WBS: Weighted Backtracking Strategy for Symbolic Testing of Embedded Software

Varsha P Suresh[1], Sujit Kumar Chakrabarti[1], Athul Suresh[1], and Raoul Jetley[2]

[1]International Institute of Information Technology Bangalore, Bengaluru, India
[2]ABB Corporate Research, Bengaluru, India

Abstract

Symbolic execution is an important program analysis technique that has found a number of applications in the last fifteen years or so. Popular symbolic execution approaches use backtracking when faced with infeasibility along a path being explored. A simple backtracking strategy (i.e. backtracking by a single decision node) may suffice when the goal is to cover the entire control flow graph (CFG). However, if the goal is to cover specific parts of the CFG through a single path, simple backtracking may lead to non-optimality or even non-termination. In this paper, we present weighted backtracking strategy (WBS) that exploits previous knowledge about the program behaviour to compute 'good' candidates as destinations of backtracking. We have integrated our heuristic to SymTest, a symbolic testing framework for embedded systems. Experiments with case-studies have demonstrated that WBS improves SymTest's performance both in its ability to achieve termination as well as in computing shorter test sequences compared to the original approach. SymTest with WBS generates shorter test sequences compared to several other existing test generation approaches based on symbolic execution.

1 Introduction

Large amount of time and effort of software development is spend in testing of the software. In automated test case generation the software under test is analysed and then test data are generated. The generated test data is used for test execution. Test execution can be simulation of the software or target testing, where the software is executed in the deployment environment. In case of real time embedded software the cost of testing goes up dramatically because such reactive systems requires interaction with other physical subsystems, human interfaces etc. Thus cost of testing has a direct impact on cost of software quality, which necessitates the optimization of software testing. In embedded systems and systems involving interactions with other physical subsystems, the *test sequence length* (defined in Section 2) plays a vital role in the cost of testing. As the test sequence length increases the interaction between physical subsystems increases which in turn increases the cost of testing. Even though there has been significant advancement in symbolic execution in recent years, there has not been much effort put towards obtaining short test sequences.

SymTest[1] is a symbolic execution framework used to generate short test sequences for embedded software. The main idea in SymTest is to explore only one path that covers the target edges. If the first path computed is feasible, SymTest indeed provides very good results in terms of the length of the test sequence (path) generated. However, if the path chosen is infeasible, the guarantee of the shortness of the generated test sequence is severely curtailed. Backtracking by one decision is the backtracking approach of SymTest which provides no assurance that the alternative path thus explored even terminates. We present a heuristic, *Weighted Backtracking Strategy* (WBS) which computes a candidate backtracking destination which may be more than one decision edge behind the last decision edge. This approach is based on weight calculations obtained from earlier runs of the system. We have implemented WBS as a part of SymTest and tested it against a number of case studies which reflects the path exploration problem. WBS aided in generating test sequences in multiple cases where SymTest fails to terminate, and generates shorter test sequences compared to SymTest and several other existing testing approaches based on symbolic execution.

The rest of this paper is organized as follows: A motivating example and an overview of the related works is discussed in Section 2. Section 3 explains the WBS and the integration of WBS in SymTest. Section 4 presents the experiment details and the obtained results. Section 5 concludes the paper.

2 Motivation and Related Works

In structural testing, the goal is to execute the system so as to cover certain parts of the system. In first-time testing, we may want to cover the entire system. But in regression

DOI reference number: 10.18293/SEKE2022-147

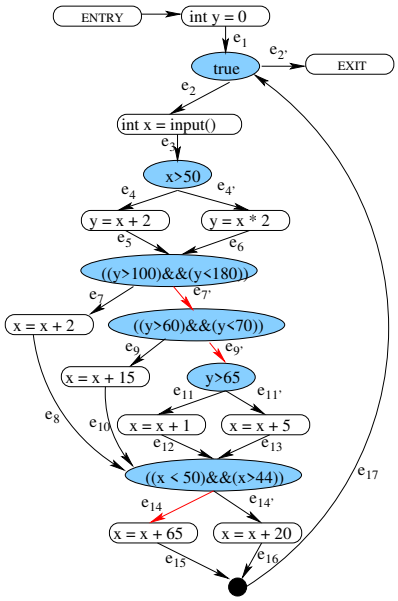

Figure 1: CFG

testing, depending on the requirement, certain parts of the system may need to be covered. In our work, we consider *edge coverage* as the coverage criterion. Our testing aims to cover a set T of edges, called as *target edges* which may be the set of all edges in the control flow graph of the system, or a proper subset thereof. A *test sequence* is a sequence of unit interactions between the system and the environment that completely covers a given target edge set T. During test execution, often the most time consuming (or resource intensive) process is the interaction with the environment which may involve network delays, mechanical movements and human actions. Hence, test sequence length, measured as the number of unit interactions of the system with the environment, may have a direct implication on cost of testing.

