Vol 7. No. 5 - December, 2014

African Journal of Computing & ICT

© 2014 Afr J Comp & ICT – All Rights Reserved - ISSN 2006-1781
www.ajocict.net

Genetic Algorithm Technique In Program Path Coverage For Improving Software Testing

Y.K. Saheed
Department of Computer Science,
Al-Hikmah University, Ilorin, Nigeria
kayodesaheed@gmail.com

A.O. Babatunde
Department of Computer Science,
University of Ilorin, Ilorin, Nigeria
babatunde.ao@unilorin.edu.ng

ABSTRACT

Software testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality. This paper proposes Genetic Algorithm Technique in program path coverage for optimizing software testing. Test data generation is a key problem in software testing and its automation will improve the efficiency and effectiveness of software testing. We used Genetic Algorithm technique for improving the efficiency of software testing by identifying the error prone path in a program. We do this by using Genetic Algorithms approach that optimize and select the program path which are weighted in accordance with the error prone path. Genetic Algorithm can present a robust non-linear search technique and also better quality of solution and therefore reduction in cost in software testing industry. Exhaustive software testing is not feasible. Only the selective parts of the software are tested. Therefore design of a set of test cases is required in such a manner that it can find out as many faults as possible.

Keywords: Software Testing, Genetic Algorithm, Path coverage, Test Data, Software Under Test.

African Journal of Computing & ICT Reference Format:
Y.K. Saheed & A.O. Babatunde (2014) Genetic Algorithm Technique In Program Path Coverage For Improving Software Testing. Afr J. of Comp & ICTs. Vol 7, No. 5. Pp 151-.158.

1. INTRODUCTION

Software testing is as old as the hills in the history of digital computers. The testing of software is an important means of assessing the software to determine its quality. Since testing typically consumes 40~50% of development efforts, and consumes more effort for systems that require higher levels of reliability, it is a significant part of the software engineering. With the development of Fourth generation languages (4GL), which speeds up the implementation process, the proportion of time devoted to testing increased. As the amount of maintenance and upgrade of existing systems grow, significant amount of testing will also be needed to verify systems after changes are made [6]. Despite advances in formal methods and verification techniques, a system still needs to be tested before it is used. Testing remains the truly effective means to assure the quality of a software system of non-trivial complexity [1], as well as one of the most intricate and least understood areas in software engineering [5]. Testing, an important research area within computer science is likely to become even more important in the future.

Now more and more critical applications are implemented globally. This increased expectation for error-free functioning of software has increased the demand for quality output from software vendors. Software Testing is the process used to help and identify the correctness, completeness, security, reliability & quality of developed software. It is one of the major phases in all the phases of Software Development Life Cycle. The potential cost savings from handling software errors within a development cycle, rather than the subsequent cycles, has been estimated at nearly 40 billion dollars by the National Institute of Standards and Technology. This figure emphasizes that current testing methods are often inadequate, and hence reduction of software bugs and errors is an important area of research with a substantial payoff [2].

Therefore reducing the efforts, time and ultimately the cost of software development had always been a challenge for both the software industry and academia. The automatic generation of test data using Genetic Algorithms (GA) has been studied by [11]. Here the use of GA is explored to automatically generate test data that covers the most error-prone path. GA are commonly applied to search problems within AI. They maintain a population of structures that evolve according to rules of selection, mutation and reproduction. Translating these concepts to the problem of test data generation, the population is set of test data. Each element in the set (e.g. a group of data items used in one run of the program) is an individual [8]. The fitness of an individual corresponds to the coverage of an error-prone path of the program under test.

1.1 Genetic Algorithms

The most popular technique in evolutionary computation research has been the genetic algorithm. In the traditional genetic algorithm, the representation used is a fixed-length bit string. Each position in the string is assumed to represent a particular feature of an individual, and the value stored in that position represents how that feature is expressed in the solution. Usually, the string is "evaluated as a collection of structural features of a solution that have little or no interactions". The analogy may be drawn directly to genes in biological organisms. Each gene represents an entity that is structurally independent of other genes. The main reproduction operator used is bit-string crossover, in which two strings are used as parents and new individuals are formed by swapping a sub-sequence between the two strings. Another popular operator is bit-flipping mutation, in which a single bit in the string is flipped to form a new offspring string

