Towards Optimal Concolic Testing

Xinyu Wang
Zhejiang University
wangxinyu@zju.edu.cn

Jun Sun
Singapore U. of Tech. and Design
sunjun@sutd.edu.sg

Zhenbang Chen
National U. of Defense Technology
zbchen@nudt.edu.cn

Peixin Zhang
Zhejiang University
zhangpeixin@zju.edu.cn

Jingyi Wang
Singapore U. of Tech. and Design
jingyi_wang@mymail.sutd.edu.sg

Yun Lin
National University of Singapore
llmhyy@gmail.com

ABSTRACT

Concolic testing integrates concrete execution (e.g., random testing) and symbolic execution for test case generation. It is shown to be more cost-effective than random testing or symbolic execution sometimes. A concolic testing strategy is a function which decides when to apply random testing or symbolic execution, and if it is the latter case, which program path to symbolically execute. Many heuristics-based strategies have been proposed. It is still an open problem what is the optimal concolic testing strategy. In this work, we make two contributions towards solving this problem. First, we show the optimal strategy can be defined based on the probability of program paths and the cost of constraint solving. The problem of identifying the optimal strategy is then reduced to a model checking problem of Markov Decision Processes with Costs. Secondly, in view of the complexity in identifying the optimal strategy, we design a greedy algorithm for approximating the optimal strategy. We conduct two sets of experiments. One is based on randomly generated models and the other is based on a set of C programs. The results show that existing heuristics have much room to improve and our greedy algorithm often outperforms existing heuristics.

ACM Reference Format:
Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards Optimal Concolic Testing. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180177

1 INTRODUCTION

Concolic testing, also known as dynamic symbolic execution, is an integration of concrete execution (a.k.a. testing) with symbolic execution [22, 41]. Concrete execution and symbolic execution naturally complement each other. On one hand, concrete execution is computationally cheap. That is, we keep sampling test inputs according to a prior probabilistic distribution of all test inputs, and concretely execute the program with the test inputs until certain test coverage criteria is satisfied. The issue is that if a certain program path has very low probability, a huge number of test inputs must be sampled to cover the program path. On the other hand, symbolic execution solves this problem by identifying the constraint which must be satisfied in order to cover the program path and solving the constraint to obtain the test input. In other words, the probability of covering the program path with symbolic execution is one[1]. The issue is that symbolic execution is often computationally expensive. Intuitively, an effective concolic testing strategy should symbolically execute those program paths with low probability and concretely execute those program paths whose path conditions are hard to solve.

It is still an open problem on what is the optimal concolic testing strategy. In the literature, there have been multiple attempts on solving the problem [6, 7, 23, 33, 38, 42]. For instance, several heuristics have been developed to answer the question: which program paths (among all program paths) do we symbolically execute in concolic testing? To name a few, Burnim *et al.* proposed the CFG strategy [6], which calculates the distance from the branches in an execution path to any of the uncovered statements and selects a branch that has the minimum distance. In [23], Godefroid *et al.* proposed the generational strategy, which measures the incremental coverage gain of each branch in an execution path and guides the search by expanding the branch with the highest coverage gain. In [33], Li *et al.* introduced a technique which steers symbolic execution to less traveled paths. While existing heuristics have been shown to be effective empirically, it is unclear whether better performance is achievable or how far they are from the optimal performance.

Furthermore, existing work has largely neglected the other part of the problem, i.e., how do we switch between concrete execution and symbolic execution to achieve the optimal performance? To the best of our knowledge, this problem was only recently discussed in [3, 4, 45]. The authors compare the effectiveness of random testing and systematic testing methods (including but not limited to symbolic execution) based on a probabilistic view of programs, and present a hybrid strategy which switches from random testing to systematic

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05...$15.00
https://doi.org/10.1145/3180155.3180177

[1] For simplicity, we assume that the constraint encoding and solving are perfect and thus there is no divergence.

testing when the latter is expected to discover more errors per unit time. Their approach however takes a very abstract view of systematic testing methods and do not consider, for instance, different strategies on applying symbolic execution. Furthermore, their algorithm is very high-level and is only validated on simulated models.

In this work, we aim to develop a framework which allows us to define and compute the optimal concolic testing strategy. That is, we aim to systematically answer when to apply concrete execution, when to apply symbolic execution and which program path to apply symbolic execution to. In particular, we make the following technical contributions. Firstly, we show that the optimal concolic testing strategy can be defined based on a probabilistic abstraction of program behaviors. Secondly, we show that the problem of identifying the optimal strategy can be reduced to a model checking problem of Markov Decision Processes with Costs. As a result, we can reuse existing tools and algorithms to solve the problem. Thirdly, we evaluate existing heuristics empirically using a set of simulated experiments and show that they have much room to improve. Fourthly, in view of the high complexity in computing the optimal strategy, we propose a greedy algorithm which approximates the optimal one. We empirically evaluate the greedy algorithm based on both simulated experiments and experiments with C programs, and show that it gains better performance than existing heuristics in KLEE [7].

The remainders of the paper are organized as follows. Section 2 defines the research problem and shows its relevance with an example. Section 3 reduces the problem to a model checking problem and compares existing heuristics to the optimal strategy. Section 4 develops a greedy algorithm which allows us to approximate the optimal strategy. Section 5 presents our implementation and evaluates the greedy algorithm. Section 6 reviews related work and Section 7 concludes.

2 PROBLEM DEFINITION

In the following, we define the problem. Without loss of generality, we define a program (e.g., Java/C) as follows.

Definition 2.1. A program is a labelled transition system $\mathcal{P} = (C, init, V, \phi, T)$ where

- C is a finite set of control locations;
- $init \in C$ is a unique entry point (i.e., the start of the program);
- V is a finite set of variables;
- ϕ is a predicate capturing the set of initial valuations of V;
- $T : C \times GC \rightarrow C$ is a transition function[2] where each transition is labeled with a guarded command of the form $[g]f$ where g is a guard condition and f is a function updating valuation of variables V.

A concrete execution (a.k.a. a test) of $\mathcal{P}$ is a sequence $\pi = \langle (v_0, c_0), gc_0, (v_1, c_1), gc_1, \cdots, (v_k, c_k), gc_k, \cdots \rangle$ where v_i is a valuation of V, $c_i \in C$, $gc_i = [g_i]f_i$ is a guarded command such that $(c_i, gc_i, c_{i+1}) \in T$, $v_i \vDash g_i$, and $v_{i+1} = f_i(v_i)$ for all i, and $v_0 \vDash \phi$ and $c_0 = init$. We say π covers a control location c if and only if c is in the sequence. A control location c is reachable if and only if there exists a concrete execution which covers c. The initial variable valuation v_0 is also referred to as a test case.

A (rooted) program path of $\mathcal{P}$ is a sequence of connected transitions $\pi = \langle (c_1, gc_1, c_2), (c_2, gc_2, c_3), \cdots, (c_k, gc_k, c_{k+1}) \rangle$ such that $c_1 = init$ and $(c_i, gc_i, c_{i+1}) \in T$ for all i. The corresponding path condition is: $PC(\pi) = \exists v_2, \cdots, v_{k+1}.\ g_1 \wedge (v_2 = f_1(v_1)) \wedge g_2 \wedge \cdots \wedge g_k \wedge (v_{k+1} = f_k(v_k))$. We write $path(\mathcal{P})$ to denote all paths of program $\mathcal{P}$.

Example 2.2. Figure 1 shows a simple Java program. The corresponding transition system is shown in the middle of Figure 1, where the commands are skipped for readability. The transition system contains 8 control locations, corresponding to the 8 numbered lines in the program. We assume that each line is atomic for simplicity. The initial condition ϕ is $x \in Int \wedge y \in Int$ where Int is the set of all integers.

For simplicity, we assume that the goal is to generate test cases so that the corresponding concrete executions cover all reachable control locations (i.e., 100% statement coverage). In the literature, there have been many approaches on test case generation [11, 12, 26]. In this work, we focus on two ways of generating test cases.

