*electronics*

MDPI

*Article*

# Scalable Path Search for Automated Test Case Generation

**Enze Ma [1,*], Xiufeng Fu [2] and Xi Wang [3]**

[1] School of Information Science and Technology, Beijing Forestry University, Beijing 100107, China
[2] Beijing Institute of Computer Technology and Application, Beijing 100854, China; xiufengfu1@gmail.com
[3] School of Computer Technology and Science, Shanghai University, Shanghai 200444, China; wangxi@t.shu.edu.cn
* Correspondence: joseph9morgan@bjfu.edu.cn

**Abstract:** Test case generation is an important task during software testing. In this paper, we present a new test-case generation framework for C programs. This approach combines dataflow analysis and dynamic symbolic execution together, and more importantly, it efficiently searches the program path space for potential faults based on the tabu search strategy and the program fault statistics. Unlike the traditional symbolic execution, which explores the program space exhaustively and is difficult to apply to complicated programs effectively, our approach automatically explores the feasible paths of hidden faults with high probability. The scalable and efficient path search strategy facilitates bug finding with much fewer test cases generated. We implemented this approach, and the experimental results presented in this paper are attractive.

**Keywords:** test-case generation; symbolic execution; heuristic path selection



**Citation:** Ma, E.; Fu, X.; Wang, X. Scalable Path Search for Automated Test Case Generation. *Electronics* **2022**, *11*, 727. https://doi.org/10.3390/electronics11050727

Academic Editors: Katarzyna Antosz, Jose Machado, Yi Ren, Rochdi El Abdi, Dariusz Mazurkiewicz, Marina Ranga, Pierluigi Rea, Vijaya Kumar Manupati, Emilia Villani and Erika Ottaviano

Received: 14 January 2022
Accepted: 23 February 2022
Published: 26 February 2022



## 1. Introduction

The most commonly used technique for validating the quality of software is testing, which is a labor intensive process. It occupies more than half the total costs during software development and maintenance [1]. Among the various kinds of testing techniques, unit testing is popular for checking the aspects of the implementation for software component under test. However, it is a tedious task for programmers to write test cases manually. However, manual test cases are difficult to evaluate by coverage criteria, such as branch coverage, path coverage, etc.

Symbolic execution [2] is a well-known technique to automate test-case generation. Instead of supplying concrete inputs to a program, symbolic execution supplies symbols that represent arbitrary values. Dynamic symbolic execution [3–5] is proposed to intertwine concrete and symbolic execution together in a way that analyzes program behaviors dynamically for automatically generating new test inputs systematically. Though dynamic symbolic execution improves the efficiency of generating test cases to an extent, it is still difficult to be applied to complicated or large programs. The main reason is that dynamic symbolic execution intends to explore the whole space that may impose the state explosion problem. In fact, the testers do not need to generate all possible test cases for the program under test, because what we care about is those test cases, which may lead to program faults with high probability.

To overcome the weakness of existing techniques, we present a new test-case generation framework. This framework combines dataflow analysis with dynamic symbolic execution and heuristically searches for program path space based on the tabu search strategy and the program fault statistics. This approach relies on the program code itself and directs program paths to automatically follow the ones that most likely contain faults, instead of searching the whole feasible path space. These techniques enable us to find more possible errors in programs with fewer test cases generated, and it scales well w.r.t. program sizes (about 5000 lines per unit). As a result, this approach can be applied to real-world programs to find potential errors with reasonable costs.

The main contribution made by this approach is the path-selection model with which the exploration process will be likely to search sub state spaces where the probability of hidden errors is high based on the statistical results. Figure 1 shows a diagrammatic comparison between a pure dynamic symbolic execution technique (CUTE [3], DART [4] ) and our approach. The boxes stand for the entire program state space, and the small circles distributed in boxes are possible bugs hidden in the state space.

Figure 1a shows the dynamic symbolic execution search, and this will exhaustively explore continuous parts of the state space. In this way, this technique may be stuck in parts of program space while other bugs cannot be uncovered in the left space where there may be more possible bugs. Based on the bug statistical report [6], those possible bugs are not in a uniform distribution in the program space. Figure 1b shows our approach, which will search those sub-spaces where the possibility of hidden bugs highly relies on statistic results.

Consequently, our approach does not search the program space in a uniform way, and it works as an intelligent searching technique to some extent to further improve the search quality. The other advantage of adopting the tabu heuristic [7–10] is to keep the search process from becoming stuck in sub-spaces, as a tabu list can help escape from the local search space.

This paper is organized as follows. We present a motivating example in Section 2. Section 3 describes the test-case generation framework. Section 4 presents our algorithm for effective path selection and discusses how to apply the tabu search strategy in order to find potential defects more efficiently. Section 5 discusses the related work. Section 6 reports the experimental results that we obtained with the implementation of our approach. The last section concludes the study.

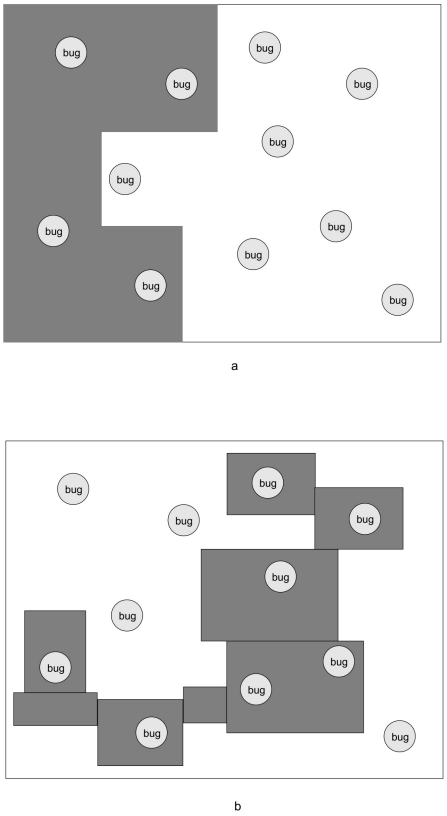


**Figure 1.** Pure dynamic symbolic execution vs. our approach. Here (**a**) shows the dynamic symbolic execution search will exhaustively explore continuous parts of the state space, and (**b**) shows our approach will search those sub-spaces where the possibility of hidden bugs highly relies on statistic results.

## 2. Motivating Example

We illustrate the benefits of our approach using a simple example. The code fragment is listed in the following, where there exists a pointer misuse in the else branch, which may lead to a runtime error. The runtime error of accessing null pointers takes place when there is only one node in the linked list.

```
struct Node {
    int v;
    struct Node *next;
};

struct Table {
    int a[1000];
    struct Node *p;
    int cnt;
};

void delete_from_table(struct Table *p_table) {
     struct Node *p1;
     int i;
     if(p_table->p == NULL){
         for(i = 1; i < p_table->cnt && i < 1000; i++)
              p_table->a[i - 1] = p_table->a[i];
     }
     else{
         p1 = p_table->p->next;
         p_table->p->next = p1->next;
         p_table->p->v = p1->v;
         free(p1);
     }
     p_table->cnt--;

}
```

The first step of using our approach is to instrument the program under test. For instance, the function delete_from_table under test will be transformed into the following, where the for structure is replaced by goto statement with an if one. Note that there are three embedded if branches in the code after the transformation. Thus, the path analysis can rely on the branches of if in a uniform way. On the other hand, some instrumented code snippets are inserted into the original in order to collect program information based on the CIL tool [11]. For simplicity, we omit those instrumented code snippets here.

```
void delete_from_table(struct Table *p_table) {
     struct Node *p1;
     int i;
     if(p_table->p == NULL){
         i = 1;
         while(1)
         {
              if(i < 1000){
                  if(i < p_table->cnt)
                        ;
                  else break;
              }
              else break;
              p_table->a[i - 1] = p_table->a[i];
```

```
                i++;
            }
        }
        else{
            p1 = p_table->p->next;
            p_table->p->next = p1->next;
            p_table->p->v = p1->v;
            free(p1);
        }
        p_table->cnt--;
}
```

For choosing those paths that contain potential bugs with high probability, we are supposed to evaluate the paths in the search space. Thus, we designed a path selection model dealing with the evaluation and choice of those feasible paths. We also designed a path analysis engine, which helps in the analysis of the paths. To achieve this goal, the engine uses control flow graph (CFG) to facilitate the analysis process. Figure 2 shows the corresponding control flow graph of function delete_from_table.

However, it is impossible to evaluate each path in the search space; thus, we construct the control flow graph of the corresponding program by which the paths can be evaluated based on some criteria. For instance, Figure 2 shows the CFG of the example, where each branch path has been evaluated with a weight that is computed by a simple error-statistical model that will be explained in detail in Section 4. For instance, the path in the else part of the transformed program is of the highest value 4.03. With the beginning of the process of test-case generation, a memory is allocated dynamically and the corresponding address is assigned to the pointer p_tabale.

