Optimizing Testing Efficiency with Error-Prone Path Identification and Genetic Algorithms

James R. Birt Renate Sitte
Griffith University, School of Information Technology
Gold Coast Campus PBM 50 Gold Coast Mail Centre,
Gold Coast 9726, Queensland, Australia
j.birt@griffith.edu.au r.sitte@griffith.edu.au

Abstract

This paper presents a method for optimizing software testing efficiency by identifying the most error prone path clusters in a program. We do this by developing variable length Genetic Algorithms that optimize and select the software path clusters which are weighted with sources of error indexes. Although various methods have been applied to detecting and reducing errors in a whole system, there is little research into partitioning a system into smaller error prone domains for testing. Exhaustive software testing is rarely possible because it becomes intractable for even medium sized software. Typically only parts of a program can be tested, but these parts are not necessarily the most error prone. Therefore, we are developing a more selective approach to testing by focusing on those parts that are most likely to contain faults, so that the most error prone paths can be tested first. By identifying the most error prone paths, the testing efficiency can be increased.

Keywords – Genetic Algorithms, optimization, testing efficiency, software reliability

1. Introduction

The purpose of this paper is to use Genetic Algorithms to study the optimization of error proneness in software, with the aim to increase testing efficiency.

The past 30 years have seen a huge growth in the size, complexity and criticality of software code development. Consequently, software reliability and its associated costs for achieving better reliability have greatly increased. Several methods for measuring software reliability are commonly used. They include software reliability models [1], automated oracles [2] and fault detection [3]. Malaiya et al. [4], proposed a link between the levels of testing coverage (that is, the expectation to detect as many errors as possible with a test case) and the reliability of software.

Testing coverage techniques have one major drawback which is they look at the entire program. Even with reduction methods there can be many test cases in a large program and it can become intractable for testing [5].

Testing comes at a high price and typically requires more than half of the project resources to produce a working program [6]. However, a working program does not always guarantee a defect free program. Depending on the criticality of the program, different percentages of testing coverage are required. Therefore a more effective way to approach coverage and testing in general is to focus on the paths most likely to reveal faults [7] or the most error prone paths [8].

To overcome this weakness, we propose a method to attribute weights to the paths using a framework for assessing quantitatively the error proneness of software modules proposed by Sitte [8]. It achieves an assessment of error proneness by analyzing the potential sources of errors (SOE) in a software construct, that is, the code instructions in software modules. The advantage of using our technique is that it can be automated and optimized. By representing the software as a flow graph[1] and subsequently a SOE-weighted connectivity sparse matrix, a search algorithm can be applied to the weighed environment to detect those paths that are the most error prone, or those paths that contribute most to the overall amount of potential errors. It should be noted that our method is not an automated test case generator. The design of the test cases still needs to be addressed. Our method focuses on the potential errors in the code instructions for selecting error prone clusters for later testing and not on the flow of control where more testing should be done. Problems such as potentially infeasible paths

---

[1] Despite their morphological similarity, a flow graph is not a control flow diagram.

associated with data-flow analysis are examined in the later stages.

Various searching algorithms have been developed for different purposes and finding a path in a directed graph is a combinatorial problem that can quickly get out of hand even for medium sized software. Therefore, we have to optimize our search, and we do this with variable length Genetic Algorithms (GA).

GA were developed and formalized by Holland [9]. They were further developed and shown to have wide applicability by Goldberg et al. [10], [11]. More recently GA have been used for generating test data and other software engineering applications as shown by Jones, Sthamer and Eyres [12]. Other methods based on local search techniques such as greedy approaches or simulated annealing [13] could possibly be applied. However, due to the lack of a global sampling capability, local search methods run greater risk of being trapped in local minima. In our case this would be unsuitable in our aim of finding the most error prone paths and given the nature of the search space with many local minima. Therefore, the motivation for choosing the variable length GA approach was based on its heuristics for optimizing large and complex search environments which is indicative of software code. Also, the paths in a software program are not all the same length, therefore variable length GA were applied rather than traditional fixed length GA.

The implications and benefits of determining the most error prone paths are many. Its main purpose is to increase testing efficiency by focusing on the most error prone parts first. This in turn aids in refining the effort and cost estimation that will be required in the testing phase.

In this paper, we present the results of our research into the application of the variable GA search approach, to identify the most error prone paths in a software construct. The paper is structured in the following way: the next section provides an overview of the sources of errors framework, which are the groundwork for quantifying error prone paths. Section 3 deals with our experiment for selecting the error prone paths and the details of the GA implementation. This is followed by the results and discussions in section 4 and finally the conclusions in section 5.

2. Quantifying error proneness

This section explains briefly the SOE framework as proposed by Sitte [8] for calculating the error proneness in software modules and how it collaborates with the GA.