One is random testing. To conduct random testing, we fix a prior distribution μ on all the test cases and then randomly sample a test case each time according to μ. Afterwards, we execute the program with the sampled test case until it finishes execution. For instance, if we assume a uniform distribution on all test cases for the program shown in Figure 1, random testing is to randomly generate a value for x and y and then concretely execute the program. The cost of random testing, in terms of time, is often small. In this work, we simply assume that the cost is 1 time unit[3]. Assume that every test case is associated with certain non-zero probability in μ, it is trivial to show eventually we can enumerate all test cases through random testing and cover all reachable control locations. Unfortunately, in practice we have limited time and budget and thus we may not be able to cover certain control locations with a limited number of random test cases. For instance, with a uniform probability distribution among all possible values for x and y, on average it takes 2^{32} random test cases to cover line 2 in Figure 1.

Another way of generating test cases is symbolic execution [12]. Given a program path, a constraint solver is employed to check the satisfiability of the path condition and construct a test case if it is satisfiable. Afterwards, we execute the program with the test case until it finishes execution. Symbolic execution may sometimes be more cost-effective than random testing. For instance, with the constraint solver Z3 [14], we can easily solve the path condition (i.e., $x == y$) for visiting line 2 in Figure 1 to generate the required test

[2] We focus on deterministic sequential programs in this work.

[3] The cost of one random testing varies widely in practice. We will extend our work with variable random testing cost in the future work.

```
void myfunc(int x, int y) {
1.   if (x==y) {
2.       x++;
     }
3.   if ((x*x)%10==9) {
4.       return;
     }
5.   if (x*y + 3*y - 5*x == 15) {
6.       if (x % 2 == 1 || y % 2 == 1) {
7.            x = x-y;
         }
     }
8.   return;
}
```

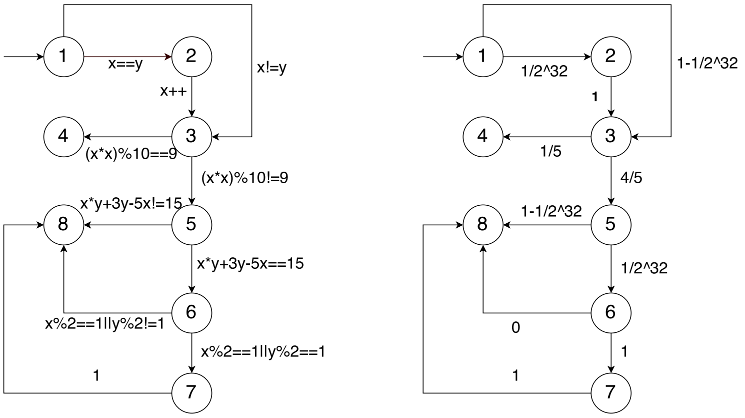

Figure 1: Abstraction

case. However, symbolic execution may not always be cost-effective. For instance, to obtain a test covering line 4, we can apply symbolic execution to solve the path condition which includes the condition at line 3. It is likely to be non-trivial due to the non-linear constraint. In comparison, generating a random test case to satisfy the condition at line 3 is much easier, i.e., on average 5 random test cases are needed. In general, the cost of symbolic execution is considerably more than that of random testing as constraint solving could be time-consuming.

Furthermore, when symbolic execution is applied to generate a test case for covering a certain control location, we can either solve the path condition of a path ending with the control location or the path condition of its prefix. For instance, in order to cover line 7, we can either solve the path composed of line 1, 3, 5, 6 and 7, or the path composed of line 1, 3, 5 and 6 (once or multiple times) to generate test cases. The latter might be more cost-effective as the constraint to be solved has fewer clauses. In this particular example, solving the latter once is sufficient to cover line 7.

Concolic testing is the integration of random testing and symbolic execution. In this work, we define a strategy for concolic testing to be a function which generates a choice between random testing or symbolic execution (on a certain path) repeatedly until the testing goal is achieved. Two extreme ones are: (1) applying random testing always, and (2) applying symbolic execution for each program path. There are many alternative ones [6, 7, 23, 33, 38, 42]. Multiple strategies have been adopted in existing concolic testing engines (e.g., KLEE [7], Pex [47] and JDart [34]). As we show above, one strategy might be more cost-effective than others for certain programs. For instance, for the example shown in Figure 1, a 'better' strategy would apply symbolic execution to the path composed of line 1 and 2 (to cover line 2), apply symbolic execution to the path composed of 1, 3, 5 and 6 (to cover line 7), and apply random testing to cover the rest of the lines. The question is then how to compare different strategies. In this work, we investigate the effectiveness of different strategies for concolic testing and answer the following open questions.

RQ1: What is the optimal concolic testing strategy given a program?
RQ2: Can we efficiently compute the optimal strategy?
RQ3: Are existing strategies good approximation of the optimal strategy?
RQ4: Is it possible to design a practical algorithm to approximate the optimal strategy?
RQ5: If the answer to RQ4 is positive, how does the algorithm compare to existing heuristics?

We answer these questions in the following sections.

We remark that we do not consider strategies which simplify complex symbolic constraints using concrete values in this work. Furthermore, we assume that the path condition encoding and solving are perfect and thus there is no divergence. Considering these would considerably complicate the discussion and thus we leave it to future work.

3 OPTIMAL STRATEGY

In this section, we show that the optimal concolic testing strategy can be defined based on the probability of program paths and the cost of constraint solving. Furthermore, it can computed through model checking.

3.1 Markov Chain Abstraction

To answer *RQ1*, we first develop an abstraction of programs in the form of Markov Chains.

Definition 3.1. A (labeled) discrete time Markov Chain (DTMC) is a tuple $\mathcal{M} = (S, Pr, \mu)$ where S is a finite set of states; $Pr : S \times S \to \mathbb{R}^+$ is a labeled transition probability function such that $\Sigma_{s' \in S} Pr(s, s') = 1$ for all $s \in S$; and μ is the initial probability distribution such that $\Sigma_{s \in S} \mu(s) = 1$.

A state $s \in S$ is called a sink state if there are no outgoing transitions from s. We often write $Pr(s, s')$ to denote the conditional probability of visiting s' given the current state s. The conditional probability $Pr(s, s')$ is also called as one-step transition probability. A path of $\mathcal{M}$ is a sequence of states $\pi = \langle s_0, s_1, s_2, \cdots \rangle$. We write $states(\pi)$ to denote the set of states in π. Let $Path(\mathcal{M})$ denote all paths of $\mathcal{M}$. The probability of π, written as $Pr(\pi)$, is the product of all the one-step transition probability, i.e., $Pr(\pi) = \mu(s_0) \times \Pi_i Pr(s_i, s_{i+1})$. Given a finite path π, we write $last(\pi)$ to denote the ending state in the sequence; and $2last(\pi)$ to denote the second last state. We say that a finite path π is maximal if $last(\pi)$ is a sink state. We write $Path^{max}(\mathcal{M})$ denote all maximal paths of $\mathcal{M}$. We write $Path^{max}(s, \mathcal{M})$ denote all maximal paths of $\mathcal{M}$ starting with s. Furthermore, we say that π is *non-repeating* if every state in π appears at most once. We write $Path(\mathcal{M}, s)$ to denote all finite paths which end with state s. The accumulated probability of all paths in $Path(\mathcal{M}, s)$ is the probability of reaching s, written as $Pr_{\mathcal{M}}(reach(s))$ for simplicity. Similarly, we write $Path(\mathcal{M}, s, s')$ to denote all finite paths which start with state s and end with state s' and $Pr_{\mathcal{M}}(reach(s, s'))$ to denote the accumulated probability of all paths in $Path(\mathcal{M}, s, s')$.

In the following, we develop a DTMC interpretation of a program, which forms the basis of subsequent discussion.

Definition 3.2. Let $\mathcal{P} = (C, init, V, \phi, T)$ be a program and μ be a prior probability distribution of the test inputs. The DTMC interpretation of $\mathcal{P}$ is a DTMC $\mathcal{M}_{\mathcal{P}} = (S, Pr, \mu)$ such that a state in S is a pair (v, l) where v is a valuation of V and l is a control location in C; and Pr is defined as follows: $Pr((v, l), (v', l')) = 1$ if and only if there exists a guarded command $gc = [g]f$ such that $T(l, gc) = l'$ and $v \vDash g$ and $v' = f(v)$; otherwise $Pr((v, l), (v', l')) = 0$.