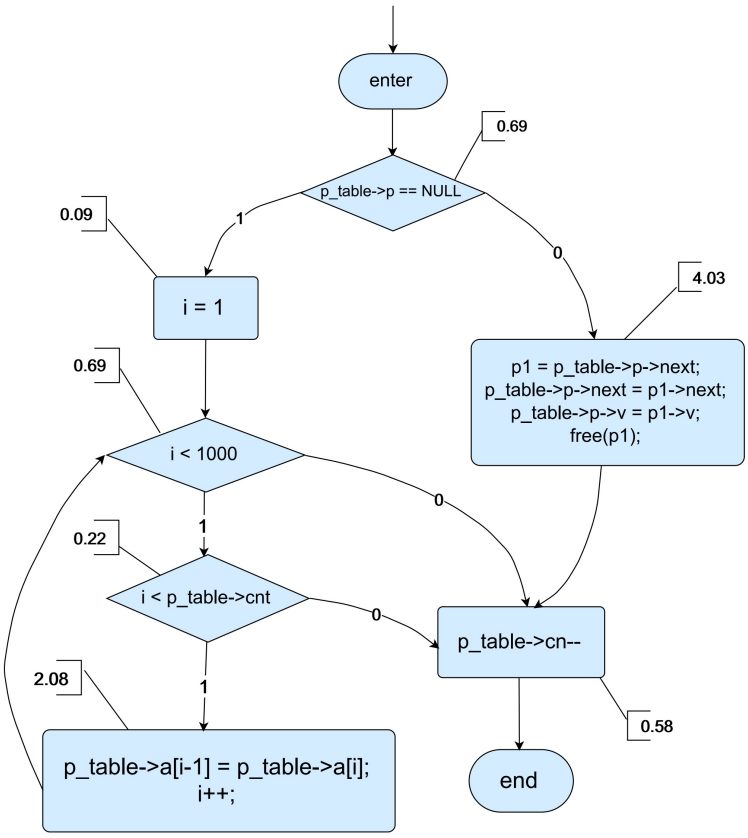


**Figure 2.** Control flow graph.

The initial values of p_table−> p and cnt are null and 0, respectively. Then, the path analysis engine first enters the if branch where (p_table−> p == NULL) holds and goes into the while loop afterwards. If we denote the if branch as 1 and else as 0, then we obtain an initial path whose first two elements are recorded as 1, and the third element is 0. Figure 3a shows this scenario, where the array stands for a real program path, and the triangle in a array represents the uncovered parts in the path.

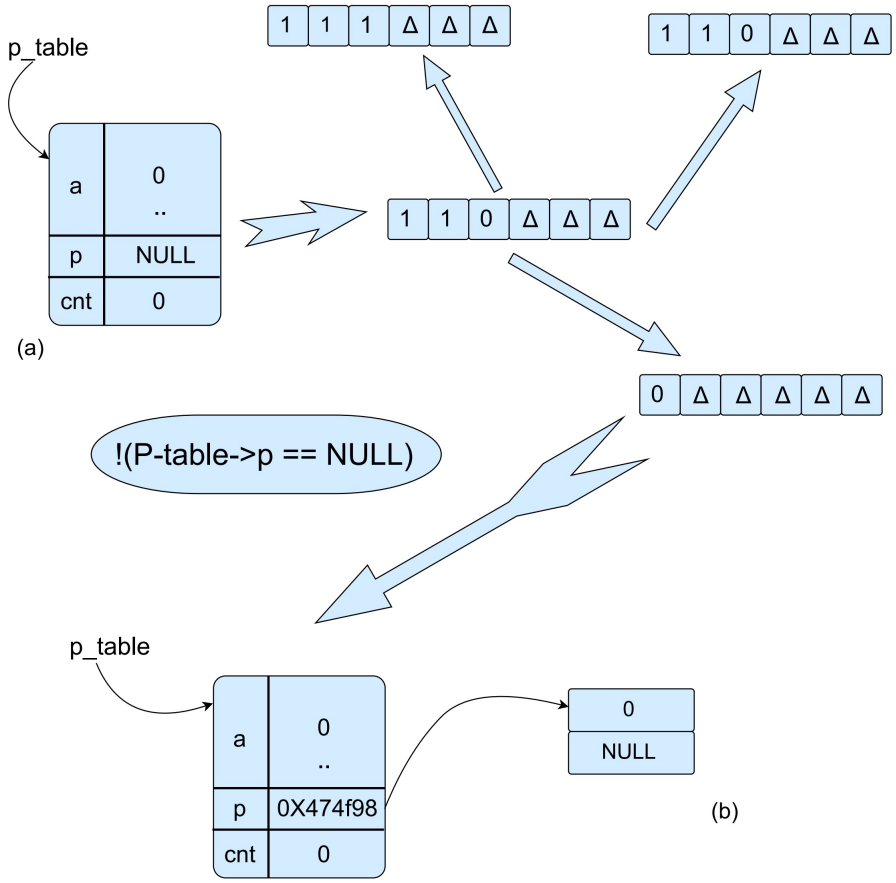


**Figure 3.** The search process. (**a**) shows a real program path. (**b**) shows a test case generated for executing of the chosen path.

In this situation, we obtain three neighbors of the current execution path. Based on our path-selection model, the next path whose weight is the highest will be selected to be explored. As a result, the path that goes into the outmost else branch will be selected. In Figure 2, the path in which the first element is 0 is chosen. As a result, the error in this path will be found in only the second execution under our path-selection model.

On the other hand, if we use the depth-first search instead of path-selection model we adopt, the path analysis engine will first enter the if branch and stay there for thousands of iterations because of the loop structure in that branch. To generate the test case that leads to the path to be explored, the dynamic symbolic execution is used in the test-case generation framework. During the execution of the current path, the path analysis engine collects the branch constraints at the same time.

When the next path is chosen, new test input data are generated by solving the corresponding constraints. For instance, by solving the constraints !(p_table−> p == NULL), the test case in Figure 3b is generated, which directs the execution of the chosen path.

## 3. Test Case Generation Framework

There are two main parts in the test-case generation framework shown in Figure 4. The first handles the dataflow analysis, which provides information for the second part.

In this part, we designed a dynamic def-use chain and a path execution tree as the engine infrastructures. The second part is the path analysis engine, which is the core part of this framework. The dynamic symbolic execution is integrated to generate test cases automatically. During the process of path exploration, tabu search is applied to select paths to be analyzed.

The process of static code analysis instruments the program unit under test and collects the path information for path selection using CIL [11]. The path analysis engine first generates test case inputs from the results of the solving constraints recorded in the previous executions. In the initial iteration, the test case input is generated randomly. The second step of the engine takes the current test input generated to execute the instrumented codes using the dynamic symbolic execution technique, which runs the program both in concrete and symbolic value states. Third, during the exploration of the path space, the engine smartly selects the next path to be executed and analyzed based on the tabu search strategy. If a new path is selected successfully, the engine enters the process of constraint solving, which generates the new test input to direct the next path execution. Otherwise, a test report is generated for the testers.

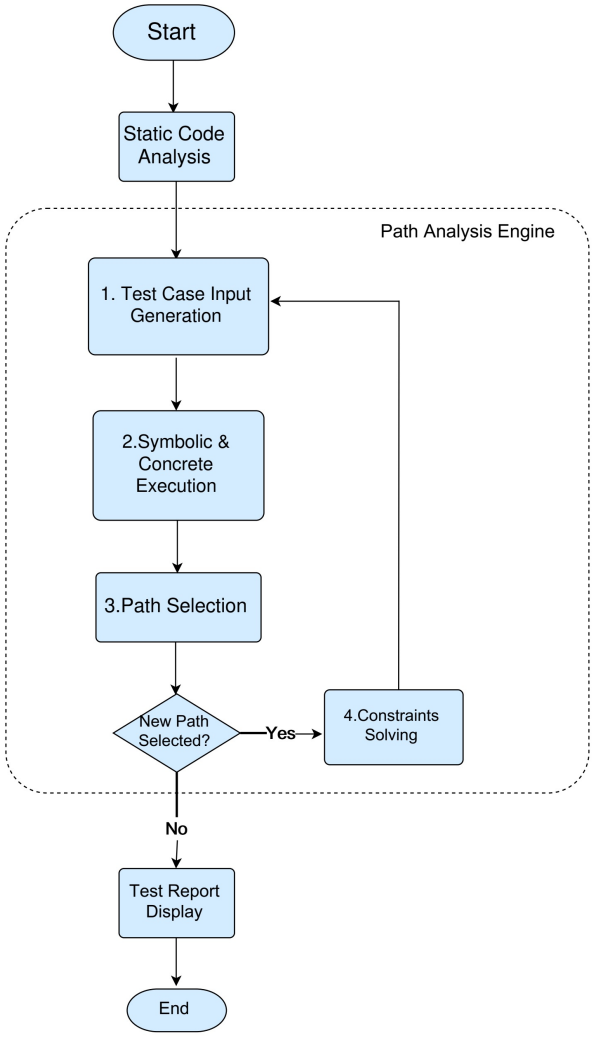


**Figure 4.** Test case generation framework.

### *3.1. Dynamic Symbolic Execution*

Traditionally, symbolic execution supplies symbols as input data to the program under test to represent arbitrary values. Our approach takes both symbols and normal concrete value as the test input. The greatest advantage of this approach is that the

program under test can be executed normally, and there is no need to implement a special simulator to execute it symbolically. This technique maintains two representation forms for variables dynamically: the symbolic form is denoted by `sym_structure`, and the concrete one denoted by `con_structure`. A dynamic def-use chain is implemented by the dataflow analysis technique, which is used to collect and record data information and path constraints during dynamic symbolic execution.

