Simulated Annealing and Improved Genetic Algorithms for Path Testing

By

Rania Joumaa
B.Sc., Lebanese American University, 1994

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at the Lebanese American University
June 1997

RT
129

Simulated Annealing and Improved Genetic Algorithms for Path Testing

By

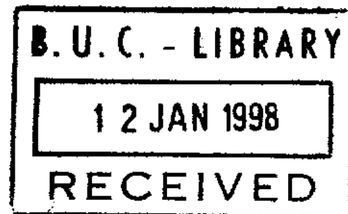
Rania Joumaa
B.Sc., Lebanese American University, 1994

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at the Lebanese American University June 1997

Signatures Redacted

Dr. Nashat Mansour (Advisor)
Assistant Professor of Computer Science
Lebanese American University



Signatures Redacted

Dr. George Nasr
Associate Professor of Electrical and Computer Engineering
Lebanese American University

Signatures Redacted

Dr. Haidar Harmanany
Assistant Professor of Computer Science
Lebanese American University

ABSTRACT

In this work, we formulate the path testing problem as an optimization problem by combining two previous ideas, and we propose a simulated annealing (SA) algorithm to solve it. We also improve a genetic algorithm (GA) that has been previously used. Our experimental results show that the SA and improved GA are useful for path testing and that they have comparable behavior and performance.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Professor Nashat Mansour, my advisor, for his support, tireless effort and encouragement. I am also grateful to the second readers, Professor George Nassar and Professor Haidar Harmanany for their help and interest.

Most of all, i am very grateful to my parents for their devotion to education as a priority in life.

CONTENTS

Chapter		
1. Introduction		**1**
2. Overview		**4**
2.1.	Levels of testing	4
2.2.	Black box and white box testing	6
	2.2.1 Black box testing	6
	2.2.2 White box testing	7
2.3.	Previous work on path testing	9
3. Path Testing		**11**
3.1.	Path testing problem	11
3.2.	Huang technique for forming path predicates and test case generation	12
	3.2.1. Forming path predicates	12
	3.2.2. Test case generation	13
3.3.	Objective function	15
4. Simulated Annealing Algorithm for Path Testing		**19**
4.1.	Solution representation	21
4.2.	Energy function	21
4.3.	Perturbation operation	22
4.4.	Accept/ reject criterion	22
4.5	Cooling schedule	23
4.6	Convergence	23
5. Improved Genetic Algorithm for Path Testing		**24**
5.1.	Chromosomal encoding	25
5.2.	Reproduction scheme	26
5.3.	Fitness evaluation	27
5.4.	Genetic operators and operator rates	27
5.5	Convergence	28
5.6	Improved features	28
6. Experimental Results		**29**
6.1.	Test case T1	29
6.2.	Test case T2	31
6.3.	Test case T3	36
6.4.	Test case T4	39
6.5	Discussion of results	41
7. Conclusion and Further Work		**43**
References		**45**

List of Figures

Chapter 4	Figure 1	Simulated Annealing Algorithm	20
Chapter 5	Figure 1	Genetic Algorithm	25
Chapter 6	Figure 1	Flow Graph of test case T2	32
	Figure 2	Flow Graph of test case T3	37

List of Tables

Chapter 6	Table 1	Results of path 1 for test case T1	30
	Table 2	Results of path 2 for test case T1	30
	Table 3	Results of path 1 for test case T2	33
	Table 4	Results of path 2 for test case T2	34
	Table 5	Results of path 3 for test case T2	34
	Table 6	Results of path 4 for test case T2	35
	Table 7	Results of path 5 for test case T2	35
	Table 8	Results of path 6 for test case T2	36
	Table 9	Results of path 1 for test case T3	38
	Table 10	Results of path 2 for test case T3	38
	Table 11	Results of path 1 for test case T4	40
	Table 12	Results of path 2 for test case T4	40
	Table 13	Results of path 3 for test case T4	41

Chapter 1

Introduction

Testing software is a means of measuring or assessing the software to ensure that its functioning correctly. Most practiced software process models include testing as a separate phase, after integration and before maintenance. However, testing is an inherent component of the software process. All the phases of any process model should be tested. During the specification phase, the specifications must be checked; the design phase must also be carefully checked at every stage. In the coding phase each module must certainly be tested, and the product as a whole needs testing at the integration phase. And it is after acceptance testing that the product goes into operation mode and that the maintenance begins.

Therefore, testing is needed in order to assess what the system actually does, and how it does it in its final environment. And, because of its importance, software developers tend to spend approximately 40% of development cost and time in testing, in order to achieve reasonable quality levels.

Testing is sometimes distinguished into static and dynamic analysis. Static techniques do not exercise the software, they only examine it, but they are considered part of testing because they do evaluate software quality. Dynamic testing techniques exercise the software using sample input values. Static analysis examines control flow and data flow, and can discover "dead code", infinite loops, uninitialized variables, unused data values, standards violations, etc. Static analysis does not replace dynamic analysis, but is a very useful quality check normally carried out before dynamic test execution. [Marciniak et al, 1994], [Goodenough et al, 1979]

In dynamic testing, test case generation can be based either on the function of the software, or on the internal structure of the code. Several techniques have been developed in both approaches, however, in our research, we will only consider structural testing.

Three forms of coverage exist in structural testing : statement coverage, branch coverage and path coverage. The easiest form is statement coverage, while the most powerful and complex one is path coverage.

Several techniques for path coverage have been proposed; for example path analysis testing, domain testing, Huang approach and data flow testing. However, path testing is a complicated problem, it is in general unsolvable in the sense that there does not exists a single algorithm that can be used to find assignments to input variables that will satisfy any given path predicate, or even to determine its satisfiability. [Huang, 1975]

The objective of our research is to present a new technique that can help in generating input values for path testing. We propose a simulated annealing (SA) algorithm to generate the desired input. We also improve a genetic algorithm (GA) developed by Jones and Sthamer for branch testing to obtain path coverage. Our new technique made use of Huang approach for forming path predicates, and converting inequalities to equalities, in order to work with complex conditions. The results of our experiments were satisfactory and we were capable to show that these random algorithms can be used to generate test cases for path testing.