Note that in the above definition, each one-step transition has probability 1 or 0 except the initial probability distribution μ. Our optimal concolic testing strategy is defined based on one particular abstraction of $\mathcal{M}_{\mathcal{P}}$, i.e., the one which abstracts away the variable valuation, defined as follows.

Definition 3.3. Let $\mathcal{P} = (C, init, V, \phi, T)$ be a program and $\mathcal{M}_{\mathcal{P}} = (S, Pr, \mu)$ be its DTMC interpretation. The data-abstract DTMC interpretation of $\mathcal{P}$ is a DTMC $\mathcal{M}^a_{\mathcal{P}} = (S_a, Pr_a, \mu_a)$ such that $S_a = C$. It is useful since we focus on statement coverage in this work.

- $\mu_a(l) = 1$ if l is $init$; and 0 otherwise;
- and Pr_a is defined as follows: for all $l \in C$ and $l' \in C$, $Pr_a(l, l')$ is

$$\frac{\Sigma\{Pr(\pi) | \exists s, s'.\ \pi \in Path(\mathcal{M}_{\mathcal{P}}, (s', l')) \wedge 2last(\pi) = (s, l)\}}{\Sigma\{Pr(\pi) | \exists s.\ \pi \in Path(\mathcal{M}_{\mathcal{P}}, (s, l))\}}$$

Intuitively, $Pr_a(l, l')$ is the probability of visiting l and immediately followed by l', over the probability of reaching l. For instance, the DTMC shown on the right of Figure 1 is the data-abstract DTMC interpretation of the program on the left, where each control location in the program becomes a state in the DTMC and each control flow between two control locations is associated with the corresponding conditional probability. For instance, the probability $\frac{1}{2^{32}}$ labeled with the transition from state 1 to 2 states that the probability of visiting state 2 after state 1 is $\frac{1}{2^{32}}$ (if we assume a uniform distribution among all test inputs).

The following proposition states that the probability of reaching a control location l is preserved in $\mathcal{M}^a_{\mathcal{P}}$.

PROPOSITION 3.4. *Let $\mathcal{P} = (C, init, V, \phi, T)$ be a program and μ be a prior probability distribution of the test inputs. For all $l \in C$, $Pr_{\mathcal{M}^a_{\mathcal{P}}}(reach(l)) = \Sigma_{v \in Val_V}\{Pr_{\mathcal{M}_{\mathcal{P}}}(reach((l, v)))\}$ where Val_V is the set of all possible valuations of V.* □

The correctness of the proposition can be established by showing the probability of reaching any l' is

$$\Sigma_{l \in C}(Pr_{\mathcal{M}^a_{\mathcal{P}}}(reach(l)) \times Pr_a(l, l'))$$

A test execution of $\mathcal{P}$ can be naturally mapped to a path of $\mathcal{M}^a_{\mathcal{P}}$. For instance, the test execution with input $x = y = 0$ given the program shown in Figure 1 is mapped to the path composed of state 1, 2, 3, 5, and 8. We say that a test execution covers a state of $\mathcal{M}^a_{\mathcal{P}}$ if it covers the corresponding control location of $\mathcal{P}$. Furthermore, a path in $\mathcal{M}^a_{\mathcal{P}}$ uniquely corresponds to a program path in $\mathcal{P}$.

3.2 Optimal Strategy

Recall that a concolic testing strategy is a sequence of choices among different test case generation methods. In this work, we define the space for the choice to be:

$$\{RT\} \cup \{SE(p) | p \in path(\mathcal{P})\}$$

where RT denotes random testing and $SE(p)$ denotes symbolic execution by solving the path condition associated with path p. To compare the cost of different choices, we need a way of measuring them. We focus on time cost in this work. Let *cost* be a function which, given $a \in \{RT\} \cup \{SE(p) | p \in path(\mathcal{P})\}$ returns its time cost. For simplicity, the time cost of generating a random test case is set to be 1 unit. The time cost of $SE(p)$ includes the time cost of encoding/solving the path condition.

We measure the effectiveness of a choice in terms of the probability of covering a set of states in $\mathcal{P}$. Given a choice $a \in \{RT\} \cup \{SE(p) | p \in path(\mathcal{P})\}$ and a set of states X of $M^a_{\mathcal{P}}$, we can compute the probability of covering exactly the set of states X with random testing as follows.

$$Pr(RT, X) = \Sigma_{\pi \in Path^{max}(\mathcal{M}^a_{\mathcal{P}}) \wedge states(\pi) = X} Pr(\pi) \quad (1)$$

For the example shown in Figure 1, $Pr(RT, \{1, 3\})$ is $1 - \frac{1}{2^{32}}$ and $Pr(RT, \{1, 4, 5\})$ is 0 since there is no test case which covers 1, 4, and 5 at the same time.

If the choice is symbolically executing program path p, i.e., $SE(p)$, we know that all states in the path p, written as $states(p)$, must be covered. Let $\Pi = \{\pi | s = last(p) \wedge \pi \in Path^{max}(s, \mathcal{M}^a_{\mathcal{P}}) \wedge states(\pi) \cup states(p) = X\}$ be the set of all maximal paths which start with the last state of path p and, together with p, cover all and only states in X. The probability of covering all and only states X with $SE(p)$, written as $Pr(SE(p), X)$, is defined as follows.

$$Pr(SE(p), X) = \begin{cases} 0 & \text{if } states(p) \nsubseteq X \\ 0 & \text{if } \Pi = \{\} \wedge states(p) \neq X \\ 1 & \text{if } \Pi = \{\} \wedge states(p) = X \\ \Sigma_{\pi \in \Pi} Pr(\pi) & \text{else} \end{cases} \quad (2)$$

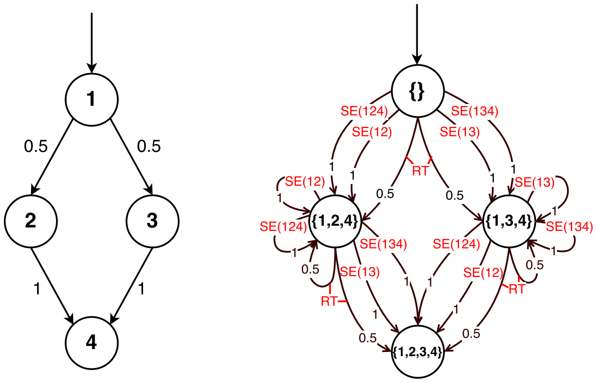

Figure 2: MDP with cost Model

For the example shown in Figure 1, $Pr(SE(13), \{1, 3, 5\})$ is $\frac{4}{5}$, i.e., by symbolically execute path 1 and 3, we have probability $\frac{4}{5}$ of covering state 1, 3, and 5. For another example, $Pr(SE(13), \{1, 5\})$ is 0 since we must cover 3.

In this work, we assume that the choice can be made depending on whether certain states have been covered or not. This makes sense intuitively since if all states along a path have been covered, it is a good idea not to apply symbolic execution to that path. A strategy is thus a function which takes as input information on whether each control location in $\mathcal{P}$ has been covered or not, and returns a choice of test case generation methods. To compare different concolic testing strategies systematically, we build the following model in the form of a Markov Decision Process (MDP) with Costs.

Definition 3.5. Let $\mathcal{M}^a_{\mathcal{P}} = (S, Pr, \mu)$ be the data-abstract DTMC interpretation of a program $\mathcal{P}$. We define $\mathcal{D}_{\mathcal{P}} = (Covered, Act, \phi, T, C)$ be an MDP with Costs such that

- $Covered \subseteq \mathbb{P}S$, where $\mathbb{P}S$ is the power set of S, i.e., each member of $\mathbb{P}S$ is a set of states in S (i.e., those which have been covered);
- $Act = \{RT\} \cup \{SE(p) | p \in path(\mathcal{P})\}$;
- $\phi \in Covered$ is the initial state which is $\emptyset$;
- T is defined such that $T(M, a)$ where $M \in Covered$ and $a \in Act$ is a probability distribution β defined as follows: $\beta(N) = \Sigma_{X:\mathbb{P}S.\ X \cup M = N} Pr(a, X)$ for all $N \in Covered$. $Pr(a, X)$ is defined by (1) and (2) above.
- C associates a cost for each $a \in Act$ as defined by function *cost*.

