Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 617685, 15 pages
http://dx.doi.org/10.1155/2015/617685

Research Article

A Hybrid Intelligent Search Algorithm for Automatic Test Data Generation

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 24 March 2014; Revised 14 July 2014; Accepted 28 July 2014

Academic Editor: Jaiji Wu

The increasing complexity of large-scale real-world programs necessitates the automation of software testing. As a basic problem in software testing, the automation of path-wise test data generation is especially important, which is in essence a constraint optimization problem solved by search strategies. Therefore, the constraint processing efficiency of the selected search algorithm is a key factor. Aiming at the increase of search efficiency, a hybrid intelligent algorithm is proposed to efficiently search the solution space of potential test data by making full use of both global and local search methods. Branch and bound is adopted for global search, which gives definite results with relatively less cost. In the search procedure for each variable, hill climbing is adopted for local search, which is enhanced with the initial values selected heuristically based on the monotonicity analysis of branching conditions. They are highly integrated by an efficient ordering method and the backtracking operation. In order to facilitate the search methods, the solution space is represented as state space. Experimental results show that the proposed method outperformed some other methods used in test data generation. The heuristic initial value selection strategy improves the search efficiency greatly and makes the search basically backtrack-free. The results also demonstrate that the proposed method is applicable in engineering.

1. Introduction

With the surge of increasingly complex real-world software, software testing plays a more and more important 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]. In 2002, National Institute of Standards and Technology (NIST) estimated the cost of software failure to the US economy at $\$6 \times 10^{10}$, which was about 0.6% of GDP at the time [3]. The same report also found that over one-third of the cost of software failure could be eliminated by an improved testing infrastructure. But manual testing is time-consuming and error-prone and is even impracticable for large-scale real-world programs such as a Windows project with millions of lines of codes (LOC) [4]. So the automation of testing is of crucial concern [5]. Furthermore, as a basic problem in software testing, path-wise test data generation (denoted by Q) is of particular importance because path-wise testing can detect almost 65 percent of the faults in the program under test (PUT) [6] and many problems in software testing can be transformed into Q.

The methods of solving Q can be categorized as dynamic and static. The dynamic methods require the actual execution of the PUT and metaheuristic (MHS) [7] methods such as simulated annealing (SA) [8] and genetic algorithm (GA) [9] are very popular. They can generate test data with appropriate fault-prone ability [10, 11], but their slow convergence speed makes the process of generating test data quite long. Recently, particle swarm optimization (PSO) [12–14] has become a hot research topic due to its convenient implementation and faster convergence speed.

The static methods utilize techniques including symbolic execution [15, 16] and interval arithmetic [17, 18] 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 [19] proposed a fault-based technique that used algebraic constraints to describe test data designed to find particular types of faults. Gotlieb et al. [20] introduced "static single assignment" into a constraint system and solved the system. Cristian et al. from Stanford University proposed a symbolic execution tool named KLEE [21] 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. [22] proposed an interval analysis algorithm using forward dataflow analysis. But no matter what techniques are adopted the static methods require a strong constraint solver.

Aiming at constructing an efficient constraint processing engine, this paper proposes a new method for static test data generation based on the abstract memory model (AMM) [23] in Code Test System (CTS) (http://ctstesting.cn/), which tests real-world programs written in C programming language. AMM underlying automatic test data generation maintains a table of memory states and the constraints related to the structure of the data types can be represented by the table.

Following are the main contributions of this paper. The problem of path-wise test data generation (Q) is defined as a constraint optimization problem (COP), which is often solved by searching strategies. We introduce two algorithms in artificial intelligence to form a hybrid intelligent search method to solve Q. Branch and bound is used as the global search method, which gives definite results with relatively less cost. Hill climbing is utilized as the local search method when searching for a fixed value for a specified variable. Specifically, the initial value selection in the process of hill climbing is based on the heuristic analysis of the monotonicity of branching conditions. In order to facilitate the search methods, the solution space is represented as state space.

The rest of this paper is organized as follows. The background underlying our research is introduced in Section 2. The problem Q is reformulated as a COP and the solution is presented in Section 3. Section 4 illustrates the proposed hybrid search algorithm and an efficient variable ordering algorithm. Section 5 describes the local search algorithm hill climbing in detail. A case study is provided in Section 6 to thoroughly explain how the hybrid search algorithm works. Section 7 makes experimental analyses and empirical evaluations on the proposed algorithm and coverage comparison with some currently existing test data generation methods. Section 8 concludes this paper and highlights directions for future research.

2. Background

State space search [24, 25] 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) [26, 27] is an efficient backtracking algorithm for searching the solution space of a problem and 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).

Hill climbing (HC) [28, 29] is a comparatively simple local search algorithm that works to improve a single-candidate solution, starting from a randomly selected starting point. From the current position, the neighboring search space is investigated. If a better candidate solution is found, the search moves to that point which replaces the current solution. The neighborhood of the new solution is then investigated. If a better solution is found, the current solution is replaced again and the process continues, until no improved neighbors can be found for the current solution. The search conducted by hill climbing relies on the starting point very much. This progressional improvement is like climbing a hill in the "landscape" of an objective function. In this landscape, the peak signifies a solution with the locally optimal objective values.

Interval arithmetic is an important static testing technique, 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. A fixed value for a variable is represented as an interval with min equal to max, for example, $[5, 5]$. 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. Let two intervals be $X = [\underline{x}, \overline{x}]$ and $Y = [\underline{y}, \overline{y}]$; some basic rules used for explanation in this paper are listed below.

(i) $X + Y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$;

(ii) $X - Y = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$;

(iii) $X \cap Y = [\max(\underline{x}, \underline{y}), \min(\overline{x}, \overline{y})]$;

(iv) $X \cup Y = [\min(\underline{x}, \underline{y}), \max(\overline{x}, \overline{y})]$, if $X \cap Y \neq \varnothing$.

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. Many forms of static test data generation make reference to the control flow graph (CFG) of the PUT [30]. 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 $e = (nr, nt) \in E$ representing a transfer of control from node n_r to node n_t. Nodes corresponding to decision statements such as ifstatements 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 [31] 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 by $\{x_1 \mapsto V_1, x_2 \mapsto V_2, \ldots, x_n \mapsto V_n\}$, $V_i \in D_i$ to make path p feasible. To be specific, each constraint defined by the PUT along p should be satisfied. In static analysis, the feasibility of a path is judged by the result of interval arithmetic. In this paper, the paths in the examples that we use are all feasible for the convenience of explanation, but our work also involves the detection of infeasible paths in the process of generating test data.

