Program-Operators to Improve Test Data Generation Search

MOHAMMAD ALSHRAIDEH
The University of Jordan
Department of Computer Science
Amman 11942
JORDAN
mshridah@ju.edu.jo

Mohammad Qatawneh
Department of Computer Science
Amman 11942
JORDAN
Mohd.qat@ju.edu.jo

Wesam AlMobaiden
Department of Computer Science
Amman 11942
JORDAN
wesmoba@ju.edu.jo

Azzam Sleit
Department of Computer Science
Amman 11942
JORDAN
azzam.sleit@ju.edu.jo

Abstract: There has recently been a great deal of interest in search based test data generation, with many local and global search algorithms being proposed. In this paper, the program operations, in the form of the program-specific operations used to increase the performance in the generation of test data. The efficacy and performance of the proposed testing approach is assessed and validated using a variety of sample programs, and the empirical investigation is shown to give more than eightfold increase in performance.

Key–Words: Genetic algorithms, program-Search operators, cost function, branch-coverage, software testing, dynamic testing, search-based.

1 Introduction

Test data generation in white-box testing (source-code based testing) is a process of finding program input on which a selected element (e.g. a not yet covered statement) is executed. Finding such input test data manually can be very labour intensive and expensive.

The goal of automatic test data generation in unit testing is to generate test data that can satisfy a given test coverage criterion. A common criterion for unit testing is branch coverage, i.e. the test set should execute every branch in the program unit under test. Branch coverage is also a common criteria for assessing research in automatic test data generation and is the criterion adopted for the empirical investigation reported in this paper. The techniques used in achieving branch coverage require the satisfaction of predicate expressions generated from the program under test and as such they can be used as the basis of methods that generate test data for a variety of coverage criteria.

A number of different automatic software test data generation methods have been investigated [3]. These methods may be placed into one of two broad categories known as static methods and dynamic methods. Static methods aim to analyse the static structure of the program under test in order to compute suitable test cases. Static methods exploit control and data-flow information and may use symbolic execution [16], [17], [4] but the program under test is not executed.

Dynamic methods aim to exploit information gained by execution of the program under test. The most basic dynamic method is random test data generation [11]. In this method, test data is generated randomly. Each test case is then executed and either retained or discarded according to whether it executes branches not executed by any other so far retained test case. Unfortunately, the likelihood that a test, generated randomly, will execute a difficult to reach branch is very low. As an example, consider the problem of generating an input to execute the target branch of the program fragment shown below.

```
...
if (s == "UNIVERSITY") {
   //TARGET
}
```

The probability that a randomly generated input will set the variable `s` to be equal to the string `UNIVERSITY` is likely to be very small. In general,

random test data generation performs poorly and is generally considered to be ineffective at covering all branches in realistic programs [9].

Guided or heuristic search is a more effective test data generation approach. Various heuristic search methods have been used to generate test data including gradient descent, tabu search and simulated annealing [27], but a commonly used method is a genetic algorithm [15] [14]. A genetic algorithm, which was used in the work reported here, has the following features:

1. The genetic algorithm maintains not one but a population of many candidate solutions. A candidate solution is an attempt to solve the problem. In the context of test data generation, a candidate is a potential test case.

2. A fitness (or cost) function which evaluates the utility of each candidate in the search for a solution.

3. A probabilistic selection function which selects candidates from the population with the highest probability of selection accorded to the most promising candidates.

4. Search operators, also known as genetic operators, which modify the selected candidates (known as parents) to create new candidates (known as offspring). Two search operators invariably associated with genetic algorithm search are crossover and mutation. The purpose of the crossover operator is to combine elements from two different but promising candidates in the hope of producing offspring that outperform their parents. Mutation is a genetic operator that modifies one or more elements of a selected candidate. The purpose of mutation is to explore the space of candidates that are similar to a given candidate.

Figure 1 shows the basic steps of a genetic algorithm. First the population is initialised, either randomly or with user-defined candidates. The genetic algorithm then iterates through an evaluate-select-produce cycle until either a solution is found or some other stopping criteria applies.

In general, Software vendors typically spend 30 $-$ 70% of their total development budget, i.e. of an organizations software development resources, on testing. Software engineers generally agree that the cost to correct a defect increases, as the time elapsed between error injection and detection increases depending on defect severity and the software testing process