For instance, given the following simple program,

```
void myfunc2(int x) {
1.   if (x >= 0) {
2.       x++;}
3.     else {x--;}
4.   return x;
}
```

we can obtain the data-abstract Markov Chain model shown on the left of Figure 2, and the corresponding $\mathcal{D}_{\mathcal{P}}$ shown on the right. The initial state of $\mathcal{D}_{\mathcal{P}}$ is $\emptyset$, i.e., none of the states have been covered. Applying RT at the initial state, we have a distribution such that with probability 0.5 we reach state $\{1, 2, 4\}$ (i.e., state 1, 2, and 4 are covered) and with probability 0.5 we reach state $\{1, 3, 4\}$. If instead symbolic execution on path $\langle 1, 2\rangle$ is applied (i.e., $SE(12)$), we have probability 1 of reaching state $\{1, 2, 4\}$. Note that if we apply symbolic execution on path $\langle 1, 2\rangle$ at state $\{1, 2, 4\}$, we reach $\{1, 2, 4\}$ again with probability 1, which is represented by the self-looping transition at state $\{1, 2, 4\}$. Assuming that $cost(RT) = 1$, $cost(SE(12)) = cost(SE(13)) = 2$ and $cost(SE(124)) = cost(SE(134)) = 3$, we can then compute the expected cost of a concolic testing strategy based on the accumulated cost of each choice. For instance, the expected cost of always applying RT is 2, whereas the expected cost of applying $SE(12)$ and then $SE(13)$ is 4.

With Definition 3.5, we can see that a strategy for concolic testing is equivalent to a policy of $\mathcal{D}_{\mathcal{P}}$, i.e., a function from S' to Act. The following then answers *RQ1*.

> *Answer to RQ1: The optimal strategy is the policy of $\mathcal{D}_{\mathcal{P}}$ which has the minimum expected cost.*

For instance, in the example shown in Figure 2, the optimal strategy is the one which applies RT always (with an expected cost 2). The problem of finding the optimal strategy is thus reduced to the problem of finding the policy with the minimum expected cost, which can be solved using existing methods [27] like value iteration, policy iteration or solving a linear programming problem. The computational complexity of finding the optimal strategy is thus bounded by the complexity of identifying the optimal policy.

> *Answer to RQ2: The complexity of identifying the optimal strategy is strongly polynomial in the number of states in $\mathcal{D}_{\mathcal{P}}$, which in turn is exponential in the number of control locations in $\mathcal{P}$.*

3.3 Evaluating Existing Heuristics

In the following, we conduct experiments to answer RQ3 empirically. That is, we compare the performance of the optimal strategy with that of the heuristics-based ones [6, 7, 23, 33, 38, 42]. The goal is to see whether existing heuristics are reasonably effective.

We randomly generate a set of Markov Chain models (with no unreachable states) which we take as abstractions of programs. Due to the high complexity in computing the optimal strategy, we generate models containing 5 to 20 states only using the method in [46]. For every state, with probability 0.5, we generate a branch, i.e., the expected branch density is 0.5. We randomly generate a transition probability for each transition. To mimic low-probability program paths, we generate transitions of probability as low as 1e-4 with probability 0.8 for 5-states models (to avoid not having low-probability transitions) and 0.2 (to avoid not having too many low-probability transitions) for 10, 15, or 20-states models. In order to simplify the experiments, instead of associating a cost of symbolic execution for each path, we associate each transition in the model with a positive integer cost[4] within 1000. We construct the corresponding MDP with Cost models for each Markov Chain and use PRISM [32] to compute the optimal strategy.

The results are shown in Table 1, where first column shows the strategy and the rest shows the results obtained with 50 random 5-state Markov Chains, 50 random 10-state Markov Chains, etc. Row *optimal* is the expected cost of the optimal strategy, which has been normalized to 1. The rest of the rows are the result of random testing (RT), the four strategies in KLEE [7]: the default random-cover new (RCN), random state search (RSS), random path selection (RPS), and depth first search (DFS), the directed automated random testing in DART [22], generational search (GS) in SAGE [23], context guided search (CGS) in [42], and sub-path guided search (SGS) in [33]. The length of sub-path in SGS is set to be 20% of the the total number of states in the model. The last row is to be ignored for now.

We use Java to implement all approaches. For each Markov Chain model, we repeat each strategy 1000 times and obtain the mean cost (to cover all the states). Note that for random testing, it may take an extremely long time to cover all states, thus we set a limit of 1000000 (test cases). From the results, we observe that all existing heuristics result in significantly higher costs than the optimal cost. Even the best performance heuristics has a cost which is one order of magnitude higher than the optimal one. Among all strategies, the strategy which adopts random testing every time performs the worst when there are 20 states. The results show that existing heuristics have much room to improve. Note that the results show in Table 1 should be taken with a grain of salt since they are based randomly generated Markov Chain models, which may not be representative of real programs.

	5 states	10 states	15 states	20 states
Optimal	1	1	1	1
RT	138.9	11.3	44.2	114.7
RCN	1.7	14.4	15.1	12.7
RSS	12.8	50.7	64.0	68.1
RPS	12.8	50.6	63.9	68.5
DFS	7.1	27.4	21.8	18.6
DART	1.8	13.0	12.8	13.0
GS	1.9	13.5	13.9	13.3
CGS	1.8	12.6	13.6	13.8
SGS	11.2	32.4	29.4	25.5
G	2.1	4.8	3.1	4.8

Table 1: Simulated experiments

Answer to RQ3: Existing heuristics could be improved.

4 APPROXIMATING OPTIMALITY

Based on the discussion in Section 3, it is clear that identifying the optimal strategy in practice is infeasible due to its high complexity, as well as difficulties in identify the probability of program paths and the cost of symbolic execution. In the following, we propose a method to approximate the optimal strategy in practice. Our proposal includes a way of approximating $\mathcal{M}^a_{\mathcal{P}}$, a way of approximating function *cost*, and a greedy algorithm for identifying optimal policy.

4.1 Estimating $\mathcal{M}^a_{\mathcal{P}}$ and Function *cost*

In the following, we present an approach to estimate $\mathcal{M}^a_{\mathcal{P}} = (S, Pr, \mu)$. Note that this is the subject of a recent line of research known as probabilistic symbolic execution [15, 18, 21]. However, probabilistic symbolic execution has a high complexity (due to the underlying model counting techniques [9]). We thus apply a lightweight approach, i.e., we estimate Pr based on the test cases which have been obtained. The essential problem that we would like to address is: if we have observed certain events (i.e., test cases which cover certain program paths), how do we estimate the probability of the seen events and those unseen events (i.e., test cases which cover other program paths)? This problem has been studied for decades and a number of methods have been proposed, e.g., the Laplace estimation [13] and Good-Turing estimation [19]. We refer the readers to [19] for comprehensive discussion on when different estimations are effective. In the following, we show how to estimate $\mathcal{M}^a_{\mathcal{P}}$ based on the Laplace estimation.

Assume that we have obtained a set of test executions X, we can estimate Pr as follows.

Definition 4.1. Given any state $s \in S$, let $\#s$ be the number of times state s is visited by samples in X. For any $t \in S$, let $\#(s,t)$ be the number of one-step transition from state s to t in X. For any state s, if it is impossible for s to reach another control location t in $\mathcal{P}$, we set $Pr(s,t)$ to be 0; otherwise, the Laplace estimation sets $Pr(s,t)$ to be $\frac{\#(s,t)+1}{\#s+n}$, where n is the total number of states s can reach with one step.

Intuitively, if a transition (i.e., a control flow) from state s to t is not observed in X because $Pr(s,t)$ is small, the Laplace estimation sets the transition probability to be $\frac{1}{\#s+n}$. It is easy to see that the estimated Pr converges to the actual Pr with an unbounded number of samples. In the following, we write *estimate*$(\mathcal{P}, X)$ to denote the estimated $\mathcal{M}^a_{\mathcal{P}}$.