An example with a program test and its corresponding CFG is shown in Figure 1, where *if_out_6*, *if_out_7*, *if_out_8*, *if_out_9*, and *exit_10* are dummy nodes. Adopting branch coverage, there are five paths to be traversed, namely, *Path1*: $0 \to 1 \to 9 \to 10$, *Path2*: $0 \to 1 \to 2 \to 8 \to 9 \to 10$, *Path3*: $0 \to 1 \to 2 \to 3 \to 7 \to 8 \to 9 \to 10$, *Path4*: $0 \to 1 \to 2 \to 3 \to 4 \to 6 \to 7 \to 8 \to 9 \to 10$, and *Path5*: $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$. The numbers along the paths denote nodes rather than edges of the CFG. Assuming *Path5* is the path to be traversed as shown in bold, see Figure 1, our work is to select $V = \{V_1, V_2\}$ from $\{D_1, D_2\}$ for $x1$ and $x2$, so that when executing test using $\{x1 \mapsto V_1, x2 \mapsto V_2\}$ as an input, the path traversed is *Path5*. There are four branching nodes *if_head_1*, *if_head_2*, *if_head_3*, and *if_head_4* along *Path5* and four corresponding branches *T_1*, *T_2*, *T_3*, and *T_4* that contain the constraints to be met.

3.2. Solution to the Problem. A COP is generally solved by search algorithms [32], which may be global or local. To date, there has been no theoretical analysis that characterizes the types of search methods (global or local) to be effective for path-wise test data generation as a COP. Global search aims to overcome the problem of local optimum in the search space and can thereby find more globally optimal solutions. Local search may become trapped in local optimum within the solution space, but can be far more efficient for simpler problems. In software testing, global search may achieve better coverage than local search, but at the cost of greater computational effort. However, Harman and McMinn [11] revealed that local search can be very effective and efficient, but there remain problems for which global search is the only technique that can successfully achieve coverage. The strong performance of local search, coupled with the necessity to retain global search for optimal effectiveness, naturally points to the consideration of hybrid search techniques. In view of that, we present a hybrid intelligent search method BB-HC, combining two search methods branch and bound and hill climbing. And we also try to heuristically find a better starting point to improve the search efficiency, rather than selecting a random value.

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). In addition, although the experiments were carried out on programs of different data types, integer variables are used as example in the following algorithms in order to simplify the explanations.

4. The Hybrid Search Strategies

This section proposes the framework of the hybrid search. Specifically, the representation of state space search is described in detail in Section 4.1, which is followed by the hybrid intelligent search algorithm in Section 4.2. And the dynamic ordering algorithm in the hybrid search algorithm is explained in Section 4.3.

4.1. The Representation of State Space Search. The state space is a quadruple (S, A, I, F), where S is a set of states, A is a set of arcs or connections between the states that correspond to the steps or operations of the search at different states, I is a nonempty subset of S denoting the initial state of the problem, and F is a nonempty subset of S denoting the final state of the problem.

A state is a quintuple (Precursor, Variable, Domain, Value, and Type). Precursor provides a link to the previous state; Variable $= x_i \in X$ $(i = 1, 2, \ldots, n)$ is the current variable; Domain $= D_i \in D$ in the form of [min, max] is the set of possible values that may be selected to instantiate Variable; Value $= V_i \in D_i$ is a value selected from Domain; Type marks the type of state which might be active, extensive, or inactive.

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 Variable is null the search ends. The path made up of all the extensive nodes in the search tree makes the solution path. The process of generating test data for path p takes the form of state space search. The state space needs to be searched 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.

```
void test(int x1, int x2)
1{  if(x1 > x2)
2       if(x1 >= 0)
3           if(x1 + x2 == 100)
4               if(x1 − x2 == 20)
5                   printf("Solved!");
  }
```

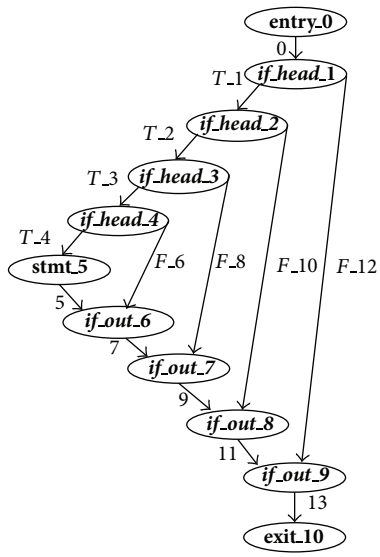

Figure 1: Program test and its corresponding CFG.

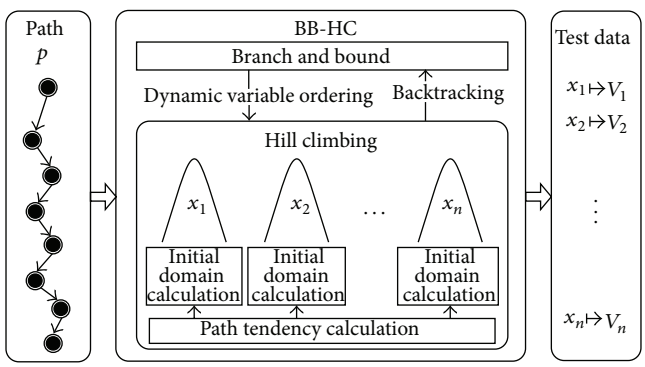

Figure 2: Overview of BB-HC for searching the test data.

4.2. The Hybrid Intelligent Search Algorithm. The idea of our algorithm is to extend partial solutions. At each stage, a variable in FV is selected and assigned a value from its domain to extend the current partial solution. Hill climbing evaluates whether such an extension may lead to a possible solution of the COP and prunes subtrees containing no solutions based on the current partial solution. Some relevant concepts in this paper are described in Table 1.

The overview of our approach can be seen from Figure 2. The path to be traversed is shown in the left part, where the 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. Then BB-HC works to generate the test data. All the variables in FV are permutated by DVO (see Section 4.3) to form a queue and its head x_1 is determined as the first variable to be instantiated. Next PTC (see Section 5.1) calculates path tendency for each variable and IDC (see Section 5.1) reduces the domain D_1 in which the initial value V_1 is selected for x_1. With all these, the initial state is constructed as (null, x_1, D_1, V_1, active), which is also the current state S_{cur}. Then the hill climbing (see Section 5.2) process begins for x_1. For brevity, our following explanation refers to the hill climbing process for each x_i in FV, and i is the sequential number of the current variables x_i. But it is only by the DVO following a successful HC that variable x_i can be determined (except x_1). Accordingly, D_i and V_i are the domain and value of the current variable x_i.

