International Journal of Computer Science and Communication Vol. 3, No. 1, January-June 2012, pp. 259-262

Application of Big Bang Big Crunch Algorithm to Software Testing

Surender Singh[1], Parvin Kumar[2]
[1]CMJ University, Shillong, Meghalya, (INDIA)
[2]Meerut Institute of Science & Technology, Meerut, UP (INDIA)
surendahiya@gmail.com, pk223475@yahoo.com

Abstract: This paper presents a Big Bang Big Crunch concept based search algorithm for automatic generation of structural software tests. Test cases are symbolically generated by measuring fitness of individuals with the help of branch distance based objective function. Evaluation of the test generator was performed using ten real world programs. Some of these programs had large ranges for input variables. Results show that the new technique is a reasonable alternative for test data generation, but doesn't perform very well for large inputs and where constraints are having many equality constraints.

Keywords: Big Bang Big Crunch, Symbolic Execution, Software Testing.

1. Introduction

Highly non-linear structure of software presents a formidable task to search algorithms for finding optimal and efficient test data from a complex, discontinuous, non-linear inputs' search space. For such environment, the search algorithm must have both types of search capabilities; local as well as global. Most successful search algorithm class is based on metaheuristic techniques such as genetic algorithm (GA), simulated annealing (SA), tabu search, ant colony optimization (ACO), particle swarm optimization (PSO) etc. Xanthakis [4] first time applied GA for automatic test case generation. Pargas *et al* [6] proposed a GA based testing technique where number of executed control dependent nodes of the target node decides the fitness of solutions population. Wagener *et al* [1] logarithmized objective function to provide better guidance for its GA based test data generator. Watkins [7] and Ropar [10] used coverage based criteria for assessing the fitness of individuals in their GA based test generator. Lin and Yeh [13] used hamming distance based metric in objective function of their GA program to identify the similarity and distance between actual path and the already selected target path for traversal in dynamic testing. Michal *et al* [9] have used GA based testing method for covering all the conditions on a path for c and c++ programs.

Tracey [12] constructed a SA based test data generator for safety critical system by using a hybrid objective function, which includes both concept; branch distance and number of executed control-dependent-nodes. Diaz *et al* [5] developed a tabu search based test data generator, which maintains a search list also called as tabu list. It uses neighbourhood information and backtracking for solving local optima. Ayari et al [11] proposed an evolutionary approach based on ACO to reduce the cost of test data generation in the context of mutation testing. This ACO based approach is enhanced by a probability density estimation technique in order to provide better guidance to the search for continuous input parameters. Windisch *et al* [2] have reported the application of particle swarm optimization technique for test data generation for dynamic testing.

Another recent search algorithm in soft computing category is Big Bang Big Crunch (BBBC) algorithm, which simulates the energy stabilization in the universe. Although this technique has been successfully employed on scores of engineering applications such as mechanical engineering and civil engineering, but its applicability in testing domain is still unexplored.

2. Methodology

2.1 Software testing as Search problem

For software testing purpose, as solution lies in searching inputs, every possible set of inputs represent the global population in search algorithm and selected inputs from this global set are represented by individuals in the population. We have chosen the all-path coverage criterion for our experimentation. It involves generation of test data for a target feasible path in such a way that on executing program, it covers all branches on that path by satisfying all the condition(s) at branch nodes of a particular path. Consequently, the problem of path testing can be formulated simply as constraint satisfaction problem. Figure 1 shows the mechanism of symbolic path test data generator.

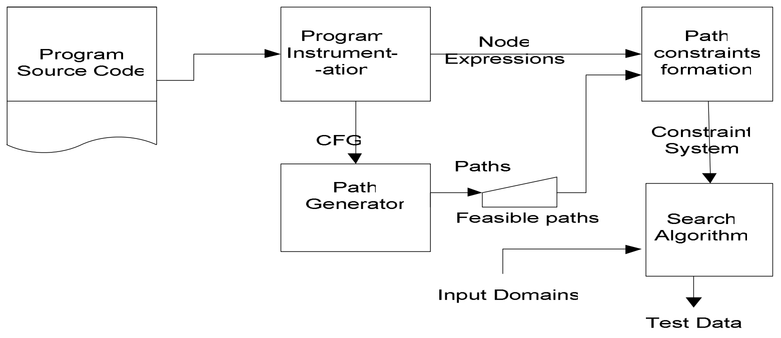

Fig 1. Automatic Symbolic Path Test Data Generator

2.2 Search Algorithm

The Big Bang and Big Crunch theory is introduced by Erol and Eksin [8], which is based upon the analogy of universe evolution where two phase of evolution is represented by expansion (Big Bang) & contraction (Big crunch). This algorithm has a low computational time and high convergence speed. In fact, the Big Bang phase dissipates energy and produces disorder and randomness. In the Big Crunch phase, randomly distributed particles (which form the solution when represented in a problem) are arranged into an order by way of a convergence operator "center of mass". The Big Bang–Big Crunch phases are followed alternatively until randomness within the search space during the Big Bang becomes smaller and smaller and finally leading to a solution. Below is given the algorithm for the BBBC algorithm in steps.