This thesis is organized as follows. Chapter 2 talks about testing in general, in chapter 3 we present an overview of path predicate and objective function construction. Chapter 4 describes the simulated annealing algorithm used while chapter 5 describes the genetic algorithm. In chapter 6, we show the results obtained by applying the simulated annealing and the genetic algorithms to some realistic programs. Finally, we give our conclusion.

Chapter 2

Background

This chapter talks about testing in general. We will list the different levels of testing, then black and white box testing will be carefully discussed. Finally, we will describe some of the previously presented approaches to solve the path testing problem.

2.1. Levels of Testing

The focus of testing is to find errors, but different types of errors are looked at each level.

Unit testing: At the lower level, the function of the basic unit of software (module) is tested in isolation. The purpose of unit testing is to find errors in the individual units, in either data or logic. Tests can be derived from the detailed logic of the unit, with any additional structural tests derived from the physical design. [Beizer, 1983], [Korel, 1990]

Integration testing: When two or more tested components are combined into a larger structure, integration testing is needed. The tests should look for errors in the interface between the components, and in the functions which can be performed by the new group. Components are combined according to an integration or link test strategy, alternatives include top-down and bottom-up. [Beizer, 1990]

System testing: After integration testing is completed, the entire system is tested as a whole. Test case selection, at this level, is derived from the functional specification or the requirements specification. System testing looks for errors in the end-to-end functionality of the system, and also for errors in nonfunctional quality attributes, such as performance, reliability, security, etc. [Howden, 1986]

Acceptance testing: After system testing is completed, the system is handed over to the customer or users, and acceptance testing is performed. Acceptance testing is different in nature from development testing for three reasons. First, acceptance testing is the responsibility of the accepting organization rather than the developing organization. Secondly, the purpose of acceptance testing is to give confidence that the system is working rather then trying to find errors. Thirdly, acceptance testing includes the testing of the user organization's working practices, to ensure that the computer system will fit with critical and administrative procedures. [Beizer, 1990]

2.2. Black Box and White Box Testing

2.2.1. Black Box Testing

Black box or functional techniques are based on the function of the software. Tests are derived from a specification, either a requirements specification for system level tests, a design specification for integration tests, or a detailed module specification for unit tests. The structure of the code or system is not taken into account when designing functional tests.

The black box techniques are concerned with selecting test cases to exercise the software in a thorough and systematic way, where a test case consists of a sample input value, the expected result and the actual result with any other necessary information [Hetzel, 1991].

Several black box techniques exists like equivalence partitioning for example. The idea behind equivalence partitioning is that testing is more effective if the tests are distributed over all the different possibilities rather than all testing the same thing. [Weyker et al, 1980], [Hamlet et al, 1990] In equivalence partitioning input domain is divided into classes, and input values which are treated the same way by the software can be regarded as being in some sense equivalent, and thus, belonging to the same class.

Testing can also be improved by using the boundary value analysis technique. In this technique values which lie at the edge of an equivalence partition are used to reveal the most common types of coding errors which occur when the boundaries of the equivalence class are out by one. [Howden, 1986]

We can also state random testing, where a user profile of typical operational use is drawn up, and test inputs are generated randomly from the user profile. In this way, we can ensure that the quality of the system is investigated where it is the most critical and important for users. [Duran et al, 1984], [Ince et al, 1986]

Other techniques are also available like cause effect graphing and syntax testing.

2.2.2. White Box Testing

White box or structural test case selection techniques are based on the internal structure of the software. The choice of test cases is driven by looking inside the "box", and choosing test items to exercise the required parts of the structure.

Any software program can be expressed as a control flow diagram. Whenever there is a condition, such as an "IF" statement or a loop, then there will be more than one way for control to proceed, either down the "true" or down the "false" outcome. The point at which the condition is evaluated is called a decision point.

Using the control flow graph, the first test case is derived by following one path from the beginning to the end of the graph. The next test case to be chosen should follow at least one segment of the control flow graph which has not been followed before.

Determining what data needs to be input in order for a program to follow a certain path is called "sensitizing" the path, and this can be fairly complicated. [Macabe, 1982]

A number of different forms of structural testing exists, like statement, branch and path coverage. Statement coverage is running a series of test cases which will ensure that every statement is executed at least once. An improvement over statement coverage is branch coverage, that is ensuring that all branches are tested at least once. The most powerful form of structural testing is path coverage, that is testing all paths. In a product with loops the number of paths can be very large. As a result, researchers have been investigating ways of reducing the number of paths to be examined while still being able to uncover more faults than would be possible using branch coverage.

In our research we were mainly dealing with path testing, and we were trying to propose a new technique for path coverage.

2.3. Previous Work on Path Testing

Many approaches to the path testing problem have been proposed. One of them is path analysis testing, which constructs test data sets for selected paths in a program. Path analysis testing strategies typically use symbolic execution to provide a symbolic representation of a path. Symbolic execution assigns symbolic names to a program's input values and executes a path through the program. Throughout this execution, variables are maintained as algebraic expressions in terms of these symbolic names. The algebraic expressions for the output values provide a symbolic representation of the path computation. The conditional branches encountered along the path are represented as constraints in terms of the algebraic expressions of the variables referenced within the condition. The symbolic representations of the path computation and path domain are used by path analysis testing strategies to direct the selection of data to test a particular path. [Clarke et al, 1982]

We can also mention domain testing. The domain testing method is a modification of path analysis testing. The domain testing method guides in the selection of tests data by analyzing the boundary of a path domain. The program is tested with these data in order to reveal domain errors or provide confidence in the correctness of the path domain. Domain testing exploits the often observed fact that points near the boundary of a domain are most sensitive to domain error. [White et al, 1982]

Huang also proposed a technique for path testing. The first step in Huang approach is to form the path predicates by traversing the code, then inequalities are transformed to equalities by following a well defined strategy. The result of this technique is a number of equations that can be solved to obtain input values that can lead to the traversal of the selected path. [Huang, 1975]

Data flow testing is also a well known approach for path testing. It can reduce the number of needed paths while preserving quality. In this technique tests are derived so that adequate definition use associations are exercised by test cases. The occurrence of a variable, in the source code, is labeled either as a definition of the variable such as $z = 1$, or a use of the variable such as $y = z + 3$ or if $z > 10$. All paths between the definition of a variable and the use of that definition should be identified, and test cases are set up for each path. All-definition-use-path coverage is an excellent test technique, since large numbers of errors are frequently detected by relatively few test cases. [Jeng, 1994], [Rapps et al, 1985]

Path testing is a complicated problem. All of the previously presented techniques are difficult to use in practice for realistic programs and require a large amount of computation. For instance, the problem of test case generation is in general unsolvable in the sense that there is does not exists a single algorithm that can be used to find assignments to input variables that will satisfy any given path predicate, or even to determine its satisfiability. [Huang, 1975] Consequently, the objective of our research is to find a new technique that can help in generating input values for path testing.