Hill climbing utilizes interval arithmetic to judge whether V_i for x_i leads to a conflict or not. If not, the peak of the hill is reached with 0 as the objective value, and the type of S_{cur} is changed into extensive, which means the hill climbing process ends for x_i, and DVO for the next variable will begin. But if a conflict is detected, we will calculate the objective function and reduce the domain of x_i (D_i) according to its return value. After selecting a new V_i from reduced D_i, interval arithmetic again will help to judge whether this V_i leads to a conflict. In summary, the hill climbing process ends with two possibilities. One is that it finally finds the optima for x_i and reaches the peak of the hill, so the type of S_{cur} is changed to extensive indicating that the local search for x_i ends and DVO for the next variable will begin. The other is that it fails to find the optima for x_i and there is no more search space, so the type of S_{cur} is changed to inactive indicating that the local search for x_i ends and backtracking is inevitable. BB-HC ends when the hill climbing processes for all the variables succeed and there is no more variables to be permutated. And the test data is the output of BB-HC as shown in the right part of Figure 2. The above mentioned search process is described by pseudocodes as shown in Algorithm 1.

4.3. Dynamic 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

Table 1: Some methods and their description used in this paper.

Name	Description
BB-HC	The proposed hybrid intelligent algorithm combining BB and HC
Dynamic variable ordering (DVO)	To permutate FV and return a variable to be instantiated
Path tendency calculation (PTC)	To calculate the path tendencies of all variables along the path, which will be used to calculate the domains in which their initial values are selected
Initial domain calculation (IDC)	To calculate the domain of a variable in which its initial value is selected according to its path tendency calculated by PTC
Hill-climbing	To judge whether a fixed value for a certain variable leads to a conflict or not by interval arithmetic and calculate the corresponding objective function

Input p: the path to be traversed
Output $result\{Variable \mapsto Value\}$: test data making p feasible
Begin
(1) $result \leftarrow null$;
(2) call Algorithm 2. Dynamic variable ordering;
(3) call Algorithm 3. Path tendency calculation;
(4) call Algorithm 4. Initial domain calculation;
(5) $V_i \leftarrow \text{select}(D_i)$;
(6) initial state $\leftarrow (null, x_i, D_i, V_i, active)$;
(7) $S_{\text{cur}} \leftarrow$ initial state;
(8) **while** $(x_i \neq null)$
(9) call Algorithm 5. Hill climbing;
(10) **if** $(S_{\text{cur}} = (Pre, x_i, D_i, V_i, inactive))$
(11) $Pre \leftarrow S_{\text{cur}}$;
(12) $S_{\text{cur}} \leftarrow (Pre, x_i, D_i, V_i, active)$;
(13) PV $\leftarrow$ PV $- \{x_i\}$;
(14) **else** $result \leftarrow result \cup \{x_i \mapsto V_i\}$;
(15) FV $\leftarrow$ FV $- \{x_i\}$;
(16) PV $\leftarrow$ PV $+ \{x_i\}$;
(17) call Algorithm 2. Dynamic variable ordering;
(18) call Algorithm 4. Initial domain calculation;
(19) $V_i \leftarrow \text{select}(D_i)$;
(20) $S_{\text{cur}} \leftarrow (Pre, x_i, D_i, V_i, active)$;
(21) final state $\leftarrow S_{\text{cur}}$;
(22) **return** $result$;
End

Algorithm 1: BB-HC.

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 1. Assuming that there are k branches along a path, the rank of a branch (n_{qa}, n_{qa+1}) $(a \in [1, k])$ marks its level in the sequence of the branches, denoted by $\text{rank}(n_{qa}, n_{qa+1})$.

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 while the latter has rank infinity. The comparison between a and infinity determines the ordering. The algorithm is described by pseudo-codes as shown in Algorithm 2.

Quicksort is utilized when variables are permutated according to remaining domain size and returns Q_i as the result. If no variables have the same domain size, then DVO returns the head of $Q_i(x_i)$. But if there are variables whose

Input FV: the set of future variables
D_i: the domain of $x_i (x_i \in \text{FV})$
$(n_{qa}, n_{qa+1})(a \in [1, k])$: k branches along the path
Output x_i: the selected variable to be next instantiated
Begin
(1) $Q_i \leftarrow$ quicksort(FV, $|D_i|$);
(2) **for** $i \rightarrow 1 : |Q_i|$
(3) **if** $(|D_i| \neq |D_j|)(j > i; x_i, x_j \in Q_i)$
(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) $x_i \leftarrow$ head (Q_i);
(11) **return** x_i;
End

Algorithm 2: Dynamic variable ordering.

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. Hill Climbing

Hill climbing is the focus of this section, and it is used to judge whether a fixed value V_i for the current variable x_i makes path p feasible. In other words, a certain V_i that makes p feasible is the peak that we are trying to search for x_i.

5.1. 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 characterized by blind search, which causes too many iterations. 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 (see Definition 3). First we give the definition of branching condition.

Definition 2. Let B be the set of Boolean values $\{true, false\}$, D_i be the domain of the variable in question (x_i), the branching condition $\text{Br}(n_{qa}, n_{qa+1})(x_i) : D_i \rightarrow B$ where n_{qa} is a branching node is defined as the following formula:

$$\text{Br}\left(n_{qa}, n_{qa+1}\right)\left(x_i\right) : D_i \longrightarrow B = \left(a_i x_i + \sum_{j \neq i} a_j x_j \right) \text{rel}\, c, \quad (1)$$

where rel is a relational operator, a_i, a_j, and c are constants, and $\sum_{j \neq i} a_j x_j$ 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 decomposed into its basic functions, then the monotonicity of the branching condition as a function can be known, and in turn the direction in which the input needs to be moved to make the function true can be determined. Following is the definition that is used for the initial value selection of variables.

Definition 3. 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 by w_i (n_{qa}, n_{qa+1}) and pw_i, which are calculated by formula (2) and formula (3), respectively. Consider

$$w_i\left(n_{qa}, n_{qa+1}\right) = \begin{cases} \dfrac{|a_i|}{|a_i| + \sum_{j\neq i}|a_j|}, & \text{if } \mathrm{Br}\left(x_i\right)\left(n_{qa}, n_{qa+1}\right) \\ & \text{is monotonically increasing,} \\ -\dfrac{|a_i|}{|a_i| + \sum_{j\neq i}|a_j|}, & \text{if } \mathrm{Br}\left(x_i\right)\left(n_{qa}, n_{qa+1}\right) \\ & \text{is monotonically decreasing,} \end{cases} \tag{2}$$

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

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 takes diversity and heuristics into account. The algorithms are expressed by pseudo-codes as shown in Algorithms 3 and 4.