There are several problems with the current methods of testing coverage; they are (i) testing covers only a portion of the software for the level of reliability required; (ii) exponential time to test or testing all paths and (iii) cost effectiveness, that is, the process might not justify the time taken to test. At first glance the first mentioned disadvantage does not appear to be a problem. However, the main issue is, how do we know that what we are testing is the best portion, that is, the most error prone? If we have a coverage level of 80%, does this really mean the most error prone 80% has been covered or is it just any 80% overall (with the most error prone possibly remaining in the untested 20%)?

The answer is that we do not know, unless we have quantitative information about the contents of errors. We do know that some software is easier to write than others, and is less error prone, while more difficult and complex software is more prone to error. A range of complexity measures have been designed, but they do not provide a quantitative answer that translates on which parts of the software the testing efforts should be concentrated. To be able to find the most error prone portions in a quantitative way, it is necessary to assess the chances that a programmer can introduce an error in an instruction. This can be done by applying a simple analysis. Figure 1 illustrates this with an example of a simple loop.

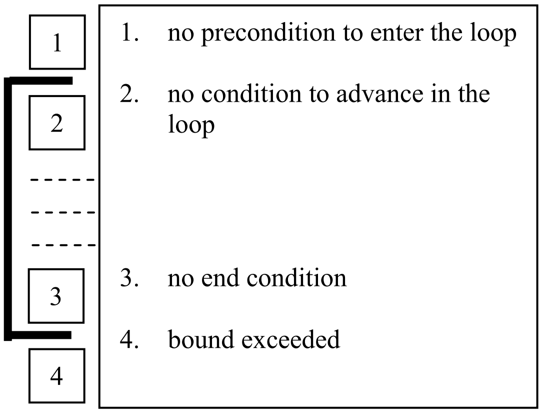

1
2
3
4
1. no precondition to enter the loop
2. no condition to advance in the loop
3. no end condition
4. bound exceeded

Figure 1. SOE analysis of simple loop [8]

This construct has at least four SOE (if those were the only possible sources of error). For simplicity an equal probability for each error is given. All other programming constructs can be analyzed in a similar way, yielding SOE values that are specific to an instruction [8]. The method provides the framework for SOE quantification; it is simple, but powerful and can quickly be obtained from the code in an automated way. While this method provides the minimum theoretical SOE, additional information from test design strategies, risk analysis and historical data (if available) can be used to calibrate and fine tune the model by adding weights to the theoretical SOE values.

By summing the SOE values (weighted or theoretical) for each instruction of a software module, it is possible to quantify the SOE for the different paths that the software could step through in its execution. The software structure is then represented as a directed, weighted graph whose heaviest paths can be identified. We do this by applying the GA. Test cases can then be designed to give priority in testing to the SOE-heavier paths with maximum coverage. The testing priority of paths can be chosen strategically from the most error prone paths first to the less error prone ones to whatever required extent e.g. as a percentage, and to the satisfaction of the reliability requirements of the software.

3. Finding error prone paths experiment

In this section we explain the experiment set up and data preparation for the GA in determining the most error prone paths.

3.1. Search environment

The first step involves translating the source code into an environment to apply a searching algorithm. This is achieved by creating a sparse connectivity matrix from the directed graph that represents the software code. For simplicity a tree structure is used to represent the flow of possible executing sequences in the software [6]. To generate the tree structure the source code is first parsed through a software parser to analyze and generate the directed graph. The most complicated step of the parser is function evaluation. Functions at this stage are treated as another node with their functionality attached to the graph at the point of the function call. Each possible end of the function is then attached to the node of the next LOC after the function call. This directed graph is then further transformed into a binary tree which is obtained in three steps.

Firstly, the loops within the graph need to be removed both for reducing the combinations of paths and generating a binary tree structure. Loops refer to both looping source statements such as "while" and recursive program calls. This is achieved by applying rules from basis path testing or Beizer's Loop Tests [14]. Beizer proposes a number of loop iteration categories. In the case of removing loops we apply two of the categories namely "Bypass" (zero times through the loop) and "Once" (single execution of the loop).

Next, all multiple outdegree nodes, that is nodes from which several edges emerge (we call these nodes multiple lead nodes) within the graph need to be transformed into sets of binary nodes. This is legitimate in graph theory. The reason for this step is to create a binary searching space for application of the searching algorithm. This is accomplished by adding a small weighted node to the graph.