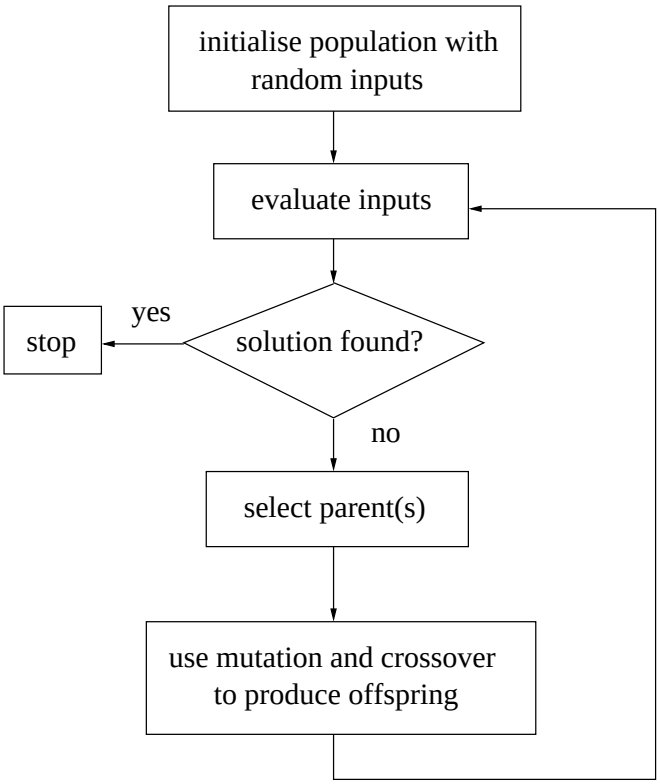

Figure 1: Flowchart of test data generation using a genetic algorithm.

maturity level. In order to achieve high testing productivity, a framework called the Integrated and Optimized Software Testing Process (IOSTP) [18] has been developed. Testing is inefficient for the detection and removal of requirements and design defects. Instead of testing out defects to achieve quality measures, quality should be designed into software. Many technical areas have evolved into engineering fields that can be deployed in IOSTP, such as modelling and simulation (M&S), design of experiments (DOE), software measurement, and the Six Sigma approach to software test process quality assurance and control. In [24] Saurabh introduced a system dynamics model of software development are presented, better understanding testing processes. Motivation for modeling testing processes is also presented along with a an executable model of the unit test phase. It motivates the importance of software cycle time reduction. The objective of the research in [24] is to provide decision makers with a model that will enable the prediction of the impact a set of process improvements will have on their software development cycle time.

In [20] advanced verification methods and testing strategies was presented.

As an example, consider the problem of generating an input to execute the target branch of the pro-

```
void func(int a) {
    if (a == 0)
        //execution required
  }
```

Figure 2: Simple predicate example

gram fragment shown in Fig. 2.

The probability that a randomly generated input will set the variable `a` to be equal to `0` may be very small. In general, random test data generation performs poorly and is generally considered to be ineffective at covering all branches in realistic programs [9].

Genetic algorithms search has been used to search for structural test data. Structural test techniques require the coverage of a certain type of structure in the source code. The fitness or cost functions used by researchers to guide the search reward inputs that are close to executing a desired structure and penalize those that are far away.

In the program of Fig. 2 suppose a test case is required to execute the true branch. If the branch is not executed, many test cases will cause `a == 0` to be false. The value of `abs(a - 0)` increases as `a` becomes far from `0`. A value of `4` has a better objective value than that of `10`, since the objective function is better (4 is more close to 0 than 10). The search is encouraged to search around the value of `4`, possibly encountering further "better" values, for example the values `1` or `2`.

It has become generally accepted that genetic algorithms benefit from the incorporation of domain specific knowledge. Many real-world problems are solved by designing a problem specific representation and corresponding operators that manipulate individuals with methods inspired by the context of the problem. The possibilities for creating more or less sophisticated heuristic operators are endless.

The aim of this work is to investigate the impact of using program-specific search operators and the seed populations in incremental evolution.

The rest of the paper is organised as follows. Section 2 overviews about String equality cost functions. Section 3 introduces the technique Program-dependent Search Operators. Section 4 present Empirical Assessment of String Search Operators and Cost Functions whilst Section 5 generalize the technique for Program-specific Search Operators for Non-string Data Types and Section 6 concludes.

2 Related Works and Background

2.1 Basic Concept

A Control Flow Graph (CFG) of a program is a directed graph $G = (N, E, s, e)$ where N is a set of nodes, E is a set of edges, while s and e are unique entry and exit nodes in the graph. Each node $n \in N$ corresponds to a statement in the program, with each edge $e = (n_i, n_j) \in E$ representing a transfer of control from node n_i to n_j. Nodes corresponding to decision statements (for example an if or while statement) are referred to as branching nodes. The branch is executed when the condition at the branching node is true, which is referred to as the true branch. Conversely, the branch is executed when the condition is false, which is referred to as the false branch. The predicate that determine whether a branch is taken, is referred to as a branch predicate. A path through a CFG is a sequence of nodes $P = (n_1, n_2, ...n_m)$ such that for each i, where $1 \leq i < m, (n_i, n_{i+1}) \in E$. Control dependency [12] is used to describe the reliance of a node's execution on the outcome at previous branching nodes. For a program node i with two exits (for example, an if statement), program node j is control dependent on i if one exit from i always results in j being executed, while the other exit may not result in j being executed.