5.2. The Hill-Climbing Process. This part focuses on the process where interval arithmetic judges whether the value assigned to a variable leads to a conflict or not. The calculating process is illustrated in Figure 3. Interval arithmetic first receives V_i, the value of the current variable x_i, which is part of the domain of all variables before evaluating the first branching condition (denoted by D^1) $(V_i = [V_i, V_i] \in D^1)$. For the k branching nodes along the path, all the k branching conditions should be true to make the path feasible if p is traversed with D_i. The value of the branching condition $\mathrm{Br}(n_{qa}, n_{qa+1})(x_i)$ $(a \in [1,k])$ depends on two factors: (1) D^a, which is the domain of all variables that satisfies all the $a-1$ branching conditions ahead and will be used as input for the calculation of the ath branching condition; (2) $\widetilde{D}^a$, which is the result when calculating $\mathrm{Br}(n_{qa}, n_{qa+1})(x_i)$ with D^a which satisfies the ath branching condition. To be specific, according to Definition 1, since $\mathrm{Br}(n_{qa}, n_{qa+1})(x_i)$ is in essence a relational expression, we can calculate the domain of each variable based on the specific form of the expression. $D^a \cap \widetilde{D}^a \neq \varnothing$ means that $D^a \cap \widetilde{D}^a$ satisfies all the $a-1$ branching conditions ahead and the ath branching condition, ensuring that interval arithmetic can continue to calculate the remaining branching conditions.

In this process, if $\mathrm{Br}(n_{qh}, n_{qh+1})(x_i) = \text{false}$ $(1 \le h \le k)$, which means a conflict is detected, then interval arithmetic is aborted, the reduction of D_i is carried out according to the result of the corresponding objective function, a new V_i is selected from the reduced domain D_i, and interval arithmetic will restart to judge whether V_i causes a conflict. The above procedure is like climbing a hill. Formula (4) is defined to calculate the objective function $F(V_i)$, where $\sum_{a=1}^{k}(D^a \cap \widetilde{D}^a)(x_i)$ is the value that is calculated according to values of D^a and $\widetilde{D}^a$ at each branch and is a definite value. Consider

$$F\left(V_i\right) = V_i - \sum_{a=1}^{k}\left(D^a \cap \widetilde{D}^a\right)\left(x_i\right). \tag{4}$$

$F(V_i) = 0$ implies that there is no conflict detected and V_i is the value judged to be appropriate for x_i. Otherwise D_i will have to be reduced according to the return value of $F(V_i)$. In the procedure of hill-climbing, the absolute value of $F(V_i)$ will approximate more closely to 0, which is the objective or the peak of the hill. The algorithm is shown by pseudo-codes as shown in Algorithm 5.

When interval arithmetic fails, $F(V_i)$ provides both the upper and the lower bounds of D_i for its reduction, determined by the sign and absolute value of $F(V_i)$, respectively. Since the reduction is taken in two directions, the efficiency of the algorithm is improved greatly.

6. Case Study

In this section, the problem mentioned in Section 3.1 is used as an example to explain how BB-HC works. The path to be traversed is *Path5* as shown in bold in Figure 1. We choose this example, because the constraints along the path are very strict for two variables. It is very obvious that $\{x1 \mapsto 60, x2 \mapsto 40\}$ is the only one solution to the corresponding COP. For simplicity, the input domains of both variables are set [1, 100] with the size 100. The path tendency of each variable is calculated by PTC as shown in Table 2. DVO serves to determine the first variable to be instantiated as shown in Table 3, which is $x1$ highlighted in bold. On determining $x1$ to be the current variable, an initial value needs to be selected from [1, 100]. The retrieval of path tendency map by IDC returns positive for $x1$, indicating that a larger value will perform better and 70 is selected.

The calculating process of 70 for $x1$ is decomposed in Figure 4, and 70 is judged not to be a solution for $x1$ or not the peak of the hill corresponding to $x1$, so it is required to calculate the objective function to determine the next search step.

Since it is the fourth branch where the conflict is detected, by $F(70) = 70 - \sum_{h=1}^{4}(D^h \cap \widetilde{D}^h)(x1) = 20 > 0$ we can reduce the domain of $x1$ to [70 − 20 (50), 70 − 1 (69)], which is much smaller than [1, 100]. So 70 for $x1$ is not a solution, but it provides information for the next search step. Supposing that 55 is selected from [50, 69] for $x1$, then there is the calculating process as shown in Figure 5.

Again a conflict is detected on the fourth branch. By $F(55) = 55 - \sum_{h=1}^{4}(D^h \cap \widetilde{D}^h)(x1) = -10 < 0$ we can reduce the domain of $x1$ to [55 + 1 (56), 55 + 10 (65)], which is much smaller than [50, 69]. So 55 for $x1$ is not a solution, but it provides information for the next search step. In the same manner, we can select a value (e.g., 60) from [56, 65] and calculate it as shown in Figure 6.

$F(60) = 60 - \sum_{a=1}^{4}(D^a \cap \widetilde{D}^a)(x1) = 0$ means that 60 is the peak of the hill that corresponds to $x1$, which starts from 70. The hill-climbing calculation process is shown in

Input FV: the set of future variables
pw_i: the path weight of variable x_i ($x_i \in$ FV; $i = 1, 2, \ldots, n$)
Output *Path-Tendency*{*Variable* $\mapsto$ *Path Tendency*}: a map used to store the path tendency of each variable
Begin
(1) *Path-Tendency* $\leftarrow$ *null*;
(2) **foreach** $x_i \in FV$
(3) **if** ($pw_i > 0$)
(4) *Path-Tendency* $\leftarrow$ *Path-Tendency* $\cup \{x_i \mapsto positive\}$;
(5) **else if** ($pw_i < 0$)
(6) *Path-Tendency* $\leftarrow$ *Path-Tendency* $\cup \{x_i \mapsto negative\}$;
(7) **return** *Path-Tendency*;
End

Algorithm 3: Path tendency calculation.

Input $D_i = [min, max]$: the domain of x_i before selecting its initial value
Path-Tendency{*Variable* $\mapsto$ *Path Tendency*}: a map used to store the path tendency of each variable
Output D_i: the domain of x_i in which its initial value is selected
Begin
(1) *Path Tendency*(x_i) $\leftarrow$ retrieval of *Path-Tendency*;
(2) **if** (*Path Tendency*(x_i) = *positive*)
(3) $D_i \leftarrow [(min + max)/2, max]$;
(4) **else if** (*Path Tendency*(x_i)) = *negative*)
(5) $D_i \leftarrow [min, (min + max)/2]$;
(6) **return** D_i;
End

Algorithm 4: Initial domain calculation.