Dynamic symbolic execution is a classical test generation technique, where the program under test starts from some random inputs and is then executed through gathering symbolic constraints on inputs from predicates in branch program points; a constraint solver is then used to generate another input to guard different feasible path executions. On the other hand, static symbolic execution [12] does not need to execute programs under test, and it uses a static analysis technique to collect constraints information. However, this approach does not perform well as complex programs always need to call APIs or functions provided by systems, but is not easy for static analysis to abstract and reason about complex functions effectively.

### *3.2. Path Execution Tree*

We designed the path execution tree to record explored paths; it is the basis of our path-selection model. This tree grows dynamically and it is a persistent data structure during the whole process of test-case generation. When a branch statement is encountered during the symbolic execution, the branch condition is collected, and a node is added to this tree if the corresponding child of the current node is NULL. Then, the current node is set to be its false branch or true branch depending on the choice of the branch statement.

Each path from root of the tree to a leaf in the execution tree stands for a real execution path in the program under test. As we use CIL to instrument the tested program so that any branch statement in the transformed program is a simple two-choice statement, a sequence of "1|0" can be used to denote a path. A node in the execution tree has two children representing the false branch and true branch, respectively. The process of building an execution tree is shown in Figure 5.

We take the program used in Section 2 as an example. The top part of Figure 5 shows the test input data. First, the branch statement "$\mathsf{if}(\mathsf{p\_table} \rightarrow \mathsf{p} == \mathsf{NULL})$" encounters, the branch condition of current node in execution tree is set to be "$\mathsf{p\_table} \rightarrow \mathsf{p} == \mathsf{NULL}$", and then the current node becomes to be the true branch of the previous current node. The next branch statement encountered is "$\mathsf{if}(\mathsf{i} < 1000)$".

The condition of this statement is used as the branch condition of the current node in the execution tree. In this case, the value of i is 1, and thus the true branch is taken, and the current node turns to the true branch of the previous current node. Then, the engine executes the branch statement "$\mathsf{if}(\mathsf{i} < \mathsf{p\_table} \rightarrow \mathsf{cnt})$", and the branch condition of current node is set to be "$\mathsf{i} < \mathsf{p\_table} \rightarrow \mathsf{cnt}$". The concrete value of i is 1 and "$\mathsf{p\_table} \rightarrow \mathsf{cnt}$" is 0. Therefore, the false branch is taken.

When this symbolic execution terminates, the path covered in this iteration can be presented as a sequence "110". Path constraints can be gained from the execution tree. For example, as Section 2 shows, the next path selected to cover is "0". Thus, the path constraint directing this path is "$!(\mathsf{p\_table} \rightarrow \mathsf{p} == \mathsf{NULL})$". If the path selected to cover is "111", the path constraints will be "$(\mathsf{p\_table} \rightarrow \mathsf{p} == \mathsf{NULL}) \wedge (\mathsf{i} < 1000) \wedge (\mathsf{i} < \mathsf{p\_table} \rightarrow \mathsf{cnt})$".

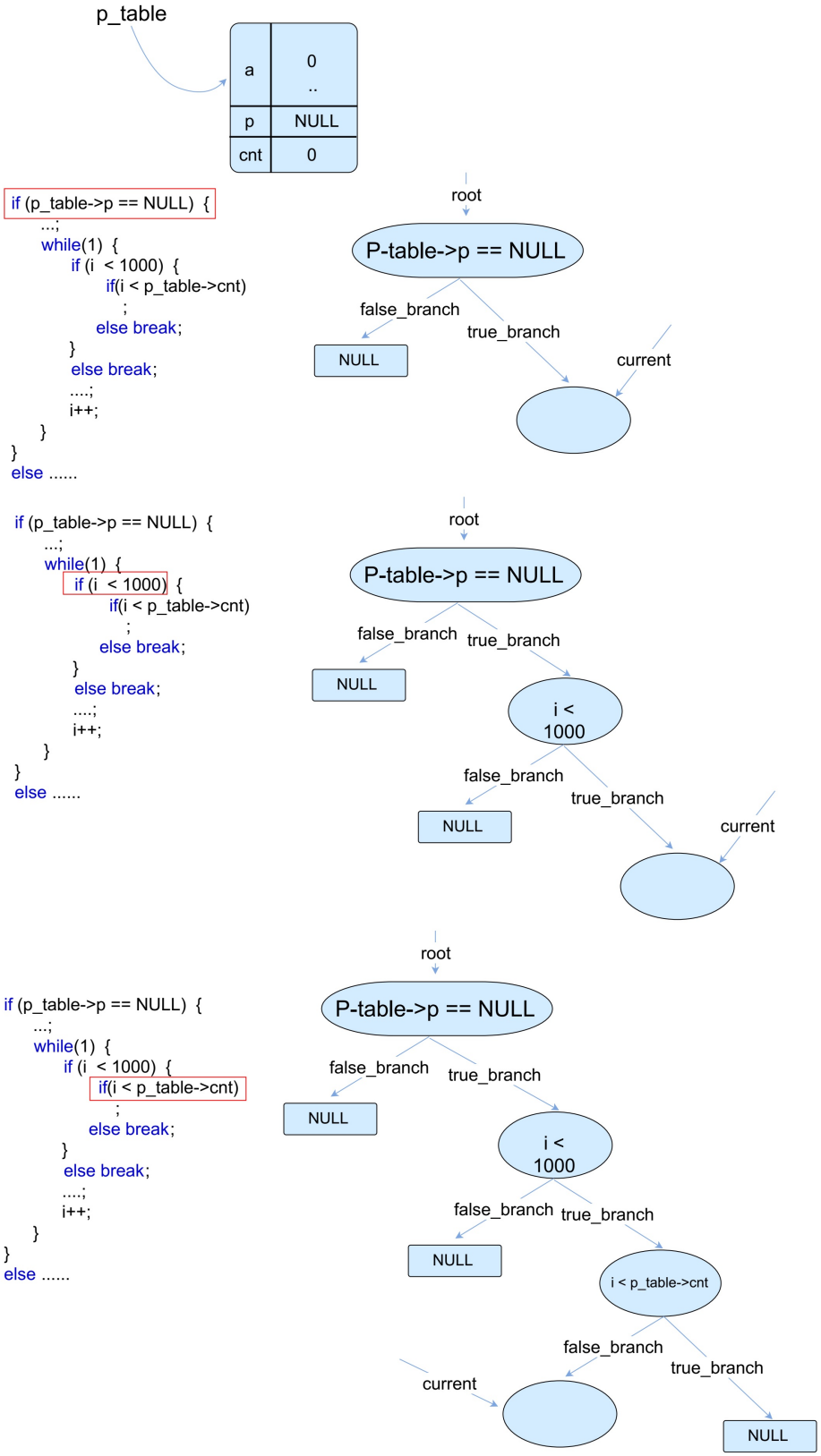


**Figure 5.** Path execution tree.

## 4. Path Selection Model

When the program unit under test is complicated, exploring every path in the program state space is an impossible job for industry applications. To alleviate the problem, we

present our path selection model and integrate it into the path analysis engine previously mentioned. This algorithm uses tabu search and a simple statistical model for efficient and effective path selection.

### *4.1. Tabu Search*

We use $\mathscr{S}$ to denote the search space of the problem under consideration, which could be either finite or infinite. For a point $s \in \mathscr{S}$ in the space, $\mathscr{N}(s)$ is called the neighborhood of $s \in \mathscr{S}$ defined by the problem, $\mathscr{N}(s) \subseteq \mathscr{S}$. A step of the tabu search algorithm is defined as moving from the current point $s$ to a neighboring point $s' \in \mathscr{N}(s)$, $s' \neq s$. Function $eval : \mathscr{S} \to \mathbb{R}$ is a function to evaluate the quality of the solutions. The general process of the tabu search algorithm is described in Listing 1.