1. Create random population of solution.
2. Evaluate Solutions.
3. The fittest individual can be selected as the center of mass.
4. Calculate new candidates around the center of mass by adding or subtracting a normal random number whose value decreases as the iterations elapse.
5. The algorithm continues until predefined stopping criteria has been met

Fig 2. BBBC Search Algorithm Workflow

2.3 Fitness Function

For path testing criterion, in order to traverse a feasible path, the control must satisfy the entire branch predicates, which falls on that particular path. In our experimentations, we have used symbolic execution technique of static structural testing. So, corresponding to each path a compound predicate (CP) is made by '***anding'*** each branch predicate of the path. The CP must be evaluated to true by a candidate solution in turn to become a valid test case. The BBBC generates population of candidate solutions and these are used to evaluate CP. If predicate is not evaluated to true by an individual then all the constraints of particular path are split into distinct predicate (DP) and one by one each DP is evaluated by taking values of its operands from candidate solution. A DP is that one, which contains only one operator (a constraint with modulus operator is exception) and can be expressed in form of expression $A \; op \; B$ where A and B are LHS and RHS of expression made of one or more operand(s) and op is relational operator. If DP is satisfied then no penalty is imposed to candidate solution, otherwise candidate solution is penalized on the basis of branch distance concept rules as shown in table 1 which is also recommended by Watkins *et al* [3] for static structural testing.

Table1. Branch Predicate based Fitness function

Violated predicate	**Penalty to be imposed in case predicate is not satisfied**	**Violated predicate**	**Penalty to be imposed in case predicate is not satisfied**
$A < B$	$A – B+ \zeta$	$A \geq B$	$B – A$
$A \leq B$	$A – B$	$A = B$	Abs(A – B)
$A > B$	$B – A+ \zeta$	$A \neq B$	ζ – abs(A – B)
ζ is a smallest constant of operands' universal domains.			

After this integrated fitness due to whole of CP is determined by adding penalty values of two DPs, if they are connected by a conditional ***'and'*** operator. If two DPs are connected by a conditional ***'or'*** operator then minimum penalties of two DPs is considered for the evaluation of whole CP fitness. If integrated fitness is zero then CP is called evaluated or satisfied by the individual whose values are replaced in CP and search process for particular path is terminated otherwise search is allowed to proceed further.

3. Experimental Setup and Results

The BBBC algorithm is implemented using MATLAB programming environment. The performance of the algorithms is measured using average test cases generated per path (ATCPP) and average percentage coverage (APC) metrics. Experiment is conducted 10 times for averaging results. In each attempt, BBBC is iterated for 100 generations for each of 10 runs. In each run, except for the first run, first-generation population is seeded with the best solution from the previous run. This is done to check premature convergence of population. Total number of real encoded individuals in each population is 30. If a solution is not found within all runs that generates total 30,000 invalid test cases then it is declared that the test case generation has failed for that particular attempt. This value has been obtained by multiplying total number of runs, generations and number of individual in each population. An invalid test case is a solution, which does not qualify to become a test case.
For comparing the BBBC method with random test generator, we have chosen two small but frequently used real world programs for test data generation activity. First program or test object is Triangle classifier (TC) which accepts three inputs as sides of a triangle and then decides whether these sides form a triangle and if yes then of what type. This program is of 35 lines and its cyclomatic complexity is 7 with nesting level 5. Second test object is called as line-rectangle classifier (LRC) program identifies whether a line cuts a rectangle or lies completely outside or lies completely inside of the rectangle. In this program total eight inputs are entered; four for co-ordinates of rectangle and other four inputs to define the line. Some of the nodes in CFG of this program have very high level of nesting of as high as 12 with overall cyclomatic complexity of 19.
Table 2 presents the comparison between random test data generator and BBBC based test data generator. The BBBC is able to generate test cases for all paths except in cases of large domain. This shows the inapplicability of BBBC for large domains of inputs. It also fails to generate test data for TC (small domain) for a path in which it has to prove triangle as equilateral. So, we can also conclude that search algorithm performance is affected by the number of equality constraints the target path involving.

