Chinese Journal of Electronics
Vol.23, No.3, July 2014

An Improved Genetic Algorithm for Test Cases Generation Oriented Paths*

MEI Jia and WANG Shengyuan

(*Department of Computer Science and Technology,Tsinghua University, Beijing 100084, China*)

Abstract — This paper discusses a method that can automatically generate test cases for selected paths using a special genetic algorithm. The special algorithm is called Queen-bee evolutionary genetic algorithm(QBEA). In this algorithm, sequences of operators iteratively executes for test cases to evolve to target paths. The best chromosome called queen among the current population is crossover with drones selected according to a certain crossover probability, which enhances the exploitation of searching global optimum. A comparative experiment results prove that the proposed method is actually a great improvement in optimization efficiency and optimization effect.

Key words — Software testing, Test data generation, Queen-bee evolutionary genetic algorithm.

I. Introduction

Computers and programs have been expanded in many sections of our life, such as in electronic transactions, media, transportation, medication and health. Because various aspects of our everyday working activities or even our own lives depend on software reliability, how to ensure the dependability of software is very important. Software testing is a way for verifying the correctness and appropriateness of a software system, or, alternatively, for ensuring that a program meets its specifications[1−3].

The time and effort usually spent on software testing may be greater than the overall system implementation cost. The task of selecting test data to exercise these components is extremely complex and accounts for a large portion of the overall cost of testing. It requires a careful analysis of the program code, a good deal of mental effort, in addition to knowledge of the underlying concepts of the criteria. The automation of this task is crucial[4,5] but infeasible in general cases, since the problem is equivalent to the undecidable "halting problem"[6].

Techniques and tools have been developed to bring some degree of automation to test data generation. Generally the techniques are either random[7] or emphasize a test philosophy (functional, structural or fault-based). Structural based test data generators are often classified as: (1) path oriented method[8−12] in which a program path is selected and an input data set is generated to execute the path. (2) goal oriented method[13] in which an input data set is generated to execute a statement irrespective of the path taken (some authors use "goal" with a broader meaning, such as dataflow associations or branches).

Automatic test data generation is a crucial problem in software testing and its implementation can not only significantly improve the effectiveness and efficiency but also reduce the high cost of software testing[14,15]. Particularly, it is notable that various structural test data generation problem can be transformed into a path-oriented test data generation problem. Furthermore, path testing strategy can detect almost 65 percent of errors in program under test[16].

The basic concepts of genetic algorithms were developed by Holland. The genetic algorithms include a class of adaptive searching techniques which are suitable for searching a discontinuous space. Genetic algorithms have been used to find automatically a program's longest or shortest execution times. In the paper about testing real-time systems using genetic algorithms[17], J. Wegener, *et. al.*[18] investigated the effectiveness of genetic algorithms to validate the temporal correctness of real-time systems by establishing the longest 'and the shortest execution times. The authors declared that an appropriate fitness function for the genetic algorithms is found, and the fitness function is to measure the execution time in processor cycles. Their experiments using genetic algorithms on a number of programs have successfully identified new longer and shorter execution times than those had been found using random testing or systematic testing. Genetic algorithms have also been used to search program domains for suitable test cases to satisfy the all-branch testing. In their papers about automatic structural testing using genetic algorithms, B. F. Jones, *et. al.*[19,20] showed that appropriate fitness functions are derived automatically for each branch pred-

*Manuscript Received Dec. 2013; Accepted Feb. 2014. This work is supported in part by National Natural Science Foundation of China (No.61272086, No.61170051), Guangxi Nature Science Fund (No.2012jjBAG0074), Guangxi Education Department Scientific Research Items (No.201106LX840), Guangxi Key laboratory of hybrid computation and IC design analysis Open Foundation (No.2012HCIC05), and National Social Science Fund (No.11CTQ008).

icate. All branches were covered with two orders of magnitude fewer test cases than random testing. Refs.[21–23] using general genetic algorithms to generate path oriented test data.