**Listing 1.** Basic scheme of the tabu search.

```
1  s = RandomlyGet(𝒮);
2   while (!StopingCondition){
3    who = null;
4    local = MIN_VALUE;
5    for each (s' ∈ 𝒩(s)){
6     if (s' ∉ tabu list ∧ eval(s') > local)
7          who = s';
8          local = eval(s');
9       else if (Aspiration(s'))
10         who = s';
11         local = eval(s');
12   }
13   s = who;
14  }
```

The tabu search algorithm maintains a list (*tabu list*) recording the features of the last $\eta$ visited solutions, preventing the searching process from revisiting solutions with such features, where $\eta$ is called the tabu tenure length. The search process selects the best solution $\hat{s} \in \mathscr{N}(s)$ that is not recorded in the tabu list as its target of the next step. To make the searching process more robust, an aspiration criteria can be introduced to accept solutions rejected under certain conditions. One widely used form is that when $s' \in \mathscr{N}(s)$ is rejected but evaluation function $eval(s')$ is better than the best solution $s^*$ ever found before. At this time, $s'$ is accepted as the next visiting point despite of its tabu state.

### *4.2. Search Space*

The search space in the path-selection model is based on the path execution tree that we designed earlier. The search space is composed of all the feasible paths in the path execution tree. For example, Figure 6a denotes a path execution tree with a height of 4. If we use 0 to denote the false branch and 1 as true branch, then a sequence of digital 0 or 1 denotes a path. In Figure 6a, the path with thick dark can be denoted the sequence 1011. Thus, all the possible combinations of digital 0 and 1 form the search space in our model. With the search space fixed, we define the neighborhood of element $v$ denoted as $\mathscr{N}(v)$. To illustrate the construction of the neighborhood, we use the same example in Figure 6a.

The construction of the neighborhood of element $v$ begins with changing the last bit of $v$. In Figure 6b, $v_1$ is one of the neighborhood of $v$ from which only the last location of $v_1$ is different. The second neighbor of $v$ is $v_2$, which is different with $v$ on the last second location. Note that the last bit of $v_2$ is set a triangle, which denotes that the last location is unknown. The reason is that the execution tree is generated dynamically, and thus the path analysis engine does not know what other exact paths should be executed except for the current path denoted as $v$.

Thus, $v_2$ actually denotes a branch of unexplored paths. Similarly, $v_3$ and $v_4$ in the neighborhood of $v$ are constructed like $v_1$ and $v_2$. As a result, although the current path $v$ has only four neighbors, and each neighbor represents a set of paths. How to choose a path to be performed from a neighbor will be discussed in the following parts.

In general, if $n$ denotes the height of the execution tree without a circle, a point in the search space has exactly $n$ elements as its neighborhood. If there are circles denoting iterations existed in the tree, we can expand the path circles as two-path branches recursively. The loop structure may lead to the infinite paths in the search space, and one advantage of adopting tabu search can avoid being stuck in the loop structure based on the path-selection model.

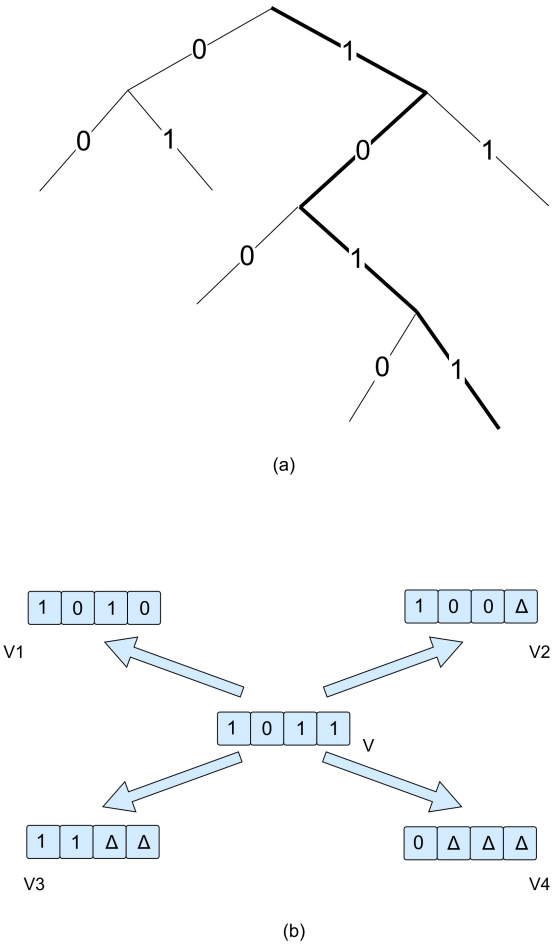


**Figure 6.** Neighborhood for one path. (**a**) denotes a path execution tree with a height of 4. (**b**) shows The construction of the neighborhood of element $v$.

### *4.3. Path Evaluation Function*

To decide which neighbor of the current path should be chosen, the path evaluation function should be designed carefully. The basic idea behind the path evaluation function *eval* is that the errors hidden in programs are not universally distributed—they may obey some statistical rules. Thus, our evaluation function is based on bug taxonomy and statistics [6] summarized for a large number of software projects using a statistical approach. Here, we enumerate the bug taxonomy considered in Table 1.

This table shows that the bug statistics model from [6], which covers from the requirement to testing phases. We only list the bugs related to the usage of statements and compute the corresponding weights based on the statistics column. For example, bugs related to pointers take up almost half of all the syntax-related bugs. This tells us that

the intensive uses of pointers may introduce more bugs and the path analysis engine is intended to set a higher priority to explore those program codes.

**Table 1.** Bug statistics.

| Bug Taxonomy | Abbreviation | Statistics | Weight |
|---|---|---|---|
| pointer | po | 7.50% | 0.47 |
| predicate | pr | 3.46% | 0.22 |
| loops | lo | 0.74% | 0.05 |
| arithmetic evaluation | ae | 1.72% | 0.11 |
| boolean evaluation | be | 1.03% | 0.06 |
| assignment | as | 1.56% | 0.09 |

To compute the usage of statements listed in Table 1, we collect those information on the corresponding control flow graph. A CFG can be represented as $\langle N, E \rangle$, where $N$ stands for the set of nodes and $E$ is a set of edges. Nodes in this graph are used to denote those branch statements, such as the if-else and loop statements; the directed edges connecting the nodes show the direction of programs running logic and keep records of a sequence of instructions that lie between the connected branch statements. Figure 7 shows the CFG with the corresponding program, where a tuple attached on the edge denotes the usage information in the current branch. For instance, tuple {1, 1, 0, 0, 1, 0} on the false branch denotes that there is one pointer, one predicate and one boolean evaluation in this path. Thus, the tuple is the usage time of pointers, assignments etc.

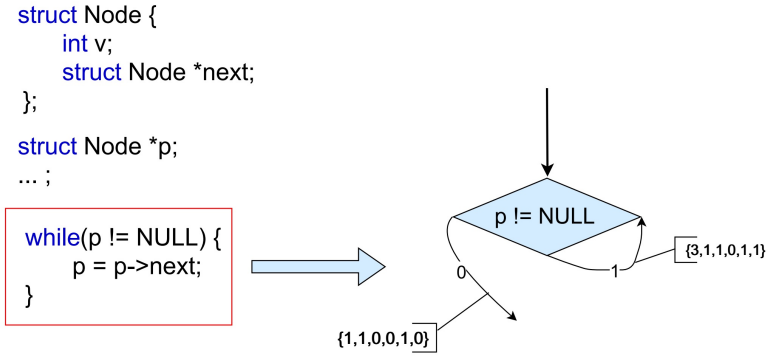


**Figure 7.** CFG with statement usages.

Next, we define the path evaluation function $eval(v)$, which is a key to the path selection algorithm, which will be introduced in the next section. Note that function $eval(v)$ has a parameter $v$, which is actually a point in search space. As shown in Figure 6, a point $v$ denotes a set of unexplored paths, and $eval(v)$ will obtain the highest weight from this set of paths as the weight of $v$. Listing 2 shows the algorithm of evaluation function $eval(v)$.

Function $eval(v)$ calls the recursive procedure $max\_tuple$, which will always select the branch with the higher weight. Function $eval\_tuple$ computes the weight of the corresponding path, where the normalization mechanism is applied to reduce the interference of the abnormal data. When dealing with a loop structure, we select and evaluate two different paths. One is the path that does not enter the loop, and the other is the path that goes into the loop once. Thus, the weight of a point containing loop structure can be computed easily. The weights in the CFG are pre-computed in our model, and this technique can improve the efficiency of the search process very much.

