Fitness-Guided Path Exploration in Dynamic Symbolic Execution

Tao Xie[1] Nikolai Tillmann[2] Jonathan de Halleux[2] Wolfram Schulte[2]
[1] Department of Computer Science, North Carolina State University, NC 27695-8206, USA
[2] Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
[1]`xie@csc.ncsu.edu` [2]`{nikolait, jhalleux, schulte}@microsoft.com`

Abstract

Dynamic symbolic execution is a structural testing technique that systematically explores feasible paths of the program under test by running the program with different test inputs to improve code coverage. To address the space-explosion issue in path exploration, we propose a novel approach called Fitnex, a search strategy that uses state-dependent fitness values (computed through a fitness function) to guide path exploration. The fitness function measures how close an already discovered feasible path is to a particular test target (e.g., covering a not-yet-covered branch). Our new fitness-guided search strategy is integrated with other strategies that are effective for exploration problems where the fitness heuristic fails. We implemented the new approach in Pex, an automated structural testing tool developed at Microsoft Research. We evaluated our new approach by comparing it with existing search strategies. The empirical results show that our approach is effective since it consistently achieves high code coverage faster than existing search strategies.

1 Introduction

Structural software testing aims at achieving full or at least high code coverage such as statement and branch coverage of the program under test. A passing test suite that achieves high code coverage provides high confidence of the dependability of the program under test. The problem of testing for finding bugs can also often be reduced to the problem of structural testing with the goal of covering all statements as follows. A bug can often be seen as a special `error` statement, which may be guarded by a condition. For example, when a test oracle is expressed by an assertion of a condition, then covering the negated condition witnesses a bug. Note that an assertion is used here in a general sense; the behavior reflected by the assertion can be expressed using contracts [12] or as part of parameterized unit tests [16].

```
public bool TestLoop(int x, int[] y) {
1   if (x == 90) {
2     for (int i = 0; i < y.Length; i++)
3        if (y[i] == 15)
4           x++;
5     if (x == 110)
6        return true;
    }
7   return false;
}
```

Figure 1. An example method under test.

Random testing [13] is one of the most commonly used techniques for software testing primarily due to its ease of implementation and the marginal overhead in choosing inputs. However, the random testing technique is not effective when inputs needed to reach a given statement are very specific and if there is only little chance of randomly finding them in the input space. For example, given the method under test shown in Figure 1, to cover the statement in Line 6, the integer value of argument `x` needs to be exactly `90` and the array elements of argument `y` needs to include exactly 20 elements whose values are `15` and 0 or more other elements whose values are not `15`. Although the example method shown in Figure 1 is specifically contrived to illustrate the issues, such similar cases commonly occur in real-world programs under test, posing challenges for automated test generation.

To address the issues faced by random testing, *dynamic* symbolic execution (DSE) [3, 6, 14] (also called directed random testing [6] or concolic testing [14]) has been recently proposed. DSE is a variation of symbolic execution [4, 10], which leverages observations from concrete executions. It executes the program under test for given inputs, while performing a symbolic execution in parallel to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution. The conjunction of all symbolic constraints along a path is called the *path condition*. DSE is typically performed iteratively to systematically increase code coverage. In each iteration, after applying DSE to an already explored path, a search strategy decides on a branching node in the path to *flip*[1].

---

[1] The execution paths span an execution tree, where a branching

Intuitively, flipping a branching node in a path means to construct a new path that shares the prefix to the node with the old path, but then deviates and takes a different branch. Whether such a flipped path is feasible is checked by building a constraint system representing the flipped path's feasibility. If a constraint solver can determine that the constraint system is satisfiable, and if the solver can compute a satisfying assignment, then, by construction, we have found a new test input that will execute along the flipped path. For the example method shown in Figure 1, assuming that the initial argument values `x` and `y` are `0` and an array `{0}`, respectively, then the false branch of Line 1 is taken and the path condition is `(x != 90)`. Negating the constraint for the branching node in Line 1 (i.e., flipping the branching node) produces a new constraint system: `(x == 90)`. Solving the constraint system produces a new test input (`x` as `90` and `y` as an array `{0}`) to cover the true branch of Line 1.