Chapter 3

Path Testing

This chapter introduces the path testing problem, then describes Huang approach for forming path predicates and test case generation. Finally, the objective function used in both the simulated annealing and genetic algorithms is presented.

3.1. Path testing problem

A path through a program correspond to some possible flow of control. The structure of any tested program is usually mapped to a program flow chart. Associated with each conditional branch is a predicate, a logical combination of relational expressions. Therefore, a path test case is derived by following one path from the beginning to the end of the program along either the true or the false branches.

Consequently, associated with each path is the path domain, the subset of the program's domain that causes execution of the path, and path computation, the function that is computed by the execution of the statements along the path.

Path testing is the most powerful form of structural testing, but it is well known that they are the most difficult to achieve specially for large programs. Many problems can be encountered like large or infinite number of paths, difficulties in finding values for input variables.

3.2. Huang Technique for Forming Path Predicates and Test Case Generation

In general it is difficult to find the path predicates by inspection and we need a systematic method to derive it from the program text.

3.2.1. Forming Path Predicates

A branch in the flowchart will be traversed in execution if some predicate, called the branch predicate is satisfied at that stage of computation. The branch predicate for any branch can be found by examining the nature of the statement associated with the node from which this branch emanates. Consider two branches associated with predicate Q and R respectively, then the path formed by concatenating these two branches will be traversed if Q and R are both true.

Huang approach, in forming the path predicate, is based on dragging the predicates backward toward the first branch. Note, that passing through an assignment statement, the predicate remains unchanged unless the variable on the left-hand side of the assignment operator occurs in the predicate. The final path predicates constructed by using this technique are always expressed in terms of constants and input variables. Consequently, to cause a path to be traversed during execution, we should find the appropriate values to the input variables so that the obtained path predicate is satisfied.

3.2.2. Test Case Generation

To find an assignment to satisfy a predicate is not difficult in principle, but it requires a program having almost the computational power of a theorem prover. Therefore, Huang presented a systematic method for finding the desired set of assignment. We should note, that the degree of difficulty in finding the desired assignments increases as the number of atomic expressions involved, and as the number of variables in common increases.

The central problem, is to find a systematic method for finding an assignment that will satisfy a logical expression of the form :

$$P_1 \text{ and } P_2 \text{ and ...... and } P_n$$

where P_i is an atomic expression possibly prefixed by a not connective.

The steps are as follows:

1. Remove all not connectives involved. if P_i is of the form (not E_1 R E_2) then replace it with equivalent expression (E_1 R' E_2).

R	R'
$=$	$\neq$
$\neq$	$=$
$<$	$\geq$
$>$	$\leq$
$\leq$	$>$
$\geq$	$<$

2. Transform inequalities to equalities. Inequalities can be stated in terms of equalities as follows:

$a \neq b$	$\Leftrightarrow$	$(\exists x)_{\neq 0}$	$(x = b - a)$
$a < b$	$\Leftrightarrow$	$(\exists x)_{>0}$	$(x = b - a)$
$a \leq b$	$\Leftrightarrow$	$(\exists x)_{\geq 0}$	$(x = b - a)$
$a > b$	$\Leftrightarrow$	$(\exists x)_{>0}$	$(x = a - b)$
$a \geq b$	$\Leftrightarrow$	$(\exists x)_{\geq 0}$	$(x = a - b)$

3. Combine the new obtained equations to form a new equation in such a way that the number of variables involved in the equation is minimal

4. Assign initial values for the variables that were added to transform inequalities to equalities, then solve the system of equations obtained.

By using this technique, we can obtain the assignments that will constitute a test case that will test the desired path.

Finally, we would like to note that the algorithms that will be described in later sections will be based on Huang method for transforming inequalities to equalities.
[Huang et al, 1975]

3.3. Objective Function

Perhaps the most important aspect of using genetic and simulated annealing algorithms is the ability to find a suitable objective function, which is a numeric measure of how close each sample test set is to the goal. In our research, we have used an objective function based on predicates for path testing, and appropriate values for the objective function are derived automatically for each path predicates.

The process can be represented as a series of mappings. Initially the input variables $\{x_1,x_2.....,x_n\}$ are set at random and converted to binary format b_i, and then manipulated by the GA and the SA. The bit strings are then converted back to integer numbers in order to evaluate the predicates of the required path..

As an example, consider a program P under test, P maps its input values to two values X_{pk} and X'_{pk} corresponding to the evaluation of the functions on the two sides of each predicate K, which occur in the control flow of the selected path. Thus if there are m predicates in the selected path, P will map the input $\{x_1,x_2.....,x_n\}$ to m pairs of values $\{X_{p1}$ and $X'_{p1}\}$... $\{X_{pk}$ and $X'_{pk}\}$ ... $\{X_{pm}$ and $X'_{pm}\}$. The mapping will be different for each path corresponding to the input. The mapping will use the integer format of the input and not the binary one, since complex expressions may be computed to evaluate each side of the predicates. [Jones, Sthamer et al, 1996]

After obtaining the values corresponding to both sides of a predicate K for example, these values are converted back to binary format and passed to the objective function. The objective function for this may be based on the hamming distance between the operands of each predicate. The hamming distance operator maps pairs of bit strings relating to the Kth predicate to a numeric value. It is a count based on the number of different bits in the bit patterns for the two operands. The hamming operator is normally weighted so that the least significant bit has a weighting of 1, the next 2 etc. This is necessary because if the most significant bits differ, then substantial changes to the input pattern are usually needed [Jones, Sthamer et al, 1996].