Function *eval* is like the algorithm of transversal of a binary tree, and the complexity of *eval* is linear with the number of nodes in the tree. It does not need to enumerate all the possible paths in the search space. Thus, the computing of the weight of the points in the neighborhood of $v$ is quick. Note that the path with highest weight may not be an executable one for the current unit under test. The reason for this is that we do not analyze

the reachability property of the path, and it can be solved by path reachability analysis before using our test-case generation framework.

**Listing 2.** Evaluation function.

```
/*
tuple : statistical information
{po, pr, lo, ae, be, as}
tuple_i : the i field of tuple
node : a node in CFG
node → tuple :
statistical information of node
node → false_branch :
false branch of node,
also a node of CFG
node → true_branch :
true branch of node,
also a node of CFG
eval : evaluation funciton
*/

procedure eval
input : node
output : value
value = eval_tuple(max_tuple(node))

procedure max_tuple
input : node
output : tuple
if (node == NULL)
return < 0, 0, 0, 0, 0, 0 >
else
tuple1 = max_tuple(node → false_branch);
tuple2 = max_tuple(node → true_branch);
if (eval_tuple(tuple1) > eval_tuple(tuple2))
return node → tuple ⊕ tuple1;
else return node → tuple ⊕ tuple2;

procedure eval_tuple
input : tuple
output : value
u = (∑_{i=1}^{6} tuple_i) / n ;
œ = sqrt( (∑_{i=1}^{n} (tuple_i − u)^2) / n );
∀i ∈ [1, 6], tuple_i = (tuple_i − u) / œ ;
value = ∑_{i=1}^{6} tuple_i * w_i;
```

### *4.4. Path Selection Algorithm*

Based on the previous discussions, we present the path selection algorithm, which is actually a variant of tabu search. The basic idea of this algorithm is to choose the best point not in the tabu list from the neighborhood of the current point. The tabu list can keep the search process from falling in the local search space. For instance, this algorithm can easily escape from the loop structure, while the depth-first search used in CUTE or DART will recursively perform in the loop. If the loop is infinite, then the search will not terminate.

Here, the best point refers to the path with highest bug-hidden probability based on our computation model. The algorithm framework is presented in Listing 3.

**Listing 3.** Path selection algorithm.

```
/*
    cp :  current path
    nps : all neighbors of cp
    bnp : best element in the neighborhood
    gbp : globally best element
    ɹ : tabu length
*/

procedure :  path selection

cp = Initial(); gbp = cp; base = √|cp|; TabuLen =
     RandomGetFrom(0.9 * base, 1.1 * base);

while(1){
  if (stopCondition())
    break;
  nps = getNeighborhoodPaths(cp);
  nps = removeVisitedorTabuedElements(nps);
  if (isEmpty(nps)){
    ɹ = 0.9 * ɹ;
    continue;
  }
  select bnp from nps;
  if (eval(bnp) > eval(gbp)){
    ɹ = 1.05 * ɹ;
    gbp = bnp;
    add other paths in nps into tabu list;
  }
randomly select a real executable path from bnp
```

The current path $cp$ is an executable one performed by the last test case. $nps$ denotes all neighbors of $cp$, each of which is a set of paths containing both performed and unperformed paths. The paths in the element of $nps$ have the same path prefix explored, which was discussed previously. We use $bnp$ and $gbp$ to denote the best element in the neighborhood and the globally best one respectively. Note that $bnp$ and $gbp$ both stand for a bunch of paths with the same path prefix.

The algorithm begins with initializing tabu length $\eta$ with a random value near to the square root of the length of the path. Then, $nps$ are obtained from the current path $cp$, and those paths explored in $nps$ are removed. If all the paths in search space are covered, then the algorithm terminates. The path is marked as a covered one if the symbolic execution finishes (i.e., an executed path), or a constraint solving fails (i.e., an unreachable path). In practice, we can set other stop conditions such that a predefined search height is arrived.

During the searching process, if all the possible moves become tabu ones and thus cannot find a element in $nps$, $\eta$ will be reduced by a factor, say 0.9, to help some points to escape from its tabu state faster. When the best path $bnp$ ever found is better than the global best path $gbp$ in one iteration, $\eta$ will be increased by another factor, say 1.05, to help the searching process to spread to further areas from the current local area. All the neighbors except $bnp$ in the current path will be added to tabu list, thus, ensuring that the searching

process will travel in the local space in the next $\eta$ iterations, where $\eta$ determines how long the tabu state of a specific point persists.

The key aspect for selecting a path containing possible potential errors to be executed in tabu search is the evaluation function: $eval : cp \rightarrow R^+$, which estimates whether the current node in the search space can be chosen as the point for the next iteration. The design of *eval* has been discussed previously. Though the evaluation function *eval* is dynamically computed from the corresponding CFG, the computation speed of evaluation function is very fast because it is linear to the number of elements in the search space.

As a neighbor of a the current path may be a branch of paths in tabu search space, we randomly choose the path as a next point. It is interesting that though the chosen path to be executed is not the one with the highest weight, the later iterations may still be performed in the local space where the highest weight path exists. If the path with the highest weight is performed, the later iteration will escape from the current local space and goes into the other parts of the search space.

## 5. Experimental Evaluation

We developed the C Analysis and Unit Testing toolkit (CAUT) based on the approach proposed here. We next present an experimental evaluation on CAUT. The evaluation is designed to answer the following research questions:

- RQ1. Is CAUT more effective and efficient than state-of-the-art testing and symbolic execution techniques?
- RQ2. Can CAUT effectively solve practical problems in real-world scenarios?

### *5.1. Results to RQ1*

#### 5.1.1. Settings

Benchmarks

We take open sourced programs as our benchmarks. These benchmarks are shown in Table 2. These benchmarks are mostly related to real-world software systems, mathematical computations, or string parsing. They also contain a large number of branches, suitable for measuring the effectiveness/efficiency of a testing technique in exploring program paths. We mark a total of 96 target program paths and guarantee that each path can be hit by at least one test input.

**Table 2.** Benchmarks used in the evaluation.

| Program | Description | Target |
|---|---|---|
| ReachSafety | Programs for reachability analysis | 8 |
| MemorySafety | Programs for verifying memory safety | 11 |
| NoOverflows | Programs for checking overflows | 13 |
| Termination | Programs for checking terminations | 12 |
| SoftwareSystems | Programs collected from real software systems | 21 |
| StrictMath | Math Libraries | 12 |
| Math | Math computations | 6 |
| FloatingDecimal | Floating point calculations | 7 |
| BigInteger | Large integer calculations | 6 |

Techniques under Comparison

We compared CAUT against two state-of-the-art testing techniques–AFL [13] and CUTE [3]. AFL is a fuzz testing framework. Given an objective program under testing, AFL implements an interface that obtains its code coverage and performs fuzz testing of the program. Specifically, AFL conducts coverage-guided, mutation-based gray-box fuzzing of the benchmarks.

CUTE is a dynamic symbolic execution tool. Like many symbolic execution tools, CUTE explores the possible execution space of an entire program by solving path con-

straints. For every possible execution path, CUTE calculates whether it is solvable, and if so, generate the input(s). We adopt Microsoft's Z3 solver as the constraint solver for CUTE.

Metrics

We use two metrics to evaluate the effectiveness and efficiency of the testing techniques:

1. Hit rate. The hit rate is the percentage of target program paths hit by tests generated to the total number of target program paths. The higher the hit rate, the better the testing technique.
2. Time/iterations. For each target program path, we count the time spent when it is first hit, or the number of iterations (i.e. how many inputs have ever been produced). The shorter the time spent, or the fewer the iterations, the more efficient the testing technique.

To answer the two research questions, we compared the three techniques in their hit rates, the time spent and the numbers of iterations. The experiment was performed on x86 Ubuntu 16.04 LTS virtual machine. Due to the randomness of test input generation, we repeated the experiment five times and calculated the mean values of each technique.

5.1.2. Result Analysis

CAUT vs. AFL

Let AFL be used for 24 h. In 24 h, AFL generated millions of test cases and triggered 67 out of 92 target program paths, achieving a hit rate of 72.8%. During 24 h, CAUT triggered all of the 67 target program paths triggered by AFL, and also triggered another 10 target program paths, achieving a hit rate of 83.7%. Therefore, CAUT outperforms AFL in the hit rate.

For the ten program paths, we recorded CAUT's time spent and numbers of iterations, as Table 3 shows. Note that the ten target program paths are not hit by AFL but only by CAUT. Indeed, after running for an hour or even a few minutes, CAUT can hit many target program paths that cannot be hit by running AFL for 24 h. Meanwhile, CAUT and AFL are of similar speed in generating tests (about 30 iterations per second).