2.2 Search Based Test Data Generation

Evolutionary algorithms [29] combine characteristics of genetic algorithms and evolution strategies, using simulated evolution as the model of a search method, employing operations inspired by genetics and natural selection. Evolutionary algorithms maintain a population of candidate solutions referred to as individuals. Individuals are iteratively recombined and mutated in order to evolve successive generations of potential solutions. The aim is to generate fitter individuals within subsequent generations (i.e., better candidate solutions). This is performed by a recombination operator, which forms offspring from the components of two parents selected from the current population. The new offspring form part of the new generation of candidate solutions. Mutation performs low probability random changes to solutions, introducing new genetic information into the search. At the end of each generation, each solution is evaluated for its fitness, using a problem-specific fitness function. Using fitness values, the evolutionary search decides whether individuals should survive into the next generation or be discarded. In applying evolutionary algorithms to structural test data generation, the individuals of the search are input vectors. The fitness function is derived from the context of the cur- rent structure of in-

terest in the program. The fitness function is to be minimized by the search: thus, lower numerical values represent fitter input vectors that are closer to executing the target structure. When a zero fitness value has been found, the required test data has also been found. The fitness function is made up of two components the approach level and the branch distance. The approach level measures how close an input vector was to executing the current structure of interest, on the basis of how near the execution path was to reaching it in the programs control flow graph. Central to the metric is the notion of a critical branching node. A critical branching node is simply a branching node with an exit that, if taken, causes the target to be missed. In other words, the set of critical branching nodes is the set of nodes on which the target structure is control dependent (either directly or transitively).

2.3 Branch Cost Functions

A conditional statement (an if-statement or a while-statement) contains a predicate expression and a pair of branches: the true branch, executed when the predicate expression is true, and the false branch, executed otherwise. The so-called multi-way or switch statements are considered to be a sequence of conditional statements. In some programs, the execution of a target branch may require the execution of a sequence of nested branches. The problem of nested branches is not significantly different from the point of view of this paper. The problem decomposes into a sequence of sub-goals, in each of which, a single branch must be executed in the context of a solution for the enclosing branch.

The cost function of $abs(\mathtt{a}-1)$ applied to the Boolean expression `a == 1` of the Flag program illustrates the cost function for the equality operator. Similar cost functions are available for other relational operators as shown below where a and b are numbers and ϵ is the smallest positive constant in the domain (i.e. 1 in the case of integer domains and the smallest number greater than zero, in the particular, real number representation).

Predicate expression	Cost of not satisfying predicate expression
$a \leq b$	$a - b$
$a < b$	$a - b + \epsilon$
$a \geq b$	$b - a$
$a > b$	$b - a + \epsilon$
$a = b$	$abs(a - b)$
$a \neq b$	$\epsilon - abs(a - b)$

2.4 Related Works

Traditionally, the search mechanism has been domain independent, that is to say the crossover and mutation operators have no knowledge of what a good solution would be. However, it has been shown [7] [8] that by using domain dependent operators good, if not better, results may be achieved

The program-specific search operators in software test data generation is already demonstrated by Alshraideh and Bottaci in [2] for string data type, where Program-specific search operators aim to exploit the structure in the region in the program from the input variable to the test goal. The structure of the computation can be used in the search by using the functions available in the program under test as the basis of search operators and constants to seed the search. The examination of the SSCLI code showed that about 65% of string predicate expressions contains a string literal, also in [10], DeMillo and Offutt showed that 58% of clauses are of the form $x\ R\ c$, where x is variable, c is a constant and R is relational expression. So a program may match a string literal with a string input, or a string derived from the input.

```
function f(s:string) {
   if (s.Equals("CHILD"){
         ...
   }
}
```

The first branch of this program is true when `s = "CHILD"`. This suggests a heuristic to guide the search for values for the strings `s`, namely, set `s` to a string literal that appears in the program under test.

In [2] by exploiting the presence of string literals in programs that process string data, a very significant improvement in performance was obtained. The program-dependent string search operators that focus the search in the region of string literals were presented, and in the empirical investigation, the use of these operators was shown to give a fivefold increase in performance.

Julstrom [1] experimented with a hybrid GA for the Rectilinear Steiner Problem that seeded the initial population with organisms that were of a higher initial fitness. Julstroms work has neither been reproduced nor has this method been applied to other problems. The seeded solution was created using the Construct Randomized Greedy Solution() method used by the GRASP algorithm with α = 5 (See the section on GRASP to read about the significance of α). In some cases, solutions obtained from another optimization algorithm are used to seed the initial population [26].

Although this bears the risk of misguiding the optimization process toward local optima, it has been proved that the seeding approach is very powerful in some cases. In Oman's approach [22], the initial population is seeded using case retrieval in order to speed up the GA in finding a solution.