Table 2. ATCPP and APC for Test Objects

Name of Program	Random Tester		BBBC Tester	
	ATCPP	APC	ATCPP	APC
TC (small Domain)	10493	47.86%	829	100%
TC (Large Domain)	17147	42.85%	7564	83.32%
LRC (small Domain)	13901	55%	2543	97.23%
LRC (Large Domain)	14159	53	11765	74.42%

Further to prove the applicability and scalability of BBBC tester, we have chosen 8 real world programs for test data generation activity. These are called test objects here and brief explanation for each test object is given below.

Table 3. Test Objects' characteristics and corresponding test coverage result

Name of Program	**Lines Code**	**Cyclomatic Complexity**	**Decision Nodes**	**Nesting Level**	**Total Paths in CFG**	**Feasible Paths**	**ATCPP**	**APC**
DBTD	123	26	22	05	1643	566	378	100
A2F	48	15	14	07	910	568	5743	100
BS	23	05	04	03	124	62	28021	48
REM	35	10	8	04	22	22	1652	100
BUB	21	04	03	03	121	31	413	100
QUAD	24	06	05	03	06	06	2247	100
MINMAX	27	04	03	03	121	121	721	100
ISPRIME	16	03	02	02	10	08	47	100

Table 3 presents the results of testing effort on 8 testing objects selected for experimentation in last two columns. Test cases for TC and LRC programs are generated from inputs by taking small as well as large domain of size 10^4 and 10^8 respectively for each path. The BBBC is able to generate test cases for all paths except in cases of TC (small as well as large domain), LRC (large domain) and BS program. This shows the inapplicability of BBBC for large domains of inputs. It also fails to generate test data for TC (small domain) frequently for a path in which it has to prove triangle as equilateral. Thereby, we can also conclude that search algorithm performance is affected by the number of equality constraints the target path involving. Other than these, the binary search is the only program for which BBBC fails to generate test cases. This may be due to requirement of inputting variable array to satisfy the boundary cases. Although we have taken a fixed size array of size 80 but its size is varied by taking an external variable 'n' during experimentation. We have used the same approach for A2F and BUB programs but in these, boundary cases are not required to be satisfied.

4. Conclusion

We have proposed a new search algorithm for the generation of test cases. Experimentations are done on two real world problems. Static testing based symbolic execution method has been used in which first, target path is selected from CFG of program and then inputs are generated using the BBBC method to satisfy composite predicate corresponding to the path. It has been observed that the BBBC method is better alternative than random testing.

References

[1] J. Wegener, A. Baresel and H. Sthamer, "Evolutionary test environment for automatic structural testing," Information and Software Technology, 2001; Vol. 43, pp. 841–54.

[2] A. Windisch, S. Wappler and J. Wegener, "Applying particle swarm optimization to software testing," Proc. conference on Genetic and evolutionary computation GECCO'07, London, England, United Kingdom, July 2007, pp. 7–11.

[3] T. Schmickl, R. Thenius and K. Crailsheim "Simulating swarm intelligence in honey bees: foraging in differently fluctuating environments," *GECCO'05*, Washington, DC, USA, 2005, pp. 273-274.

[4] S. Xanthakis, C. Ellis, C. Skourlas, A. Gall, S. Katsikas and K. Karapoulios, "Application of genetic algorithms to software testing," In The fifth international conference on *software engineering* 1992; pp. 625–36.

[5] E. Díaz,T. Javier, B. Raquel and J. Jose, "A tabu search algorithm for structural software testing," Computers and Operations Research (2007), doi:10.1016/j.cor. 2007.01.009

[6] R. Pargas, M. Harrold and R. Peck, "Test-data generation using genetic algorithms," Journal of Software Testing, Verification and Reliability 1999; 9(4): pp. 263–82.

[7] A. Watkins, "The automatic generation of test data using genetic algorithms," In The fourth software quality conference 1995; 2: pp. 300–309.

[8] Erol OK and, Eksin I, A new optimization method: Big Bang-Big Crunch, *Advances in Engineering Software*, 37, 2006, 106-111.

[9] C. Michael, G. McGraw and M. Schatz, "Generating software test data by evolution," IEEE Transactions on Software Engineering 2001; 27(12): pp. 1085–1110.

[10] M. Roper, "Computer aided software testing using genetic algorithms," *In 10th International Software Quality Week*, San Francisco, USA, 1997.

[11] K. Ayari, S. Bouktif and G. Antoniol, "Automatic mutation test input data generation via ant colony," GECCO'07, July 7–11, 2007, London, England, United Kingdom.

[12] N. Tracey, A Search-Based Automated Test-Data Generation Framework for Safety Critical Software, PhD thesis, University of York, 2000.

[13] J. Lin and P. Yeh, "Automatic test data generation for path testing using Gas" Information Sciences 2001; 131: pp. 47–64.