Sixty seven program paths are hit by both techniques successfully. We recorded the number of iterations of CAUT and AFL. CAUT is 82.5–99.7% less than AFL in the number of iterations, with an average of 97.2%. That is, CAUT produces much useful input than AFL. As the two techniques are of similar speeds in generating test inputs, it can be seen that CAUT outperforms AFL by 36× in the efficiency on average.

One reason for this is that CAUT has the assistance of Tabu search and does not need to explore a large number of redundant execution paths in the program. Furthermore, during the process of exploring the search space, AFL cannot distinguish the quality of the inputs, while CAUT can choose promising ones with low costs. Therefore, let the cost of calculation be not very large. CAUT can greatly increase the hit rate and efficiency of exploring target program paths.

**Table 3.** Results of CAUT on 10 target program paths not hit by AFL within 24 h.

| Target | CAUT Hit Time | CAUT Iterations | AFL Hit Time | AFL Iterations |
|---|---|---|---|---|
| FloatingDecimal-readString-1 | 49 min 55 s | 105.4 k | >24 h | 2.87 M |
| FloatingDecimal-readString-2 | 2 min 21 s | 3060 | >24 h | 2.87 M |
| FloatingDecimal-parseHexString-1 | 13 min 59 s | 25.8 k | >24 h | 2.98 M |
| FloatingDecimal-parseHexString-2 | 25 min 45 s | 55.2 k | >24 h | 2.98 M |
| StrictMath-remPiOver2-1 | 1 min 46 s | 3499 | >24 h | 1.74 M |
| StrictMath-remPiOver2-2 | 2 min 40 s | 5244 | >24 h | 1.74 M |
| StrictMath-sinh-1 | 1 min 38 s | 2969 | >24 h | 1.74 M |
| BigInteger-bitLength-1 | 10 min 32 s | 22.4 k | >24 h | 2.21 M |
| BigInteger-divide-1 | 1 h 12 min | 130.0 k | >24 h | 2.21 M |
| BigInteger-Init-1 | 56 s | 1184 | >24 h | 1.79 M |

CAUT vs. CUTE

CUTE has a general requirement that the program's input needs to be symbolized. We keep 26 target program paths meeting the requirements for evaluation. We set the time budget for testing of each benchmark to 20 min. The experimental results are shown in Table 4. It can be seen that CAUT's hit rate is better CUTE after 5 min and continues to increase, and CUTE only triggered 12 of the target program paths.

A further analysis of the results shows that, for most of programs, CUTE produces the final results or exits with errors within 2 min (e.g., CUTE cannot process library functions). CUTE fails in generating test inputs for some programs within 20 min due to the path explosion problem. Four target program paths are not hit by CUTE because of the incomplete searches by symbolic execution. This shows that CUTE's ability will be greatly bounded by constraint solving when facing real-world programs—CUTE's strict requirements on inputs make it not suitable for analyzing many programs. Comparatively, CAUT is both effective and efficient in hitting specific target program paths–its hit rate is higher than that of the coverage-guided fuzzing tool AFL; it is also more effective than the symbolic execution tool CUTE.

**Table 4.** Comparison of hit rates between CAUT and CUTE over time.

| Technique | Time | | | |
|---|---|---|---|---|
| | 2 min | 5 min | 10 min | 20 min |
| CAUT | 31.5% | 66.2% | 79.2% | 96.2% |
| CUTE | | | 46.2% | |

*5.2. Results to RQ2*

Here we describe the testing process in detail by the memory allocation function kmalloc from earlier Linux kernel.

5.2.1. Settings

Subject Program

We use the kmalloc function from earlier Linux kernel release as the experimental source code. The kmalloc function uses bucket allocation algorithm. First, the function searches an array of predefined bucket lists that can fit the allocation unit from 32 bytes to 4096 bytes. If the appropriate bucket list is not found, the function will then try to use page allocation functions for retrieving a sequence pages of memory to fit the request, and otherwise the function will search every allocation descriptor in the bucket list for free space and return the address of first free space it found.

When all allocation descriptors in the list are out of memory, the function will check a global variable free_bucket_desc for an unused allocation descriptor. If there is no unused

descriptor, the function will request one 4096-byte page from system and initialize the page to 256 allocations descriptors and add them to free_bucket_desc. After finding and adding an unused descriptor to the bucket list, the function requests another 4096-byte page, associates the unused descriptor and the page of memory and finally returns the free memory space to the caller of kmalloc. The size of kmalloc is about 1200 lines.

Test Preprocessing

First, in order to manage all variables that can affect the execution paths of the program, we transform all global variables to function parameters. Then, certain mock functions are introduced instead of those functions that cannot be managed by the testing tool but may affect the execution results. To evaluate the ability for both CUTE and CAUT, we injected three faults into kmalloc under test and checked if these were detected by the algorithms we evaluated. The fault form we injected here was assert(0). The first fault injected was placed in a mock function, which was used to simulate page allocation.

The second one was placed in the code of processing page allocation fault. The last one was placed in the branch that handles memory allocation requests larger than 4096 bytes.

5.2.2. Result Analysis

Table 5 shows some interesting results. CUTE found the first two faults more quickly than CAUT. However, CUTE missed the third one, while CAUT caught it in short iterations. In this experiment, we found the test input of parameter len of kmalloc from CUTE was always below 32, and for every iteration almost all CUTE attempted to do was to extend the bucket list for 32-byte allocation units. It went into a situation of selecting paths from an unbounded growth of alternatives targeting at comparing if there existed free spaces in bucket list and fell into searching a local space.

The first and second faults were placed in the paths that all requests in the range from 0 to 4096 could go through, and more importantly the path had an unbounded set of alternative successive paths on comparing list items, while the third fault was far from this path. The depth first algorithm always attempted to search at the end of the current execution path, which is why CUTE could find the first two faults quickly but could not find the third one.

**Table 5.** Experiment results.

| Algorithm under Test | Sample | Fault Found | Iteration Passed |
|---|---|---|---|
| CUTE | $S_1$ | T | 3 |
| | $S_2$ | T | 4 |
| | $S_3$ | F | N/A |
| CAUT | $S_1$ | T | 33 |
| | $S_2$ | T | 42 |
| | $S_3$ | T | 9 |

The first path CAUT found was the same as the one CUTE found. Then, according to the tabu search we adopted, CAUT realized alternative paths related to input parameter len that had higher priority. Thus, CAUT would attempt to generate different values of len so as to not result in the situation that CUTE met. This experiments showed that our path selection algorithm can explore more paths in the search space within a reasonable time.

We also used SGLIB library [3] as the experimental source codes. To evaluate the ability of finding bugs for both algorithms, we planted an artificial fault by assert(0) in the programs under test. Figure 8 shows a number of runs needed by the CUTE and CAUT algorithms, respectively. The horizontal line above in Figure 8 denotes the maximum runs; here, the number is set to one million. A point in the lines of Figure 8 represents the fault located after a number of runs for a fixed size of program.

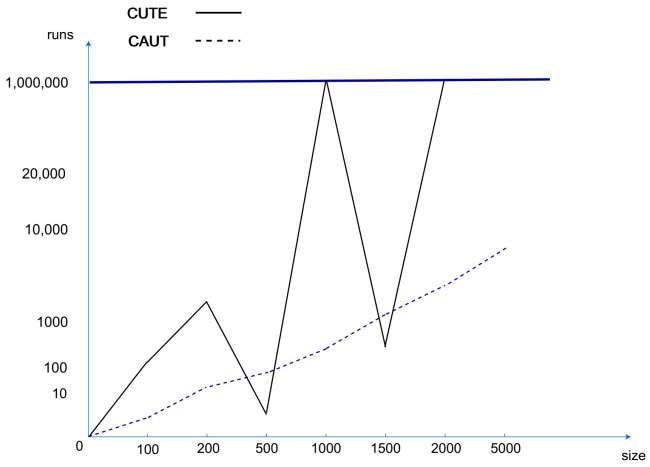


**Figure 8.** Experimental results.

The line that represents our algorithm grows steadily with the increase of size of programs while the one representing the CUTE algorithm rises and falls dramatically. In some cases, CUTE runs out of the maximum number because of its search strategy. This shows that the path selection algorithm we designed performed very well in these situations, and it can avoid falling into a local search space without reducing the ability of finding bugs.

## 6. Related Work

Automating unit testing is a great challenge and is still a dream for most software vendors. One key obstacle for achieving automating unit testing is how to generate appropriate test inputs. Many methodologies [4,14–16] have been devised in recent decades. Symbolic execution [2,17–19] is a traditional approach in static test-case generation, which was initially proposed by James C. King in 1976. By this methodology, a program is analyzed to build an execution tree without actually executing the program. The result tree is complete enough to be explored to find possible bugs.