In a study done by Boomsma [5], He investigated whether adaptive operator scheduling (AOS) can provide a solution to his research's problem. Experiments were done on instances of the symmetrical Traveling Salesman Problem (TSP), a well known NP-hard combinatorial problem for which a multitude of operators exist. It can be defined as the search for a minimal Hamiltonian cycle in a complete graph, and can be understood as the problem of visiting *n* cities (and returning to the first), using the path of smallest total distance. The main concern in the original investigation was that a large number of operators might slow down the optimisation process compared to an algorithm using the optimal choice of operators. It was shown that this concern was unfounded: the algorithm using AOS converged as fast as the best combination of mutation and crossover operators with equally good results. The AOS scheme used in these initial experiments (Davis) was however rather cumbersome to implement.

Software test metrics and their ability are discussed to show objective evidence necessary to make process improvements in a development organization. When used properly, test metrics assist in the improvement of the software development process by providing pragmatic, objective evidence of process change initiatives. Lazic et. el. in [19] described several test metrics that can be implemented, a method for creating a practical approach to tracking & interpreting the metrics, and illustrates one organization's use of test metrics to prove the effectiveness of process changes. Also, their work provided the Balanced Productivity Metrics (BPM) strategy and approach in order to design and produce useful project metrics from basic test planning and defect data. Software test metrics is a useful for test managers, which aids in precise estimation of project effort, addresses the interests of metric group, software managers of the software organization who are interested in estimating software test effort and improve both development and testing processes.

3 Program-specific Search Operators

The performance and usefulness of program-specific operators has been demonstrated for strings it seems clear that the technique generalises to other data types.

```
void  f1(int a, int b) {
   int v = 5;
   if (a == 10) {
       v = v + b;
   }
   if (v == 20) {
    //Target executed
    //with a = 10, b = 15
   }
}
```

Figure 3: Alternative internal variable example

```
void InverseSin (double a, double b) {
    double v = 5.0;
    if (a == 25.0) {
        v = v + sin(b);
    }
    if (abs(v - 6.0) <= Double.Epsilon){
        //Target executed
    }
}
```

Figure 4: To execute the target, `b` equal to $\frac{\Pi}{2}$

This is illustrated for the numeric data type in the example in Fig. 3. Although none of the three integer values 5, 10 and 20 that occur in the program are input values that execute the target branch (to execute the target branch `a = 10` and `b = 15`) they do provide reasonable starting points for a guided search. To get the variable `b = 15`, simply inverse the arithmetic operation plus `(v = v + b)` which is (20 = 5 + b) then `b` = 20 - 5.

Numerical types may be converted from integer and double as required by the input domain. Numerical types may be also converted to character data type and vice versa if possible. In general, adding the literal that appears in the program is not straightforward. If the data types of the input parameters are integer but the literal collected from the program is double, the input domain is different from the literal data type: the literal data types are converted to confirm with the input domain data type.

Fig. 4 shows a program in which the first branch is executed when `a = 25`. The required branch is executed only when the first "if statement" is executed and $sin(b)$ = 1. This happens only when `b` = $\frac{\Pi}{2}$. It is easy to execute this branch if the mutation operator sequence of $sin^{-1}(6.0 - 5.0)$ is used to create a value for `b`. The arithmetic operations in this

program are : Plus and sin, the proposed mutation operators to execute the required branch might be Minus and Arcsin.

This has motivated the introduction of additional genetic operators to increase the performance of searching the program under test by analysis and extracting arithmetic operators from the program under test, then reversing these operators to induce the mutation operators.

In general, when any arithmetic operator[1] or trigonometric function[2] occurs in the program under test, this operator or function and its inverse are used as mutation operators (e.g. Plus and Minus, sin and Arcsin). Note that Arcsin and Arccos are used when the parameter is in the range (-1 to 1) only.

The polygon classification program has an array of 6 or 8 of positive real numbers. The length of the array represents the figure shape: 6 means the figure might be Triangle, 8 means the figure might be Square or Rectangle or other Polygon. The first half of the input parameters (3 or 4) represents the angles and the rest represents the side lengths of the figure sides. The goal of the program is to determine the figure, Square, Rectangle, Triangle or other shape and also if the figure is Triangle, to categorize the triangle type. The program consists of 22 branches, all the branch cost functions have a gradient which illustrates the usefulness of program-specific search operators for program where branch coverage can be found by straight forward branch cost distance instrumentation. No branch has a branch distance cost which is locally flat. The program was executed by GAs with and without using program-specific search operators and over an average of 20 trials the number of executions required to find test data to achieve all branch coverage without using program-specific search operators was 43872; the number of executions required to find test data with program-specific search operators was 1542. It is clear that there is a significant improvement in performance by using program-specific search operators.

4 Empirical assessment of program-specific search operators

In this section, the specifications of the experimental environment utilized by this work are presented. These specifications include both the hardware and software modules used in implementing the simulator. More specifically, the hardware specifications that are used in the experiments include a Dual-Core Intel Processor (CPU 2.66 GHz), 2 MB L2 Cache per CPU, and 1 GB RAM. Moreover, the software specifications that are used in the experiments include windows XP. Also, the tested programs that have been used to evaluate our proposed approach are described in this section.