Code inspection reveals that covering the true branch of Line 5 (called the test target) needs exactly 20 executions of the true branch of Line 3 inside the loop. However, applying DSE to hunt for such a case (thus covering the test target) faces significant challenges. First, the number of loop iterations in Lines 2-4 depends on the length of the array `y`, which can range from 0 to $2^{31}-1$. A breadth-first or depth-first search strategy would not be able to explore and cover the test target within a reasonable amount of time. Second, even when we put a bound such as 20 for the loop iterations (given that we need at least 20 loop iterations to achieve the test target coverage), the number of paths for the loop iterations is still 2^{20}, which is too large in practice.

The example method in Figure 1 illustrates a general exploration problem for DSE: to explore and cover a program that contains one or more branches with relational conditions (here, `(x == 110)`), where the operands are scalar values (integers or floating-point numbers) that are computed based on control-flow decisions connected to the test inputs through data flow (here, `if (y[i]==15) x++;`). Such indirect relationships between the program conditions and the test inputs pose a challenge for search strategies of exploring relevant paths.

To tackle this exploration problem, we propose a novel approach called Fitnex for guided path exploration in DSE to achieve test target coverage quickly. The guided search provided by our Fitnex approach alleviates issues encountered by previous DSE approaches with (bounded) exhaustive search [3, 6, 14] or random search [2]. In particular, our approach assigns to already explored paths *fitness values* computed by program-derived fitness functions. (Fitness functions have been traditionally used in search-based test generation [11].) A *fitness function* measures how close an explored path is in achieving test target coverage. A *fitness gain* is also measured for each explored branch: a higher fitness gain is given to a branch if flipping a branching node for the branch in the past helped achieve better fitness values. Then during path exploration, our Fitnex strategy would prefer to flip a branching node whose corresponding branch has a *higher fitness gain* in a previously explored path with a *better fitness value*.

The core Fitnex strategy is effective for only certain exploration problems — those amenable to fitness functions. To address the issue, we integrate the Fitnex strategy with other search strategies, which work well for other types of exploration problems. Our integration of the Fitnex and other strategies achieves the effect of getting the best of both in practice.

This paper makes the following main contributions:

- We propose a fitness-guided strategy for path exploration in (dynamic) symbolic execution. To the best of our knowledge, our strategy is the first one that uses fitness values to directly and effectively guide path exploration.
- We integrate the fitness-guided strategy with other strategies to address exploration problems where the fitness heuristic fails.
- We implement the proposed Fitnex strategy in Pex [15], an automated structural testing tool for .NET developed at Microsoft Research. The Fitnex implementation has been released as open source in the Pex Extensions project webpage[2]. The Fitnex strategy has been integrated into the default search strategy in Pex, which consists of a combination of various individual strategies. Pex has been previously used internally at Microsoft to test core components of the .NET architecture and has found serious bugs [15]. Recent public releases of Pex [15] (since 17 September 2008, with thousands of download counts) have incorporated our new approach as Pex's default test generation mechanism.
- We evaluate our approach on 30 benchmarks created to reflect various typical exploration problems encountered in testing real, complex C# code under test. The evaluation results show that our new approach consistently achieves higher code coverage faster than other strategies, including random and breadth-first strategies.

The rest of the paper is organized as follows. Section 2 presents our illustrative example. Section 3 presents dynamic symbolic execution in Pex. Section 4 presents our new Fitnex strategy for fitness-guided path exploration and its integration with other search strategies. Section 5

---

node represents an instance of a conditional branch in the code. An `if-then-else` statement in the code can correspond to multiple branching nodes in the execution tree, and even along a path in the execution tree. For example, when a loop contains an `if-then-else`, it may be executed multiple times in the path.

[2] `http://www.codeplex.com/Pex`

presents the evaluation results. Section 6 discusses related work. Section 7 discusses research issues and future work, and Section 8 concludes.