Every path along the tree is a set of constraints that can affect the execution of the program. By using a theorem prover, the satisfiability of constraints can be reasoned [20]. Unfortunately, due to the limitations of theorem provers, a constraint containing statements, such as function calls, is not solvable. The exclusive use of symbolic execution cannot process pointers or complex data structures. For example, Symstra [21] cannot handle such code that contains array indexing with variables.

Dynamic test-case generation [22] is a more practicable approach. It can be treated as a combined technique to an extent by applying static analysis, symbolic execution and constraint solving. It executes the target program repeatedly, usually starts with random inputs and then collects symbolic constraints along the real executing path. By using a constraint solver, these collected constraints are solved to infer alternative sets of inputs that direct the next execution paths.

One mainstream of dynamic test generation is adaptive random testing, which aims to more evenly spread the test cases over the input domain. [23,24]. Cristian Cadar et al. devised a dynamic bug-finding method [5,25,26]. That method is based on the observation that code can generate its own test cases at run-time by a combination of symbolic and concrete execution. Its symbolic execution has a special feature that is bit-level symbolic execution. Based on this method, a prototype EGT system [25] was developed and applied to real programs.

DART [4], abbreviation of Directed Automated Random Testing, proposed by Patrice Godefroid et al., is a typical tool using dynamic test-case generation technique. The goal of DART is to systematically execute all feasible program paths to detect latent runtime errors. It adopts an improved random testing technique to achieve better coverage. Systematic

Modular Automated Random Testing (SMART) [27] is an extension to DART. SMART extends DART by testing functions in isolation, encoding test results as function summaries expressed using input preconditions and output post-conditions and then re-using those summaries when testing higher-level functions.

The motivation is to achieve a scalable DART, as it is clear that systematically executing all feasible paths does not fit large programs. CUTE [3] is another tool on dynamic test-case generation. It mixes dynamic concrete and symbolic execution—called concolic execution, which is good for dealing with pointers and complex data types and has some optimized constraint solving algorithms. It supports bounded depth-first search to avoid search space explosion.

Hybrid Concolic Testing [28,29] extends the CUTE work, where random search and bounded depth-first search are combined. A bounded depth-first search algorithm attempts to explore all neighborhoods of the current paths exhaustively, while a random search algorithm has the ability of reaching deep program branches quickly. The branch coverage by using this method has a notable boost. Many other concolic unit testing tools do exist, such as those in [30,31]. Compared to Hybrid CUTE, the difference is the path selection. We use the tabu strategy instead of random search, and this is an orthogonal improvement for scalability without reducing the ability to find bugs.

A key problem of dynamic test-case generation technique is the selection of alternative paths. Although a bounded depth-first algorithm can improve the branch coverage observably, it is not helpful to find bugs quickly, especially in large and complex program units. The depth-first path selector has to explore every branch exhaustedly in a fixed and unpredictable order, as if exploring in the dark. In some extreme situations, such as an infinite loop, depth-first search may cause a path set explosion and lose other paths as in the given experiment.

To an extent, bounded depth-first search in program paths can be almost classified as a random algorithm, because the locations of targets (bugs) as well as the paths are completely unpredictable to the search algorithm. Fortunately, by some statistical work [6], there is evidence to show that the bugs appearing in programs are not distributed completely randomly. The heuristical algorithm introduced in this paper benefits from those previous works.

The algorithm in this paper combined with control flow graph analysis is good at selecting latent error-relevant paths as well as avoiding path set explosions. The tabu search strategy directs coming dynamic executions to alternative paths where bugs may appear with high probability. The resulting set of test cases generated by our approach was designed to indicate bugs earlier with greater code coverage.

Tabu search is also frequently used for test generation. Tabu search is a local heuristic method based on the neighborhood. It prohibits already visited solutions and others through user-provided rules, significantly enhancing the performance of searches. Tabu search has been applied to test generation and/or prioritization.

Díaz et al. designed a testing technique that combines Tabu search with the Korel chaining approach to obtain a specific coverage in software testing [32]. They then presented a tabu search metaheuristic algorithm for generating structural software tests. The test generator has a cost function for intensifying the search and another for diversifying the search. It also combines the use of memory with a backtracking process to avoid becoming stuck in local minima [33].

Perumal et al. combined Cuckoo and Tabu Search in test data generation [34]. Srivastava et al. presented a search algorithm for the automatic test generation and prioritization through a clustering technique of data mining [35]. They also presented an approach for test data generation using the cuckoo search and tabu search algorithms (CSTS) [36]. It uses the cuckoo algorithm for converging to the solution in minimal time, and uses the tabu mechanism of backtracking from local optima by Lévy flight.

Zamli et al. proposed a hybrid t-way test generation strategy HHH. HHH adopts Tabu search as the meta-heuristic and leverages four low level meta-heuristics. HHH is

able to adaptively select the most suitable meta-heuristic at any particular time [10]. Yu et al. generated test cases based on tabu search and genetic algorithm, aiming at improving the effectiveness of generating test case for algebraic specification [9].

To ensure the quality of current highly configurable software systems, Hasan et al. presented a search-based strategy to generate constrained interaction test suites to cover all possible combinations [7]. The strategy generates the set of all possible t-tuple combinations, and then filters out the set by removing forbidden t-tuples. The strategy also utilizes a mixed neighborhood tabu search to construct optimal or near-optimal constrained test suites.

Rathore et al. generated test-data by combining genetic and tabu search algorithms [37]. The approach uses genetic algorithm to generate test-data, supplemented by a tabu search heuristic in mutation step. It also incorporates backtracking process that moves search away from local optima. Sharma et al. optimized the cost of testing using Tabu search, which provides maximum code coverage along with an Aspiration criteria of Tabu Search in order to optimize the cost and generate a minimum cost path with maximum coverage [38].

Comparatively, our approach combines dataflow analysis with dynamic symbolic execution and heuristically searches for program path space based on the tabu search strategy and the program fault statistics. These techniques are combined together, enabling us to find more possible errors in programs with fewer test cases; it also scales well w.r.t. program sizes and, thus, can be applied to real-world software systems with reasonable costs.

## 7. Conclusions

In this paper, we proposed an interesting approach towards a scalable path search for the automated test-case generation of C programs. This approach combines dataflow analysis and effective path selection algorithm with a dynamic symbolic execution framework together to achieve better efficiency for finding bugs. The path-selection model adopted in this paper has two advantages. First, based on the path-evaluation function we designed, the search process can avoid falling into a sub-space without visiting other parts of the space; second, it is prone to exposing more program faults with less test cases generated because the path analysis engine searches for those sub-spaces that are more likely to hide faults.

The preliminary experiments are encouraging. The CAUT tool based on our approach was developed, and we will attempt further experiments and improve the performance of the tool at the same time. We are also investigating a new path-selection model that is based on a dynamic statistics model instead of the static one that we developed here, and this could help to locate bugs more precisely.

**Author Contributions:** Conceptualization, E.M. and X.W.; methodology, E.M.; software, E.M.; validation, E.M., X.W. and X.F.; formal analysis, E.M.; investigation, E.M.; resources, X.W.; data curation, E.M.; writing—original draft preparation, E.M.; writing—review and editing, X.W. and X.F.; visualization, X.W.; supervision, X.W.; project administration, E.M. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Utting, M.; Legeard, B. *Practical Model-Based Testing—A Tools Approach*; Morgan Kaufmann: Burlington, MA, USA, 2007.
2. King, J.C. Symbolic Execution and Program Testing. *Commun. ACM* **1976**, *19*, 385–394. [CrossRef]
3. Sen, K.; Marinov, D.; Agha, G. 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, Lisbon, Portugal, 5–9 September 2005; Wermelinger, M., Gall, H.C., Eds.; ACM: New York, NY, USA, 2005; pp. 263–272.