Input D_i: the current domain of x_i
V_i: the current value of x_i
Output S_{cur}: (*Pre*, x_i, D_i, V_i, *inactive*) when hill climbing fails
S_{cur}: (*Pre*, x_i, D_i, V_i, *extensive*) when hill climbing succeeds
Begin
(1) $F(V_i) \leftarrow 0$;
(2) **while** ($|D_i| > 1$)
(3) **for** $a \rightarrow 1 : k$
(4) $\text{Br}(n_{qa}, n_{qa+1})(x_i) \leftarrow$ false;
(5) $\widetilde{D}^a \leftarrow$ calculate $\text{Br}(n_{qa}, n_{qa+1})(x_i)$ with D^a;
(6) **if** ($D^a \cap \widetilde{D}^a \neq \varnothing$)
(7) $\text{Br}(n_{qa}, n_{qa+1})(x_i) \leftarrow true$;
(8) $D^{a+1} \leftarrow D^a \cap \widetilde{D}^a$;
(9) **else** $F(V_i) \leftarrow V_i - \sum_{h=1}^{a} (D^h \cap \widetilde{D}^h)(x_i)$;
(10) **break**;
(11) **if** ($F(V_i) = 0$)
(12) $S_{\text{cur}} \leftarrow$ (*Pre*, x_i, D_i, V_i, *extensive*);
(13) **return** S_{cur};
(14) **else if** ($F(V_i) < 0$)
(15) $D_i \leftarrow [V_i + 1, V_i + |F(V_i)|]$;
(16) **else** $D_i \leftarrow [V_i - |F(V_i)|, V_i - 1]$;
(17) $V_i \leftarrow$ select(D_i);
(18) $S_{\text{cur}} \leftarrow$ (*Pre*, x_i, D_i, V_i, *inactive*);
(19) **return** S_{cur};
End

Algorithm 5: Hill climbing.

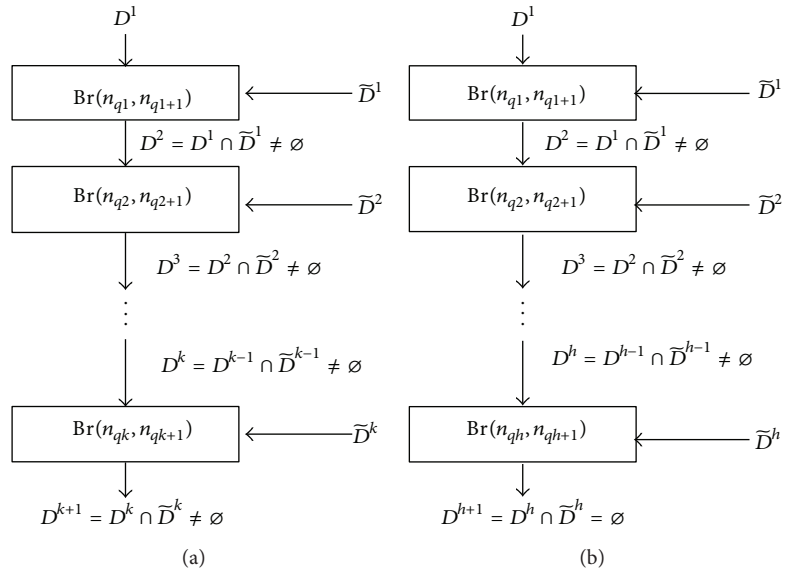

Figure 3: The conflict detecting process by interval arithmetic.

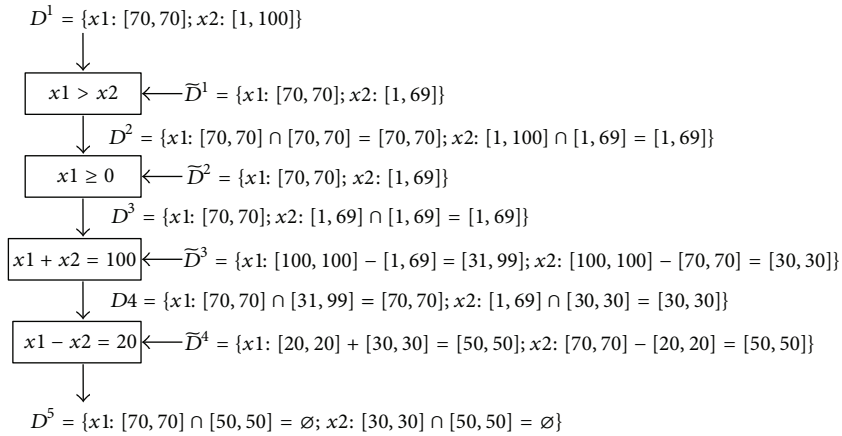

Figure 4: The calculating process of 70 for $x1$.

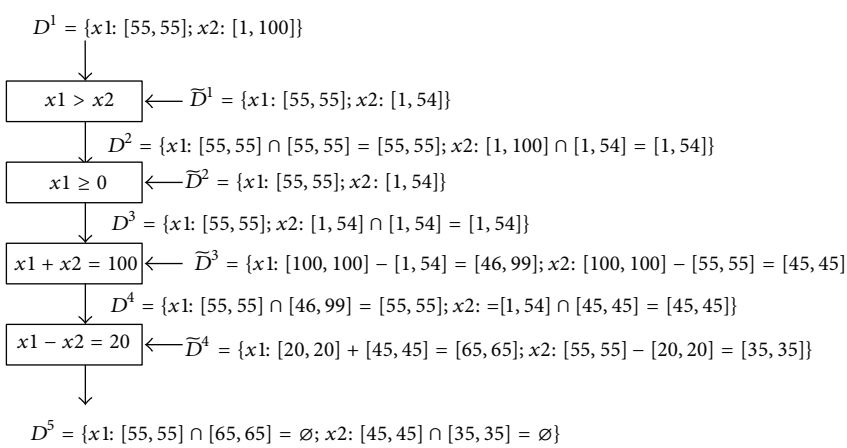

Figure 5: The calculating process of 55 for $x1$.