Thus, the objective function is obtained by the following formula.

Σ (index of the bits that differ * corresponding weight)

Consider the following example where the Kth predicate is : A<4, and where A is an input variable and not an expression. Assume that the GA or the SA algorithm generated in one of its paths the value 7 for A . Then, 7 is converted to binary format to obtain 111, 4 is also converted to binary format to obtain 100. The two strings differ in bit number 0 and bit number 1, and since we are using weighted hamming distance then the weight of the first bit is 1 and the second bit is 2. Consequently, and by applying the previously stated formula, the value of the objective function is

((1*1)+(2*2)) = 5.

In the case of compound conditions, the value of the objective function for each predicate may be determined separately, and an overall value calculated by using multiplication for *or* predicates and addition for *and* predicates. Consider the following example, where the path predicate is : a<b and a+1 < c . Assume that the value of the objective function obtained for the first predicate (a<b) is 10, and that of the second predicate (a+1<c) is 20, then the overall objective function value would be 30=10+20 since the conjunction relating the two predicates is « *and* » .

We would like to add that the proposed algorithms that will be described in later sections can generate negative and positive numbers. Therfore, when the problem requires generation of only negative or positive values, then these constraints should be added to the path predicate already formed. This is essential, because these constraints are problem dependent and not genreal.

Moreover, the additional variables that will be added in order to convert inequalities to equalities are considered as input variables and are manipulated by the genetic and simulated annealing algorithms.

Chapter 4

Simulated Annealing Algorithm for Path Testing

Simulated annealing (SA) is motivated by an analogy to statistical mechanics of annealing in solids. To coerce some material into a low energy state, the material is annealed, it is heated to a temperature that permits many atomic rearrangements, then cooled carefully and slowly, allowing it to come to thermal equilibrium at each temperature, until the material freezes. A low energy state usually means a highly ordered state.

The simulated annealing algorithm simulates the natural phenomenon by a search process in the solution space optimizing some cost function. It starts with some initial solution at a high temperature and then reduces the temperature gradually to a freezing point. At each temperature, regions in the solution space are searched by the algorithm.

Simulated annealing proved to be successful in solving many optimization problems. In this section, we describe how simulated annealing can be adapted to solve not only optimization but also structural path testing problem. An outline of the SA algorithm is given in Figure 1.

```
Initial configuration;
Determine initial temperature T(0);
Determine freezing temperature Tf;
While (T(i) > Tf ) do
        for i= 1 to CONST
                Perturb configuration;
                If (accept_perturbation) then
                        Accept perturbation;
                        Save best so far;
                End if;
        End for;
T(i+1) = x T(i);
End while;


Procedure accept_perturbation()
        If (Δz <= 0) then
                Return (true);
        Else
                If (random(0,1) < e^(-Δz/T(i))) then
                        Return (true);
                Else
                        Return (false);
                End if;
        End if;
```

Figure 1. SA Algorithm

4.1. Solution Representation

An important decision in the simulated annealing algorithm is to present the sample test in a suitable form. In our work, we have found that the most efficient way to search the domain is to express all the input variables in a single, concatenated binary bit string. Therefore, if there are N input variables and the ith variable occupies n_i bits, the total size of the bit string is thus, $S = \sum n_i$ (i = 1...N).

The variables may be of different data types and of complicated data structures such as records.

In our algorithms, n_i was constant for all types of variables and it was equal to 8.

And since we were using binary representation, values for each variable were formed by using the signed integer representation.

4.2. Energy Function

The energy function is based on a weighted hamming distance between the operands in each predicate. In the case of compound conditions, the value of the objective function for each predicate may be determined separately, and an overall value calculated by using multiplication for *or* predicates and addition for *and* predicates. More details are given in section 3.3.

4.3. Perturbation Operation

First, the configuration is randomly initialized, starting and freezing temperatures are computed based on the fitness value of the initial population. Then, perturbation to the configuration takes place by randomly selecting a bit position from the configuration and changing its value to the opposite value; if the original value is 1 then it is modified to 0 and vice versa. It can be seen that the perturbation loop works for a constant number of iterations, CONST, which is dependent on the complexity of the problem. In our work, we use

CONST = 10 * number of bits in the configuration.

In each iteration the perturbation is either accepted or rejected. [Rutenbar, 1989]

4.4. Accept/ reject criterion

The acceptance criterion is outlined in Figure 1 in the accept_configuration procedure. This procedure checks the change in the cost function due to a perturbation. If the change decreases the cost function, the configuration is accepted. However, if the perturbation causes the cost function to increase, it is accepted only with a probability $e^{-\Delta z/T(i)}$. Note that for smaller temperature values T(i), the probability of accepting uphill moves becomes smaller, while at very low temperatures uphill moves are no longer accepted. [Rutenbar, 1989]

4.5. Cooling schedule

The initial temperature T(0) is the temperature that yields a high initial acceptance probability of uphill moves. It is calculated by deviding the cost value of the initial configuration by the logarithmic value of the selected acceptance probability. In our experiment, the acceptance probability is equal to 0.9

The freezing point is the temperature at which the acceptance probability is very small (2^{-30}), making uphill moves impossible and allowing only greedy moves.

The cooling schedule used in this work is imply $T(i+1) = \alpha\, T(i)$, with $\alpha = 0.95$

4.6. Convergence

Convergence is reached when no improvement is obtained in the fitness function over 100 consecutive passes. However, as the annealing algorithm searches the solution space, the best so far solution found is always saved. This guarantees that the output of the algorithm is the best solution it finds regardless of the temperatures and solutions it terminates at.

Chapter 5

Improved Genetic Algorithm for Path Testing

Genetic algorithms (GAs) are loosely based on ideas from genetics and Darwinian evolution by combining samples to achieve new and fitter individuals. In every generation, a new set of strings is created using bits and pieces of the fittest of the old. Thus, Genetic algorithms are an example of a class of adaptive searching techniques which are distinguished from classical methods because they sample the whole search space, calculate a fitness for each sample, and then evolve into a new set of samples in an attempt to generate a fitter test set [Holland, 1975].

