AUTOMATIC TEST-DATA GENERATION FOR MODIFIED CONDITION/ DECISION COVERAGE USING GENETIC ALGORITHM

P. MARAGATHAVALLI
Department Of Information Technology, Pondicherry Engineering College, Puducherry-605014, India

M. ANUSA, P. GEETHAMALINI, S. PRIYADHARSINI
Department Of Information Technology, Pondicherry Engineering College, Puducherry-605014, India

Abstract:
One area where SBSE has seen much application is test data generation. Search based test data generation techniques have been applied to automatically generate data for testing functional and non-functional properties of software. For structural testing, most of the time the criterion used is branch coverage. However, a single criterion is not enough for effective testing. For the wider acceptance of search based test data generation techniques, much stronger criteria like MCDC is needed. Structural testing provides confidence in the correct functioning of the software in its intended environment. Experiments have been performed using simulated annealing to generate test data for simple programs and the parameters of simulated annealing have been optimized through a small set of experiments. This concept can be extended further to include other search based algorithms and to test programs with difficult branching structures. Comparisons are made with the results obtained from genetic algorithm versus random testing. The experimental results show that the generated test cases give a higher structural coverage such as condition, statement, MCDC and multiple condition coverage.

Keywords: Search-based software engineering, MCDC, coverage criterions, structural testing, genetic algorithm.

1. Introduction
"Search-based software engineering is a reformulation of software engineering as a search problem, in which the solution to a problem is found by sampling a large search space of possible solutions."

According to Harman and Jones, metaheuristic search techniques, such as evolutionary algorithms, are applicable to many software engineering related optimisation and search problems because there are usually need to balance competing constraints and cope with inconsistency without any precise rules to compute the best solution. There are usually too many solutions without a perfect one, but still good solutions can be recognised from the bad ones.

Search methods with metaheuristic nature such an evolutionary algorithms, tabu search, and simulated annealing are used as sampling techniques [2]. The most used search techniques in search-based software engineering are evolutionary algorithms including techniques such a genetic algorithms (GA), genetic programming (GP), evolution strategy (ES), and evolution programming (EP).

One challenge of search-based software engineering is to reformulate software engineering problems as search problems. Different methods based on evolutionary algorithms have been successfully applied to
software engineering problems. One area where Search-Based Software Engineering has seen much application is test data generation.

The remaining of this paper is organized as follows: Section 2 describes the evolutionary testing. Section 3 describes the search-based techniques. Section 4 tells about the various coverage criterions and Section 5 contains the process of Automatic test data generation, Section 6 has the Results and Discussion, while the paper is concluded with Section 7 and 8 containing the conclusion and future work.

2. Evolutionary Testing

Evolutionary Testing (ET) is a search-based software testing approach based on the theory of evolution. It formulates the task to generate relevant test data (relevant in terms of the testing objective at hand, such as maximizing structural coverage) as one or several search problems. Each search problem consists of the definition of the search space based on the input domain of the target program (e.g., its relevant parameters), and a fitness function that ET constructs.

In the case of structural testing, such a search problem aims at finding a test data leading to the coverage of a particular branch. Each search problem is tried to be solved using an evolutionary algorithm: a pool of candidate test data, the so-called individuals, is iteratively manipulated by applying fitness evaluation, selection, mutation, and crossover in order to eventually obtain a relevant test data. Such iteration is named as a generation.

For fitness evaluation, the candidate test data is executed. Better fitness values are assigned to individuals that are better able to solve the search problem at hand, e.g., coming closer to covering the target branch during execution. ET has been found to achieve better performance than random testing as it concentrates the search toward finding test data with high fitness values.

2.1 Structural testing:

For structural testing, such as branch testing, the fitness value is usually determined based on how close the candidate test data comes to cover the target branch during execution. The closeness is typically expressed as the sum of the two metrics approximation level and local distance. Approximation level is defined in terms of the control flow graph of the target program and criticality of branching nodes. A branching node is critical if no path exists from one of its outgoing branches to the target. Approximation level is measured as the number of critical branching nodes between the path taken during execution and the target branch. Local distance is defined in terms of the condition of that critical branching node at which execution diverged away from the target branch. Its values are within [0, 1].

