Ant Colony Optimization for Object-Oriented Unit Test Generation

Dan Bruce[1], Héctor D. Menéndez[2], Earl T. Barr[1], and David Clark[1]

[1] University College London, London, UK
{dan.bruce.17,e.barr,david.clark}@ucl.ac.uk
[2] Middlesex University London, London, UK
h.menendez@mdx.ac.uk

Abstract. Generating useful unit tests for object-oriented programs is difficult for traditional optimization methods. One not only needs to identify values to be used as inputs, but also synthesize a program which creates the required state in the program under test. Many existing Automated Test Generation (ATG) approaches combine search with performance-enhancing heuristics. We present Tiered Ant Colony Optimization (Taco) for generating unit tests for object-oriented programs. The algorithm is formed of three Tiers of ACO, each of which tackles a distinct task: goal prioritization, test program synthesis, and data generation for the synthesised program. Test program synthesis allows the creation of complex objects, and exploration of program state, which is the breakthrough that has allowed the successful application of ACO to object-oriented test generation. Taco brings the mature search ecosystem of ACO to bear on ATG for complex object-oriented programs, providing a viable alternative to current approaches. To demonstrate the effectiveness of Taco, we have developed a proof-of-concept tool which successfully generated tests for an average of 54% of the methods in 170 `Java` classes, a result competitive with industry standard Randoop.

1 Introduction

Generating unit tests for object-oriented programs is so difficult that the conventional wisdom in the ACO community is that Automated Test Generation (ATG) for complex object-oriented programs (OOP) is not currently possible for ACO. Indeed, in 2015 Mao et al. [23] said “for complex types such as String or Object, the current coding design in ACO cannot effectively handle them”, and in 2018 Sharifipour et al.[27] identified generation of strings and objects as future work. Solving ATG for OOP requires calling methods in the correct order, with the correct inputs in order to explore the unit under test. This is a problem with a gigantic search space. Solving it automatically would be highly profitable, both in terms of time saved and potential increased coverage of a program.

ATG techniques can be broadly classified as static or dynamic, i.e. those that only observe the code or those that execute it. In recent years, many dynamic approaches have used genetic algorithms (GA) [18]. GAs typically mutate and crossover candidate solutions, which, in the case of creating test programs, can lead to invalid states. Instead, following pheromone levels attributed to available methods produces legitimate test programs, guided by the fitness of previous tests. It is this observation that has motivated our exploration of ACO for object-oriented ATG. ACO has been applied to generating test cases for programs in the past. However, those applications were dominated by numerical programs, where the problem was simply finding the required values of primitive inputs [5,8,23,27]. Whilst these works show ACO's effectiveness at test data generation, they do not support its applicability to automated test generation for real world object-oriented software.

We introduce Taco, a Tiered Ant Colony Optimization algorithm that can generate complex test cases for complex object-oriented programs. Taco does so by following three tiers: 1) it selects a test coverage goal within the program under test, 2) it synthesizes test programs by creating sequences of methods, and 3) it generates numeric and string data values required as inputs by the test program. It is, to the best of our knowledge, the first complete ACO technique capable of generating valid `Java` test cases. Taco has been evaluated on 170 `Java` classes taken from SF110 [14], a well known `Java` testing benchmark, and successfully created tests for 54% of methods per class, covering an average of nearly 50% of lines of code. Taco achieves higher branch and line coverage than the industry standard tool, Randoop, not all of which overlaps, suggesting that additional engineering to cover further constructs would yield significant improvements. Taco demonstrates the potential of ACO for automated test generation for object-oriented code; further research and engineering effort may allow ACO to compete with the current state of the art in ATG.

Contributions

- Taco is the first complete ACO technique capable of creating real test cases for complex object-oriented programs.
- Taco's Tier II synthesizes test programs by building sequences of method calls, thereby creating complex objects required as inputs (Section 3.3).
- Taco has been realized as a tool and used to generate JUnit tests for real `Java` classes competitively with Randoop (Section 4).

