Testing Software Using Swarm Intelligence: A Bee Colony Optimization Approach

Omar El Ariss
Computer & Mathematical Sciences
The Pennsylvania State University
Harrisburg, PA, USA
1 (717) 948-6669
oelariss@psu.edu

Steve Bou ghosn
Computer Science &Information Systems
Westfield State University
Westfield, MA
1(413) 572- 5294
sboughosn@westfield.ma.edu

Weifeng Xu
Department of Computer Science
Bowie State University
Bowie, MD
1 (301) 860-3965
wxu@bowiestate.edu

ABSTRACT

Software testing is a critical activity in increasing our confidence of a system under test and improving its quality. The key idea for testing a software application is to minimize the number of faults found in the system. Software verification through testing is a crucial step in the application's development life cycle. This process can be regarded as expensive and laborious, and its automation is valuable. We propose a multi-objective search based test generation technique that is based on both functional and structural testing. Our Search Based Software Testing (SBST) technique is based on a bee colony optimization algorithm that integrates adaptive random testing from the functional side and condition/decision and multiple condition coverage from the structural side. The constructive approach that the bee colony algorithm uses for solution generation allows our SBST to address the limitations of previous approaches relying on fully random initial solutions and single objective evaluation. We perform extensive experimental testing to justify the effectiveness of our approach.

Categories and Subject Descriptors

D.2.4 Software/Program Verification; D.2.5 Testing and Debugging; F.1.2 Modes of Computation; I.2.8 Problem Solving, Control Methods, and Search.

General Terms

Algorithms, Reliability, Verification.

Keywords

Swarm Intelligence; unit testing; automated test generation; branch coverage; search based testing

1. INTRODUCTION

It is estimated that software testing corresponds to 30% to 50% of a project's budget [4]. Generating test cases is a very challenging problem because it is unfeasible to find a suite of test cases that fully evaluates a program, as the input domain of most programs is nearly infinite. Thus, human testers spend a lot of time striving to find a high quality set of test cases that will allow them to detect a large percentage of faults in a software system. Automated testing research attempts to find ways to make this process more efficient by automatically generating the test cases.

A lot of research has been done in the area of search based methods of automated test case generation. Some of the SBST research has focused on local search methods such as hill climbing, simulated annealing and tabu search. In [2] a hill climbing approach was used to generate test cases, while in [24] the authors experimented with the use of simulated annealing. In [7] a tabu search metaheuristic algorithm was used to generate tests for structured software. Another local search technique used for test case generation is the alternating variable method[13][14].The main problem with the use of local search techniques is that because they only consider the neighborhood of a high quality solution they often get stuck in local optima. Advanced local search techniques like simulated annealing and tabu search address this shortcoming by either restarting the search with random values or temporarily accepting low quality solutions.

Due to the limitations of local search techniques, various research efforts have been dedicated to global search methods that are less prone of getting stuck in local optimal. One of the most popular techniques are evolutionary algorithms such as Genetic Algorithm (GA). Various work has been done using GA techniques, for example [12], [20] and [3] are some of the early works. One of the problems with global search methods like

BICT 2015, December 03-05, New York City, United States

DOI 10.4108/eai.3-12-2015.2262529

genetic algorithms when applied to test generation is that they tend to alter the solutions too much. Operators like crossover and mutation drastically change the solution and make it harder to improve in a more focused way that relates to aspects of the test cases. In addition to that, these changes might generate invalid or illegal input.

To overcome this problem many recent approaches have used a hybrid method that combines genetic algorithms with other approaches, for example a local search method. One such work is the augmentation of the GA approach with constraint-based testing [17]. In addition, the authors in [8] use a memetic algorithm while in [21] an approach that combines genetic algorithms and tabu search is applied. Another limitation of genetic algorithms approaches for software testing is having a candidate solution represent a test case and the entire population represent a single test suite. This choice of representation strips the GA approach from one of its main features, which is working on multiple candidate solutions instead of one. In addition, the genetic operators like crossover and mutation tend to not be very suitable when dealing with test cases as the chromosomes. Also the use of randomness when selecting an initial population of solutions isn't appropriate when performing test case generation, it seems more appropriate to base it on a partial set of functional tests [1].

