Automatic Mutation Test Input Data Generation via Ant Colony

K. Ayari, S. Bouktif and G. Antoniol
Département de Génie Informatique, École Polytechnique de Montréal
C.P. 6079, succ. Centre-ville Montréal (Québec) H3C 3A7 Canada
kamel.ayari@polymtl.ca,
salah.bouktif@polymtl.ca, antoniol@ieee.org

ABSTRACT

Fault-based testing is often advocated to overcome limitations of other testing approaches; however it is also recognized as being expensive. On the other hand, evolutionary algorithms have been proved suitable for reducing the cost of data generation in the context of coverage based testing. In this paper, we propose a new evolutionary approach based on ant colony optimization for automatic test input data generation in the context of mutation testing to reduce the cost of such a test strategy. In our approach the ant colony optimization algorithm is enhanced by a probability density estimation technique. We compare our proposal with other evolutionary algorithms, e.g., Genetic Algorithm. Our preliminary results on JAVA testbeds show that our approach performed significantly better than other alternatives.

Categories and Subject Descriptors

D [**Software**]: Miscellaneous; D.2.5 [**Software Engineering**]: Testing and Debugging —*Testing tools (e.g., data generators, coverage testing)*

General Terms

Algorithms, Experimentation, Languages

Keywords

Test input data generation, Search based testing, Mutation testing, Ant colony optimization.

1. INTRODUCTION

In many software organizations software testing accounts for more than 40-50% of the total development costs [23]. Also, testing and test case generation, are among the most manual labor intensive and technically difficult activities in any software project, so techniques reducing the need for manual intervention will likely affect project costs.

Indeed, thorough testing is often unfeasible because of the potentially infinite execution space or high cost with respect to tight budget limitations. Other techniques such as code inspection are known to be more effective, but even more costly than testing. Unfortunately, defects slipped into deployed software may crash safety or mission critical applications with catastrophic consequences.

Fault-based testing techniques such as mutation analysis and mutation testing are often advocated to overcome limitations of other testing approaches. Mutation analysis identifies techniques to mutate, i.e., to modify, software artifacts, while mutation testing tests adequacy criteria based on mutation analysis.

Mutation testing, originally proposed by DeMillo [6] in 1978, has been studied by numerous researchers. Walsh [30] found empirically that mutation testing is more powerful than statement and branch coverage. Frankl et al. [7] and Offutt et al. [21] affirm that mutation testing is more successful in finding faults than data flow based testing. Andrews et al. show in a recent empirical study reported in [22] that mutation analysis is potentially useful to assess and compare test suites and criteria in terms of their cost-effectiveness. They suggest that mutation analysis can be used to compare and assess new testing techniques. They also recommend the adoption of mutation analysis in more practical situations, for example, when a software development organization needs to empirically determine what levels of coverage are required to attain acceptable detection rates.

In mutation testing the typical testing situation is somehow reversed. It is assumed that a test suite is available together with mutated copies of the original program. A good test case is one that *kills* one or more mutants, for which mutant outputs are different from those of the original program. In this framework, a set of test cases is considered adequate if it distinguishes the original program from all its mutants.

In a real world situation, simple faults are injected into the original program to obtain faulty versions of the program, the mutants. Then test input data are produced to attain the highest possible mutation score,i.e., to kill the highest number of mutants. A set of test cases is more adequate than another if it kills a larger number of mutants. On the other hand, a test suite is preferred over others if it contains fewer test cases and is closer to the adequacy criterion, i.e., has the highest mutation score. Intuitively, mutation testing promotes high quality test suites and has high potential for automation.

This paper proposes the use of metaheuristic approaches, more precisely, ant colonies coupled with mutation analysis to generate high quality test input data suites. Indeed, mutation testing did not attain a wide acceptance level mainly due to technical issues of mutation analysis and test input data generation which has been considered difficult and resource-intensive.

Several techniques have been developed in the past to try to make mutation testing and analysis more cost-effective. In general, these

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'07, July 7–11, 2007, London, England, United Kingdom.
Copyright 2007 ACM 978-1-59593-697-4/07/0007 ...$5.00.

techniques follow one of three strategies: do fewer, do smarter or do faster [22]. The "do fewer" strategy looks for ways of generating and running fewer mutants without losing efficiency; among them it is worth mentioning selective mutation [17] and mutant sampling [5, 13]. "do smarter" approaches look for ways to distribute the expensive computational phases over several machines or to avoid complete execution. For example, weak mutation [8] is a strategy belonging to these latter "do smarter" approaches. The "do faster" based techniques look for ways of generating and running each mutant as quickly as possible; among them are schema-based mutation [22] and separate compilation [29]. Automatically determining which mutants are equivalent [1] is also an important way to reduce the manual labors and promote the acceptance of mutation testing.

This paper focuses on automatic test input data generation attempting to alleviate deficiencies of previous approaches. The first general and implemented attempt to apply mutation analysis to generate adequate test input data for mutation testing was proposed by Offutt in his Ph.D. dissertation [18]. The technique is referred to as Constraint-Based Test data generation technique (CBT) and is based on the observation that a test case is able to kill a mutant if it satisfies three conditions. The first condition, called the reachability condition, states that the mutated statement must be reached. The second, the necessary condition, requires that the execution state of the mutant program differs from that of the original program after some execution of the mutated statement. The third condition, called sufficiency condition, requires that the state difference be propagated to cause incorrect output. The constraint based satisfaction technique suffers from several drawbacks, partly due to weakness of the underlying unsophisticated search procedure. To overcome some CBT problems, another technique called Dynamic Domain Reduction (DDR) was successively developed by Offutt [19]. The basis is the same as CBT although it uses a more sophisticated back-tracking search procedure to help bisection domain-splitting.

