An Intelligent Method Based on State Space Search for Automatic Test Case Generation

Ying Xing
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China
School of Electronic and Information Engineering, Liaoning Technical University, Huludao, China
Email: faith.yingxing@gmail.com

Junfei Huang, Yunzhan Gong, Yawen Wang and Xuzhou Zhang
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China
Email: { huangjunfei, gongyz, wangyawen }@bupt.edu.cn, laomao22311@126.com

***Abstract*—Search-Based Software Testing reformulates testing as search problems so that test case generation can be automated by some chosen search algorithms. This paper reformulates path-oriented test case generation as a state space search problem and proposes an intelligent method Best-First-Search Branch & Bound to solve it, utilizing the algorithms of Branch & Bound and Backtrack to search the space of potential test cases and adopting bisection to lower the bounds of the search space. We also propose an optimization method by removing irrelevant variables. Experiments show that the proposed search method generates test cases with promising performance and outperforms some MetaHeuristic Search algorithms.**

***Index Terms*—Search-Based Software Testing, test case generation, Branch & Bound, backtrack, state space search, bisection**

I. INTRODUCTION

Software is being integrated into more and more systems, so it is becoming increasingly important to fully test these systems. One challenge to testing software systems is how to generate test cases systematically in an effective fashion [1]. It is estimated that testing cost has accounted for almost 50 percent of the entire development cost [2], if not more. Therefore, a rational response is to automate the testing process as much as possible, and automatic test case generation naturally plays a key role in this process [3]. Specifically, the automation of path-oriented test case generation (which belongs to the typical control flow testing including those using statement coverage, branch coverage and MC/DC coverage) will efficiently improve testing quality and save the cost of software development [4].

A trend in the automation of path-oriented test case generation is the application of MetaHeuristic Search (MHS) algorithms [5]. The main reason is that test case generation problems can often be reexpressed as search problems. Some MHS algorithms that have been employed for Search-Based Software Testing (SBST) are genetic algorithms [6], simulated annealing [7], and ant colony optimization [8].These algorithms all require the actual execution of the Program Under Test (PUT) and the results are not definite. That is, due to the adoption of the theory of probability they are categorized as cut-and-try methods. Usually a large amount of iterations are made to automatically generate an input which meets the coverage criteria, sometimes causing iteration exception. Therefore, choosing the right algorithm for the problem is very crucial to the search [9].

In this paper, considering the drawbacks of MHS methods mentioned above and on the base of static analysis [10] techniques including interval computation, we introduce the algorithms of Branch & Bound and Backtrack from the field of artificial intelligence to tackle the problem of path-oriented test case generation, which is reformulated as state space search. Bisection is used to lower the bounds of the search space. We also make optimization by irrelevant variable removal (IVR).We have made experiments on C programs, and the results show that the proposed method performs encouragingly in test case generation and has an advantage over some MHS methods in terms of coverage.

The rest of this paper is organized as follows: Section II introduces some relevant concepts including Branch & Bound, Backtrack and bisection; Section III reformulates path-oriented test case generation as a state space search problem; Section IV overviews how the state space is searched dynamically; Section V describes the proposed algorithm and its details; Section VI implements experiments and presents analysis to the results; Section VII concludes this paper and provides the direction of future work.

II. RELATED WORK

Branch & Bound (BB) [11] is an efficient method for searching the solution space of a problem. The advantage of the BB strategy lies in alternating branching and bounding operations on the set of active and extensive nodes of a search tree. Branching refers to partitioning of

doi:10.4304/jsw.9.2.358-364

the solution space (generating the child nodes); bounding refers to lowering bounds used to construct a proof of feasibility without exhaustive search (evaluating the cost of new child nodes). In BB frame, bisection [12] is often used to help prune unneeded part of the solution space. Bisection is also normally used in test case generation [13].

Backtrack [14] is an optimum seeking method which searches forward according to the selection conditions to achieve a goal. If at a certain step of the search, it is found that the goal turns out to be unachievable, then a step backward is taken. The point satisfying the backtrack condition is a backtrack point.