In our research we propose three main contributions. First, our implementation is based on a Bee Colony Optimization (BCO) algorithm, which is a very recent heuristic proposed in [16] and used successfully to solve many different problems [22], [6], [15]. We believe BCO would work very well because of its use of a constructive approach to solution generation. A test suite can be generated adaptively using these techniques as opposed to generating complete random solutions as in GA. The second is encoding a suite of test cases as a candidate solution instead of an independent test case; some recent research seems to be following this approach too [8]. We find this approach to be more natural and have a one to one correspondence to how software testers construct a test suite. We seldom see them directly think of the entire test suite. Instead, they constructively modify the test suite by adding to it more test cases in order to improve the test coverage. We believe that genetic algorithms due to the limitations mentioned above wouldn't be particularly feasible for this approach. Our third contribution lies in the objective function used. Previous work in search based testing focused on a single objective function, which is mainly decision coverage [10, 11]. Our fitness function is multi objective, where the focus is on achieving condition and decision coverage, while multi-condition coverage is considered as a bonus objective.

This paper is divided as follows: in Section 2 we give an introduction to BCO, in section 3 we describe our implementation using BCO for automated test case generation. In section 4 we perform some experiments on our approach using a known set of test case samples. Finally in section 5 we present our observations and conclusions.

2. BEE COLONY OPTIMIZATION (BCO)

Colonies of social insects such as ants and bees have highly organized behavior that enables them to work collectively to solve problems and thus perform much more efficiently than having each member working individually [22]. This interaction and collective behavior of the decentralized agents or members of the colony constitutes swarm intelligence. In a bee colony an organized collaborative effort is used in order to find flowers that are potential food sources and exploit them, allowing bees to harvest nectar from different food sources separated by long distances. There are two groups of bees that are formed as part of this strategy, scouts and workers. The set of bees working as scouts are constantly searching the environment for new potentially promising nectar sources. Any scout that finds a promising source returns to the hive and communicates the information to its peers by performing a special dance called a waggle dance. Other bees in the colony that are initially idle will observe the waggle dances performed by the scout bees and become worker bees on one of the advertised food sources. The amount of worker bees that will join on exploiting a particular food source is directly proportional to the quality of that source. Only a small percentage of the bees in the colony assume the role of scouts at any given time leaving the majority of the workforce to be concentrated on exploiting nectar sources. As long as sources are still deemed profitable, the bees working on them will continue to advertise them thus optimizing the workforce in the colony to focus on the best areas.

When using bee colony optimization (BCO), the search space of all possible solutions is represented by the field the bees are exploring. Each possible partial solution is a point in the field which is basically a potential source of nectar. The quality of the source is determined by an objective function that evaluates how optimal the solution is in solving the problem. There are two stages in BCO, the forward phase and the backward phase. The forward phase represents the bees flying to search for food sources, every source found becomes a new partial solution that is being

analyzed. During the backward phase bees return to the hive and scout bees use the waggle dance to advertise those partial solutions. At that stage, some bees will join the workforce on each of those partial solutions that show promise while other bees will assume the role of scouts and search for new sources.

The effectiveness of the bee algorithm relies on the use of both an intensification and diversification strategies when constructing solutions. While a large number of the bees are performing a local search by exploring existing solutions (intensification), a small number of bees make sure the algorithm doesn't get stuck in a local optimal by always exploring new areas of the search space (diversification). This combination of both a local search and a global search and the fact that solutions are being constructed dynamically rather than randomly generated is one of the most important features of BCO.

3. AUTOMATED TEST GENERATION USING BCO

In this section we describe the Bee Colony Optimization (BCO) approach to unit testing. Our testing strategy is a white-box one, where we rely on the source code to generate the test cases. Therefore the Control Flow Graph (CFG) of the function under test is used to determine the coverage criteria and to guide the testing process. Adaptive random testing, a black-box approach, is also integrated into BCO to diversify the input values.

Our proposed work brings various novelties to the current techniques used by evolutionary search methods. Firstly, we address some of the limitations of known approaches such as random initial solutions [1], fixed test suite size and evolving operators that alter solutions too much [1]. We do that by encoding a population of test suites. The reason we chose to use BCO as opposed to other evolutionary algorithms is that BCO is inherently a constructive algorithm. While other approaches start with fully random solutions, BCO starts with an empty solution and constructs it gradually through an iterative process.

Another feature of our approach is that in contrast to previous works like [18] that constrain the input domain of test variables to make the search space more manageable, and [9] that limit the range of values for bloat control, our approach imposes no limitations in the input domain. A final innovation lies in how we evaluate the quality of the tests. We use a multi-objective fitness function, contrary to the prevalent use of a single objective function in most previous work.

3.1 Solution Encoding

The problem of automated test case generation is to come up with suite of tests that exercise the component under test. In this paper we target standalone functions, therefore the output of our BCO algorithm is a test suite that is suitable for efficiently evaluating the function under test. BCO is a population based algorithm where multiple solutions are constructed at the same time. Each solution stands for a full test suite. An individual test suite is a set of test cases, where each test case is a tuple of values that stands for the input variables of the function under test. We focus in this paper on integer values but the approach can be easily modified to handle other data types.