SymTest ensures generating shorter test sequences if the initial path obtained from the control flow graph is satisfiable. If backtracking occurs, then due to path explosion problem the shorter test sequences are not guaranteed. Consider the control flow graph (CFG) shown in Fig. 1. The edges marked in red in the CFG are the target edges we want to cover during testing. The path, $e_1e_2e_3e_4e_5e_{7'}e_{9'}e_{11}e_{12}e_{14}e_{15}e_{17}e_{2'}$ is computed and on performing symbolic execution on the obtained path, it is found to be unsatisfiable. Next step is to figure out the alternative optimal feasible path which covers the target edge. Optimal path here means syntactically shortest path that covers the target edges. Search strategy in SymTest follows backtracking by one decision edge to find the next path. In the example given, the new path obtained is also found to be unsatisfiable by the SMT solver. Backtracking proceeds further to search the feasible path and SymTest fails to terminate as it repeatedly takes the edge e_4 on each iteration in its search. Therefore this strategy does not ensure finding optimal path after backtracking. Contrary to this when we use WBS in SymTest it redirects the execution along the edge of $e_{4'}$, the resulting path is : $e_1e_2e_3e_{4'}e_6e_{7'}e_{9'}e_{11}e_{12}e_{14}e_{15}e_{17}e_{2'}$. This turns out to be feasible, thus resulting in a successful generation of test sequence of short length.

Classical search strategies - depth first search (DFS) have been used in tools DART [2], CUTE [3]. JPF [4] [5] has the capability to choose the search strategy DFS or BFS. JPF also comprises structural heuristics for path exploration. CREST and KLEE are two concolic testing tools which have been adopted for testing in industrial applications [6][7]. CREST uses control flow directed search, where static structure of the program is considered to explore program's path space. The other search strategies devised for concolic testing in CREST includes bounded depth-first search, uniform random search, random branch search [8]. The commonly used search strategies in KLEE[9] are - Coverage-Optimized Search, Depth First Search, Random Path Select, Random State Search. Fitnex [10] is a search strategy used to guide path exploration in dynamic symbolic execution to achieve test target coverage. Program-derived fitness functions were used to calculate the fitness values for the explored paths. The fitness function measures how close an explored path is in achieving test target coverage. For effective exploration, the core Fitnex strategy has been integrated with other search strategies.

Search strategies used in the symbolic execution tools like KLEE, CREST focus on achieving high code coverage. They do not ensure target edge coverage in least number of iterations. On the other hand the integration of WBS in original SymTest focuses on target edge coverage in fewest number of iterations.

3 Proposed Approach

Before explaining the WBS in detail in Section 3.2, notations used in the paper are introduced here and Section 3.1 illustrates how WBS is used in SymTest to improve the generation of test sequences.

Notations Some of the notational conventions followed in the sequel is given here.

- $\mathcal{P}$: Program under test
- $\mathcal{G}$: Control flow graph of $\mathcal{P}$
- $V(\mathcal{G})$: Node set of $\mathcal{G}$
- $E(\mathcal{G})$: Edge set of $\mathcal{G}$
- Nodes in $V(\mathcal{G})$: represented by names like n, n', n_i, , where $i \in \mathbb{N}$
- Edges in $E(\mathcal{G})$: represented by names like e, e', e_i, where $i \in \mathbb{N}$
- Each decision node in $\mathcal{G}$ has two outgoing decision edges e and e'. We say that $e = flip(e')$ and $e' = flip(e)$.
- T: Target edge set
- $\mathcal{T}$: Computational tree
- $V(\mathcal{T})$: Node set of $\mathcal{T}$

- $E(\mathcal{T})$: Edge set of $\mathcal{T}$
- Nodes in $V(\mathcal{T})$ are represented by names like $\mathbf{n}$, $\mathbf{n}'$, $\mathbf{n_1}$, $\mathbf{n_i}$, where $i \in \mathbb{N}$
- $\rightarrow_{\mathcal{G}}$ is a relation between two decision edges $e_i, e_j \in E(\mathcal{G})$ such that e_i is an immediately preceding decision edge to e_j in at least one run of the program.