2 Related Work

Relatively little research into automated test generation uses ant-based approaches [18,23]. Those that have applied ACO to software testing have focused on generating useful input values. The classical ACO-based test generation process typically follows 3 main steps: 1) partition the input space, 2) project each partition into each dimension or variable and, 3) decrease the partition granularity for those parts which are more interesting for the test purpose. In this way, each pair (partition, dimension) becomes a node for the graph that an ant can traverse to find inputs for a program [23]. This approach is usually applied to discreet domains, but there are several variations. Ayari et al. [5], for example, use $ACO_{\mathbb{R}}$, which is an ACO variant that was developed to optimize values from continuous domains [28]. They find numeric inputs required by specific methods. Similarly , Mao et al. [23] and Sharifipour et al. [27] independently compare ACO with other metaheuristics on small numerical programs. Both of these approaches generate values only for a specific method within the program under test, and they do not consider object-oriented programs. However, in both cases the ACO approach outperforms the other meta-heuristics. In previous work, we used an extended version of $\text{ACO}_{\mathbb{R}}$ for data generation, including strings, and then used heuristics to create objects and call methods [9]. In contrast, Taco uses ACO to synthesis a test program and build objects prior to data generation. Vats et al. [31] gather different ACO applications on software testing, where some of the work focus on OOP testing. The most relevant either generate simple inputs for programs following the classical method of partitioning the domain [12,22], or focus on other aspects of testing like iteration testing [11] and test suite minimization [29]. None of this work considers generating sequences of methods and input values as part of the same process, as Taco does.

Current, pure ACO test generation methodologies cannot deal with object-oriented programs, for which they need to create sequences of methods for each test case. Srivastava et al. proposed a technique related to our Tier II and generated test sequences of events, although the events considered did not require inputs [29]. Our technique has a stage after sequence synthesis which generates instances of all required inputs. ACO-based program synthesis is known as Ant Programming (AP) [26]. This field, inspired by genetic programming (GP), aims to create programs using ACO methods. Some examples are the work of Rojas and Bentley, which used ACO to synthesize programs which solve boolean functions [25], and Toffola et al. that compared A* and ACO for guiding program synthesis and found ACO to be effective at solving bottlenecks [30]. GP has been directly applied to the task of defining OOP test programs [32,16]. These techniques build trees of method calls and their required parameters. However, we contend that pheromone-guided synthesis is more suitable for generating method sequences than GP's mutation and crossover operators [32]. Taco creates test programs that are correct by construction, whereas mutation and crossover of existing test programs, as is done in GP, can lead to invalid programs that need to be fixed or discarded. Other approaches combine GP and ACO for program synthesis, such as the work of Hara et al. [17]. This combination is called Cartesian Ant Programming and is normally applied to circuits [17,21]. Although this work does solve some programming problems, to the best of our knowledge, there currently exists no pure ACO-based work that synthesizes complex `Java` programs.

3 TACO Algorithm

Solving ATG entails generating programs. Programs are discrete objects. Viewed as a combinatorial optimization problem, ATG is infeasible, because the search space is vast and under-constrained. Crucially, test programs require not only syntactic correctness but also semantic correctness in order to build a valid program state. Others have sought to constrain the problem via reformulation as multi-objective search; this tack constrains the fitness of solutions, but not the search space. Our key insight is that ATG naturally decomposes into a three deep hierarchy of successive subproblems – and each subproblem further constrains the overall problem. The three tiers are (1) goal selection, (2) test program synthesis, and (3) primitive and string types data generation.

Tier I prioritizes and selects goals within the program under test. As its performance metric is branch coverage, it targets a branch. In each iteration, Taco focuses on covering one branch at a time. Taco gains great leverage from its first tier: by selecting promising goals, Taco restricts the second two tiers to only the relevant subset of the search space. Taco's program synthesis (Tier II) is its core novelty. ATG of OOP is unique in that it often requires building a valid program state to reach a goal. This subtask requires calling a potentially large number of methods in sequence. A simple example is a goal guarded by an `if` that depends on a variable, which, in turn, can only be set by calling a method. Tier II searches over possible test programs, building objects and generating sequences of methods with data holes for primitives and strings. ACO shines at this subtask, quickly finding sequences of available methods that reach the goal. Finally, the last tier, data generation, is well-understood and well-solved by the ACO community [28]: here, ACO efficiently finds primitive and string values to fill the data holes in Tier II's method sequences.

Taco's three tiers operate independently. Therefore, one can easily replace or modify the specific algorithm each tier uses if a new state of the art emerges for its task — this could be another ACO variant or an entirely different algorithm. For example, TIII's algorithm could be modified to exploit research into numerical data generation when testing numerical programs.

