A Dynamic Fitness Function for Search Based Software Testing

Xiong Xu
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
xux@ios.ac.cn

Li Jiao
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China
ljiao@ios.ac.cn

Ziming Zhu
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
zhuzm@ios.ac.cn

ABSTRACT

Search Based Software Testing (SBST) formulates testing as an optimization problem, hence some search algorithms can be used to tackle it. The goal of SBST is to improve various test adequacy criteria. There are different types of coverage criteria, and in this paper, we deal with branch coverage, as it is the mostly used criterion. However, the state of the art fitness function definitions used for branch coverage testing have changed little. To fill this gap, this paper proposes a novel fitness function for branch coverage. Concretely, we first use a negative exponential model to evaluate the hardness of covering essential branches of the program, based on which we approximately evaluate the distance of a test candidate to the target branch. Finally, the experiment reveals the promising results of our proposal.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; *Software verification and validation*;

KEYWORDS

Search based software testing, test data generation, branch coverage, fitness functions

ACM Reference Format:
Xiong Xu, Li Jiao, and Ziming Zhu. 2018. A Dynamic Fitness Function for Search Based Software Testing. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, Article 4, 2 pages. https://doi.org/10.1145/3205651.3205700

1 INTRODUCTION

Search Based Software Testing (SBST) has achieved a great deal of recent attention. It uses meta-heuristic algorithms to automate the generation of test inputs that meet test adequacy criteria [3]. One of the most widely-studied test adequacy criteria in SBST is branch coverage [1, 2, 4], which is considered in this paper.

The fitness function is the most critical part in SBST, as it is used to express the heuristic information of the problem, by which the heuristic search algorithms can work efficiently [5]. However, most of the researches for SBST focus on changing the search algorithms, rather than the underlying fitness functions on which all heuristic search relies. This paper takes a novel approach and proposes a dynamic fitness function for branch coverage. In particular, we use a negative exponential model to evaluate the coverage hardness of essential branches in the program, then design an approximation algorithm to evaluate efficiently the distance of a test candidate to the target branch. As the branch hardness varies with the search iteration, a dynamic fitness function can be obtained. This fitness function can explore dynamically the heuristic information of the problem, thus it is richer and more expressive than its counterparts.

The rest of the paper is organized as follows. The next section introduces evaluating the hardness of covering essential branches. Section 3 proposes the dynamic fitness function, with the corresponding experiment in Section 4. Section 5 draws the conclusion.

2 ESSENTIAL BRANCH HARDNESS

Given a program under test and a target branch, we say an **if**-statement [**if** c **then** B_c **else** $B_{\neg c}$] *controls* the target, iff the target appears in B_c or $B_{\neg c}$. If the target is in B_c, we say the branch representing the condition c *controls* the target. If the target is in $B_{\neg c}$, we say the *mutually-exclusive* branch (the branch representing the condition $\neg c$) *controls* the target. Similarly, we say a **while**-statement [**while** c **do** B] *controls* the target, iff the target appears in B. Now, we can define the essential branches with respect to the target branch in the program.

Definition 2.1 (Essential Branches). A branch is called *essential* iff it can control the target branch.

In order to explore more heuristic information of the problem, we can evaluate the hardness of covering each essential branch. Let $\prec$ be the control relation on branches. For two branches b_1 and b_2, we say $b_1 \prec b_2$ iff b_1 controls b_2. The following theorem denotes that $\prec$ is a strictly total order.

Theorem 2.2 (Strictly Total Order). *$(EB, \prec)$ is a strictly totally ordered set, where EB is the set of essential branches.*

Let b_i be the branch whose rank is i in the strictly totally ordered set EB. Let n_i be the number of test candidates that can cover b_i. If $i < j$ ($b_i \prec b_j$), then any test candidate that covers b_j can as well cover b_i, i.e., $n_i > n_j$, which means b_j is harder than b_i. Therefore, we can use the value $1/n_i$ to express the hardness of the branch b_i. However, n_i may be 0, which means $1/n_i$ makes no sense. Thus, we propose an approach to approximate n_i using $\widehat{n}_i$ and guarantee

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5764-7/18/07.
https://doi.org/10.1145/3205651.3205700

that $\widehat{n}_i > 0$. As n_i is inversely proportional to i, we can assume that n_i approximately has negative exponent relation to i. Concretely, we can assume $\widehat{n}_i = \exp(\alpha_0 + \alpha_1 i)$ for two parameters α_0 and α_1. These two parameters can be obtained easily by the Least Square Method. Obviously, $\widehat{n}_i > 0$ and we can let $h(b_i) = 1/\widehat{n}_i$ denote the hardness of b_i though $n_i = 0$.

3 THE DYNAMIC FITNESS FUNCTION

If the test candidate meets an essential branch but cannot satisfy the branch condition, then it executes the mutually-exclusive branch. However, the target branch may have no chance to be covered after executing the mutually-exclusive branch. This kind of essential branches is called the critical branch, which is defined as follow.

Definition 3.1 (Critical Branches). A branch is called *critical* iff (1) it is essential and (2) there is no path from the start node to the target branch through its mutually-exclusive branch.

If a critical branch is not executed, the program execution is terminated because its execution cannot lead to the target branch. Thus, if the test candidate $\boldsymbol{\tau}$ meets a critical branch b but cannot satisfy the branch condition c, then $\boldsymbol{\tau}$ has to compulsively execute b, and should pay the cost $\delta(\boldsymbol{\tau}, c) \cdot h(b)$, where $\delta(\boldsymbol{\tau}, c)$ is the branch distance [4] of $\boldsymbol{\tau}$ on satisfying c, and $h(b)$ is the hardness of covering b. However, compulsively executing critical branches repeatedly may lead to high computing cost. Thus, if a critical branch b cannot be satisfied for twice, then we estimate the remaining cost of $\boldsymbol{\tau}$ reaching the target branch from current branch b, and therefore a candidate fitness value can be obtained. After that, we put the candidate fitness value into the fitness set, and return the minimum element in the fitness set as the fitness value of the test candidate.

The remaining cost here is similar with the path distance defined in [1]. Let π be the shortest path from b to the target. By the symbolic execution, we can get the predicate set $P(\pi)$ of the path π. Each predicate $p \in P(\pi)$ corresponds to a branch b_p in π. Then, the remaining cost can be computed as

$$\Delta(\boldsymbol{\tau}, b, h) = \sum_{p \in P(\pi)} \delta(\boldsymbol{\tau}, p) \cdot h(b_p),$$

and the candidate fitness can be computed as $cost + \Delta(\boldsymbol{\tau}, b, h)$, where $cost$ is the cost has been payed before $\boldsymbol{\tau}$ meets b.

If the test candidate meets an essential but not critical branch, a candidate fitness can be obtained by the remaining cost and is put into the fitness set, then the mutually-exclusive branch is executed. If the target is covered, the program execution is terminated and the current cost that has been payed is returned directly as the fitness value of the test candidate. The correctness and complexity of the fitness evaluation can be demonstrated by the following theorems.

Theorem 3.2 (Correctness). *A test candidate that can cover the target branch iff its fitness value is evaluated as the minimum* 0.

Theorem 3.3 (Complexity). *The expected time complexity of the fitness evaluation is* $\Theta(|EB|)$*, where EB is the set of essential branches.*

Since the number n_i of test candidates that can cover the branch b_i varies with the search iteration, the fitness value of a test candidate is different at each iteration, which means this fitness function is dynamic and can explore in the search process the dynamic heuristic information of the problem.

Table 1: The average SR and TC of each fitness function

Performance	Fitness Functions			
	F_{AB}	F_{SE}	F	F_D
Average SR	38.2%	47.3%	73.3%	**81.8%**
Average TC (s)	**4.201**	6.708	8.119	7.140

4 EXPERIMENT

Let F_D be the dynamic fitness function proposed in this paper. We compare our proposal F_D to the state of the art, i.e., the most commonly used fitness function F_{AB} [4], and the newly proposed fitness function F_{SE} [1]. These two fitness functions are representative and were proposed for branch coverage. Besides, in order to observe the effect of the dynamic hardness evaluation for branches, we record as well the performance of the fitness function F, a simplified version of F_D without taking into account branch hardness.

We describe our experiment with 15 programs, and they are `ei`, `gammp`, `bessj`, `bessik`, `ran2`, `expint`, `plgndr`, `brent`, `dbrent`, `BinTree`, `IntRedBlackTree`, `NodeCachingLL`, `SinglyLinkedList`, `TreeSet`, and `WBS`. These programs usually have complex structures and constraints, and some of them have been as examples in several papers on SBST. For each program sample, we randomly select with replacement 100 hard branches as the targets. For each target, we perform one search on each fitness function. Each search does not terminate until the target branch is covered or the result cannot be improved in 100 iterations. The success of each search is recorded, along with the time cost required to find the test data. From this, the success rate (SR) and the average time cost (TC) of each approach (fitness function) can be calculated. In this paper, we use the GA as the search algorithm, and the parameters of the GA are based on the recommendation of [4]. The average experimental results can be seen in Table 1, which reveals the effectiveness of the dynamic fitness function proposed in this paper.

5 CONCLUSION

In this paper, we proposed a novel fitness function to boost the performance of SBST for branch coverage. The experiment reveals that our proposal can improve the SR significantly although the TC is high. In fact, the SR of testing is more important than the TC, especially for some safety-critical software systems, because obtaining a lower TC is not very hard, while achieving a higher SR usually needs a deep understanding of the specific problems. In future, we will focus on extending our proposal for other coverage criteria, and performing more experiments on industrial programs.

REFERENCES

[1] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. 2011. Symbolic Search-Based Testing. In *IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 53–62.
[2] G. Fraser and A. Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.
[3] M. Khari and P. Kumar. 2017. An Extensive Evaluation of Search-based Software Testing: A Review. *Soft Computing* (November 2017), 1–14.
[4] J. Wegener, A. Baresel, and H. Sthamer. 2001. Evolutionary Test Environment for Automatic Structural Testing. *Information and Software Technology* 43, 14 (2001), 841–854.
[5] X. Xu, Z. Zhu, and L. Jiao. 2017. An Adaptive Fitness Function Based on Branch Hardness for Search Based Testing. In *The Genetic and Evolutionary Computation Conference*. ACM, 1335–1342.