In classical BB search, nodes are always fully expanded, that is, for a given leaf node, all child nodes are immediately added to the so called open list. However, considering that only one solution is enough for path-oriented test case generation, Best-First-Search is our first choice, so permutation of variables is required for branching to prune the branches stretching out from unneeded variables. Meanwhile because the domain of a variable is a finite set of possible values which may be quite large, bounding is necessary to cut the unneeded or infeasible solutions. Hence this paper proposes a new algorithm Best-First-Search Branch & Bound (BFS-BB) that conducts state space search dynamically to find the test case. We also integrate Backtrack algorithm to make full use of the past data obtained during the search process.

III. Reformulation of The Problem

Many forms of test case generation make reference to the control flow graph (CFG) [15] of the program in question. A CFG for a program P is a directed graph G=(N, E, s, e), where N is a set of nodes, E is a set of edges, and s and e are respective unique entry and exit nodes to the graph. Each node $n\in N$ is a statement in the program, with each edge $e=(n_r,n_t)\in E$ representing a transfer of control from node n_r to node n_t. A path p through a CFG is a sequence $p=(n_1,n_2,\ldots,n_q)$, such that for all r, $1\le r<q$, $(n_r,n_{r+1})\in E$. A path is said to be feasible if there exists a program input for which the path is traversed, otherwise the path is said to be infeasible. The feasibility of a path is judged by interval computation. Interval computation analyzes and calculates the ranges of the variables' values in the PUT and provides precise information for further program analysis [16]. We enhance interval computation by adding a library of inverse functions in case of the occurrences of library functions in the PUT.

The path-oriented test case generation problem can be reformulated as a search problem: X is a set of variables$\{x_1,x_2,\ldots,x_n\}$, $D=\{D_1,D_2,\ldots,D_n\}$ is the set of domains and $D_i\in D(i=1,2,\ldots,n)$ is a finite set of possible values for variable x_i. For each path, D is defined based on the variables' acceptable ranges. One solution to the problem is a set of values for each variable inside its domain denoted as $V=\{V_1,V_2,\ldots,V_n\},V_i\in D_i$, to make the path feasible. There might be one, more or no solutions. If there is at least one solution, then the search succeeds otherwise it fails. The solution space is represented by a dynamically constructed tree where each node represents a step of the search. With the aid of intelligent rules for selecting nodes to explore and pruning those that do not lead to a solution, the complexity of the search can be drastically reduced as compared to that of an exhaustive implicit enumerative search. The search process is based on the result of interval computation and a heuristic estimate of the remaining part.

IV. Overview of BFS-BB

We introduce state space search [17], which is an important issue in artificial intelligence to tackle the search problem. In order to facilitate the implementation of BFS-BB, we propose the following definitions.

Definition1. A ***state space*** is a quadruple(S, A, I, F), where S is the set of states, A is the set of arcs or connections between the states including *Permutate*, *Select*, *Reduce Domain* and *Backtrack* that correspond to the steps or operations of the search at different states, I is a non-empty subset of S denoting the initial state of the problem and F is a non-empty subset of S denoting the final state of the problem.

Definition 2. A ***state*** is a tuple(*Precursor*, *Variable*, *Domain*, *Value*, *Type*, *Queue*). In a certain stage of the search process, from the perspective of current state S_{cur}, *Precursor* provides a link to the previous state; $Variable=x_i\in X$ (i=1,2,…,n) is an input variable of PUT; $Domain=D_{ij}\subseteq D_i\in D$, ($i$=1,2,…,$n$; j=1,2,…,m) in the form of [min, max]is the current domain of *Variable* which is the set of possible values that may be selected for *Variable*; $Value=V_{ij}\in D_{ij}$ is a value selected from *Domain*; *Type* marks the type of S_{cur}, which may be *active*, *extensive* or *inactive*; *Queue* is a sequence of variables corresponding to S_{cur}.