Branch distance measures how far a conditional statement's control expression is from a different evaluation outcome in a particular program state. For example, the branch distance of numerical equality, $a == b$, is often computed as $|a - b|$. Given the conditional statement `x == 10`, `x := 8` creates a program state that is a distance of 2 from evaluating to true, while `x := 100` creates a state that is 90. Branch distance has long been used in ATG. All three of Taco's tiers use it as their fitness function to evaluate ants. For numerical conditional statements, Taco uses Korel's measures [20]; for string comparisons, it uses Levenshtein distance [2]. Often in ATG, approach level guides branch distance. Approach level is the number of control dependent statements, required to reach a target, that were not executed [6]. Tier I defines goals such that their approach level is always 0 (Section 3.2), so Taco does not use it.

Taco has two phases, shown in Figure 1. The first phase generates ants using three-tiered ACO. The second phase converts the ants into test cases, which it executes, observes and then uses to calculate fitness. Taco uses these fitnesses to update its ant archives and pheromones lists for both goals and methods. Although Taco generates an ant for a specific goal, that ant may be effective, or even cover, other goals. Therefore, once an ant's test case is executed, Taco checks for incidental coverage and passes the ant to every goal whose source

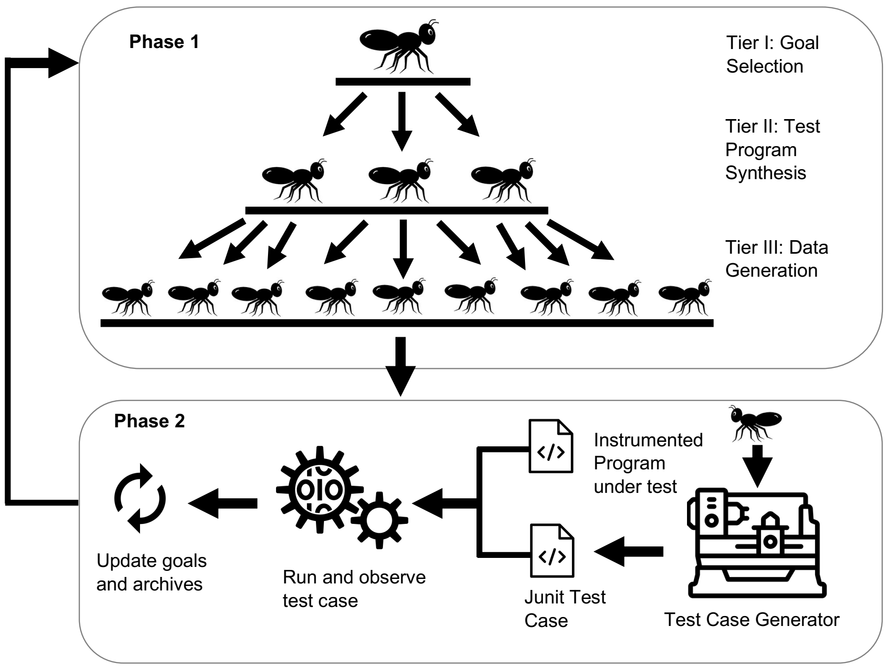
Phase 1
Tier I: Goal Selection
Tier II: Test Program Synthesis
Tier III: Data Generation
Phase 2
Instrumented Program under test
Update goals and archives
Run and observe test case
Junit Test Case
Test Case Generator

Fig. 1. An overview of the TACO approach. Phase 1 generates new ants. The distribution of work amongst the tiers is tunable; TACO chooses 1 goal per iteration, synthesizes 5 test programs per goal and generates 20 sets of data values per test program. Phase 2 converts each ant from phase 1 into a JUnit test case, then executes the instrumented program under test on that test case. Finally, TACO updates its goals and archives based on an ant's coverage and fitness.

unit it covered. TACO then checks the ant against a goal's best so far archive, and updates the pheromones and archives of each method in the ant's test program. If the ant is the new best for a goal, TACO resets the goal's counter. If an ant covers a new branch or block, TACO adds it to the test suite. TACO removes covered goals and updates the list of goals to include those that now meet the criteria (Section 3.2).

