Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2014, Article ID 642630, 19 pages
http://dx.doi.org/10.1155/2014/642630

Research Article

Path-Wise Test Data Generation Based on Heuristic Look-Ahead Methods

Ying Xing,[1,2] Yun-Zhan Gong,[1] Ya-Wen Wang,[1,3] and Xu-Zhou Zhang[1]

[1] *State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China*
[2] *School of Electronic and Information Engineering, Liaoning Technical University, Huludao 125105, China*
[3] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*

Correspondence should be addressed to Ying Xing; faith.yingxing@gmail.com

Received 5 July 2013; Revised 1 December 2013; Accepted 2 December 2013; Published 14 May 2014

Academic Editor: John Gunnar Carlsson

Path-wise test data generation is generally considered an important problem in the automation of software testing. In essence, it is a constraint optimization problem, which is often solved by search methods such as backtracking algorithms. In this paper, the backtracking algorithm branch and bound and state space search in artificial intelligence are introduced to tackle the problem of path-wise test data generation. The former is utilized to explore the space of potential solutions and the latter is adopted to construct the search tree dynamically. Heuristics are employed in the look-ahead stage of the search. Dynamic variable ordering is presented with a heuristic rule to break ties, values of a variable are determined by the monotonicity analysis on branching conditions, and maintaining path consistency is achieved through analysis on the result of interval arithmetic. An optimization method is also proposed to reduce the search space. The results of empirical experiments show that the search is conducted in a basically backtrack-free manner, which ensures both test data generation with promising performance and its excellence over some currently existing static and dynamic methods in terms of coverage. The results also demonstrate that the proposed method is applicable in engineering.

1. Introduction

Software testing plays an irreplaceable role in the process of software development, as it is an important stage to guarantee software reliability [1], which is a significant software quality feature [2]. It is estimated that testing cost has accounted for almost 50 percent of the entire development cost [3], if not more, but manual testing is time-consuming and error-prone with low efficiency and is even impracticable for large-scale programs such as a Windows project with millions of lines of codes (LOC) [4]. Therefore, the automation of testing is an urgent issue. Furthermore, as a basic problem in software testing, path-wise test data generation (denoted as Q) is of particular importance because path-wise testing can detect almost 65 percent of the faults in the program under test (PUT) [5] and many problems in software testing can be transformed into Q.

The methods of solving Q can be categorized as static and dynamic. The static methods utilize techniques including symbolic execution [6, 7] and interval arithmetic [8, 9] to analyze the PUT without executing it. The process of generating test data is definite with relatively less cost. They abstract the constraints to be satisfied and propagate and solve these constraints to obtain the test data. Due to their precision in generating test data and the ability to prove that some paths are infeasible, the static methods have been widely studied by many researchers. DeMillo and Offutt [10] proposed a fault-based technique that used algebraic constraints to describe test data designed to find particular types of faults. Gotlieb et al. [11] introduced "static single assignment" into a constraint system and solved the system. Cadar et al. from Stanford University proposed a symbolic execution tool named KLEE [12] and employed a variety of constraint solving optimizations. They represented program states compactly and used searching heuristics to reach high code coverage. In 2013, Yawen et al. [13] proposed an interval analysis algorithm using forward data-flow analysis. But no matter what techniques are adopted, the static methods require a strong constraint solver.

The dynamic methods including metaheuristic search (MHS) algorithms [18] such as genetic algorithms [19], ant colony optimization [15], and simulated annealing [20] all require the actual execution of the PUT. They select a group of test data (usually randomly) in advance and execute it to observe whether the goal is reached; that is, coverage criteria are satisfied or faults are detected and if not, they spot the problem and alter the values of some input variables to make the PUT execute in the expected way. They are flexible methods as they can easily change the input in the testing process, but they are sensitive to the search-space size, the diversity of initial population, the effectiveness of evolution operators, and the quality of fitness function [21]. The repeated exploration requires a large number of iterations, sometimes even causing iteration exception. The randomness of initial values is also a big problem because it brings uncertainty to the search result [22].

In this paper, considering the drawbacks of the dynamic methods mentioned above and the demand for static methods, we propose a new static test data generation method based on Code Test System (CTS) (http://ctstesting.cn/), which is a practical tool to test codes written in C programming language. Our contribution is threefold. First, path-wise test data generation is defined as a constraint optimization problem (COP). Two techniques (state space search and branch and bound) in artificial intelligence are integrated to tackle the COP. Second, heuristics are adopted in the look-ahead stage of the search to improve the search efficiency. Third, an optimization method is proposed to reduce the search space. We try to evaluate the performance of our method and the relationship between look-ahead and look-back techniques through the experimental results.

The rest of this paper is organized as follows. Section 2 provides the background underlying our research. The problem Q is reformulated as a COP and the solution is presented in Section 3. Section 4 illustrates the proposed search strategies in detail and describes the optimization method used to reduce the search space. The heuristic look-ahead techniques with a case study are given in Section 5. Section 6 focuses on experimental analyses and empirical evaluations of the proposed approach as well as coverage comparisons with some existing test data generation methods. Section 7 concludes this paper and highlights directions for future research.

2. Background

State space search [23, 24] is a process in which successive states of an instance are considered, with the goal of finding a final state with a desired property. Problems are normally modeled as a state space, a set of states that a problem can be in. The set of states forms a graph where two states are connected if there is an operation which can be performed to transform the first state into the second. State space search characterizes problem solving as the process of finding a solution path from an initial state to a final state. In state space search, the nodes of the search tree are corresponding to partial problem solution, and the arcs are corresponding to steps in a problem-solving process. State space search differs from traditional search methods because the state space is implicit; the typical state space is too large to generate and store in memory. Instead, nodes are generated as they are explored and typically discarded thereafter.

Branch and bound (BB) [25, 26] is an efficient backtracking algorithm for searching the solution space of a problem as well as a common search technique to solve optimization problems. 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 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). The techniques for improving BB are categorized as look-ahead and look-back methods. Look-ahead methods [27] are invoked whenever the search is preparing to extend the current partial solution, and they concern the following problems: (1) how to select the next variable to be instantiated or to be assigned a value; (2) how to select a value to instantiate a variable; (3) how to reduce the search space by maintaining a certain level of consistency. Look-back methods are invoked whenever the search encounters a dead end and is preparing for the backtracking step, and they can be classified into chronological backtracking and backjumping.

An important static testing technique adopted in this paper is interval arithmetic [8, 9, 28–30], which represents each value as a range of possibilities. An interval is a continuous range in the form of [min, max], while a domain is a set of intervals. For example, if an integer variable x ranges from -3 to 6, but it cannot be equal to 0, then its domain is represented as $[-3,-1] \cup [1,6]$, which is composed of two intervals. Interval arithmetic has a set of arithmetic rules defined on intervals. It analyzes and calculates the ranges of variables starting from the entrance of the program and provides precise information for further program analysis efficiently and reliably.

3. Reformulation of Path-Wise Test Data Generation

This section addresses the reformulation of path-wise test data generation. Problem definition and its solution are presented in Sections 3.1 and 3.2, respectively.

3.1. Problem Definition.

3.1. Problem Definition. Many forms of test data generation make references to the control flow graph (CFG) of the program in question [31]. In this paper, a CFG for a program P is a directed graph $G = (N, E, i, o)$, where N is a set of nodes, E is a set of edges, and i and o are respective unique entry and exit nodes to the graph. Each node $n \in N$ is a statement in the program, with each edge $e = (n_r, n_t) \in E$

```
void test(int x1, int x2, int x3)
1 { if (x1-x2<=0)
2         printf("Path1");
3   else if(x3-x2<=0)
4         printf("Path2");
5       else if(3*x3+5>=0)
6             printf("Path3");
  }
```

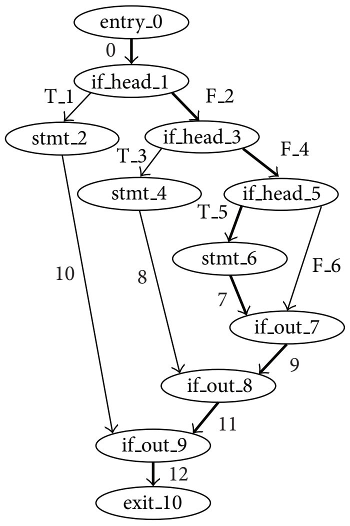


Figure 1: Program *test* and its corresponding CFG.

representing a transfer of control from node n_r to node n_t. Nodes corresponding to decision statements such as if statements are branching nodes. Outgoing edges from these nodes are referred to as branches. A path through a CFG is a sequence $p = (n_1, n_2, \ldots, n_q)$, such that for all r, $1 \leq r < q$, $(n_r, n_{r+1}) \in E$. A path p is regarded as feasible if there exists a program input for which p is traversed; otherwise p is regarded as infeasible. Then the problem Q can be reformulated as a COP [32, 33] as follows. X is a set of variables $\{x_1, x_2, \ldots, x_n\}$, $D = \{D_1, D_2, \ldots, D_n\}$ is a set of domains, and $D_i \in D$ $(i = 1, 2, \ldots, n)$ is a finite set of possible values for x_i. For each path, D is defined based on the variables' acceptable ranges. One solution to the problem is a set of values to instantiate each variable inside its domain denoted as $V = \{V_1, V_2, \ldots, V_n\}$, $V_i \in D_i$ to make path p feasible. Particularly, each constraint defined by the PUT along p should be met to make it feasible.

An example with a program test and its corresponding CFG is shown in Figure 1, where if_out_7, if_out_8, if_out_9, and exit_10 are dummy nodes. Adopting branch coverage, there are four paths to be traversed, namely, Path1: $0 \to 1 \to 2 \to 9 \to 10$, Path2: $0 \to 1 \to 3 \to 4 \to 8 \to 9 \to 10$, Path3: $0 \to 1 \to 3 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$, and Path4: $0 \to 1 \to 3 \to 5 \to 7 \to 8 \to 9 \to 10$. The numbers along the paths denote nodes rather than edges of the CFG. Assuming that Path3 is the path to be traversed as shown in bold, our work is to select $V = \{V_1, V_2, V_3\}$ from $\{D_1, D_2, D_3\}$ for x_1, x_2, and x_3, so that when executing test using $\{V_1, V_2, V_3\}$ as an input, the path traversed is Path3. There are three branching nodes, if_head_1, if_head_3, and if_head_5, along Path3 and three corresponding branches, F_2, F_4, and T_5, that contain the constraints to be met.

3.2. Solution to the Problem. A COP is generally solved by search strategies, among which backtracking algorithms [34] are widely used. In this paper, state space search and the backtracking algorithm BB are introduced to solve the COP mentioned above. The process of exploring the solution space is represented as state space search. This representation will facilitate the implementation of BB. 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 one solution is enough for path-wise test data generation, best-first-search is our first choice. To find the best, ordering of variables is required for branching to prune the branches stretching out from unneeded variables. In addition, as 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. In BB frame, bisection [35] is often used to help prune unneeded part of the solution space. Employing bisection, this paper proposes best-first-search branch and bound (BFS-BB) to automatically generate the test data.

It has been observed empirically that the enhancement of look-ahead methods is sometimes counterproductive to the effects of look-back methods [36]. As for BFS-BB, heuristics are adopted in the look-ahead search. Particularly, they are used in the dynamic ordering of variables, the selection of the values to assign a variable, and the judgment of the feasibility of the path after the assignment to a variable and the reduction of the search space. Chronological backtracking is used for look-back. And from the results of the experiments, we try to seek out the relationship between look-ahead and look-back methods.