The constructive nature of the BCO algorithm supports our choice of encoding. Once a test case is added to the test suite, the test case will not change. This simplifies the calculation of the coverage criteria and makes it an incremental one. Other approaches, such as the GA based ones, have to recalculate the coverage criteria for the entire test suite every time the crossover and mutation operators are applied. This is time consuming and has a detrimental effect on the performance of the automated test generation process. This also addresses other limitations of recent evolutionary approaches. Operators that function on the test case level evolve and modify the solution excessively and impact negatively on the effectiveness of the search. Because these approaches encode solutions as a single test case, even a slight mutation of the solution brings too much alteration, the use of a test suite as a solution addresses these problems because operators alter an individual test case within the larger solution rather than all of it.

The BCO algorithm as the search engine and the full test suite solution encoding described above make a straightforward and natural way to automatic software test generation that parallels how testers think about the problem. Our approach works by building a test suite one test case at a time, In other words, we start with an empty test suite and then try to add new test cases, the test that is added in every iteration should be the most suitable and profitable for efficiently testing the System Under Test (SUT) and should be based on the current tests already added to the suite. The technique we use to generate the tests takes into consideration many criteria of coverage and is an integral part of our approach; we discuss the coverage criteria later in this section.

3.2 Fitness Function

When evaluating the quality of a given solution we are trying to determine how thorough is the set of tests composing that solution with respect to evaluating the

SUT. This is where the fitness function plays a role; this function indicates how close a solution is to the optimal test suite. The fitness function is calculated while taking into consideration the following coverage criteria:

Condition/Decision Coverage: Also called branch/condition coverage. The test cases should guarantee that both condition and decision coverage are satisfied:

Branch Coverage: Also known as decision coverage. The test cases should guarantee that each branch in a control flow graph is exercised at least once. In other words, all decisions (whether they are simple or compound) should be evaluated to true and false. For example the entire condition (x>8 && x<50) should be exercised when it is *true* and when it is *false*.

Condition Coverage: The test cases should guarantee that each simple condition in the program is evaluated as true and false at least once. For example the entire condition (x>8 && x<50) should be exercised when it is *true* and when it is *false*. For example to cover the condition (x>8 && x<50) we need to test *x>8* and *x<50* when it is *true* and *false*, *where TF and FT combinations is enough*.

Multiple Condition Coverage: The test cases should guarantee that all true-false combinations of simple conditions in a compound predicate are evaluated at least once. In other words to cover (x>8 && x<50) (which is composed of two simple conditions) we need TT, TF, FT and FF combinations. This coverage criterion isn't a feature tested in most previous works.

We consider the minimum optimal solution to be a test suite that has a complete condition/decision with the least number of test cases. A higher fitness value than the minimal optimal solutions indicates better multi-condition coverage. The value of the fitness function for a test suite that is constructed by a single bee b_i (the ith bee) is determined with the following formula:

$$F_i = (mc + 1) * (cd + \left\lceil \sum_{j=1}^{|cc|} DistCost(b_i, c_j) \right\rceil) \quad (1)$$

Where ***mc*** stands for the number of multiple conditions that were not covered, ***cc*** for the number of simple conditions, and ***cd*** stands for the amount of conditions and decisions that weren't covered. Since the value of the fitness is higher when the coverage is lower this fitness function should be used with a bee algorithm that seeks to minimize the fitness.

The *DistCost* function is used to determine how far the solution is from covering a compound condition (decision) in the program. This function helps in distinguishing test suites that are closer to covering a condition from solutions that are farther away. This function is applied to all compound conditions in the function under test and their total sum is added to the fitness value. The distance cost is evaluated in the following way:

$$DistCost(b_i, c_j) = \begin{cases} 0 \; if \; the \; condition \; has \; been \; met \; by \; the \; bee \; b_i \\ 1 \; if \; the \; condition \; is \; unreachable \; by \; the \; bee \; b_i \\ \frac{dis(c_i)}{1+dis_{max}} \; otherwise \end{cases} \quad (2)$$

Where the value of dis(C_i) depends on the type of the compound predicate, and is computed as described on table 1 similar to the authors' work in[23]. We decided to target the coverage criteria at the source code level instead of the byte code level such as the work done in [5, 19]. Byte code instructions have a simpler decision structure, where compound decision statements are transformed into simple nested decision statements. Our goal is to determine the capability of our algorithm on complex decision structures.

3.3 BCO Algorithm Process

Our bee colony algorithm follows the general format described in [16]. In this approach we start with a total of *n* number of bees and a *k* number of recruiter bees among them, where k < n and n > 1. These two values are determined by tuning parameter that can be adjusted during the testing phase, we normally try to set the number of recruiter bees to 25% of the total number of bees. These tuning parameters are referred hereafter in this work as ***numBees*** and ***numRecruiters***. The experimental testing and parameter tuning process is discussed in details in Section 4.