In this paper, we use an improved genetic algorithm: queen-bee evolutionary genetic algorithm, to explore the program's input domain, searching for an input data set that executes the desired path. Queen-bee evolution is similar to nature in that the queen-bee, the fittest individual in a generation, crossbreeds with the other bees selected as parents by a selection algorithm. Experimental results with typical problems show that the queen-bee evolution algorithm can enhance the exploration efficiency and make the generated set of test cases quickly approach the global optimum.

II. Concepts of Path Oriented Program Modeling

Path testing is one kind of software testing methodologies which searches the program domain for suitable test cases such that after executing the program with the test cases, a predefined path can be reached. Since a program may contain an infinite number of paths, it is only practical to select a specific subset path to perform path testing. To illustrate our method, related formal definition for path oriented program modeling is given as follows:

Definition 1 A control flow graph is a directed graph G augmented with a unique entry node $START$ and a unique exit node $STOP$ such that each node in the graph has at most two successors. We assume that nodes with two successors have attributes "T" (True) and "F" (False) associated with the outgoing edges in the usual way, We further assume that for any node N in G there exists a path from $START$ to N and a path from N to $STOP$.

Definition 2 A node V is post-dominated by a node W in G if every directed path from V to $STOP$ (not including V) contains W. (Note that this definition of post-dominance does not include the initial node on the path. In particular, a node never post-dominates itself.)

Definition 3 Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X iff

(1) There exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y.

(2) X is not post-dominated by Y.

If Y is control dependent on X then X must have two exits. Following one of the exits from X always results in Y being executed, while taking the other exit may result in Y not being executed. Condition (1) can be satisfied by a path consisting of a single edge. Condition (2) is always satisfied when X and Y are the same node. This allows loops to be correctly accommodated by our definition.

When applied to a loop in the control flow graph, our definition of control dependence determines a Strongly connected region (SCR) of control dependences whose nodes consists of predicates that determine an exit from the loop. The other nodes in the control flow graph loop not in the SCR of control dependences lie on some path of control dependence edges from a node in the SCR. Intuitively, these correspond to the body of the loop.

We show in Fig.1 a program example (a) with its corresponding control flow graph (b) and control dependence sub graph (c). Nodes 1, 2 and 6 are all controlled dependent on program entry; They could thus be executed in parallel, if no data dependences exist between them. This relationship is not immediately apparent from the control flow graph. Control dependences to nodes at the operator level are determined by the corresponding control dependences at the statement and predicate level, which are simply extended to all contained operators.

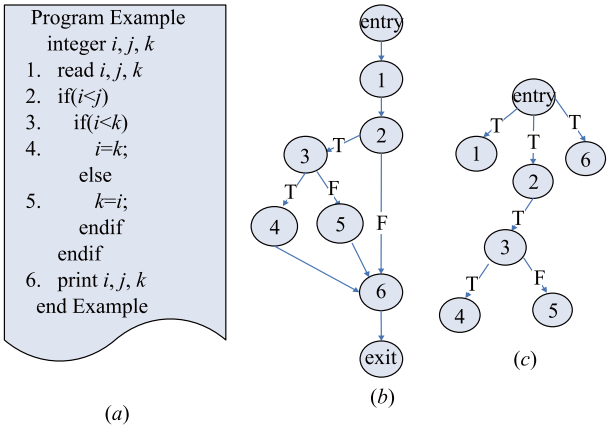


Fig. 1. Program with its corresponding control flow graph and control dependence sub graph

III. Queen-Bee Evolutionary Genetic Algorithm

The Queen-bee evolution based on genetic algorithm (QEGA) is described as follows:

QEGA

```
//t: time //
//n : population size //
//P : populations //
//γ :rate of crossing over with queen-bee//
//p_m : mutation probability //
//I_q : a queen-bee//
//I_m : selected bees //
01 t ← 0
02 initialize P(t)
03 while (not termination-condition)
04 do
05 t ← t + 1
06 producing new generation of populations by crossover
between I_q(t) and I_m
//the amount of selected bodies is γ * (n − 1)//
07 do mutation with p_m (other bodies)
08 select a body whose fitness is maximum, Remember to F
09   if fitness of F >fitness of I_q(t)
10     Fitness of I_q(t + 1) =fitness of F
11     else
12       I_q(t + 1) = I_q(t), F replace the body whose fitness
is minimum
13     end if
14 end
```

Queen-bee evolution is similar to nature in that the queen-bee, the fittest individual in a generation, crossbreeds with the

other bees selected as parents by a selection algorithm. From the above algorithm flow, we can find some differences between conventional genetic algorithm and QEGA. Queen-bee evolution has two major differences. First, the parents $P(t)$ in the original algorithm are composed of the n number of individuals selected by a selecting algorithm such as a roulette wheel selection, while parents $P(t)$ in queen-bee evolution consist of the $\gamma \times (n-1)$ number of couples of a queen-bee $I_q(t-1)$, where $q = \arg\max\{(f_t(t-1), 1 \leqslant i \leqslant n\}$ and each selected bee $I_m(t-1)$ is selected by selection algorithm. Secondly, $(1-\gamma) \times (n-1)$ individuals in the original algorithm are mutated with mutation probability p_m. Thirdly, the queen-bee selection is also different from general genetic algorithm.

IV. Queen-Bee Evolutionary Genetic Algorithm for Test Cases Generation

The Queen-bee evolutionary genetic algorithm works with a population of input data sets for the program; each input data set is a candidate solution to the problem of executing the desired path (testing requirement). From each new generation, in an iterative process, the input values combinations close to executing the desired path are selected. Such process leads to a progressive increase of the solutions' quality, directing the search to the program's input sub-domain associated to data sets that execute the desired path.

The algorithm is given as follows (The execution process is shown in Fig.2). The fitness is decided by numbers of predicates in the CDGPath which covered the Target and one-point crossover is used in used.

Algorithm Test Cases Generation Based on QEGA

```
input     Program: instrumented version of a program to
                   be tested
          CDG: control-dependence graph for Program
          Initial: initial population of test
          TestReq: list of (test requirements, mark)
output    Final: set of test cases for P
declare   CDGPaths: a set of predicate paths in CDG
          Scoreboard: record of satisfied test
                      requirements
          CurrentPopulation, NewPopulation:  set of test
              cases
          Target: test requirement for which a test case
                  is to be generated
          MaxAttempts(): function that returns True when
                         maximum number of attempts for a
                         single Target has been exceeded
          OutOfTime(): function that returns true when
                       the time limit is exceeded, and
                       False Otherwise
```

V. Example

The example is shown in Fig.1. Four test cases are in CurrentPopulation (shown in Table 1). The statement traces of the four test cases cover all statements except Statement 5. Thus, the Statement 5 should be select for requirement. Because $t1$ and $t2$ are corresponding to two predicates (*i.e.* 2, 3) in common with TestReq and $t3$ and $t4$ are corresponding to no predicates with TestReq. By related definition, the CDGPaths corresponding to the TestReq is {*entryT, 2T, 3T*}. The fitness value of test cases $t1$ through $t4$ are {2, 2, 0, 0}, respectively (*entry* is ignored as a predicate because it is on all paths): $t1$ and $t2$ have two predicates (*i.e.* 2, 3) in common with Target and $t3$ and $t4$ corresponding to no predicates with Target.

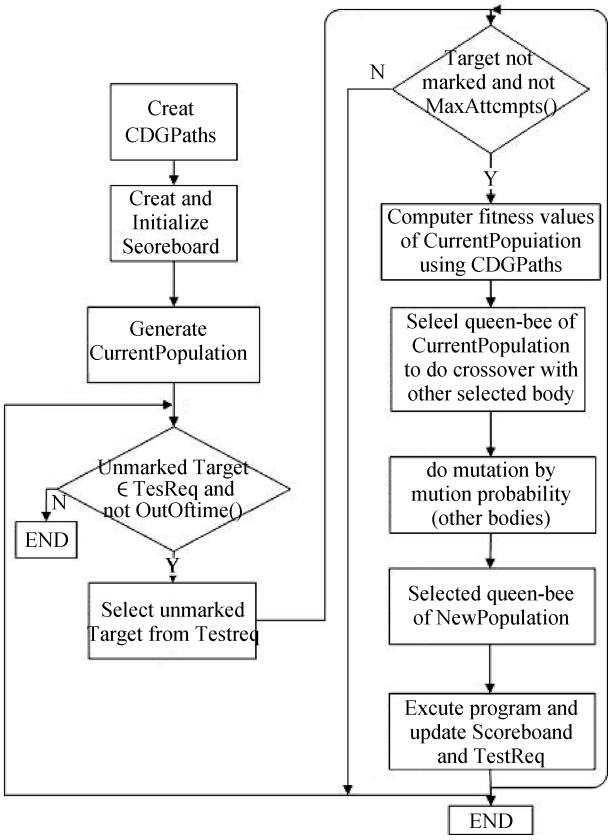


Fig. 2. The execution process of test cases generation algorithm based on QEGA

Table 1. Test cases and statement traces for example of Fig.1

Test case	Test cases(i, j, k)	Statement trace
$t1$	1, 6, 9	1, 2, 3, 4, 6
$t2$	0, 1, 4	1, 2, 3, 4, 6
$t3$	5, 0, 1	1, 2, 6
$t4$	2, 2, 3	1, 2, 6

We use the example to explain the execution of the algorithm with setting $\gamma = 0.33$. In the second generation, we get the test case "(1, 6, 4)" which corresponding to the path "(1, 2, 3, 5, 6)" satisfied the Target.

Table 2. The algorithm execution about example of Fig.1

Generation	Set of test case	Queen-bee	Crossover	mutation
1	{(1, 6, 9); (0, 1, 4); (5, 0, 1); (2, 2, 3)}	(1, 6, 9)	(1, 6, 9); (0, 1, 4)	(2, 2, 3)
2	{(1, 6, 4); (0, 1, 9); (5, 0, 1); (1, 4, 5)}	(1, 6, 4)		

VI. Experiment and Results

An experiment was carried out to evaluate the performance

of the Algorithm proposed in this paper. The computation method for fitness refers to above example.

1. Programs and paths

Three small programs were used. Short descriptions of the three programs are given below. Table 3 contains a summary of them.

Program quotient

Calculates the quotient and the remainder of the division of two positive integers. Contains an IF statement inside a loop While. Some selected paths iterate the loops many times (path lengths up to 49 nodes) executing different sub-paths inside the loops in each iteration.

Program tritype

Accepts three integers representing lengths of a triangle's sides (a, b, c). Classifies the type of triangle and calculates its area. For executing the path that classifies a rectangle triangle an input data set must satisfy: $(a \geqslant b)$ and $(b \geqslant c)$ and $(a < (b + c))$ and $(a \neq b)$ and $(b \neq c)$ and $(a^2 = b^2 + c^2)^d$.

Program find

Accepts an input array A of integers with N elements and an index F; returns the array with every element to the left of $A[F]$ less than or equal to $A[F]$ and every element to the right of $A[F]$ greater than or equal to $A[F]$. It has enchained loops, logical predicates (*e.g.*, if $(!b)$) and some with comparisons between array elements (*e.g.*, (while $(a[i] < a[f])$)). Some of the selected paths are long (up to 119 nodes) and are executed only for very special input characteristics (*e.g.*, all elements of A with the same value).