3.1 Problem Definition

A program, P, can be viewed as a Control Flow Graph (CFG) where the nodes are basic blocks, and edges are branches. A basic block (subsequently called block) is a sequence of statements which has one entry point, one exit point and, if the first statement is executed, all statements must be executed once and in order [1]. Every block has a unique label and the set of all blocks within P is denoted by L. A branch is defined as a transition of execution from one block to another $l_i \rightarrow l_j$ written as b_{ij} with B as the set of all branches.

A test case, t_k, is a program which calls P with some input and has a test oracle that checks the correctness of some program state [7]. This may be a specific output or merely the absence of failure during execution. From execution of t_k, one can observe which blocks and branches which have been covered. A test suite is a collection of test cases, where commonly the goal is some form of coverage, be that block, branch or some other criterion. Branch coverage of a test suite with n test cases is $C = \bigcup_{k=1}^{n} branches(t_k)$ and block coverage $U = \bigcup_{k=1}^{n} blocks(t_k)$.

3.2 Tier I: Goal Prioritization and Selection

We define goals to be $G = \{b_{i,j} \mid l_i \in U \wedge b_{i,j} \notin C\}$, thereby restricting goals to only those uncovered branches whose source block has been covered. This prevents allocating resources to uncovered branches that are control dependent upon another uncovered branch. At the start of each iteration, Taco selects one goal for which it generates a number of test cases. Taco uses an Ant System to select goals: a goal's probability, $p(g)$ in Equation 1, is based upon its pheromone level τ_g, which is the number of uncovered branches that can be reached from the goal, and the heuristic value $\eta_g = (1 - c_g \cdot \delta)$. Each selection of a goal increases a counter, c_g, which is multiplied by the decay factor, δ (0.01 for Taco).

$$p(g) = \frac{\tau_g \cdot \eta_g}{\sum_{k \in G} \tau_k \cdot \eta_k} \tag{1}$$

Pheromone does not decay, instead the heuristic is used as a decay mechanism, using a counter to decay rather than reducing the pheromone at every time-step. Taco enforces a minimum pheromone level of 0.1. This process favours goals that lead to larger regions of uncovered code, and those for which Taco is regularly discovering new, best test cases. The counter helps to avoid wasting time on infeasible goals, as once Taco gets as close as is possible, the counter will not be reset again and the probability of selecting the target will only decrease. Previous ATG tools have used counters in this way [3].

3.3 Tier II: Test Program Synthesis

In Tier II of Taco, we take a non-traditional approach to program synthesis: holes in our program are considered as data holes and are missing primitive and string data values, not arbitrary code fragments [15]. Furthermore, to the best of our knowledge, we are the first to apply ACO to object-oriented program synthesis, which has been dominated by enumeration and constraint solving. For each goal selected by Tier I, many test programs (ants) are synthesized to allow optimization towards a covering test program (five in our implementation). Each goal has an archive of the best performing ants and pheromone levels for each method that has been called by ants considered the goal. Algorithm 1 and 2 show the pseudocode for Tier II.

When deciding which test program to execute next, Taco, can select an existing test program from the archive, or synthesize a new one. At line 1 of

Algorithm 1 Tier II: The **testSynth** algorithm builds a test program, represented as a sequence of method calls, $\langle m, i, o\rangle$. The *select*$_{1,2}$ and *getAvailableMethods* functions are described in the text. *buildMethodSeq* calls Algorithm 2.

```
Input: $P$, the program under test.
Input: $g$, the goal selected in Tier I.
Input: $A_g$, a list of previously generated $M$s ordered by performance at $g$.
Input: $s_g$, a function that outputs the pheromone of a method at $g$.
Output: $M$, a sequence of method call tuples with holes for primitive or string data.
1: if $random(0.0, 1.0) > select\_threshold$ then
2:     return select$_1(A_g)$ {This helper function is described in text.}
3: $M := \langle\rangle$
4: repeat
5:     $M_a$ := getAvailableMethods(P, M) {methods in scope}
6:     $m := select_2(M_a, s_g)$ {This helper function is described in text.}
7:     $M := buildMethodSeq(M, method, s_g)$ {See Algorithm 2}
8: until ($resources\ exhausted \lor m = NULL$)
9: return $M$
```

Algorithm 1, a global parameter dictates the probability of selecting versus generating a test program (in our implementation the probability of either is 50%). A test program is selected from the archive, *select*$_1$ on line 2, with probability proportional to its position within the archive. As the archive is sorted by branch distance, the test program with the lowest branch distance is the most likely.