2 Example

We use the example method shown in Figure 1 to illustrate our Fitnex strategy. In particular, we explain how our Fitnex strategy helps cover test targets such as the true branch of Line 5. During DSE's path exploration [3, 6, 14], a key decision in each iteration is which branching node to flip next. Recall that flipping a branching node in a path means to construct and decide the satisfiability of a constraint system that represents all conditions in the path prefix before the branching node to flip, conjuncted with the negation of the condition of the branching node to flip.

Fitness computation for a path. We introduce fitness functions to select the most promising path along which a branching node should be flipped. Our fitness functions are derived from the boolean binary predicates [18] that appear in the program under test. Fitness functions compute fitness values, reflecting how close a path's execution is to covering the test target (e.g., a not-yet-covered branch). Exploration then prefers the fittest paths, i.e. paths that are closest to covering the test target. For example, for the predicate `(x == 110)` in Line 5, the fitness function is "`if (|110 - x| == 0) then 0 else |110 - x|`". The smaller a fitness value is, the closer (fitter or better) the path's execution is to covering the test target. The fitness value of 0 represents the case where the test target is covered.

Assume that five existing test inputs `Tests 0-4` (generated via five iterations of path exploration) explored `Paths 0-4` as listed below.

```
Test 0:
   TestLoop(0, new int[] {0});
    Path 0: 1F
Test 1:
   TestLoop(90, new int[] {0});
    Path 1: 1T, 2T, 3F, 2F, 5F
Test 2:
   TestLoop(90, new int[] {15});
    Path 2: 1T, 2T, 3T, 2F, 5F
Test 3:
   TestLoop(90, new int[] {15, 0});
    Path 3: 1T, 2T, 3T, 2T, 3F, 2F, 5F
Test 4:
   TestLoop(90, new int[] {15, 15});
    Path 4: 1T, 2T, 3T, 2T, 3T, 2F, 5F
```

A path is denoted by the sequence of line numbers for taken branches followed by `T` or `F` to represent true and false branches, respectively. Each item in the sequence represents a branching node.

Recall that the true branch of Line 5 is the test target. Based on the fitness function for the test target, the fitness values for `Paths 0-4` are the worst (largest) fitness value (due to not even reaching the location of the test target), `20` (`|110 - 90|`), `19` (`|110 - 91|`), `19` (`|110 - 91|`), and `18` (`|110 - 92|`). Because the fitness value of `Path 4` (being `18`) is better (e.g., smaller) than those of `Paths 0-3`, in the subsequent iteration, `Path 4` is given higher priority over the other four paths for branching-node flipping.

Fitness-gain computation for a branch. We next describe how we give higher flipping priority to a more promising branching node in a path. We first compute the *fitness gain* of a branch. The fitness gain reflects how much the fitness value has improved across paths after a branching node for the branch was flipped in the past. For example, when we flipped the branching node for the false branch of Line 3 `(y[i] == 15)` in `Path 1` to the true branch, we derive `Path 2`, whose fitness value has improved from `20` to `19`, i.e., the fitness gain is `1`. The same fitness gain, namely `1`, is achieved if flipping a branching node for the same branch from `Path 3` to `Path 4`. Therefore, the computed fitness gain for the false branch of Line 3 is `1` (on average). Note that a fitness gain can be negative indicating undesirable consequence. For example, the computed fitness gain for the true branch of Line 3 is `-1`, because flipping a branching node for this branch to the false branch could lead to fitness gain of `-1` (on average).

When we flip the branching node for the false branch of Line 2 (loop predicate) of `Path 2` to the true branch, we unfold the loop, deriving `Path 3` or `Path 4` (depending on whether the constraint solver assigns the additional array element with 15). Assuming that `Path 4` is derived, since `Path 4`'s fitness value is `18` (with a fitness gain of `1`), the computed fitness gain for the false branch of Line 2 is `1`.