4.1 Experimental parameters

Each of the cost functions and associated search operators were implemented in a prototype test data generation tool. The tool has been constructed by modifying the JScript (JavaScript) language compiler within the SSCLI and can therefore be used to test functions within programs written in the JScript language. The program must include directives to specify any input domain constraints that are to be applied. The program is then parsed and semantic analysis is done. The tool then inserts instrumentation code at each branch in the function. This instrumentation code calculates the cost of each branch predicate whenever it is executed.

The cost of each relational predicate expression was calculated according to the cost functions given in the previous section (Section 2.3). Where branch predicate expressions consist of two or more relational predicates joined by logical connectives, and, or and not, the cost values were combined according to the scheme given in Bottaci [6]. In the case of logical and, for example, the costs of the constituent operands are added whenever they are both false. For nested branches, the costs of the branches in the control dependency condition of the target branch were similarly combined to provide an overall cost value for the candidate input. Unexecuted branches were assigned a high fixed cost.

4.2 Subject Programs

In order to assess the reliability of the new approach introduced in the previous section, an empirical investigation was done. A number of JavaScript test programs were assembled including both open and propriety source, these programs already used in used in test-data generation research [13], [25], [21], [23], such as "Polygon classifier" (polygon), "middle value" (Mid), "Bubble Sort" (Bub), etc. The other programs are synthetic programs containing complex structures. Table 1 shows details of the programs: The first column, Program, gives the name we used to designate the program; the second column, LOC, shows the number of lines of code in the program; the third column, number of branch in the program; the fourth column, Description, provides a description of

[1] Plus, Minus, Multiply, Divide, PostIncrement, PostDecrement, Pow, Sqrt, Modulus and Absolute value

[2] sin, Arcsin, cos, Arccos, tan and Arctan

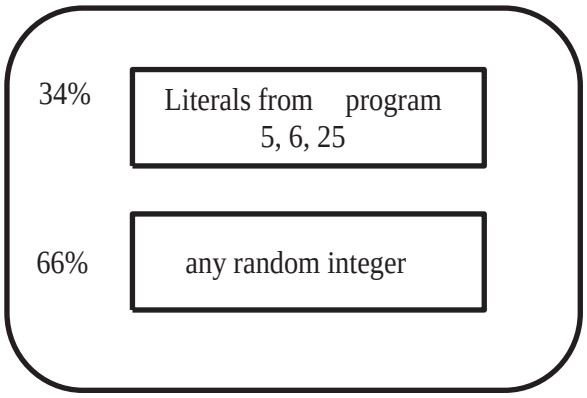

Figure 5: The integer domain

the program, and for each program, an attempt was made to generate inputs to achieve branch coverage. The programs are available from the authors on request.

4.3 Input domains

The domain of the variable is extracted from variables domain and numerical literals appear in the program under test as shown in Fig. 5.

4.4 Genetic algorithm

The search was directed to generate data for one branch at a time. The order in which the branches of the program were targeted was arbitrary except that no nested branch was targeted before the containing branch. This is not, in general, a good strategy since the search will become stuck at an infeasible branch but it is adequate for the experimental purposes of this research given that all the branches in the sample programs are feasible.

A steady-state style genetic algorithm, similar to Genitor [28], was used in this work. The cost function values computed for each candidate input were used to rank candidates within the population in which no duplicate genotypes are allowed. A probabilistic selection function selected parent candidates from the population with a probability based on their rank, the highest ranking having the highest probability. More specifically, for a population of size n, the probability of selection is

$$\frac{2(n - rank + 1)}{n(n-1)}$$

A single tree-structured representation was used, both for candidate inputs (phenotype) and for crossover and mutation (genotype). At the top-level, a candidate is an array of objects in 1-1 correspondence with the parameters of the program under test. Each object may be a primitive value, i.e. a number, character or string, or an array. This representation has the advantage that all candidates have the same structure. Candidates differ only in the lengths of strings and these occur only at the leaf nodes of the structure.

Single point crossover was used. A genetic algorithm has a number of parameters that may be modified to suit a given problem. The size of the population and the frequency with which selected candidates are mutated are two examples. In the context of test data generation, a search algorithm must be able to perform effectively without significant human intervention as such intervention is not cost effective hence no parameter was "tuned" to suit any particular program under test. In the work reported here, a population size of 100 was always used. At each evaluate-select-produce cycle, either mutation or crossover was applied with equal probability. This means that a third of selected candidates were mutated since two candidates are selected for a single application of crossover.