Definition 3. ***State space search*** is all about finding, in a state space (which may be extremely large), one final state. 'Final' means that every variable has been given a definite value successfully and the path is proved to be feasible with all these values by interval computation. At the start of the search *Precursor* is null, and when *Queue* is null the search ends. The path made up of all the *extensive* states makes the solution path of the search. State space search is accomplished by BFS-BB in this paper.

When constructing each state, *Type* is *active*. $Queue=Q_{ipre}$ and *Variable* is the head of *Queue*. An interval computation is carried out to each *active* state to determine the direction of the next step of search. If the interval computation succeeds, then *Type* becomes *extensive*, the remaining variables will be permutated to get $Queue=Q_{inext}$, S_{cur} becomes *Precursor*, and the head of Q_{inext} will be *Variable* of next state. If interval computation fails, *Type* remains *active*, according to the information from the failed interval computation *Reduce domain* is conducted with bisection, and *Value* is reselected from the reduced domain, all of which mean the search will expand to a state with a different value for the same variable. If for the same variable all the values within its domain are tried out or the interval computation for it has reached the time limit *m*(which is the branching factor, or a threshold used to control the breadth of the search tree ,namely, the limit on the number of times of interval computation taken for one variable under the same condition), then its *Type* becomes *inactive*, indicating that the search arrives at a backtrack point and will have to backtrack to the previous state *Precursor* at the higher level of the search tree.

The process of generating a test case for path *p* takes the form of state space search. We need to search the state space to find a solution path from an initial state to a final state. We can decide where to go by considering the possible moves from the current state, and trying to look ahead as far as possible.

V. DETAILS OF BFS-BB

We proceed by first giving an outline of algorithm BFS-BB and then giving the detailed explanations to the key parts. The outline of BFS-BB is shown in Fig.1.