At first, available methods are constructors or static methods of the program under test. Then, moving forwards, any method of an object that has been instantiated within M and in scope. The function s_g returns the current pheromone level of a method with respect to goal g. A method's pheromone starts at ρ_0; ants that perform well, and are added to a goal's archive, add pheromone to each method they visit. Pheromone change is shown in Equation 2, where n is the number of ants added to the goal's archive that call method m_i, a is the number of ants generated that call m_i and γ_m, and δ_m are algorithm parameters that dictate amount of pheromone laid and removed[3]. Therefore, pheromone decays every time a method is added to a test program, rather than at each time-step. At time N, $\rho^N_{m_i}$ gives the pheromone of method m_i in Equation 3.

$$\Delta\rho_{m_i} = (n \times \gamma_m) - (a \times \delta_m) \qquad (2) \qquad\qquad \rho^N_{m_i} = \rho^{N-1}_{m_i} + \Delta\rho_{m_i} \qquad (3)$$

At each step, Taco selects a method, line 6 of Algorithm 1 (*select*$_2$), probabilistically according to pheromone levels of available methods. Taco can choose to end the test program before reaching the max length at this point, by selecting *NULL* in place of a method. When adding the selected method to the sequence in Algorithm 2, any primitive or string values required are left as holes within the program (line 6). Tier III later searches over the input domain of these holes

[3] For our implementation of Taco the following values were used: $\rho_0 = 50$, $\gamma_m = 0.5$, $\delta_m = 0.05$. With a minimum pheromone of 1 and maximum of 100.

Algorithm 2 Tier II: This **buildMethodSeq** adds a method call, *m*, its parameters, and a reference to its output, to the sequence of methods being generated, *M*. It is recursive, because some of *m*'s parameters might be methods or be an abstract data type one of whose constructors we must call. **buildMethodSeq** leaves data holes in the sequences for primitive or string parameters. $select_2$ and *insert* are described in the text.

Input: *m*, the method selected to be added to the test program.
Input: *M*, the method sequence (test program) which *m* should be added to.
Input: s_g, a function that outputs the pheromone of a method at *g*.
Output: *M*, a sequence of method call tuples with holes for primitive or string data.

```
 1: $inputs := \langle\rangle, rid := NULL$
 2: if $!isVoid(m)$ then
 3:     $rid := getNonce()$
 4: for all $p \in getParameters(m)$ do
 5:     if $instanceof(p) \in primitives \cup \{String\}$ then
 6:         $inputs$ += $HOLE : instanceof(p)$
 7:     else
 8:         $C_a := getAvailableConstructors(p)$
 9:         $c := select_2(C_a, s_g)$
10:         $M := buildMethodSeq(M, c, s_g)$ {Recursive call, returns M with c inserted}
11:         $inputs$ += $getRid(M, c)$
12: $M := insert(M, \langle m, inputs, rid\rangle)$ {This helper function is described in text.}
13: return $M$
```

to find a set of instances that minimize branch distance to the goal. Object parameters are referenced using their *rid*, an output identifier which is independent of position within sequence. When the tuple defining a method call is added to a test program, line 12 of Algorithm 2 (*insert*), it is injected at the last position in the sequence where it still has an affect. For example, Figure 2 shows the JUnit representation of a sequence of method calls. When `v1.methodUserObj()` (line 6) was selected, it had to be inserted after `v1` was defined (line 5), but before it was used (line 7).

Tier II's **testSynth** generates sequences, but a natural way to view them is as programs with data holes. The JUnit test case in Figure 2 is obtained from the following sequence

```
M = ⟨(new ExClass(), ⟨⟩, v0),(new UserObj, ⟨HOLE: String⟩, v1),
(v1.methodUserObj, ⟨⟩, NULL),(v0.method1(), ⟨v1⟩, v2),
(v0.setValue, ⟨HOLE: int⟩, NULL), (v0.target(), ⟨⟩, NULL)⟩
```

To construct *M*, **testSynth** takes a goal, assumed to be within `target()`. Early in the search-process, **testSynth** generates nearly random method sequences, as pheromone levels initially provide no guidance. As Taco iterates, methods that create states that execute branches close to the goal will accumulate pheromones. Pheromone levels will rapidly suggest selecting `target()`. The selection of `v0.method1()` triggers **addMethodSeq**, which processes `v0.method1()`'s parameter of type `UserObj`. **addMethodSeq** then probabilistically selects one of

```java
public class ExampleClassTest() {
    @Test
    public void testMethod1() {
        ExClass v0 = new ExClass();
        UserObj v1 = new UserObj(<HOLE:String>);
        v1.methodUserObj();
        boolean v2 = v0.method1(v1);
        v0.setValue(<HOLE:int>)
        v0.target()
    }
}
```

Fig. 2. Output of Tier II: a method sequence realized as a JUnit test program with data holes.