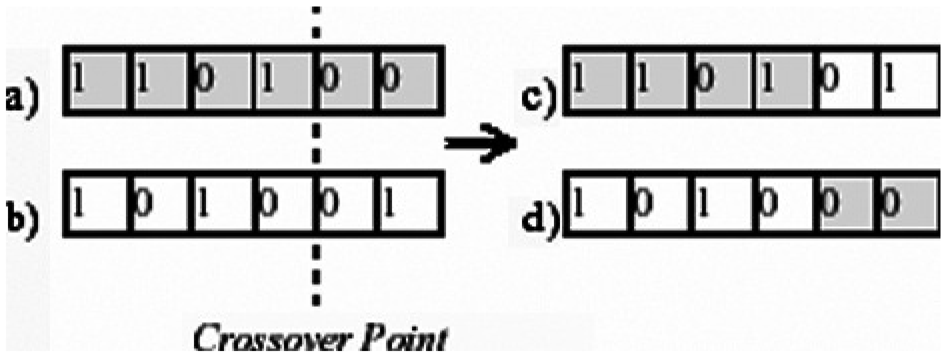


Figure 1: Bit-string crossover of parents a & b to form offspring c & d

A variety of other operators have also been developed, but are used less frequently (e.g., inversion, in which a subsequence in the bit string is reversed). A primary distinction that may be made between the various operators is whether or not they introduce any new information into the population. Crossover, for example, does not while mutation does. All operators are also constrained to manipulate the string in a manner consistent with the structural interpretation of genes. For example, two genes at the same location on two strings may be swapped between parents, but not combined based on their values. Traditionally, individuals are selected to be parents probabilistically based upon their fitness values, and the offspring that are created replace the parents. For example, if N parents are selected, then N offspring are generated which replace the parents in the next generation. In nature, an individual in population competes with each other for virtual resources like food, shelter and so on. Also in the same species, individuals compete to attract mates for reproduction.

Due to this selection, poorly performing individuals have less chance to survive, and the most adapted or "fit" individuals produce a relatively large number of offspring's. It can also be noted that during reproduction, a recombination of the good characteristics of each ancestor can produce "best fit" offspring whose fitness is greater than that of a parent. After a few generations, species evolve spontaneously to become more and more adapted to their environment. In 1975, Holland developed this idea in his book "Adaptation in natural and artificial systems". He described how to apply the principles of natural evolution to optimization problems and built the first Genetic Algorithms. Holland's theory has been further developed and now Genetic Algorithms (GAs) stand up as a powerful tool for solving search and optimization problems. Genetic algorithms are based on the principle of genetics and evolution.

Comparison of natural evolution and genetic algorithm terminology

Natural evolution	**Genetic algorithm**
Chromosome	String
Gene	Feature or character
Allele	Feature value
Locus	String position
Genotype	Structure or coded string
Phenotype	Parameter set, a decoded structure

2. CONVENTIONAL OPTIMIZATION AND SEARCH TECHNIQUES

The basic principle of optimization is the efficient allocation of scarce resources. Optimization can be applied to any scientific or engineering discipline. The aim of optimization is to find an algorithm, which solves a given class of problems. There exist no specific method, which solves all optimization problems. Consider a function,

$F(x)$: $[X^{1,}X^{u}] \longrightarrow [0,1]$: …………………(1)

where,

$$F(x) = \begin{cases} 1, & if\ ||x - a|| < \epsilon, \epsilon > 0 \\ -1 & elsewhere \end{cases}$$

For the above function, F can be maintained by decreasing € or by making the interval of $[X^1,X^u]$ large. Thus a difficult task can be made easier. Therefore, one can solve optimization problems by combining human creativity and the raw processing power of the computers.
The various conventional optimization and search techniques available are discussed as follows:

2.1 Gradient-Based Local Optimization Method

When the objective function is smooth and one need efficient local optimization, it is better to use gradient based or Hessian based optimization methods. The performance and reliability of the different gradient methods varies considerably.