During the search process, variables are divided into three sets: past variables (short for PV, already instantiated), current variable (now being instantiated), and future variables (short for FV, not yet instantiated). All the variables involved in this paper are symbolic variables. In the interest of simplicity, the transformation from input variables to symbolic variables and the inverse transformation are beyond the scope of this paper. In addition, although the experiments were carried out on benchmarks in the literature or industrial programs of different variable types, integer variables are used for brevity in the following algorithms.

4. The Proposed Search Strategies

This section proposes the framework of the search strategies. Particularly, the representation of state space search is described in detail in Section 4.1, which is followed by the search algorithm BFS-BB in Section 4.2. And an optimization method in BFS-BB is explained in Section 4.3.

4.1. The Representation of State Space Search. A *state* is a tuple (*Precursor*, *Variable*, *Domain*, *Value*, *Type*, and *Queue*). Precursor provides a link to the previous state; *Variable* $= x_i \in X$ $(i = 1, 2, \ldots, n)$ is the current variable; *Domain* $= D_{ij} \subseteq D_i \in D$ $(i = 1, 2, \ldots, n;\ j = 1, 2, \ldots, m$, m is the branching factor or the threshold used to control the breadth of the search tree) in the form of $[\min, \max]$ is the set of possible values to be selected to instantiate *Variable*; *Value* $= V_{ij} \in D_{ij}$ is a value selected from *Domain*; *Type* marks the type of *state*: *active*, *extensive*, or *inactive*; *Queue* is a sequence of variables corresponding to the state in question.

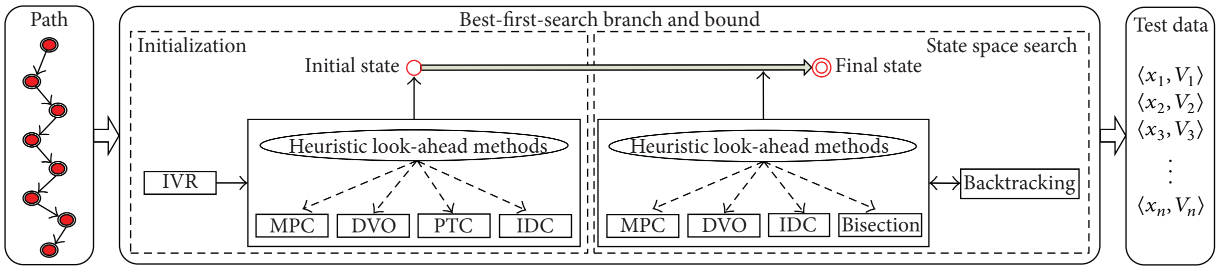

Path
Best-first-search branch and bound
Initialization
State space search
Initial state
Final state
Heuristic look-ahead methods
IVR
MPC
DVO
PTC
IDC
Heuristic look-ahead methods
MPC
DVO
IDC
Bisection
Backtracking
Test data
$\langle x_1, V_1\rangle$
$\langle x_2, V_2\rangle$
$\langle x_3, V_3\rangle$
$\vdots$
$\langle x_n, V_n\rangle$

Figure 2: Program *test* and its counterpart with branching conditions decomposed into basic functions.

State space is a quadruple (S, A, I, F), where S is a set of states, A is a set of connections between the states in accordance with the search operations, I is a nonempty 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.

State space search is all about finding one final state in a state space (which may be extremely large). *Final* means that every variable has been instantiated with a definite value successfully. 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* nodes in the search tree makes the solution path. The state space needs to be searched to find a solution path from an initial state to a final state.

4.2. The Search Algorithm BFS-BB. The idea of the search algorithm BFS-BB is to extend partial solutions. At each step, a variable in FV is selected and assigned a value from its domain to extend the current partial solution. It is checked whether such an extension may lead to a possible solution of the COP and the subtrees containing no solutions based on the current partial solution are pruned. Some concepts in BFS-BB are explained as follows.

Irrelevant variable removal (IVR) identifies variables relevant to the path to be traversed and removes those irrelevant. Dynamic variable ordering (DVO) permutates FV and returns a queue. Path tendency calculation (PTC) calculates the path tendencies of all relevant variables along the path, which will be used to calculate the domains in which their initial values are selected.

Initial domain calculation (IDC) calculates the domain of a variable in which its initial value is selected according to its path tendency calculated by PTC. Bisection reduces the domain of the current variable when its value just assigned fails to satisfy a constraint on the path. Maintaining path consistency (MPC) utilizes interval arithmetic to determine whether the domains of all variables satisfy the constraints along the path.

The overview of our approach can be seen from Figure 2. The path to be traversed is shown in the left part, where the red circles represent nodes and the arrows represent edges of the CFG. The path contains the constraints to be met, the set of input variables, and the domains corresponding to the variables. The first stage is to perform the initialization operations. At first, IVR (see Section 4.3) is called to reduce the search space by removing irrelevant variables and leaving only those relevant to the path. Then four heuristic look-ahead methods take effect. MPC (see Section 5.3) is used to partially reduce the input domains of all variables and find infeasible paths on occasion. All the relevant variables in FV are permutated by DVO (see Section 5.1) to form a queue Q_1 and its head x_1 is determined the best or the first variable to be instantiated. Next PTC (see Section 5.2) calculates path tendency of each variable, and IDC (see Section 5.2) reduces the domain D_{11} in which the initial value V_{11} is selected for x_1. With all these, the initial state is constructed as $(null, x_1, D_{11}, V_{11}, active,$ and $Q_1)$, which is also the current state, S_{cur}, shown as the red ring.

The second stage implements state space search. Four heuristic look-ahead methods work in this stage. To each active state, MPC is carried out to determine the direction of the next search step. If MPC succeeds, *Type* becomes *extensive*, the variables in FV will be permutated by DVO to get *Queue* $= Q_i$, S_{cur} becomes *Precursor*, and the head of $Q_i(x_i)$ will be *Variable* of next state. Then IDC is used to calculate the domain D_{i1} in which the initial value V_{i1} is selected for x_i. With all these, a new state $(Pre, x_i, D_{i1}, V_{i1}, active,$ and $Q_i)$ is constructed, for which the MPC check continues. If after a successful MPC check no variable needs to be permutated, then all the relevant variables have been assigned the right values to make p feasible. The final state is reached, shown as the red double ring. Finally giving the irrelevant variables random values fulfills the generation of the test data, which is the output of BFS-BB as shown in the right part of Figure 2. If a MPC check fails, *Type* remains *active*, bisection (see Section 5.2) is conducted to reduce the domain of *Variable* using the information from the failed MPC check, and *Value* is reselected from the reduced domain, all of which indicate that the search will expand to a state with a different value for the same variable x_i. If all the values within its domain for x_i are tried out or the number of MPC checks has reached the upperbound m, then x_i is moved out of PV and *Type* becomes *inactive*. In this case, the search will have to backtrack to *Precursor* at the higher level of the search tree, as shown by the bidirectional arrow between backtracking and the heuristic look-ahead methods. The above-mentioned search process is described by pseudocodes as Algorithm 1.

4.3. Irrelevant Variable Removal. As mentioned above, $X = \{x_1, x_2, \ldots, x_n\}$ is the set of input variables for a program P. The search space needs to involve every x_i $(i = 1, 2, \ldots, n)$ in X. However, it is possible that not every variable will be responsible for determining whether every path in P will be traversed or not. Therefore, when attempting to generate test data for a particular path p, the search effort on the value

```
**Input**  $p$: the path to be traversed
**Output** $result$ $\langle Variable, Value\rangle$: the test data making $p$ feasible
**Stage 1**: **Initialization**
(1)  call **Algorithm Irrelevant variable removal**;
(2)  $result \leftarrow null$;
(3)  call **Algorithm Maintaining path consistency**;
(4)  call **Algorithm Dynamic variable ordering**;
(5)  call **Algorithm Path tendency calculation**;
(6)  $x_1 \leftarrow$ head $(Q_1)$;
(7)  call **Algorithm Initial domain calculation**;
(8)  $V_{11} \leftarrow$ select $(D_{11})$;
(9)  initial state $\leftarrow (null, x_1, D_{11}, V_{11}, active, Q_1)$;
(10) $S_{\text{cur}} \leftarrow$ initial state;
**Stage 2**: **State space search**
**Begin**
(11) **for** $(Pre, x_i, D_{ij}, V_{ij}, active, Q_i)$ $(i \rightarrow 1{:}\,n)$
(12)     $path\ consistent \leftarrow false$;
(13)     call **Algorithm Maintaining path consistency**;
(14)     **if** $(path\ consistent = true)$
(15)         $S_{\text{cur}} \leftarrow (Pre, x_i, D_{ij}, V_{ij}, extensive, Q_i)$;
(16)         $result \leftarrow result \cup \{\langle x_i, V_{ij}\rangle\}$;
(17)         $\text{FV} \leftarrow \text{FV} - \{x_i\}$;
(18)         $\text{PV} \leftarrow \text{PV} + \{x_i\}$;
(19)         call **Algorithm Dynamic variable ordering**;
(20)         **if** $(Q_i = null)$
(21)             $S_{\text{cur}} \leftarrow$ final state;
(22)             **foreach** $x^* \in X_{\text{irrel}}$
(23)                 $result \leftarrow result \cup \{\langle x^*, V_{\text{random}}\rangle\}$;
(24)         **else** $Pre \leftarrow S_{\text{cur}}$;
(25)             $x_i \leftarrow$ head $(Q_i)$;
(26)             call **Algorithm Initial domain calculation**;
(27)             $V_{i1} \leftarrow$ select $(D_{i1})$;
(28)             $S_{\text{cur}} \leftarrow (Pre, x_i, D_{i1}, V_{i1}, active, Q_i)$;
(29)     **else if** $(|D_{ij}| > 1\ \&\&\ j < m)$
(30)             call **Algorithm Bisection**;
(31)             $V_{ij} \leftarrow$ select $(D_{ij})$;
(32)             $S_{\text{cur}} \leftarrow (Pre, x_i, D_{ij}, V_{ij}, active, Q_i)$;
(33)         **else** $S_{\text{cur}} \leftarrow (Pre, x_i, D_{ij}, V_{ij}, inactive, Q_i)$;
(34)             $Pre \leftarrow S_{\text{cur}}$;
(35)             $S_{\text{cur}} \leftarrow (Pre, x_i, D_{ij}, V_{ij}, active, Q_i)$;
(36)             $\text{PV} \leftarrow \text{PV} - \{x_i\}$;
(37) **return** $result$;
**End**
```

Algorithm 1: Best-first-search branch and bound.

of a variable which is not relevant to p is wasted since it cannot influence the traversal of p. Thus, removing irrelevant variables from the search space and only concentrating on the variables relevant to the path of interest may improve the performance of the search. Hence we propose an optimization method irrelevant variable removal (IVR). Relevant variable and irrelevant variable are defined as follows.

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

```
Input  Br($n_{qa}$, $n_{qa+1}$) ($a \in [1,k]$): k branching conditions along the path
       $X = \{x_1, x_2, \ldots, x_n\}$: the set of input variables
Output $X_{\text{rel}}$: the set of relevant variables to the path
       $X_{\text{irrel}}$: the set of irrelevant variables to the path
(1) $X_{\text{rel}} \leftarrow \emptyset$;
(2) $X_{\text{irrel}} \leftarrow \emptyset$;
(3) foreach Br($n_{qa}$, $n_{qa+1}$) ($a \in [1,k]$)
(4)         if ($X_{\text{rel}} = X$)
(5)             break;
(6)         else if ($a_j \neq 0$)
(7)             $X_{\text{rel}} \leftarrow X_{\text{rel}} \cup \{x_j\}$;
(8) $X_{\text{irrel}} \leftarrow X - X_{\text{rel}}$;
(9) return $X_{\text{rel}}$, $X_{\text{irrel}}$;
```

Algorithm 2: Irrelevant variable removal.