We assign a composite fitness value to each branching node for a branch b in an explored path p as ($F(p)$ - $FGain(b)$), where $F(p)$ is the fitness value for p and $FGain(b)$ is the fitness gain for b. We prioritize branching nodes for flipping among all the branching nodes (from the explored paths) based on these nodes' composite fitness values: the lower a node's composite fitness value is, the higher priority the node has. For example, we give the highest priority to flip the branching node for the false branch of Line 2 in `Path 4` since this node has the lowest composite fitness value `17`, being `(18 - 1)`. Such a node flipping unfolds the loop, helping get closer to the coverage of the test target. Eventually, after a relatively small number n of more iterations[3] beyond the five iterations/paths (i.e., `Paths 0-4`), our path exploration leads to a path that has fitness value `0`, i.e., covers the test target. Such a small number n of iter-

---

[3] n ranges from 18 to 36 depending on whether the additional array element after each iteration of loop expansion is assigned the value of 15 by the constraint solver (causing to cover the true branch of Line 3). A "smart" constraint solver can potentially assign a historically rewarding value (e.g., 15) to new array elements due to loop-iteration expansion; in this case, n is 18.

```
/*intuitively, $J$ is the set of already analyzed program inputs*/
Set $J := \emptyset$
loop
    Choose program input $i$ such that $i \notin J$
    stop if no such $i$ can be found
    Output $i$
    Execute $i$; record path condition $C$    /*$C(i)$ holds*/
    $J := J \cup C$        /*viewing C as the set $\{i \mid C(i)\}$*/
end loop
```

Figure 2. General iterative DSE algorithm

ations in addition to the five iterations is in sharp contrast to 2^{20}, the bounded search space for path exploration, highlighting the benefits brought by our Fitnex strategy.

3 Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) [3, 6, 14] is a variation of conventional static symbolic execution [4, 10]. DSE executes the program starting with arbitrary inputs, while performing a symbolic execution in parallel to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution. Then a constraint solver is used to compute variations of the previous inputs in order to steer future program executions along different execution paths. In this way, all feasible execution paths will be exercised eventually through such iterations of input or path variations. Figure 2 shows the general iterative DSE algorithm implemented by Pex [15]. A crucial aspect of the iterative DSE algorithm is the choice of the new program inputs i in each loop iteration. This choice decides in which order the different execution paths of the program are enumerated.

When enumerating paths of the example method in Figure 1, one strategy could always choose to unroll the loop further, and never, not even eventually, visit the *true* branch inside the loop. This observation illustrates the need for a *fair choice* between different branches when enumerating execution paths. Furthermore, this observation illustrates the need for a *guided choice* between different branches when enumerating execution paths since both the true branch inside the loop and the loop-unrolling branch are most desirable to explore among different branches.

In practice, it turns out that this choice should not be left to the constraint solver, but that it is more appropriate to leverage structural information about the program and previously executed paths to guide the search, and to provide a fair and guided choice.

To this end, Pex implements a variation of the algorithm in Figure 2. All execution paths of the program belong to its *execution tree*. Each node of this tree, called *branching node*, is an instance of a control-flow point of the program. If the program has loops or nested branches, a single control-flow point (such as a branch) of the program might have several instances (branching nodes) in the execution tree.

Through DSE, Pex explores the reachable portion of this tree one path at a time.[4] In each iteration of Pex's exploration, it selects (for flipping) a branching node from the explored portion of the execution tree where at least one of the node's immediate outgoing branches is not explored. In the process of flipping the selected branching node, Pex forms the next constraint system to solve as

- the conjunction of the constraints leading to the selected branching node,
- conjuncted with the negation of the disjunctions of the constraints of the already explored immediate outgoing branches of the selected branching node.

If the constraint system turns out to be infeasible, the selected branching node is marked as exhausted and discarded.

To guarantee that our algorithm will visit all reachable control-flow points eventually, we need a search strategy that performs a fair and guided choice among all control-flow points. Our new approach described in the next section provides a fitness-guided search strategy in combination with other strategies.

4 Approach

The core of our approach is the Fitnex search strategy guided by fitness values computed with a fitness function (Section 4.1). To deal with program branches not amenable to a fitness function, our approach includes integration of the Fitnex strategy with other search strategies (Section 4.2).

