order to continue to improve given more time. As can be seen in Table II, there are nine classes of the largest 100 on which Dorylus achieves higher coverage than EvoSuite, implying that Dorylus' technique could be used to complement EvoSuite in specific circumstances.

Possible threats to validity include the different implementations of the tools. As previously mentioned, all three algorithms in EvoSuite achieve similar results. It would be worth investigating an implementation of Dorylus in EvoSuite to get a better comparison with less confounding factors.

Classes	Branches	Algorithm	Coverage (%)	Time(s)	Best
All 936	7.6	WTS	89.47	32.6	58
		MOSA	85.66	29.4	44
		MIO	87.58	36.2	19
		Dorylus	79.12	30.2	29
Largest 100	42.4	WTS	54.56	101.9	16
		MOSA	54.28	98.49	27
		MIO	51.85	112.1	4
		Dorylus	31.29	75.67	9

Table 2. Results on the second corpus for WTS, MOSA, MIO and Dorylus. It shows mean time, coverage and the number of programs where each tool gets the higher coverage than all other approaches.

5 Related Work

There are many search-based methods that can be used for automated test generation [11]. What these algorithms have in common is that they obtain solutions guided by a fitness function [9]. The way in which these test cases are created or how the search is guided is what distinguishes these methods from one another. In the context of automated test generation the most prominent search based method used is Genetic Algorithms [4]. A well-known example is the work of Fraser et al. on EvoSuite [5]. EvoSuite produces unit tests for Java programs using a genetic algorithm with some coverage metrics as the fitness functions. It has been tested for both unit testing and system testing and has achieved the highest score compared with state of the art tools for a number of years [12]. Owing to its success a number of algorithms have been integrated into the EvoSuite tool.

Firstly, the Many Objective Search Algorithm (MOSA) uses a multi-objective approach, where every target is an objective [14]. Many test cases are compared based on their performance across all objectives in order to obtain the best performing test cases. In this approach, infeasible branches may cause wasted effort, as targets that have a control dependency on an infeasible branch are infeasible targets, and therefore not worth considering. This improvement was implemented in Dynamic MOSA (DynaMOSA) where targets are updated to only include those immediately reachable from currently covered branches. Dorylus uses a similar design by updating targets dynamically [13]. In doing this, an infeasible region in the control flow graph will only be represented as a single

objective, the branch guarded by the infeasible predicate. This approach was proved to be at least as good as the original algorithm [13]. DynaMOSA creates many test cases which are scored according to their position in the Pareto front of all objectives. Conversely, our work aims to prioritise targets in order to speed up exploration of required ancestors before effort is spent on descendants. It is possible for the Pareto front of DynaMOSA to be affected by infeasible targets with very low branch distances, although they will have less impact on performance than on MOSA. Unfortunately we were unable to compare with DynaMOSA as it was not in the release of EvoSuite used in the experiments.

The existing tool most similar to Dorylus is Many Independent Objectives (MIO), in which a population is maintained for each of the objectives [2, 3]. MIO was compared against random search, MOSA and whole test suite and found to outperform all three [3]. The goal of MIO is to produce the highest covering test suite in the allotted timeframe. It handles this goal starting with the easiest to solve branches, which provide immediate coverage. This means that where MIO punishes complex and difficult to solve targets, Dorylus puts effort into attempting to solve them unless they are heuristically identified as infeasible. Infeasible branches are implicitly ignored by MIO as it prioritises those targets for whom better performing test cases are being found. Once a plateau is reached for an infeasible branch no new test cases will be found. Dorylus aims to get many paths through to the targets, thereby introducing path diversity. In the case that current members of the archive are unable to affect change on a predicate, Dorylus backtracks to the previous target in order to find new paths. Backtracking can take many steps in order to find new rare probability paths, which are needed to unlock specific regions of code. On the other hand, MIO has a counter for each target which is incremented on every attempt to find a new test case, and resets when a test case is added to the archive. Therefore, if the correct path does not make it to the target, it will implicitly be seen as infeasible and therefore given less attention.

6 Conclusion

We have put forward a new tool to generate input data to cover hard to reach regions of code. It is a search-based method, founded on the ideas of Ant Colony Optimisation, with novel heuristics to assist in unlocking difficult predicates. The concept of a conditional statementś dependency on an input variable for a specific path has proved in practice to work well and assist in speed and quality of coverage. Our tool has been compared against EvoSuite, a highly successful test case generation tool. We demonstrated that Dorylus can outperform EvoSuite on programs with complex predicates and code structure. Furthermore, we have shown Dorylus can handle a large number of real-world programs and provide good coverage. Two steps for future work have been identified. Firstly, to incorporate other coverage criterion to improve the effectiveness of the generated test suite. The second step is to combine the approaches of Dorylus with those of EvoSuite

to create a hybrid thereby improving EvoSuite's effectiveness on the programs which Dorylus beat it.

References

1. Alshraideh, M., Bottaci, L.: Search-based software test data generation for string data using program-specific search operators. Software Testing, Verification and Reliability **16**(3), 175–203 (2006)
2. Arcuri, A.: Many Independent Objective (MIO) Algorithm for Test Suite Generation. In: International Symposium on Search Based Software Engineering. pp. 3–17. Springer (2017)
3. Arcuri, A.: Test suite generation with the Many Independent Objective (MIO) algorithm. Information and Software Technology **104**, 195–206 (2018)
4. Campos, J., Ge, Y., Fraser, G., Eler, M., Arcuri, A.: An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation. In: International Symposium on Search Based Software Engineering. pp. 33–48. Springer (2017)
5. Fraser, G., Arcuri, A.: 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. pp. 416–419. ACM (2011)
6. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering **39**(2), 276–291 (2013)
7. Fraser, G., Arcuri, A.: A large-scale evaluation of automated unit test generation using evosuite. ACM Transactions on Software Engineering and Methodology (TOSEM) **24**(2), 8 (2014)
8. Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: Software Reliability Engineering (ISSRE), IEEE 24th International Symposium. pp. 360–369 (2013)
9. Harman, M., Clark, J.: Metrics are fitness functions too. In: Software Metrics, 2004. Proceedings. 10th International Symposium on. pp. 58–69. Ieee (2004)
10. Korel, B.: Automated software test data generation. IEEE Transactions on software engineering **16**(8), 870–879 (1990)
11. McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability **14**(2), 105–156 (2004)
12. Panichella, A., Campos, J., Fraser, G.: EvoSuite at the SBST 2019 Tool Competition. In: International Workshop on Search-Based Software Testing (SBST). pp. 29–32 (2019)
13. Panichella, A., Kifetew, F., Tonella, P.: Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. IEEE Transactions on Software Engineering **44**(2), 122–158 (2018)
14. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: Software Testing, Verification and Validation (ICST), IEEE 8th International Conference on. pp. 1–10. IEEE (2015)
15. Panichella, A., Kifetew, F.M., Tonella, P.: A large scale empirical comparison of state-of-the-art search-based test case generators. Information and Software Technology **104**, 236–256 (2018)
16. Socha, K., Dorigo, M.: Ant colony optimization for continuous domains. European journal of operational research **185**(3), 1155–1173 (2008)