While CBT and DDR are based on constraints resolution and input domain splitting, this paper advocates the use of an evolutionary approach to generate data that kill mutants in the context of mutation testing. In our technique, test input data generation is mapped into a minimization problem guided by a cost function, a fitness function inspired by Bottaci [3] proposal. The Bottaci fitness function is defined in a way that a test case is able to kill a mutant if it satisfies the same three conditions used by Offutt in CBT, namely, the reachability, the necessary and the sufficiency conditions.

As stated above, we adopted Ant Colony Optimization (ACO) as the metaheuristic algorithm to solve the minimization problem. Two reasons justify our choice. First, evolutionary algorithms have been proven to be suitable approaches for data generation in the context of coverage based testing. Second, ACO leads to implement *do smarter* approaches in a natural way because ACO intrinsically allows a parallel search. In our approach ants have the mission of killing one mutant each time by searching for a test input datum that satisfies the three Offutt conditions. Furthermore, our ACO algorithm is enhanced by a probability density estimation process that automatically guides and refines the search in promising regions.

The main contributions and innovations of this paper are:

- A new evolutionary approach for automatic test input data generation in the context of mutation testing, which naturally reduce the computational cost in such a test strategy.
- Exploitation of a new emergent search technique, ACO, to facilitate input data generation and compare results with Hill Climbing (HC), Genetic Algorithm (GA) and random search (RND) on two programs;
- Incorporation of new ideas based on a probability density estimation process to automatically refine and guide the search in promising search regions;
- A customization of ACO to the problem of generating input test data to kill mutants.

The remainder of this paper is organized as follows: Section 2 will summarize what we consider as related work to our contribution. Section 3 will introduce notions and concepts used in our approach. Section 4 will present our customization of the Ant Colony Optimization to the problem of input data generation in the context of mutation testing. Section 5 will describe our experiment and results. In Section 6 we will draw our conclusions and future works.

2. RELATED WORK

In the last two decades due to powerful new computers there has been a renewed interest in techniques to automatically generate test data.

Automation of structural coverage criteria and structural testing have been the most widely investigated subjects. Local search was first used by Miller and Spooner [16] with the goal of generating input data to cover particular paths in a program. This work was later extended by Korel [11]. In brief, to cover a particular path, the program is initially executed with some arbitrary input. If an undesired branch is taken, an objective function derived from the predicate of the desired branch is used to guide the search. The objective function value, referred to as *branch distance*, measures how close the predicate is to being true. The idea of minimizing such an objective function was refined and extended by several researchers to satisfy coverage criteria of certain given procedural-program structures like branches, statements, paths, or conditions.

To overcome the limitations associated with local search, Tracey et al. [26] applied Simulated Annealing (SA) and defined a more sophisticated objective function for relational predicates. GA, likely to be the best known evolutionary algorithm that overcomes the problems of local search, was first used by Xanthakis [33] to generate input data satisfying the all branch predicate criterion. Evolutionary approaches where search algorithms, and in particular GA, are tailored to automate and support testing activities i.e., to generate test input data such as the contributions [31, 9, 27, 32] are often referred to as evolutionary based software testing or simply Evolutionary Testing (ET). A survey of ET and related techniques is beyond the scope of this paper; the interested reader can refer to the detached survey by Phill Mcminn [14].

Most of the research on ET makes use of some form of Control Flow Graph (CFG) as the data structure to be manipulated in order to obtain information guiding test input data generation at the unit level, typically function or method. A recent contribution by Tonella [25] has demonstrated ET applicability to the problem of object-oriented testing, more precisely to unit testing of classes. The applications of ET to black box testing have been studied by Tracey et al. [27] and Jones [10]. Tracey et al. used GA and SA to test the specification conformity of a program written in Pascal. Jones used search based techniques to generate data from **Z** specification.

Generate test data for mutation testing has not attracted a lot of attention from the ET community. We are aware of only few works directly pertaining to ET such as Bottaci [3] work. This research defined a fitness function that expresses the cost of satisfying the three conditions proposed by Offutt in his seminal work [18].

An other work was achieved by Zhan and Clark [34] in which they generated test data in the context of mutation testing for Matlab/Simulink models.

As [3] is the first and the sole previous contribution to apply ET to mutation testing, similarities can be found to the current work. In particular our objective function was inspired by Bottaci and can be considered an adaptation of his fitness function.

Since it is a recently proposed search based technique, the first work dates back to the 90's, ACO has not been fully investigated in its application to software testing. The work of McMinn and Holcombe presented in [15] is a simple initial experiment using ACO in order to deal with the problem of searching the chaining tree. More recently Huaizhong et al. [12] used an ACO algorithm to generate test input sequences in the context of state based testing.

It is the author's opinion that, there are many new emergent search based techniques that have not been sufficiently investigated by the ET community. Our objective here is to demonstrate the feasibility of applying ET to mutation testing by using ACO. It is not only a new and emergent technique but also intuitively well suited to the intrinsically parallel nature of the mutants killing problem.

To our knowledge, this is the first work using ACO to generate test input data for mutation testing.