`UserObj`'s constructors, relying on the pheromone levels laid by ants in previous iterations. This constructor builds the `v1` *rid*, which **addMethodSeq** passes to `v0.method1()`. While **addMethodSeq** does not directly change pheromone levels, it does indirectly affect them: its addition of methods to a method sequence means that ants will traverse and update those method's pheromone levels in subsequent iterations. **addMethodSeq**'s addition of `UserObj`'s constructor makes `v1.methodUserObj()` available to subsequent iterations, as the `v1` *rid* is within the method sequence.

3.4 Tier III: Input Data Generation

Having progressed through the two previous tiers, the search space has been reduced from all valid test programs for the program under test to the input domain of the primitive and string holes. For Figure 2, the input domain is one `String` and one `int`. For each test program, there are still a huge number of possibilities, which is why the optimization process samples many possible values for each (20 in the case of TACO's implementation).

A goal has an archive of primitive and string values for every method which has been called in a test program considered at the goal. Each of these archives operates in accordance with ACO$_{\mathbb{R}}$, allowing new values to be sampled based on the contents of the archive [28]. When values are needed for a method, a guide is selected from the method's archive based on position. For string values the method for sampling is as in Dorylus [9], mutating the guiding value by inserting, removing and swapping characters. For primitives, the guide value is used to define a Gaussian distribution, from which TACO samples a new value for each variable. Equation 4 and 5 are the taken from directly from ACO$_{\mathbb{R}}$.

$$G_e^d(x) = \frac{1}{\sigma_e^d\sqrt{2\pi}} e^{-\frac{\left(x - v_e^d\right)^2}{2\sigma_e^{d2}}} \quad (4) \qquad \sigma_e^d = \zeta \sum_{l=1}^{k} \frac{abs\left(v_l^d - v_e^d\right)}{k-1} \quad (5)$$

v_e^d is the value of variable d in the guide e. The standard deviation, σ_e^d, is calculated as the mean difference between the guide and all other values in the archive Equation 5, its size controlled by ζ.

When an archive has spare capacity, Taco adds the input values of any ant that calls the method to it. Once capacity is reached, the ant must have a smaller branch distance than the current bottom of the archive.

This tier is where most related work on automated test generation operates, with the holes in a test program forming the vector of inputs for the ant algorithm. As such, the specific variant of ACO used could easily be swapped and experimented with, which we plan to do in future work.

4 Evaluation

To evaluate Taco, we implemented it in `Java`. It instruments the class under test and obtain control flow graphs for methods within classes. Taco handles arrays and lists, treating length as an integer hole and the contents as parameters. Taco does not currently handle other `Java` builtins, such as maps, sets, stacks etc. Our implementation used parameter values as given in Section 3; please note: these are not optimized values. Future work will study the effects of different values and search for optimal default settings.

We evaluated Taco's ability to automatically generate JUnit tests for 170 `Java` classes. These classes are part of the SF110 corpus [14]. SF110 contains 23,886 classes from 110 `Java` projects selected from SourceForge between 2012 and 2014. We selected these 170 classes uniformly at random. They came from 46 projects, and have an average of 21 branches, 66 lines of code and 16 methods each. When testing, we allowed two minutes of test case generation per class (each repeated ten times). The process of compiling, running and measuring coverage of the test suites was performed after, and not timed. Coverage data was obtained by running the output test suite on the original class with JUnit and Jacoco[4].

The state of the art in ACO applied to ATG does not handle object-oriented programs. Our central result is that Taco is the first ACO approach to ATG for object-oriented programs: Taco successfully generated test cases for an average of 54% of methods across the 170 classes, covering nearly 50% of lines of code. `Java` is a large language with huge industrial uptake. Generating test suites for the remaining 46% of methods would rely on further engineering to implement all of `Java`'s many constructs. These include filesystem and network interactions, which Taco has no control over.