Generally, for a particular path, whether an input variable is relevant or irrelevant, cannot be completely decided due to the complex structure of programs. But we can make conservative estimate of irrelevancy with static control flow technique. We give the most common condition in PUTs. Assume that there are k branches along a path, each branch (n_{qa}, n_{qa+1}) $(a \in [1,k])$ needs to be traversed to find the set of relevant variables. The removal of irrelevant variables involves the judgment of whether a variable appears on each branch, so we give the definition below, which is utilized by Algorithm 2. And considering the relation between the complexity of BFS-BB and the number of variables, we give Proposition 4 about the effectiveness of IVR.

Definition 3. The branching condition $\text{Br}(n_{qa}, n_{qa+1})$ is the constraint on the branch (n_{qa}, n_{qa+1}) $(a \in [1,k])$, and it can be represented as

$$\text{Br}\left(n_{qa}, n_{qa+1}\right) = \sum_{j=1}^{n} a_j x_j \mathbb{R} c, \tag{1}$$

where $\mathbb{R}$ is a relational operator and a_j $(j \in [1,n])$ and c are constants.

Proposition 4. *IVR may result in test data being searched out with fewer MPC checks for a particular path p than if all variables are considered.*

Proof. The algorithm bisection involves the search steps taken for a certain variable under the same condition of other variables, which move in breadth (m) until a value is found to make MPC succeed. Then m is the base of the complexity of BFS-BB and the number of variables is the exponent. Let X_{rel} denote the set of relevant variables to path p, and let X_{irrel} be the set of irrelevant variables; one more element in X_{rel} will involve more MPC checks on an exponential basis. If all the irrelevant variables are removed from the search space, the complexity will be reduced by $m^{|X_{\text{irrel}}|}$. $|X_{\text{irrel}}|$ is the cardinality of the set of irrelevant variables. □

We conduct IVR for all the paths in Figure 1, and the process is shown in Table 1. The position where a variable is judged relevant to the path of interest is highlighted in bold.

5. The Heuristic Look-Ahead Methods

In this section, the heuristic look-ahead methods in BFS-BB are explained in detail in Sections 5.1, 5.2, and 5.3, respectively. And Section 5.4 provides a case study to illustrate these methods.

5.1. Heuristics in Variable Ordering. In practice, the chief goal in designing variable ordering heuristics is to reduce the size of the overall search tree. In our method, the next variable to be instantiated is selected to be the one with the minimal remaining domain size (the size of the domain after removing the values judged to be infeasible), because this can minimize the size of the overall search tree. The technique to break ties is important, as there are often variables with the same domain size. We use variables' ranks to break ties. In case of a tie, the variable with the higher rank is selected. This method gives substantially better performance than picking one of the tying variables at random. Rank is defined as follows.

Definition 5. The rank of a branch (n_{qa}, n_{qa+1}) $(a \in [1,k])$ marks its level in the sequence of the branches along a path, denoted as rank (n_{qa}, n_{qa+1}).

The rank of the first branch is *1*, the rank of the second one is *2*, and the ranks of those following can be obtained analogously. The variables appearing on a branch enjoy the same rank as the branch. The rank of a variable on a branch where it does not appear is supposed to be *infinity*. As a variable may appear on more than one branch, it may have different ranks. The rule to break ties according to the ranks of variables is based on the heuristics from interval arithmetic that the earlier a variable appears on a path, the greater influence it has on the result of interval arithmetic along the path. Therefore, if the ordering by rank is taken between a variable that appears on the branch (n_{qa}, n_{qa+1}) and a variable that does not, then the former has a higher rank. That is because, on the branch (n_{qa}, n_{qa+1}), the former has rank a,

Table 1: IVR process for each path of test in Figure 1.

Path	Branching condition	a_1	a_2	a_3	X_{rel}	X_{irrel}
Path 1: $0 \to 1 \to 2 \to 9 \to 10$	$x1 - x2 \le 0$	**1**	**−1**	0	$\{x1, x2\}$	$\{x3\}$
Path 2: $0 \to 1 \to 3 \to 4 \to 8 \to 9 \to 10$	$x1 - x2 > 0$	**1**	**−1**	0	$\{x1, x2, x3\}$	$\varnothing$
	$x3 - x2 \le 0$	0	−1	**1**		
Path 3: $0 \to 1 \to 3 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$	$x1 - x2 > 0$	**1**	**−1**	0	$\{x1, x2, x3\}$	$\varnothing$
	$x3 - x2 > 0$	0	−1	**1**		
	$3x3 \ge -5$	—	—	—		
Path 4: $0 \to 1 \to 3 \to 5 \to 7 \to 8 \to 9 \to 10$	$x1 - x2 > 0$	**1**	**−1**	0	$\{x1, x2, x3\}$	$\varnothing$
	$x3 - x2 > 0$	0	−1	**1**		
	$3x3 < -5$	—	—	—		

```
Input   FV: the set of future variables
        $D_i$: the domain of $x_i$ ($x_i \in$ FV)
        $(n_{qa}, n_{qa+1})$ ($a \in [1, k]$): $k$ branches along the path
Output  $Q_i$: a queue of FV
Begin
(1) $Q_i \leftarrow$ quicksort (FV, $|D_i|$);
(2) for $i \to 1$: $|Q_i|$
(3)     if $(|D_i| \ne |D_j|)$ ($j > i, x_i, x_j \in Q_i$)
(4)         break;
(5)     else for $(n_{qa}, n_{qa+1})$ ($a \in [1, k]$)
(6)             if (rank$(n_{qa}, n_{qa+1})(x_i)$ = rank$(n_{qa}, n_{qa+1})(x_j)$)
(7)                 $a$++;
(8)             else permutate $x_i, x_j$ by rank$(n_{qa}, n_{qa+1})$;
(9)                 break;
(10) return $Q_i$;
End
```

Algorithm 3: Dynamic variable ordering.

while the latter has rank *infinity*. The comparison between a and *infinity* determines the ordering. The algorithm is described by pseudocodes in Algorithm 3.

Quicksort is utilized when permutating variables according to remaining domain size and returns Q_i as a result. If no variables have the same domain size, then DVO finishes. But if there are variables whose domain sizes are the same as that of the head of Q_i, then the ordering by rank is under way, which will terminate as soon as different ranks appear.

5.2. Heuristics in Value Selection. DVO determines the next variable to be instantiated and then the value selection strategies are employed. Considering the difference between the variable in question (e.g., x_i) and other variables, the branching condition defined by formula (1) can be further represented as a function of x_i:

$$\operatorname{Br}(n_{qa}, n_{qa+1})(x_i) : D_i \longrightarrow B = \left(a_i x_i + \sum_{j \ne i} a_j x_j\right) \mathbb{R} c, \quad (2)$$

where D_i is the domain of x_i and B is a set of Boolean values $\{true, false\}$. $\sum_{j \ne i} a_j x_j$ is the linear combination of the variables except x_i and is regarded as a constant. Then we can design the value selection strategies, starting from the monotonic relation between the branching condition and x_i. Monotonicity describes the behavior of a function in relation to the change of the input. It gives an indication whether the output of the function moves in the same direction as the input or in the reverse direction. If a branching condition is a function whose monotonicity is known, the direction in which the input needs to be moved to make the function *true* can be determined. The following proposition gives an attribute of a function composed of piecewise monotonic functions.

Proposition 6. *Assume that $f_1 : X_1 \to Y_1$, $f_2 : X_2 \to Y_2, \ldots, f_m : X_m \to Y_m$ is a family of piecewise monotonic functions with $Y_i \subseteq X_{i+1}$. Let $F_m : X_1 \to Y_m$ be a composed function $f_m \circ f_{m-1} \circ \cdots \circ f_1$. On this assumption, F_m is also piecewise monotonic.*

Proof. Mathematical induction is used to prove the proposition.

(i) Case $F_1 = f_1$. Function f_1 is piecewise monotonic by assumption. F_1 is equal to f_1, so it has the same attribute.

(ii) Case $F_{i+1} = f_{i+1} \circ F_i$. The composed function F_i is piecewise monotonic by the induction assumption. let I be a subset of its domain's partition, and let x and x' be two arbitrary elements in I with $x \le_X x'$; then one of the monotonicity conditions holds; that is, either $F_i(x) \le_{Y_i} F_i(x')$ or $F_i(x) \ge_{Y_i} F_i(x')$. For simplicity, we denote it as $F_i(x) \mathfrak{R} F_i(x')$, where $\mathfrak{R} \in \{\le, \ge\}$. Function f_{i+1} is piecewise monotonic by assumption. The monotonicity condition is satisfied by $F_i(x)$ and $F_i(x')$ if both lie in the same subset $I'$of its domain's partition. Then $f_{i+1}(x) \mathfrak{R} f_{i+1}(x')$ holds and f_{i+1} is also monotonic on I'.

After decomposing a branching condition into its basic functions, its monotonicity can be utilized in the selection of the initial value as well as other values of the variable in question. □

5.2.1. Initial Value Selection. Initial values of variables are of great importance to a search algorithm. On the one hand, in a backtrack-free search, the initial value of a variable is almost part of the solution. On the other hand, the selection of initial values affects whether the search will be backtrack-free. Initial values are often selected at random in MHS methods, which return different test data each time allowing diversity but randomness without any heuristics is a kind of blind search and causes too many iterations, sometimes even exception. Meanwhile, midvalues are selected in methods using bisection, so it is obvious that sometimes the same result may be returned since the same initial value is always selected. In our method, the above two methods are combined, and the initial value of a variable is determined based on its path tendency, which is defined and calculated as follows.

Definition 7. Path tendency $\in \{positive, negative\}$ is an attribute of a variable on a path, which is in favor of the satisfaction of all the branching conditions along the path. And it provides the information about where to select its initial value. *Positive* implies that a larger initial value will work better, while *negative* implies that a smaller initial value is better.

The calculation of the path tendency of a variable x_i involves the calculation of its weight on each branch (n_{qa}, n_{qa+1}) $(a \in [1,k])$ and its path weight, denoted as $w_i(n_{qa}, n_{qa+1})$ and pw_i, which are calculated as (3)

$$w_i\left(n_{qa}, n_{qa+1}\right) = \begin{cases} \dfrac{|a_i|}{|a_i| + \sum_{j\neq i}|a_j|}, & \text{if } \mathrm{Br}\left(x_i\right)\left(n_{qa}, n_{qa+1}\right) \\ & \text{is monotonically increasing} \\ -\dfrac{|a_i|}{|a_i| + \sum_{j\neq i}|a_j|}, & \text{if } \mathrm{Br}\left(x_i\right)\left(n_{qa}, n_{qa+1}\right) \\ & \text{is monotonically decreasing} \end{cases}$$

$$pw_i = \sum_{a=1}^{k} w_i\left(n_{qa}, n_{qa+1}\right). \tag{3}$$

Path tendency calculation (PTC) gleans the path tendency of each variable with pw_i. Subsequently, initial domain calculation (IDC) works on the result of PTC. In this way, the initial value selection allows for both diversity and heuristics. The algorithms are expressed by pseudo-codes in Algorithms 4 and 5.

5.2.2. Bisection by Tendency.

Bisection functions only when a value (including the initial value) assigned to the current variable x_i is judged to be infeasible and the conflicted branch (n_{qa}, n_{qa+1}) with the *false* branching condition is located. Then the tendency of x_i is used by bisection, defined as follows.