4. Sen, K. DART: Directed Automated Random Testing. In *Hardware and Software: Verification and Testing, Proceedings of the 5th International Haifa Verification Conference, HVC 2009, Haifa, Israel, 19–22 October 2009*; Revised Selected Papers; Lecture Notes in Computer Science; Namjoshi, K.S., Zeller, A., Ziv, A., Eds.; Springer: New York, NY, USA, 2009; Volume 6405, p. 4. [CrossRef]
5. Cadar, C.; Engler, D.R. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Model Checking Software, Proceedings of the 12th International SPIN Workshop, San Francisco, CA, USA, 22–24 August 2005*; Lecture Notes in Computer Science; Godefroid, P., Ed.; Springer: New York, NY, USA, 2005; Volume 3639, pp. 2–23.
6. Beizer, B. *Software Testing Techniques*, 2nd. ed.; Van Nostrand Reinhold: New York, NY, USA, 1990.
7. Hasan, I.H.; Ahmed, B.S.; Potrus, M.Y.; Zamli, K.Z. Generation and Application of Constrained Interaction Test Suites Using Base Forbidden Tuples with a Mixed Neighborhood Tabu Search. *Int. J. Softw. Eng. Knowl. Eng.* **2020**, *30*, 363–398. [CrossRef]
8. Rathee, N.; Chhillar, R.S. Model Driven Approach to Secure Optimized Test Paths for Smart Samsung Pay using Hybrid Genetic Tabu Search Algorithm. *Int. J. Inf. Syst. Model. Des.* **2018**, *9*, 77–91. [CrossRef]
9. Yu, B.; Qin, Y. Generating test case for algebraic specification based on Tabu search and genetic algorithm. *Clust. Comput.* **2017**, *20*, 277–289. [CrossRef]
10. Zamli, K.Z.; Alkazemi, B.Y.; Kendall, G. A Tabu Search hyper-heuristic strategy for t-way test suite generation. *Appl. Soft Comput.* **2016**, *44*, 57–74. [CrossRef]
11. Necula, G.C.; McPeak, S.; Rahul, S.P.; Weimer, W. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction, Proceedings of the 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, 8–12 April 2002*; Lecture Notes in Computer Science; Horspool, R.N., Ed.; Springer: New York, NY, USA, 2002; Volume 2304, pp. 213–228.
12. Beyer, D.; Chlipala, A.; Henzinger, T.A.; Jhala, R.; Majumdar, R. Generating Tests from Counterexamples. In Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), Edinburgh, UK, 23–28 May 2004; Finkelstein, A., Estublier, J., Rosenblum, D.S., Eds.; IEEE Computer Society: Piscataway, NJ, USA, 2004; pp. 326–335.
13. Gutmann, P. Fuzzing Code with AFL. *Login Usenix Mag.* **2016**, *41*, 11–14.
14. Csallner, C.; Smaragdakis, Y.; Xie, T. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.* **2008**, *17*, 1–37. [CrossRef]
15. Tillmann, N.; de Halleux, J. Pex-White Box Test Generation for .NET. In Proceedings of the Tests and Proofs—2nd International Conference, TAP 2008, Prato, Italy, 9–11 April 2008; Beckert, B., Hähnle, R., Eds.; Springer: New York, NY, USA, 2008; Volume 4966, pp. 134–153.
16. Zhang, C.; Yan, Y.; Zhou, H.; Yao, Y.; Wu, K.; Su, T.; Miao, W.; Pu, G. Smartunit: Empirical evaluations for automated unit testing of embedded software in industry. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, 27 May–3 June 2018; Paulisch, F., Bosch, J., Eds.; ACM: New York, NY, USA, 2018; pp. 296–305.
17. Braione, P.; Denaro, G.; Mattavelli, A.; Pezzè, M. Combining symbolic execution and search-based testing for programs with complex heap inputs. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, 10–14 July 2017; Bultan, T., Sen, K., Eds.; ACM: New York, NY, USA, 2017; pp. 90–101.
18. de Boer, F.S.; Bonsangue, M.M. Symbolic execution formally explained. *Formal Asp. Comput.* **2021**, *33*, 617–636. [CrossRef]
19. He, W.; Shi, J.; Su, T.; Lu, Z.; Hao, L.; Huang, Y. Automated test generation for IEC 61131-3 ST programs via dynamic symbolic execution. *Sci. Comput. Program.* **2021**, *206*, 102608. [CrossRef]
20. Ruan, H.; Zhang, J.; Yan, J. Test Data Generation for C Programs with String-Handling Functions. In Proceedings of the Second IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE 2008, Nanjing, China, 17–19 June 2008; pp. 219–226.
21. Xie, T.; Marinov, D.; Schulte, W.; Notkin, D. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, 4–8 April 2005; Lecture Notes in Computer Science; Halbwachs, N., Zuck, L.D., Eds.; Springer: New York, NY, USA, 2005; Volume 3440, pp. 365–381.
22. Do, T.; Khoo, S.; Fong, A.C.M.; Pears, R.; Quan, T.T. Goal-oriented dynamic test generation. *Inf. Softw. Technol.* **2015**, *66*, 40–57. [CrossRef]
23. Huang, R.; Sun, W.; Xu, Y.; Chen, H.; Towey, D.; Xia, X. A Survey on Adaptive Random Testing. *IEEE Trans. Softw. Eng.* **2021**, *47*, 2052–2083. [CrossRef]
24. Huang, R.; Chen, H.; Sun, W.; Towey, D. Candidate test set reduction for adaptive random testing: An overheads reduction technique. *Sci. Comput. Program.* **2022**, *214*, 102730. [CrossRef]
25. Cadar, C.; Ganesh, V.; Pawlowski, P.M.; Dill, D.L.; Engler, D.R. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* **2008**, *12*, 1–38. [CrossRef]
26. Yang, J.; Sar, C.; Twohey, P.; Cadar, C.; Engler, D.R. Automatically Generating Malicious Disks using Symbolic Execution. In Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006), Berkeley, CA, USA, 21–24 May 2006; pp. 243–257.
27. Godefroid, P. Compositional dynamic test generation. In Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, 17–19 January 2007; Hofmann, M., Felleisen, M., Eds.; ACM: New York, NY, USA, 2007; pp. 47–54.

28. Majumdar, R.; Sen, K. Hybrid Concolic Testing. In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, 20–26 May 2007; pp. 416–426.
29. Godboley, S.; Dutta, A.; Mohapatra, D.P. Java-HCT: An approach to increase MC/DC using Hybrid Concolic Testing for Java programs. In Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, 11–14 September 2016; Ganzha, M., Maciaszek, L.A., Paprzycki, M., Eds.; IEEE: Piscataway, NJ, USA, 2016; Volume 8, pp. 1709–1713.
30. Kim, Y.; Choi, Y.; Kim, M. Precise concolic unit testing of C programs using extended units and symbolic alarm filtering. In Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, 27 May–3 June 2018; Chaudron, M., Crnkovic, I., Chechik, M., Harman, M., Eds.; ACM: New York, NY, USA, 2018; pp. 315–326.
31. Ahmadi, R.; Jahed, K.; Dingel, J. mCUTE: A Model-Level Concolic Unit Testing Engine for UML State Machines. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, 11–15 November 2019; pp. 1182–1185.
32. Díaz, E.; Tuya, J.; Blanco, R. Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search. In Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), Montreal, QC, Canada, 6–10 October 2003; pp. 310–313.
33. Díaz, E.; Tuya, J.; Blanco, R.; Dolado, J.J. A tabu search algorithm for structural software testing. *Comput. Oper. Res.* **2008**, *35*, 3052–3072. [CrossRef]
34. Perumal, K.; Ungati, J.M.; Kumar, G.; Jain, N.; Gaurav, R.; Srivastava, P.R. Test Data Generation: A Hybrid Approach Using Cuckoo and Tabu Search. In Proceedings of the Swarm, Evolutionary, and Memetic Computing—Second International Conference, SEMCCO 2011, Visakhapatnam, India, 19–21 December 2011; Panigrahi, B.K., Suganthan, P.N., Das, S., Satapathy, S.C., Eds.; Springer: New York, NY, USA, 2011; Volume 7077, pp. 46–54.
35. Srivastava, P.R.; Vijay, A.; Barukha, B.; Sengar, P.S.; Sharma, R. An Optimized technique for Test Case Generation and Prioritization Using 'Tabu' Search and 'Data Clustering'. In Proceedings of the 4th Indian International Conference on Artificial Intelligence, IICAI 2009, Tumkur, Karnataka, India, 16–18 December 2009; Prasad, B., Lingras, P., Ram, A., Eds.; Springer: New York, NY, USA, 2009; pp. 30–46.
36. Srivastava, P.R.; Khandelwal, R.; Khandelwal, S.; Kumar, S.; Ranganatha, S.S. Automated Test Data Generation Using Cuckoo Search and Tabu Search (CSTS) Algorithm. *J. Intell. Syst.* **2012**, *21*, 195–224. [CrossRef]
37. Rathore, A.; Bohara, A.; Prashil, R.G.; Prashanth, T.S.L.; Srivastava, P.R. Application of genetic algorithm and tabu search in software testing. In Proceedings of the 4th Bangalore Annual Compute Conference, Compute 2011, Bangalore, India, 25–26 March 2011; Shyamasundar, R.K., Shastri, L., Eds.; ACM: New York, NY, USA, 2011; p. 23.
38. Sharma, A.; Jadhav, A.; Srivastava, P.R.; Goyal, R. Test Cost Optimization Using Tabu Search. *J. Softw. Eng. Appl.* **2010**, *3*, 477–486. [CrossRef]