The algorithm keeps alternating between a forward phase and a backward phase. Every iteration of the process consists of those two phases and the algorithm continues iterating until a satisfaction criterion is fulfilled. Each bee is assigned exclusively to work in one specific solution at a time, many bees could be working on the same solution but no bee can be working in more than one solution at the same time. The solutions as described before each consist of a full test suite. Figure 1 illustrates the general process of the algorithm.

During the forward phase all bees work on their current solutions by generating a new test to be added to the test suite they are constructing. During every forward phase, each bee generates and adds only one test to its solution. The process to generate the tests will be explained later in this section. During the backward phase all bees return to

the hive and among them a number equal to ***numRecruiters*** will be selected and those bees will advertise their solutions to try to convince other bees to follow them. When a bee follows another it basically abandons the solution it was working on and shifts its work power to the solution of the recruiter. The way the recruiters are selected among the bees depends on the quality of the solutions they are working on. The higher the quality of a solution, the higher the probability of becoming a recruiter. The quality of a given solution is determined by using the fitness function described in Section 3.2. Before determining which bees will be recruiters in the backward phase of a given iteration, all the fitness values of all solutions are first normalized.

Table 1. How the dis(C_i) value is computed

Expression *e*	Distance Measure *dis(e)*, *k*=2
x < y	x – y + k
x ≤ y	x – y
x = y	|x – y|
x ≠ y	k
x > y	y - x + k
x ≥ y	y - x
$e_1 \wedge e_2 \wedge \ldots e_n$	$\sum_{i=1}^{n} dis(e_i)$
$e_1 \vee e_2 \vee \ldots e_n$	$dis_{min}\ \forall e_i\ where\ dis(e_i) > 0$

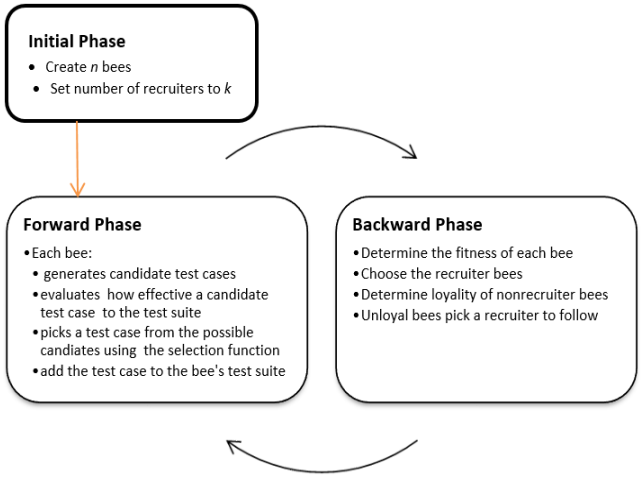

Figure 1. The general process of our BCO algorithm

For each fitness value we compute the normalized fitness using the following formula:

$$NormF_i = 1 - \frac{F_i}{F_{max}} \quad (3)$$

We also compute the fitness probability using the following formula:

$$p_i = \frac{NormF_i}{\sum_{j=1}^{n} NormF_j} \quad (4)$$

In eq. 3 we are normalizing the fitness function of a particular solution *i* over the bee with the maximum fitness value. We are then subtracting this value from one to turn the problem from a minimization to a maximization one, since the fitness function gives lower values to higher quality solutions as described before. In eq.4 we divide the normalized fitness of solution *i* over its sum and selecting based on that. After the normalization and calculating the fitness probability, the selection process continues as follows:

The first recruiter is selected using elitism that is the bee that has the highest quality solution is always chosen first. The remaining recruiters are selected using a roulette wheel selection based on the fitness probabilities of their solutions. For all the non-recruiter bees we decide for each one whether they will be loyal or not. A loyal bee will continue working on its current solution during that iteration, while a non-loyal bee will follow one of the recruiter bees. For non-loyal bees the recruiter it is assigned to is picked randomly from the existing recruiters. To determine if a bee is loyal or not we compute a loyalty probability and compare it with a random value, if the loyalty probability of that bee is higher, then the bee is considered loyal otherwise it is considered non loyal. The loyalty probability of bee b_i is computed using the following formula:

$$l_{b_i}^{u+1} = 2\pi^{-\frac{NormF_{max} - NormF_i}{\sqrt{u}}} \quad (5)$$

Where *u* stands for the number of forward phases the algorithm went through while $NormF_{max}$ stands for the maximum normalized fitness value among all bees. Parameter *u* is introduced to allow the bees to easily change the solution they are currently working on at early stages of the algorithm while making it harder in later stages.