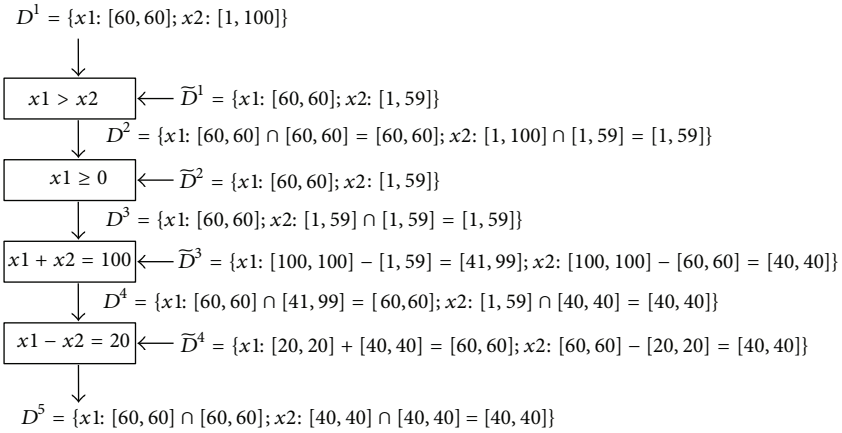

Figure 6: The calculating process of 60 for $x1$.

Table 2: PTC process for $x1$ and $x2$.

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 $b1 = x1 - x2$ $f(b1 > 0)$: increasing	Br($x1$): increasing Br($x2$): decreasing	$w1 = 0.5$ $w2 = -0.5$	$pw1 = 1.5$ $pw2 = -0.5$	$\{x1 \mapsto$ positive, $x2 \mapsto$ negative$\}$
$x1 \geq 0$	$f(x1) = x1$: increasing $f(x1 \geq 0)$: increasing	Br($x1$): increasing	$w1 = 1$		
$x1 + x2 = 100$		—			
$x1 - x2 = 20$		—			

Table 3: DVO process for $x1$ and $x2$.

Ordering rule	Condition for each variable	Tie encountered?	Ordering result
Domain size	$\|D1\| = 100$, $\|D2\| = 100$	Yes (both have the same domain size)	
Rank1	Rank1($x1$) = 1, Rank1($x2$) = 1	Yes ($x1$ and $x2$ both have Rank1)	**x1** $\rightarrow$ $x2$
Rank2	Rank2($x1$) = 2, Rank2($x2$) = ∞	No ($x1$ has Rank 2 while $x2$ has *infinity*)	

Table 4: The hill-climbing process for $x1$.

V_1	$F(V_1)$	$\|F(V_1)\|$	Peak reached?
70	20	20	No
55	–10	10	No
60	0	0	Yes

Table 4. And since there is only one value 40 for $x2$, the search succeeds with $\{x1 \mapsto 60, x2 \mapsto 40\}$.

7. Experimental Analyses and Empirical Evaluations

To observe the effectiveness of BB-HC, 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 BB-HC as input. The experiments were performed in the environment of Ubuntu 12.04 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 Java Runtime Environment (JRE). The experiments include two parts. Section 7.1 presents the performance evaluation on BB-HC, and Section 7.2 tests the capability of BB-HC to generate test data in terms of coverage and makes comparisons with some currently existing static and dynamic methods.

7.1. Performance Evaluation. The number of variables and the number of expressions (path constraints) [33, 34] are two important factors that affect the performance of test data generation methods. Hence, in this part, experiments were carried out to evaluate the effectiveness of the initial value selection strategy and the hill-climbing process for varying numbers of input variables and varying numbers of expressions, and we also paid attention to the number of backtracking. Specifically, three methods were used: random initial value and no hill climbing (RI&NHC), random initial value and hill climbing (RI&HC), and heuristic initial value

and hill climbing which is BB-HC. Due to the variety in generation time for different cases, the axes of generation time of both cases are normalized for simplicity.

7.1.1. Varying Number of Variables. The testing of the relationship between the performance of test data generation methods and the number of variables was accomplished by repeatedly running the three methods 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 50 if statements (equivalent to 50 branching conditions or 50 expressions along the path) and there was only one path to be traversed of fixed length, which was the one consisting of entirely true branches; that is, all the branching conditions were the same as the corresponding predicates. The expression of each if statement was a linear combination of all the n variables in the form of

$$[a_1, a_2, \ldots, a_n]\,[x_1, x_2, \ldots, x_n]'\ rel\_op\ \text{const}\,[c]\,, \quad (5)$$

where $a_1, a_2, \ldots, a_n$ were randomly generated numbers, either positive or negative, $rel\_op \in \{>, \geq, <, \leq, =, \neq\}$, and const$[c]$ ($c \in [1, 50]$) was an array of randomly generated constants within [0, 1000]. The randomly generated a_i and const$[c]$ should be selected to make the path feasible. This arrangement constructed the tightest linear relation between the variables. In addition, we ensured that there was at least one "=" in each program to test the equation solving capability of the methods. The programs for various values of n ranging from 1 to 50 were each tested 50 times and the average time required to generate the data for each test was recorded. The results are presented in Figure 7.

It can be seen that the average generation time of BB-HC is far less than RI&NHC and RI&HC. RI&NHC takes the longest time. The points corresponding to RI&NHC and RI&HC are not very regular, so we did not try to make fitting curves for them. For BB-HC, it is clear that the relation between average generation time and the number of variables can be represented as a quadratic curve very well and the quadratic correlation relationship is significant at 95% confidence level with P value far less than 0.05. Besides, average generation time increases at a uniformly accelerative speed as the increase of the number of variables. The differentiation of average generation time indicates that its increase rate rises by $y = 1.06x - 8.682$ 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. And according to our statistics, the numbers of backtracking conducted by BB-HC were all 0 for all the 50 cases while those of the others were not, so this search was completely backtrack-free.

7.1.2. Varying Number of Expressions. The testing of the relationship between the performance of test data generation methods and the number of expressions was accomplished by repeatedly running the three methods on generated test programs, each of which had 50 input variables. Adopting statement coverage, in each test, the program contained u ($u \in [1, 50]$) if statements (equivalent to u branching

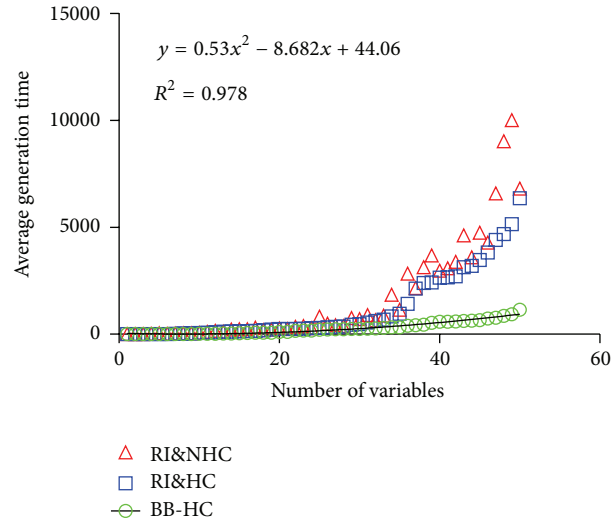

Figure 7: The relationship between generation time and the number of variables.

