INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
ISSN 1841-9836, 9(4):497-509, August, 2014.

Optimized Branch and Bound for Path-wise Test Data Generation

Y.W. Wang, Y. Xing, Y.Z. Gong, X.Z. Zhang

Ya-Wen Wang (1,2), **Ying Xing*** (1,3), **Yun-Zhan Gong** (1), **Xu-Zhou Zhang** (1)
(1) State Key Laboratory of Networking and Switching Technology
Beijing University of Posts and Telecommunications, Beijing, China
10 Xitucheng Road, Beijing, China
E-mail: wangyawen@bupt.edu.cn, gongyz@bupt.edu.cn, laomao22311@126.com
(2) State Key Laboratory of Computer Architecture
Institute of Computing Technology, Chinese Academy of Sciences
(3) School of Electronic and Information Engineering
Liaoning Technical University
*Corresponding author: lovelyjamie@yeah.net

Abstract: The increasing complexity of 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 satisfaction problem solved by search strategies. In this paper, the search algorithm branch and bound is introduced and optimized to tackle the problem of path-wise test data generation. The optimized branching operation is fulfilled by a dynamic variable ordering algorithm with a heuristic rule to break ties. The optimized bounding operation is accomplished by analyzing the results of interval arithmetic. In order to facilitate the search methods, the solution space is represented as state space. Experimental results prove the effectiveness of the optimized branching and bounding operations, and show that the proposed method outperformed some other methods used in test data generation. The results also demonstrate that the proposed method is applicable in engineering.
Keywords: test data generation, constraint satisfaction problem, branch and bound, state space search.

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 [1]. In 2002, National Institute of Standards and Technology (NIST) found that over one third of the cost of software failure could be eliminated by an improved testing infrastructure [2]. But manual testing is time-consuming and error-prone, and is even impracticable for real-world programs. So the automation of testing is of crucial concern [3]. Furthermore, as a basic problem in software testing, path-wise test data generation (denoted as Q) is of particular importance because 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 program under test (PUT), and the metaheuristic (MHS) [4] methods are very popular. Recently, the MHS method particle swarm optimization (PSO) [5] has become a hot research topic due to its convenient implementation and faster convergence speed. But dynamic methods often consume a large number of iterations, and the definition of objective function is also a big problem. The static methods utilize techniques including symbolic execution [6] and interval arithmetic [7] 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

Copyright © 2006-2014 by CCC Publications

have been widely studied by many researchers. Demillo and Offutt [8] proposed a fault-based technique that used algebraic constraints to describe test data designed to find particular types of faults. Gotlieb et al. [9] 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 [10] and employed a variety of constraint solving optimizations. In 2013, Wang et al. [11] proposed an interval analysis algorithm using forward dataflow analysis. No matter what techniques are adopted, static methods require a strong constraint solver.

In this paper, considering the drawbacks of the dynamic methods and the demand for static methods, we propose a new static test data generation method based on Code Test System (CTS)(http://ctstesting.com), 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 satisfaction problem (CSP). Two techniques (state space search and branch and bound) in artificial intelligence are integrated to tackle the CSP. Second, the branching operation is optimized with a heuristic variable ordering algorithm. Third, the bounding operation is optimized in different stages of the search to reduce the search space greatly. Through experimental results, we try to evaluate the performance of our method, especially the optimized branching and bounding operations. We also make comparison experiments to find whether our method outperforms some currently existing test data generation methods in terms of coverage.

2 Problem definition and solving strategies

2.1 Problem definition

A control flow graph (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 $e=(n_r,\ n_t)\in E$ 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, $1\leq r<q$, $(\mathrm{n}_r,\ \mathrm{n}_{r+1})\in E$. A path p is regarded as feasible if there exists a program input for which p is traversed, otherwise p is regarded as infeasible. The problem Q is in essence a CSP [12]. $X$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,...,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$\{x_1\mapsto V_1,x_2\mapsto V_2,\ldots,x_n\mapsto V_n\}$.