Using genetic algorithms in structural testing was proposed by Jones and Sthamer [Jones, Sthamer et al, 1996]. They showed that GA has been applied successfully to obtain full branch coverage and controlling the number of a conditional loop.

In this work, we have improved their GA to obtain path coverage by using Huang technique described in the previous section. This improved GA is explained in this chapter and its outline is given in Figure 1.

```
Random generation of initial population, size POP;
Repeat
    Evaluate fitness of individuals;
    Preserve the fittest so far;
    Rank individuals and allocate reproduction trials;
    for i=1 to POP step 2
        Randomly select 2 parents from list of reproduction trials;
        Apply crossover and mutation;
    Endfor
Until (convergence)
Solution = fittest;
```

Figure 1. Genetic Algorithm

5.1. Chromosomal Encoding

All input variables are represented in a single concatunated binary bit string. More details are given in section 4.1.

We would like to mention, that the population size for the GA should depend on the complexity of the problem. It is preferable that the population size is equal to the length of the bit string. But, for large problems (where S is large) the number of members in the population may be limited without damaging the efficiency of the genetic algorithm.

5.2. Reproduction scheme

At the beginning of the evolution, each variable is initialized randomly. Then the formation of the next generation is done by selecting the fitter individuals from the old generation. The reproduction scheme consists of ranking of individuals followed by random selection of mates. In ranking, the individuals are sorted by their fitness value and are assigned a number of copies according to a predefined scale of equidistant values for the population. The ranks assigned to the fittest and least fit individual are 1.2 and 0.8 respectively. Individuals which rank greater than 1 are first assigned single copies. Then, the fractional part of their ranks and the ranks of the lower half of individuals are treated as possibilities for assignment of copies. [Jones, Sthamer rt al, 1996]

The proposed GA differs from the traditional GAs by the re-insertion of the best so far solution in the population size whenever there is no improvement in the solution. When a new population is obtained, the fitness value of the individual with the least fitness value is compared with the fitness of the best so far individual. If the fitness of the best so far individual is worst than that of the best individual in the new population, then the new individual replaces the best so far; otherwise we replace the individual with the worst fitness function in the new population with the best so far individual. This new technique lead to a great improvement in the performance of the GA.

5.3. Fitness evaluation

The fitness function for this is based on the weighted hamming distance between the operands of each predicate. And in the case of compound conditions, the fitness for each predicate may be determined separately, and an overall fitness calculated by using multiplication for *or* predicates and addition for *and* predicates. More details are given in section 3.3.

5.4. Genetic operators and operator rates

Genetic algorithms comprise two main operations : crossover and mutation. For the crossover operation, two members of the population are chosen, a point along the bit string is chosen at random, and the tails of the two bit strings are exchanged.

Mutation simply means that any bit at a position chosen at random is flipped from 0 to 1 or vice versa [Jones, Sthamer rt al, 1996]. In the implemented algorithms, the maximum number of mutation allowed in each generation is equal to half the size of the population. The crossover and mutation operators are performed probabisticaly, the rate of crossover used is 0.7, while the mutation rate is 0.01.

5.5. Convergence

Convergence is reached when no improvement in the objective function have been obtained in 100 consecutive generations. However, as the genetic algorithm manipulates the input variables, the best so far solution found is always saved. This guarantees that the output of the algorithm is the best solution it finds regardless of the final population it reached.

5.6. Improved Features

The proposed GA, as stated earlier, is based on the technique described by Jones and Sthanmer [Jones, Sthamer et al, 1996]. However, two major improvements are done. First, the technique was modified to obtain path coverage and not branch coverage as it was devised for. Secondly, our improved algorithm made use of Huang approach for converting inequalities to equalities. This added feature made it possible to work with complex conditions forming the path predicate.

Chapter 6

Experimental Results

In this chapter, we present a number of test cases and their results to explain and verify the use of the GA and SA algorithms for path testing.

In our experiment, the algorithnms were developped under Turbo C, and they were run on a Pentium 166 Mhz with 32 MB of memory and 2 GB of hard disk.

We would like to note that, through all the experiments the population size for the GA is equal to 50.

6.1. Test Case T1

This test case was taken from the paper published by Jones and Sthamer [Jones, Sthamer et al, 1996], and it consists only of one predicate. The predicate is $X_1^2 - X_2 + 5 = 0$.

This test case reads two variables and tries to evaluate the expression listed above.

Since there is only one predicate, the algorithm should select test cases that should execute two paths :

- Path 1 : $X_1^2 - X_2 + 5 = 0$
- Path 2 : $X_1^2 - X_2 + 5 \,!= 0$

Path 1	**G.A.**	**S.A.**
Fitness	0	0
Values	$X_1 = 3$; $X_2 = 14$	$X_1 = 8$; $X_2 = 69$
Path Coverage	Yes	Yes
Time	3.4 s	2.8 s

Table 1. Results of path 1 for test case T1

Path 2	**G.A.**	**S.A.**
Fitness	0	0
Values	$X_1 = 4$; $X_2 = 103$	$X_1 = 5$; $X_2 = 78$
Path Coverage	Yes	Yes
Time	3.6 s	2.8 s

Table 2. Results of path 2 for test case T1

6.2. Test Case T2

This test case was taken from a paper published by Agrawal for incremental regression testing [Agrawal, 1993]. It is a program that aims at classifying the type of a triangle. This program reads the lengths of the three sides of a triangle, classifies the triangle as one of a scalene, isosceles, right or an equilateral triangle.

The flow graph of this program is presented in Figure 1.