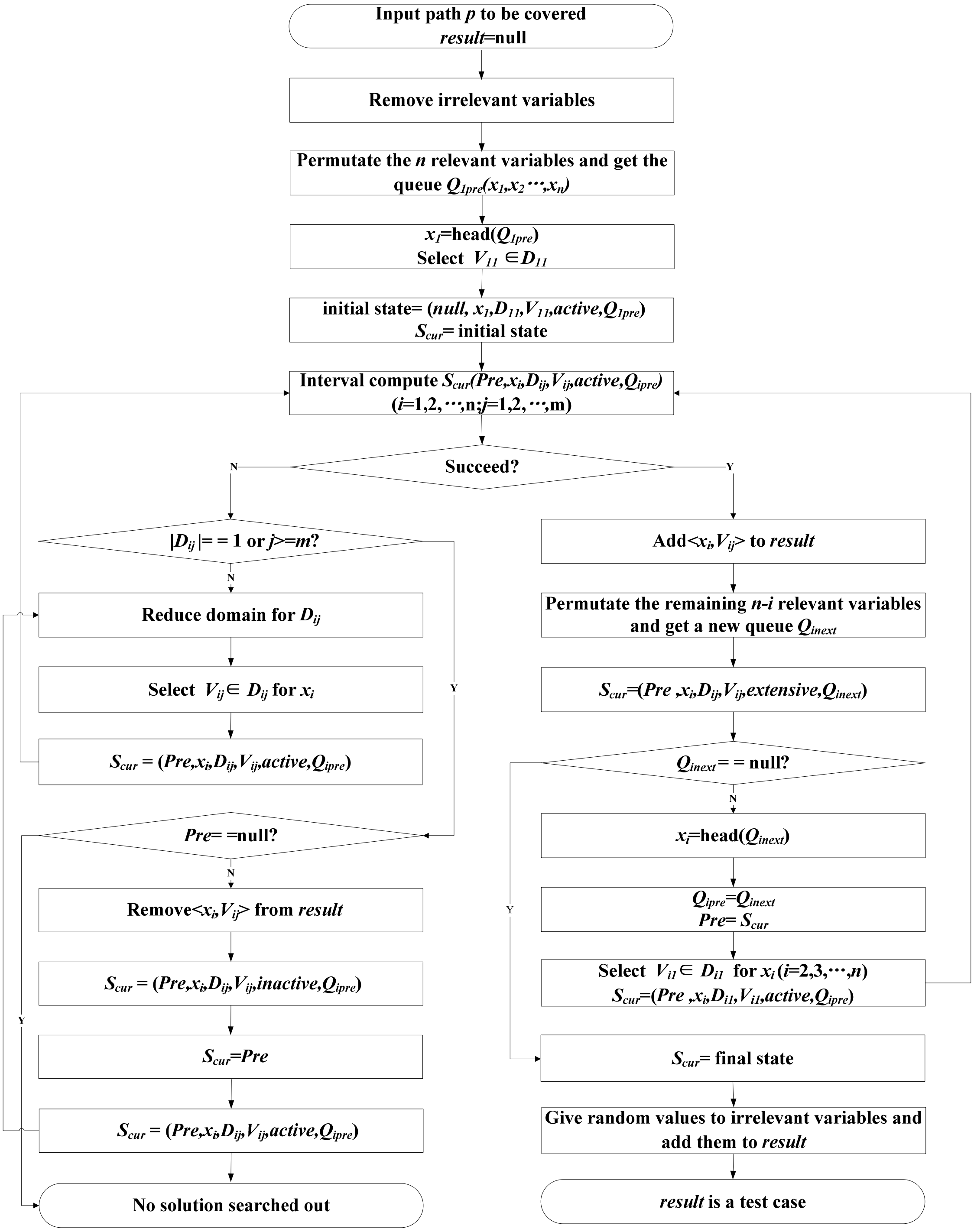


Figure 1.The outline of the search algorithm

A. Irrelevant Variable Removal

As mentioned above, $X=\{x_1,x_2,\ldots,x_n\}$ is the set of input variables for the program *P*. The state space should concern every x_i (i=1,2,…,n) in *X*. However, it is possible that not every input variable will be responsible for determining whether every path in *P* will be traversed or not. A simple example follows with a program *test1* and its corresponding CFG shown in Fig.2 where *if_out_5*, *if_out_6* and *exit_7* are virtual nodes. Adopting branch coverage, there are three paths to be covered, while the input variable *x3* is only relevant to *Path 2:0->1->3->4->5->6->7* and *Path 3: 0->1->3->5->6->7*, but not to *Path 1:0->1->2->6->7*. The numbers along the path denote nodes rather than edges of the CFG. Therefore, when attempting to generate test case for *Path 1*, search effort on the value of *x3* is wasted since it cannot influence the traversal of *Path 1*. Thus, removing irrelevant input variables from the search space and only concentrating on input variables relevant to the path of interest may improve the performance of the search process. Relevant and irrelevant variables are defined as follows.

Definition 4. A ***relevant variable*** is an input variable that can affect whether a particular path *p* will be traversed or not. To put it more precisely, for all the input variables $\{x_i\,|x_i\in X,\ i=1,2,\ldots,n\}$, there exists a corresponding set of values $\{V_i\ |V_i\in D_i,\ i=1,2,\ldots,n\}$, with which *p* is not traversed, but when the value of a particular variable is changed, for example, when the value of $x_g(V_g)$ is changed into $V_g^{'}$, *p* is traversed with the input $\{V_1,V_2,\ldots,V_g^{'},\ldots,V_n\}$, then x_g is a relevant variable to path *p*.