We ran the same experiments with two highly developed, industry standard, `Java` unit test generation tools; Randoop and EvoSuite. Randoop has been under active development for over a decade, it uses feedback-directed random test generation to build a test suite for the class or program under test [24]. It has found previously unknown errors in widely used libraries and is currently used in industry[5]. It has been used as a baseline in the Search-Based Software

[4] JaCoCo is a free code coverage library for `Java`: https://www.eclemma.org/jacoco/

[5] https://randoop.github.io/randoop/

	Coverage Criterion				
Tool	Branch	Line	Instruction	Complexity	Method
Randoop	19.0%	48.3%	44.3%	46.7%	56.0%
Taco	20.2%	48.7%	47.9%	47.5%	54.2%
EvoSuite	47.5%	70.3%	69.1%	70.2%	78.4%

Table 1. Average coverage of Randoop, Taco and EvoSuite on the 170 classes selected from SF110, as reported by Jacoco; Taco's performance respectably falls between two state-of-the-art ATG tools that have enjoyed substantial, longterm, and ongoing engineering effort.

Testing (SBST) tool competition, where it achieved the second highest score out of five tools in 2019 [19]. EvoSuite is the state of the art in search-based unit test generation [13]. Similarly to Randoop, it has been actively developed for close to a decade. At its core it uses a genetic algorithm but has become a collection of state of the art techniques for generating unit tests for `Java` (including filesystem and network mocking [4]). The prowess of EvoSuite is demonstrated by the fact it has won six of seven recent Search-Based Software Testing (SBST) tool competitions [10].

Despite the huge engineering advantage of Randoop, Taco's results are promising, beating Randoop in all measures except method coverage (Table 1). EvoSuite's combination of advanced search techniques and enormous engineering effort allows it to generate tests for 78% of methods on average, covering close to 50% of branches. Unsurprisingly, it beats both Taco and Randoop in all measures. For future ACO ATG for OOP variants, EvoSuite both defines a performance target to meet (or beat) and, given EvoSuite's history of amalgamating best-of-class search techniques, a target to join and extend.

5 Conclusion

This paper has presented a novel Ant Colony Optimization algorithm, Taco, which applys ACO to object-oriented unit test generation. Taco combines a unique tiered structure with a new ACO technique for synthesising test programs for object-oriented code. We have developed a prototype tool which implements Taco and have run it on real `Java` programs, generating tests for more than 50% of methods, on average. ACO is a powerful meta-heuristic and we hope that this paper has served as a proof of concept that it can be used to generate complex test cases for complex object-oriented programs. Future work will close the engineering gap between Taco and the other tools to provide a framework for comparing ACO variants in the domain of object-oriented ATG.