Definition 8. Tendency $\in \{positive, negative\}$ is an attribute of a variable at a branch (n_{qa}, n_{qa+1}) $(a \in [1,k])$ determined by the analysis on the monotonicity of the corresponding branching condition, and it provides the information about where to select a value to better satisfy the branching condition. *Positive* implies that a larger value will work better, while *negative* implies that a smaller value is better. It is calculated according to the following formula:

$$Tendency\left(x_i\right) = \begin{cases} positive, & \text{if } \mathrm{Br}\left(x_i\right)\left(n_{qa}, n_{qa+1}\right) \\ & \text{is monotonically increasing} \\ negative, & \text{if } \mathrm{Br}\left(x_i\right)\left(n_{qa}, n_{qa+1}\right) \\ & \text{is monotonically decreasing.} \end{cases} \tag{4}$$

Each branch holds a tendency map $\langle Variable, Tendency \rangle$, which includes the variables appearing on the branch and their corresponding tendencies. With the tendency map, bisection can be applied to reduce the domain of $x_i(D_{ij})$, leading the branching condition to be true as presented by pseudo-codes in Algorithm 6.

For example, if the conflicted branch is the first branch of *Path3* in Figure 1, then the corresponding branching condition is $x1 - x2 > 0$, which has different monotonic relations with $x1$ and $x2$, respectively. Table 2 shows how to use bisection to reduce the domains of variables. If the current variable is $x1$, then retrieval of tendency map returns *positive* indicating that a larger value will help satisfy the branching condition, so we reduce its domain to the larger part. But if the current variable is $x2$, bisection will function in the opposite way due to the opposite monotonic relation.

5.3. Heuristics in Maintaining Path Consistency.

As mentioned in Section 4.2, MPC can be used in both stages of BFS-BB. In this part, the focus is on the state space search stage. A value assigned to the current variable x_i, no matter it is the initial value or another value selected after bisection, should be examined by interval arithmetic to see whether it is part of the solution. Path consistency is a prerequisite for the success of interval arithmetic. In the implementation of BFS-BB, interval arithmetic is enhanced to provide more precise interval information. The enhancement is to make clear how the value of the branching condition defined by formula (2) is calculated, as shown in formula (5). Here we use D^a to denote the domain of all variables before calculating the ath branching condition. Besides, a library of inverse functions is added in case of the occurrences of library functions in the PUT. Consider

$$\mathrm{Br}\left(n_{qa}, n_{qa+1}\right)\left(x_i\right) = \begin{cases} true, & \text{if } \left(n_{qa}, n_{qa+1}\right) \text{ is traversed} \\ & \text{with } D^a\left(V_{ij} \in D^a\right) \\ false, & \text{otherwise.} \end{cases} \tag{5}$$

Hence, for k branching nodes along path p, all the k branching conditions should be *true* to maintain path consistency. MPC receives the value of the current variable x_i (V_{ij}), which is part of the domain of all variables denoted as D^1 $(V_{ij} = [V_{ij}, V_{ij}] \in D^1)$ and evaluates the branching condition corresponding to the branch (n_{q1}, n_{q1+1}), where n_{q1} is the first branching node. The branching condition $\mathrm{Br}(n_{q1}, n_{q1+1})$

```
Input $X_{\text{rel}}$: the set of relevant variables to the path
      $pw_i$: the path weight of variable $x_i$ ($x_i \in X_{\text{rel}}$)
Output Path-Tendency ⟨Variable, PathTendency⟩: a map used to store the path tendency of each variable in $X_{\text{rel}}$
Begin
(1) Path-Tendency ← null;
(2) foreach $x_i \in X_{\text{rel}}$
(3)         if ($pw_i > 0$)
(4)                 Path-Tendency ← Path-Tendency ∪ {⟨$x_i$, positive⟩};
(5)         else if ($pw_i < 0$)
(6)                 Path-Tendency ← Path-Tendency ∪ {⟨$x_i$, negative⟩};
(7) return Path-Tendency;
End
```

Algorithm 4: Path tendency calculation.

```
Input  $D_i$ = [min, max]: the domain of $x_i$
       Path-Tendency ⟨Variable, PathTendency⟩: a map used to store the path tendency of each variable in $X_{\text{rel}}$
Output $D_{i1}$: the domain of $x_i$ in which its initial value is selected
Begin
(1) PathTendency($x_i$) ← retrieval of Path-Tendency;
(2) if (PathTendency($x_i$) = positive)
(3)         $D_{i1} \longleftarrow \left[\frac{(\min + \max)}{2}, \max\right]$;
(4) else if (PathTendency($x_i$) = negative)
(5)         $D_{i1} \longleftarrow \left[\min, \frac{(\min + \max)}{2}\right]$;
(6) return $D_{i1}$;
End
```

Algorithm 5: Initial domain calculation.

```
Input  $D_{ij}$ = [min, max]: the current domain of $x_i$
       $V_{ij}$: the current value of $x_i$ that causes $\text{Br}(x_i)(n_{qa}, n_{qa+1})$ to be false
       $(n_{qa}, n_{qa+1})$: the conflicted branch
Output $D_{ij}$: the reduced domain of $x_i$
Begin
(1) $V' \leftarrow V_{ij}$;
(2) Tendency($x_i$) ← retrieval of tendency map held by $(n_{qa}, n_{qa+1})$;
(3) j++;
(4) if (Tendency($x_i$) = positive)
(5)         $D_{ij} \leftarrow [V' + 1, \max]$;
(6) else if (Tendency($x_i$) = negative)
(7)         $D_{ij} \leftarrow [\min, V' - 1]$;
(8) return $D_{ij}$;
End
```

Algorithm 6: Bisection.

Table 2: An example of bisection.

Current variable	Monotonicity	Tendency	Current value	Domain before bisection	Domain after bisection
$x1$	Increasing	Positive	$V1$	[min1, max1]	[$V1$+1, max1]
$x2$	Decreasing	Negative	$V2$	[min2, max2]	[min2, $V2$–1]

```
Input   $D^1$: the domain of all variables before checking path consistency
        $\mathrm{Br}(n_{qa}, n_{qa+1})$ ($a \in [1, k]$): $k$ branching conditions along the path
Output  $D^{k+1}$: the reduced domain of all variables after a successful path consistency check
        $(n_{qa}, n_{qa+1})$: the conflicted branch spotted by path consistency check
Begin
(1) for $a \rightarrow 1{:}\ k$
(2)     calculate $\mathrm{Br}(n_{qa}, n_{qa+1})$ with $D^a$;
(3)     if ($\mathrm{Br}(n_{qa}, n_{qa+1}) = true$)
(4)         $D^{a+1} \sqsubseteq D^a$;
(5)     else return $(n_{qa}, n_{qa+1})$;
(6) path consistent $\leftarrow$ true;
(7) return $D^{k+1}$;
End
```

Algorithm 7: Maintaining path consistency.

is generally not satisfied for all the values in D^1 but for values in a certain subset $D^2 \subseteq D^1$ ensuring the traversal of the branch (n_{q1}, n_{q1+1}); that is, $D^1 \xrightarrow{Br(n_{q1},n_{q1+1})} D^2$. Next, the branching condition $\mathrm{Br}(n_{q2}, n_{q2+1})$ is evaluated given that the domain of all variables is D^2. Again, generally, $\mathrm{Br}(n_{q2}, n_{q2+1})$ is only satisfied by a subset $D^3 \subseteq D^2$. This procedure continues along p until all the branching conditions are satisfied to maintain path consistency and D^{k+1} is returned as the domain of all variables. The process of maintaining path consistency is the propagation of the branching conditions along p in the form of $D^1 \xrightarrow{\mathrm{Br}(n_{q1},n_{q1+1})} D^2 \xrightarrow{\mathrm{Br}(n_{q2},n_{q2+1})} D^3 \cdots D^k \xrightarrow{\mathrm{Br}(n_{qk},n_{qk+1})} D^{k+1}$, where $D^1 \supseteq D^2 \supseteq D^3 \cdots \supseteq D^k \supseteq D^{k+1}$. But if in this procedure $\mathrm{Br}(n_{qh}, n_{qh+1}) = false (1 \le h \le k)$, which means a conflict is detected, then MPC is terminated and bisection will function according to the result of MPC at the conflicted branch (n_{qh}, n_{qh+1}). The process of checking whether path consistency is maintained is shown by pseudocodes in Algorithm 7.

5.4. Case Study. In this part, the problem mentioned in Section 3.1 is used as an example to explain how BFS-BB works, especially the heuristic look-ahead methods proposed ahead. The input is *Path3* as shown in bold in Figure 3, where each branching condition is decomposed into its basic functions in the right. The IVR process has been illustrated in detail in Table 1, and all the three variables are determined relevant to *Path3*. For simplicity, the input domains of all variables are set $[-2, 2]$ with the size 5. In the initialization stage, MPC check reduces their domains to $x1$: $[-1, 2]$, $x2$: $[-2, 1]$, and $x3$: $[-1, 2]$. The path tendency of each variable is calculated by PTC as shown in Table 3. DVO serves to determine the first variable to be instantiated as shown in Table 4, with the head of the queue ($x2$) highlighted in bold. On determining $x2$ to be the current variable, an initial value needs to be selected from$[-2, 1]$. The retrieval of path tendency map by IDC returns *negative* for $x2$, indicating that a smaller value will perform better and -1 is selected.

MPC checks the domains of all variables which are $x1$: $[-1, 2]$, $x2$: $[-1, 1]$, and $x3$: $[-1, 2]$. It succeeds and reduces the domains of $x1$ and $x3$ to $[0, 2]$ and $[0, 2]$, respectively. Then DVO determines the next variable to be instantiated as shown in Table 5, with the head of the queue ($x1$) highlighted in bold.

1 is selected for $x1$ after IDC. MPC checks whether $x1$: $[1, 1]$, $x2$: $[-1, -1]$, and $x3$: $[0, 2]$ works. It succeeds and in the same manner $x3$ is assigned 1. Finally, $\{\langle x1, 1\rangle, \langle x2, -1\rangle, \langle x3, 1\rangle\}$ is checked by MPC to be suitable for *Path3*. No variable needs to be permutated and BFS-BB succeeds with the test data $\{\langle x1, 1\rangle, \langle x2, -1\rangle, \langle x3, 1\rangle\}$. Table 6 shows how the domains of variables are changed during the search process. The changed domains are highlighted in bold. The changes listed in the fourth column are owing to variable assignments according to the results of IDC, and the changes listed in the fifth column are owing to domain reduction by MPC checks. The process of generating the test data $\{\langle x1, 1\rangle, \langle x2, -1\rangle, \langle x3, 1\rangle\}$ is presented as the search tree in Figure 4. It is a backtrack-free search that accounts for an extremely large proportion in the implementation of BFS-BB. Each variable consumes one MPC check in the state space search stage, and the initial values of each variable make the solution. The solution path is shown by the bold arrows.

6. Experimental Results and Discussion

To observe the effectiveness of BFS-BB, we carried out a large number of experiments in CTS. Within the CTS framework, the PUT is automatically analyzed, and its basic information is abstracted to generate its CFG. According to the specified coverage criteria, the paths to be traversed are generated and provided for BFS-BB as input. The generated test data will be used for mutation testing that requires a high coverage, ideally 100% [37]. This is a challenge for test data generation.

The experiments were performed in the environment of MS Windows 7 with 32 bits, Pentium 4 with 2.8 GHz and 2 GB memory. The algorithms were implemented in Java and run on the platform of eclipse. The experiments include two parts. Section 6.1 presents the performance evaluation of BFS-BB, and Section 6.2 tests the capability of BFS-BB to generate test data in terms of coverage and makes comparisons with some currently existing static and dynamic methods.