Figure 2 gives an example of the forward phase. Here we have four bees working on the solution. In the first forward phase, each bee selects one test case from the set of candidate test cases (8 possible test cases in this example). The bee selects a test case based on the probability of this test case improving the bee's fitness function. Here we can see that both *bee 1* and *bee 2* select Candidate Test Case (CTC) one as their first test in their respective paths. On the

other hand, the third and fourth bees select *CTC5* and *CTC7* respectively. After the backward phase, which is not shown in the figure, the next forward phase commences. Each bee will add again a single test case to its test suite (path).

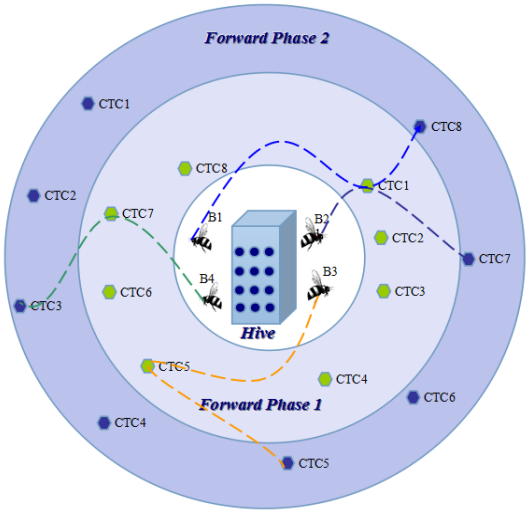

Figure 2. The forward Phase of our BCO implementation

Figure 3 shows an example of the backward phase. After the second forward phase, the four bees return to the hive to evaluate their constructed paths (their partial solutions). The first step is to determine the recruiter bees. In this example we have a single recruiter, which is the fourth bee. The next step is to determine the loyalty of the rest of the bees. The first and second bees decide to be loyal to their paths while the third bee abandons its paths. The third bee in phase 3 will have the same two test cases as the fourth bee but will choose a new test case on its own.

3.4 Test Generation

As mentioned before, during every forward phase each independent bee will generate a new test and incorporate it to its current solution. When generating these new tests we integrate three main features: graph coverage, local search and randomness. We try to generate many different tests using different techniques including having a focus on branch and condition coverage, local search, pure randomness and adaptive techniques. The bee will then pick the best test among them using roulette wheel to be the actual test added in that phase. So basically we generate ***n*** tests cases, we then evaluate the fitness function of the whole test suite and for each test case check how much improvement that particular test case will do to the overall solution. We then use roulette wheel to select the most suitable test case to add to the bee's path.

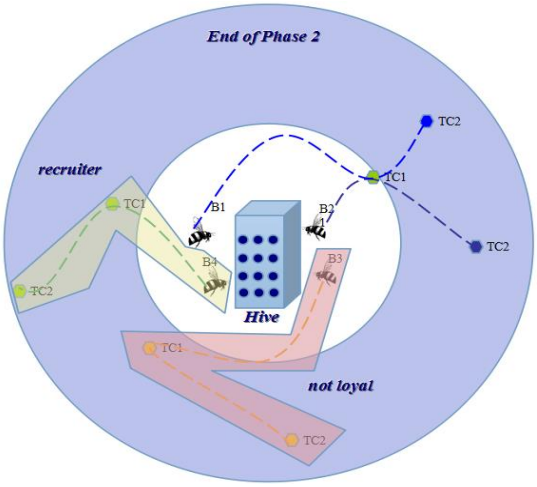

Figure 3. The backward phase of our BCO implementation

For graph coverage, the way this is performed is we first generate one test case that targets a specific simple condition in the control flow graph. The test case generation here makes use of the concept of boundary value analysis. For each simple condition with a relational operator a test case is randomly generated by choosing between three options: on the boundary, above or below the boundary by a certain percentage. For example, for the simple condition x = 9, the input is generated to cause the variable x to be 9, 9+δ and 9-δ where δ is a value within 0 to 30% of the value of x, in this case 9.

Hence we will have one test case for each control flow condition. That is the particular test case will be partially random with the only requirement being that when that test is performed that particular condition in the program flow graph will either be covered or $\pm\,\delta$ covered.

Then we generate a number of test cases equal to the number of input variables that the STU has, these test cases are slight variations of the last test case in the bee's path (test suite). Those test cases are modified based on a probability constant by just adding a small change to the input variable that corresponds to that test case, since each test case is assigned to a specific input parameter. This basically amounts to a local search being performed with those test cases.