Estimating function *cost*, i.e., the cost of constraint solving, is highly nontrivial due to the sophisticated constraint solving techniques adopted by constraint solvers like Z3 [14]. It is itself a research topic [29, 31]. In this work, we adopt the approach in [29], which works as follows. Firstly, the authors of [29] collected the time costs of solving constraints generated from analyzing a set of real-world programs through symbolic execution. Assuming the cost of constraint solving is the weighted sum of the primitive operations (e.g., the `Add` and `Mul` operation) in the constraint, they then estimate the weight of each primitive operation type through function fitting. Afterwards, given a constraint c, its solving cost is

[4] This effectively assumes solving a constraint ϕ takes less time than solving $\phi \wedge \alpha$, which may not be always true.

estimated as the weighted sum of all primitive operations in c. For example, if c is $a * b > 0$, its solving cost is the sum of weighted cost of multiplication and that of the greater-than comparison. We refer the readers to [29] for details.

4.2 A Greedy Algorithm

Even with a reasonable approximation of $\mathcal{M}^a_\mathcal{P}$ and function *cost*, the algorithm for identifying the optimal strategy remains overly complicated (refer to the answer to RQ2). In the following, we present a greedy algorithm with much lower complexity. The idea is to estimate $\mathcal{M}^a_\mathcal{P}$ on-the-fly and apply a test case generation method which improves test coverage in the most cost-effective way locally based on the estimation.

The details are shown in Algorithm 1. At line 1, we start with an empty set of test cases. At line 2, we initialize a set *toIgnore* for storing paths which are to be ignored for symbolic execution. The loop from line 3 to 14 iteratively generates test cases until the coverage criteria is achieved. During each iteration, we first construct an estimation of $\mathcal{M}^a_\mathcal{P}$ at line 4. Afterwards, we call function *localOptimal* to choose the local-optimal test generation method. If the choice is random testing, we generate a random test case at line 7; otherwise, we apply symbolic execution to the selected program path. If the selected path is infeasible or solving the path condition times out, we add the path into *toIgnore*.

Function *localOptimal*$(\mathcal{M}, X, toIgnore)$ is shown in Algorithm 2. Intuitively, we define the "reward" of a test generation method to be the number of uncovered states which is expected to be covered with the newly generated test case and select the method with the largest expected reward per unit of cost. At line 2, we first compute the expected reward of random testing based on the current estimation $\mathcal{M} = (S, Pr, \mu)$. It is computed by extending $\mathcal{M}$ with reward (i.e., 1 unit reward is associated with one unvisited state) and solving the problem of expected reward using existing methods [2]. In the following, we show how it can be solved by solving an equation system.

Let R_s where $s \in S$ be the reward of visiting s. We build an equation system as follows.

$$R_s = \begin{cases} 1 + \Sigma_{t \in S}\{Pr(s,t) \times R_t\} & \text{if } s \notin \textit{visited} \\ \Sigma_{t \in S}\{Pr(s,t) \times R_t\} & \text{if } s \in \textit{visited} \end{cases}$$

The expected reward of random testing is then: $\Sigma_{s \in S}\{\mu(s) \times R_s\}$. Note that we associate one reward for visiting each unvisited state since our goal is to cover every state.

Next, we compare the expected reward of random testing to that of symbolic execution. Ideally, we would compute the cost of symbolically executing every path as well as the corresponding reward, and then choose the most profitable one. However, the number of such paths is often huge (i.e., infinite if there are loops). Thus, we heuristically focus on paths which contain no uncovered states except the ending state. This way, it is guaranteed to visit at least 1 unvisited state if symbolic execution is applied. Note that similar to [12, 22], we assume that a bound on the number of iterations for any loop is provided and we only consider paths with fewer iterations. The expected reward of applying symbolic execution to the path ending with state s is denoted as R_s, which can be obtained using the same equation system discussed above.

Algorithm 1: *greedy*$(\mathcal{P}, \mu)$ where $\mathcal{P}$ is a program and μ a prior distribution on test inputs

```
1  let $X$ be an empty set of test cases;
2  let $toIgnore$ be an empty set of paths;
3  while there is an unvisited control location do
4      let $\mathcal{M}$ be $estimate(\mathcal{P}, X)$;
5      let $a := localOptimal(\mathcal{M}, X, toIgnore)$;
6      if $a$ is random testing then
7          randomly generate a test case $t$ according to $\mu$;
8          add $t$ into $X$;
9      if $a$ is $SE(p)$ then
10         solve $PC_p$ to generate a test case $t$;
11         if $p$ is unsatisfiable or solving $PC_p$ times out
           then
12             add $p$ into $toIgnore$;
13         else
14             add $t$ into $X$;
15 return $X$;
```

Algorithm 2: *localOptimal*$(\mathcal{M}^a_\mathcal{P}, X, toIgnore)$

```
1  let $visited$ be the set of visited states given $X$;
2  let $reward$ be the expected reward of random testing;
3  let $toReturn$ be $random\ testing$;
4  for all path $\pi$ s.t. the only uncovered state is $last(\pi)$ do
5      if $\pi \notin toIgnore$ then
6          let $reward_\pi$ be the expected reward of solving $\pi$;
7          if $reward_\pi / cost(\pi) > reward$ then
8              $toReturn := SE(\pi)$; $reward :=$
               $reward_\pi / cost(\pi)$;
9  return $toReturn$;
```

The details are shown in Algorithm 2 (line 4 to 8). At line 5, we check if the selected path π is to be ignored. If it is not, we compute the expected reward of solving π, by solving the same equation system to obtain R_s where s is the ending state of π. Intuitively, this is because by solving the path π, we have probability one of visiting s and obtaining all of its expected reward. That is, if $last(\pi)$ is s, $reward_\pi$ is R_s. At line 7, we compare the reward per unit cost (where function *cost* is approximated as discussed in Section 4.1) of $SE(\pi)$ and the current best choice, and keep the better one. Note that we assume the path condition is precise. If a test input generated by solving the path condition diverges and thus not reach s, we add the path to *toIgnore* as well.

In the following, we illustrate how the algorithm works for the program shown in Figure 1. For illustration purpose, we

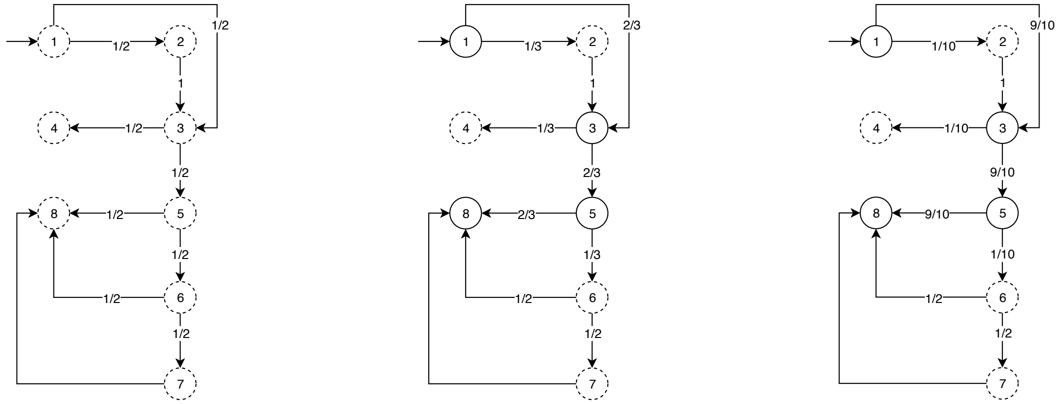

Figure 3: Abstraction

assume that solving a linear (in)equality or their conjunctions has time cost 4, solving a non-linear (in)equality has cost of 10; and solving a boolean combination of non-linear (in)equalities has cost of 50. Initially, since we have no test executions yet, the estimation $\mathcal{M}$ is shown at the left of Figure 3 where uncovered states are dash-lined. Note that all outgoing transitions from the same state have the same probability. Based on this estimation, we compute that the expected reward of random testing, by solving the following equation systems.

$$
\begin{aligned}
&R_1 = 1 + 0.5 * R_2 + 0.5 * R_3\\
&R_2 = 1 + R_3\\
&R_3 = 1 + 0.5 * R_4 + 0.5 * R_5\\
&R_5 = 1 + 0.5 * R_8 + 0.5 * R_6\\
&R_6 = 1 + 0.5 * R_8 + 0.5 * R_7\\
&R_7 = 1 + R_8\\
&R_4 = R_8 = 1
\end{aligned}
$$

The expected reward of random testing is $\mu(s_1) * R_1$ which is 4.875. Since all states are unvisited, the candidate path we select for symbolic execution is the one containing state 1 only. Since the path condition is true, applying symbolic execution to this path is the same as random testing. Note that this implies that we always start with random testing. Assume that the random test case we generate covers control location 1, 3, 5 and 8. The estimation is then updated, as shown on the middle of Figure 3. Next, we compute the expected reward of random testing by solving the following equation systems.

$$
\begin{aligned}
&R_1 = \frac{1}{3} * R_2 + \frac{2}{3} * R_3\\
&R_2 = 1 + R_3\\
&R_3 = \frac{1}{3} * R_4 + \frac{2}{3} * R_5\\
&R_4 = 1\\
&R_5 = \frac{2}{3} * R_8 + \frac{1}{3} * R_6\\
&R_6 = 1 + 0.5 * R_8 + 0.5 * R_7\\
&R_7 = 1 + R_8\\
&R_8 = 0
\end{aligned}
$$

We have $R_1 = 1$, $R_2 = \frac{5}{3}$, $R_4 = 1$ and $R_6 = 1.5$. The candidate paths for symbolic execution include the path from 1 to 2, the path from 1, 3, to 4, and the path from 1, 3, 5, to 6. The costs are 4, 50, and 50 respectively. The expected rewards are $\frac{5}{3}$, 1, and 1.5 respectively. Thus, the chosen method is random testing. For simplicity, assume that the first 8 random test executions all cover 1, 3, 5, and 8. As a result, $\mathcal{M}$ is updated as shown on the right of Figure 3. The expected reward of random testing is computed as 0.335, whereas the expected reward of solving the path from 1 to 2 is 1.235. We thus conclude that the latter is more cost-effective (with a reward per cost 0.30875) and thus apply symbolic execution to the path. Intuitively, we switch from random testing to solving certain program path only when covering the path requires a large number of random test cases, which would cost more than that of symbolic execution.

The complexity of Algorithm 1 is reasonable. In order to choose the right test case generation method, during each round, we pay the price of solving an equation system whose number of variables equals to the number of control locations in the program. Modern equation system solvers are often rather efficient and the overhead is reasonable. We can further optimize the algorithm for solving the equation system since R_s changes after one iteration only if s can reach a state which has been newly covered.

5 EVALUATION

To answer RQ4, we first compare the performance of our greedy algorithm against the optimal strategy using the randomly generated Markov models (as in Section 3.3). That is, we run the greedy algorithm, assuming that we know the cost of constraint solving but not the transition probability, and measure its performance. In other words, the transition probability is estimated on-the-fly as shown in Algorithm 1. The results are shown in the last row of Table 1. It can be observed that compared with the existing heuristics, the greedy algorithm offers better performance.

Answer to RQ4: It is possible to design a practical algorithm to approximate the optimal strategy.

So far we have experiment with different strategies on abstract models. In order to answer $RQ4$ and $RQ5$ based on real-world programs, we implement our approach based on KLEE [7]. Note that KLEE is a symbolic execution engine, i.e., it only maintains the symbolic values of symbolic variables. We thus first extend KLEE with a concolic execution engine which maintains both the symbolic and concrete values of each variable. As a result, we are able to switch between random testing and symbolic execution at runtime according to the greedy algorithm (if necessary). During (symbolic or concrete) execution, when a branching statement is encountered, we fork a state which corresponds to the un-selected branch, without considering its feasibility.

To estimate $\mathcal{M}^a_{\mathcal{P}}$, we first construct the inter-procedural control graph (ICFG), whose nodes are the states of $\mathcal{M}^a_{\mathcal{P}}$. The transition probability is then estimated on-the-fly as in Algorithm 1. Recall that we need to solve an equation system to select the local optimal test generation method. In our implementation, we use Eigen [16] to solve the equation system. When the choice is symbolic execution, the path constraint of the symbolic state is solved to check the state's feasibility. If the state is not feasible, the path is marked infeasible. If the choice is random testing, we generate a random test case with concrete values.

Experimental Setup To evaluate the effectiveness of different concolic testing strategies, we need a set of programs which contain complicated path conditions (so that constraint solving the path conditions takes a non-trivial amount of time) as well as non-trivial control flow (so that different strategies may choose different test generation methods or paths). We use the programs in GNU Scientific Library (GSL) [25]. Functions in GSL often have both complex arithmetic operations and complex control flow. GSL has been previously analyzed using KLEE [29, 39]. We rank the functions in GSL using the code coverage which was achieved in the previous study for each function and choose the functions with the lowest coverage. Table 2 lists the functions used in our experiment, where the second column shows the number of nodes in the ICFG of the function (i.e., the number of basic blocks in the function or invoked functions). Note that we filter functions which have complex input types such as function pointers, complex arrays and complex struts. To analyze these programs, we have implement the method in [40] for analyzing floating point programs. The basic idea is to convert floating point operations to the integer simulation functions. We use softfloat [43] as the library in our implementation. Furthermore, under-constrained symbolic execution [17] is implemented for analyzing arbitrary functions.

Table 2: Programs in the experiments

Name	#N	Function in GSL
FG0_ser	1822	coulomb_FG0_series
FGmhalf	1819	coulomb_FGmhalf_series
dilog_xge0	1167	dilog_xge0
Chi_e	1233	gsl_sf_Chi_e
bessel_Inu	1249	gsl_sf_bessel_Inu_scaled_asymp_unif_e
bessel_JY	2175	gsl_sf_bessel_JY_mu_restricted
bessel_Knu	1249	gsl_sf_bessel_Knu_scaled_asymp_unif_e
bessel_cos	492	gsl_sf_bessel_cos_pi4_e
bessel_sin	492	gsl_sf_bessel_sin_pi4_e
coupling	1179	gsl_sf_coupling_6j_e
elljac_e	1524	gsl_sf_elljac_e
exprel_n_e	1769	gsl_sf_exprel_n_e
hyperg_U	1719	gsl_sf_hyperg_U_large_b_e
lngamma_e	1677	gsl_sf_lngamma_e
lnpoch_sgn	1925	gsl_sf_lnpoch_sgn_e
lnpoch_pos	1836	lnpoch_pos
cyc	1116	solve_cyc_tridiag
cyc_non	1141	solve_cyc_tridiag_nonsym
tri	1104	solve_tridiag
tri_non	1092	solve_tridiag_nonsym

For baseline comparison, we compare our approach with the four search heuristics supported in KLEE as explained in Section 3.3. Notice that tools implementing other searching strategies, e.g., [23, 33, 42], are either not maintained or target different programming languages. Each function is analyzed using 6 strategies and we measure the *instruction coverage* achieved with different timeouts. Our experiments were conducted on a server having 64GB RAM and 3.2GHz XEON CPUs with 16 cores. The timeout for each constraint solving is 10 seconds. The cost of constraint solving is estimated using the formula in [29]. Each experiment is repeated 3 times and we report the average as the result. Our implementation and programs are available at [30].

Evaluation Results Table 3 shows the coverage achieved for each function with a timeout of 5 minutes, 15 minutes and 30 minutes respectively, where column **G** is the result of our greedy algorithm and R is the always random testing strategy. The winner for each setting is highlighted in bold. Note that due to randomness, it is not always guaranteed that better coverage will be achieved with more time. It can be observed that our greedy algorithm not only achieves much better coverage but also achieves it much faster. For instance, after 5 minutes, our greedy algorithm covers 72.6% of the instructions whereas all KLEE strategies cover less than 30%. After 30 minutes, our coverage is 76.9% whereas KLEE strategies are less than 37%. Note that random testing achieves better performance than KLEE strategies as well (although worse than our strategy). This result suggests that existing concolic testing strategies should better-integrate random testing and our strategy offers an effective way to achieve that.

Figure 4 visualizes the trend of coverage over time for each strategy. It can be observed that within one minute, our greedy strategy is able to achieve a much higher coverage than those in KLEE. This is because we are able to strategically choose the "most" rewarding method each time and cover those easier-to-cover instructions quickly. Afterwards, our strategy slowly gains more coverage by solving path

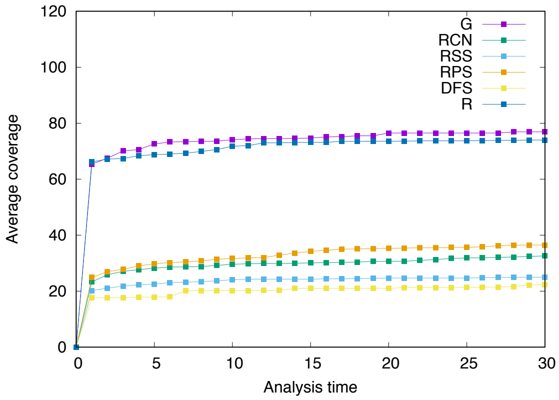

Figure 4: Average coverage w.r.t. analysis time.

Programs	5 Minutes		15 Minutes		30 Minutes	
	SE	RT	SE	RT	SE	RT
`FGO_ser`	0	3	3	2	3	4
`FGmhalf_series`	9	2	16	1	4	1
`dilog_xge0`	145	22	214	27	260	20
`Chi_e`	89	1	77	25	49	33
`bessel_Inu`	28	3	52	10	47	30
`bessel_JY_mu`	19	1	0	7	40	7
`bessel_Knu`	22	9	62	23	76	37
`bessel_cos`	23	3	57	7	51	5
`bessel_sin`	23	3	57	6	42	5
`coupling_6j`	23	34722	23	103926	23	137164
`elljac_e`	64	47	50	55	143	44
`exprel_n_e`	36	2	84	1	302	3
`hyperg_U`	9	19	12	28	9	11
`lngamma_e`	5	111	6	272	11	155
`lnpoch_sgn`	42	2	133	6	249	7
`lnpoch_pos`	24	6	54	16	52	17
`cyc_tridiag`	0	7676	0	20989	0	32337
`cyc_tridiag_non`	0	8203	0	21204	0	31134
`tridiag`	0	7731	0	20901	0	30088
`tridiag_non`	0	7672	0	20553	0	30281

Table 4: Number of times of RT and SE

conditions which are hard to solve or generating random test cases. In general, other strategies gain slightly more coverage after the first minute too, although not always. In a few cases, e.g., `dilog_xge0`, a boost in the coverage is observed with the RPS strategy at 15 minutes timeout. This is likely because the strategy switches to solving a different path, which leads to many uncovered instructions.

In order to get a view on the choice of test case generation methods by our strategy, we summarize in Table 4 the number of times random testing and symbolic execution are applied. Note that each entry in the table corresponds to a different run with different timeout and thus it is not guaranteed that the numbers in the 30 minutes' column are the largest. It can be observed that our strategy does not always favor one method over another. Furthermore, the ratio between the number of times random testing is chosen and that of symbolic execution varies significantly from function to function. For instance, for the last four functions, due to the complex path conditions, random testing is consistently the choice, which turns out to be effective in achieving high coverage. Function `exprel_n_e` shows another extreme, i.e., symbolic execution is often the choice due to its simple constraints. This suggests that our strategy adapts to different functions.

Answer to RQ5: Our greedy algorithm outperforms existing heuristics in KLEE.

Threats to Validity. Our test subjects are all numeric functions and thus the results could be biased. We plan to apply the proposed approach to other programs (e.g., programs operating on non-trivial data structures) to further validate its effectiveness in general.

6 RELATED WORK

This work is closely related to many existing searching strategies for concolic testing. Besides those mentioned in previous sections, there are other search heuristics. In [14], Liu *et al.* proposed to empirically predict the cost of solving a path constraint and prioritizes those paths with smaller solving cost. In [38], Park *et al.* proposed the CarFast strategy, which always selects a branch whose opposite branch is not yet covered, and has the highest number of statements control-dependent on that branch. Xie *et al.* [48] introduced a fitness guided path exploration technique, which calculates fitness values of execution paths and branches to guide the next execution towards a specific branch. The fitness function measures how close a discovered path is to a not-yet-covered branch. Marinescu *et al.* [36] guides symbolic execution towards the software patches. It exploits a provided test suite to identify a good test case and uses symbolic execution with several heuristics to generate more related inputs to test the patches. In [42], Seo *et al.* proposed the context-guided search strategy which selects a branch under a new context (i.e., a local sequence of branch choices) for the next input generation. In [8], Cadar *et al.* applies a Best-First Search strategy, which checks all execution states and forces symbolic execution towards dangerous operations (e.g., a pointer de-reference). Compared with the above-mentioned approaches, ours is the first one to formally define what is the optimal strategy and subsequently develop a practical algorithm. We provide a framework for systematically comparing the effectiveness of random testing and symbolic execution.

This work is related to work on combining random testing and symbolic execution. Besides [3, 4] which have been discussed in Section 1, Kong *et al.* [31] discussed different strategies on combining random testing and symbolic execution in the setting of verifying hybrid automata. They too make use of transition probability and cost in choosing where to apply symbolic execution. However, their approach remains a heuristics (i.e., choosing a branch with low cost,

Programs	5 Minutes (%)						15 Minutes (%)						30 Minutes (%)					
	G	RCN	RSS	RPS	DFS	R	**G**	RCN	RSS	RPS	DFS	R	**G**	RCN	RSS	RPS	DFS	R
FG0_ser	**96.8**	14.8	14.8	14.8	14.8	94.7	**96.8**	14.8	14.8	14.8	14.8	**96.8**	94.7	14.8	14.8	14.8	14.8	**98.8**
FGmhalf	95.1	19.1	19.1	19.1	13.8	**96.9**	**95.1**	19.1	19.1	19.1	13.8	**96.9**	**98.7**	19.1	19.1	19.1	13.8	**98.7**
dilog_xge0	**36.0**	25.3	25.0	33.7	16.6	35.9	36.0	25.3	25.3	**69.5**	17.7	41.8	35.5	27.2	25.3	**72.5**	17.7	42.4
Chi_e	**70.7**	12.0	12.0	12.0	15.2	**70.7**	**70.7**	12.0	12.0	12.0	15.2	**70.7**	**70.7**	12.0	12.0	12.0	15.2	**70.7**
bessel_Inu	**95.6**	29.1	14.3	29.1	10.3	74.4	**95.6**	29.1	14.8	29.1	11.8	**95.6**	**95.6**	29.1	14.8	29.1	11.8	**95.6**
bessel_JY	**36.2**	30.5	30.5	30.5	28.7	27.9	**31.0**	30.5	30.5	30.5	28.7	29.5	**43.7**	30.5	30.5	30.5	28.7	38.4
bessel_Knu	**95.5**	29.2	14.4	29.2	10.4	74.2	**95.5**	29.2	14.9	29.2	11.9	**95.5**	**95.5**	29.2	14.9	29.2	11.9	**95.5**
bessel_cos	**89.9**	43.4	23.9	50.4	14.2	85.5	**92.0**	53.1	29.2	59.3	14.2	88.8	**89.9**	62.5	31.3	72.6	14.2	88.2
bessel_sin	**89.9**	44.9	23.9	50.4	14.2	84.3	**89.9**	53.1	29.2	59.3	14.2	**89.9**	92.0	62.5	31.3	72.6	14.2	**94.7**
coupling	**95.9**	20.0	20.0	20.0	17.3	95.5	**95.9**	22.8	22.8	22.8	22.8	**95.9**	**95.9**	22.8	22.8	22.8	22.8	**95.9**
elljac_e	13.1	**20.4**	**20.4**	**20.4**	11.1	12.5	**36.3**	20.4	20.4	20.4	11.1	20.2	**28.6**	20.4	20.4	20.4	11.1	13.1
exprel_n_e	**32.9**	24.5	23.9	25.8	19.1	25.5	33.9	28.0	25.8	**34.4**	19.1	27.7	36.8	33.8	27.4	**39.5**	25.0	32.6
hyperg_U	**38.6**	18.4	17.7	18.7	15.9	26.6	**53.8**	18.7	18.4	18.7	15.9	37.9	**54.0**	18.7	18.6	18.7	15.9	30.5
lngamma_e	**60.8**	38.4	34.3	38.4	24.9	28.1	**57.6**	38.4	37.9	41.8	24.9	48.0	**57.6**	47.5	38.4	47.5	40.1	50.1
lnpoch_sgn	**51.0**	31.6	22.0	32.8	14.4	43.4	39.6	31.9	24.7	33.1	14.4	**64.2**	**70.1**	33.4	31.5	34.0	14.4	56.1
lnpoch_pos	**75.3**	16.7	14.5	21.2	14.8	54.7	**78.6**	24.4	15.0	28.4	14.8	75.2	**99.1**	27.7	15.0	28.9	14.8	**99.1**
cyc	**93.4**	25.8	23.1	26.0	14.7	**93.4**	**93.4**	27.5	23.3	27.5	25.5	**93.4**	**93.4**	27.5	23.3	28.8	31.7	**93.4**
cyc_non	**94.2**	19.8	19.4	21.4	30.4	**94.2**	**94.2**	22.4	19.6	34.7	30.4	**94.2**	**94.2**	31.5	19.6	36.0	30.4	**94.2**
tri	**96.4**	42.6	33.3	42.7	22.1	**96.4**	**96.4**	42.7	37.9	42.7	41.5	**96.4**	**96.4**	42.7	37.9	42.7	41.5	**96.4**
tri_non	**94.8**	55.4	43.8	58.4	35.6	**94.8**	**94.8**	58.4	51.1	58.4	56.7	**94.8**	**94.8**	58.4	51.5	58.4	56.7	**94.8**
Average	**72.6**	28.1	22.5	29.7	17.9	65.5	**73.9**	30.1	24.3	34.3	21.0	72.7	**76.9**	32.6	25.0	36.5	22.3	74.0