Table 3: PTC process for $x1$, $x2$, and $x3$.

Branching condition	Basic functions and corresponding monotonicity	Monotonicity of branching conditions	Weight	Path weight	Path tendency
$x1 - x2 > 0$	$f(x1) = x1 - x2$: increasing $f(x2) = x1 - x2$: decreasing $f(b1) = b1 > 0$: increasing	Br($x1$): increasing Br($x2$): decreasing	$w1 = 0.5$ $w2 = -0.5$	$pw1 = 0.5$ $pw2 = -1$ $pw3 = 1.5$	$\{\langle x1$, positive$\rangle$, $\langle x2$, negative$\rangle$, $\langle x3$, positive$\rangle\}$
$x3 - x2 > 0$	$f(x2) = x3 - x2$: decreasing $f(x3) = x3 - x2$: increasing $f(b2) = b2 > 0$: increasing	Br($x2$): decreasing Br($x3$): increasing	$w2 = -0.5$ $w3 = 0.5$		
$3 * x3 \geq -5$	$f(x3) = 3 * x3$: increasing $f(b3) = b3 \geq -5$: increasing	Br($x3$): increasing	$w3 = 1$		

Table 4: DVO process for $x1$, $x2$, and $x3$.

Ordering rule	Condition for each variable	Tie encountered?	Ordering result
Domain size	$\|D1\| = 4$, $\|D2\| = 4$, $\|D3\| = 4$	Yes (all three have the same domain size)	$\mathbf{x2} \rightarrow x1 \rightarrow x3$
Rank 1	Rank 1($x1$) = 1, Rank 1($x2$) = 1, Rank 1($x3$) = ∞	Yes ($x1$ and $x2$ both have Rank 1)	
Rank 2	Rank 2($x1$) = ∞, Rank 2($x2$) = 2	No ($x2$ has Rank 2 while $x1$ has infinity)	

Table 5: DVO process for $x1$ and $x3$.

Ordering rule	Condition for each variable	Tie encountered?	Ordering result
Domain size	$\|D1\| = 3$, $\|D3\| = 3$	Yes (both have the same domain size)	$\mathbf{x1} \rightarrow x3$
Rank 1	Rank 1($x1$) = 1, Rank 1($x3$) = ∞	No ($x1$ has Rank 1 while $x3$ has infinity)	

Table 6: Domain changes in the search process.

Stage	Function	Before IDC	After IDC and before MPC	After MPC
Initialization	Initial domain reduction	—	$x1$: [−2, 2], $x2$: [−2, 2], $x3$: [−2, 2]	**x1**: [**−1**, **2**], **x2**: [**−2**, **1**], **x3**: [**−1**, **2**]
State space search	MPC check when $x2$ is assigned −1	$x1$: [−1, 2], $x2$: [−2, 1], $x3$: [−1, 2]	$x1$: [−1, 2], **x2**: [**−1**, **−1**], $x3$: [−1, 2]	**x1**: [**0**, **2**], $x2$: [−1, −1], **x3**: [**0**, **2**]
	MPC check when $x1$ is assigned 1	$x1$: [0, 2], $x2$: [−1, −1], $x3$: [0, 2]	**x1**: [**1**, **1**], $x2$: [−1, −1], $x3$: [0, 2]	$x1$: [1, 1], $x2$: [−1, −1], $x3$: [0, 2]
	MPC check when $x3$ is assigned 1	$x1$: [1, 1], $x2$: [−1, −1], $x3$: [0, 2]	$x1$: [1, 1], $x2$: [−1, −1], **x3**: [**1**, **1**]	$x1$: [1, 1], $x2$: [−1, −1], $x3$: [1, 1]

```
void test(int x1, int x2, int x3)
{ if (x1-x2<=0)
      printf("Path1");
  else if(x3-x2<=0)
          printf("Path2");
      else if(3*x3+5>=0)
              printf("Path3");
}
```

```
void test(int x1, int x2, int x3)
{ int b1=x1-x2;
  if (b1<=0)
      printf("Path1");
     else {int b2=x-3x2;
          if(b2<=0)
               printf("Path2");
          else{int b3=3*x3;
               if(b3>=-5)
                  printf("Path3");
             }
        }
}
```

Figure 3: Overview of our approach for searching the test data.

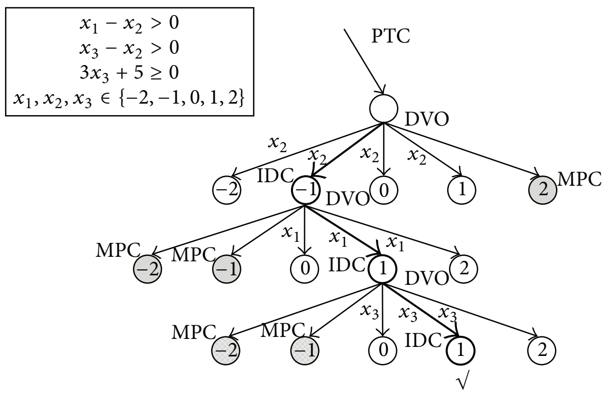


FIGURE 4: The search tree of generating the test data for *Path3* using BFS-BB.

6.1. Performance Evaluation. The number of relevant variables is an important factor that affects the performance of BFS-BB, so in this part experiments were carried out to evaluate the performance of BFS-BB for varying numbers of input variables. To be specific, our major concern is (1) the relationship between the number of MPC checks (exclusive of the one taken in the initialization stage) and the number of relevant variables; (2) the relationship between generation time and the number of relevant variables. This was accomplished by repeatedly running BFS-BB on generated test programs having input variables $x_1, x_2, \ldots, x_n$ where n varied from 1 to 50. Adopting statement coverage, in each test the program contained five if statements (equivalent to five branching conditions along the path for MPC check) and there was only one path to be traversed of fixed length, which was the one consisting of entirely true branches (TTTTT); that is, all the branching conditions are the same as the corresponding predicates. Considering the relationship between variables, experiments involving two situations were conducted that (1) the variables are all independent of each other and (2) the variables are linearly related in the tightest manner. Generation time varied greatly in these two cases, so the axes of generation time of both cases are normalized for simplicity.

6.1.1. Variables Are All Independent of Each Other. The predicate of each if statement is an expression in the form of

$$\begin{aligned} &a_1 x_1 \operatorname{rel}_{\mathrm{op}_1} \mathit{const}[1] \wedge a_2 x_2 \operatorname{rel}_{\mathrm{op}_2} \mathit{const}[2] \\ &\quad \wedge \cdots \wedge a_n x_n \operatorname{rel}_{\mathrm{op}_n} \mathit{const}[n], \end{aligned} \tag{6}$$

where $a_1, a_2, \ldots, a_n$ are randomly generated numbers either positive or negative, $\operatorname{rel}_{\mathrm{op}_i}$ $(i = 1, 2, \ldots, n) \in \{>, \geq, <, \leq, =, \neq\}$, and *const* is an array of randomly generated constants. The randomly generated a_i and $\mathit{const}[i]$ should be selected to make the path feasible. This arrangement constructs a relationship that all the variables are independent of each other but all of them are relevant to the path. The programs for various values of n ranging from 1 to 50 were each tested 50 times, and the number of MPC checks and time required to generate the data for each test were recorded. The results can be seen from Figures 5 and 6.

Figure 5 shows the relationship between the number of MPC checks and the number of variables (n) for variables that are all independent of each other, and from (a) to (d) represent four different situations, marked by the ordinates. It can be seen that since the relation in formula (6) is the simplest one between variables, the number of MPC checks remains linearly increasing with the number of variables no matter in which situation, from (a) to (d). $y = x$ means that, for this kind of constraint, one relevant variable requires only one MPC check. It also can be seen that $R^2 = 1$ in all the four situations and the number of MPC checks increases completely linearly with the number of variables.

Figure 6 shows the relationship between generation time and the number of variables (n) for variables that are all independent of each other, and from (a) to (d) represent four different situations marked by the ordinates. It can be seen that generation time increases approximately linearly with the number of variables and the linear correlation relationship is significant at 95% confidence level with P value far less than 0.05. By the increase of the number of variables, generation time increases at an even speed. The minimum value can be commendably represented as a straight line, showing that it is the most ideal in the four situations with a larger value of R^2. Variations between tests with the same values of n were attributed to the randomness in the difference in the selection of the initial values.

6.1.2. Variables Are Linearly Related in the Tightest Manner. The predicate of each if statement is a linear combination of all the n variables in the form of

$$[a_1, a_2, \ldots, a_n][x_1, x_2, \ldots, x_n]' \operatorname{rel}_{\mathrm{op}} \mathit{const}[c], \tag{7}$$

where $a_1, a_2, \ldots, a_n$ are randomly generated numbers either positive or negative, $\operatorname{rel}_{\mathrm{op}} \in \{>, \geq, <, \leq, =, \neq\}$, and $\mathit{const}[c]$ $(c \in \{1, 2, 3, 4, 5\})$ is an array of randomly generated constants. The randomly generated a_i and $\mathit{const}[c]$ should be selected to make the path feasible. This arrangement constructs the tightest linear relation between the variables, all

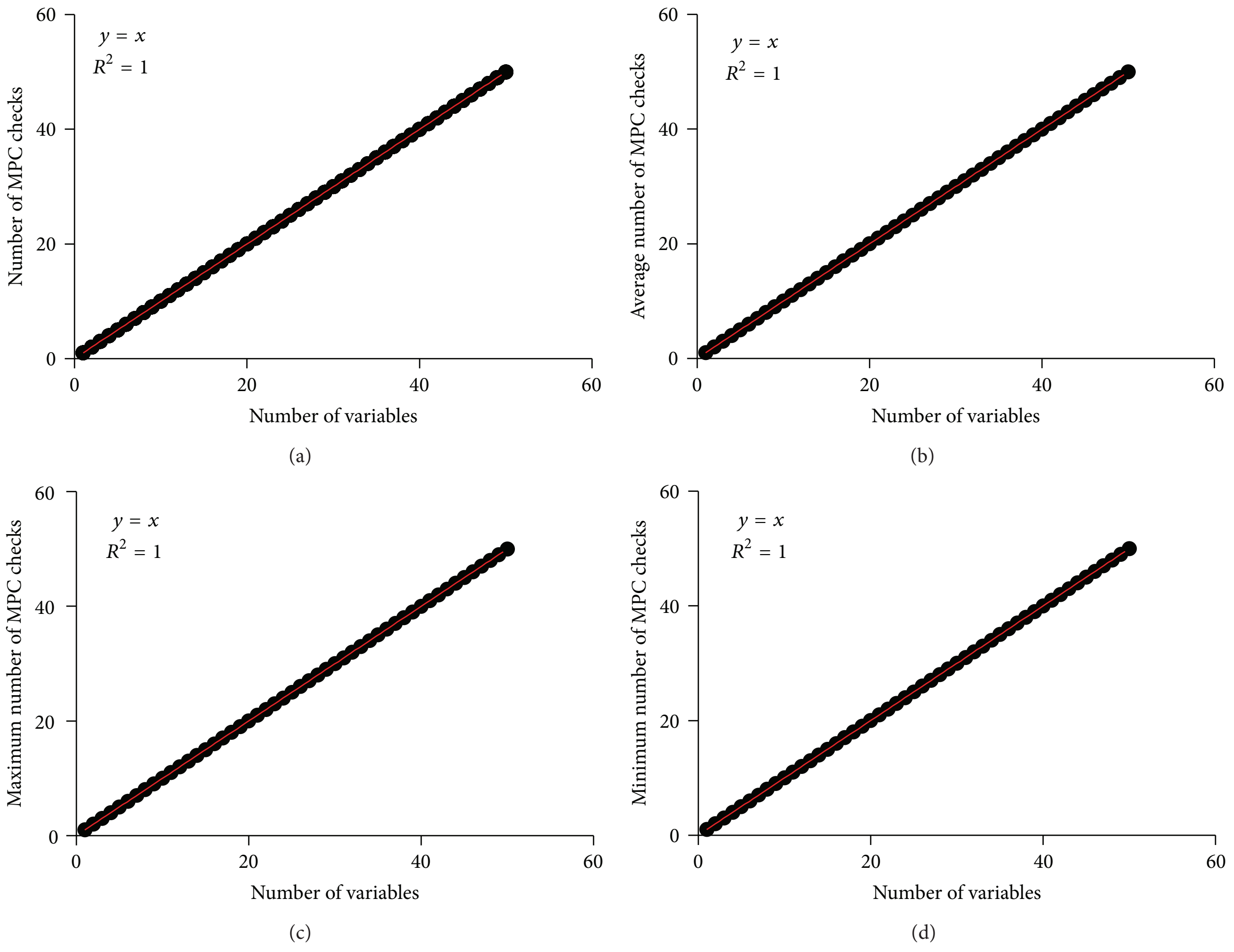


Figure 5: Relationship between the number of MPC checks and the number of variables for variables that are all independent of each other.

of which are relevant to the path. The programs for various values of n ranging from 1 to 50 were each tested 50 times and the number of MPC checks and time required to generate the data for each test were recorded. The results can be seen from Figures 7 and 8.