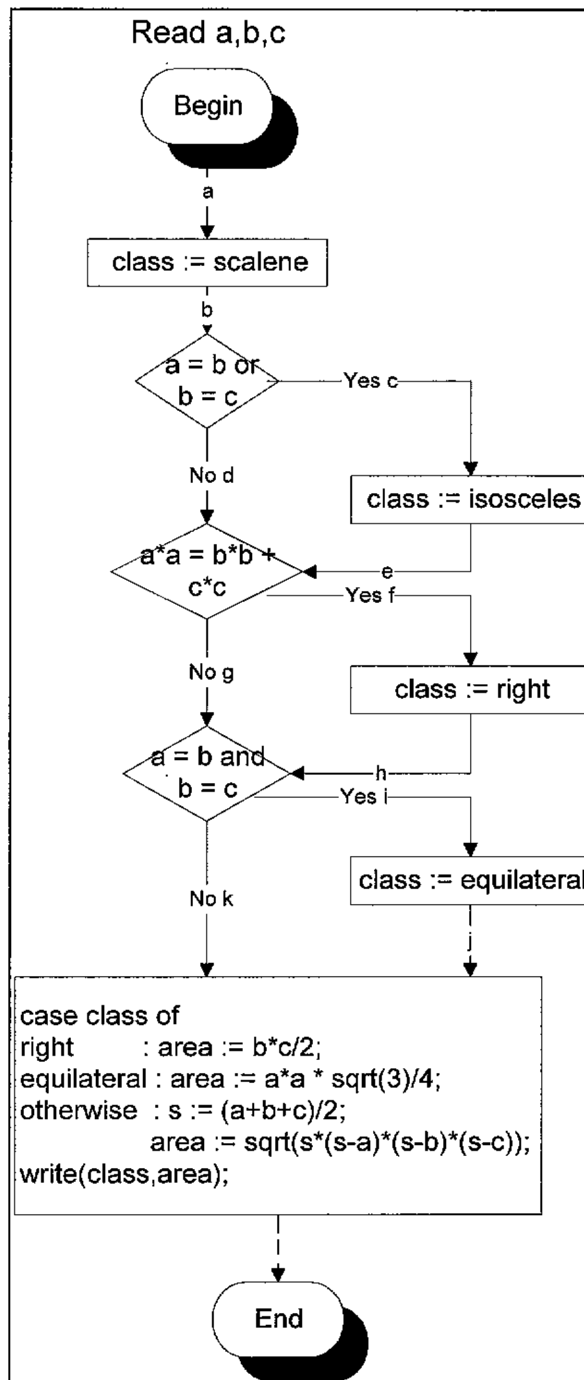


Figure 1. Flow Graph of Test Case T2

This program consists of many paths, six of these paths where selected to be tested by the algorithms.

- Path 1 (isosceles triangles) : abcegkl
- Path 2 (scalene triangles) : abdgkl
- Path 3 (rectangle triangles) : abdfhkl
- Path 4 (equilateral triangles) : abdgijl
- Path 5 : abcefhkl
- Path 6 : abcegijl

Path 1	G.A.	S.A.
Fitness	14	47
Values	a = 6 ; b = 19 ; c = 19	a = 11 ; b = 11 ; c = 10
Path Coverage	Yes	Yes
Time	9 s	10.6 s

Table 3. Results of path 1 for test case T2

Path 2	G.A.	S.A.
Fitness	217	66
Values	a = 104 ; b = 53 ; c = 90	a = 6 ; b = 8 ; c = 10
Path Coverage	Yes	Yes
Time	9.5 s	10.2 s

Table 4. Results of path 2 for test case T2

Path 3	G.A.	S.A.
Fitness	33	213
Values	a = 59 ; b = 33 ; c = 37	a = 124 ; b = 40 ; c = 118
Path Coverage	No	No
Time	9.4 s	10.4 s

Table 5. Results of path 3 for test case T2

Path 4	G.A.	S.A.
Fitness	129	129
Values	a = 9 ; b = 9 ; c = 9	a = 53 ; b = 52 ; c = 13
Path Coverage	Yes	No
Time	9.4 s	9 s

Table 6. Results of path 4 for test case T2

Path 5	G.A.	S.A.
Fitness	14	305
Values	a = 5 ; b = 5 ; c = 2	a = 76 ; b = 76 ; c = 90
Path Coverage	No	No
Time	8.8 s	10.7 s

Table 7. Results of path 5 for test case T2

Path 6	G.A.	S.A.
Fitness	256	86
Values	a = 125 ; b = 127 ; c = 126	a = 62 ; b = 62 ; c = 2
Path Coverage	No	No
Time	9.6 s	10.2 s

Table 8. Results of path 6 for test case T2

6.3. Test Case T3

This test case is taken from the paper published by Huang for path testing [Huang et al, 1975], and it is based on the program whose flow graph is presented in Figure 2.