3. BACKGROUND NOTIONS

In the following subsections we introduce mutation testing and then summarize key elements such as Bottaci's fitness function and algorithms used as "benchmarks" for the evaluation of the proposed approach.

3.1 Problem Formulation

Let Pg be a program under test and $I = (x_1, x_2, \dots, x_k)$ be the vector of its input variables. Each input variable x_i takes its values in a domain $D_i, i = 1, \dots, k$, thus, the domain of the program Pg, without any further knowledge, is the cross product $D = D_1 \times D_2 \dots \times D_k$. Further assume that R is a set of mutation operators; each mutation operator is a representative of a typical programming error and it produces a single modification in a single program point giving rise to a mutated version of Pg.

By applying mutation operator $r \in R$ to Pg, N mutated copies $M_1, M_2, \dots, M_N$ of Pg are obtained. In other words, $M_i = r_i(P_g)$ with $r_i \in R$, r_i the ith, $i = 1, \dots, N$, a selected mutation operator that mutates Pg by injecting a simple fault at a statement s_m, called the Pg *mutated statement*.

The problem of test data generation in the context of mutation testing consists of finding a set of test input values that maximizes the number of killed mutants. The essential problem is to find assignments of values to input variables $(x_1, x_2, \dots, x_k)$, called *test cases*, such that when the test suite is executed over the set of mutants $M_1, M_2, \dots, M_N$ it kills the highest possible number of mutants.

As already mentioned, each mutant $M_j, j = 1, \dots, N$, is killed if the three conditions promulgated by Offut [18] are satisfied. The first condition (the reachability condition) states that mutated statement s_m in the mutant M_j must be reached. The second condition requires the value of the mutated expression, once executed, in the statement s_m to differ from its value before mutation. In other words, at the mutated statement s_m, the state of the mutant is different from the original program's one. The third condition (the sufficiency condition) requires the mutated value, i.e., the mutated state at s_m, to propagate to the mutant output. In this paper, we will collectively refer to these conditions as *the killing conditions*.

If an input test case t *kills* a mutant M_j this latter is said to be killed or killed by t; otherwise M_j is said to be still alive. Therefore, if T is the set of test cases killing d mutants, the adequacy of T is assessed by its mutation score $MScore(T)$ given by the following formula:

$$MScore(T) = 100\frac{d}{N - eq}$$

where eq is the number of equivalent mutants i.e., mutants that cannot be distinguished from Pg. Incidentally, we note that the equivalent mutants problem is beyond the scope of this paper. In our study we rely on semi-manual detection of equivalent mutants.

Input variables $x_1, x_2, \dots, x_k$ taking values in $D_1 \times D_2 \dots \times D_k$ in the present work are assumed to be either integer or real values. Strings, pointers and more general data structures will be the subjects of future works.

3.2 Bottaci's Fitness Function

Bottaci [3] incorporates Offutt's necessary conditions in a fitness function to tackle mutation testing from an ET perspective.

In brief, the three conditions are mapped into three cost terms as follows. The reachability cost for a given test case, computed as the goal path minus the number of nodes in the longest common prefix of the test case path and goal path. As explained in [3], there may be several feasible "goal paths" and it is not important which one is considered to compute the cost. In the case of identical costs for two distinct test cases, Bottaci proposes adding to the first reachability cost component, a second cost component, namely the cost of satisfying the common failed decision node on the goal path. The calculation scheme of the second cost is given in [3].

Suppose that mutation changes a condition e into e' at statement s_m. The necessity cost is quantified as the cost of satisfying the predicate $e \neq e'$ by the test case under consideration. This cost is calculated using the same scheme as the second part of reachability cost.

3.3 Hill Climbing

Hill Climbing is the simplest and probably best known search based algorithm. Our goal was to simplify the task of comparison with ACO. In order to kill a given mutant, HC starts by choosing a random test case as an initial solution. The quality of the test case is evaluated by the same fitness function used in ACO and GA; details of this function are given in Section 4.5. HC attempts to improve the current test case by moving to better points in a neighborhood of the current solution. This iterative process continues until a termination criterion (e.g., mutant is killed or a stagnation criterion) in not met. The neighborhood of a test case is defined as the set of test case obtained by modifying the values of one or more input variables. Such a modification is accomplished by incrementing or decrementing by a step the value of the input variable.

3.4 Genetic Algorithm

We will compare GA, the most successful metaheuristic search algorithm used in ET, with our proposed ACO. GA starts by creating an initial population of n test cases chosen randomly from the domain D of the program being tested. Each chromosome represents a test case; genes are values of the input variables. In an iterative process, GA tries to improve the population from one generation to another. Test cases in a generation are selected according to their fitness in order to perform reproduction, i.e., crossover and/or mutation. Then, a new generation is constituted by the l fittest test cases of the previous generation and the offspring obtained from crossover and mutation. To keep the population size constant, we keep only the n best test cases in each new generation. The iterative process continues until a stopping criterion is met(e.g., mutant

is killed or stagnation criteria). Like the cases of HC and ACO, the fitness of a test case measures how close it is to killing the mutant in question (see details in Section 4.5).

In our experiment, crossover was chosen to be the uniform crossover: in the offspring test case, the value of an input variable x_i will be the value of x_i in one of the parent test cases chosen randomly. Mutation was performed by a random modification of one input value in the test case. A second alternative of the mutation operator consists of modifying the test case in a similar way as a in HC; by moving in the neighborhood of the test case to be mutated.