3.1 SymTest-WBS

A revised version of the SymTest algorithm using WBS is presented in Algorithm 1. We use FINDCFPATH [1] to find an optimal syntactic path through the control flow graph that covers all members in the target edge set T. The path thus computed is pushed into the $stack$ as a sequence of decision edges. We symbolically execute along this path

Algorithm 1 SymTest-WBS

```
1:  procedure SYMTEST(G, T, W)
2:      stack ← <>
3:      while true do
4:          stack ← FINDCFPATH(G, T, stack)
5:          sympath ← SYMEX(G, stack)
6:          pc ← PC(sympath)
7:          tf, M ← SOLVE(pc)
8:          if tf then
9:              return M
10:         else
11:             b ← BTP(G, stack, W, T)
12:             stack ← BACKTRACK(stack, flip(b))
13:             PUSH(stack, b)
```

giving us the symbolic trace along that path ($sympath$). We convert this into a logic formula and input it to an SMT solver (SOLVE). If the formula is found satisfiable (indicated by true value of the true or false (tf) component of the value returned by SOLVE), our search was successful; the test input can be directly extracted from the model M returned by the solver. However, if the solver fails to solve the formula, it indicates that the path computed by FINDCFPATH is infeasible. We need to backtrack (BACKTRACK) and explore an alternative path.

On meeting infeasibility this version of SymTest backtracks by potentially multiple decision edges in $stack$. The decision edge b to which this backtracking takes place is computed by BTP. BTP function takes as a parameter $W : E(\mathcal{G}) \times E(\mathcal{G}) \rightarrow \mathbb{R}$. W is a map which takes takes two edges e_1 and e_2 in $E(\mathcal{G})$ as inputs and returns a weight that is related with the probability of computing a *short* test path that reaches e_2 through e_1. The BTP function itself is presented in Algorithm 2, after introducing the prerequisite matter about the preprocessing steps.

3.2 Weighted Backtracking Strategy

An overview of WBS architecture is given in Fig 2. The preprocessing and computing backtrack point are the major steps of WBS. In preprocessing, the experience about the program under test is collected through previous runs. Note that once this preprocessing step is done, its output is usable by any number of runs of SymTest. After preprocessing, backtracking point is computed using the pending targets to be covered and edge ordering.

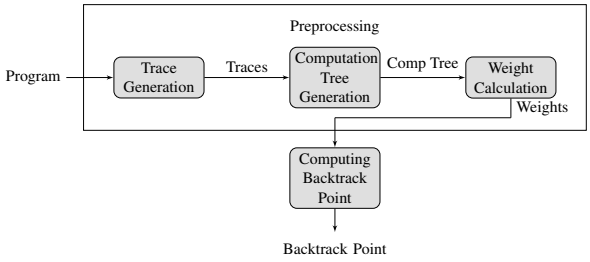

Figure 2: WBS Architecture

3.2.1 Trace Generation

We define a trace π as a list of decision edges $\{e_1, e_2, ..., e_n\}$ which are traversed during a particular run of the program. To generate traces, we instrument the program under test such that every time a edge is traversed during its execution, its corresponding edge id is printed into a file. Let D be the pre-assigned maximum depth of execution. This instrumented program is executed N number of times to generate the set of traces $\Pi = \{\pi_1, \pi_2, ..., \pi_N\}$. These are used to generate the computation tree.

3.2.2 Computation Tree Generation

A computation tree is the compressed representation of set of all the traces (Π). The computation tree is a suffix tree defined over the collected traces Π. Let ($treepath(\pi_i)$) be the path in computation tree ($\mathcal{T}$) which corresponds to the trace π_i in Π. If two traces π_1 and π_2 have a common prefix, this will result in their corresponding paths ($treepath(\pi_1)$ and $treepath(\pi_2)$ respectively) in the computation tree to be merged from the root till the point where π_1 and π_2 diverge. The computation tree CT has a unique root node denoted by $CT.root$. Each node $\mathbf{n}$ in a tree is a triple (e, C, np) where $\mathbf{n}.e$ corresponds to the decision edge in the control flow graph ($\mathcal{G}$) of the program ($\mathcal{P}$), $\mathbf{n}.C$ is the set of child nodes of $\mathbf{n}$ and $\mathbf{n}.np$ is the number of individual tree paths in which $\mathbf{n}$ occurs.