2.2 Random Search

Random search is an extremely basic method. It only explores the search space by randomly selecting solutions and evaluates their fitness. This is quite an unintelligent strategy, and is rarely used by itself. Nevertheless, this method sometimes worth being tested. It doesn't take much effort to implement it, and an important number of evaluations can be done fairly quickly. For new unresolved problems, it can be useful to compare the results of a more advanced algorithm to those obtained just with a random search for the same number of evaluations. Nasty surprises might well appear when comparing for example, genetic algorithms to random search. It's good to remember that the efficiency of GA is extremely dependant on consistent coding and relevant reproduction operators. Building a genetic algorithm, which performs no more than a random search happens more often than we can expect.
If the reproduction operators are just producing new random solutions without any concrete links to the ones selected from the last generation, the genetic algorithm is just doing nothing else that a random search.
Random search does have a few interesting qualities. However good the obtained solution may be, if it's not optimal one, it can be always improved by continuing the run of the random search algorithm for long enough. A random search never gets stuck in any point such as a local optimum. Furthermore, theoretically, if the search space is finite, random search is guaranteed to reach the optimal solution. Unfortunately, this result is completely useless. For most of problems we are interested in, exploring the whole search space takes far too long an amount of time.

2.3 Stochastic Hill Climbing

Efficient methods exist for problems with well-behaved continuous fitness functions. These methods use a kind of gradient to guide the direction of search. Stochastic Hill Climbing is the simplest method of these kinds. Each iteration consists in choosing randomly a solution in the neighborhood of the current solution and retains this new solution only if it improves the fitness function. Stochastic Hill Climbing converges towards the optimal solution if the fitness function of the problem is continuous and has only one peak (unimodal function).

On functions with many peaks (multimodal functions), the algorithm is likely to top on the first peak it finds even if it is not the highest one. Once a peak is reached, hill climbing cannot progress anymore, and that is problematic when this point is a local optimum. Stochastic hill climbing usually starts from a random select point. A simple idea to avoid getting stuck on the first local optimal consists in repeating several hill climbs each time starting from a different randomly chosen points. This method is sometimes known as iterated hill climbing. By discovering different local optimal points, it gives more chance to reach the global optimum. It works well if there is not too many local optima in the search space. But if the fitness function is very "noisy" with many small peaks, stochastic hill climbing is definitely not a good method to use. Nevertheless such methods have the great advantage to be really easy to implement and to give fairly good solutions very quickly.

2.4 Simulated Annealing

Simulated Annealing was originally inspired by formation of crystal in solids during cooling i.e., the physical cooling phenomenon. As discovered a long time ago by iron age blacksmiths, the slower the cooling, the more perfect is the crystal formed. By cooling, complex physical systems naturally converge towards a state of minimal energy. The system moves randomly, but the probability to stay in a particular configuration depends directly on the energy of the system and on its temperature.

2.5 Symbolic Artificial Intelligence (AI)

Most symbolic AI systems are very static. Most of them can usually only solve one given specific problem, since their architecture was designed for whatever that specific problem was in the first place. Thus, if the given problem were somehow to be changed, these systems could have a hard time adapting to them, since the algorithm that would originally arrive to the solution may be either incorrect or less efficient. Genetic algorithms (or GA) were created to combat these problems. They are basically algorithms based on natural biological evolution. The architecture of systems that implement genetic algorithms (or GA) is more able to adapt to a wide range of problems.

3. A SIMPLE GENETIC ALGORITHM

An algorithm is a series of steps for solving a problem. A genetic algorithm is a problem solving method that uses genetics as its model of problem solving. It's a search technique to find approximate solutions to optimization and search problems. Basically, an optimization problem looks really simple. One knows the form of all possible solutions corresponding to a specific question. The set of all the solutions that meet this form constitute the search space. The problem consists in finding out the solution that fits the best, i.e. the one with the most payoffs, from all the possible solutions. If it's possible to quickly enumerate all the solutions, the problem does not raise much difficulty. But, when the search space becomes large, enumeration is soon no longer feasible simply because it would take far too much time. In this it's needed to use a specific technique to find the optimal solution.

Genetic Algorithms provides one of these methods. Practically they all work in a similar way, adapting the simple genetics to algorithmic mechanisms. GA handles a population of possible solutions. Each solution is represented through a chromosome, which is just an abstract representation. Coding all the possible solutions into a chromosome is the first part, but certainly not the most straightforward one of a Genetic Algorithm. A set of reproduction operators has to be determined, too. Reproduction operators are applied directly on the chromosomes, and are used to perform mutations and recombinations over solutions of the problem. Appropriate representation and reproduction operators are really something determinant, as the behavior of the GA is extremely dependant on it.