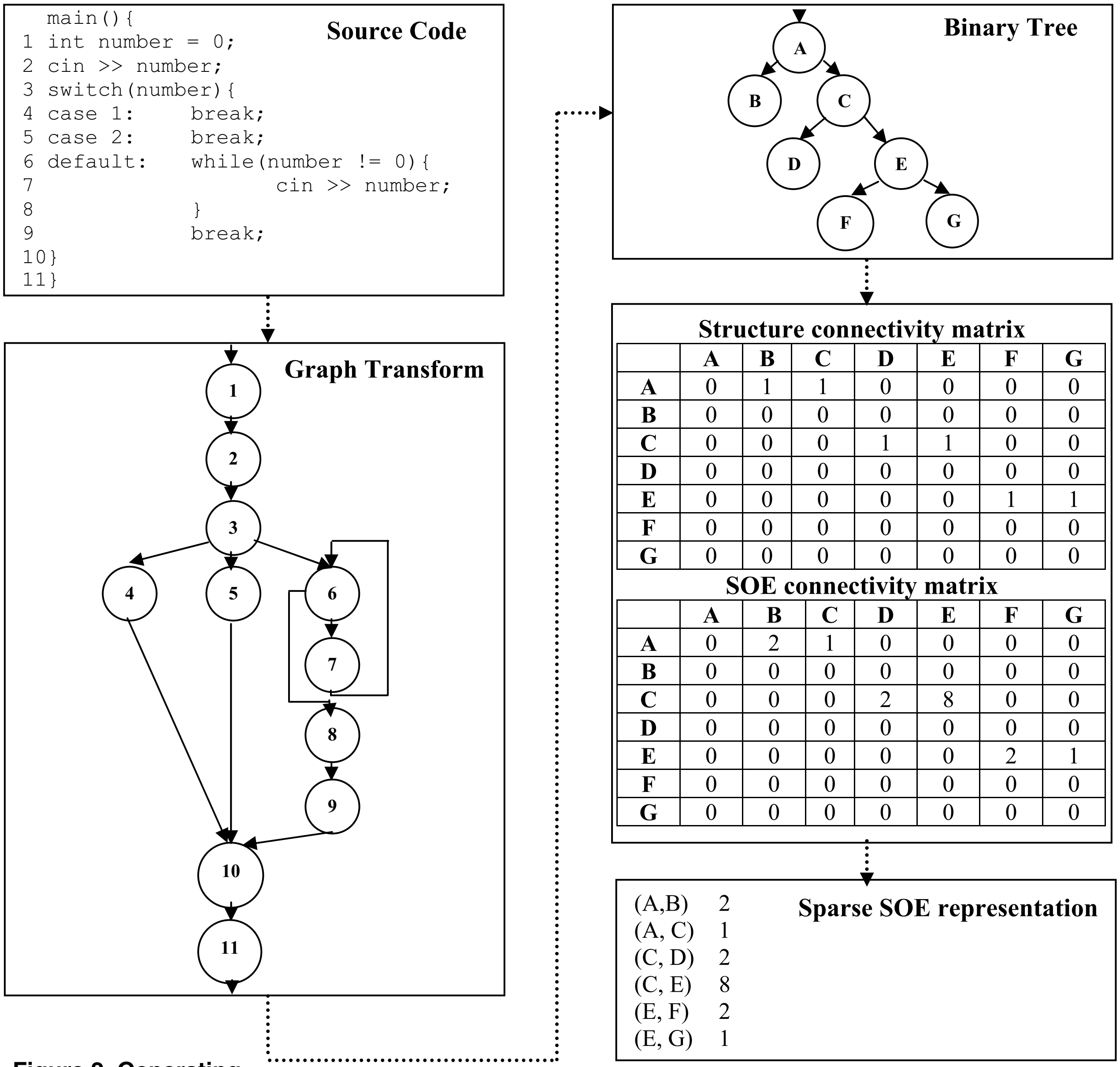
Finally, the trivial paths need to be removed. This step is used to reduce the size of the search space and reduce time spent analyzing equivalent SOE path structures. It is achieved by collapsing the trivial paths down into one node. This node then contains a summed SOE of the previous structure for path selection with the GA. Figure 2 illustrates this process with a small example.

In this example the source code is parsed for structure and the directed graph is derived. An instruction (now a node) can have the SOE from more than one construct type. For example, in node 6 the SOE comes from a "while" and an "assignment". Next, the tree building rules are applied to transform the graph into a binary search tree. Where A = 1,2,3; B = 4,10,11; C = inserted node; D = 5,10,11; E = 6; F = 8,9,10,11; G = 7,8,9,10,11; Next, the tree is represented as a connectivity matrix and finally the matrix is converted to a sparse representation to save on storage as this is an $O(N^2)$ storage space problem. The environment is now suitable for the GA search.

3.2. GA implementation and experiment

Using the SOE as a basis of fitness, a GA can be employed to find the most error prone paths that is, the paths with the highest SOE are best. The advantage of the SOE framework is that the structure of the software, or its components, can be represented as a network, in the form of a connectivity matrix, whose path is a 0/1 string. For this research variable length GA were employed as they provide the necessary advantages for application in this domain (that is, optimizing a large and variable sized search environment [10], [11]).