4. ANT COLONY OPTIMIZATION BASED APPROACH

4.1 ACO to Generate Test Cases for Mutation Testing

Similar to other metaheuristic techniques, ACO has to be customized to the particular problem under study. We would like to exploit ants' foraging behavior to generate test input data, test cases for short, killing as many mutants as possible. Since each test case is made up by realizations of input parameters $x_1, x_2, \ldots, x_k$, values chosen in parameter domains, the ants task is to *select good* assignments of parameters values.

Our ACO application can be summarized as follows. Ants start searching for a test case that kills a given mutant by initially randomly choosing test cases. For each test case chosen by an ant, the mutant is executed and the quality of the test case is evaluated. Quality is quantified as closeness to satisfy the killing conditions. Then the ant deposits pheromone trail, i.e., marks with pheromone the values forming the test case. The quantity of deposed pheromone is proportional to the test case quality. As in nature, artificial ants tend to follow pheromone trails. This indirect communication between ants progressively promotes parameter assignments, i.e., test cases, closer to satisfying the killing conditions and eventually killing the mutant.

The above conceptual description need to be detailed by defining a representation of the test case generation problem used by ants to construct test case; a quality measure of the test case, a strategy for the pheromone update, and finally a moving rule that decides which direction the ant should deviate, i.e., how the test case parameter assignment has to be modified.

4.2 Test Input Data Case Construction Mechanism

The test case generation problem was modeled as a directed graph $Gr(V, E)$; each vertex in V, the set of vertices, represents an input parameter x_i. Parameter domains are assumed to be finite, numerable and quantized with suitable quantization steps *a priori* known; thus, domain D_i, $i = 1, \ldots, k$ is quantized into a set of values QD_i containing $|QD_i|$ values.

Nodes are considered ordered and circular. They are ordered by the relative position in the program parameter list; thus, x_2 follows x_1 and precedes x_3. They are circular in that the node x_1 follows node x_k. For any given vertex x_i outgoing edges (incoming to x_{i+1}) represent all possible assignments to parameter x_i from the quantized domain; thus, there are as many edges between two *consecutive* nodes as there are values in the quantized domain D_i.

Figure 1 reports an example of such a graph traversed by ants to construct test cases.

At each iteration, all ants start their trails from the vertex representing the parameter x_1, complete one tour visiting all vertices, and then return back to x_1. When an ant on vertex x_i moves to

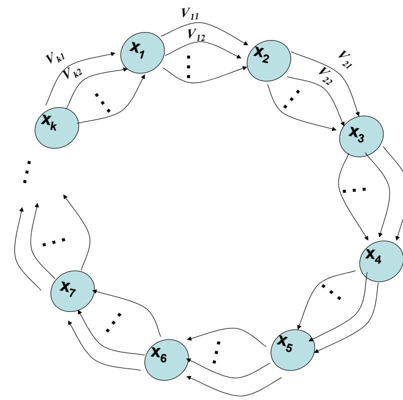

Figure 1: Graph for the solution construction mechanism

the next node x_{i+1}, it chooses an edge (i, j) representing the jth value, $v_{ij} \in QD_i$.

At the beginning, an arbitrary order is used by ants to move from one vertex to the following one. In the following iteration, the choice of the edge to be traversed depends on the amount of pheromone accumulated on that edge. The higher the amount, the higher will be the probability of choosing that edge. When all ants complete one tour, each one deposits a pheromone amount on the edges of the traversed path. Each chosen path represents one candidate test case generated in the current tour. The process is iterated, and at each tour more and more adequate test cases are constructed until a stopping criterion is met.

As described above, the solution construction mechanism assumes that the used graph (see Figure 1) is *static*, built on quantized parameter domains QD_i; all possible values of the input parameters are pre-determined, listed, and used to build the static graph. Thus, test cases, the candidate solutions, are derived from simple combinations of edges. This approach is useful in many real-word situations where parameters take their values in finite and countable sets or in sets that once quantized still represent with sufficient accuracy the problem domain. Examples can include user interface testing or sensor data processing with input parameters acquired from analog-to-digital converters (e.g., audio or video encoding, industrial process control, etc). Clearly, even if some input fields are Boolean, as the number of input parameter increases explicitly constructing all the possible combination may be unfeasible; indeed, it may turn out that there are too many combinations to cover all possible scenarios. In such a situation, the mechanism of solution construction based on static graph is applied in a straightforward manner.

However, other categories of programs require choosing input values in theoretically continuous or infinite domains. Despite the fact that the mathematical real number implementation in computer terms is discrete and finite, it is unfeasible to build a static representation out of them. Furthermore, it may often be the case that quantization is not acceptable; indeed, it is not very difficult to write a program where quantization will never produce input test data killing certain mutants.

Overall, there is the need to derive approaches to deal with continuous non quantizable parameters. Here it is worth noticing a major limitation of most techniques for test data generation such as those using genetic algorithm, with, for example, binary representation (as the most flexible representation). Mutation, flipping bits, or crossover combining chromosomes operate on a quantized representation with quantized steps.

Other techniques tackle the problem of exploring continuous ranges of values by converting them into finite sets. This is achieved, in the case of the genetic algorithm, by the mutation operator, for which a kind of static steps are specified to try to better cover ranges of real parameters [14]. This is not always convenient, especially when the ranges of possible values are wide. The result is that only a limited region of the input space is being explored [28].