conditions or u expressions) and there was only one path with entirely true branches to be traversed; that is, all the branching conditions were the same as the corresponding predicates. The expression of each if statement was an expression in the form of

$$[a_1, a_2, \ldots, a_{50}]\,[x_1, x_2, \ldots, x_{50}]'\ rel\_op\ \text{const}\,[u]\,, \quad (6)$$

where $a_1, a_2, \ldots, a_{50}$ were randomly generated numbers either positive or negative, $rel\_op \in \{>, \geq, <, \leq, =, \neq\}$, and const$[u]$ was an array of randomly generated constants within [0, 1000]. The randomly generated a_v ($v = 1, 2, \ldots, 50$) and const$[u]$ should be selected to make the path feasible. This arrangement constructed the strongest linear relation between variables. In addition, we ensured that there was at least one "=" in each program to test the equation solving capability of the methods. The programs for various values of u ranging from 1 to 50 were each tested 50 times and the average time required to generate the data for each test was recorded. The results are presented in Figure 8.

It can be seen that the average generation time of BB-HC is far less than RI&NHC and RI&HC. RI&NHC takes the longest time. The points corresponding to RI&NHC and RI&HC are not very regular, so we did not try to make fitting curves for them. For BB-HC, it is clear that the average generation time increases approximately linearly with the number of expressions and the linear correlation relationship is significant at 95% confidence level with P value far less than 0.05. As the increase of the number of expressions, average generation time increases at an even speed. And according to our statistics, the numbers of backtracking conducted by BB-HC were all 0 for all the 50 cases while those of the others were not, so this search was completely backtrack-free.

The above searches conducted by BB-HC were both completely backtrack-free, which is encouraging. Surely there are nonlinear constraints, which will sometimes cause backtracking. But according to statistical data [35, 36], nonlinear

Table 5: The details of comparison with method 1.

Project	Program	Function	Function calls	Statement		Branch		MC/DC	
				AC by method 1	AC by BB-HC	AC by method 1	AC by BB-HC	AC by method 1	AC by BB-HC
Dell8i-2	atanl.c	atan2l()	1	75%	**100%**	72%	**100%**	71%	**100%**
	sinl.c	sinl()	4	92%	92%	92%	**97%**	83%	**87%**
		cosl()	4	87%	87%	91%	**97%**	82%	**83%**
Masscan	ranges.c	rangelist_pick2_destroy()	1	50%	**100%**	60%	**80%**	—	**71%**
	string_s.c	memcasecmp()	2	50%	**93.2%**	50%	**93.2%**	—	**94.9%**

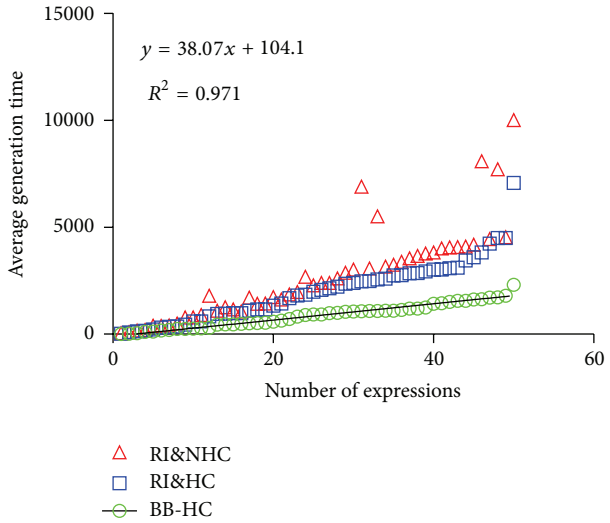

Figure 8: The relationship between average generation time and the number of expressions.

constraints in real-world programs only account for a very small proportion of program constraints, so BB-HC will be useful for most of the cases. It is safe to conclude that BB-HC functions are stably given a PUT of regular structure, which lays a solid foundation for its application in engineering.

7.2. Coverage Evaluation. To evaluate the capability of BB-HC to generate test data in terms of coverage, we used some real-world programs to compare BB-HC with both static and dynamic methods adopted in test data generation.

7.2.1. Comparison with a Static Method. This part presents the results from an empirical comparison of BB-HC with the static method [22] (denoted by "method 1" to avoid verbose description), which was implemented in CTS prior to BB-HC. Three of the test beds were from an engineering project *del18i-2* at http://www.moshier.net/ with numeric data types and two were from a TCP port scanner Masscan at https://github.com/robertdavidgraham/masscan with complex data types. The comparison adopted three coverage criteria: statement, branch, and MC/DC. For each test bed, the experiments were carried out 100 times and the average coverage (AC) was used for comparison, that is, the average of achieved coverage of all tests in 100 times. The details of the comparison are shown in Table 5.

From Table 5, it can be seen that BB-HC reached higher coverage than method 1 for most of the cases as shown in bold. That is largely due to the heuristic methods utilized in BB-HC. Method 1 was unable to handle complex data types with MC/DC as the coverage criterion. But it also shows that there are some programs where BB-HC could not achieve 100% coverage. By examining those programs, we found that there are some logical operators that make the static analysis even harder, thus resulting in more difficulty in generating test data for the paths containing them. To deal with such programs will be part of our next work.

7.2.2. Comparison with PSO. This part presents results from an empirical comparison of BB-HC with PSO, which is mentioned in Section 1 as a popular MHS method with relatively fast convergence speed. Following is a brief introduction to some parameters used in PSO.

Suppose the population size is s in the D-dimensional search space, a particle represents a potential solution. The velocity V_i^d and position X_i^d of the dth dimension of the ith particle can be updated by the following formulae:

$$V_i^d(t) = wV_i^d(t-1) + c_1 \cdot r_1 \cdot \left(p\text{best}_i^d - X_i^d(t-1)\right) + c_2 \cdot r_2 \cdot \left(g\text{best}^d - X_i^d(t-1)\right), \quad (7)$$
$$X_i^d(t) = X_i^d(t-1) + V_i^d(t),$$

where $X_i = (X_i^1, X_i^2, \ldots, X_i^D)$ is the position of the ith particle, $V_i = (V_i^1, V_i^2, \ldots, V_i^D)$ is the velocity of particle i, $p\text{best}_i^d$ is the personal best position found by the particle assigned for dimension d, and $g\text{best}^d$ is the global best position of dimension d. The parameters c_1 and c_2 are the acceleration constants reflecting the weight of stochastic acceleration terms that pull each particle towards $p$best and $g$best, respectively. r_1 and r_2 are random numbers in the range [0, 1]. And a particle's velocity on each dimension is clamped to a maximum $V_{\max}$. Inertia weight w is used to balance the global and local search abilities and it controls the impact of history on the new velocity. The parameter setting in Table 6 is typical and customary for PSO and it is used for our comparison.