4.1 Fitness-Guided Search Strategy

A fitness function (Section 4.1.1) gives a measurement on how close an explored path is to achieving a test target (e.g., covering a not-yet-covered branch). We compute a fitness value for each already explored path and prioritize these known paths based on their fitness values (Section 4.1.2). We compute a fitness gain for each branch in the program under test and prioritize branching nodes based on their corresponding branches' fitness gains (Section 4.1.3). During path exploration, we give higher priority to flipping a branching node with a better (higher) fitness gain in a path with a better (lower) fitness value (Section 4.1.4).

4.1.1 Fitness Functions for Target Predicates

A common type of test targets can be to satisfy a target predicate in the program under test. For example, we may want

---

[4] Using summarization techniques [5], it is possible to collapse subtrees into nodes.

Table 1. Fitness functions of predicates

Predicate	Fitness function	
	True	False
$F(a == b)$	0	$\lvert a-b \rvert$
$F(a > b)$	0	$(b-a)+K$
$F(a >= b)$	0	$(b-a)$
$F(a < b)$	0	$(a-b)+K$
$F(a <= b)$	0	$(a-b)$

to satisfy the predicate (or its negation) in a conditional so that the not-yet-covered true (or false) branch of the conditional can be covered. Given a target predicate, we measure how *close* its evaluation at runtime is to covering the target predicate with fitness functions [18] as listed in Table 1 (these fitness functions are commonly used in search-based test generation [11]). Column 1 shows the form of a target predicate. Columns 2 and 3 show the fitness function for the target predicate. In particular, the fitness value shown in Column 2 is 0 when the predicate in Column 1 evaluates to true. When the predicate in Column 1 evaluates to false, the expression shown in Column 3 computes the fitness value. In the fitness functions, K is a failure constant (such as 1) and is added when the predicate is false. For example, for a predicate $(a > b)$, if $(a > b)$ evaluates to true, then the fitness value is 0; otherwise, the fitness value is $(b - a) + K$.

Our path-exploration process tries to minimize the fitness values computed by the fitness function for paths being explored. If the fitness value computed for a path is 0, then we cover the target predicate (e.g., covering the target branch). Our approach implemented in Pex does not need to deal with composite predicates (e.g., predicates including logical operator $\&\&$ or $||$), because Pex operates at the .NET instruction level where composite predicates in conditionals are typically decomposed into multiple conditionals with simple predicates at the .NET instruction level.

Besides target predicates for branches (in the program under test) that are not yet covered, we also associate fitness values with non-branching target predicates for boolean binary expressions (in the program under test) whose true or false values have not yet been exercised. The motivating case for covering these target predicates is illustrated below:

```
bool b = (x > y); if (b) { ... }
```

Suppose that the true branch of the conditional "`if (b)`" is our test target and thus `(b)` is our target predicate. However, there exists no good fitness function for a predicate in the form of (*bool*) where *bool* is a boolean variable; that is why we do not list such a predicate type in Table 1. That is, we do not have a good way to measure how *close* the evaluation of (*bool*) is to cover (*bool*) since there are only two outcomes: either covering it (i.e., *bool* being true) or not covering it (i.e., *bool* being false). Our preceding technique addresses this issue by associating a fitness value with the target predicate for the binary boolean expression `(x > y)`. Such fitness guidance helps the exploration to exercise the true value of the expression and subsequently cover the true branch of the conditional.

If a test requirement such as `(x < 0)` at a program execution point is to be satisfied but `(x < 0)` does not appear in a conditional or even a boolean expression in the program, we still can insert an auxiliary statement `if (x < 0) {} else {}`, which simply induces a case split for the search, allowing Fitnex to be applied.

4.1.2 Fitness-Value Computation for Paths

This section presents our technique for computing and assigning a fitness value to a path based on the fitness function for a given target predicate.

