International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 1 Issue 9, November- 2012

Ant Colony Optimization : Algorithms of Mutation Testing

Mohit Tiwari
M.Phil(IT)/MCA/B.Sc (PCM)

Dr. Vaibhav Bansal
Ph.D(CSE)/M.Tech(IT)/B.E.Hons(CSE)

Arti Bajaj
MCA/B.Sc (Computer Application)

Abstract:
To reduce the cost of test data generation in the context of mutation testing, this paper has proposed an evolutionary approach based on Ant Colony Optimization. Inspired by Bottaci, they have defined and implemented a fitness function. This function measures how close a test case is to kill a mutant. To better guide the search for continuous input parameters the Ant Colony Optimization based approach is enhanced by a probability density estimation technique. The enhanced Ant Colony Optimization approach performed significantly better than Genetic Algorithm, Hill Climbing and Random Search in terms of attained mutation score as well as computational cost.

For the better evaluation of the approach, it should be compared to an enhanced Genetic Algorithm that will involve the probability density estimation technique. Further improvements of the approach will also be achieved by considering other types of inputs and predicate expressions using strings, arrays and Booleans.

1. Introduction

Ant Colony Optimization is an example of Swarm Intelligence (SI is the collective behaviour of decentralized, self organized systems, natural or artificial).
Ant Colony Optimization (ACO) is a paradigm for designing metaheuristic algorithms for combinatorial optimization problems. The essential trait of ACO algorithms is the combination of prior information about the structure of a promising solution with later information about the structure of previously obtained good solutions.

Metaheuristic algorithms are algorithms which, in order to escape from local optima, drive some basic heuristic: either a constructive heuristic starting from a null solution and adding elements to build a good complete one, or a local search heuristic starting from a complete solution and iteratively modifying some of its elements in order to achieve a better one. The metaheuristic part permits the low-level heuristic to obtain solutions better than those it could have achieved alone, even if iterated. Usually, the controlling mechanism is achieved either by constraining or by randomizing the set of local neighbour solutions to consider in local search (as is the case of simulated annealing), or by combining elements taken by different solutions (as is the case of evolution strategies and genetic algorithms) [2].

Keywords: Swarm Intelligence, Met heuristic algorithms

1.1 Advantages Of Ant Colony Optimization

1. Inherent parallelism is possible
2. Positive feedback accounts for rapid discovery of good solutions
3. Efficient for traveling salesman problem and similar other problems
4. ACO can be used in dynamic applications (adapts to changes such as new distances, etc)

1.2 Disadvantages Of Ant Colony Optimization

1. Theoretical analysis is difficult in ACO.
2. Sequences of random decisions are not possible.
3. Probability distribution changes by iteration.
4. Research is experimental rather than theoretical.
5. Time to convergence is uncertain but convergence is guaranteed [4].

1.2 Software Analysis

Software testing accounts for more than 40-50% of the total development costs in many software organizations. In any software project, software testing and test case generation are among the most manual labor intensive and technically difficult activities. Indeed, thorough testing is often unfeasible because of the potentially infinite execution space or high cost with respect to tight budget limitations.

A set of test cases is more adequate if it kills a larger number of mutants than another set of test cases. 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.

1.3 Ant System

Ant Colony Optimization is a class of algorithms, whose first member is called as The Ant System which was initially proposed by Colorni, Dorigo and Maniezzo. The main underlying idea, loosely inspired by the behavior of real ants, is that of a parallel search over several constructive computational threads based on local problem data and on a dynamic memory structure containing information on the quality of previously obtained result. The collective behavior emerging from the interaction of the different search threads has proved effective in solving combinatorial optimization (CO) problems.

Ant System was the first algorithm inspired by real ant's behavior. Ant System was initially applied to the solution of the traveling salesman problem but was not able to compete against the state-of-the art algorithms in the field. On the other hand he has the merit to introduce ACO algorithms and to show the potentiality of using artificial pheromone and artificial ants to drive the search of always better solutions for complex optimization problems.

The next researches were motivated by two goals:

1. The first was to improve the performance of the algorithm
2. The second was to investigate and better explain its behavior.