Frequently, it can be extremely difficult to find a representation, which respects the structure of the search space and reproduction operators, which are coherent and relevant according to the properties of the problems. Selection is supposed to be able to compare each individual in the population. Selection is done by using a fitness function. Each chromosome has an associated value corresponding to the fitness of the solution it represents. The fitness should correspond to an evaluation of how good the candidate solution is.

The optimal solution is the one, which maximizes the fitness function. Genetic Algorithms deal with the problems that maximize the fitness function. But, if the problem consists in minimizing a cost function, the adaptation is quite easy. Either the cost function can be transformed into a fitness function, for example by inverting it; or the selection can be adapted in such way that they consider individuals with low evaluation functions as better. Once the reproduction and the fitness function have been properly defined, a Genetic Algorithm is evolved according to the same basic structure. It starts by generating an initial population of chromosomes. This first population must offer a wide diversity of genetic materials. The gene pool should be as large as possible so that any solution of the search space can be engendered. Generally, the initial population is generated randomly.

Then, the genetic algorithm loops over an iteration process to make the population evolve. Each iteration consists of the following steps:

i. SELECTION: The first step consists in selecting individuals for reproduction. This selection is done randomly with a probability depending on the relative fitness of the individuals so that best ones are often chosen for reproduction than poor ones.
ii. REPRODUCTION: In the second step, offspring are bred by the selected individuals. For generating new chromosomes, the algorithm can use both recombination and mutation.
iii. EVALUATION: Then the fitness of the new chromosomes is evaluated.
iv. REPLACEMENT: During the last step, individuals from the old population are killed and replaced by the new ones.

4. STEPS FOR AUTOMATIC TEST DATA GENERATION

Test-data selection, and consequently generation, is all about locating test-data for a particular test criterion. Test data generation for path testing consists of four (4) basic steps:

1. In this step, the source program is transferred to a graph that represents the control flow of the program.
2. Target path selection: In path testing, paths are extracted from the control flow graph, and some paths might be very meaningful and need to be selected as target path for testing.
3. Test case generation and execution: In this step, the algorithm automatically creates new test cases to execute new path and leads the control flow to the target path. Finally, a suitable test case that executes the target paths could be generated.
4. Test result evaluation: This step is to execute the selected path and to determine the test criteria is satisfied.

5. METHODOLOGY

This section describes generation of test data using GA that achieve a certain level of coverage of the program. Our approach uses a weighted Control Flow Graph (CFG) technique. Path testing searches the program domain for suitable test cases that covers every possible path in the software under test. However, it is generally impossible to achieve this goal due to following reasons [10].

- A program may contain an infinite number of paths when the program has loops.
- The number of paths in a program is exponential to the number of branches in it and many of them may be unfeasible.
- The number of test cases is too large, since each path can be covered by several test cases.

Since it is impossible to cover all paths in software, the problem of path testing selects a subset of paths to execute and find test data to cover it. Here a program is viewed as control flow graph. It is a simple notation for the representation of control flow. The control flow of a program can be represented by a directed graph with a set of nodes and a set of edges [10], [11]. Each node represents a statement. The edges of the graph are then possible transfers of control flow between the nodes. A path is a finite sequence of nodes connected by edges. An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph an independent path must move along at least edge that has not been traversed before the path is defined.

6. CONTROL FLOW GRAPH

Input - CFG of code Assign weights to edges of CFG – More weights are assigned to edges that are more error prone. Firstly weight is assigned to initial node of CFG. If the CFG contains large number of edges then large weight is assigned to first node, otherwise small weight such as 10 is assigned. Then on the basis of this initial node weights are assigned to other nodes. Incoming weight is divided and distributed to all the outgoing edges of the node. More weight is given to branches and loops and less weight is given to edges of sequential path. The CFG for test function is shown in figure 2.

```
0testFunc(int k, int j){
1      int r;
2if(k==0)
3       return0
4   if(k>j){
5          r = j;
6          j = k;
7          k = r;
        }
8     r=j%k;
9     while(r!=0){
10       j = k;
11       k =r;
12       r=j%k;
      }
13   return k
     }
```

Figure 2 Control Flow Graph for Test function.

6.1 Initialization

An initial test set is generated randomly in the space of possible input values.