Table 3: Coverage Results

similar to the approach in [14]) as there is no definition of the optimal strategy. Hybrid concolic testing [35] combines random testing and concolic testing. The idea is to start with random testing to quickly reach a deep state of the program by executing a large number of random test cases. When the random testing stops improving coverage for a while, it switches to concolic testing to exhaustively search the state space from the current program state. Garg et al. [20] proposed to combine feedback-directed unit test generation with concolic testing. They start with random unit testing similar to Randoop [37] and switches to concolic testing when the unit testing reaches a coverage plateau. A similar idea was proposed in [49]. Compared to the above-mentioned approaches, our method formally analyzes the effectiveness of random testing and symbolic execution and allows us to choose the more effective in every iteration.

This work is remotely related to work on reducing the cost of symbolic execution and concolic testing, through methods like pruning paths [1, 5, 10, 24, 28] and parallelism [44].

7 CONCLUSION

In this work, we propose a framework to derive optimal concolic testing strategies, based on which we analyze existing heuristics and propose a new algorithm to approximate the optimal strategy. The evaluation on randomly generated models and a set of real-world C programs shows that our algorithm outperforms most existing heuristic-based algorithms often.

For future work, we would like to investigate alternative ways of estimating probability and solving cost of program paths. Furthermore, we would like to extend our framework to other test case generation methods.

ACKNOWLEDGEMENT

This research was supported by Singapore Ministry of Education grant MOE2016-T2-2-123 and the National Basic Research Program of China (the 973 Program) under grant 2015CB352201, NSFC Program (No. 61572426). The third author is supported by NSFC Program (61472440, 61632015 and 61690203).

REFERENCES

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*, pages 367–381, 2008.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[3] Marcel Böhme and Soumya Paul. On the efficiency of automated testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, pages 632–642, 2014.

[4] Marcel Böhme and Soumya Paul. A probabilistic analysis of the efficiency of automated software testing. *IEEE Trans. Software Eng.*, 42(4):345–360, 2016.

[5] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*, pages 351–366, 2008.

[6] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering ASE*, pages 443–446, 2008.

[7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, pages 209–224, 2008.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, 2008.

[9] Supratik Chakraborty, Dror Fried, Kuldeep S. Meel, and Moshe Y. Vardi. From weighted to unweighted model counting. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI*, pages 689–695, 2015.

[10] Ting Chen, Xiaosong Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Comp. Syst.*, 29(7):1758–1773, 2013.

[11] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*,

8(2):244–263, 1986.

[12] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[13] G Cochran. Laplace s ratio estimator. *Contributions to survey sampling and applied statistics*, pages 3–10, 1978.

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*, pages 337–340, 2008.

[15] Matthew B. Dwyer, Antonio Filieri, Jaco Geldenhuys, Mitchell J. Gerrard, Corina S. Pasareanu, and Willem Visser. Probabilistic program analysis. In *Grand Timely Topics in Software Engineering - International Summer School GTTSE*, pages 1–25, 2015.

[16] Eigen 3.3.4. Eigen Website. http://eigen.tuxfamily.org/.

[17] Dawson R. Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 1–4. ACM, 2007.

[18] Antonio Filieri, Marcelo F. Frias, Corina S. Pasareanu, and Willem Visser. Model counting for complex data structures. In *Model Checking Software - 22nd International Symposium, SPIN*, pages 222–241, 2015.

[19] William A Gale and Geoffrey Sampson. Good-turing frequency estimation without tears*. *Journal of Quantitative Linguistics*, 2(3):217–237, 1995.

[20] Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *35th International Conference on Software Engineering, ICSE*, pages 132–141, 2013.

[21] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 166–176, 2012.

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[23] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2008.

[24] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 43–56, 2010.

[25] GSL 2.1. GNU Scientific Library (GSL). http://www.gnu.org/software/gsl/.

[26] Richard G. Hamlet. Testing programs with finite sets of data. *Comput. J.*, 20(3):232–237, 1977.

[27] Ronald A. Howard. The M.I.T. Press, 1960.

[28] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. Boosting concolic testing via interpolation. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, pages 48–58, 2013.

[29] Liu Jingde, Chen Zhenbang, and Wang Ji. Solving cost prediction based search in symbolic execution. *Journal of Computer Research and Development*, pages 1086,1094, 2016.

[30] Sun Jun. http://sav.sutd.edu.sg/research/smartconcolic.

[31] Pingfan Kong, Yi Li, Xiaohong Chen, Jun Sun, Meng Sun, and Jingyi Wang. Towards concolic testing for hybrid systems. In *FM 2016: Formal Methods - 21st International Symposium*, pages 460–478, 2016.

[32] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *Computer performance evaluation: modelling techniques and tools*, pages 200–204. Springer, 2002.

[33] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA*, pages 19–32, 2013.

[34] Kasper Søe Luckow, Marko Dimjasevic, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamaric, and Vishwanath Raman. Jdart: A dynamic symbolic analysis framework. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*, pages 442–459, 2016.

[35] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE*, pages 416–426, 2007.

[36] Paul Dan Marinescu and Cristian Cadar. High-coverage symbolic patch testing. In *Model Checking Software - 19th International Workshop, SPIN*, pages 7–21, 2012.

[37] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 815–816, 2007.

[38] Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. Carfast: achieving higher statement coverage faster. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 35, 2012.

[39] Minghui Quan. Hotspot symbolic execution of floating-point programs. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, pages 1112–1114, 2016.

[40] Anthony Romano. Practical floating-point tests with integer code. In *Proc. International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2014*, pages 337–356. Springer, 2014.

[41] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[42] Hyunmin Seo and Sunghun Kim. How we get there: a context-guided search strategy in concolic testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, pages 413–424, 2014.

[43] SoftFloat 2b. Berkeley SoftFloat. http://www.jhauser.us/arithmetic/SoftFloat.html.

[44] Matt Staats and Corina S. Pasareanu. Parallel symbolic execution for structural test generation. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA*, pages 183–194, 2010.

[45] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2016.

[46] Deian Tabakov and Moshe Y. Vardi. Experimental evaluation of classical automata constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR*, pages 396–411, 2005.

[47] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *Tests and Proofs, Second International Conference, TAP*, pages 134–153, 2008.

[48] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, pages 359–368, 2009.

[49] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 160–170, 2014.