Figure 7 shows the relationship between the number of MPC checks and the number of variables (n) for variables that are the tightest linearly related, and from (a) to (d) represent four different situations marked by the ordinates. It can be seen that the number of MPC checks remains approximately linearly increasing with the number of variables and the fitting curves are all near $y = x$. The linear correlation relationship is significant at 95% confidence level with P-value far less than 0.05. The general, average, and maximum numbers of MPC checks are all larger than those in the experiment for variables that are all independent of each other, because the relation in formula (7) is the tightest linear one between variables. The minimum number of MPC checks can be completely represented as $y = x$ with $R^2 = 1$, which means that the minimum number is the most ideal in the four situations.

Figure 8 shows the relationship between generation time and the number of variables (n) for variables that are the tightest linearly related, and from (a) to (d) represent four different situations marked by the ordinates. It is clear that the relation between generation time and the number of variables can be commendably represented as a quadratic curve and the quadratic correlation relationship is significant at 95% confidence level with P-value far less than 0.05. The better fitting curves of average and minimum generation times show that average generation time is perfectly stable and minimum generation time is still the most ideal. Variations between tests with the same values of n were attributed to the randomness in (1) the difference in the selection of the initial values and (2) the difference in the expressions along the path (an equality relational operator will generally require more calculation than an inequality relational operator). Besides, generation time increases at a uniformly accelerative speed by the increase of the number of variables. Take (b) for example, the differentiation of average generation time indicates that its increase rate rises by $y = 9.994x - 77.34$ as the number of variables increases. We can roughly draw the conclusion that generation time is very close for n ranging from 1 to 8, while it begins to increase when n is larger than 8.

The above cases are both completely backtrack-free search owing to the linear correlation relationship between

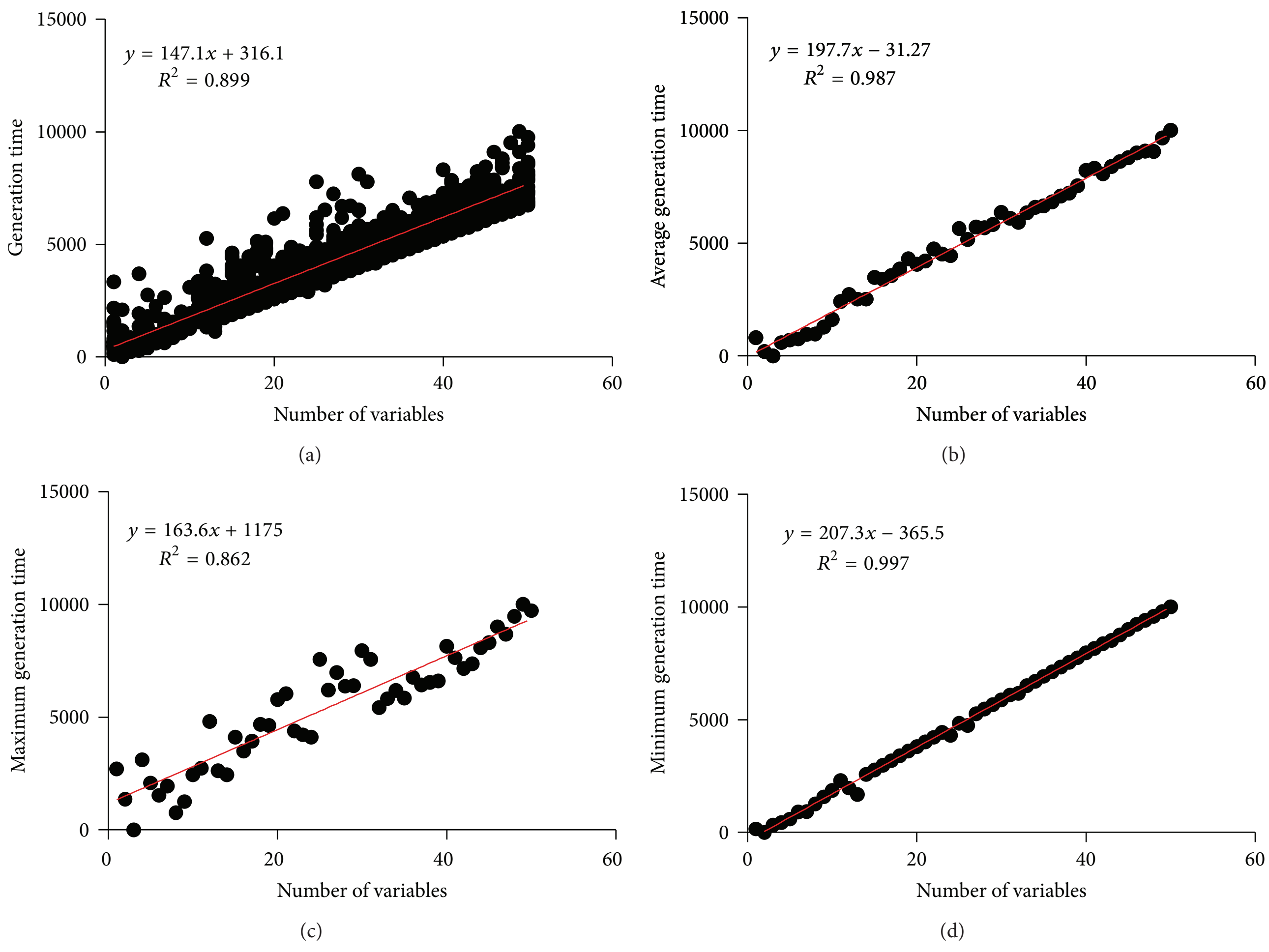


Figure 6: Relationship between generation time and the number of variables for variables that are all independent of each other.

the number of MPC checks and the number of relevant variables. Surely they cannot include all the relations between variables in engineering, so the analyses in this part are just from the theoretic perspective. The real-world PUTs are much more complex. What's more, 50 tests were conducted for each case of n ranging from 1 to 50. So the results from the samples can only approximate the actual situation. But it can be concluded that BFS-BB functions are stably given a PUT of regular structure, which lays a solid foundation for its application in engineering.

6.2. Coverage Evaluation. To evaluate the capability of BFS-BB to generate test data in terms of coverage, four experiments were carried out. The first involves the testing with a benchmark used in CTS, the second aims at generating test data for a project in engineering, the third compares BFS-BB with a static method, and the last compares it with dynamic methods.

6.2.1. Testing a Benchmark in CTS. In this part, test data were automatically generated to meet three coverage criteria which were statement, branch, and MC/DC. The test bed was branch_bound.c, a benchmark in CTS with 402 LOC, 29 input variables, and complex structure trying to include more content that might appear in engineering. m was set 10 for each variable as the upperbound of the number of MPC checks, so it can be estimated that the simplest backtracking will consume at least 11 MPC checks for the variable in question.

The result is shown in Table 7. The numbers of paths was different owing to different coverage criteria adopted. BFS-BB was able to generate test data for all the feasible paths, no matter which coverage criterion was taken. The MC/DC coverage did not reach 100%, because it is relatively strict and difficult to meet and subsumes statement and branch coverage [38]. But tolerable coverage was achieved within tolerable time. There exists a trade-off between efficiency and success rate. IVR had no significant influence on coverage, but it did on generation time. Generation time after IVR was much less than that without IVR. Note that the amount of generation time reduced by IVR is determined by the structure of the PUT. The numbers of generation time reduction in Table 7 are only related to the program branch_bound.c. Our following analyses all concern BFS-BB with IVR. Average number of MPC checks per relevant variable adopting statement coverage was larger than the results in Section 6.1,

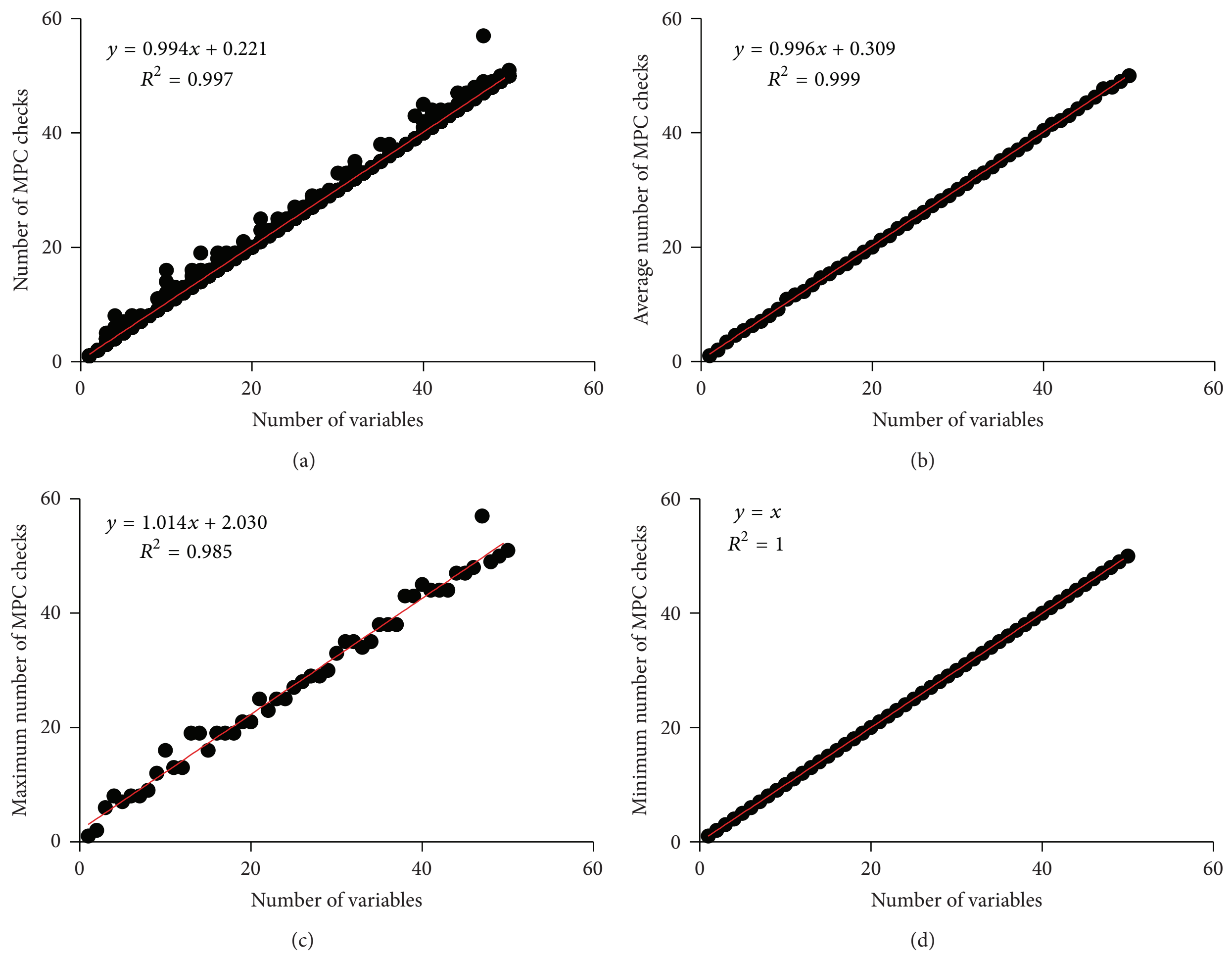


Figure 7: Relationship between the number of MPC checks and the number of variables for variables that are linearly related in the tightest manner.