We used three real-world programs, which are the well-known benchmark programs and have been widely adopted

Table 6: Parameter setting for PSO.

Parameter	Value
Population size	30
Max generations	100
Inertia weight w	Ranging from 0.2 to 1
Acceleration constants c_1 and c_2	$c_1 = c_2 = 2$
Maximum velocity V_{max}	Set according to the input space of the tested program, such as $V_{max} = 24$ for the program *triangleType*

Table 7: The details of comparison with PSO.

Program	LOC	Branches	Variables	Paths	Description	Source	AC by PSO	AC by BB-HC
triangleType	31	3	5	6	To classify type for a triangle	Referring to [37]	99.88%	**100%**
cal	53	18	5	7	To calculate the number of days between the two given days in the same year	Referring to [38]	96.85%	**100%**
calDay	72	11	3	20	To calculate the day of the week	Referring to [39]	97.35%	**100%**

by other researchers. And branch coverage was taken as the adequacy criterion. For each test bed, the experiments were carried out 100 times. The coverage achieved by the two methods might be different each time and AC was used for comparison. Table 7 shows the details of the test beds and the comparison results.

Obviously BB-HC achieved 100% coverage as shown in bold on all the three benchmark programs, which are rather simple programs for BB-HC and it outperformed the algorithm in comparison. Two factors contribute to the better performance of BB-HC. One is that the initial values of variables are selected by heuristics on the path, so BB-HC reaches a relatively high coverage for the first round of the search. The other is that BB-HC coordinates BB and HC flexibly to make sure that solution can be found for each variable efficiently.

8. Conclusion

The increasing demand of testing large-scale real-world programs makes the automation of the testing process necessary. In this paper, the problem of path-wise test data generation (Q) which is a basic problem in software testing is reformulated as a constraint optimization problem (COP), and a hybrid intelligent algorithm BB-HC is presented to solve it, hybridizing two search methods: branch and bound (BB) and hill climbing (HC). BB is used as the global search method and HC is dedicated to local search. They are highly integrated by dynamic variable ordering (DVO) and the backtracking operation. With a heuristic rule to break ties, DVO permutates variables to be instantiated. The monotonicity analysis of branching conditions is applied in the selection of the initial values by path tendency calculation (PTC) and initial domain calculation (IDC). Starting from the heuristically selected initial value, the process of determining a fixed value for a specified variable resembles climbing a hill, the peak of which is the value judged by interval arithmetic that does not cause conflict. To facilitate the search procedure, the solution space is represented as state space. Empirical experiments were conducted to evaluate the performance of BB-HC. The results show that it searches in a basically backtrack-free manner for linear constraints, generates test data on programs of complex structure and strong constraints with promising performance, and outperforms some current static and dynamic methods in terms of coverage. The application of BB-HC in engineering proves its effectiveness.

Our future research will involve how to generate test data to reach high coverage. The programs where BB-HC did not achieve 100% coverage, especially, will be put more emphases on. 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.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

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] J. Yue, C. Bojan, M. Tim, and L. Jie, "Incremental development of fault prediction models," *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 10, pp. 1399–1425, 2013.

[3] T. Gregory, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, RTI Project 7007.011, 2002.

[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] A. Bertolino, "Software testing research: achievements, challenges, dreams," in *Proceedings of the Future of Software Engineering (FoSE '07)*, pp. 85–103, Minneapolis, Minn, USA, May 2007.

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

[7] 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.

[8] T. Nigel, 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*, vol. 23, pp. 73–81, 1998.

[9] C. Sharma, 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.

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

[11] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2010.

[12] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceeding of the IEEE International Conference on Neural Networks (ICNN '95)*, vol. 4, pp. 1942–1948, Perth, Wash, USA, December 1995.

[13] Y. Shi and R. Eberhart, "Modified particle swarm optimizer," in *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC '98)*, pp. 69–73, May 1998.

[14] C. Mao, B. X. Yu, and J. Chen, "Swarm intelligence-based test data generation for structural testing," in *Proceedings of the IEEE/ACIS 11th International Conference on Computer and Information Science (ICIS '12)*, pp. 623–628, June 2012.

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

[16] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 504–515, 2012.

[17] 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.

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

[19] 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.

[20] 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*, vol. 23, pp. 53–62, 1998.

[21] C. Cristian, D. Dunbar, and D. 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, 2008.

[22] W. Yawen, G. Yunzhan, and X. Qing, "A method of test case generation based on necessary interval set," *Journal of Computer-Aided Design and Computer Graphics*, vol. 25, no. 4, pp. 550–563, 2013.

[23] T. Rong, W. Yawen, and G. Yunzhan, "Research on abstract memory model for test case generation," in *Proceedings of the 7th China Test Conference (CTC '12)*, pp. 144–149, 2012.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] K. Mahdavi, M. Harman, and R. M. Hierons, "A multiple hill climbing approach to software module clustering," in *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, pp. 315–324, September 2003.

[29] M. Harman and P. McMinn, "A theoretical & empirical znalysis of evolutionary testing and hill climbing for structural test data generation," in *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pp. 73–83, July 2007.

[30] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*, CRC Press, New York, NY, USA, 2013.

[31] A. E. Eiben and Z. Ruttkay, *Constraint Satisfaction Problems*, IOP Publishing Ltd and Oxford University Press, New York, NY, USA, 1997.

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

[33] 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.

[34] R. Zhao, M. Harman, and Z. Li, "Empirical study on the efficiency of search based test generation for EFSM models," in *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW 2010)*, pp. 222–231, April 2010.

[35] Y. W. Wang, Y. Z. Gong, Q. Xiao, and Z. H. Yang, "A method of variable range analysis based on abstract interpretation and its applications," *Acta Electronica Sinica*, vol. 39, no. 2, pp. 296–303, 2011.

[36] N. Gupta, P. Aditya, and M. L. Soffa, "Automated test data generation using an iterative relaxation method," *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 6, pp. 231–244, 1998.

[37] C. Mao, X. Yu, and J. Chen, "Generating test data for structural testing based on ant colony optimization," in *Proceedings of the 12th International Conference on Quality Software (QSIC '12)*, pp. 98–101, August 2012.

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

[39] 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.