3.2.3 Weight Calculation

The selection of backtracking point is based on the preferability of a decision edge as a backtracking destination. If the current path traversed is defined by the sequence $\{e_1, e_2, ..., e_n\}$ where we meet unsatisfiability at e_n then the set of *candidate backtracking points* (CBP) are $\{e'_1, e'_2, ..., e'_n\}$ where for a decision edge e_i, $e'_i = flip(e_i)$ is the complimentary decision edge of e_i. Selection of the candidate backtracking points yielding the best result (i.e. the shortest test sequence covering all target edges) is undecidable. Hence, we have devised a heuristic to determine the probably best backtracking point and is computed by calculating a preferability index called *weight* (W) of each CBP. Weight for a CBP is calculated taking into account two metrics: probability of covering all the pending target

edges and length of the resulting test sequence.

Probability Weight (W_P): The probability of reaching a target edge represents how often the target edge is covered by the paths obtained. Let e be the current decision edge, P be the total number of all explored paths passing through e, $T = \{e_{t_1}, e_{t_2}, ...e_{t_n}\}$ be the set of all target edges, $P_{(e,e_{t_i})}$ be the total number of paths that cover the edge e and the target edge e_{t_i} in that order. Then, we estimate the probability to reach target edge e_{t_i} from current decision edge e, which we call the probability weight ($W_P(e, e_{t_i})$), is given by:

$$W_P(e, e_{t_i}) = \frac{P_{(e,e_{t_i})}}{P} \tag{1}$$

Length Weight (W_L) : Shortness of the test sequence is indicated by the length weight (W_L). Shorter the test sequence larger will be the value of W_L. W_L is calculated using individual length weight w_L, ($w_L : V(\mathcal{T}) \times E(\mathcal{G}) \rightarrow \mathbb{R}$). For any treenode $\mathbf{n} \in V(\mathcal{T})$, and target edge $e_{t_i} \in T$, the individual length weight is given as:

$$w_L(\mathbf{n}, e_{t_2}) = \sum_{\mathbf{n}' \in N_2} \frac{\mathbf{n}'.np}{L(\mathbf{n}, \mathbf{n}') + 1} \tag{2}$$

where, $N_2 = \{\mathbf{n}' \in V(\mathcal{T}) | \mathbf{n}'.e = e_{t_2}\}$, $\mathbf{n}'$ is reachable from $\mathbf{n}$, np is defined in Section 3.2.2, $L(\mathbf{n}, \mathbf{n}')$ is the length of the path from $\mathbf{n}$ and $\mathbf{n}'$ measured as the number of occurrences of the true branch of the branching node representing the outer loop.

Length weight, W_L of a decision edge $e \in E(\mathcal{G})$ is the sum of the individual length weights of each treenode $\mathbf{n} \in V(\mathcal{T})$.

$$W_L(e_1, e_{t_2}) = \sum_{\mathbf{n} \in N_1} w_L(\mathbf{n}, e_{t_2}) \tag{3}$$

where, $N_1 = \{\mathbf{n} \in V(\mathcal{T}) | \mathbf{n}.e = e_1\}$

Individual Weight (W_I): Individual weight $W_I(e', e_{t_i})$ is defined as the weighted sum of the probability weight and length weight between edges e' in CBP and e_{t_i} in target edge set. Thus:

$$W_I(e', e_{t_i}) = W_P(e', e_{t_i}) + W_L(e', e_{t_i}) \tag{4}$$

Composite Weight (W): Composite weight $W(e')$ of a decision edge e' is the cumulative weight of e' for all target edges given by:

$$W(e') = \sum_{e_{t_i} \in T'} W_I(e', e_{t_i}) \tag{5}$$

where T' is the pending target edge set(defined in Section 3.2.4) for e' with program control state defined by $stack$.

3.2.4 Computing Backtracking Point

At any point during the symbolic execution, the control state of the execution is captured by the state of $stack$. Some of the edges in the target edge set T may be included in this $stack$. These target edges do not need to be covered in the further part of the test sequence. The remainder of the target edges are the *pending targets* (T'), which need to be covered in the further part of the generated test sequence. $T'(e_i{}')$ denotes the pending target edges to be covered from $e_i{}'$.