4.3 Extended Solution Construction Mechanism

To deal with the problem of better exploring search spaces for continuous parameters, we need to extend our mechanism of solution construction, described above, by adding some dynamic component. The static model can be thought of as a special ant colony where ants never deviate from a pre-determined trail, but rather only choose among a finite set of possible steps. In the real world, it is true that ants follow a trail but it is also true that each ant follows its trail which locally may slightly deviate from those of others. What is required is a mechanism to make ants slightly deviate from a given path and thus dynamically build the graph.

An intuitive way to do this is to model the infinite search space with a *dynamic* graph, in which edges (i.e., values) appear or disappear according to their probability to produce "good" or "bad" test cases. Therefore, instead of choosing a value v_{ij} (edge on the graph and value of QD_i) based on pheromone information, an ant generates a new value for each input parameter by sampling a continuous Probability Density Function (PDF). The details of the PDF sampling are presented in section 4.4.

In this new approach, the solution construction mechanism builds an initial test suite by choosing a set of random values from its domain D_i for each input parameter x_i. Then, ants generate a population P of n test cases by randomly exploring this initial search space.

Here the term "population" is not used in the context of GA but in the context of statistical learning as a population of individuals, the most promising test cases, out of which we need to learn the PDF. Thus, for each input parameter x_i, a PDF is "dynamically" learned from a population P. The size n of the population is a parameter of our algorithm. Then, if m ants are used in our algorithm, m test cases are generated and added to the population P, at each iteration, by sampling PDFs. To keep the population size constant, we eliminate the m worst test cases from the population. An iterative process of learning, sampling PDFs, and updating the population will continue until the subject mutant is killed.

To be consistent with the ACO basic features, during each iteration we could dynamically update a graph by adding and deleting some edges associated with the newly generated and eliminated input values. However, since all the pheromone information is surrogated by the PDF that is learned only from the test cases in the population, we no longer need the graph structure to modify the pheromone information.

In summary, in this new formulation, the PDF sampling and the population management will together ensure the transition of the ants to build test cases and the pheromone update to guide the search to a promising region in the infinite space.

4.4 Transition Function and Pheromone Update Strategy

The ant's task is to assign a value to the input parameter x_i by sampling from a PDF using the population P. Two phases are needed. First, the PDF of x_i is estimated and then a new value is generated from the estimated distribution. For the first task ants use a Gaussian kernel PDF which was used by Bosman and Thierens in their iterative density estimation framework [24] and by Blum and Socha [2] to train neural network using ant colony. The form of the PDF G that an ant uses for the input parameter x_i is given by the following equation:

$$G(x) = \sum_{j=1}^{n} w_j g_j(x) = \sum_{j=1}^{n} w_j \left(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu_j)^2}{2\sigma_j^2}}\right), x \in \Re. \quad (1)$$

where x is a generic input parameter.

The PDF is a weighted sum of n Gaussian functions g_j,$j = 1, \ldots, n$, for which μ_j and σ_j are respectively the means and the standard deviation. The weight w_j of g_j is calculated as follows: The test cases are sorted in descending order of quality, i.e., their closeness to kill the targeted mutant, with the best test case as 1. Then, the weight w_j is evaluated using a Gaussian function given by the following equation:

$$w_j = \frac{1}{qn\sqrt{2\pi}} e^{-\frac{(j-1)^2}{2(qn)^2}}, \quad (2)$$

The Gaussian has a mean of 1.0 and a standard deviation of qn where q is a parameter of the algorithm. The reason behind the choice of a Gaussian form for w_j is its ability to flexibly model the variation of the pheromone intensity over the test cases in the population. In fact, q, a preference parameter, modifies the probability that top ranked test cases will be selected as a base for the PDF sampling. Large values of qn make best-ranked test cases strongly preferred.

Sampling a PDF composed by n distinct Gaussian PDF as defined in equation 1 is not a trivial task. To alleviate computational costs in [2] the authors propose a two-step process. The first step consists of choosing one of the Gaussian functions composing the kernel Gaussian. The probability of choosing the lth component g_l is computed by the following formula:

$$p_l = \frac{w_l}{\sum_{j=1}^{n} w_j}.$$

Then a casino wheel based selection is used. If the ant selects g_l, in its k steps of solution construction, it will use the Gaussian function associated with the test case of rank l. The second step consists of sampling the chosen Gaussian g_l. This is accomplished by using a random number generator based on the Box-Muller method [4], which generates values from a parameterized Gaussian distribution. For such a sampling, we need to specify the mean μ_l and the standard deviation σ_l. Since the lth test case has been selected, the mean chosen is value (i.e., v_l) of the input parameter being sampled; σ_l is calculated as the average distance between the chosen mean and all the values of the input parameter in question:

$$\sigma_l = \rho \sum_{j=1}^{n} \frac{\sqrt{(v_j - v_l)^2}}{n-1},$$

where v_j is the value of the input parameter x in the test case of rank j, and ρ is a parameter of the algorithm that plays the same role as the pheromone evaporation rate.

4.5 Measure of Test Case Adequacy

Our measure of a test case adequacy is inspired by the fitness function of Bottaci [3]. However, up to now we only implemented the reachability component of this adequacy measure. The necessity and sufficiency components will be subjects of a further work. Given a mutant, the fitness of a test case t is defined as follows:

$$f(t) = \begin{cases} 0 & if \;\; Mutant \;\; is \;\; killed, \\ 1 - \frac{1}{2+ReachCost(t)} & otherwise. \end{cases} \quad (3)$$

where $ReachCost(t)$ is the cost of reaching the mutated statement, computed as:

$$ReachCost(t) = NodDistance(t) + Satis(t, e)$$

$NodDistance(t)$ is the number of not traversed decision nodes controlling the mutated statement. $Satis(t, e)$ is the normalized cost of satisfying the topmost failed decision node (i.e., an expression e) on which the mutated statement depends.