6.2 Selection

The selection of parents for reproduction is done according to a probability distribution based on the individual's fitness values. First the fitness value is calculated using the Fitness function proposed in the algorithm. Weights are used to determine the relative contribution of a path to the fitness calculation. Thus, more weight is assigned to a path which is more "critical". Criticality of the path to test data generation is based on the fact that predicate, loop and branch nodes are given preference over sequential nodes during software testing. The fitness function we are using here is

F = $\sum_{i=1}^{n} wi$ ....(2)

Where, wi = weight assigned to i-th edge on the path under consideration.
Higher weights are assigned to the edges of path corresponding to the critical section of the code for example loops, branch statements, control statements etc. for which testing is essential. After all the fitness function values are calculated, the probability of selection pj for each path j, so that

pj = Fj/ $\sum_{j} fj$ ....(3)

Where, j=1 to n
n= initial population size
Then cumulative probability ck is calculated for each path k with equation:

ck=$\sum_{j=1}^{k} pj$..................................(4)

6.3 Crossover

Crossover probability (Cp) is decided. It is an adjustable parameter. For each parent selected, a random real number r is generated in the range [0, 1]; if r < Cp then select the parent for crossover. After that, the selected data is formatted randomly. Each pair of parents generates two new paths, called offspring. For the problem in hand, one point crossover is suitable.

6.4 Mutation

Mutation probability (Mp) is decided. It is an adjustable parameter. To perform mutation, for each chromosome in the offspring and for each bit within the chromosome, generate a random real number r in the range [0, 1]; if r < Mp then mutate the bit. These major components including the fitness function will evolve test data to better ones, trying to find a candidate that covers the target path. The crossover process tries to create better test data from fitter ones, while mutation introduces diversity into population, avoiding being stuck at local optima solutions.

According to [11], GA improves the search from one generation to the next, and performs better than random testing, where the search was absolute random and does not show improvement through the generations. Double crossover is more successful in path coverage. Also selecting parent for reproduction according to their fitness is more efficient than random selection and mutation rate is better adjusted with program at hand. [12] says GA requires up to two orders of magnitude fewer tests than random testing and achieves 100% branch coverage. The advantage of GAs is that through the search and optimization process, test sets are improved such that they are at or close to the input sub domain boundaries. According to [13], test data generation using GA performs better compared to random test data generation.

Program	Random	Genetic Algorithms
Binary search	53.3	66.7
Bubble sort 1	100	100
Bubble sort 2	44.4	44.4
Insertion sort	100	100
Triangle Classification	48.6	84.3
Warshall's Algorithm	91.7	100

Table 1. Comparative results of test data generation using GA and random testing

The comparative results on small math programs with the goal of achieving condition decision coverage are shown in Table 1. Genetic search outperformed random test data generation by a considerable margin in most of programs and always performed at least as well.

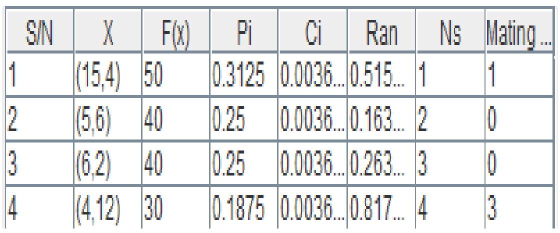
Sample 1

Initial population: (k, j) [(15, 4), (5, 6), (6, 2), (4, 12)]

Fitness function used:

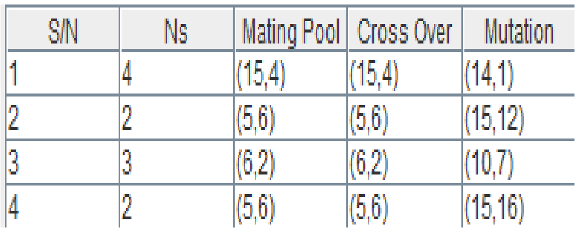
Summation of weights of path traversed by a given input data in CFG

For example (15, 4) will travel the path 0-1-2-4-5-6-7-8-9-6-7-8-9-6-10-11-12-13and therefore its fitness value is 50 Since the mating pool consists of only (15, 4) therefore this is the test data that should be used for the testing of the code during execution.



S/N	X	F(x)	Pi	Ci	Ran	Ns	Mating ...
1	(15,4)	50	0.3125	0.0036...	0.515...	1	1
2	(5,6)	40	0.25	0.0036...	0.163...	2	0
3	(6,2)	40	0.25	0.0036...	0.263...	3	0
4	(4,12)	30	0.1875	0.0036...	0.817...	4	3

Figure 3 Test Result Generation 1



S/N	Ns	Mating Pool	Cross Over	Mutation
1	4	(15,4)	(15,4)	(14,1)
2	2	(5,6)	(5,6)	(15,12)
3	3	(6,2)	(6,2)	(10,7)
4	2	(5,6)	(5,6)	(15,16)

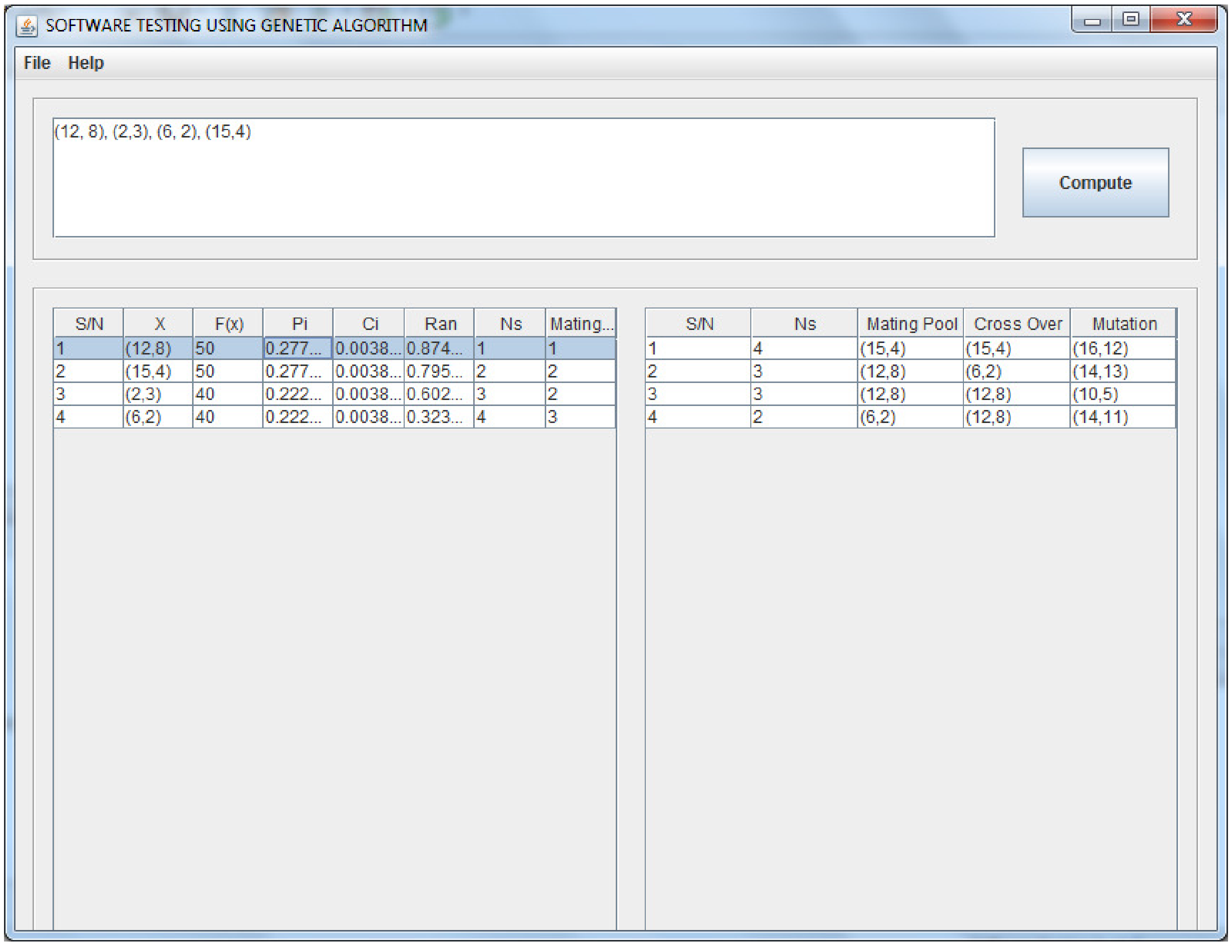
Figure 4 Crossover and Mutation

Sample 2

Initial population: (k, j)

(12, 8), (2, 3), (6, 2), (15, 4)

Fitness function values of input population is calculated in coloumn 3, then probability. Coloumn 5 show the cumulative probability. Random number are generated to simulate the GA process.



(12, 8), (2,3), (6, 2), (15,4)

S/N	X	F(x)	Pi	Ci	Ran	Ns	Mating...
1	(12,8)	50	0.277...	0.0038...	0.874...	1	1
2	(15,4)	50	0.277...	0.0038...	0.795...	2	2
3	(2,3)	40	0.222...	0.0038...	0.602...	3	2
4	(6,2)	40	0.222...	0.0038...	0.323...	4	3

S/N	Ns	Mating Pool	Cross Over	Mutation
1	4	(15,4)	(15,4)	(16,12)
2	3	(12,8)	(6,2)	(14,13)
3	3	(12,8)	(12,8)	(10,5)
4	2	(6,2)	(12,8)	(14,11)

Figure 5 Test Result

7. CONCLUSION

In software testing, the generation of testing data is one of the key steps which have a great effect on the automation of software testing. The greatest merit of genetic algorithm in program testing is its simplicity. Genetic algorithms are often used for optimization problems in which the evolution of a population is a search for a satisfactory solution given a set of constraints. In this study it was shown that it is possible to apply Genetic Algorithm techniques for finding the most error prone paths for improving software testing efficiency. The study conducted so far are based on relatively small sample and more research needs to be conducted with larger commercial samples. Further research in this area will ultimately reduce the costs associated with software testing.

REFRENCES

[1] E. F. Miller, (1980) Introduction to Software Testing Technology, Tutorial: Software Testing & Validation

[2] G.Raghurama and Praveen Ranjan Srivastava, Software Testing using Optimization Techniques, a proposal for research.

[3] [Harmen-Hinrich Sthamer, (1995) The Automatic Generation of Software Test Data Using Genetic Algorithms, Ph.D thesis, University of Glamorgan, Nov. 1995.

[4] Hermadi I., Lokan C. and Sarker R.,(2010) Genetic Algorithm Based Path Testing: Challenges and Key Parameters Second WRI World Congress on Software Engineering, 2010

[5] J. A. Whittaker, (2000) What is Software Testing? And Why Is It So Hard? IEEE Software, January 2000,

[6] J. J. Marciniak,(1994) Encyclopedia of software engineering, Volume 2, New York, NY: Wiley, 1994, pp. 1327-1358

[7] Maha Alzabidi, Ajay Kumar and A.D. Shaligram, (2009) Automatic Software Structural Testing by using Evolutionary Algorithms for Test Data Generations, IJCSNS International Journal of Computer Science and Network Security, VOL.9 No.4, April 2009.

[8] Marc Roper, Iain Maclean, Andrew Brooks, James Miller and Murray Wood, Genetic Algorithms and the Automatic Generation of Test Data, University of Strathclyde, U.K.

[9] Maziyar Karbalaee Shabani, Mansour Ahmadi, Siroos Keshavarz, Faezeh Sadat Babamir and Seyed Mehrdad Babamir, (2010) GA based-Software Test Data Generator, Using Dynamic Repetition Frequency, (IJCNS) International Journal of Computer and Network Security, Vol. 2, No. 7, July 2010.

[10] Srivastava P.R, and T. Kim T.K, (2009), Application of Genetic Algorithm in Software Testing International Journal of Software Engineering and Its Applications,Vol.3, No.4,2009,pp.87-96. Techniques, Second Edition, IEEE Catalog No. EHO 180-0, pp. 4-16

[11] Maha Alzabidi, Ajay Kumar, and A.D. Shaligram, (2009) “Automatic Software Structural Testing by Using Evolutionary Algorithms for Test Data Generations”, IJCSNS International Journal of Computer Science and Network Security, VOL.9, No.4, April 2009.

[12] B. Jones, H. Sthamer, D.Eyres. (1996) Automatic Structural Testing using Genetic Algorithms. Software Engineering Journal 11(5), september 1996, pp 299-306

[13] C. C. Michael, G. E. McGraw and M. A. Schatz, (2001) ”Generating software test data by evolution”, IEEE Transactions on Software Engineering, Vol. 27, No.12, pp. 1085-1110, 2001.