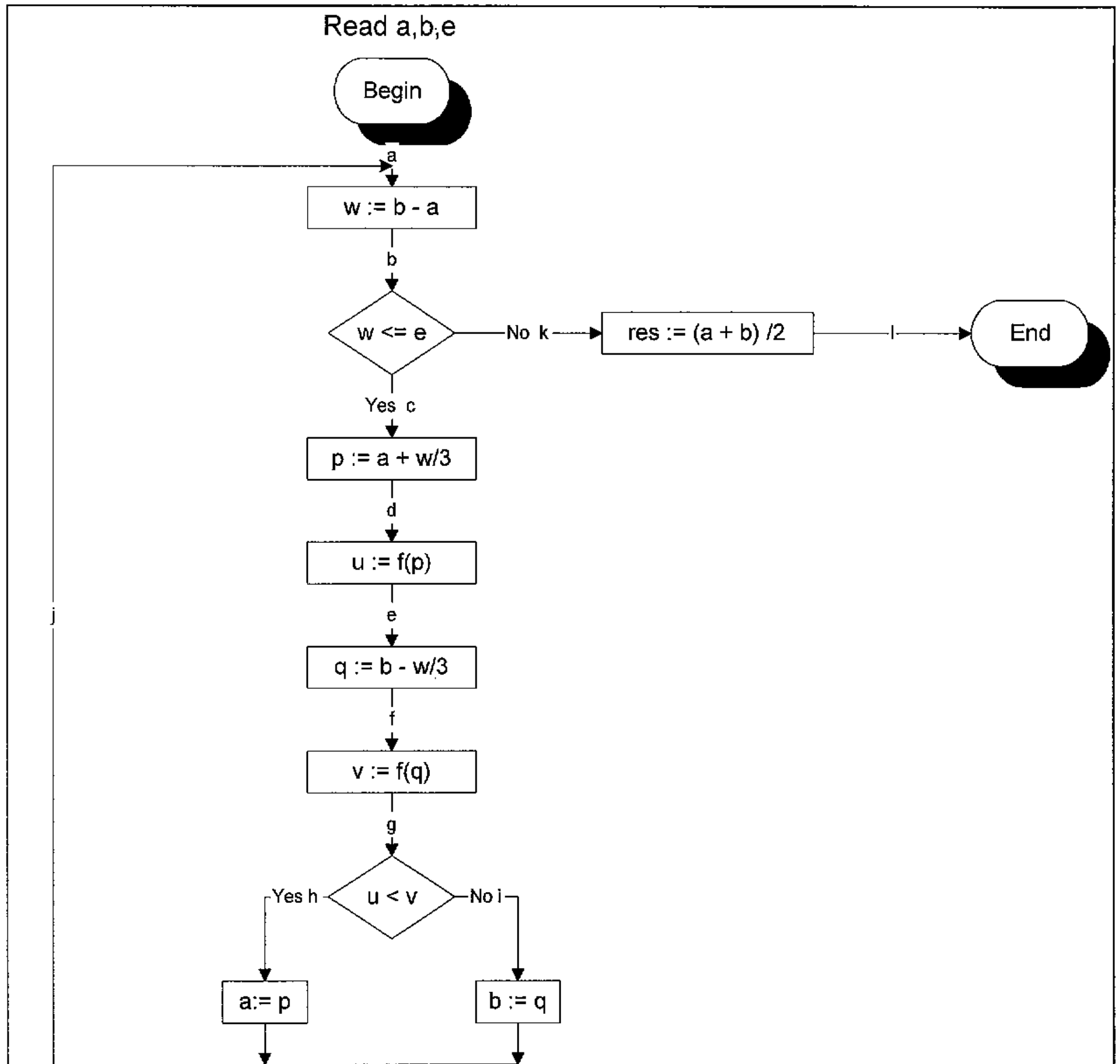


Figure 2. Flow Graph of Test Case T3

This program was tested to cover two of its paths :

- Path 1 : abcdefghjbkl
- Path 2 : abcdefgijbkl

Path 1	G.A.	S.A.
Fitness	0	0
Values	a = 11 ; b = 50 ; e = 38	a = 44 ; b = 112 ; e = 49
Path Coverage	Yes	Yes
Time	5.6 s	4.3 s

Table 9. Results of path 1 for test case T3

Path 2	G.A.	S.A.
Fitness	22	1
Values	a = 49 ; b = 59 ; e = 11	a = 110 ; b = 114 ; e = 4
Path Coverage	No	No
Time	5.6 s	4 s

Table 10. Results of path 2 for test case T3

6.4. Test Case T4

This test case is based on the program whose flow graph is given in Figure 3

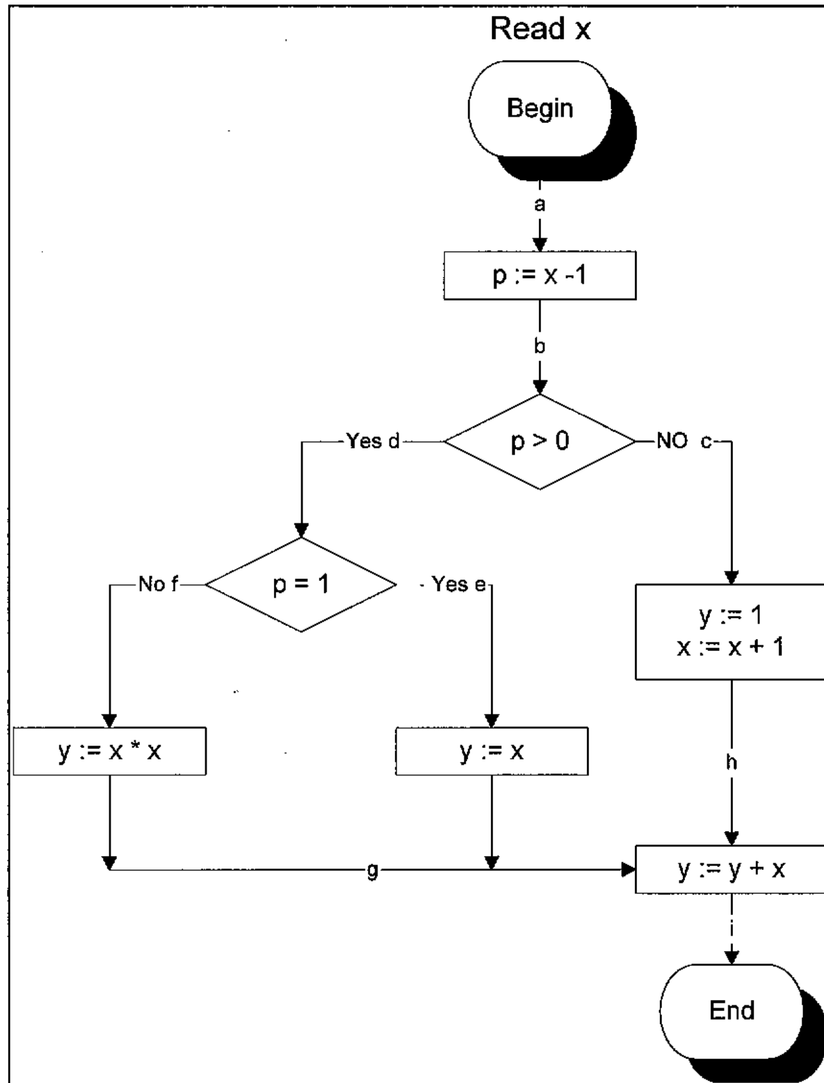


Figure 3. Flow Graph of Test Case T4

This program was tested to cover all of its paths :

- Path 1 : abchi
- Path 2 : abdegi
- Path 3 : abdfgi

Path 1	**G.A.**	**S.A.**
Fitness	0	0
Values	x = 1	x = 1
Path Coverage	Yes	Yes
Time	3.6 s	1.6 s

Table 11. Results of path 1 for test case T4

Path 2	**G.A.**	**S.A.**
Fitness	0	100
Values	x = 2	x = 6
Path Coverage	Yes	No
Time	4 s	2.2 s

Table 12. Results of path 2 for test case T4

Path 3	G.A.	S.A.
Fitness	0	158
Values	x = 8	x = 6
Path Coverage	Yes	Yes
Time	4 s	2.5 s

Table 13. Results of path 3 for test case T4

6.5. Discussion of results

The experimental results show that the GA was capable of finding the required values for the input variables to traverse all the selected paths for both T1 and T4 test cases. However, for test case T2 the GA was successful in finding input values only for path 1, 2 and 4; while it was close in reaching the desired input for the rest of the paths. For test case T3, path 1 was covered by the input produced by the GA; however the input for path 2 failed to lead to the traversal of that path.