4.5 Prototype Implementation

Fig. 6 shows the system architecture and dataflow of Test Data Generation(TDGen). A coverage table is established to record the branch information of the program under test, and keeps track of whether a branch is tested or not. When the coverage table is initialized with the seed inputs, the test data generator gets the next untested branch from the coverage table and send query requests of the Program Dependency Graph (PDG) analysis with regard to the target branch to the program analyzer, which takes in the source code of the program, generates a system program dependence graph including a PDG for each procedure, accepts query requests from the test data generator. After getting query results from the program analyzer, the test data generator converts the query results to constraints. A genetic algorithm is used to satisfy the constraints and then test cases are generated. If the constraints have temporary variables in them, and the GA needs their values to evaluate fitness functions, the values of the temporary variables can be obtained by augmenting the source code of the tested program to output them. Finally, the program is run with the test cases generated by the test data generator to see if the desired result is reached. If the targeted branch or some other untested branches are traversed, the coverage table will be updated, and then a new target will be selected from the coverage table for the next cycle of test data generation.

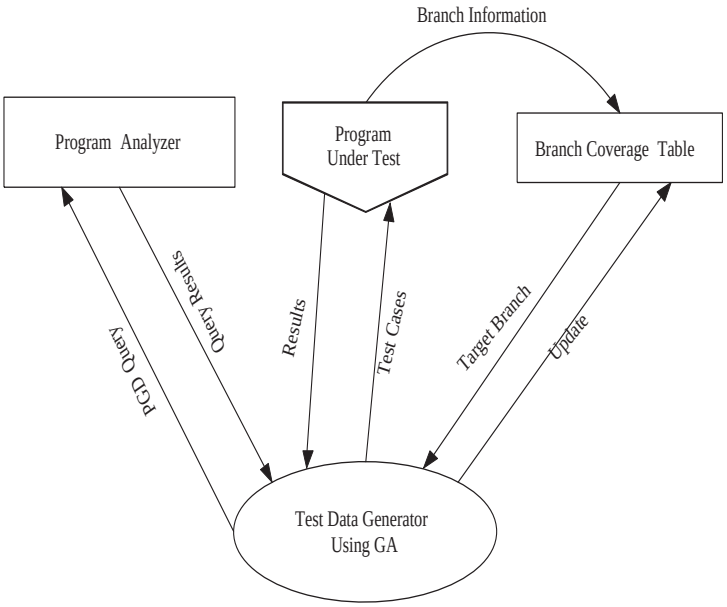

Figure 6: System architecture and dataflow diagram.

5 Results

By using the program-specific search operators in the program of Fig. 7, although none of the two integer values 0 and 1 that occur in the program are input values that execute the target branch (to execute the target branch a must be equal to $\frac{\Pi}{2}$), they do provide reasonable starting points for a guided search. In particular, to set the variable a = $\frac{\Pi}{2}$, it is possible to invert the trigonometric function $sin(a)$ with parameter value equal to 1 which is $sin^{-1}(1)$ then a = $\frac{\Pi}{2}$. Fig. 8 shows different paths to a solution to test data generation problem shown in Fig. 7.

In this research, we try to measure how effective is the program-specific search operators in reducing the number of generations?
We applied the studies procedure to measure the number of generations that are required for generating the test cases, and recorded the number of generations for each test requirement.

Table 2 shows the number of generations required for satisfying the branch coverage with and without using program-specific search operators (average over 20 trials). For example, without program-specific operators needs 7350 generation to satisfy branch coverage of programs Mid and only 282 generation to satisfy branch coverage of the same program when program-specific search operators are used. On the other hand the number of generation needs to satisfy branch coverage without program-specific is 3421 for program $DateDiff$, while the same program needs 2943 to satisfy the same program coverage with program-specific search operators, in this program it

```
void  FlagAvoid(double a){
   int x = 0;
   double y = 0.0;

        y = Math.sin (a);
         //y in  [-1, 1]
        y = Math.abs(y);
         //y in  [0, 1]
        x = Math.floor(y);
         // x = 0 or 1 only

   if (x == 1){
        //EXECUTED ONLY WHEN ALL
        // VALUES OF a EQUAL TO 90.
        //target executed
   }
}
```