To determine the local distance, a condition-specific distance function is constructed by composing operator-specific atomic distance functions. For each relational operator of the programming language, an atomic distance function exists. The lower its value, the closer the condition is to being evaluated in favor of the non-critical branch. Since both metrics are non-negative and 0 in the case of an optimal test data, the evolutionary searches aim at minimizing the fitness values of the candidate test data.

1. Metaheuristic Search Techniques:

This section provides a brief overview of three metaheuristic search techniques that have been most widely applied to problems in software engineering: hill climbing, simulated annealing and genetic algorithms.

3.1. Hill climbing

In hill climbing, the search proceeds [2] from randomly chosen point by considering the neighbors of the point. Just what constitutes a near neighbour is problem specific, but typically neighbors are a ‘small mutation away’ from the current solution. A move is made to a neighbor that improves fitness. There are two choices: In next ascent hill climbing, the move is made to the first neighbour found to have an improved fitness. In steepest ascent hill climbing, the entire neighborhood set is examined to find the neighbor that gives the greatest increase in fitness. If there is no fitter neighbor, then the search terminates and a (possibly local) maximum has been found. Figuratively speaking, a ‘hill’ in the search landscape close to the random starting point has been climbed. Clearly, the problem with the hill climbing approach is that the hill located by the algorithm may be local maxima, and may be far poorer than global maxima in the search space. For some landscapes, this is not a
problem because repeatedly restarting the hill climb at a different location may produce adequate results (this is known as multiple restart hill climbing). Despite the local maxima problem, hill climbing is a simple technique which is both easy to implement and surprisingly effective.

3.2. Simulated annealing

Simulated annealing can be thought of as a variation of hill climbing that avoids the local maxima problem by permitting moves to less fit individuals. Simulated annealing is a simulation of metallurgical annealing, in which a highly heated metal is allowed to reduce in temperature slowly, thereby increasing its strength. As the temperature decreases the atoms have less freedom of movement. However, the greater freedom in the earlier (hotter) stages of the process allows the atoms to ‘explore’ different energy states. A simulated annealing algorithm will move from some point \( x_1 \) to a worse point \( x_1 \) with a probability that is a function of the drop in fitness and a ‘temperature’ parameter that (loosely speaking) models the temperature of the metal in metallurgical annealing. The effect of ‘cooling’ on the simulation of annealing is that the probability of following an unfavorable move is reduced.

At the end of the simulated annealing algorithm, the effect is that of pure hill climbing. However, the earlier ‘warmer’ stages allow productive exploration of the search space, with the hope that the higher temperature allows the search to escape local maxima. The approach has found application in several problems in search based software engineering [8].

3.3. Tabu search

This section, presents how tabu search is applied to software testing. The developed generator [11], TSGen, has the goal of covering all the branches of the program under test, i.e. to cover all the nodes of its control flow graph CFG. TSGen generates tests (partial solutions) and executes them as input for the program under test. A test \( x \) is formed by a vector (or tuple) of given values \( (v_1, v_2, ..., v_n) \) for the input variables \( (x_1, x_2, ..., x_n) \) of the program under test. The set of values for a variable \( x_i \) is determined by its type (integer, float or character). TSGen generates tests based on the test that is the Current Solution (CS). Initially, the Current Solution is a random test, but, inside the loop, TSGen selects it according to which sub-goal node has to be covered. Using the Current Solution, TSGen generates a set of neighbouring test candidates. When a test is generated, TSGen checks whether it is a tabu test. A test is tabu if it is stored in the TSGen memory. In short, TSGen has a memory formed by two tabu lists: the short-term tabu list (ST) and the long-term tabu list (LT). If a generated test is not tabu, the instrumented program under test is executed to check which branches (nodes) it has covered and the cost incurred by said test. However, if a generated test is tabu, it will be rejected. During the search process, the best solutions found are stored together with their costs in the CFG. Thus, when an executed test has a lower cost in a CFG node than the current cost stored in that node, that test is stored as the best solution for that node. The program under test could have unfeasible or very difficult branches to be covered. For this reason, TSGen includes a backtracking process. In the backtracking process, TSGen will reject the Current Solution and store it in its LT memory. As a result of the backtracking process, TSGen will regenerate the search to the sub-goal node or will mark the sub-goal node as unfeasible and start the search to reach a new non-covered sub-goal node.