Definition 5. An ***irrelevant variable*** is an input variable that is not capable of influencing whether a particular path *p* will be traversed or not. To put it more precisely, for all the sets $\{V_i\ |V_i\in D_i,\ i=1,2,\ldots,n\}$ of the search space of path *p* , with which *p* is not traversed, regardless of the change in the value of a particular variable, for example, the value of $x_g(V_g)$ is changed into $V_g^{'}$, *p* is still not traversed with the input $\{V_1,V_2,\ldots,V_g^{'},\ldots,V_n\}$, then x_g is an irrelevant variable to path *p*.

```
void test1(int x1, int x2, int x3)
1{   if (x1- x2>0)
2          printf("Path 1");
3    else if(x3-x1>0)
4            printf("Path2 ");
  }
```

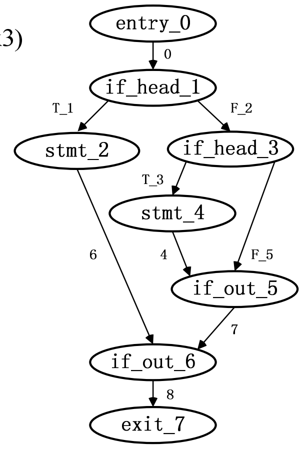


Figure 2. Program *test1* and its corresponding CFG

Generally, for a particular path, whether an input variable is relevant or irrelevant cannot be completely determined due to the complex structure of programs [18]. But we can make conservative estimate of irrelevancy with static analysis techniques. Irrelevant variable removal (IVR) can result in test case being searched out with fewer interval computations for a particular path *p* than if all variables are considered.

B. Reducing Domain

Bisection is used to reduce the domain of a variable which has been selected a value making path *p* infeasible, together with *Tendency*, which is a property of a variable at a certain position (especially at branches of a CFG) providing the direction of the next step of search.

Definition 6. ***Tendency map*** is a mapping table <*Variable*, *Tendency*>denoting the relationship between *Variable* and *Tendency* held by a specific branch *Br* along path *p*. *Tendency* is determined by branch predicates and expressions at the branch.

Note that there might be more than one tendency maps in a program, for each of them is held by a single branch. Still take *test1* as an example, as mentioned above there are three paths to be covered, which are *Path 1:0->1->2->6->7*, *Path 2:0->1->3->4->5->6->7* and *Path 3: 0->1->3->5->6->7*, respectively. Accordingly we can get tendency maps as shown in TABLE I.

Take *Path 1* as an instance, if $S_{cur}=(Pre,x_1,D_{11},V_{11},active,Q_{1pre})$ and interval computation fails at branch *T_1*, then we retrieve the corresponding tendency map and get the tendency of *x1* as the result which is *positive*. Through the retrieval of tendency map we can propagate the constraints made up of the branch predicates in a more and more precise manner as presented by Fig.3.

TABLE I.
TENDENCY MAPS OF FIGURE 2.

Path	Branch	Tendency map
Path 1	*T_1*	{<*x1*,*positive*>,<*x2*,*negative*>}
Path 2	*F_2*	{<*x1*,*negative*>,<*x2*,*positive*>}
	T_3	{<*x3*,*positive*>,<*x1*,*negative*>}
Path 3	*F_2*	{<*x1*,*negative*>,<*x2*,*positive*>}
	F_5	{<*x3*,*negative*>,<*x1*,*positive*>}

Algorithm. ***Reducing domain***
Input D_{ij}=[*min*,*max*]:the domain of x_i
Output D_{ij}: the reduced domain of x_i
begin
1: *j*++;
2: *Br*←position of failure;
3: *Tendency* = get(x_i);
4:// retrieve the tendency map held by *Br*
5: **if**(*Tendency*== *positive*)
6: D_{ij}=[V_{ij}+1,*max*];
7: **else if**(*Tendency*==*negative*)
8: D_{ij}=[*min*,V_{ij}-1];
9: **return** D_{ij};
end

Figure 3. The algorithm *Reducing Domain*

VI. Experimental Results and Discussion

To observe the effectiveness of BFS-BB, we carried out a large number of experiments in our team Code Testing System (CTS). Within the CTS framework, the PUT is automatically analyzed, its basic information is abstracted to form the Abstract Syntax Tree (AST) [19], and its CFG is generated. According to the specified coverage criteria, the paths to be covered are generated and provided for BFS-BB as input. After test cases have been generated by BFS-BB, test drive is generated to provide the environment to execute the test case. There are some auxiliary functions in CTS, including coverage observation, presentation of the covered code lines as well as the execution results, and the management of test cases for the convenience of regression testing. These functions of CTS provide comfortable experience for users such as the testing personnel.