Figure 7: A difficult to execute branch in a program FlagAvoid

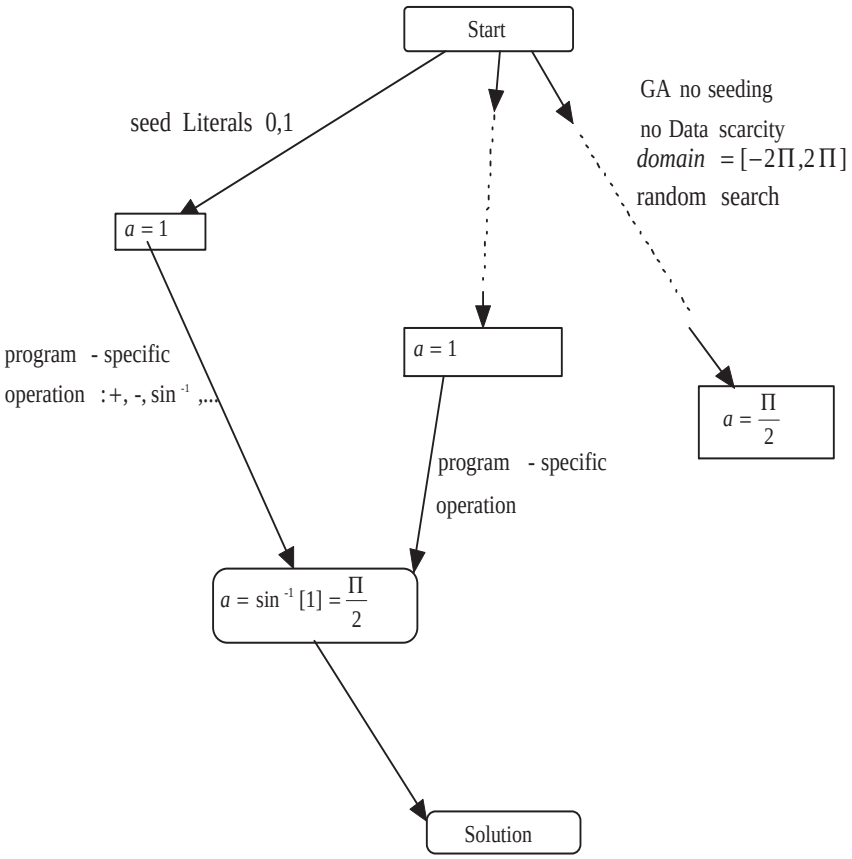

Figure 8: The different paths to a solution to the test data generation problem for program shown in Fig. 7.

seems that using new approach is not effective and this is return to nature of the program (the literal and mathematica operation inside the program).

The results of these empirical studies show that, for the subjects we studied, the number of generations to cover the test requirements by using program-specific search operators is eightfold increase performance than without using program-specific search operators.

The usefulness of using program data constants by itself without program functions and operators has been investigated here. Similarly we have investigated program operators and functions without data constants. The Results are shown in Table 3. There seems to be little advantage in using each of these by itself.

Another study we concern is to answer the following research question:
How effective is using program-specific search operators in reducing search time?
We applied the studies procedure to measure the search time for generating the test suite, recorded the start and end times of the execution, and calculated the search time
search time = end time - start time.

Table 4 shows the search time for generating the test suite to satisfy the branch coverage criterion (average over 20 trials). For example, without using program-specific search operators needs 3352 seconds to satisfy branch coverage for program $Find$, but it needs only 115 seconds by using program-specific search operators.

The results of the studies show that, for the subjects we studied, the branch coverage satisfy by using program-specific search operators takes less time than without using program-specific search operators.

6 Conclusion

This paper presents program-dependent for numerical data type search operators that focus the search in the region of such numerical values. Empirical investigation of the use of program-specific search operators with numerical data type is shown to give more than eightfold increase in performance. Also, experiments have been done to investigate using program data constants by itself without program functions and using program operators and functions without data constants. The Results shown that little advantage in using each of these by itself.

References:

[1] Julstrom B. A., *Seeding the population: improved performance in a genetic algorithm for the rectilinear steiner problem.*, Proceedings of the 1994 ACM symposium on Applied computing, March 1994, pp. 222–226.

[2] M. Alshraideh and L. Bottaci, *Automatic software test data generation for string data using heuristic search with domain specific search operators*, Software Testing, Verification and Reliability. **16** (2006), no. 3, 175–203.

[3] M. Alshraideh, L. Bottaci, and B. A. Mahafzah, *Using program data-state scarcity to guide automatic test data generation*, Software Quality Journal **18** (2010), no. 1, 109–144.

[4] B. Beizer, *Software testing techniques*, 2nd ed., van Nostrand Rheinhold, New York, 1990.

[5] W. Boomsma, *Using adaptive operator scheduling on problem domains with an operator manifold: applications to the travelling salesman problem*, In: Proceedings of the 2003 Congress on Evolutionary Computation (CEC2003) **2** (2003), 12741279.

[6] L. Bottaci, *Predicate expression cost functions to guide evolutionary search for test data*, Genetic and Evolutionary Computation Conference (GECCO 2003), July 2003, pp. 2455–2464.

[7] R. Bruns, *Knowledge-augmented genetic algorithm for production scheduling*, Workshop on Knowledge based Production Planning, Scheduling and Control,IJCAI 93. (1993).

[8] E. K. Burke, D. G. Elliman, and R. F. Weare, *A genetic algorithm for university timetabling*, East-West Conference on Computer Technologies in Education, Crimea, Ukraine. (1994), 35–40.

[9] P. Coward, *Symbolic execution and testing*, Information and Software Technique **33** (1991), no. 1, 229–239.

[10] R. DeMillo and A. Offutt, *experimental results of automatically generated adequate test sets*, in proc. 6th Ann. Pacific Northwest Software Quality Conf. (1988), 209–232.

[11] J. Duran and S. Ntafos, *An evaluation of random testing*, IEEE Transactions on Software Engineering **10** (1984), no. 4, 438–443.