Then we also generate one completely random test case and an adaptive random test case [5]. Adaptive random testing usually outperforms ordinary random testing since the input is more evenly distributed. The way this adaptive random test case is generated is the following:

For a given bee we create a random test case that is the farthest test case from all cases used before by that bee. The way we do this is to first generate ten random test cases and for each case compare it and calculate the Euclidean distance with all the test cases used by the bee then choose the test case among the random cases that is farthest away.

The test generation step does not concentrate on one test strategy, whether it is black box or white box. We do not want our approach to mimic boundary value testing or random testing, we want the algorithm to select either of both as required. The bee will choose the test case type that best benefits it during a particular forward phase. Thus, showing adaptability and flexibility.

4. EVALUATION

The evaluation of search based testing approaches is a challenging process since there is no single agreement on the best evaluation strategy. It is usually very difficult to compare one approach to previous approaches since each approach has different constraints such as the range of values used, the component that is under test (whether it is a standalone function or a class), the number of test cases generated or the size of the test suite. In addition, most SBST approaches are randomized algorithms and therefore produce different results on multiple runs. On the other hand, comparison of SBST techniques that are white box strategies with black box testing strategies such as random testing does not really help since each strategy has its own merits.

We opted to evaluate our approach through the use of mutation testing. After all, the ultimate question is whether the generated test suite is adequate enough. We ran our algorithm until a 100% condition/decision coverage or after a fixed number of forward phases. The case study used here is the triangle program, which is a very popular example and the benchmark in software testing literature. The program receives the lengths of the three sides of a triangle as input. The value of each side is an integer value. The output of the function is a string that indicates whether the input does not form a triangle or the type of the triangle whether it is isosceles, equilateral or scalene. The code, which is based on [19], is shown in Figure 4 while Figure 5 shows its corresponding control flow diagram. The optimal test suite size for this problem is 5.

We generated 46 different mutants from the original triangle program. Twenty eight of them are first order mutants, where a single fault is introduced to the original program using one of the mutation operators. The rest of the mutants are high order, where each mutant contained two to three different mutations. An example of the generated mutants can be seen in Figure 5; the highlighted code shows two mutations. We then run our BCO algorithm several times and obtained thirteen different solutions. As described before each solution consists of a complete test suite. Table 2 shows a description for each of the test suites generated including the number of test cases comprising the solution, the range of the input values used, the number of bees, the number of recruiters and the code coverage. We tried to vary the values of the first 4 parameters to determine their effect on the quality of the test suite.

```
String triangleType(int a, int b, int c)
{
        String type;
        if (a > b)
        {
            int t = a; a = b; b = t;
        }
        if (a > c)
        {
            int t = a; a = c; c = t;
        }
        if (b > c)
        {
            int t = b; b = c; c = t;
        }
        if (a + b <= c)
            type = "NOT A TRIANGLE";
        else
        {
            type = "SCALENE";
            if (a == b && b == c)
                type = "EQUILATERAL";
            else if (a == b || b == c)
                type = "ISOSCELES";
         }
        return type;
    }
```

Figure 4. The triangle type program

We then evaluated each test suite solution against all the generated mutants to see if the set of tests can detect all the software defects. Table 3 shows the results of our experiment for all tests and all mutants. The letter *Y* denotes that the given test suite killed the mutant, while an *N* indicates that the mutant wasn't killed. It is interesting to notice that most mutants were killed by all the test suites. Higher order mutants, as expected, were more challenging than first order mutants. Seven of the mutants were not killed, where one of them (Figure 6) is an equivalent mutant. The rest of the live mutants are on the boundary type faults such as *if (a > b)* is mutated into *if (a >= b)*; these types of faults are hard to kill. One final thing to notice is that although all test suites have the same

coverage percentage some of them were unable to kill a mutant while others did. For example the 2nd and 11th test suite where unable to detect the 46th mutant, which contains an on the boundary type fault. The rest of the test suites were capable of killing this mutant.