References

1. Allen, F.E.: Control flow analysis. In: ACM Sigplan Notices. vol. 5, pp. 1–19. ACM (1970)

2. Alshahwan, N., Harman, M.: Automated web application testing using search based software engineering. In: International Conference on Automated Software Engineering (ASE). pp. 3–12. IEEE/ACM (2011)
3. Arcuri, A.: Many Independent Objective (MIO) Algorithm for Test Suite Generation. In: International Symposium on Search Based Software Engineering (SSBSE). pp. 3–17. Springer (2017)
4. Arcuri, A., Fraser, G., Galeotti, J.P.: Generating tcp/udp network data for automated unit test generation. In: Joint Meeting on Foundations of Software Engineering (ESEC/FSE). pp. 155–165. ACM (2015)
5. Ayari, K., Bouktif, S., Antoniol, G.: Automatic mutation test input data generation via ant colony. In: Annual Conference on Genetic and Evolutionary Computation (GECCO). pp. 1074–1081. ACM (2007)
6. Baars, A., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., Vos, T.: Symbolic search-based testing. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). pp. 53–62. IEEE (2011)
7. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. Transactions on Software Engineering **41**(5), 507–525 (2014)
8. Bidgoli, A.M., Haghighi, H.: Augmenting ant colony optimization with adaptive random testing to cover prime paths. Journal of Systems and Software **161**, 110495 (2020)
9. Bruce, D., Menéndez, H.D., Clark, D.: Dorylus: An ant colony based tool for automated test case generation. In: International Symposium on Search Based Software Engineering (SSBSE). pp. 171–180. Springer (2019)
10. Campos, J., Panichella, A., Fraser, G.: Evosuite at the sbst 2019 tool competition. In: International Workshop on Search-Based Software Testing (SBST). pp. 29–32. IEEE/ACM (2019)
11. Chen, X., Gu, Q., Zhang, X., Chen, D.: Building prioritized pairwise interaction test suites with ant colony optimization. In: International Conference on Quality Software. pp. 347–352. IEEE (2009)
12. Farah, R., Harmanani, H.M.: An ant colony optimization approach for test pattern generation. In: Canadian Conference on Electrical and Computer Engineering. pp. 001397–001402. IEEE (2008)
13. Fraser, G., Arcuri, A.: Evolutionary generation of whole test suites. In: International Conference On Quality Software (QSIC). pp. 31–40. IEEE (2011)
14. Fraser, G., Arcuri, A.: A large scale evaluation of automated unit test generation using evosuite. Transactions on Software Engineering and Methodology (TOSEM) **24**(2), 8 (2014)
15. Gulwani, S., Polozov, O., Singh, R., et al.: Program synthesis. Foundations and Trends in Programming Languages **4**(1-2), 1–119 (2017)
16. Gupta, N.K., Rohil, M.K.: Using genetic algorithm for unit testing of object oriented software. In: International Conference on Emerging Trends in Engineering and Technology. pp. 308–313. IEEE (2008)
17. Hara, A., Watanabe, M., Takahama, T.: Cartesian ant programming. In: International Conference on Systems, Man, and Cybernetics. pp. 3161–3166. IEEE (2011)
18. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. Computing Surveys (CSUR) **45**(1), 11 (2012)
19. Kifetew, F., Devroey, X., Rueda, U.: Java unit testing tool competition-seventh round. In: International Workshop on Search-Based Software Testing (SBST). pp. 15–20. IEEE/ACM (2019)

20. Korel, B.: Automated software test data generation. Transactions on Software Engineering **16**(8), 870–879 (1990)
21. Kushida, J.i., Hara, A., Takahama, T., Mimura, N.: Cartesian ant programming introducing symbiotic relationship between ants and aphids. In: International Workshop on Computational Intelligence and Applications (IWCIA). pp. 115–120. IEEE (2017)
22. Li, K., Zhang, Z., Liu, W.: Automatic test data generation based on ant colony optimization. In: International Conference on Natural Computation. vol. 6, pp. 216–220. IEEE (2009)
23. Mao, C., Xiao, L., Yu, X., Chen, J.: Adapting ant colony optimization to generate test data for software structural testing. Swarm and Evolutionary Computation **20**, 23–36 (2015)
24. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: International Conference on Software Engineering (ICSE). pp. 75–84. IEEE (2007)
25. Rojas, S.A., Bentley, P.J.: A grid-based ant colony system for automatic program synthesis. In: Late Breaking Papers at the Genetic and Evolutionary Computation Conference. Citeseer (2004)
26. Roux, O., Fonlupt, C.: Ant programming: Or how to use ants for automatic programming. In: Proceedings of ANTS. vol. 2000, pp. 121–129. Springer (2000)
27. Sharifipour, H., Shakeri, M., Haghighi, H.: Structural test data generation using a memetic ant colony optimization based on evolution strategies. Swarm and Evolutionary Computation **40**, 76–91 (2018)
28. Socha, K., Dorigo, M.: Ant colony optimization for continuous domains. European Journal of Operational Research **185**(3), 1155–1173 (2008)
29. Srivastava, P.R., Baby, K.: Automated software testing using metahurestic technique based on an ant colony optimization. In: International Symposium on Electronic System Design. pp. 235–240. IEEE (2010)
30. Toffola, L.D., Pradel, M., Gross, T.R.: Synthesizing programs that expose performance bottlenecks. In: International Symposium on Code Generation and Optimization (CGO). pp. 314–326. ACM (2018)
31. Vats, P., Mandot, M., Gosain, A.: A comparative analysis of ant colony optimization for its applications into software testing. In: Innovative Applications of Computational Intelligence on Power, Energy and Controls with their impact on Humanity (CIPECH). pp. 476–481. IEEE (2014)
32. Wappler, S., Wegener, J.: Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In: Annual Conference on Genetic and Evolutionary Computation. pp. 1925–1932 (2006)