Find has a "dynamic interface" (the number of program input variables is itself defined by an input variable : N). To deal with this characteristic the Genetic Algorithm works with the defined upper bound of the number of positions for A, but calls the program under test passing only the N variables (N defined in execution time), *i.e.*, not all encoded variables really influence the execution. This solution allows the test data generation, but introduces some "noise" that degrades the Genetic Algorithm's performance. Other authors test this program using fixed values for N, a simpler solution but that may induce unfeasibility of the (otherwise feasible) selected paths in the program.

If $F > N$ the program enters an infinite loop. This bug does not disturb the test data generation because program instrumentation is such that the program halts in these situations.

Table 3. Programs characteristics

Program	Number of nodes	Cyclomatic complexity
Program quotient	13	6
Program tritype	22	9
Program find	22	9

2. Comparison and analysis

We choose above three programs to test and Test cases generation methods are based on simple genetic algorithm (TSGA), elitist preserved genetic algorithm (TEPGA) and queen-bee evolutionary genetic algorithm (TQEGA) respectively. We mainly compare three index as Average number of evolution generations (ANEG), Rate of success (RS, the probability of algorithm which stops less than 2000 iterations) and Average maximum fitness(AMF). By MATLAB tools, the experiment result is given as follows (shown in Table 4):

Table 4. Experiment result

	Program quotient			Program tritype			Program find		
	ANEG	RS	AMF	ANEG	RS	AMF	ANEG	RS	AMF
TSGA	7.1345	71.34%	0.8331	145.18	91.5%	2.3337	443.7247	94.18%	7.5164
TEPGA	5.1427	84.24%	0.8929	141.01	100%	2.7675	373.4379	94.72%	7.6290
TQEGA	4.2373	94.72%	0.9184	127.43	100%	2.8814	323.5705	100%	7.9002

Compare to TSGA, operations of searching and replacing optimum body are added to TEPGA and TQEGA. TQEGA also includes unique operations as random generating new bodies. The experimental results show that the optimization efficiency and optimization effect of TQEGA are better than TSGA and TEPGA significantly.

VII. Conclusions and Future Work

In this paper, the queen-bee genetic algorithm is used to automatically generate test cases for path testing. The experimental results show that this method is more effective and more efficient than existing similar methods. But it also has follow limitations:

– Manual CDG construction takes a considerable amount of time to do.
– Manual target paths identification requires tester creativity.

Future work will try to address the above limitations. Moreover, we will also try to investigate capabilities to allow automatic identification of program paths. Testing object oriented software and adaptation of the technique for goal oriented and error-based test data generation will be another objective of our future research.

References

[1] A. Bertolino, "Software testing research: Achievements, challenges, dreams", *Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, pp. 85–103, 2007.

[2] G. J. Myers, *The Art of Software Testing 2nd ed*, John Wiley & Sons Inc., 2004.

[3] Ahmed W, Wu Y W., "A survey on reliability in distributed systems", *Journal of Computer and System Sciences*, Vol.79, No.8, pp.1243–1255, 2013.

[4] Prasad, Bokil, *et al.*, "System and method for automatic generation of test data to satisfy modified condition decision coverage", U.S. Patent, No.8, 612, 938, 2013.

[5] Cadar C, Godefroid P, Khurshid S, *et al.*,"Symbolic execution for software testing in practice: Preliminary assessment", *Proceedings of the 33rd International Conference on Software Engineering*, ACM, pp.1066–1071, 2011.

[6] Devert, Alexandre, Nicolas Bredeche, and Marc Schoenauer., "Robustness and the halting problem for multicellular artificial ontogeny", *IEEE Transactions on Evolutionary Computation*, Vol.15, No.3, pp.387–404, 2011.

[7] D. L. Bird and C. U. Munoz, "Automatic generation of random self-checking test cases", *IBM System Journal*, Vol.22, No.3, pp.229–245, 2011.

[8] Zhang Y., Gong D W., "Evolutionary genetation of test data for path coverage based on automatic reduction of searchspace", *Acta Electronica Sinica*, Vol.40, No.5, pp.1011–1016, 2012. (in Chinese)

[9] Feiyu L., Yunzhan G., Yawen W., "Automatic test data generation method for complex structure based on memory modeling", *Journal of Computer-Aided Design & Computer Graphics*, Vol.24, No.2, pp.262–270, 2012.

[10] McMinn P., "Search-based software test data generation: A survey". *Software Testing, Verification, and Reliability*, New York: Wiley, Vol.14, No.2, pp.105–156, 2004.

[11] Jiang C., Ying S., Hu S., *et al.*, "Construction method of exception control flow graph for business process execution language process", *Computer Engineering and Networking*, Springer International Publishing, pp.345–353, 2014.

[12] Bagnara R., Carlier M., Gori R., *et al.*, "Symbolic path-oriented test data generation for floating-point programs", *Proc. of the 6th IEEE Int. Conf. on Software Testing, Verification and Validation*, Luxembourg, pp.1–10, 2013.

[13] Ensan A., Bagheri E., Asadi M., *et al.*, "Goal-oriented test case selection and prioritization for product line feature models", *Information Technology: New Generations, 2011 Eighth International Conference on. IEEE*, Las Vegas, NV, pp.291–298, 2011.

[14] Chen Yong and Zhong Yong, "Automatic path-oriented test data generation using a multi-population genetic algorithm", *Proceedings of Fourth International Conference on Natural Computation*, Jinan, China, Vol.1, pp.566–570, 2008.

[15] Chen Yong, Zhong Yong, Bao Shengli, and He Famei, "Structural test data generation using immune genetic algorithm", *The International Conference 2007 on Information Computing and Automation*, Chengdu, China, 2008.

[16] B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style: McGraw-Hill*, Inc. New York, NY, USA, 1982.

[17] Bansal P., Sabharwal S., Malik S., *et al.* "An approach to test set generation for pair-wise testing using genetic algorithms", *Search Based Software Engineering*, Springer Berlin Heidelberg, pp.294–299, 2013.

[18] J. Wegener., H. Sthmer, B. F. Jone and D. E. Eyres, "Testing real-time system using genetic algorithms", *Software Quality Journal*, Vol.6, No.2, pp.127–153, 1997.

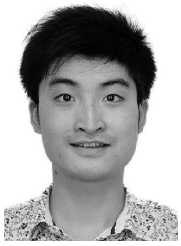
[19] B. F. Jones, H. H. Sthamer, X. Yang and D. E. Eyres, "The automatic generation of software test data sets using adaptive search techniques", *Third International Conference on Software Quality Management*, Seville, pp.435–444, 1999.

[20] B. F. Jones, D. E. Eyres, H.-H Sthamer, "A strategy for using genetic algorithms to automate branch and fault-based testing", *The Computer Journal*, Vol.41, pp.98–107, 1998.

[21] Lin J. C., Yeh P. L., "Automatic test data generation for path testing using Gas", *Information Sciences*, Vol.131, No.1, pp.47–64, 2001.

[22] Bueno P. M. S., Jino M. "Automatic test data generation for program paths using genetic algorithms", *International Journal of Software Engineering and Knowledge Engineering*, Vol.12, No.6, pp.691–709, 2002.

[23] Pargas R. P., Harrold M. J., Peck R. R. "Test-data generation using genetic algorithms", *Software Testing Verification and Reliability*, Vol.9, No.4, pp.263–282, 1999.



MEI Jia was born in Nanning, Guangxi, China, in 1982. He received the Ph.D. degree in computer science from the Shanghai University, Shanghai, China. Now he is an post-doctor fellow of Tsinghua University, Beijing, China. His research interests include software engineering and algorithm design. (Email: meijia@tsinghua.edu.cn)

WANG Shengyuan was born in Shanxi Province, China, in 1964. He received the Ph.D. degree in computer science from the Peking University, Beijing, China. Now he is an associate professor of Tsinghua University, Beijing, China. His research interests focus on Programming Languages and Systems.