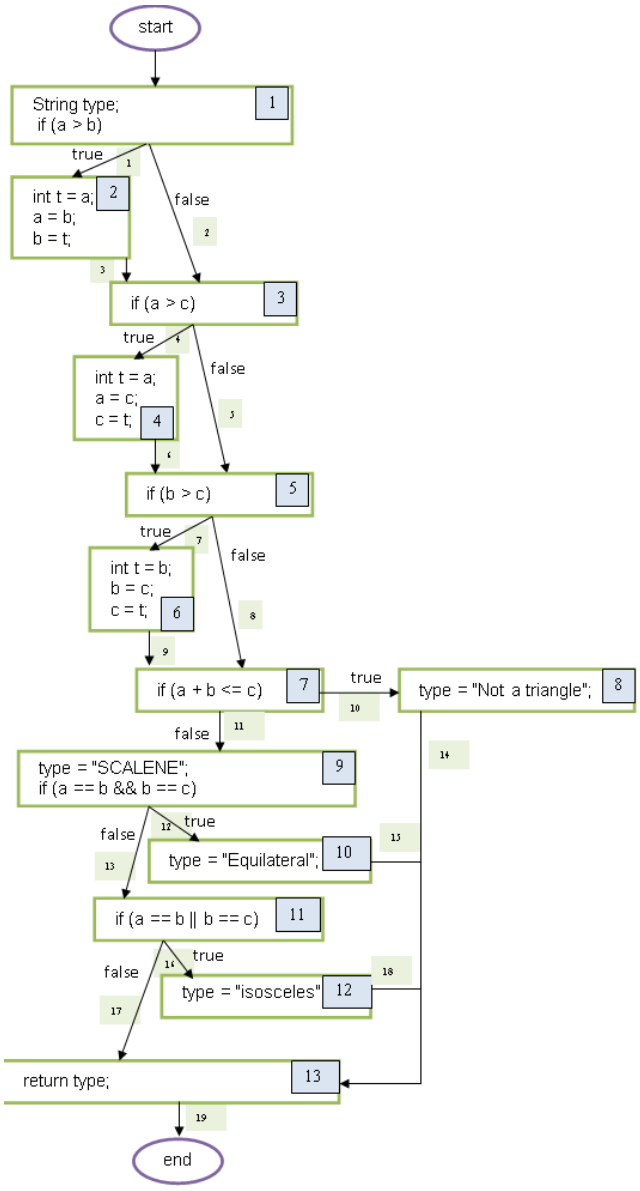

Figure 5. CFG for the triangle program

```
String triangleType(int b, int a, int c)
        {
            String type;
            if (a > b)
            . . .
```

Figure 6. An example of a mutant of the triangle program

5. CONCLUSIONS

In this paper we used a Bee colony optimization technique in order to perform automated test generation. This is one of the first works that applies this kind of swarm intelligence technique to the problem of software testing. With regards to previous approaches, our work improves on them by not limiting the input domain, avoiding random initial solutions and by using a dynamic constructive approach, using test suites as solutions rather than independent tests and by taking into consideration multiple condition coverage.

Table 2. Summary of the test suites used

	# of Test Cases	Range	# of Bees	# of Recruiters	Cond/Dec Cov	Multi Con
Ts1	5	[min, max]	25	5	100%	100%
TS 2	6	[min, max]	25	5	100%	100%
TS 3	7	[min, max]	25	5	100%	100%
TS 4	8	[min, max]	25	5	100%	100%
TS 5	5	[0, 1000]	25	5	100%	100%
TS 6	5	[0, 1000]	25	5	100%	100%
TS 7	9	[0, 1000]	25	5	100%	100%
TS 8	6	[0, 1000]	25	5	100%	100%
TS 9	11	[0, 1000]	25	5	100%	100%
TS 10	8	[0, 1000]	150	37	100%	100%
TS 11	5	[min, max]	150	37	100%	100%
TS 12	6	[min, max]	150	37	100%	100%
TS 13	9	[min, max]	150	37	100%	100%

Table 3. Mutation Results

	First Order Mutants		Higher Order Mutants					
	M 1 to M23	M24-M28	M29-M40	M41	M42	M43, M44	M45	M46
TS1	Y	N	Y	Y	Y	N	Y	Y
TS 2	Y	N	Y	Y	Y	N	N	N
TS 3	Y	N	Y	Y	Y	N	Y	Y
TS 4	Y	N	Y	Y	Y	N	Y	Y
TS 5	Y	N	Y	Y	Y	N	Y	Y
TS 6	Y	N	Y	Y	Y	N	Y	Y
TS 7	Y	N	Y	Y	Y	N	Y	Y
TS 8	Y	N	Y	Y	N	N	Y	Y
TS 9	Y	N	Y	Y	Y	N	Y	Y
TS 10	Y	N	Y	Y	Y	N	Y	Y
TS 11	Y	N	Y	N	Y	N	Y	N
TS 12	Y	N	Y	Y	Y	N	Y	Y
TS 13	Y	N	Y	Y	Y	N	Y	Y

We tested our approach by using the popular triangle type program and by generating a set of mutants from it. We then verified how well all the test suites generated by our approach were able to detect the defects on the mutated programs. The results looked highly promising and encouraging as almost 90% of the software defects were found.

In the future we plan to improve on our approach by experimenting with different selection techniques. In this paper we were using roulette wheel and plan to try alternatives like tournament selection. We will try to test our BCO approach on software systems that contain more

complex decision structures. We will also address the few mutants that weren't detected by improving the fitness function and the test case generation.

6. REFERENCES

[1] Aljahdali, S., Taif, A., Ghiduk and El-Telbany, M. 2010. The limitations of genetic algorithms in software testing. In *Computer Systems and Applications*, Ham-mamet, 2010.

[2] Arcuri, A. and Yao, X. 2008. Search based software testing of object-oriented containers. *Inf. Sci.* 178, 15 (August 2008), 3075-3095.

[3] Baresel, A., Sthamer, H., and Schmidt, M. 2002. Fitness Function Design to Improve Evolutionary Structural Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO 2002), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1329-1336.

[4] Blanco, R., Tuya, J., and Adenso- Díaz, B. 2009. Automated test data generation using a scatter search approach. *Inf. Softw. Technol.* 51, 4 (April 2009), 708-720.

[5] Chen, T.Y., Leung, H., Mak, I.K. 2004. Adaptive random testing. In *Proceedings of the Ninth Asian Computing Science Conference* (ASIAN'04), Lecture Notes in Computer Science, Vol 3321, 320–329.

[6] Chong, C. S., Sivakumar, A. I., Malcolm Low, Y. H., Gay, K. L. 2006. A bee colony optimization algorithm to job shop scheduling. In *Proceedings of the 38th conference on Winter simulation* WSC '06, California, 1954-1961.

[7] Díaz, E., Tuya, J., Blanco, R., and Dolado, J.J. 2008. A tabu search algorithm for structural software testing. *Comput. Oper. Res.* 35, 10 (October 2008), 3052-3072.

[8] Fraser, G., Arcuri, A., and McMinn, P. A Memetic Algorithm for Whole Test Suite Generation. *Journal of Systems and Software.* 103, 311-327.

[9] Fraser, G., and Arcuri, A. 2013. Whole Test Suite Generation, IEEE Transactions on Software Engineering, vol.39, no.2, (February 2013), 276,291.

[10] Harman, M., Jia, Y., and Zhang, Y. 2015. Achievements, Open Problems and Challenges for Search Based Software Testing. Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference. (April 2015), 1,12, 13-17, 13-17.

[11] Harman, M., Afshin Mansouri, S., and Zhang, Y. 2012. Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45, 1, Article 11.

[12] Jones, B.F., Sthamer, H.H., and Eyres, D.E. 1996. Automatic Structural Testing Using Genetic Algorithms. Software Engineering Research Journal. (September 1996), 299-306.

[13] Kempka, J., McMinn, P. and Sudholt, D. To Appear. Design and Analysis of Different Alternating Variable Searches for Search-Based Software Testing. Theoretical Computer Science.

[14] Korel, B.. 1990. Automated software test data generation. Software Engineering, IEEE Transactions on, vol.16, no.8, 870-879.

[15] L.P.Wong, M. Y. H. low, and C. S. Chong. 2008. A Bee Colony Optimization Algorithm for Travelling Salesman Problem. Second Asia International Conference on Modeling & Simulation, IEEE Computer Society, Washington, DC, USA, 818-823.

[16] Lucic, P. and Teodorovic, D. 2001. Bee system: Modeling combinatorial optimization transportation engineering problems by swarm intelligence. In *Preprints of the Tristan IV Triennial Symposium on Transportation Analysis*. SaoMiguel, Azores Islands, Portugal, 441-445.

[17] Malburg, J., and Fraser, G. 2011. Combining search-based and constraint-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (ASE '11). IEEE Computer Society, Washington, DC, USA, 436-439.

[18] Mansour, N., and Salame, M. 2004. Data generation for path testing. Software Quality Journal, 2004, 121-36.

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

[20] Pargas, R. P., Harrold, M.J., and Peck, R.R. 1999. Test Data Generation Using Genetic Algorithms. Journal of Software Testing, Verifications, and Reliability. 9, (September 1999), 263-282.

[21] Rathore, A., Bohara, A., Gupta Prashil, R., Lakshmi, T.S., et al. 2011. Application of genetic algorithm and tabu search in software testing. In *Proceedings of the Fourth Annual ACM Bangalore Conference* (COMPUTE '11). ACM, New York, NY, USA, Article 23, 4 pages.

[22] Teodorovic, D., and Dell, M. O. 2005. Bee colony optimization - a cooperative learning approach to complex transportation problems. In *Proceedings of 10th EWGT Meeting and 16th Mini EURO Conference*, 51- 60.

[23] Tracey, N., Clark, J., Mander, K., and McDermid, J. 1998. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering*. Hawaii, USA, 1998, 285–288.

[24] Waeselynck, H., Fosse, P. T., and Kaddour, O. A. 2007. Simulated annealing applied to test generation: landscape characterization and stopping criteria. *Empirical Softw. Engg.* 12, 1 (February 2007), 35-63.