3.4. Genetic algorithm

The genetic algorithm (GA) is an example of an evolutionary algorithm which is actually a heuristic search technique. The basic idea of the algorithm is that to start with a randomly initialized population of individuals. Each individual is a potential candidate solution of a given problem. A fitness function is used to evaluate the adequacy and quality of each individual. After this, a selection process, which is based on the fitness associated to each individual, extracts a subset from the current population. This means that fitter solutions are more likely to be selected. These selected individuals are combined to form a new generation of population. The combination is usually done through a crossover operation, which takes two individuals and exchanges their information at a random selected position. Often a mutation process is applied, to prevent that individuals become too similar and thus the population is evaluated again, and the process is repeated until a specific termination condition is satisfied. The basic GA process steps are shown below:
The algorithm will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available). Fig. 1 shows the flowchart of a simple genetic algorithm.

3.5. Scatter search technique

Scatter Search is an evolutionary method that works on a set of solutions, called the Reference Set, which stores the best solutions that have been generated so far. The solutions in this set are combined in order to obtain new ones, trying to generate each time better solutions, according to quality and diversity criteria. The scatter search algorithm begins by using a diversity generation method to generate \( P \) diverse solutions, to which an improvement method is applied. Then the reference set is created with the best solutions from \( P \) and the most diverse in relation to the solutions already in the reference set. As new solutions are generated, the algorithm produces subsets of the reference set using a subset generation method, and applies a solution combination method in order to obtain new solutions, to which an improvement method is applied. Then a ‘reference set update method’ evaluates the new solution to verify whether they can update the reference set, or not. If so, the best solutions are included in the reference set and the worst solutions are dropped.

3.6. Ant colony optimization

Ant Colony Optimization (ACO) algorithm is inspired by observation on real ants. Individually each ant is blind, frail and almost insignificant yet by being able to cooperate with each other the colony of ants demonstrates complex behaviors. One of these is the ability to find the closest route to a food source or some other interesting land mark. This is done by laying down special chemicals called pheromones. As more ants use a particular trail, the pheromone concentration on it increases hence attracting more ants.

ACO like any other meta-heuristic algorithm [12], 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 meta-heuristic 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 neighbor solutions to consider in local search, or by combining elements taken by different solutions. The particular way of defining components and associated probabilities is problem-specific, and can be designed in different ways, facing a trade-off between the specificity of the information used for the conditioning and the number of solutions which need to be constructed before effectively biasing the probability distribution to favor the emergence of good solutions.

4. Coverage Criteria:

The various coverage criterions [7] considered in search-based software testing are as follows:

- **Statement coverage (SC)**: every statement in the program has been executed at least once.
- **Decision coverage (DC)**: every statement in the program has been executed at least once, and every decision in the program has taken all possible outcomes at least once.
- **Decision/condition coverage (D/CC)**: every statement in the program has been executed at least once, every decision in the program has taken all possible outcomes at least once, and very condition in each decision has taken all possible outcomes at least once.
- **Condition coverage (CC)**: every statement in the program has been executed at least once, and every condition in each decision has taken all possible outcomes at least once.
- **MCDC**: Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect the decision’s outcome.
- **Multiple condition coverage (MCC)**: every statement in the program has been executed at least once, and all possible combinations of condition outcomes in each decision have been invoked at least once.

The detailed coverage criteria decision making is shown in table 1.

Table 1. Various coverage criterions used in search-based software testing (SBST)

<table>
<thead>
<tr>
<th>Coverage Criteria</th>
<th>Statement Coverage</th>
<th>Decision Coverage</th>
<th>Condition Coverage</th>
<th>Condition/Decision Coverage</th>
<th>MC/DC</th>
<th>Multiple Condition Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Every point of entry and exit in the program has been invoked at least once</td>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Every statement in the program has been invoked at least once</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Every decision in the program has taken all possible outcomes at least once</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>Every condition in a decision in the program has taken all possible outcomes at least once</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Every condition in a decision has been shown to independently affect that decision’s outcome</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Every combination of condition outcomes within a decision has been invoked at least once</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. Automatic Test Data Generation:

Most of the work on Software Testing has concerned the problem of generating inputs that provide a test suite that meets a test adequacy criterion. Often this problem of test input generation is called “Automated Test Data Generation (ATDG)” though [13], strictly speaking, without an oracle, only the input is generated. Fig. 2 illustrates the generic form of the most common approach in the literature, in which test inputs are generated according to a test adequacy criteria. The test adequacy criterion is the human input to the process. It determines the goal of testing.
The adequacy criteria can be almost any form of testing goal that can be defined and assessed numerically. For instance, it can be structural (cover branches, paths, statements) functional (cover scenarios), temporal (find worst/best case execution times) etc.