Table 7: Coverage achieved by BFS-BB on branch_bound.c.

Coverage criterion	Number of paths	Average coverage %	Generation time reduced by IVR %	Average MPC checks per relevant variable
Statement	61	100	34	1.34
Branch	119	100	37	1.73
MC/DC	125	98	42	2.29

because there are some library functions as well as nonlinear constraints in the PUT, which require more MPC checks. But from all the average values of the number of MPC checks, it can be concluded that the tests for all the three coverage criteria were basically backtrack-free, and not all the tests include the bisection operation.

6.2.2. Testing Programs from a Project in Engineering. In this part, seven programs were selected from the project de118i-2 at http://www.moshier.net/ as the test beds, each of which contains several functions or loops. Three coverage criteria were adopted, which were statement, branch, and MC/DC. The information of the programs and the coverage results are shown in Table 8.

From Table 8, it can be seen that BFS-BB performed encouragingly for programs with complex structure in engineering. Some of the tests adopting MC/DC coverage did not reach 100%, resulting from the fact that MC/DC is a relatively strict criterion. But all the tests were basically backtrack-free with a high coverage, proving its effectiveness in engineering.

6.2.3. Comparison with a Static Method. This part presents the results from an empirical comparison of BFS-BB with the static method [13] (denoted as "method 1" to avoid verbose description), which was implemented in CTS prior to BFS-BB. The details of the test beds are shown in Table 9. The comparison adopted three coverage criteria: statement, branch, and MC/DC. And the result is shown in Table 10.

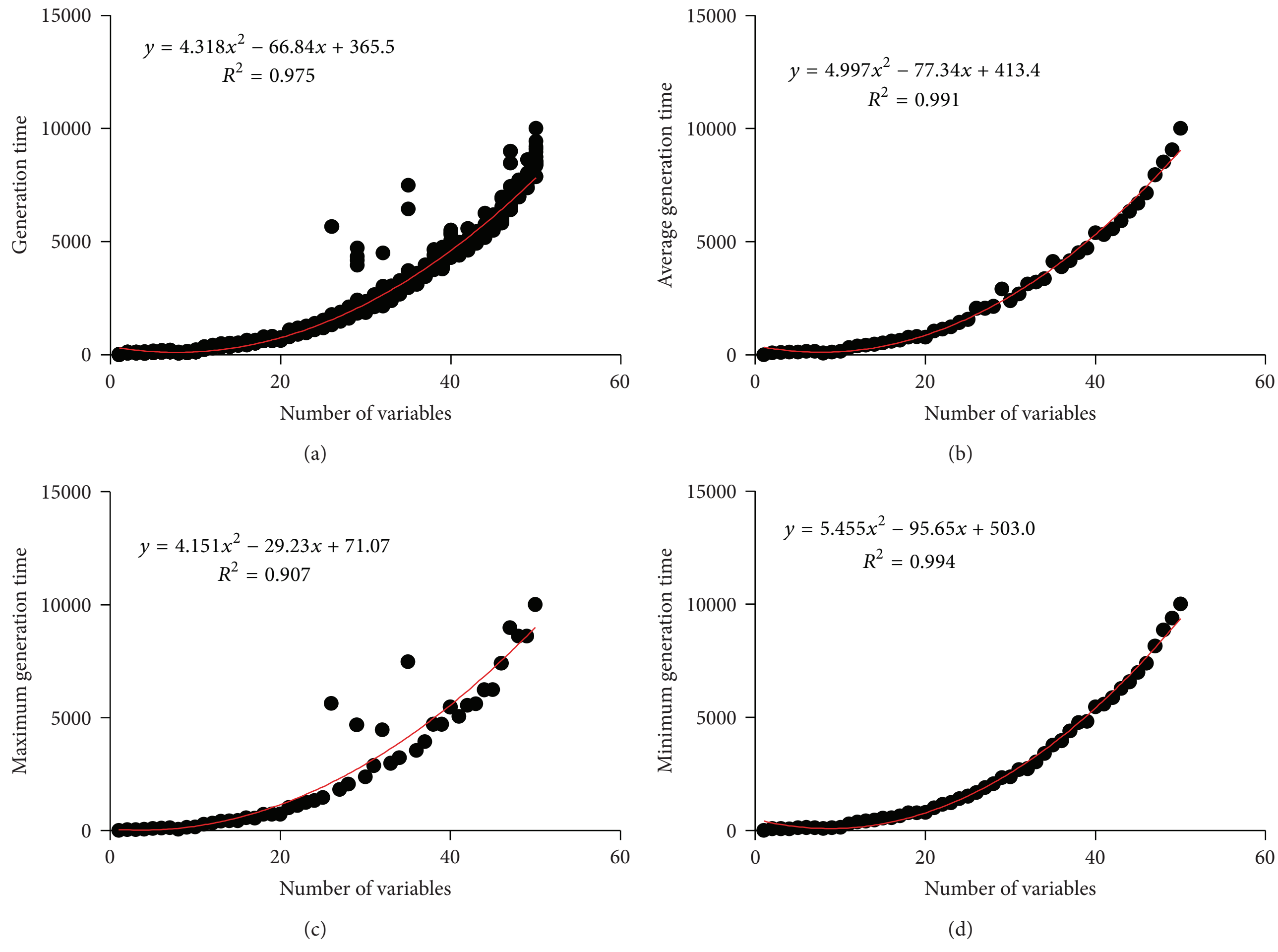


FIGURE 8: Relationship between generation time and the number of variables for variables that are linearly related in the tightest manner.

TABLE 8: Test result achieved by BFS-BB on programs from *de118i-2*.

Program	LOC	Number of variables	Number of functions	Number of loops	Statement coverage %	Branch coverage %	MC/DC coverage %
runge.c	1626	8	2	8	100	100	89
oblate.c	581	14	4	4	100	100	94
jplmp.c	996	6	5	5	100	100	88
floorl.c	432	7	5	2	100	100	92
atanl.c	267	3	2	0	100	100	100
adams4.c	767	28	10	12	100	100	85
tanl.c	256	4	3	0	100	100	100

TABLE 9: Programs used for comparison with method 1.

Program	LOC	Number of branches	Number of variables	Description	Source
Bonus	29	10	1	To calculate bonus according to profit	[13]
Days	33	17	3	To calculate the day number within a year	[13]
Statistics	21	8	5	To count the number of every kind of characters	[13]
gcd	38	5	2	To calculate greatest common denominator	[14]

Table 10: Comparison result with method 1 using three coverage criteria.

Program	Coverage criterion	Number of paths	Average coverage by method 1 %	Average coverage by BFS-BB %
Bonus	Statement	6	25	100
	Branch	6	37	100
	MC/DC	6	30	100
Days	Statement	17	100	100
	Branch	17	100	100
	MC/DC	14	94	100
Statistics	Statement	4	100	100
	Branch	5	100	100
	MC/DC	11	82	100
gcd	Statement	3	85	100
	Branch	3	77	100
	MC/DC	5	75	100

Table 11: Programs used for comparison with GA and SA.

Program	LOC	Number of branches	Number of variables	Description	Source
Triangle type	31	3	5	To classify type of a triangle	[15]
Valid date	59	16	3	To check whether a date is valid or not	[15]
Cal. day	72	11	3	To calculate the day of the week	[16]
Cal.	53	18	5	To calculate the number of days between the two given days in the same year	[17]

Since interval arithmetic has been improved in CTS, for the sake of validity, test data were generated for all the test beds on the same foundation of interval arithmetic. It can be seen that BFS-BB reached 100% coverage for all the test beds using three coverage criteria, while method 1 did not. That is largely due to the heuristic methods utilized in BFS-BB. There are modulus operations in the program *days*, which had been difficult for the interval arithmetic in method 1 to handle. But the functionality of interval arithmetic has been improved by adding a library of inverse functions, so it was not so difficult even for method 1.

6.2.4. Comparison with Dynamic Methods. This part presents results from an empirical comparison of BFS-BB with two dynamic methods which are genetic algorithm (GA) and simulated annealing (SA) on four different benchmark programs using branch coverage. The details of the benchmark programs are shown in Table 11. In order to obtain unbiased experimental results, we made a number of experiments with different parameter settings and chose the one that GA and SA performed the best as shown in Table 12. Test data were automatically generated for each program using GA, SA, and BFS-BB, with each tested 100 times. The average coverage was recorded to make the comparison, and the result was presented in Table 13.

It can be seen that BFS-BB reached 100% coverage on all the four benchmark programs, which are rather simple programs for BFS-BB, and it outperformed the algorithms in comparison. The better performance of BFS-BB is due to three factors. The first is that the initial values of variables are selected by heuristics on the path, so BFS-BB reaches a relatively high coverage for the first round of the search. The second is that MHS crashes 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. The third is that MPC is checked not only in the state space search stage but also in the initialization stage, which reduces the domains of the variables to ensure a relatively small search space that follows.

7. Conclusion

This paper presents an intelligent algorithm best-first-search branch and bound (BFS-BB) for path-wise test data generation (Q). The problem Q is reformulated as a constraint optimization problem (COP) and two techniques from artificial intelligence are introduced to tackle the COP, which are state space search and branch and bound (BB). The former is used to construct the search tree dynamically and the latter is used as the search method. Heuristics are adopted in the look-ahead stage. Dynamic variable ordering (DVO) is presented with a heuristic rule to break ties. Maintaining path consistency (MPC) is achieved through analysis on the result of interval arithmetic. The monotonicity analysis on branching conditions is applied both in the selection of initial values by path tendency calculation (PTC) and initial domain

Table 12: Parameter setting for GA and SA.

Item	Parameter	Value
Common issues	Population size	30
	Number of max generations	100
GA	Selection strategy	Roulette wheel
	Crossover probability	0.90
	Mutation probability	0.05
SA	Initial temperature	1.00
	Cooling coefficient	0.95

Table 13: Comparison result with GA and SA using branch coverage.