Fitness-value computation in dynamic symbolic execution. Computing fitness values in the context of symbolic execution is complicated since the fitness function may be applied on symbolic values and thus the fitness value would be symbolic. Comparing symbolic fitness values is expensive, requiring pairwise comparison and invocations of a constraint solver. To reduce analysis cost, taking advantage of dynamic symbolic execution (the context where our approach is applied), our technique uses concrete variable values (collected at runtime) to compute fitness values based on fitness functions.

Fitness-value assignment to a path. Given a target predicate and an explored path, we assign a fitness value to the path with the following procedure. We first collect all the occurrences of the target-predicate evaluation (when a target branch is within a loop, the branch's target predicate can be encountered and evaluated multiple times in the explored path). We then compute the fitness value for each occurrence of the target-predicate evaluation, and assign the best (lowest) fitness value among these fitness values to the path. When there is no occurrence of the target-predicate evaluation in the path, we assign the worst fitness value (e.g., the maximum 32-bit integer) to the path. Intuitively, we would give higher priority to flipping a branching node in a path with a better (lower) fitness value. The next section describes a technique for further helping determine flipping priority for branching nodes in a path.

4.1.3 Fitness-Gain Computation for Branches

Selecting a branching node in a path to flip can be reduced to selecting a branching node that (1) has at least one immediate outgoing branch not explored yet and (2) once flipped has the best potential for improving the path's fitness value. To measure the potential of each branching node, we first compute the fitness gain for each branch in the program under test as illustrated next.

In each iteration of path exploration, assume that a branching node bn (whose corresponding branch is b) in path p_i (with fitness value as fv_i) is flipped and a new path

p_{i+1} (with fitness value as fv_{i+1}) is produced. Then the fitness improvement from p_i to p_{i+1} is $(fv_i - fv_{i+1})$. That is, the fitness gain for flipping the branching node of b is $(fv_i - fv_{i+1})$. We finally compute the *fitness gain* for b as the average of all the fitness gains for flipping branching nodes of b in the past. (Note that a fitness gain can be negative when $(fv_i - fv_{i+1})$ is negative, indicating that such flipping is not desirable.) Intuitively, we would give higher priority to flipping a branching node (in a path) for a branch with a better (higher) fitness gain.

4.1.4 Fitness-Guided Exploration

To help prioritize branching nodes for flipping among all the branching nodes (from the explored paths), we compute a composite fitness value for each branching node as below. A *composite fitness value* of a branching node (for a branch b) in an explored path p is computed as $(F(p) - FGain(b))$, where $F(p)$ is the fitness value of p and $FGain(b)$ is the fitness gain of b. We give higher flipping priority to a branching node with a better (lower) composite fitness value.

The prioritization of branching nodes produced by our Fitnex search strategy basically implements the first line of the loop body in Figure 2: in each iteration, the branching node with the highest priority is flipped to form a new program input (exploring a new path). Once a branching node has been flipped, our strategy removes it from the prioritized list of branching nodes, to avoid it from being flipped again in the future.

The next section describes how we integrate our Fitnex search strategy with other search strategies to effectively address exploration problems (in one program under test or across different programs under test), each of which may be amenable to only one or a few specific strategies (including the Fitnex strategy) being integrated.

4.2 Integration of Search Strategies

A straightforward and fair search strategy would be a *random* strategy, which chooses branching nodes to flip randomly. While such a strategy often performs reasonably well, it suffers from a grave problem: it would result in a random distribution of path lengths. If the program contains a loop over an unbounded unsigned integer of 32 bits, then a path would have $2^{32}/2 = 2^{31}$ branches on average. In other words, this strategy tends to dwell on un-rollable loops.

There are many well-known simple search strategies such as breadth-first and depth-first search strategies. However, each such strategy is biased towards particular control-flow points. While breadth-first search favors initial branches in the program paths, the depth-first search favors final branches.

To avoid any particular bias such as those preceding ones, Pex combines various strategies into a top-level meta-strategy. To this end, Pex provides a rich set of basic strategies and strategy combinators.

A strategy is informed about new branching nodes (in short as nodes), flipped nodes, and nodes that have been exhausted. Initially, a root node is announced. A strategy can be asked to provide the next node to flip; the strategy can choose to decline the request, e.g., when its test target has been covered.