Consider CBPs e'_1 and e'_2 such that $W(e'_1) > W(e'_2)$. We may want to select e'_1 as the final backtracking point. However, this may turn out to be a mistake. Let $T'(e'_1) = \{e^1_{t_1}, e^1_{t_2}\}$ and $T'(e'_2) = \{e^2_{t_1}, e^2_{t_2}\}$ denote the set of pending target edges for e'_1 and e'_2 respectively. The test sequence to be computed for e'_1 would have a suffix $ts^1_1 = \langle e'_1, e^1_{t_1}, e^1_{t_2}\rangle$ or $ts^1_2 = \langle e'_1, e^1_{t_2}, e^1_{t_1}\rangle$; likewise, for e'_2, they will be $ts^2_1 = \langle e'_2, e^2_{t_1}, e^2_{t_2}\rangle$ or $ts^2_2 = \langle e'_2, e^2_{t_2}, e^2_{t_1}\rangle$. Whether e'_1 should be finally chosen as the backtracking point, or e'_2, depends on which of ts^1_1, ts^1_2, ts^2_1 and ts^2_2 turn out to be the best path. For example, if the weights of the path suffixes ts^1_1, ts^1_2,ts^2_1, ts^2_2 (path weight will be defined below) turn out to be related in the following manner: $W(ts^2_1) > W(ts^2_2) > W(ts^1_1) > W(ts^1_2)$, it is then better to select e'_2 as the backtracking point inspite of the fact that $W(e'_1) > W(e'_2)$. In summary, to make a decision on the backtracking point, we must not just consider the weight of a CBP, but also the weight of the path that will be followed thereafter. In order to achieve this, *edge ordering* of targets edges is performed.

Unfortunately, for a given CBP e'_i, problem of computing the best test sequence suffix $\langle e'_i, ts^i_1, ts^i_2, ..., ts^i_n\rangle$ is akin to *vehicle routing problem* (VRP) A brute force approach would compute the path weights corresponding to all permutations of the pending target edges T' which is exponential in $|T'|$. For an arbitrarily large T', this would be too expensive.

Algorithm 2 Computing Backtracking Point

```
1: function BTP(G, stack, W, T)
2:     s ←< flip(e)∀(e, tf) ∈ stack >
3:     D' ← the subset of s such that it consists of upto
       N elements of s whose composed weights are the max-
       imum.
4:     for all d ∈ D' do
5:         T' ← PENDINGTARGETS(d, T, stack)
6:         EO[d] ← EDGEORDERING(d, T')
7:     return argmin_{d∈D'} EO[d]
```

In order to perform target edge ordering, initially a *weighted graph*($\mathcal{G}_W$) is created using a given set of pending target edges T' in $E(\mathcal{G})$. $\mathcal{G}_W$ is a complete weighted digraph such that:

- The node set $V(\mathcal{G}_W)$ corresponds to the edge of pending target edge set T'. $\forall n \in V(\mathcal{G}_W)$, $n.e$ denotes the target edge corresponding to it.
- $\forall n_i, n_j \in V(\mathcal{G}_W)$,

$$n_i \rightarrow_{G_W} n_j \text{ if } W(n_i.e, n_j.e) \geq W(n_j.e, n_i.e)$$
$$n_i \rightarrow_{G_W} n_j \text{ otherwise}$$

  where $n_i \rightarrow_{G_W} n_j$ means that there exists an edge between n_i and n_j.
- An edge $e(n_i, n_j)$ from a node n_i to n_j is annotated by a number $e(n_i, n_j).w$ given by:

$$e(n_i, n_j).w = \frac{1}{W(n_i.e, n_j.e)}$$

Augmented Weighted Graph ($\mathcal{G}'_W$) The weighted graph is augmented with the candidate backtracking point to get $\mathcal{G}'_W$. The edge ordering is performed on $\mathcal{G}'_W$. For a given CBP e and weighted graph $\mathcal{G}_W$, We define an augmented weighted graph $\mathcal{G}'_W$ as follows:

- $\mathcal{G}'_W$ has all the nodes and edges of $\mathcal{G}_W$.
- $V(\mathcal{G}'_W) = V(\mathcal{G}_W) \cup \{n\}$ such that $n.e = e$.
- For all nodes $n' \in V(\mathcal{G}'_W) \setminus \{n\}$, there exists an edge $e(n, n')$ such that $e(n, n').w = \frac{1}{W(n.e, n'.e)}$.