On the other side, the experimental results show that the SA was successful only in finding the needed input for test case T1, so that all its paths are traversed. On test case T4 he failed only to find input for path 2, and was capable of finding the desired input for the rest of the paths. However, for test case T2 he was successful only in finding input for two paths : path 1 and 2.

With respect to test case T3, its performance was similar to that of the GA , path 1 was covered while path 2 was not.

The experiments show that the proposed algorithms are good compared to known techniques, for two reasons. First, both algorithms are a fully automated path testing approach, while most of the known techniques require user intervention, at some stage, in order to produce relatively acceptable results. Secondly, even when these algorithms fail to lead to the traversal of the desired path, the results produced can help the user to identify the cause of the failure. Consequently, small modification can be done to obtain the needed results.

In Comparing the two techniques, we can state that both techniques are good for path coverage. However, in some test cases the GA was better than the SA like in test case T4, and in test case T2, since more paths were covered by the GA than the SA. But, we would like to note that, when both techniques fail in producing the desired input, both were yielding values around the boundaries of the needed results. In some test cases the SA was closer to the required results than the GA like in Test case T2 path 3 and test case T3 path 2; while in test case T2 path 5 and 6 the GA was better.

We would like to note also, that it is not necessary for the fitness value of the produced input to be equal to 0 to be sure that the path is covered, while the reciprocal is always true. To explain this, we can state that both algorithms use additional variables in order to convert inequalities to equalities (Huang approache), and these additional variables may sometimes prohibit the fitness value from reaching 0, but they do not prohibit the path from being traversed.

Chapter 7

Conclusion and Further Work

In this research we were capable to show that the simulated annealing algorithm is a good technique for solving the path coverage problem in both time and solution quality. We were also capable to show that our improved version of the genetic algorithm was also a good approach for path testing. Our experimental results show that both techniques are useful for path testing and have comparable behavior and performance.

It is essential to note that both approaches are excellent to use when the tester wants to combine accurate and boundary value analysis in path testing. This is achievable since if both algorithms fail to lead to the desired input they always produce boundary values where the likelihood of revealing an error is higher. More clearly stated, identifying test cases near to a subdomain boundary as defined by a predicate switching between true and false, are optimal because they reveal common errors in the predicate such as the use of $>$ instead of $\geq$.

Based on the work presented in this thesis, further research tasks can be pursued. First, the SA algorithm can be improved by finding a technique that, while the algorithm is in process, will improve the configuration based on previous results obtained. This technique may allow the SA to climb faster to the desired values. Secondly, the experiments need to be done for large size modules. This needs tools to generate the control-flow graph for the given module, and the path predicate for all paths that need testing.

References

H. Agrawal, J. Horgan, E. Krauser and S. London, 'Incremental Regression Testing', *IEEE Trans.*, 1993.

B. Beizer, 'Software Testing Techniques 2ed', New York, York. Van Nostranf Reinhold, 1990, pp. 145- 172.

B. Beizer, *SoftwareTesting Techniques*, Van Nostrand Reinhold, New York, 1983.

L. Clarke, J. Hassel, D. Richardson, 'A Close look at Domain Testing', *IEEE Trans.*,1982.

J. Duran and S. Ntafos, 'An Evaluation of Random Testing', Transaction on Software Engineering, July 1984, vol. SE-10, no.4, pp. 438-443.

B. Goodenough, L. McGowam, 'A Survey of Program Testing Issues', *Research Direction in Software Technology*, MIT Press, Cambridge, 1979

G. Hamlet, R. Taylor, 'Partition testing does not inspire Confidence', *IEEE Trans. Softw. Eng.*, pp. 1402 - 1411, DEC 1990

B. Hetzel, 'Software Testing : Some Troubling Issues and Opportunities', Presented to the British Computer Society Special Interest Group in Software Testing , 1991.

H. Holland, 'Adaptation in Natural and Artificial Systems', University of Michigan press, Ann arbor, 1975

J. Huang, 'An Approach to Program Testing ', University of Houston, 1975.

D. Ince, S. Hekmatpour, 'An Empirical Evaluation of Random Testing', Cpmput. J, pp. 380, 1986.

B. Jeng, 'Integrating Data Flow and Domain Testing', Asia-pacific Software Engineering Conference, Tokyo, Japan, 1994, PP 123- 135.

B. Jones, H. Sthamer, D. Eyres, 'Automatic Structural testing using Genetic Algorithms', *Software Engineering journal*, 1996, pp. 299-306.

B. Korel, 'Automated Software Test Generation', *IEEE Trans. Softw. Eng.*, pp. 870-879, 1990.

J. MaCabe, 'Structured Testing', US Government Prating Office, Washington, D.C, 1982.

J. Marciniak, B. Curts et al, 'Encyclopedia of Software Engineering', New York, John wiley and Sons inc., 1994, pp.1330- 1358.

S. Rapps and E Weyuker, 'Selecting Software Test Data Using Data Flow Information', *IEEE Trans. Softw. Eng.*, pp. 367-375, April 1985.

A. Rutenbar, 'Simulated Annealing Algorithms : An overview', *IEEE Trans.*,1989.

L. White and E. Cohen, ' A Domain Strategy for Computer Program Testing', Transaction on Software Engineering, 1982, pp. 231-237.

W. Howden, 'A functional Approach to Program Testing and Analysis', Transaction on Software Engineering, October 1986, vol. SE-12, no.10, pp. 997-1005.

E. Weyker and T. Ostrand, 'Theories of Program Testing and Application of Revealing Subdomains', Transaction on Software Engineering, May 1980, pp. 236-246