Depending on the expression of the predicate associated with the topmost failed decision node, the numerical value of the $Satis(t, e)$ is determined according to the following schema:

	Expression: e	***Satis(t,e)***
a, b are	$a = b$; $a \geq b$; $a \leq b$	$\frac{abs(a-b)}{C}$ *
numerical	$a < b$; $a > b$	$\frac{abs(a-b)+\alpha}{C}$ *
A,B	A;$\neg A$;$A = B$	α
are	$A \wedge B$	$Satis(t, A) + Satis(t, B)$
boolean	$A \vee B$	$\min(Satis(t, A), Satis(t, B))$

* C and α are respectively large and small positive constants.

Table 1: Normalized cost for predicate satisfaction.

5. EXPERIMENTAL RESULTS

In this section, we report results from a preliminary experimental study carried out to evaluate the performance of our approach for automatic test input data generation using ACO. The ACO based approach is compared to three other searching strategies: random search (RND), HC and GA. In the next subsections, we briefly describe two Java programs used as a testbed, the fixed hypotheses and the main experimental steps (see Subsection 5.1). In Subsection 5.2, we detail the algorithmic settings, and finally, in Subsection 5.3 we present results and their interpretation.

5.1 Experimental Design

Two programs serve as testbeds in our experiment. The first one is a triangle classification program (Triangle); triangle classification is a well-known problem used as a benchmark in many testing works. This program takes three real inputs representing the lengths of triangle sides and decides whether the triangle is irregular, scalene, isosceles or equilateral. The second program, NextDate, takes a date as three integers, validates it and determines the date of the next day. The two programs are written in JAVA. They count respectively 55 and 72 lines of code and can be downloaded from https://web.soccerlab.polymtl.ca/repos/soccer-lab/testing-resources/mutation-testing/

Mutants were generated by using the mutation system for Java developed by Ma, Offutt and Kwon [20] available at http://ise.gmu.edu/~ofut/mujava/. Since Triangle and NextDate do not exhibit object oriented features, mutation was performed via $\mu Java$ *traditional* operators; 94 and 104 mutants were created for Triangle and NextDate program respectively.

To obtain evidence of the superiority of the ACO based approach we formulated the following hypotheses:

- Null hypothesis $H0_1$: There is no significant difference between the number of killed mutants within ACO approach and that within each alternative approach, i.e, RND, HC and GA.
- Alternative hypothesis, H1: Our ACO based approach kills a number of mutants significantly larger than RND, HC and GA based algorithms.
- Null hypothesis $H0_2$: There is no significant difference between the cost needed by the ACO approach and that needed by each alternative approach, i.e, RND, HC and GA.
- Alternative hypothesis, H2: Our ACO based approach is significantly more cost-effective than RND, HC and GA based algorithms.

For $H0_1$, the base of comparison is the number of killed mutants (expressed by the mutation score $MScore$) and for $H0_2$we choose the numbers of test case evaluation (referred as $NbrEvaluation$) needed to achieve the same number of killed mutants.

For each hypothesis, the steps of the experiment consist of running 10 times RND, HC, GA and ACO in order to kill non-equivalent mutants derived for each testbed (72 mutants from Triangle and 93 from NextDate). For each algorithm, the mutation score $MScore$ is recorded as well as the needed number of test case evaluations $NbrEvaluation$. In addition, for each mutant killing tentative, the performances of RND, HC, GA and ACO are recorded in terms of the number of needed test case evaluations.

Obviously, in order to kill one particular mutant it is not practical to keep running an algorithm indefinitely. Hence, we fixed a limit on the maximum number of test case evaluations performed without killing a mutant. This threshold, called $MaxNbrEvaluation$, was determined empirically and used by the four algorithms (i.e., ACO, RND, HC and GA).

5.2 Algorithmic Settings

The sole common parameter between RND, HC, GA and ACO is the termination criterion $MaxNbrEvaluation$. Based on several runs, it was observed that if a mutant is not killed with fewer than 500 test case evaluations, it would not be killable with more evaluations. Thus $MaxNbrEvaluation$ was set to 500. Other algorithmic settings pertain only to GA and ACO. For GA the elitist strategy was used; in each iteration, the entire population was replaced, except for the 5% fittest individuals (i.e., test cases). The number of test cases in a generation was 50. The values of p_c (crossover probability) and p_m (mutation probability) were set to 0.70 and 0.06 respectively. Our ACO based approach requires four parameters: the population size was set to 130 to allow for a more accurate estimation of the PDFs. The number of used ants was $m = 3$ as suggested in [2], the pheromone evaporation rate ρ was set to 85%, and the preference parameter of search regions was set to 0.028.

5.3 Experimental Results and Interpretation

Equivalent mutants are eliminated semi-manually in two steps: first we automatically identify the set of non killable mutants by all the algorithms and then we manually investigate the semantic of these mutants.