A CSP is generally solved by search strategies, among which backtracking algorithms are widely used. In this paper, state space search [13] and the backtracking algorithm branch and bound (BB) [14] are introduced to solve the CSP. The process of exploring the solution space is represented as state space search, as introduced in our previous work [15] . 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. So this paper proposes best-first-search branch and bound (BFS-BB) to automatically generate the test data, and the branching and bounding operations have both been optimized.

Table 1: Some methods used in this paper

Name	Operation	Stage
Dynamic variable oredering (DVO)	Branching	State space search
Initial domain reduction (IDR)	Bounding	Initialization
Hill climbing (HC)	Branching	State space search

2.2 The search strategies

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). BFS-BB includes two stages: initialization and state space search. Some methods in BFS-BB are described in Table 1. BFS-BB is described by pseudo-codes as follows.

Algorithm 1 BFS-BB

Input p:the path to be traversed

Output $result\{Variable \mapsto Value\}$:test data making p feasible

1: $result \leftarrow null$
2: $path\ infeasible \leftarrow true$
3: call **Algorithm 3. Initial domain reduction**
4: **if** $path\ infeasible = true$ **then**
5: **return** infeasible path
6: call **Algorithm 2. Dynamic variable ordering**
7: $V_1 \leftarrow$select (D_1)
8: initial state$\leftarrow(null,\ x_1,\ D_1,\ V_1,\ active)$
9: $S_{cur} \leftarrow$initial state
10: **while** $x_i \neq null$ **do**
11: call **Algorithm 4. Hill climbing**
12: **if** $S_{cur}=(Pre, x_i, D_i, V_i,\ inactive)$ **then**
13: backtrack
14: **else** $result \leftarrow result \bigcup \{x_i \mapsto V_i\}$
15: $FV \leftarrow FV\text{-}\{x_i\}$
16: $PV \leftarrow PV+\{x_i\}$
17: call **Algorithm 2. Dynamic variable ordering**
18: $V_i \leftarrow$select (D_i)
19: $S_{cur} \leftarrow (Pre, x_i, D_i, V_i,\ active)$
20: final state $\leftarrow S_{cur}$
21: **return** $result$

The first stage is to perform the initialization operations corresponding to lines 1 to 9. At first, IDR (Section 3.2.1) is used to partially reduce the input domains of all variables and find infeasible paths on occasion. All the variables in FV are permutated by DVO (Section 3.1) to form a queue and its head x_1 is determined the first variable to be instantiated. A value V_1 is selected from the domain of x_1 (D_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 (Section 3.2.2) process begins for x_1. For brevity, our following explanation refers to the hill climbing process for each x_i in FV.

In the state space search stage as shown by lines 10 to 21 , Hill climbing utilizes interval arithmetic to judge whether V_i for x_i leads to a conflict or not. 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 into *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 into *inactive*, indicating that the local search for x_i ends and backtracking is inevitable. BFS-BB ends when the hill climbing processes for all the variables succeed and there is no more variable to be permutated. All the *extensive* nodes make the solution path.

3 Optimized branching and bounding operations

3.1 optimized branching operation

This part focuses on the branching operation, which concerns the ordering of variables that determines the next variable to be instantiated. 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 infeasible), because this can minimize the size of the overall search tree. The technique to break ties works when 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∈[1, k]) marks its level in the sequence of the branches, 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 while the latter has rank *infinity*. The comparison between a and *infinity* determines the ordering. The algorithm is described by pseudo-codes as follows.

Quicksort is utilized when permutating variables 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 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.

3.2 Optimized bounding operation

This part focuses on the bounding operation, which in fact is the optimization of interval arithmetic. For the purpose of improving efficiency, the optimized bounding operation is taken in both stages of BFS-BB.

Initial domain reduction

The optimized bounding operation taken in the initialization stage is used for initial domain reduction as well as infeasible path detection. Following is the introduction to the process that interval arithmetic functions. First we give the definition of branching condition.

Algorithm 2 Dynamic variable ordering

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 x_i:the selected variable to be next instantiated

```
1: Q_i ← quicksort(FV, |D_i|)
2: for i → 1:|Q_i| do
3:     if |D_i| ≠ |D_j| (j > i; x_i, x_j ∈ Q_i) then
4:         break
5:     else
6:         for (n_qa, n_qa+1)(a ∈ [1, k]) do
7:             if rank(n_qa, n_qa+1)(x_i) = rank(n_qa, n_qa+1)(x_j) then
8:                 a++
9:             elsepermutate x_i, x_j by rank(n_qa, n_qa+1)
10:                break
11: x_i ← head(Q_i)
12: return x_i
```

Definition 2. Let B be the set of Boolean values {true, false} and D^a be the domain of all variables before the a^{th} branch, if there are k branches along the path, the branching condition $Br(n_{qa}, n_{qa+1})$: $D^a \to B$ ($a \in [1, k]$)where n_{qa} is a branching node is calculated by formula (1).

$$Br(n_{qa}, n_{qa+1}) = \begin{cases} true, & D^a \bigcap \widetilde{D}^a \neq \emptyset; \\ false, & D^a \bigcap \widetilde{D}^a = \emptyset. \end{cases} \tag{1}$$

In formula (1), D^a satisfies all the a-1 branching conditions ahead and will be used as input for the calculation of the a^{th} branching condition, and $\widetilde{D}^a$which is a temporary domain is the result when calculating $Br(n_{qa}, n_{qa+1})$ with D^a and satisfies the a^{th} branching condition. $D^a \bigcap \widetilde{D}^a \neq \emptyset$ means that $D^a \bigcap \widetilde{D}^a$ satisfies all the a-1 branching conditions ahead and the a^{th} branching condition, ensuring that interval arithmetic can continue to calculate the remaining branching conditions, while $D^a \bigcap \widetilde{D}^a = \emptyset$ means that the path has been detected infeasible. In the initialization stage, following algorithm can be quite useful for infeasible path detection as well as initial domain reduction since it is quite clear that $D^1 \supseteq D^2 \ldots \supseteq D^k \supseteq D^{k+1}$.

Algorithm 3 Initial domain reduction

Input D^1:the input domain of all variables
Output D^{k+1}:the reduced domain of all variables

```
1: for i → 1:k do
2:     Br(n_qa, n_qa+1) ← false
3:     D̃^a ← calculate Br(n_qa, n_qa+1) with D^a
4:     if D^a ⋂ D̃^a ≠ ∅ then
5:         Br(n_qa, n_qa+1) ← true
6:         D^a+1 ← D^a ⋂ D̃^a
7:     else return
8: path infeasible ← false
9: return D^k+1
```

If the path is not infeasible, then D^{k+1} will be input as the domain of all variables for the state space search stage. If it is, because interval arithmetic analyzes the ranges of variables in

a conservative way, the path to be covered is no doubt infeasible, and it is unnecessary to carry out the following state space search.

Hill climbing

Hill climbing 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. Initial value is very important for hill climbing. Our initial value selection strategy was introduced thoroughly in [16], which is quite effective. This part focuses on the process that interval arithmetic judges whether the value (V_i) assigned to a variable (x_i) leads to a conflict or not. Different from the initialization stage, a conflict detected in state space search stage implies that the current V_i for x_i will not lead the search in the right direction, but another V_i may not. So what we are trying to find in this conflict is the information useful for the next search step. To be exact, we seek to find some information to help reduce the domain of the current variable x_i (D_i). In accordance with the model of state space search, we give the following formula of the objective function $F(V_i)$, which is used to reduce the current domain of D_i.

$$F(V_i) = V_i - \sum_{a=1}^{k} (D^a \bigcap \widetilde{D}^a)(x_i) \tag{2}$$

The reduction of D_i is carried out according to the result of the 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 likened to climbing a hill. $F(V_i)$=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 again 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 following pseudo-codes.

It can be seen that $F(V_i)$ provides both the upper and the lower bounds of D_i for its reduction, so the efficiency of the algorithm is improved greatly. After the conduction of Algorithm 4, the hill climbing for a variable(x_i) ends with two possibilities, which can be identified by *Type* of S_{cur}: *extensive* means that the peak is found and DVO for the next variable may continue, while *inactive* means that no search space is left for x_i, the peak is not found, and backtracking is inevitable.

3.3 Case study

In this part, we give a case study to explain in detail how BFS-BB works. An example with a program *test* and its corresponding CFG is shown in Figure 1. Adopting statement coverage, there is only one path to be traversed, namely, *Path*1: 0→1→2→3→4→5→6→7→8→9→10. We choose this example, because the constraints along the path are very strict for two variables. It is very obvious that $\{x_1 \mapsto 65, x_2 \mapsto 35\}$ is only one solution to the corresponding CSP.

For simplicity, the input domains of both variables are set [1, 100] with the size 100. Then IDR reduces them to $x1$:[31,99],$x2$:[1,69] as shown in Figure 2, which is used as input for the state space search stage. Then DVO works to determine the first variable to be instantiated as Table 2 shows, and $x1$ is picked out as shown in bold in the last column.

The initial value of $x1$ is selected from [31, 99] according to the method introduced in [16]. Assume that 80 is selected, then after three times of conflict and reduction to D_1, 65 is found for x_1 at last. $F(65)$=0 means that 65 is the peak of the hill that corresponding to x_1, which starts from 80. The hill climbing process is shown in Table 3. And since there is only one value 35 for $x2$, the search succeeds with $\{x_1 \mapsto 65, x_2 \mapsto 35\}$.

Algorithm 4 Hill climbing

Input D_i: the 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 succeeds
S_{cur}:$(Pre, x_i, D_i, V_i,\ extensive)$when hill climbing fails

1: **while** $|D_i|>1$ **do**
2: **for** $i \rightarrow 1{:}k$ **do**
3: $Br(n_{qa},\ n_{qa+1}) \leftarrow false$
4: $\widetilde{D}^a \leftarrow$calculate $Br(n_{qa}, n_{qa+1})$with D^a
5: **if** $D^a \bigcap \widetilde{D}^a \neq \emptyset$ **then**
6: $Br(n_{qa},\ n_{qa+1}) \leftarrow true$
7: $D^{a+1} \leftarrow D^a \bigcap \widetilde{D}^a$
8: **else** calculate $F(V_i)$
9: **break**
10: **if** $F(V_i) = 0$ **then**
11: $S_{cur} \leftarrow (Pre, x_i, D_i, V_i,\ extensive)$
12: **return** S_{cur}
13: **else**
14: **if** $F(V_i) < 0$ **then**
15: $D_i \leftarrow [V_i+1, V_i+F(V_i)]$
16: **else**$D_i \leftarrow [V_i\text{-}F(V_i), V_i\text{-}1]$
17: $V_i \leftarrow$select(D_i)
18: $S_{cur} \leftarrow (Pre, x_i, D_i, V_i,\ inactive)$
19: **return** S_{cur}

```
void test(int x1, int x2){
1:  if(x1>x2)
2:    if(x1+x2==100)
3:      if(x1>=20)
4:        if(x1-x2==30)
5:          printf("Solved! ");
}
```

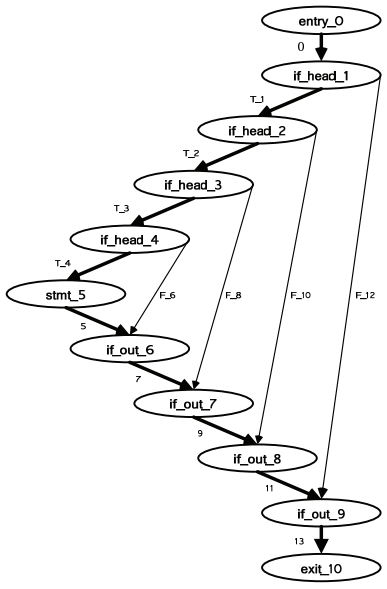

Figure 1: Program *test* and its corresponding CFG

Table 2: DVO process for $x1$ and $x2$

Ordering rule	Condition of each variable	Tie encountered?	Ordering result
Domain size	$\|D1\|$=69, $\|D2\|$=69	Yes	$\boldsymbol{x1}\rightarrow x2$
Rank1	Rank1($x1$)=1, Rank1($x2$)=1	Yes	
Rank2	Rank1($x1$)=2, Rank1($x2$)=2	Yes	
Rank2	Rank2($x1$)= 3, Rank2($x2$)=∞	No	

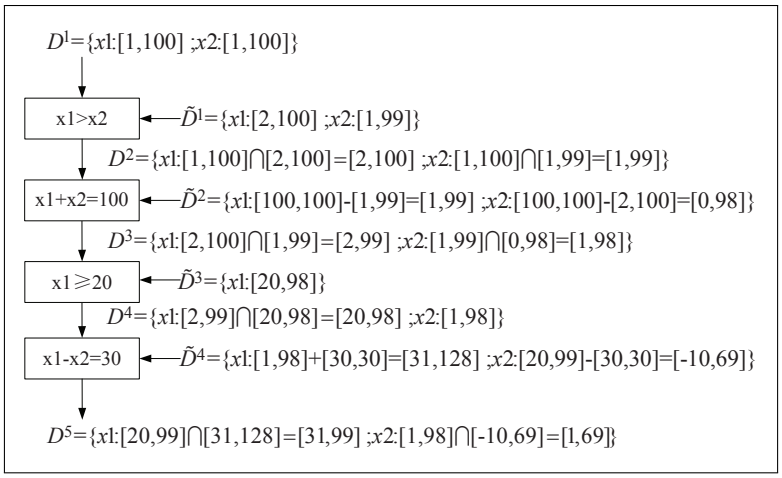

Figure 2: The IDR process

4 Experimental Analyses and Empirical Evaluations

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 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.

4.1 Performance evaluation

The number of variables is an important factor that affects the performance of test data generation methods [17]. Hence, in this part, experiments were carried out to evaluate the effectiveness of the optimized branching and bounding operations for varying numbers of input variables.

Testing the optimized branching operation

This part presents the comparison between our branching algorithm DVO (A) which is also BFS-BB and the method which orders variables only by remaining domain sizes (B). The other operations (the bounding operations) in the two methods were both accomplished by IDR and HC. The comparison was accomplished by repeatedly running the two methods on generated test programs having input variables x_1, x_2,..., x_n where n varied from 2 to 50. Adopting statement coverage, in each test the program contained n *if* statements (equivalent to n branching conditions or n 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. In each test, the expression of

Table 3: The hill climbing process for $x1$

D_1	V_1	$F(V_i)$	$\lvert F(V_i)\rvert$	Peak reached?
[31, 99]	80	30	30	No
[50, 79]	60	-10	10	No
[61, 70]	65	0	0	Yes

the i^{th} $(1 \leq i \leq n)$ *if* statement was in the form of

$$[a_1, a_2, \ldots, a_n][x_1, x_2, \ldots, x_n]\prime rel - op \quad const[i] \tag{3}$$

In formula(3), a_1, a_2,..., a_i were randomly generated numbers either positive or negative, $rel\text{-}op \in \{>, \geq, <, \leq, =, \neq\}$, and $const[j]$ $(j \in [1, i])$ was an array of randomly generated constants within [0, 1000]. The randomly generated a_j and $const[i]$ should be selected to make the path feasible. This arrangement constructed the tightest linear relation between the variables. The programs for various values of n ranging from 2 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 3.

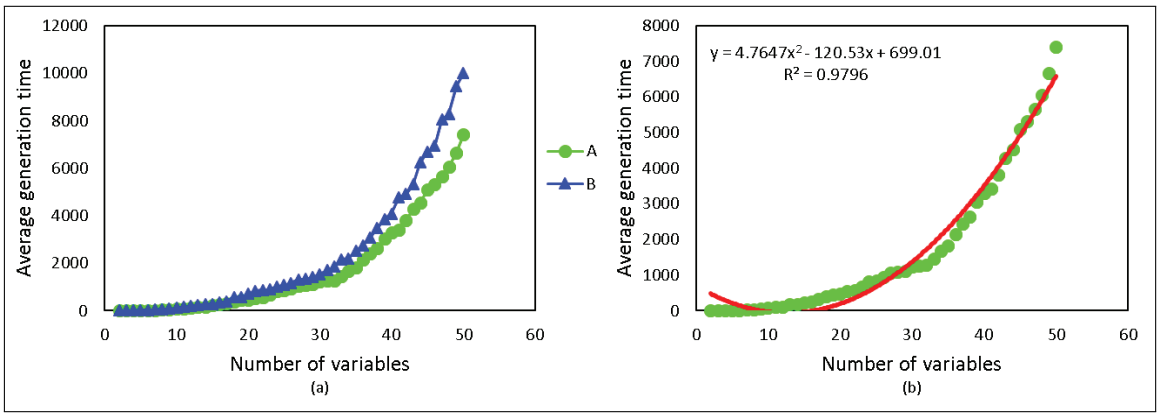

Figure 3: Test result of DVO

In Figure 3, (a) shows that A had a better performance than B, but it was not very obvious when the number of variables (expressions) was not very large, because there was no requirement for an optimized ordering algorithm, since remaining domain size was enough to determine the next variable to be instantiated. So the more variables, the better DVO works. For BFS-BB (A), it is clear that the relation between average generation time and the number of variables can be represented as a quadratic curve very well as shown in (b) 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=9.5294x-120.53 as the number of variables increases. We can roughly draw the conclusion that generation time using DVO is very close for n ranging from 1 to 25, while it begins to increase when n is larger than 25. So DVO will be very useful for PUTs with more variables, especially the large-scale real-world programs.

Testing the optimized bounding operation

This part presents the comparison between our bounding algorithm HC (A) which is also BFS-BB and the method without HC (B). The other operations (DVO and IDR) in the two

methods were totally the same. The comparison was accomplished by repeatedly running the two methods on generated test programs having input variables x_1, x_2,..., 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. The expression of each *if* statement was in the form of

$$[a_1, a_2, \ldots, a_n][x_1, x_2, \ldots, x_n]\prime rel - op \quad const[c] \tag{4}$$

In formula(4), a_1, a_2,..., a_i were randomly generated numbers either positive or negative, $rel\text{-}op \in \{>, \geq, <, \leq, =, \neq\}$, and $const[c]$ ($c \in [1, 50]$) was an array of randomly generated constants within [0, 1000]. The randomly generated a_i ($1 \leq i \leq n$) 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 comparison result is presented in Figure 4.

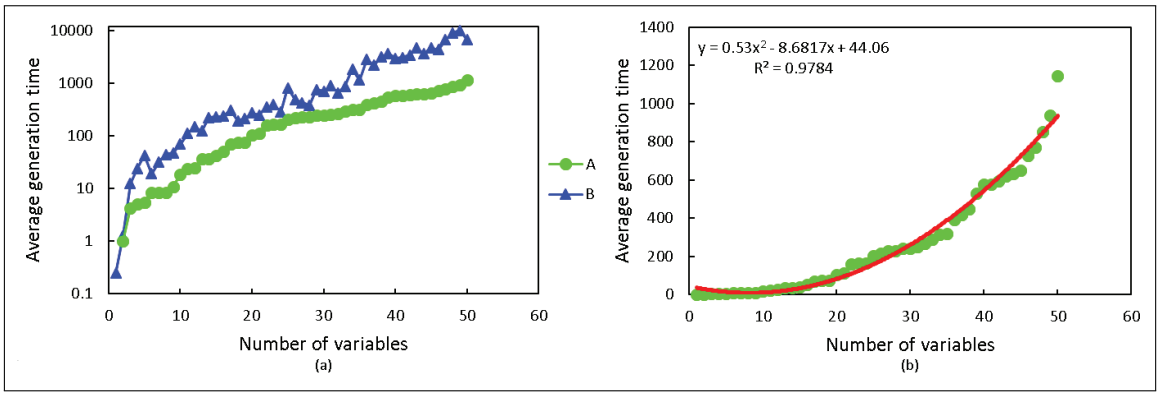

Figure 4: Test result of HC

We exponentiated the axis representing average generation time as shown in (a). It can be seen that the average generation time of A is far less than B. For BFS-BB (A), it is clear that the relation between average generation time and the number of variables can be represented as a quadratic curve very well as shown in (b) 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-8.6817 as the number of variables increases. We can roughly draw the conclusion that generation time using HC is very close for n ranging from 1 to 8, while it begins to increase when n is larger than 8.

4.2 Coverage evaluation

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

Comparison with a static method

This part presents the results from an empirical comparison of BFS-BB with the static method [11] (denoted as "method 1" to avoid verbose description), which was implemented in CTS prior

Table 4: The details of comparison with method 1

Project	program	Function	AC by method 1	AC by BFS-BB
qlib	sin.c	radian	55%	**100%**
	floor.c	ceil	66%	**100%**
dell8i-2	asinl.c	acosl	55%	**100%**
	tanl.c	cotl	66%	**100%**

Table 5: 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

to BFS-BB. The test beds were from two engineering projects at http://www.moshier.net/. The comparison adopted statement coverage as the adequacy criterion. For each test bed, the experiments were carried out 100 times, and average coverage (AC) was used for comparison. The details of the comparison are shown in Table 4. From Table 4, it can be seen that BFS-BB reached higher coverage than method 1 for all the test beds as shown in bold. That is largely due to the optimization methods utilized in BFS-BB. The applicability in engineering remains one of our primary goals in the future.

Comparison with PSO

This part presents results from an empirical comparison of BFS-BB with PSO, which is mentioned in Section 1 as a popular MHS method with relatively fast convergence speed. Table 5 is a brief introduction to some parameters used in PSO. We used three real-world programs, which are well-known benchmark programs and have been widely adopted by other researchers [18]-[20]. Branch coverage was taken as the adequacy criterion. For each test bed, the experiments were carried out 100 times, and AC was used for comparison. Table 6 shows the details of the test beds and the comparison results. Obviously BFS-BB achieved 100% coverage as shown in bold on all the three benchmark programs, which are rather simple programs for BFS-BB, and it outperformed the algorithm in comparison. The better performance of BFS-BB is due to two 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 the optimized bounding operation is conducted not only in the state space search stage but in the initialization stage as well, which reduces the domains of the variables to ensure a relatively small search space that follows.

Table 6: The details of comparison with PSO

Program	LOC	Branches	Variables	AC by PSO	AC by BFS-BB
triangleType	31	3	5	99.88%	**100%**
cal	53	18	5	96.85%	**100%**
calDay	72	11	3	97.35%	**100%**

5 Conclusions and Future Works

The increasing demand of testing large-scale real-world programs makes the automation of the testing process necessary. In this paper, path-wise test data generation (Q) which is a basic problem in software testing is defined as a constraint satisfaction problem (CSP), and the algorithm best-first-search branch and bound (BFS-BB) is presented to solve it, combining two techniques in artificial intelligence which are state space search and branch and bound (BB). The branching and bounding operations in BFS-BB are both optimized. For the branching operation, dynamic variable ordering (DVO) is proposed to permutate variables with a heuristic rule to break ties. The bounding operation is optimized in both stages of BFS-BB. Initial domain reduction (IDR) functions in the initialization stage to reduce the search space as well as detect infeasible paths. In the state space search stage, 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 a conflict. To facilitate the search procedure, the solution space is represented as state space. Empirical experiments show that the optimized branching operation is especially useful for large-scale programs, while the advantage of the optimized bounding operation hill climbing (HC) is very obvious. The results also show that BFS-BB outperforms some current static and dynamic methods in terms of coverage. Our future research will involve how to generate test data to reach high coverage. The effectiveness of the generation approach continues to be our primary work.

Acknowledgment

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).

Bibliography

[1] Michael R. Lyu; Sampath Rangarajan; Ada P. A. van Moorse. (2002); Optimal allocation of test resources for software reliability growth modeling in software development, *IEEE Transactions on Reliability*, ISSN 1841-9836, 51(2): 183-192.

[2] Tassey Gregory.(2002);The economic impacts of inadequate infrastructure for software testing, *National Institute of Standards and Technology, RTI Project 7007.011.*

[3] Weyuker Elaine J.(1999); Evaluation techniques for improving the quality of very large software systems in a cost-effective way, *Journal of Systems and Software*, ISSN 0164-1212, 47(2): 97-103.

[4] Shaukat Ali, Lionel C. Briand; Hadi Hemmati; Rajwinder K. Panesar-Walawege. (2010); A systematic review of the application and empirical investigation of search-based test case generation, *IEEE Transactions on Software Engineering*, ISSN 0098-5589, 36(6): 742-762.

[5] Mao Chengying; Yu Xinxin; Chen Jifu.(2012); Swarm Intelligence-Based Test Data Generation for Structural Testing, *Proceedings of 11th International Conference on Computer and Information Science (ICIS 12)*,623-628.

[6] Suzette Person; Guowei Yang; Neha Rungta; Sarfraz Khurshid. (2012); Directed incremental symbolic execution, *IACM SIGPLAN Notices*, ISSN 0362-1340, 46(6): 504-515.

[7] Moore Ramon Edgar; R. Baker Kearfott; Michael J. Cloud.(2009);*Introduction to interval analysis*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[8] Richard A. DeMillo; A. Jefferson Offutt. (1991); Constraint-based automatic test data generation, *IEEE Transactions on Software Engineering*, ISSN 0098-5589, 17(9): 900-910.

[9] Arnaud Gotlieb; Bernard Botella; Michel Rueher.(1998);Automatic test data generation using constraint solving techniques, *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 23(2):53-62.

[10] Cristian Cadar; Daniel Dunbar; Dawson Engler.(2008);KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs, *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, 209-224.

[11] Wang Yawen; Gong Yunzhan; Xiao Qing. (2013); A Method of Test Case Generation Based on Necessary Interval Set, *Journal of Computer-Aided Design & Computer Graphics*, ISSN 1003-9775, 25(4): 550-556.

[12] A.E. Eiben; Zs Ruttkay.(1997);*Constraint satisfaction problems*, pp. C5.7:1-8, New York, NY, USA: IOP Publishing Ltd and Oxford University Press.

[13] Ling-Ling Wang; Wen-Hsiang Tsai. (1988); Optimal assignment of task modules with precedence for distributed processing by graph matching and state-space search, *BIT Numerical Mathematics*, ISSN 0003-3835, 28(1): 54-68.

[14] Lianbo Gao; Shashi K. Mishra; Jianming Shi. (2012); An extension of branch-and-bound algorithm for solving sum-of-nonlinear-ratios problem, *Optimization Letters*, ISSN 1862-4472, 6(2): 221-230.

[15] Ying Xing; Junfei Huang; Yunzhan Gong; Yawen Wang; Xuzhou Zhang. (2014); An Intelligent Method Based on State Space Search for Automatic Test Case Generation, *Journal of Software*, ISSN 1796-217X, 9(2): 358-364.

[16] Ying Xing; Junfei Huang; Yunzhan Gong; Yawen Wang; Xuzhou Zhang. (2014); Path-wise Test Data Generation Based on Heuristic Look-ahead Methods, *Mathematical Problems in Engineering*, ISSN 1024-123X,volume 2014, Article ID 642630.

[17] Matthew J Gallagher; V. Lakshmi Narasimhan. (2012); Adtest: A test data generation suite for Ada software systems, *IEEE Transactions on Software Engineering*, ISSN 0098-5589, 23(8): 473-484.

[18] Mao Chengying; Yu Xinxin; Chen Jifu.(2012); Generating Test case for Structural Testing Based on Ant Colony Optimization, *Proceedings of the 12th International Conference on Quality Software (QSIC12)*, 98-101.

[19] Ammann Paul; Jeff Offutt.(2008); *Introduction to Software Testing*, Cambridge University Press, New York, NY, USA.

[20] E. Alba; F. Chicano. (2008); Observation in Using Parallel and Sequential Evolutionary Algorithms for Automatic Software Testing, *Computers & Operators Research*, ISSN 0305-0548, 35(10): 3161-3183.