GA are mathematical constructs based on survival of the fittest which were formalized by Holland [9]. Each generation consists of a population of character strings, or variables, in terms of an alphabet, that are analogous to the chromosomes that we see in our DNA. Each individual character or 'gene' represents a point in a search space and a possible solution. A fitness score is assigned to each solution representing the abilities of an individual to 'compete'. The individual with the optimal (or more generally the near optimal) fitness score is sought.



```
  main(){
1 int number = 0;
2 cin >> number;
3 switch(number){
4 case 1:    break;
5 case 2:    break;
6 default:   while(number != 0){
7                   cin >> number;
8            }
9            break;
10}
11}
```

Structure connectivity matrix

	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	0	0	0	0	0	0	0
C	0	0	0	1	1	0	0
D	0	0	0	0	0	0	0
E	0	0	0	0	0	1	1
F	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0

SOE connectivity matrix

	A	B	C	D	E	F	G
A	0	2	1	0	0	0	0
B	0	0	0	0	0	0	0
C	0	0	0	2	8	0	0
D	0	0	0	0	0	0	0
E	0	0	0	0	0	2	1
F	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0

Sparse SOE representation

(A,B)	2
(A, C)	1
(C, D)	2
(C, E)	8
(E, F)	2
(E, G)	1

Figure 2. Generating the search environment

The GA aims to use selective 'breeding' of the solutions to produce 'offspring' more fit than the parents by combining information from the chromosomes. This is achieved by using crossover and mutation of the strings. Eventually, when the population has converged and is not producing offspring that are noticeably different from those in previous generations, the algorithm is said to have converged to a set of solutions to the problem at hand.

The genomes used for the GA are the paths constructed through the connectivity matrix by a string that is made of zeroes and ones. They represent a right (0) or left (1) direction using the tree structure of the software system under examination. Using the matrix from

Figure 2 this would give us chromosomes such as the paths A-C-D = 01 or A-C-E-F = 001. It is this sequence of zero-and-one genes that is used to develop a chromosome for application with the GA search approach.

Bearing in mind that the nodes are weighted with the SOE, a path is explored and its corresponding SOE values read and summed.

If it is heavy in SOE, that is, if the path is highly error prone then the GA will "reward" it. It is on these heavier paths where priority for testing takes place. To achieve a desired level of reliability paths can be drawn and tested. When a path is drawn it is removed from the search space. This can be continued until a desired level of reliability has been achieved.

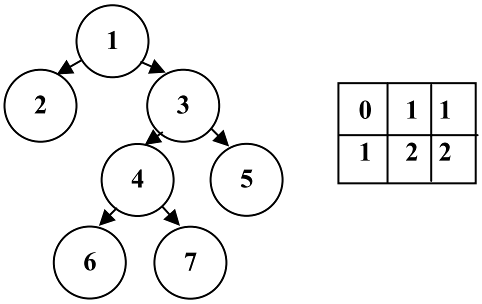
There are two main problems facing a variable length GA. (i) chromosome evaluation and (ii) local minima. The strategies for resolving local minima are explained in the next section.

With a standard fixed length GA and balanced tree structure, the evaluation of the chromosomes can occur without problem; this is because all gene values (directions in a tree) are valid. The issue with using a fixed length GA is that the source code environment is not fixed in length. This can be seen clearly in

Figure 2 where paths of different lengths can occur. Therefore, we need to examine variable length GA.

When using variable length GA invalid genes (directions) can occur. To overcome this, a technique is used to replace invalid genes with a representation character. This effectively splits the chromosome into two parts, one part having the valid genes the other the invalid. This is similar to the splicing and recombination techniques outlined in Goldberg et al. research [11]. Figure 3 outlines an example of the selection and representation process.

In this example two chromosomes of different lengths have been selected for crossover (011) and (1) however, the crossover works only on fixed length representations. Therefore a representation character (in this case 2) has been applied to pad the remaining genes.



0	1	1
1	2	2

Figure 3. Selecting chromosomes

The crossover and mutation phases occur as in fixed length GA. After these stages an evaluation stage determines if a chromosome has undergone change. Continuing the example, after crossover and mutation, we have two chromosomes (0 2 2) and (1 1 1). The evaluation process will determine the validity of the chromosomes. If a gene within the chromosome is invalid then a split occurs and the remainder of the chromosome is set to (2). If the gene is valid then a random (0, 1) gene is assigned.

Using as an example the chromosome (0 2 2) would be evaluated by checking the validity of the first gene, second etc. This would result in the first gene passing as valid the second gene being assigned a random (0, 1) value and the final gene being assigned a value based on the validity of the previous gene. If the second gene is assigned (0) the last gene would remain as 2 if however, the gene was set to 1 then another (0, 1) value would be assigned to the gene which could result in a chromosome such as (0 1 0). By using this process the variable path lengths are removed and the functionality of fixed length GA can be adopted.

A set of experiments was conducted to determine the potential effectiveness of the variable GA approach in identifying the most error prone software paths. The experiments were performed on five field examples ranging from 764 to 3151 lines of code. Many different GA strategies were applied to reduce the problems of local minima and these are outlined in

Table 3. The experiment conducted was to select top 10% and 25% error prone paths within the search environments to illustrate the effectiveness of the GA in selecting top percentage error prone paths. These experiments were performed using Matlab^{TM} on the Griffith University supercomputer. Table 1 shows information of the environments used in the experiments. Loops are included in branch according to the loop rules applied from Beizer [14].