The experiments were performed in the environment of MS Windows 7 with 32-bits and run on Pentium 4 with 2.8 GHz and 2 GB memory. The algorithms were implemented in Java and run on the platform of eclipse. Section A presents a performance evaluation about BFS-BB, Section B concerns whether BFS-BB outperforms other commonly used MHS algorithms in terms of coverage. Four programs served as our test beds including a benchmark program used in CTS and three others in test case generation, and the details of them are shown in TABLE II.

TABLE II.
BENCHMARK PROGRAMS USED FOR EXPERIMENTAL ANALYSIS

Program	LOC	Variables	Description	Source
branch_bound	402	27	A benchmark used in CTS	by authors
isValidDate	59	16	To check whether a date is valid or not	referring to[8]
calDay	72	3	To calculate the day of the week	referring to[20]
cal	53	5	To calculate the number of days between the two given days in the same year	referring to[21]

A. Performance Evaluation

To evaluate the performance of BFS-BB in test case generation, test cases were automatically generated to meet three different coverage criteria: statement, branch, and MC/DC. In this section, we utilized branch_bound.c , which is a relatively long program for unit testing with 402 lines and 27 input variables and complex structure trying to include more content that might appear in real-world PUTs. Since not all of the 27 variables are relevant for a specific path, comparison of search time is made to evaluate the effect of IVR.

Results of branch_bound.c using different coverage criteria are shown in TABLE III. The numbers of paths and average branches are different owing to different coverage criteria taken. BFS-BB generated test cases for all the feasible paths, trying to reach 100% coverage. IVR had no significant influence on the coverage, but it did on the search time. After the adoption of IVR, the search time was reduced greatly. Our following analyses all involve BFS-BB with IVR.

For branch_bound.c, BFS-BB was able to cover almost every branch, and generating test cases took a few seconds for all the feasible paths. The MC/DC coverage [22] (which is relatively strict and subsumes statement coverage and branch coverage) did not reach 100%, because we set time limit for the search time for each path as well as the threshold *m* mentioned above for each variable. But we achieved tolerable coverage within tolerable time. There exists a trade-off between efficiency and success rate.

TABLE III.
EXPERIMENTAL RESULT USING THREE DIFFERENT COVERAGE CRITERIA WITH BFS-BB

Adequacy criterion	Paths	Average Branches	Average Coverage %	Search time reduced by IVR%
statement	61	29	100	34
branch	119	43.33	100	37
MC/DC	125	43	94	42

B. Coverage Evaluation

This section presents results from a practical comparison of BFS-BB with GA and SA on three different benchmark programs using branch coverage as the adequacy criterion, which offers a favorable trade-off between costs and efficiency [23].The result is shown in TABLE IV.

It can be seen that BFS-BB reached 100% branch coverage on all three test beds which are relatively simple programs for BFS-BB and outperformed the algorithms in comparison.

The better performance of BFS-BB results from two factors. One is that random testing [24] is a cheap and easy technique that can obtain reasonable coverage, simple yet effective in finding software fault, so for most of the cases, BFS-BB reached a relatively high coverage for the first round of search with a high speed. The second is that MHS crashed on several occasions due to the iteration exception, while the probability of aborting is quite low for BFS-BB because it has no demand for iteration.