When we conducted the evaluation (Section 5) for evaluating our new approach, the main strategy of Pex was defined as follows:

```
ShortCircuit(
  CodeLocationPrioritized[Shortest],
  DefaultRoundRobin)
```

This main strategy uses the following strategies and strategy combinators:

- The `ShortCircuit`$(s_0, \ldots, s_n)$ combinator combines a sequence of strategies $s_0, \ldots, s_n$ in the following way: as long as an earlier strategy s_i provides nodes to flip, a later strategy s_j (where $j > i$) will not be asked to provide nodes to flip.
- The `CodeLocationPrioritized`$[S]$ strategy partitions all nodes into equivalence classes based on the control-flow locations (branches) of which the nodes are instances. For each equivalence class, an inner frontier of strategy type S is maintained, which is informed about only nodes of its equivalence class. When a node is to be selected, a fair choice is performed between all equivalence classes, and then the inner frontier of the chosen equivalence class is asked to provide a node. When the inner frontier does not provide a node, another equivalence class is chosen. If all equivalence classes have been exhausted, no node is provided.
- The `Shortest` strategy maintains a list of nodes ordered by their depths in the execution tree, and the strategy remembers the smallest depth observed so far. When asked to provide a node, the strategy would remove the first node of the list and provide this node only if it has the smallest observed depth.
- The `DefaultRoundRobin` strategy combines a set of strategies[5] (one of which is Fitnex). This strategy makes a fair choice among the combined strategies when it is asked to choose the next branching node to flip. Most of the strategies are variations of breadth-first search over the execution tree, except that these strategies do not proceed strictly along the depth of the branches in the tree, but instead they proceed along a border that is defined by properties of the branching

---

[5] More details of these strategies can be found at `http://ase.csc.ncsu.edu/projects/fitnex/`

nodes such as structural coverage and calling contexts. The intention is to create diversity in the exploration beyond the notion of the depth of a branch.

In practice, we observe that Pex's other strategies in combination with Fitnex perform quite well (based on our experience and feedback of the Pex releases, which by default uses the combination of Pex's other strategies and Fitnex).

Pex's other strategies have a bias towards flipping nodes with short depths in the execution tree; the intuition is that easy cases should be covered fast. These other strategies partition nodes into various equivalence classes based on structural coverage, leading to diversity, while avoiding the general combinatorial explosion.

When using the frontier for the Fitnex strategy, Pex eagerly climbs local fitness hills without distraction from other frontiers by using *progress boost*: when the Fitnex frontier makes progress (i.e., a fitness value improves), the Fitnex frontier notifies its outer frontier, which may boost its probability of being used more often later on.

In summary, Fitnex works for only a particular type of problems (e.g., those amenable to fitness functions), and Pex's other strategies work well for other types of problems. These other strategies are in fact more general, and tend to work on a broader scope of programs. Our integration of Fitnex and other strategies achieves the effect of getting the best of both in practice.

5 Evaluation

We have implemented our Fitnex strategy and its integration with other strategies in Pex [15], an automated structural testing tool for .NET developed at Microsoft Research. To evaluate our Fitnex strategy and its integration, we compare the following different search strategies:

- *Pex with Fitnex*: Pex's default strategy as described in the previous section.
- *Pex without Fitnex*: a variation of Pex's default strategy for evaluation purposes, where the Fitnex strategy has been removed.
- *Random*: a strategy where branches to flip are chosen randomly in the already explored execution tree (but no branch is selected twice).
- *Iterative Deepening*: a strategy where breadth-first search is performed over the execution tree.

The *Random*, *Iterative Deepening* (breadth-first), and *Depth-First* strategies are representatives of search strategies commonly adopted in existing state-of-the-art DSE tools [2,3,5–7,14]. Comparing our proposed approach with these strategies helps assess the new benefits of our new approach over these existing strategies. In our evaluation results shown in Section 5.2, we did not include the *Depth-First* strategy commonly used in other DSE tools [3,6,14], as it consistently performs abysmally in most subjects used in the evaluation, since most subjects contain loops whose bounds are related to the program inputs, and the *Depth-First* strategy keeps unrolling the last loop instead of attempting to achieve test targets.