Algorithm 2 presents BTP function that selects a decision edge from the candidate backtrack points (CBP) after performing edge ordering on the pending targets edges.

4 Experimental Evaluation

In this section we first describe the experimental set up. We then explain the benchmarks used to evaluate the implementation followed by a discussion of the results.

4.1 Set up

We have implemented WBS and integrated it with SymTest. The weight is calculated using 500 previous runs of the program under test. To evaluate the effectiveness of SymTest-WBS, we considered the following factors:

- time taken for test sequence generation
- test sequence length i.e., the number of iterations taken to cover the target edges
- number of execution paths processed to generate the test sequence covering the target edges

A large number of execution paths processed for test sequence generation will result in an increase in the time for test sequence generation. More number of iterations to cover the target edge increases interaction with the external environment. Both these facts contribute to the cost of testing of an embedded software.

We compared SymTest-WBS with following tools on the basis of above parameters:

- SymTest [1]: SymTest framework where backtracking proceeds by a single node.
- KLEE [9]: Symbolic execution tool build over LLVM, that generates test cases with high code coverage.

Each of the above mentioned tools is evaluated against different test programs. We ran our tool on the benchmark programs on an Intel Quad Core i7-3770 3.40GHz machine running Ubuntu 18.04.4.

The number of iterations of the main loop is considered as the performance metric for measuring the effectiveness of symbolic execution search strategies[10]. For all our experiments we have set the maximum depth of execution (i.e. number of iterations of the main loop) as 5.

4.2 Benchmarks

TABLE 1 shows the list of benchmark programs that we used in our experiments and the results obtained. We used the benchmark programs which are used by other symbolic execution tools [11] [12] [1]. As we are concentrating on embedded software, apart from available benchmarks, we have converted a set of single task programmable logic control programs [13] into its equivalent C program. The other test programs are from SVCOMP test suite [14], CREST test programs [12], RERS challenge [15].

4.3 Results

TABLE 1 shows the results of experiments. The "#iter" column gives the number of iterations taken by the generated test case to cover the target edges,"#path" column gives the number of explored paths, "Target Cov" gives the percentage of the target edges covered. The "Time (ms)" column gives the time taken by the tool in milliseconds. The total number of iterations is calculated by taking the sum of iterations taken by each test data. A time out (TO) happens at a time limit of 600000ms. A time out means that the tool could not generate a test sequence that covers all the target edges within the stipulated time-out period.

On comparing the proposed approach with SymTest, It is observed that in test_program, cfg1 SymTest-WBS produces shorter test sequence length in less time compared to SymTest. In problem1M, cfg2 SymTest timed out, but SymTest-WBS succeeded to cover target edges. On com-