Table 1. Search environments

Environment	Branch	Nested Branch	Calls	Leads	LOC
1	30	9	21	9	977
2	8	17	17	0	882
3	18	23	5	0	764
4	38	26	40	9	1870
5	45	198	92	19	3151

Branch - loops and branches in the data;
Nested Branch – nested branches;
Calls - user function calls not library calls;
Leads - Multiple leading nodes e.g. “switch”;
LOC - lines of code

We also compared the GA performance with two alternative search algorithms; these were Depth First Search (DFS) and Random Restart Greedy Local Search (RGLS).

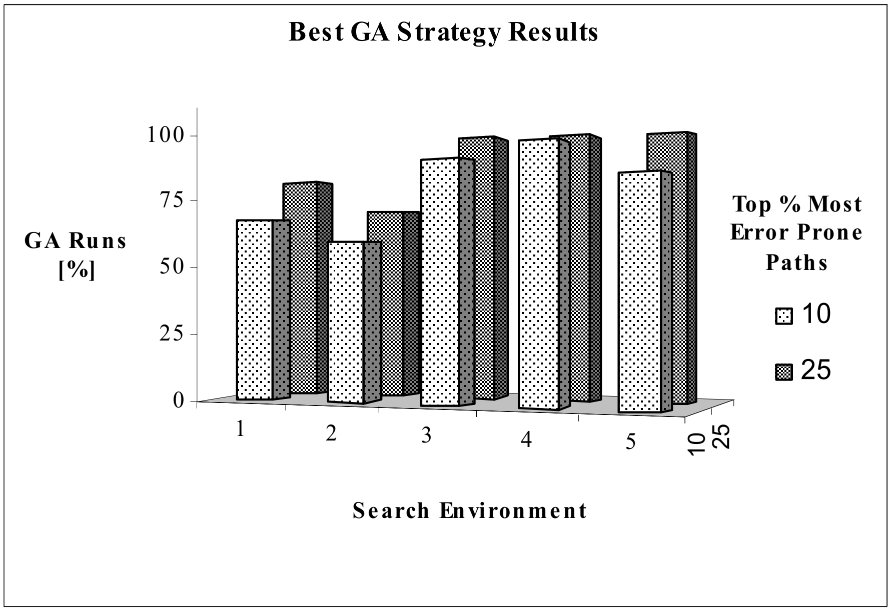


Figure 4. GA path selection efficiency

The DFS approach had a dual purpose: (i) results of the DFS were used to determine all the possible paths in the environment, their coinciding fitness and exhaustive time to search; and (ii) to determine the effectiveness of the GA in outperforming an exhaustive search. At this stage we want to judge the effectiveness of the GA in selecting top 10% and 25% error prone paths.

The second comparison used a RGLS (A greedy search extension that can randomly restart at any point in the local neighborhood to reduce local minima) to determine the effectiveness of the GA in outperforming a local search technique. We compared by taking the average GA search time and setting the RGLS to run for this time or exiting if it out performs the GA.

4. Experimental results and discussion

We found the variable GA performed reasonably well at both the 10% and 25% top error prone paths selections. Figure 4 summarizes the best GA results for each environment from the experiments and Table 2 the strategies used to achieve those results.

The search environments are based on the environments detailed in Table 1. The graph shows the average percentage of GA runs out of 50 trials on the y–axis for the five different matrices. The results are plotted for the top 10% and 25% of most error prone paths within the search environment. The results indicate that the variable GA approach performs consistently across all environments with top 10% error prone path selection efficiency at an average of 79.6% and 91.2% for the top 25% error prone paths.

Next, we compared the GA with two alternative search algorithms these were DFS and RGLS. Figure 5 and Figure 6 illustrate the results from the comparison between the three search techniques. These results are based on an average from 50 trials of each search algorithm on each environment. The figures show that the complexity of the search environment causes varying results with the GA and RGLS technique.

In particular the local search technique performs quite erratically over the different populations. This can be seen if we examine Figure 5.

Table 2. GA path selection criteria

Enviro nment	X (%)	M (%)	Select (type)	Pop (size)	X (type)
1	80	1/L	Roulette	20	Single
2	80	1/L	Roulette	20	Single
3	70	1/L	Roulette	25	Uni.
4	70	1/L	Roulette	30	Uni.
5	70	1/L	Roulette	35	Uni.