Program	Number of paths	Average coverage by GA %	Average coverage by SA %	Average coverage by BFS-BB %
Triangle type	6	95	99.88	100
Valid date	5	99.95	98.21	100
Cal. day	20	96.31	99.97	100
Cal.	7	99.02	99.27	100

calculation (IDC) and in the selection of other values by bisection when MPC encounters a conflict. An optimization method irrelevant variable removal (IVR) is also proposed to reduce the search space. Empirical experiments were conducted to evaluate the performance of BFS-BB. The results show that it searches in a basically backtrack-free manner, generates test data on programs of complex structure with promising performance, and outperforms some currently existing static and dynamic methods in terms of coverage. The application of BFS-BB in engineering proves its effectiveness. From the perspective of BFS-BB, the heuristics used in the look-ahead search are counterproductive to the look-back search.

Our future research will involve how to generate test data to reach high coverage with more types of constraints so as to give scalability to BFS-BB. We will also study how coverage criteria, generation approach, and system structure jointly influence test effectiveness. The effectiveness of the generation approach continues to be our primary work. Particularly the MC/DC coverage criterion will be given more emphases.

Conflict of Interests

The authors declare no competing financial interests.

Acknowledgments

This work was supported by the National Grand Fundamental Research 863 Program of China (no. 2012AA011201), the National Natural Science Foundation of China (no. 61202080), the Major Program of the National Natural Science Foundation of China (no. 91318301), and the Open Funding of State Key Laboratory of Computer Architecture (no. CARCH201201).

References

[1] M. R. Lyu, S. Rangarajan, and A. P. A. Van Moorsel, "Optimal allocation of test resources for software reliability growth modeling in software development," *IEEE Transactions on Reliability*, vol. 51, no. 2, pp. 183–192, 2002.

[2] Z. Xiaonan, Y. Junfeng, D. Siliang, and H. Shudong, "A new method on software reliability prediction," *Mathematical Problems in Engineering*, vol. 2013, Article ID 385372, 8 pages, 2013.

[3] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, NY, USA, 2nd edition, 1990.

[4] E. J. Weyuker, "Evaluation techniques for improving the quality of very large software systems in a cost-effective way," *Journal of Systems and Software*, vol. 47, no. 2, pp. 97–103, 1999.

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

[6] J. C. King, "Symbolic execution and program testing," *Communications of the Association for Computing Machinery*, vol. 19, no. 7, pp. 385–394, 1976.

[7] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, vol. 47, no. 6, pp. 504–515, New York, NY, USA, June 2011.

[8] T. Hickey, Q. Ju, and M. H. Van Emden, "Interval arithmetic: from principles to implementation," *Journal of the ACM*, vol. 48, no. 5, pp. 1038–1068, 2001.

[9] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*, Society for Industrial and Applied Mathematics, Philadelphia, Pa, USA, 2009.

[10] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[11] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 53–62, 1998.

[12] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, vol. 8, pp. 209–224.

[13] W. Yawen, G. Yunzhan, and X. Qing, "A method of test case generation based on necessary interval set," *Journal of Computer-Aided Design & Computer Graphics*, vol. 25, no. 4, pp. 550–556, 2013.

[14] A. Bouchachia, "An immune genetic algorithm for software test data generation," in *Proceedings of the 7th International Conference on Hybrid Intelligent Systems (HIS '07)*, pp. 84–89, September 2007.

[15] M. Chengying, Y. Xinxin, and C. Jifu, "Generating test case for structural testing based on ant colony optimization," in *Proceedings of the 12th International Conference on Quality Software (QSIC '12)*, pp. 98–101, 2012.

[16] E. Alba and F. Chicano, "Observations in using parallel and sequential evolutionary algorithms for automatic software testing," *Computers and Operations Research*, vol. 35, no. 10, pp. 3161–3183, 2008.

[17] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, New York, NY, USA, 2008.

[18] S. Ali, L. C. Briand, H. Hemmati, and R. 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, pp. 742–762, 2010.

[19] S. Chayanika, S. Sabharwal, and R. Sibal, "A survey on software testing techniques using genetic algorithm," *International Journal of Computer Science Issues*, vol. 10, no. 1, pp. 381–393, 2013.

[20] N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98)*, vol. 23, no. 2, pp. 73–81, 1998.

[21] A. Sakti, Y. G. Guéhéneuc, and G. Pesant, "CSBT: constrained search-based test data generation for software," in *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, p. 55, 2012.

[22] M. J. Gallagher and V. L. Narasimhan, "Adtest: a test data generation suite for ada software systems," *IEEE Transactions on Software Engineering*, vol. 23, no. 8, pp. 473–484, 1997.

[23] L.-L. Wang and W.-H. Tsai, "Optimal assignment of task modules with precedence for distributed processing by graph matching and state-space search," *BIT Numerical Mathematics*, vol. 28, no. 1, pp. 54–68, 1988.

[24] K. L. McMillan and D. K. Probst, "A technique of state space search based on unfolding," *Formal Methods in System Design*, vol. 6, no. 1, pp. 45–65, 1995.

[25] L. Gao, S. K. Mishra, and J. Shi, "An extension of branch-and-bound algorithm for solving sum-of-nonlinear-ratios problem," *Optimization Letters*, vol. 6, no. 2, pp. 221–230, 2012.

[26] E. I. Goldberg, L. P. Carloni, T. Villa, and R. K. Brayton, "Negative thinking in branch-and-bound: the case of unate covering," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 3, pp. 281–294, 2000.

[27] D. Frost and R. Dechter, "Look-ahead value ordering for constraint satisfaction problems," in *Proceedings of the International Joint Conference on Artificial Intelligence*, vol. 1, pp. 572–578, 1995.

[28] R. E. Moore, *Interval Arithmetic and Automatic Error analysis in digital computing [Ph.D. thesis]*, Stanford University, 1962.

[29] R. E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1966.

[30] R. E. Moore, *Methods and Applications of Interval Analysis*, vol. 2 of *Society for Industrial and Applied Mathematics*, Philadelphia, Pa, USA, 1979.

[31] P. McMinn, "Search-based software test data generation: a survey," *Software Testing Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[32] A. Petcu and B. Faltings, "A scalable method for multiagent constraint optimization," in *Proceedings of the International Joint Conference on Artificial Intelligence*, vol. 5, pp. 266–271, 2005.

[33] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo, "Adopt: asynchronous distributed constraint optimization with quality guarantees," *Artificial Intelligence*, vol. 161, no. 1-2, pp. 149–180, 2005.

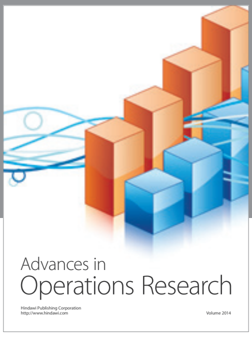
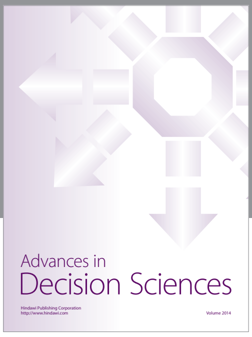
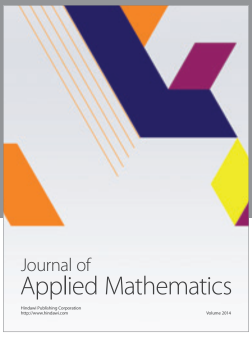
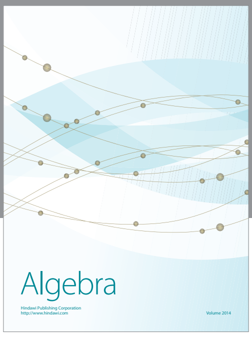
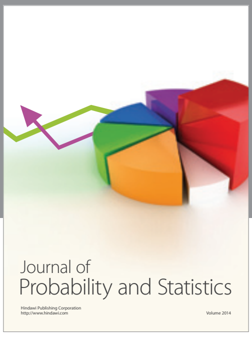
[34] D. Szer, F. Charpillet, and S. Zilberstein, "MAA$*$: a heuristic search algorithm for solving decentralized POMDPs," in *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI '05)*, pp. 576–583, July 2005.

[35] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. F. Werneck, "Exact Combinatorial Branch-and-Bound for Graph Bisection," in *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX '12)*, pp. 30–44, 2012.

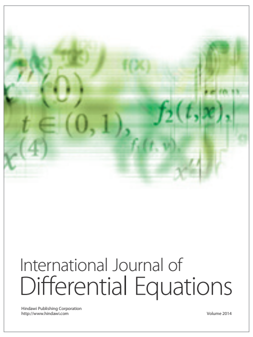
[36] X. Chen and P. van Beek, "Conflict-directed backjumping revisited," *Journal of Artificial Intelligence Research*, vol. 14, pp. 53–81, 2001.

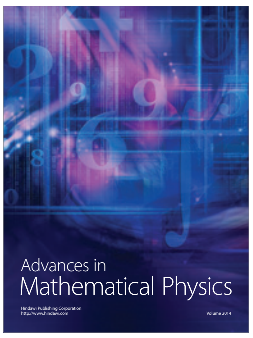
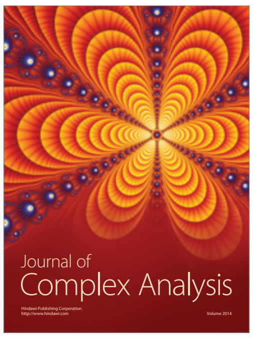
[37] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 787–805, 2013.

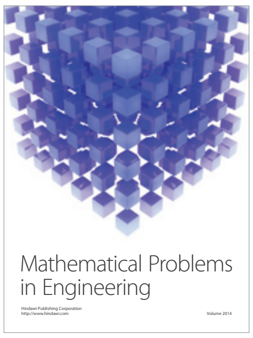
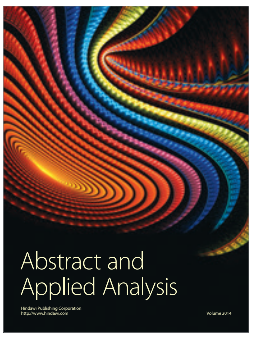
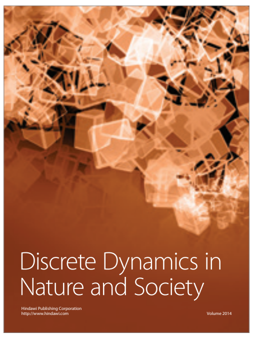
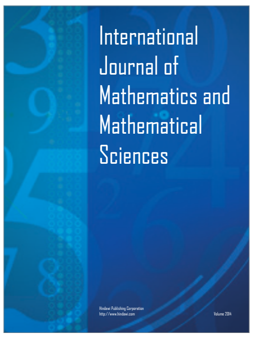
[38] A. Rajan, M. W. Whalen, and M. P. E. Heimdahl, "Distinguished paper: the effect of program and model structure on MC/DC test adequacy coverage," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 161–170, May 2008.


Advances in
Operations Research

Advances in
Decision Sciences

Journal of
Applied Mathematics

Algebra

Journal of
Probability and Statistics

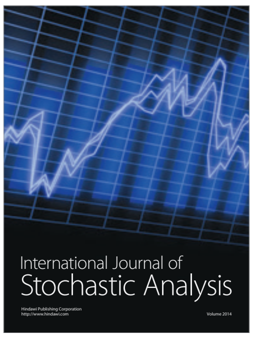
The Scientific
World Journal


International Journal of
Differential Equations

International Journal of
Combinatorics

Advances in
Mathematical Physics

Journal of
Complex Analysis

Journal of
Mathematics

Mathematical Problems
in Engineering

Abstract and
Applied Analysis

Discrete Dynamics in
Nature and Society

International
Journal of
Mathematics and
Mathematical
Sciences

Journal of
Discrete Mathematics

Journal of
Function Spaces

International Journal of
Stochastic Analysis

Journal of
Optimization