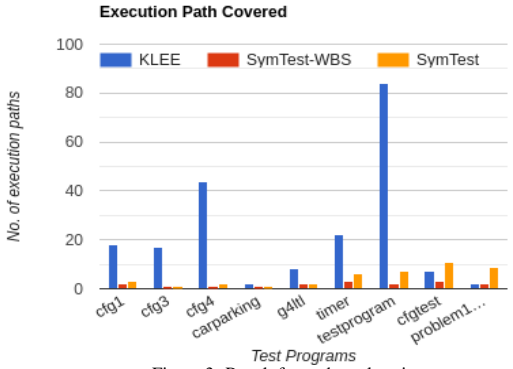

Figure 3: Result for path exploration

Sl No.	Program	SymTest-WBS				SymTest				KLEE			
		#iter	#path	Target Cov	Time(ms)	#iter	#path	Target Cov	Time(ms)	#iter	#path	Target Cov	Time(ms)
1	cfg1	1	2	100	2081.27	2	3	100	2357.11	7	18	100	9126.10
2	cfg2	1	2	100	1040.38	-	-	-	TO	4	14	100	9018.61
3	cfg3	1	1	100	124.10	1	1	100	245.50	3	17	100	8094.71
4	cfg4	1	1	100	337.14	1	2	100	175.77	4	44	100	5804.59
5	test_program	1	2	100	513.50	4	7	100	1001.21	4	84	100	12095.66
6	timer	1	3	100	126.96	1	6	100	234.60	5	22	100	72.26
7	car_parking	1	1	100	117.89	1	1	100	192.36	4	2	100	78.81
8	burglar_alarm	3	2	100	1173.94	3	9	100	5542.12	-	-	-	T0
9	g4ltl1	1	2	100	923.90	1	2	100	1594.23	5	8	100	159.56
10	g4ltl2	1	2	100	923.90	1	2	100	1594.23	5	7	100	163.30
11	trex03	1	1	100	231.54	1	1	100	240.01	1	102	100	1346.62
12	cfg_test	2	3	100	292.42	2	11	100	271.34	2	7	100	380.27
13	problem1M	2	4	100	3819.30	-	-	-	TO	-	-	-	TO
14	problem2M	2	3	100	399.79	2	10	100	8586.02	-	-	-	TO
15	problem11M	2	2	100	510.04	2	9	100	1795.25	2	2	100	73.15
16	problem1	-	-	-	TO	-	-	-	TO	-	-	-	TO
17	problem11	-	-	-	TO	-	-	-	TO	-	-	-	TO

Table 1: Comparison Results

paring the proposed approach with KLEE, it is observed that for programs having majority of statements which are reachable, KLEE takes more time in test sequence generation compared to SymTest and SymTest-WBS for cfg_test, test_program. KLEE focuses on code coverage. If we are interested in complete code coverage (as in first-time testing), KLEE is one of the best choices for test data generation. But if we want to compute test data to reach specific program points which are deep in the code (as often seen in regression testing), then SymTest-WBS performs better. The number of iterations to cover the same target edges is more in KLEE. This can be observed in g4ltl1, timer, car_parking. In problem1M, problem2M KLEE timed out, but SymTest-WBS was successful in generating test sequence.

In problem1, problem11 it is observed that all the three tools timed out. Fig. 3 shows the results of path exploration for those programs which was successful in generating test sequence by all the three tools. SymTest-WBS requires less number of paths to attain target edge coverage compared to SymTest and KLEE. It is worthwhile observing that the number of TO by SymTest-WBS is less compared to SymTest and KLEE. Both SymTest and KLEE explore more number of execution paths to cover target edges. This explains the timeout observed in SymTest and KLEE.

5 Conclusions and Future Work

In this paper we proposed a backtracking heuristic named Weighted Backtracking Strategy. The knowledge about system behaviour through previous runs is exploited to create a computation tree. Using the computation tree the optimal backtracking point is selected. WBS is implemented in SymTest. Experiments shows that the WBS is effective in increasing the set of cases where SymTest achieves termination and has further shortened the test sequence length with respect to those achieved by SymTest with simple backtracking. Our future work involves using machine learning techniques for selecting backtracking point.

References

[1] S. Chakrabarti and R. S, "Symtest a framework for symbolic testing of embedded software," in *SymTest*. ISEC, 2016.

[2] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[3] *CUTE: a concolic unit testing engine for C*. 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-13, 2005.

[4] S. Anand, C. S. Păsăreanu, and W. Visser, "Jpf-se: A symbolic execution extension to java pathfinder," in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 134–138. [Online]. Available: http://dl.acm.org/citation.cfm?id=1763507.1763523

[5] *Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis*. Automated Software Engineering, 2013.

[6] M. Kim, Y. Kim, and Y. Jang, "Industrial application of concolic testing on embedded software: Case studies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 390–399.

[7] Y. Kim, M. Kim, Y. J. Kim, and Y. Jang, "Industrial application of concolic testing approach: A case study on libexif by using crest-bv and klee," *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1143–1152, 2012.

[8] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 443–446. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.69

[9] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[10] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, June 2009, pp. 359–368.

[11] J. Jaffar, R. Maghareh, S. Godboley, and X.-L. Ha, "Tracerx: Dynamic symbolic execution with interpolation (competition contribution)," in *Fundamental Approaches to Software Engineering*, H. Wehrheim and J. Cabot, Eds. Cham: Springer International Publishing, 2020, pp. 530–534.

[12] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446.

[13] https://www.sanfoundry.com, [Online; accessed 19-September-2020].

[14] https://sv-comp.sosy-lab.org/2021/benchmarks.php, [Online; accessed 19-September-2021].

[15] http://rers-challenge.org/2017/index.php, [Online; accessed 10-September-2021].