[12] J. Ferrante, K. Ottenstein, and J. Warren, *The program dependency graph and its use in optimization*, ACM Transactions on Programming Languages and Systems **9** (1987), no. 1, 319–349.

[13] M. R. Girgis, *utomatic test data generation for data flow testing using a genetic algorithm*, Journal of Universal computer Science **11** (2005), no. 5, 898–915.

[14] D. E. Goldberg, *Genetic algorithms in search optimization and machine learning*, Addison Wesley, 1989.

[15] J. H. Holland, *Adaptation in natural and artificial systems*, University of Michigan Press, 1975.

[16] J. King, *A new approach to program testing*, In Proceedings of the International Conference on Reliable Software (1975), 228–233.

[17] J. King, *Symbolic execution and program testing*, Communications of the ACM **19** (1976), no. 7, 385–394.

[18] L. Lazic, , and N. Mastorakis, *A framework of integrated and optimized software testing process*, **2** (2003), no. 1, 15–23.

[19] L. Lazic, *Cost effective software test metrics*, WSEAS Transactions on Computers **7** (2008), no. 6, 599–619.

[20] T. lertphumpanya and T. Senivongse, *A basis path testing framework for ws-bpel composite services*, 2008, pp. 483–496.

[21] G. McGraw, C. Michael, and M. Schatz, *Generating software test data by evolution*, IEEE Transactions on Software Engineering **12** (2001), no. 8, 1085–1110.

[22] S. Oman and P. Cunningham, *Using case retrieval to seed genetic algorithms.*, Int J Comput Intell Appl **1** (2001), no. 1, 7182.

[23] R. Pargas, M. Harrold, and R. Peck, *Test-data generation using genetic algorithms*, Software Testing, Verification and Reliability **9** (1999), no. 4, 263–282.

[24] k. Saurabh, *Software development and testing: A system dynamics simulation and modeling approach*, The 9th WSEAS International Conference on SOFTWARE ENGINEERING, PARALLEL and DISTRIBUTED SYSTEMS (SEPADS '10) (University of Cambridge, UK), February 20-25 2010, pp. 67–72.

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

[26] R. Thomsen, G. Fogel, and T. Krink, *A clustal alignment improver using evolutionary algorithms.*, Proceedings of the 2002 Congress on Evolutionary ComputationCEC02. (2002), 30914.

[27] N. Tracey, J. Clark, and K. Mander, *Automated program flaw finding using simulated annealing*, Software Engineering Notes **23** (1998), no. 2, 73–81.

[28] D. Whitley, *The genitor algorithm and selective pressure: Why rank-based allocation of reproductive trials is best*, Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89) (1989), 116–121.

[29] D. Whitley, *An overview of evolutionary algorithms :practical issues and common pitfalls*, Information and Software Technology **43** (2001), 817–831.

Table 1: The JScript functions used for empirical investigation

Program Name	Lines of code	Number of branchs	Description of program
Bub	32	4	Given an array of integers, the program bubble sorts the array.
Find	66	5	Given array A[], and index F, the program places all elements less than or equal to A[F] to the left of A[F], and all elements that are greater than or equal to A[F]to the right of A[F].
Mid	21	5	Given three integers, the program determines the middle value.
Bisect	36	3	Given an epsilon and X, the program computes sqrt(X) within epsilon using bisection method.
Fourballs	82	7	Given four integers representing the weights of balls, the program determines the weights of the balls relative to each other.
MM	61	7	Given an array of numbers,MM is a program to find the minimum and maximum numbers within the array.
DateDiff	46	6	The program determines the number of days between two dates.
Polygon	163	22	Described above in Section 3.

Program Name	Number of execution without using Program-specific	Number of execution using Program-specific
Bub	2600	81
Find	3352	115
Mid	7350	282
Bisect	1622	77
Fourballs	6892	407
MM	1289	75
DateDiff	3421	2943
Polygon-Classification	43872	1542

Table 2: A comparison of the number of executions of the program under test required to find test data to achieve branch coverage (average over 20 trials) with and without program-specific search operators.

Program Name	Number of execution using Literal seeding and program-search operators	Number of execution using Literal seeding	Number of execution using program-search operators
Bub	81	81	1870
Find	115	1118	2789
Mid	282	371	3255
Bisect	77	155	233
Fourballs	407	2156	4677
MM	75	75	1927
DateDiff	2943	3390	5899
Polygon-Classification	1542	20677	38823

Table 3: The number of executions of the program under test required to find test data to achieve branch coverage (average over 20 trials) with program-specific search operators.

Program Name	Time (S) without using Program-specific	Time(S) using Program-specific
Bub	261	21
Find	322	27
Mid	601	54
Bisect	185	38
Fourballs	498	182
MM	134	45
DateDiff	389	325
Polygon-Classification	402	62

Table 4: A search time in Seconds to achieve branch coverage (average over 20 trials) with and without using program-specific search operators.