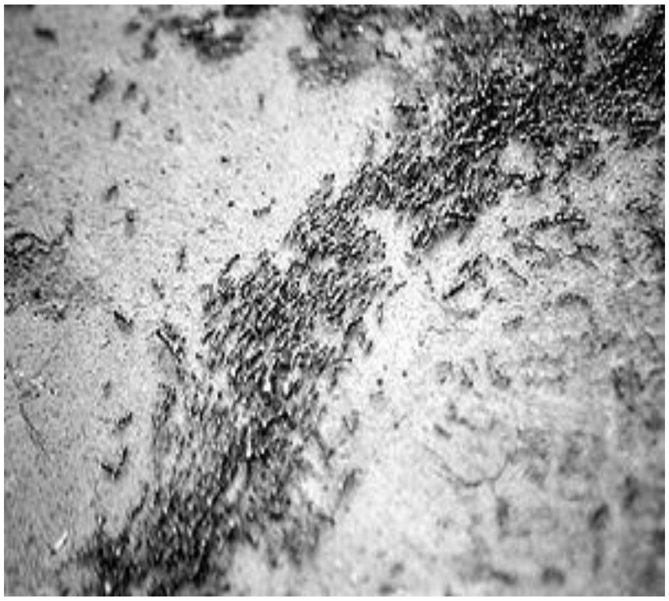
Figure 1.3: Ant System

1.4 Mutation Testing

A method of software testing, that involves modifying programs source code or byte code in small ways is called Mutation Testing. It is also called mutation analysis or program mutation. A test suite which does not detect and reject the mutated code can be considered defective. These so-called mutations are based on well-defined mutation operators. These mutation operators either force the creation of valuable tests (such as driving each expression to zero) or mimic typical programming errors (such as using the wrong operator or variable name). The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution. Mutation testing was originally proposed by DeMillo in 1978.

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

1. **Do Fewer**: The "do fewer" strategy looks for ways of generating and running fewer mutants without losing efficiency; among them it is worth mentioning selective mutation and mutant sampling.

2. **Do Smarter:** The "do smarter" approaches look for ways to distribute the expensive computational phases over several machines or to avoid complete execution. For example, weak mutation is a strategy belonging to the "do smarter" approach.

3. **Do Faster**: The "do faster" approach look for ways of generating and running each mutant as quickly as possible; among them are schema based mutation and separate compilation .

1.4.1 Offutt's Techniques

This paper focuses on automatic test input data generation which attempts to alleviate deficiencies of the approaches that were used previously. 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.
There are two techniques used by Offutt namely:
(i) Constraint Based Test Data Generation Technique
(ii)Dynamic Domain Reduction Technique

Constraint Based Test Data Generation Technique

This technique is based on the observation that a test case is able to kill a mutant if it satisfies three conditions :

1. The first condition is called - The Reachability Condition which states that the mutated statement must be reached.
2. The second condition is called - The Necessary Condition which requires that the execution state of the mutant

program must differ from that of the original program after some execution of the mutated statement.

3. The third condition is called – The Sufficiency Condition which requires that the state difference should 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.

Dynamic Domain Reduction Technique

To overcome some CBT problems this technique was successively developed by Offutt .

The basis of this technique 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 this technique, test input data generation is mapped into a minimization problem guided by a cost function, a fitness function inspired by Bottaci proposal.

2. Algorithms of Mutation Testing

2.1 Formulation Of The Problem

Let P be a program under test and I = (x_1, x_2, . . . , x_k) be the vector of its input variables. Each input variable x_i takes its values in a domain D_i where i = 1, 2 , . . . , k. Therefore the domain of the program P is the cross product D where $D = D_1 \times D_2 \ldots \times D_k$. Let us further assume that R is a set of mutation operators where 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 P.

By applying mutation operator r ε R to P, N mutated copies $M_1, M_2, \ldots, M_N$ of P are obtained. In other words, $M_i = r_i(P)$ with r_i ε R, r_i the i^{th}, i = 1, . . . ,N, a selected mutation operator that mutates P by injecting a simple fault at a statement s_m, called the P 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, . . . , x_k), called test cases, such that when the test suite is executed over the set of mutants $M_1, M_2, \ldots, M_N$ it kills the highest possible number of mutants.

As already mentioned, each mutant M_j , j = 1, . . . ,N, is killed if the three conditions promulgated by Offut 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 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 M Score(T) given by the following formula:

$$M\ Score\ (T) = 100\ \ (d / N – eq)$$

where eq is the number of equivalent mutants i.e., mutants that cannot be distinguished from P. Input variables x_1, x_2, . . . , x_k taking values in $D_1 \times D_2 \ldots \times D_k$ are assumed to be either integer values or real values.

2.2 Fitness Function

A fitness function is a particular type of objective function that prescribes the optimality of a solution (that is, a chromosome) in a genetic algorithm so that that particular chromosome may be ranked against all the other chromosomes. Optimal chromosomes, or at least chromosomes which are more optimal, are allowed to breed and mix their datasets by any of several techniques, producing a new generation that will (hopefully) be even better.

An ideal fitness function correlates closely with the algorithm's goal, and yet may be computed quickly. Speed of execution is very important, as a typical genetic algorithm must be iterated many, many times in order to produce a usable result for a non-trivial problem. This is one of the main drawbacks of Genetic Algorithms in real world applications and limits their applicability in some industries.

Two main classes of Fitness Functions exist:

- The first one where the fitness function does not change, as in optimizing a fixed function or testing with a fixed set of test cases
- The second one where the fitness function is mutable, as in niche differentiation or co-evolving the set of test cases [7].

2.2.1 Bottaci's Fitness Function

This 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. Bottaci mapped the three conditions into three cost terms which goes as: The reachability cost for a given test case is computed as the goal path minus the number of nodes in the longest common prefix of the test case path and goal path. 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. 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.
To solve the minimization problem, Ant Colony Optimization (ACO) is chosen as the metaheuristic algorithm.