According to the hypotheses being tested, the results of executing RND, HC, GA and ACO are organized in two groups. The first one contains for each algorithm the maximum score of mutation obtained in each run. The second group tracks for each algorithm the evolution of the score according to the number of performed test case evaluations. These two groups are recorded for Triangle and NextDate and summarized in the following figures. As mentioned above, for each testbed each algorithm was executed over 10 times. Figure 2 summarizes for each algorithm the first group of results in the form of box-plots presenting the distributions of the mutation scores ($MScore$) obtained for Triangle and NextDate. These box-plots show that the mutation score attained by ACO is significantly larger than those obtained by RND, HC and GA. When the performance of the algorithms is expressed in term of realized mutation

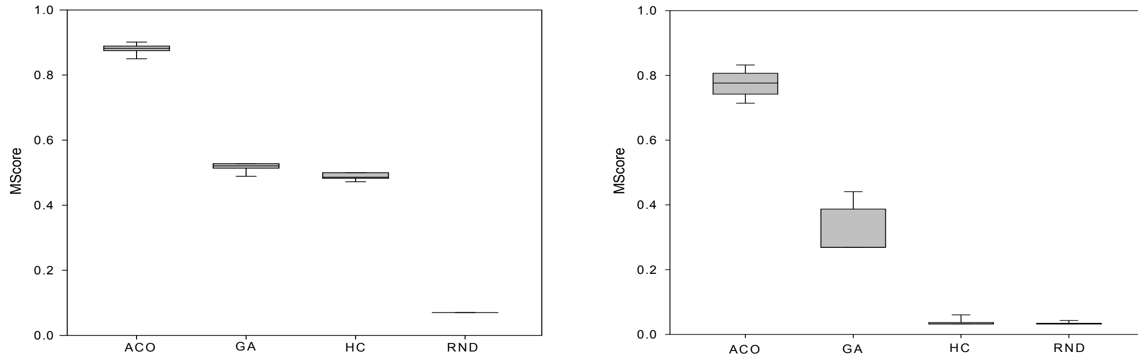

Figure 2: Achieved score of mutation by the different compared algorithms: ACO Vs. RND, HC, GA.

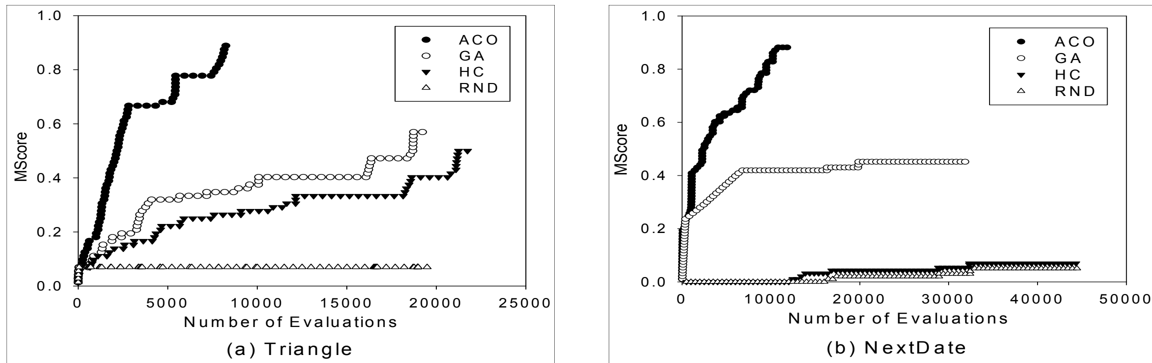

Figure 3: Mutation score evolution according to the number of test case evaluations : ACO Vs. RND, HC, GA.

score, the null hypothesis $H0_1$(see Hypothesis H1) is rejected with a strong evidence (*p-value* < 0.01 in a pairwise comparison (ACO vs. GA, HC, RND), for both Triangle and NextDate).

Figure 3 summarizes the second group of results in the form of a cumulative average plot tracking the evolution of cumulative mutation score ($MScore$) as a function of the number of test case evaluations. In this plot, the stagnation parameter $MaxNbrEvaluation$, i.e, the termination criterion for one mutant killing, was set to 500. This means that the score does not increase if an algorithm fails to kill a mutant after 500 test case evaluations. Figure 3 (a) and (b) show that ACO clearly outperforms all the other algorithms in terms of attained mutation score as well as in terms of needed number of test case evaluations. In particular for the Triangle case, when ACO attained its maximum mutation score 89%, GA, the best alternative approach, was not able to kill more than, 35%. Table 2 shows means and standard deviations of mutation scores by different algorithms at the ACO maximum score (i.e., after 8248 evaluations in the Triangle case and 11823 evaluations for NextDate).

	ACO	GA	HC	RND
Triangle	89%(1.5%)	35%(1.5%)	26%(1.4%)	7%(0%)
NextDate	88%(4%)	42%(7.8%)	0%(0%)	0%(0%)

Table 2: Mutation Scores when ACO attains its maximum score.

Figure 3 shows that with any performed number of test case evaluations, the score realized by ACO is clearly better than the scores attained respectively by GA, HC and RND. To measure the significance of the difference between the cost-effectiveness of ACO and the other algorithms, we assumed that GA, HC and RND could attain the same score as ACO after a large number of test cases evaluations (e.g., 50000) and we applied the Kolmogorov-Smirnov test. This test uses the maximum vertical deviation (the statistic D) between each two compared curves (ACO-GA, ACO-HC, ACO-RND). In all the cases the maximum deviation occurred when ACO attained its maximum score, so the statistic D ranged between 46% and 80%. Consequently, the *p-value* is less than 0.01 and the null hypothesis $H0_2$(see alternative Hypothesis H2) is rejected with a strong evidence.

6. CONCLUSION

We have proposed an evolutionary approach based on ACO to reduce the cost of test data generation in the context of mutation testing. Inspired by Bottaci, we defined and implemented a fitness function that measures how close a test case is to kill a mutant. Our ACO based approach is enhanced by a probability density estimation technique in order to better guide the search for continuous input parameters. Our preliminary results on two testbeds show that our enhanced ACO approach performed significantly better than GA, HC, RND in term of attained mutation score as well as computational cost.

Issues of future research include the extension of our test case adequacy function to evaluate the costs of the necessary and the sufficiency conditions to kill a mutant. In order to better evaluate our approach, we will compare it to an enhanced GA that will

involve the probability density estimation technique. Further improvements of our approach will also be achieved by considering other types of inputs and predicate expressions using strings, arrays and Booleans.

Finally, further work will focus on applying our approach to programs developed with other programming languages (e.g., C and C++), and to other adequacy criteria.

7. ACKNOWLEDGMENTS

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (Research Chair in Software Evolution #950-202658).

8. REFERENCES

[1] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation Conference*, pages 1338–1349, 2004.

[2] C. Blum and K. Socha. Training feed-forward neural networks with ant colony optimization: An application to pattern classification. In *5th International Conference on Hybrid Intelligent Systems*, pages 233–238, 2005.

[3] L. Bottaci. A genetic algorithm fitness function for mutation testing. In *proceedings of SEMINAL: Software Engineering using Metaheuristic INovative Algortithms, Workshop 8, ICSE 2001, 23rd International Conference on Software Engineering*, pages 3–7, 2001.

[4] G. Box and M. Muller. A note on the generation of random normal deviates. *Annals. Math. Stat.*, 29:610–611, 1958.

[5] T. A. Budd. *Mutation analysis of program test data*. PhD thesis, Yale University, New Haven , CT , USA, 1980.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[7] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs. mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software*, 38(3):235–??, Sept. 1997.

[8] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[9] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[10] B. Jones, H. Sthamer, X. Yang, and D. E. T. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of the 3rd International Conference on So Quality Management, Seville, Spain*, pages 435–444, 1995.

[11] B. Korel. Dynamic method of software test data generation. *Softw. Test, Verif. Reliab*, 2(4):203–213, 1992.

[12] H. Li and C. P. Lam. An ant colony optimization approach to test sequence generation for statebased software testing. In *Evolvable Hardware*, pages 255–264. IEEE Computer Society, 2005.

[13] P. May, K. Mander, and J. Timmis. Software vaccination: An artificial immune system approach to mutation testing. In *Artificial Immune Systems, Second International Conference, ICARIS 2003, Edinburgh, UK, September 1-3, 2003, Proceedings*, volume 2787, pages 81–92, 2003.

[14] P. McMinn. Search-based software test data generation: a survey. *Softw. Test, Verif. Reliab*, 14(2):105–156, 2004.

[15] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Genetic and Evolutionary Computation – GECCO-2003*, pages 2488–2498, Berlin, 2003. Springer-Verlag.

[16] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, Sept. 1976.

[17] E. S. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Softw. Test, Verif. Reliab*, 9(4):205–232, 1999.

[18] J. Offutt. *Automatic test data generation*. Ph.d. dissertation, Georgia Institute of Technology, Atlanta , GA , USA, 1988.

[19] J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Softw, Pract. Exper*, 29(2):167–193, 1999.

[20] J. Offutt, Y. S. Ma, and Y. R. Kwon. An experimental mutation system for java. In *Proceedings/ACM SIGSOFT SEN*, pages 1–4, 2004.

[21] J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software—Practice and Experience*, 26(2):165–176, Feb. 1996.

[22] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA*, pages 45–55, 2000.

[23] R. S. Pressman. *Software Engineering: A Practitioner's Approach 3rd edition*. McGraw-Hill, 1992.

[24] D. Thierens and P. A. N. Bosman. Multi-Objective Mixture-based Iterated Density Estimation Evolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 663–670, 2001.

[25] P. Tonella. Evolutionary testing of classes. In *ISSTA*, pages 119–128. ACM, 2004.

[26] N. Tracey, J. A. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *ISSTA*, pages 73–81, 1998.

[27] N. Tracey, J. A. Clark, K. Mander, and J. A. McDermid. Automated test-data generation for exception conditions. *Softw, Pract. Exper*, 30(1):61–79, 2000.

[28] Z. Tu and Y. Lu. Global optimization of continuous problems using stochastic genetic algorithm. In *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 1230–1236, 2003.

[29] R. Untch, J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *ISSTA*, pages 139–148, 1993.

[30] P. J. Walsh. *A measure of test completeness*. PhD thesis, State University of New York at Binghamton, 1985.

[31] A. Watkins. The automatic generation of test data using genetic algorithms. In *Proceedings of the Fourth Software Quality Conference*, pages 300–309. ACM, 1995.

[32] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14):841–854, 2001.

[33] S. Xanthakis, C. Ellis, C. Skourlas, A. L. Gall, S. Katsikas, and K. Karapoulios. Application des algorithmes genetiques au test des logiciels. In *5th Int. Conference on Software Engineering and its Applications*, pages 625–636, 1992.

[34] Y. Zhan and J. A. Clark. Search-based mutation testing for simulink models. In *Genetic and Evolutionary Computation Conference, Proceedings*, pages 1061–1068, 2005.