TABLE IV.
COMPARISON WITH SA AND GA USING BRANCH COVERAGE

Program	Paths	Branches	GA Average Coverage %	SA Average Coverage%	BFS-BB Average Coverage%
isValidDate	5	16	99.95	98.21	100
calDay	20	11	96.31	99.97	100
cal	7	18	99.02	99.27	100

VII. CONCLUSION AND FUTURE WORK

This paper presents an intelligent search algorithm for path-oriented test case generation that utilizes the classical search algorithms of Branch & Bound and Backtrack. Experiments show that BFS-BB with IVR performs well on C programs. We also conducted empirical experiments to compare BFS-BB with some commonly used MHS methods, which produced encouraging results. This paper makes two major innovative improvements.

First, path-oriented test case generation is often solved by optimizing techniques, which may often suffer from the problem of local minimal or the initial starting point being too far from the solution. Our approach is flexible because backtrack is used to change direction of the search with efficiency. Second, bisection with tendency maps and IVR are used to optimize BFS-BB and accelerate the search process.

Our future research concerns not only how to generate test cases to reach high coverage but how coverage criteria, generation approach, and system structure jointly influence test effectiveness. The fault-finding capability of test cases and the effectiveness of the generation approach will be our focus for future work.

ACKNOWLEDGMENT

We appreciate anonymous reviewers for their supportive and constructive comments. The work described in this paper was supported by the National Grand Fundamental Research 863 Program of China (No. 2012AA011201), and Key Project of the National Natural Science Foundation of China (No. 61202080).

REFERENCES

[1] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder K. Panesar-Walawege, “A systematic review of the application and empirical investigation of search-based test case generation,” IEEE Transactions on Software Engineering, vol. 36, no. 6, 2010, pp. 742-762.

[2] B.Beizer, “Software Testing Techniques,” Second Edition, Van Nostrand Reinhold, New York, 1990.

[3] Chen Lina, “Automatic test cases generation for statechart specifications from semantics to algorithm,” Journal of Computers, vol. 6, no. 4, 2011, pp. 769-775.

[4] Shan Jinhui, Wang Ji, and Qi Zhichang, “Survey on path-wise automatic generation of test data,” ACTA ELECTRONICA SINICA, vol. 32, no.1, 2004, pp. 109-113.

[5] W. Afzal, R. Torkar, and R. Feldt, “A systematic review of search-based testing for non-functional system properties,” Information and Software Technology, vol. 51, 2009, pp. 957-976.

[6] Sharma, Chayanika, Sangeeta Sabharwal, and Ritu Sibal, “A survey on software testing techniques using genetic algorithm,” International Journal of Computer Science Issues, vol. 10, issue. 1, no. 1, 2013, pp. 381-393.

[7] Patil, Manisha, and P. J. Nikumbh, “Pair-wise testing using simulated annealing,” Procedia Technology, vol. 4, 2012, pp. 778-782.

[8] Mao Chengying, Yu Xinxin, and Chen Jifu, “Generating test data for structural testing based on ant colony optimization,” Proc. the 12th International Conference on Quality Software (QSIC 12), IEEE Computer Society, 2012, pp. 98-101.

[9] Mark Harman, Phil McMinn, Jefferson Teixeira de Souza, and Shin Yoo, “Search based software engineering: techniques, taxonomy, tutorial,” Empirical Software Engineering and Verification, Springer Berlin Heidelberg, 2012, pp. 1-59.

[10] Fuad, Mohammad Muztaba, Debzani Deb, and Jinsuk Baek. “Static analysis, code transformation and runtime profiling for self-healing,” Journal of Computers, vol. 8, no.5, 2013, pp. 1127-1135.

[11] Jalilvand, Abolfazl, and S. Khanmohammadi, “A new method for constructing the search tree in branch and bound algorithm,” Proc. the 9th International Multitopic Conference (INMIC 05), IEEE Computer Society, 2005, pp.1-5.

[12] Delling, D., Goldberg, A. V., Razenshteyn, I., and Werneck, R. F., “Exact combinatorial branch-and-bound for graph bisection,” ALENEX, 2012, pp. 30-44.

[13] Sami Beydeda and Volker Gruhn., “BINTEST-search-based test case generation,” Computer Software and Applications In Computer Software and Applications Conference (COMPSAC 03), IEEE Computer Society, 2003, pp. 28-33.