Reasons Justifying the Choice

1. Evolutionary algorithms have been proven to be suitable approaches for data generation in the context of coverage based testing.
2. 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. Automatically determining which mutants are equivalent is also an important way to reduce the manual labors and promote the acceptance of mutation testing.

2.3 Hill Climbing Algorithm

In computer science, Hill Climbing is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

For example, hill climbing can be applied to the traveling salesman problem. It is easy to find an initial solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much shorter route is likely to be obtained.

Hill climbing is good for finding a local optimum (a good solution that lies relatively near the initial solution) but it is not guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space).

The relative simplicity of the algorithm makes it a popular first choice amongst optimizing algorithms. It is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms. Although more advanced algorithms such as simulated annealing or tabu search may give better results, in some situations hill climbing works just as well. Hill climbing can often produce a better result than other algorithms when the amount of time available to perform a search is limited, such as with real-time systems.

2.3.1 Variants Of Hill Climbing

1. Simple Hill Climbing - In this variant the first closer node is chosen.

2. Steepest Ascent Hill Climbing - In this variant all successors are compared and the closest to the solution is chosen. This is similar to best-first search, which tries all possible extensions of the current path instead of only one. This fails if there is no closer node, which may happen if there are local maxima in the search space which are not solutions.

3. Stochastic Hill Climbing - This variant does not examine all neighbors before deciding how to move. Rather, it selects a neighbor at random, and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

4. Random-Restart Hill Climbing - This variant is a meta-algorithm built on top of the hill climbing algorithm. It is also known as Shotgun Hill Climbing. It iteratively does hill-climbing, each time with a random initial condition x_0. The best x_m is kept: if a new run of hill climbing produces a better x_m than the stored state then it replaces the stored state. Random-Restart Hill Climbing is a surprisingly effective algorithm in many cases. It turns out that it is often better to spend CPU time exploring the space, than carefully optimizing from an initial condition [5].

2.3.2 **Problem Structure**

Hill Climbing is the simplest and probably best known search based algorithm. The goal of this paper was to simplify the task of comparison with Ant Colony Optimization. In order to kill a given mutant, Hill Climbing 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 Ant Colony Optimization and Genetic Algorithm. Hill Climbing 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) is 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 the value of the input variable by a step.

2.4 <u>**Genetic Algorithm**</u>

A genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, crossover, selection. In genetic algorithms of computing, **mutation** is a genetic operator used to maintain genetic diversity from one generation of a population of algorithm chromosomes to the next. It is analogous to biological mutation.

2.4.1 <u>**Methodology**</u>

In a genetic algorithm, a population of strings (called chromosomes), which encode candidate solution (called individuals, creatures) to an optimization problem, evolves toward better solutions. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached.

A typical genetic algorithm requires:

- a genetic representation of the solution domain.
- a fitness function to evaluate the solution domain.

The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple crossover operations. Variable length representations may also be used, but crossover implementation is more complex in this case.

The fitness function is defined over the genetic representation and measures the quality of the represented solution. The fitness function is always problem dependent. Once we have the genetic representation and the fitness function defined, GA proceeds to initialize a population of solutions randomly, and then improve it through repetitive application of mutation, crossover, inversion and selection operators.

2.4.2 <u>Initialization</u>

Initially many individual solutions are randomly generated to form an initial population. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. Traditionally, the population is generated randomly, covering the entire range of possible solutions (the search space). Occasionally, the solutions may be "seeded" in areas where optimal solutions are likely to be found.

2.4.3 <u>Problem Domains</u>

Problems which appear to be particularly appropriate for solution by genetic algorithms include timetabling and scheduling problems, and many scheduling software packages are based on Genetic Algorithms. Genetic Algorithms have also been applied to engineering. Genetic algorithms are often applied as an approach to solve global optimization problems.

As a general rule genetic algorithms might be useful in problem domains that have a complex fitness landscape as mixing, i.e., mutation in combination with crossover, is designed to move the population away from local optima that a traditional hill climbing algorithm might get stuck in. Observe that commonly used crossover operators cannot change any uniform population. Mutation alone can provide ergodicity of the overall genetic algorithm process (seen as a Markov chain) [6].

2.4.4 Comparison Of Genetic Algorithm To ACO

This paper compares Genetic Algorithm, with the proposed Ant Colony Optimization. Genetic Algorithm 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 and genes are values of the input variables. In an iterative process, Genetic Algorithm 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 one 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 M_i is killed or stagnation criteria).The experiment performed in this paper chooses, crossover to be the uniform crossover: where 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.

Reference:

1. Automatic mutation test input data generation via ant colony by K.Ayari, S.Bouktif, G.Antoniol
2. http://www.idsia.ch/~luca/aco2004.pdf
3. http://en.wikipedia.org/wiki/Ant_colony_optimization
4. http://www.slideshare.net/pratikpoddar05051989/ant-colony-optimization
5. http://en.wikipedia.org/wiki/Hill_climbing
6. http://en.wikipedia.org/wiki/Genetic_algorithm
7. http://en.wikipedia.org/wiki/Fitness_function
8. http://www.springerlink.com/content/x3a34th2vt046mw8
9. http://en.wikipedia.org/wiki/Probability_density_function