X (%) – Crossover probability;
M (%) – Mutation probability – 1/pathLength;
Select (type) – Selection type;
Pop (size) – Population size;
X (type) – Crossover type

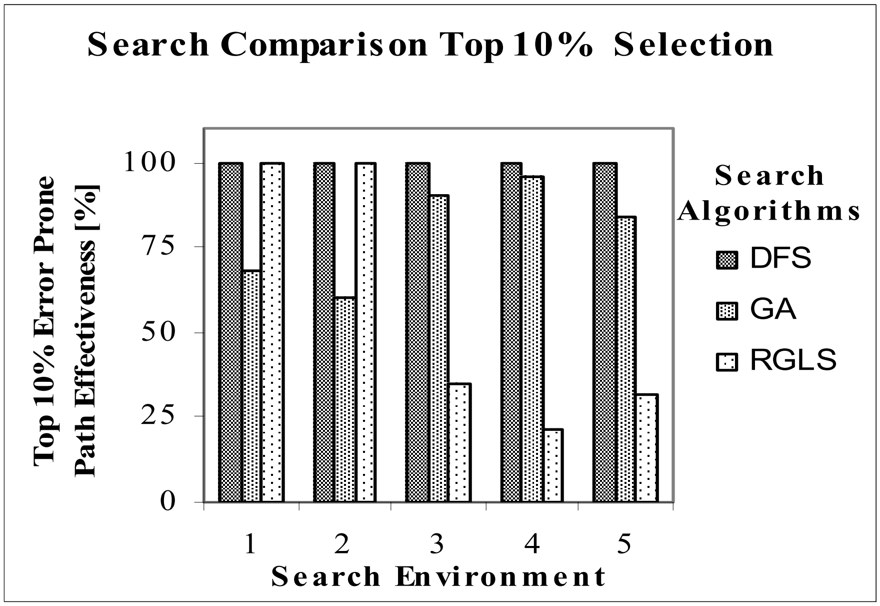


Figure 5. Search comparison top 10% selection

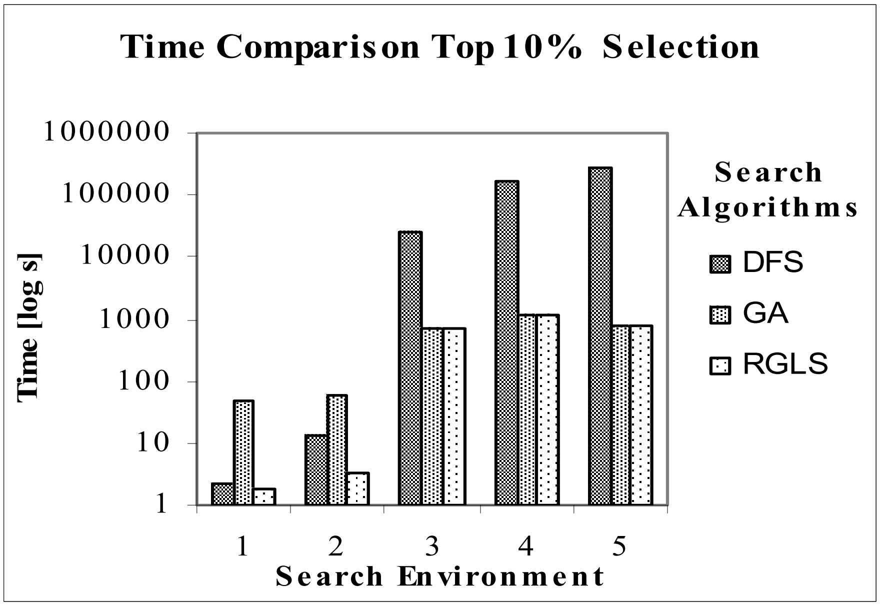


Figure 6. Time comparison top 10% selection

The result of the RGLS technique at search environment (1) is 100% probability of selecting the top 10% error prone paths compared with 21% probability at search environment (4). This is to be expected as the local search technique suffers for a lack of global sampling and quickly gets stuck in local minima. The GA techniques do suffer from the local minima problem however, the results are more consistent across the environments and overall outperform the RGLS technique. Figure 6 shows the time comparison between the techniques.

In the smaller less complex environments (1 and 2) the GA performance was rather slow in comparison with the other algorithms with times of 49.78 seconds and 61.42 seconds. This compared with 2.13 seconds and 13.31 seconds for DFS, and 1.75 seconds and 3.32 seconds for RGLS. It was concluded that these results were due to the size of the population used for the GA.