This generic nature of Search-Based Testing (SBT) has been a considerable advantage and has been one of the reasons why many authors have been able to adapt the SBT approach different formulations.

The adequacy criteria must be captured by a fitness function. This has to be designed by a human, but once a fitness function has been defined for a test adequacy criterion, C, then the generation of C-adequate test inputs can be automated using SBSE.

The SBSE tools that implement different forms of testing all follow the broad structure outlined in Fig.4. They code the adequacy as fitness, using it to assess the fitness of candidate test inputs. In order to assess fitness, the ATDG system has to cause the program to be executed for the candidate inputs. The ATDG system then monitors the execution to assess fitness (how well does the input meet the test adequacy criterion?).

![Fig. 2. A generic search-based test input generation scheme](image)

6. Results and Discussion:

The Genetic Algorithm results for the sample programs such as binary search, Line rectangle classifier and finding no. of days are taken and it is shown in table 2. The parameters considered are no. of test-cases generated, percentage of coverage (statement, condition, modified condition decision coverage, multiple conditions) and the search time. By using Genetic Algorithm, the generated test cases are reduced to minimum set with the maximum coverage; and also the time required for searching could be reduced by getting optimal solutions with highest fitness values.

On comparing GA with random testing for simple programs like linear search, quadratic equation, finding gcd etc. the results obtained [table 3] are not too significant. When the complexity of the program or the input domain grows, GA outperforms random testing. These results are shown in table 4 for triangle classifier problem.
Table 2. Results obtained for programs satisfying various coverage criterions

<table>
<thead>
<tr>
<th>Coverage criterion</th>
<th>Binary Search problem</th>
<th>Line Rectangle Classifier problem</th>
<th>No. of days problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of test cases</td>
<td>Coverage (%)</td>
<td>Time (sec)</td>
</tr>
<tr>
<td>Statement</td>
<td>30</td>
<td>77.76</td>
<td>3</td>
</tr>
<tr>
<td>Condition</td>
<td>40</td>
<td>87.50</td>
<td>5</td>
</tr>
<tr>
<td>MCDC</td>
<td>40</td>
<td>90.00</td>
<td>4</td>
</tr>
<tr>
<td>Multiple condition</td>
<td>40</td>
<td>91.66</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3. Results obtained for simple programs using random testing

<table>
<thead>
<tr>
<th>Program name</th>
<th>Range of the input variables</th>
<th>% of the branch coverage</th>
<th>No. of test cases generated</th>
<th>Time consumed (in secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear search</td>
<td>1 to 50</td>
<td>95.26</td>
<td>10</td>
<td>1.78</td>
</tr>
<tr>
<td>Quadratic equation</td>
<td>-10 to 10</td>
<td>83.33</td>
<td>5</td>
<td>3.66</td>
</tr>
<tr>
<td>Greatest common divisor</td>
<td>1 to 100</td>
<td>80.3</td>
<td>10</td>
<td>1.97</td>
</tr>
</tbody>
</table>

Table 4. Results obtained for triangle classification problem using GA

<table>
<thead>
<tr>
<th>Testing</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA</td>
<td>2.04</td>
<td>3.82</td>
<td>1.36</td>
<td>1.08</td>
<td>2.36</td>
<td>2.13</td>
</tr>
<tr>
<td>RM</td>
<td>18.88</td>
<td>19.52</td>
<td>22.24</td>
<td>14.24</td>
<td>2.40</td>
<td>19.78</td>
</tr>
</tbody>
</table>

7. Conclusion:

Search based test data generation techniques have been applied to automatically generate data for testing functional and non-functional properties of software. For structural testing, most of the time the criterion used is branch coverage. Along with branch coverage, statement, decision, MCDC and multiple condition coverage criterions are also considered for test data generation using genetic algorithm. The experiments have been
performed for various sample programs show that genetic algorithm results with maximum coverage and minimum time requirement.

8. Future work:

Future research will involve comparing GA selected conditions in larger test data and further refining the method presented. This research would help in generating various software test cases. Other search algorithms can be combined with GA by considering different parameters for solving real-world problems.

References: