PROGRAM TEST DATA GENERATION FOR BRANCH COVERAGE WITH GENETIC ALGORITHM: COMPARATIVE EVALUATION OF A MAXIMIZATION AND MINIMIZATION APPROACH

AnkurPachauriand Gursaran

Department of Mathematics, Dayalbagh Educational Institute, Agra 282110
ankurpachauri@gmail.com, gursaran.db@gmail.com

ABSTRACT

In search based test data generation, the problem of test data generation is reduced to that of function minimization or maximization.Traditionally, for branch testing, the problem of test data generation has been formulated as a minimization problem. In this paper we define an alternate maximization formulation and experimentally compare it with the minimization formulation. We use a genetic algorithm as the search technique and in addition to the usual genetic algorithm operators we also employ the path prefix strategy as a branch ordering strategy and memory and elitism. Results indicate that there is no significant difference in the performance or the coverage obtained through the two approaches and either could be used in test data generation when coupled with the path prefix strategy, memory and elitism.

KEYWORDS

Search based test data generation, program test data generation, genetic algorithm, software testing

1. INTRODUCTION

Search-based software test data generation has emerged [1, 2, 3, 4, 5, 6] as a significant area of research in software engineering. In search based test data generation, the problem of test data generation is reduced to that of function minimization or maximization. The source code is instrumented to collect information about the program as it executes. Collected information is used to heuristically measure how close the test data is to satisfying the test requirements. The measure is then used to modify the input parameters to progressively move towards satisfying the test requirement. It is here that the application of metaheuristic search techniques has been explored. Traditionally, for branch testing, the problem of test data generation has been formulated as a minimization problem. In this paper we define an alternate maximization formulation and experimentally compare with the traditional minimization formulation.

During testing, program P under test is executed on a *test se*t of *test data* - a specific point in the input domain - and the results are evaluated. The test set is constructed to satisfy a test adequacy criterion that specifies test requirements [7, 8]. The branch coverage criterion is a test adequacy criterion that is based on the program flow graph. More formally, a test set T is said to satisfy the branch coverage criterion if on executing P on T, every branch in P's flow graph is traversed at least once.

Metaheuristic techniques such as genetic algorithms [9], quantum particle swarm optimization [10], scatter search [11] and others have been applied to the problem of automated test data generation and provide evidence of their successful application. Amongst these several have addressed the issue of test data generation with program-based criteria [10]and in particular the branch coverage criterion [10, 11, 12, 13, 14, 15, 16, 17, 18].Further, [12, 13, 14, 19, 20] have formulated the problem as a minimization problem.

In this paper, in Section 2 we describe the Genetic Algorithm (GA) and we outline the application of GAs for test data generation in Section 3. In Section 3 we also introduce the maximization and minimization approaches. In Section 4 we present the experimental setup and in Section 5 we discuss the results of the experiments. Section 6 concludes the paper.

2. GENETIC ALGORITHM

Genetic Algorithm (GA)is a search algorithm that is based on the idea of genetics and evolution in which new and fitter set of string individuals are created by combining portions of fittest string individuals of the parent population[21]. A genetic algorithm execution begins with a random *initial population* of candidate solutions $\{s_i\}$ to an objective function f(s). Each candidate s_i is generally a vector of parameters to the function f(s) and usually appears as an encoded binary string (or *bit string*) called a *chromosome* or a *binary individual*. An encoded parameter is referred to as a *gene*, where the parameter's values are the gene's *alleles*. If there are *m* inputs parameters with the i^{th} parameter expressed in n_i bits, then the *length* of the chromosome is simply $\sum_i n_i$. In this paper each binary individual, or chromosome, represents an encoding of test data.

After creating the initial population, each chromosome is evaluated and assigned a *fitness* value. Evaluation is based on a *fitness function* that is problem dependent. From this initial selection, the population of chromosomes iteratively evolves to one in which candidates satisfy some termination criteria or, as in our case, fail to make any forward progress. Each iteration step is also called a *generation*.

Each generation may be viewed as a two stage process [21]. Beginning with the *current population*, *selection* is applied to create an *intermediate population* and then *recombination* and *mutation* are applied to create the *next population*. The most common selection scheme is the *roulette-wheel selection* in which each chromosome is allocated a wheel slot of size in proportion to its fitness. By repeatedly spinning the wheel, individual chromosomes are chosen using "stochastic sampling with replacement" to construct the intermediate population. Additionally with *elitism* the fittest chromosomes survive from one population to the other.

After selection, *crossover*, i.e., recombination, is applied to randomly paired strings with a probability. Amongst the various crossover schemes are the *one point*, *two point* and the *uniform crossover* schemes [21]. In the one point case a crossover point is identified in the chromosome bit string at random and the portions of chromosomes following the crossover point, in the paired chromosomes, are interchanged. In addition to crossover, mutation is used to prevent permanent loss of any particular bit or allele. Mutation application also introduces genetic diversity. Mutation results in the flipping of bits in a chromosome according to a mutation probability which is generally kept very low.

The chromosome length, population size, and the various probability values in a GA application are referred to as the *GA parameters* in this paper. Selection, crossover, mutation are also referred to as the *GA operators*.

3. TEST DATA GENERATION FOR BRANCH COVERAGE USING GA

Let *P* be the program under test, then a general sequence of steps for test data generation using genetic algorithm is described in Figure 1.

1. Choose an appropriate test adequacy criterion. This in our case is the *branch coverage* criterion.
2. Setup the genetic algorithm.
    a. Select a representation for test data to be input to program *P*.
    b. Define a fitness function.
    c. Instrument the program *P* to create program P_t. The instrumented program P_t is used directly for test data generation.
    d. Select suitable genetic algorithm parameters and operators.
3. Generate test data.
    a. Run the genetic algorithm for test data generation using P_t for fitness computation.
    b. Identify and eliminate infeasibility.
    c. Regenerate test data if necessary.

Figure 1.Test data generation using genetic algorithm

In search based test data generation, test data is generated to meet the requirements of a particular test adequacy criterion. The criterion in our case is the branch coverage criterion. The setup phase begins with the choice of a suitable representation for test data and the identification of a suitable fitness function.

The inputs for one execution of *P*, i.e., a single test data, are represented in a binary string also called a *binary individual*. For instance, if the input to *P* is a pair of integers *x*= (*I1*, *I2*), then this pair is represented as shown in Figure 2. The length of the substrings representing *I1* and *I2* are chosen to represent the largest legal value that can be input to *P*. The length of the complete string is the sum of the lengths of the two substrings.

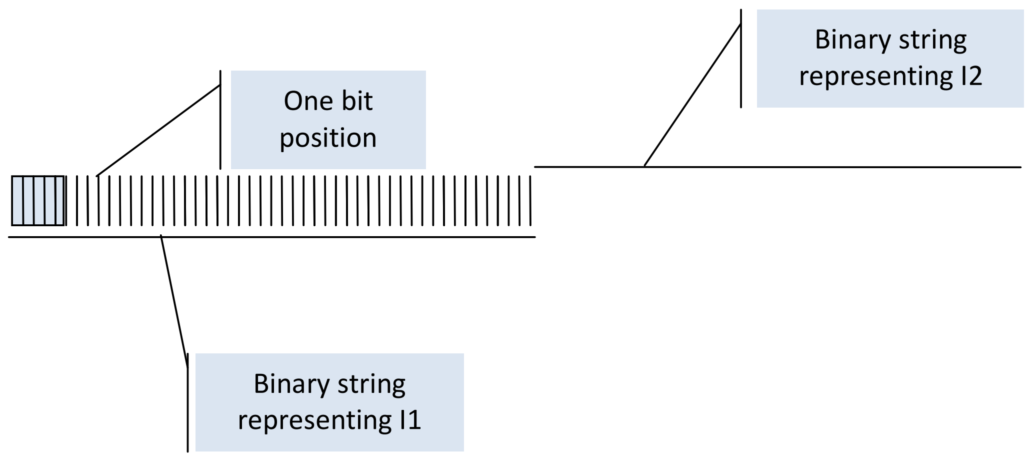

Figure 2. Binary string representation

The fitness of a binary individual is computed as

Fitness (x) = Approximation Level + Normalized Branch Distance

Traditionally, test data generation problem is formulated as a minimization problem as in [12, 13, 14, 19, 20] in which the approach level numbering starts from the target branch and the normalized branch distance is computed as,

$$\text{Normalized Branch Distance} = 1 - 1.001^{-distance}$$

As opposed to this, the test data generation problem can also be formulated as a maximization problem. The definition of approximation level and normalized branch distance is also different from [2] although the basic idea is similar.

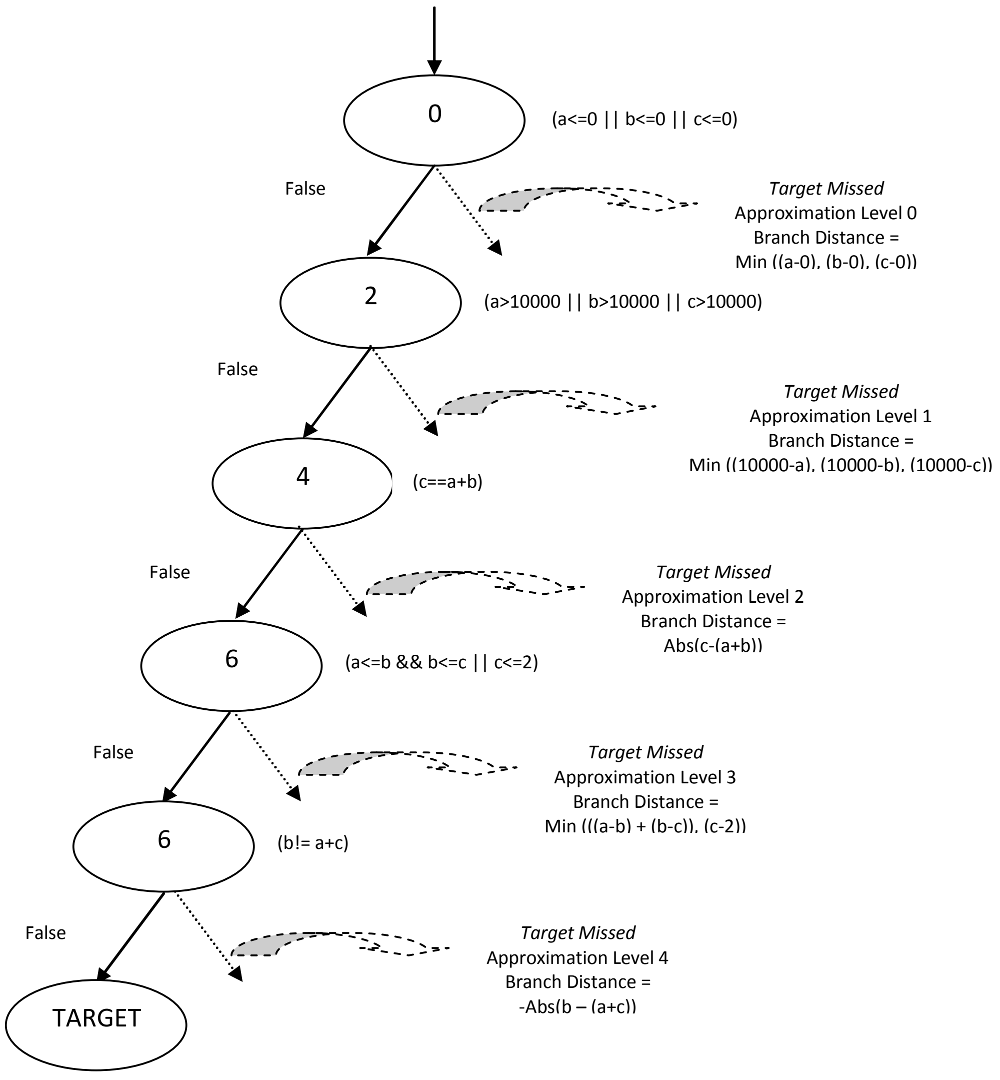

Figure 3.Approximation Level and Branch Distance Computation

The *approximation Level* is a count of the number of predicate nodes in the shortest path from the first predicate node in the flow graph to the predicate node with the *critical branch*- a branchthat leads the target to be missed in a path through the program -as shown in Figure 3.

The *Normalized Branch Distance* is computed according to the formula

$$Normalized\ Branch\ Distance = (1/(1.001^{distance}))$$

where, *distance*, or *branch distance*, as defined in [20, 22], is computed at the node with the critical branch using the values of the variables or constants involved in the predicates used in the conditions of the branching statement.

Table 1 summarizes the computation of distance.

Table 1.Branch distance computation.

	Decision Type	Branch Distance
1	a < b	a – b
2	a <= b	a – b
3	a > b	b – a
4	a >= b	b – a
5	a == b	Abs(a – b)
6	a != b	Abs(a – b)
7	a && b	a + b
8	a || b	min(a , b)

Entries one through five are the same as in [19]. Table 1 also describes the computation of distance in the presence of logical operators AND (&&) and OR (||). In both these cases, the definition takes into account that branch distance is to be minimized whereas the fitness is to be maximized.

In general, in order to generate test data to satisfy the branch coverage criterion using a genetic algorithm, the sequence in which the branches will be selected for coverage must be defined. A chosen branch may become difficult to cover if the corresponding branch predicate is not reached by any of the test data or individuals in the current population. One of the proposals made by Pachauri and Gursaran [23] for sequencing is the *path prefix strategy*. We adopt this strategy for the experiments described in this paper. Further, each time a branch is traversed for the first time, it may be necessary to store the test data that traverse the branch and inject these into the population when the sibling branch is selected for traversal. This is referred to as *memory* and is used in this paper. In order to ensure that individuals reaching the sibling branch of the target are not destroyed by the genetic algorithm operators, elitism is adopted. Up to 10% of fit individuals, with a minimum of one individual, are carried forward to the next generation.Furthermore, it is also possible to initialize the population each time a new branch is selected for coverage or leave it uninitialized. In the experiments described in this paper, the population is not initialized.

Infeasibility may prevent test data from being generated to satisfy *C*. It may be dealt with as follows. If the search is attempting to traverse a particular branch, but is unable to do so over a sufficiently large, predetermined, number of iterations, then the search run is aborted and the branch is manually examined for infeasibility. If the branch is found to be infeasible then it is marked as traversed and the search is rerun.

4. EXPERIMENTAL SETUP

In this section we describe the various experiments carried out to test the performance of test data generation with genetic algorithm.

4.1 Benchmark Programs

Benchmark programs chosen for the experiments have been taken from[11,24]. These programs have a number of features such as real inputs, equality conditions with the AND operator and deeply nested predicates that make them suitable for testing different approaches for test data generation.

- *Line in a Rectangle Problem*: This program takes eight real inputs, four of which represent the coordinates of rectangle and other four represents the coordinates of the line. The program determines the position of the line with respect to the position of rectangle and generates one out of four possible outputs:
    A. The line is completely inside the rectangle;
    B. The line is completely outside the rectangle;
    C. The line is partially covered by the rectangle; and
    D. Error: The input values do not define a line and/or a rectangle.

  The maximum nesting level is 12. In total this program's CFG has 54 nodes and 18 predicate nodes.

- *Number of Days between Two Dates Problem*: This program calculates the days between two given dates of the current century. It takes six integer inputs- three of which represent the first date (day, month, and year) and other three represents the second date (day, month, and year). The CFG has 43 predicate nodes and 127 nodes

- *Calday:* This routine returns the Julian day number. There are three integer input to the program. First input represent month, second represent day and the third represent the year. It's CFG has 27 Nodes with 11 predicate nodes. It has equality conditions, remainder operator. The maximum nesting level is 8.
- *Complex Branch*:It accepts 6 short integer inputs. In this routine there are some complex predicate conditions with relational operators combined with complex AND and OR conditions, it also contains while loops and SWITCH-CASE statement. Its CFG contains 30 nodes

- *Meyer's Triangle Classifier Problem*: This program classifies a triangle on the basis of its input sides as non triangle or a triangle, i.e., isosceles, equilateral or scalene. It takes three real inputs all of which represent the sides of the triangle. It's CFG has 14 Nodes with 6 predicate nodes. The maximum nesting level is 5. It has equality conditions with AND operator, which make the branches difficult to cover.

- *Sthamer's Triangle Classifier Problem*: This program also classifies a triangle on the basis of its input sides as non triangle or a triangle that is isosceles, equilateral, right angle triangle or scalene. It takes three real inputs; all of them represent the sides of the triangle but with different predicate conditions. It's CFG has 29 Nodes with 13 predicate nodes. The maximum nesting level is 12. It has equality conditions with AND operator and complex relational operators.

- *Wegener'sTriangle Classifier Problem*: This program also classifies a triangle on the basis of its input sides as non triangle or a triangle that is isosceles, equilateral, orthogonal or obtuse angle. It takes three real inputs; all of them represent the sides of the triangle but with different predicate conditions. It's CFG has 32 Nodes with 13 predicate nodes. The maximum nesting level is 9.
- *Michael's Triangle Classifier Problem*: This program also classifies a triangle on the basis of its input sides as non triangle or a triangle that is isosceles, equilateral or scalene. It takes three real inputs; all of them represent the sides of the triangle but with different predicate conditions. It's CFG has 26 Nodes with 11 predicate nodes. The maximum nesting level is 6.

4.2 GA Operator and Parameter Settings

Table 2 lists the various operator and parameter settings for the genetic algorithm used in this study.

Table 2.Operator and Parameter Settings

	Parameter/ Operator	**Value**
1	Population Size	6, 10, 16, 20, 26, …, 110.
2	Crossover type	Two point crossover
3	Crossover Probability	1.0
4	Mutation Probability	0.01
5	Selection Method	Binary tournament
6	Branch Ordering Scheme	Path Prefix Strategy
7	Fitness Function	As described in Section 3.
8	Population Initialization	Initialize once at the beginning of the GA run
9	Population Replacement Strategy	Elitism with upto 10% carry forward
10	Maximum Number of Generations	10^7
11	Memory	Yes

5. RESULTS

For each population size, hundred experiments were carried out and the following statistics were collected:

- Mean number of generations. It may be noted that the termination criterion for each experiment is either full branch coverage or 107 generations whichever occurs earlier. The number of generations to termination over hundred experiments is used to compute the mean. The mean does not tell us if all the branches were covered.
- Mean percentage coverage achieved.

Additionally ANOVA was carried out using SYSTAT 9.0 to determine significant difference in means.

In all the experiments full (100%) coverage was achieved for all population sizes, for all benchmark programs and for both maximization and minimization approaches. This implies that the differentiating factor would have to be the difference in the mean number of generaitons.
Figure 4 and Figure 5 plot the mean number of generations for both the maximization and minimization approach. Table 3 summarizes the results of ANOVA with F and p values. Considering a significance level of 0.05, it can be seen that the difference for all the benchmark programs is not significant except for some isolated cases which are not generalizable.

Further analysis in our case shows that with the path prefix strategy and memory, individuals are present in each generation that cause a traversal of the sibling branch of the target. This coupled with elitism may actually speed up the test data discovery process. It may be interesting to check if in the absence of a branch ordering strategy, memory and elitism, is the performance of the two approaches is comparable.

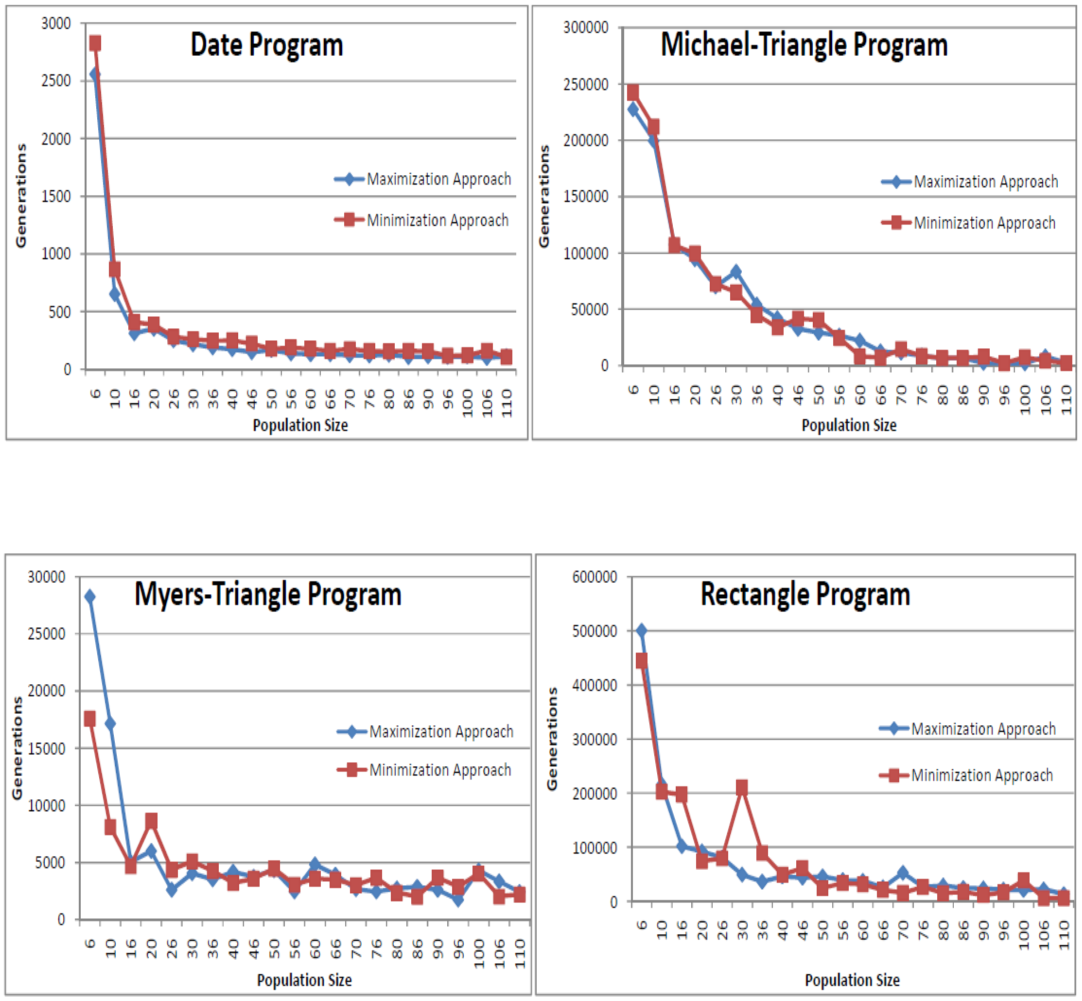

Figure 4 Plots of Mean Number of Generations for some Benchmark Programs

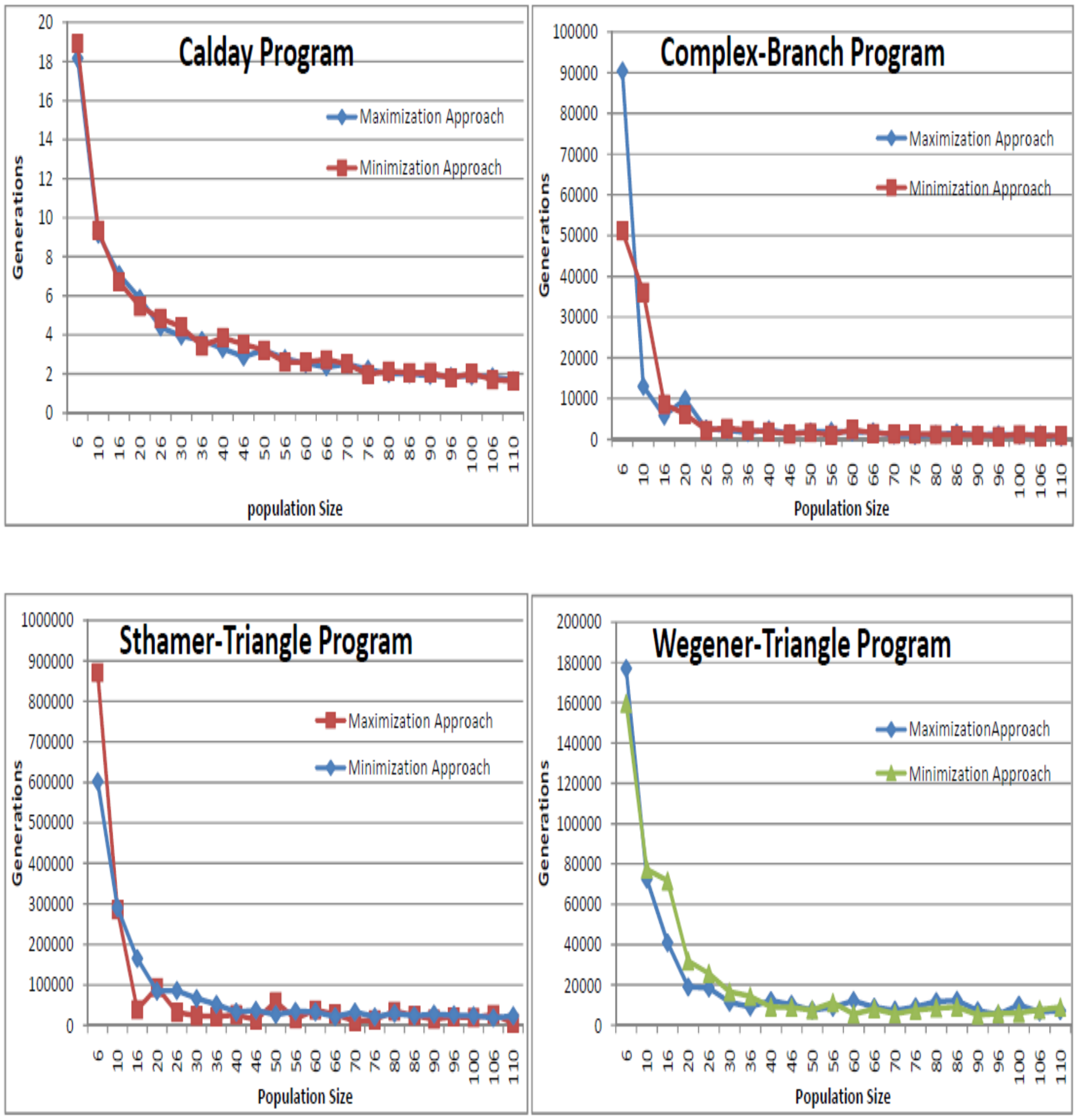

Figure 5 Plots of Mean Number of Generations for some Benchmark Programs

Benchmark Program		ANOVA test															
		Calday Program		Complex-Branch Program		Date Program		Michael-Triangle Program		Myers-Triangle Program		Rectangle Program		Sthamer-Triangle Program		Wegener-Triangle Program	
		F value	P value	F value	P value	F value	P value	F value	P value	F value	P value	F value	P value	F value	P value	F value	P value
Population Size	**6**	0.091	0.763	0.753	0.386	0.02	0.888	0.007	0.933	2.227	0.137	0.073	0.788	1.324	0.251	0.046	0.831
	10	0.028	0.866	2.813	0.095	0.696	0.405	0.005	0.942	5.244	0.023	0.01	0.921	0.003	0.955	0.06	0.808
	16	0.264	0.608	1.196	0.275	3.645	0.058	0.0	0.997	0.063	0.803	0.805	0.371	19.198	0.0	1.045	0.308
	20	0.678	0.411	0.682	0.41	0.513	0.475	0.003	0.959	1.795	0.182	0.269	0.605	0.135	0.713	2.945	0.088
	26	1.21	0.291	0.189	0.664	0.818	0.367	0.001	0.971	3.671	0.57	0.002	0.961	11.193	0.001	1.793	0.182
	30	1.447	0.231	0.27	0.604	2.028	0.156	0.079	0.778	0.804	0.371	3.701	0.356	3.244	0.073	1.737	0.189
	36	0.676	0.412	1.318	0.252	4.311	0.039	0.03	0.863	0.378	0.539	1.278	0.26	8.636	0.004	2.409	0.122
	40	2.627	0.107	0.484	0.487	7.526	0.007	0.043	0.836	0.726	0.395	0.033	0.856	0.434	0.511	1.016	0.315
	46	5.333	0.022	0.119	0.731	9.734	0.002	0.067	0.796	0.027	0.869	0.825	0.365	6.326	0.013	0.19	0.664
	50	0.001	0.974	0.085	0.771	0.294	0.589	0.039	0.843	0.013	0.91	1.65	0.2	1.423	0.234	0.01	0.922
	56	0.508	0.477	1.866	0.171	5.479	0.02	0.007	0.934	0.596	0.441	0.083	0.773	4.498	0.035	0.673	0.413
	60	0.074	0.785	0.618	0.433	4.997	0.027	0.809	0.37	1.702	0.302	0.12	0.729	0.141	0.708	3.731	0.055
	66	1.758	0.186	0.597	0.441	2.587	0.109	0.296	0.587	0.192	0.662	0.264	0.608	0.681	0.41	0.056	0.813
	70	0.006	0.937	0.579	0.447	7.706	0.006	0.074	0.786	0.11	0.74	3.305	0.083	10.215	0.002	0.908	0.342
	76	0.995	0.32	1.121	0.291	3.356	0.068	0.001	0.977	0.879	0.35	0.0	0.987	1.679	0.197	0.361	0.549
	80	0.305	0.582	0.0	0.992	1.89	0.171	0.01	0.921	0.33	0.566	1.621	0.204	0.121	0.728	0.695	0.405
	86	0.084	0.772	1.483	0.225	8.597	0.004	0.0	0.992	0.688	0.408	0.273	0.602	0.001	0.971	0.81	0.369
	90	0.345	0.558	0.007	0.933	6.33	0.013	1.268	0.262	0.682	0.41	2.21	0.139	4.008	0.047	1.345	0.248
	96	0.115	0.734	2.19	0.14	0.506	0.478	0.0	0.987	1.365	0.244	0.298	0.586	0.166	0.684	0.234	0.629
	100	0.342	0.559	0.139	0.709	0.273	0.602	1.125	0.29	0.028	0.867	0.493	0.483	0.418	0.519	1.64	0.202
	106	0.371	0.543	2.304	0.131	8.216	0.005	0.647	0.422	0.669	0.415	3.546	0.061	0.7	0.404	0.241	0.624
	110	0.018	0.892	2.732	0.1	0.148	0.701	0.008	0.927	0.048	0.827	2.155	0.144	27.076	0.0	0.819	0.367

Table 3Results of test of ANOVA

6. CONCLUSION

In search based test data generation, the problem of test data generation is reduced to that of function minimization or maximization. Traditionally, for branch testing, the problem of test data generation has been formulated as a minimization problem. In this paper we have defined an alternate maximization formulation and experimentally compared it with the minimization formulation. We have used a genetic algorithm as the search technique and in addition to the usual genetic algorithm operators we have also employed the path prefix strategy as a branch ordering strategy and memory and elitism. Results indicate that there is no significant difference in the performance or the coverage obtained through the two approaches and either could be used in test data generation if coupled with the path prefix strategy, memory and elitism.

It may be interesting to check if in the absence of a branch ordering strategy, memory and elitism, is the performance of the two approaches is comparable.

ACKNOWLEDGEMENTS

This work was supported by the UGC Major Project Grant F.No.36-70/2008 (SR) for which the authors are thankful.

REFERENCES

[1] M. Harman & B. Jones, (2001), "Search Based Software Engineering", Journal of Information and Software Technology, vol. 43, No. 14,pp833-839.

[2] P. McMinn, (2004), "Search-Based Software Test Data Generation: A Survey," Software Testing, Verification and Reliability, vol. 14, No.2, pp105-156.

[3] M. Harman, A. Mansouri, and Y. Zhang, (2009),"Search based software engineering: A comprehensive analysis and review of trends techniques and applications", Technical Report TR-09-03, Department of Computer Science, King's College London.

[4] M. Harman and A. Mansouri, (2010),"Search Based Software Engineering: Introduction to the Special Issue of the IEEE Transactions on Software Engineering,"IEEE Transactions onSoftware Engineering, vol.36, No.6, pp737-741.

[5] S. Ali, L.C. Briand, H. Hemmati, R.K. Panesar-Walawege, (2010), "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation",IEEE Transactions onSoftware Engineering, vol.36, No.6, pp742-762.

[6] M. Harman and P. McMinn, (2010), "A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search",IEEE Transactions onSoftware Engineering, vol.36, No.2, pp226-247.

[7] H. Zhu, A. V. PatrickHall, and H. R. John, (1997), "Software Unit Test Coverage and Adequacy",ACM Computing Surveys, vol. 29, No. 4, pp366–427.

[8] M. Harman, L. Hu, R.M. Hierons, C. Fox, S. Danicic, A. Baresel, H. Sthamer, and J. Wegener, (2002). "Evolutionary Testing Supported by Slicing and Transformation", In Proceedings of IEEE International Conference on Software Maintenance, Montreal, Canada (ICSM '02), pp 285–285.

[9] C. Michael, G. McGraw and M. Schatz, (2001), "Generating Software Test Data by Evolution",IEEE Transaction on Software Engineering, vol. 27, pp1085-1110.

[10] K. Agarwal and Gursaran, (2010), "Towards software test data generation using discrete quantum particle swarm optimization. In Proceedings of the 3rd India software engineering conference (ISEC '10). ACM, New York, NY, USA, pp65-68.

[11] R. Blanco, J. Tuya and B. Adenso-Díaz. (2009). "Automated test data generation using a scatter search approach" Inf.Softw. Technol. Vol. 51, No. 4, pp.708-720.

[12] B.F. Jones, D. Eyres, and H. Sthamer, (1998), "A Strategy for using Genetic Algorithms to Automate Branch and Fault-based Testing",Computer Journal, vol. 41,No. 2, pp98–107.

[13] J. Wegener, A. Baresel and H. Sthamer (2001), "Evolutionary Test Environment for Automatic Structural Testing",Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms, vol. 43,No. 14, pp841–854.

[14] E. D´ıaz, J. Tuya and R. Blanco (2003), "Automated Software Testing using a Metaheuristic Technique based on Tabu Search". In Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE '03), Montreal, Canada, pp310–313.

[15] P. McMinn, D. Binkley and M. Harman, (2009), "Empirical evaluation of a nesting testability transformation for evolutionary testing", ACM Trans. Softw. Engg. Methodol. Vol. 18, No. 3.

[16] M. Harman, (2008), "Testability Transformation for Search-Based Testing", In Keynote of the 1st International Workshop on Search-Based Software Testing (SBST) in conjunction with ICST 2008, Lillehammer, Norway.

[17] M.A. Ahmed and I. Hermadi (2008), "GA-based Multiple Paths Test Data Generator", Computers & Operations Research, vol. 35, No. 10, pp3107–3124.

[18] Y. Chen, Y. Zhong, T. Shi, and J. Liu (2009), "Comparison of Two Fitness Functions for GA-Based Path-Oriented Test Data Generation", In Proceedings of the 2009 Fifth International Conference on Natural Computation (ICNC '09), IEEE Computer Society, Washington, DC, USA, Vol. 4, pp177-181.

[19] B. Korel, B. (1990),"Automated Software Test Data Generation". In Transactions on Software Engineering, SE vol. 16, No. 8, pp870–879.

[20] A. Baresel, H. Sthamer and M. Schmidt. (2002), "Fitness Function Design to Improve Evolutionary Structural Testing", Inproceedings of the 2002 Conference on Genetic and Evolutionary Computation (GECCO '02), New York, USA, pp1329–1336.

[21] D.E. Goldberg, (1989), Genetic Algorithms in search optimization & machine learning, Pearson Education Asia.

[22] P. McMinn and M. Holcombe, (2006), "Evolutionary Testing Using an Extended Chaining Approach",Evolutionary Computation, vol.14, No.1, pp. 41-64.

[23] A. Pachauri and Gursaran, (2011), "Software Test Data Generation using Path Prefix Strategy and Genetic Algorithm", In Proc. Of the International Conference on Science and Engineering (ICSE 2011),ISBN: 978-981-08-7931-0, pp131-140. Available on http://rgconferences.com/proceed/icse11/pdf/152.pdf

[24] E. D´ıaz, J. Tuya and R. Blanco and J. J. Dolado (2008), "A Tabu Search Algorithm for Structural Software Testing", Computers & Operations Research, vol. 35, No. 10, pp3052–3072.