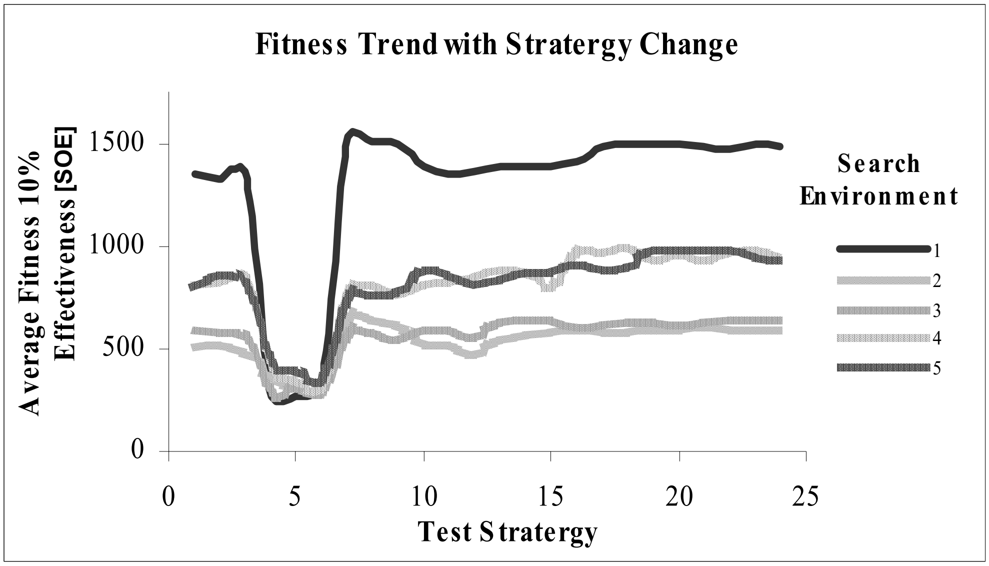


Figure 7. Average fitness trend over strategies

A smaller population had the potential to improve convergence time but a higher probability for lower selection results. The major point to notice was the average search time for the DFS approach increased from a smaller time on the smaller search domains to greater than 350 times for the larger environments. These results are not unexpected, as DFS is based on an exhaustive search approach, implying that if the size of the search space increases, so does the time to search it.

The major problem encountered with the GA was local minima or sub optimal solutions, which are a well known problem, reported by Holland [9] and Goldberg [10]. To achieve better results, a range of different strategies was experimented with ensuring to maintain diversity in the GA.

These strategies are listed in

Table 3. Some variables were selected based on work by Jones, Sthamer and Eyres [12]. The strategy changes included increasing the crossover probability, increasing the mutation probability, using different selection methodologies and changing the size of the population being examined. Therefore by examining strategy 1, the crossover probability will be 70%; the mutation probability will be the reciprocal of the pathlength; the crossover type used will be single point crossover; the selection strategy will be roulette wheel and the population size will be 20.

The graph in Figure 7 displays the average fitness trends for all these GA strategies under test and shows the increase and decrease in average GA fitness as the algorithm strategies are changed. This is calculated by averaging the results for each strategy over the 50 trials and graphing this result. The search environments are based on the details from Table 1.

A number of interesting results emerged from these experiments. Firstly, the major feature noticed in Figure 7 is the dip in average fitness due to the binary tournament selection strategy. This strategy selects two chromosomes randomly and presents the one with the highest fitness for crossover and mutation.

Table 3. GA strategies

Strategy	Variable Type
1-6, 10-24	crossover probability = 0.7
7-9	crossover probability = 0.8
1-9, 13-24	mutation probability = 1/pathlength
10-12	mutation probability = 1/pathlength +6
1-3, 7-24	selection type = roulette wheel
4-6	selection type = binary tournament
1,4,7,10,13, 16, 19,22	crossover type = single point
2,5,8,11,14, 17, 20,23	crossover type = two point
3,6,9,12,15, 18, 21,24	crossover type = uniform
1-12	population = 20
13-15	population = 25
16-18	population = 30
19-21	population = 35
22-24	population = 40

It was found that the average effectiveness of selecting a top 10% error prone path for the roulette wheel strategies (1, 2 and 3) was 56.47%.

In contrast, when using a binary tournament selection strategy (4, 5 and 6) the selection effectiveness dropped to an average of 5.67%. The conclusion raised from this experiment was the population diversity was not kept with the binary tournament selection and convergence occurred too quickly.

Secondly, in this current domain using higher levels of crossover resulted in a more effective selection strategy. It was found that the average effectiveness of the GA increased from 56.47% (1, 2 and 3) to 58.33% (7, 8 and 9). This was predicted as an increase in crossover rate increased the probability of getting the fittest solution. However, the main increase came in the smaller search domains which were more influenced by the increase. In environment (5) the opposite was experienced with a decrease in selection effectiveness. It was concluded that it was caused by decrease in the overall diversity of the population.

Thirdly, it was found that increasing the mutation rate from the reciprocal of the path length (genome length) to the reciprocal of the path length + 6 increased overall effectiveness of the GA. However, the time to find a fitter solution also increased. While this might seem paradox at first, it is consistent with the results found by Jones, Sthamer and Eyres [12], that is, as the diversity of the population increases a fitter solution is achieved and it takes longer for the population to converge on this fitter solution.

Finally, based on work by Spears and Anand [15], we conducted experiments with different population sizes. Spears and Anand found that the size of the population was crucial to the performance of the GA. In general, small populations find good solutions quickly, but are often stuck on local optima. Larger populations are less likely to be caught by local optima, but generally take longer to find good solutions. This theory was corroborated by finding, that while the increase in population increased the effectiveness of the selection the time to find the more fit solution also increased. This is reflected in our results where an increase in population has increased the average effectiveness from 56.47% (1, 2 and 3) to 78.45% (22, 23 and 24) and the times from 352.45 seconds to 1235.78 seconds.

From these experimental results it can be seen that many factors cause varying rates of success with the variable GA approach. For the smaller search spaces, the GA does not perform as well as it does on the larger search areas. This is to be expected based on research by Spears and Anand [15]. However, it can also be seen that the GA performs well in selecting the higher SOE paths from the larger search spaces, which is more representative of a program (as illustrated by our examples). The variable GA approach also out-performs the DFS and RGLS techniques as shown in Figure 5 and Figure 6.

The path clusters selected using the GA technique can then be drawn for testing using traditional testing means like structural, dataflow and functional testing. Therefore if – say - 80% path coverage was required, one could run the GA iteratively to draw paths in a relatively sound order from the higher error prone, down to the lower error prone. The time taken to run the GA many times on a large search environment might be a constraining factor, but this can be easily overcome with a more powerful computer.

An important result is that the GA approach does perform better than exhaustive DFS time. If several iterations are done, then most likely the best strategy becomes part of the resulting set of path clusters that cover a minimum desired percentage of SOE to be detected in testing. The GA technique performs quickly with reasonable results, even if it does not find the global optimum at first run.

While the GA perform better on larger examples, the time to run the GA also grow with the larger examples.

5. Conclusions

In this paper we have demonstrated that it is possible to apply variable length Genetic Algorithm techniques for finding the most error prone paths for improving software testing efficiency. This was achieved with Genetic Algorithms performing well across all the environments tested by finding on average up to 80% of the paths within the 10% most error prone paths, and expanding to 90% within the 25% most error prone paths. The Genetic Algorithms also outperformed the exhaustive search and local search techniques.

In conclusion, by examining the most error prone paths first, we obtain a more effective way to approach testing which in turn helps to refine effort and cost estimation in the testing phase. Our experiments conducted so far are based on relatively small examples and more research needs to be conducted with larger commercial examples. At this stage we want to judge the effectiveness of the GA in selecting top 10% and 25% error prone paths. Future research will involve comparing GA selected paths in larger test data and further refining the method presented.

6. References

[1] M.R. Lyu, *"Handbook of Software Reliability Engineering"*, McGraw-Hill Publishing Company and IEEE Computer Society Press, New York, 1995.

[2] J.M. Bieman & H.Yin, "Designing for software testability using automated oracles", *Proc. International Test Conf.*, Sep 1992, pp 900-907.

[3] W.E. Wong, J. R. Horgan, A. P. Mathur & A. Pasquini, "Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application", *Journal of Systems and Software*, *Vol. 48*, No. 2, October 1999, pp 79-89.

[4] Y.K. Malaiya, N. Li, J. Bieman, R. Karcich & B. Skibbe, "The relationship between test coverage and reliability", *Proc. of the International Symposium on Software Reliability Engineering*, Nov 1994, pp 186-195.

[5] T. J. McCabe & C. W. Butler, "Design Complexity Measurement and Testing", *Communications of the ACM,* Vol 32, No. 12, December 1989, pp 1415-1425.

[6] B. Beizer, "*Software Testing Techniques"*, 2nd edition, New York: Van Nostrand Reinhold, 1990.

[7] T. Ball & J.R. Larus, "Programs follow paths", *Technical Report MSR-TR-99-01 Microsoft Research,* Microsoft Research, Redmond, WA, January 1999.

[8] R. Sitte, "A Framework for Quantifying Error Proneness in Software", *Proceedings of the first Asia-Pacific Conference on Quality Software*, Hong Kong, October 30-31 1999, pp 63-68.

[9] J. H. Holland, *"Adaptation in Natural and Artificial Systems"*, *The University of Michigan Press,* 1975.

[10] D.E. Goldberg, *"Genetic Algorithms in Search, Optimisation and Machine Learning",* Addison Wesley, Reading, 1989.

[11] D.E. Goldberg, K. Deb & B. Korb "Messy Genetic Algorithms: Motivation, analysis, and first results", *Complex Systems, Vol. 3,* 1989, pp. 493-530.

[12] B. F. Jones, H. Sthamer & D.E. Eyres, "A strategy for using genetic algorithms to automate branch and fault-based testing", *Computer Journal, Vol. 41,* No. 2, 1998, pp 98-107.

[13] S. Kirkpatrick, C. D. Gelatt & M. P. Vecchi. "Optimization by simulated annealing", *Science, Vol. 220,* pp. 671-680.

[14] B. Beizer, "*Black-box Testing: techniques for functional testing of software and systems"*, Wiley, New York, 1995.

[15] W. M. Spears & V. Anand, "A study of crossover operators in genetic programming", *6th International Symposium on Methodologies for Intelligent Systems*, Charlotte, N.C., USA, 16. - 19. October 1991, pp 409-418