[14] Lisgara, E. G., G. I. Karolidis, and G. S. Androulakis, “Advancing the backtrack optimization technique to obtain forecasts of potential crisis periods,” Applied Mathematics, vol. 3, no.30, 2012, pp. 1538-1551.

[15] Phil McMinn, “Search-based software test data generation: a survey,” Software Testing, Verification and Reliability, vol. 14, no.2, 2004, pp. 105-156.

[16] Hickey, Timothy, Qun Ju, and Maarten H. Van Emden., “Interval arithmetic: from principles to implementation,” Journal of the ACM, vol. 48, no. 5, 2001, pp. 1038-1068.

[17] Daniel Szer, Francois Charpillet, and Shlomo Zilberstein. , “MAA*: A heuristic search algorithm for solving decentralized POMDPs,” Proc. the 21st Conference on Uncertainty in Artificial Intelligence (UAI 05), Edinburgh, Scotland, July 2005, pp. 576–583.

[18] Phil McMinn, Mark Harman, Kiran Lakhotia, Youssef Hassoun, and Joachim Wegener, “Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation,” IEEE Transactions on Software Engineering, vol. 38, no. 2, 2012, pp.453-477.

[19] Pattanayak, Binod Kumar, Sambit Kumar Patra, and Bhagabat Puthal, “Optimizing AST node for Java script compiler a lightweight interpreter for embedded device,” Journal of Computers, vol. 8, no.2, 2013, pp. 349-355.

[20] E.Alba and F.Chicano, "Observation in using parallel and sequential evolutionary algorithms for automatic software testing," Computers and Operators Research, vol. 35, 2008, pp. 3161-3183.
[21] P.Ammann and J.Offutt, "Introduction to software testing," Cambridge University Press, 2008, pp. 32
[22] Rajan, Ajitha, Michael W. Whalen, and Mats PE Heimdahl., "The effect of program and model structure on MC/DC test adequacy coverage," Proc. ACM/IEEE 30th International Conference on Software Engineering (ICSE 08), IEEE Computer Society, 2008, pp. 161-170.
[23] Fan Chunrong, Chen Zhenyu and Xu Baowen, "Comparing logic coverage criteria on test case prioritization," Science China Information Sciences, vol. 55, no. 12, 2012, pp. 2826-2840.
[24] Patrice Godefroid, Nils Klarlund, and Koushik, "DART: directed automated random testing," ACM Sigplan Notices. vol. 40, no. 6, ACM, 2005, pp. 213-223.

Ying Xing was born in Liaoning Province, China. She is a Ph.D. candidate in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications.

Also she is a lecturer in School of Electronic and Information Engineering, Liaoning Technical University. She obtained her Master Degree from Liaoning Technical University in 2007. Her research interests include software testing and static analysis.

Junfei Huang was born in Zhejiang Province, China in 1977. He received his Ph.D. from Beijing University of Posts and Telecommunications in 2004.

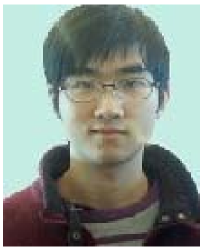
He is currently a lecturer in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. His research interests include static analysis and cloud computing.

Yunzhan Gong was born in Shandong Province, China in 1962. He received his Ph.D. from Institute of Computing Technology, Chinese Academy of Sciences in 1991.

He is currently a professor and supervisor of doctoral students in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. His research interests include fault tolerant computing and software testing.

Yawen Wang was born in Shanxi Province, China in 1983. She received her Ph.D. from Beijing University of Posts and Telecommunications in 2010.

She is currently a lecturer in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. Her research interests include static analysis and automatic software testing.



Xuzhou Zhang was born in Hebei Province, China in 1987. He received his B.S from HuaZhong University of Science and Technology.

He is currently a postgraduate in the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications. His research interest is software testing.