Table 2. Evaluation subjects and results

subject	#basic blocks	#runs Pex with Fitnex	#runs Pex without Fitnex	#runs random	#runs iterative deepening
1	9	15	22	12	227
2	16	45	58	13	127
3	29	26	30	14	50
4	40	**9**	**9**	**15**	**22**
5	20	42	22	16	42
6	28	**17**	**127**	**19**	**28**
7	21	35	27	21	51
8	34	91	30	24	65
9	29	**18**	**25**	**26**	**24**
10	25	**18**	**26**	**26**	**24**
11	27	**18**	**26**	**26**	**24**
12	27	**18**	**26**	**26**	**24**
13	27	**18**	**26**	**26**	**24**
14	39	**11**	**11**	**27**	**31**
15	34	**16**	**17**	**33**	**16**
16	9	**13**	**26**	**41**	**295**
17	40	**12**	**26**	**41**	**1000**
18	18	122	68	43	369
19	11	**20**	**33**	**45**	**135**
20	18	**20**	**35**	**46**	**962**
21	25	**17**	**17**	**55**	**39**
22	25	65	52	55	118
23	19	31	30	112	33
24	16	31	30	112	33
25	44	**62**	**104**	**185**	**113**
26	11	**22**	**171**	**277**	**823**
27	9	**23**	**249**	**566**	**1000**
28	9	**23**	**249**	**566**	**1000**
29	21	**24**	**73**	**1000**	**1000**
30	62	**101**	**775**	**1000**	**1000**
mean improvement over random		5.2	1.9	n/a (1)	0.9

Through this evaluation, we intend to answer the following research questions:

- Is the integrated Fitnex strategy (i.e., *Pex with Fitnex*) effective in achieving high code coverage fast?
- To what extent does the Fitnex strategy degrade or improve the performance of other strategies when they are integrated?
- How does the performance of the integrated Fitnex strategy (i.e., *Pex with Fitnex*) compare to the performance of the other strategies?

5.1 Subjects

The evaluation subjects listed in Table 2 (whose Column 2 shows the number of a subject's basic blocks) are a collection of benchmark subjects routinely used by the Pex developers to evaluate Pex's performance[6]. Note that these subjects were created independently of the Fitnex approach (existing before the conception and development of Fitnex) so evaluating Fitnex on these subjects provides an unbiased assessment, whose results reflect a strong indication of general performance in the field.

The subjects were created by extracting characteristic exploration problems from real, complex C# programs. As a result, their size may seem small (each has less than 100 basic blocks), but such small size is because they contain only the essence of an individual exploration problem. Each subject contains one or more hard-to-reach statements that were pre-designated. More specifically, each subject consists of a sequence of boolean guards, followed by hard-to-reach statements that can be reached only by fulfilling all guards. A simple example of such a subject is listed below, where the target is to create a string as the test input that starts with the word "Hello", ends with "World!", and contains at least one space.

```
public void HelloWorld(string value)   {
    if (value.StartsWith("Hello") &&
        value.EndsWith("World!") &&
        value.Contains(" "))
        MustReach();
}
```

One subject is a small parser for a Pascal-like language, and the target is to create a legal program; the smallest legal program to be created as input to the subject is of the following form: `program X; begin end.`

Another set of subjects is similar to the example `TestLoop` shown in Figure 1, where a loop iterates over program inputs, and then values computed by the loop are used later on.

These subjects contain few boolean variables, most of them contain one or more loops, and most of them contain strings, e.g., arrays of characters. Specifically, several subjects encode constraint systems over strings. Pex explores these subjects by analyzing the called string methods as well, which often contain loops over individual characters of the strings.

5.2 Results

In the evaluation, we measured how many *runs* were needed to cover designated hard-to-reach statements in our evaluation subjects. Each run indicates that a node flipped by the combined top-level search strategy indeed leads to the discovery of a new feasible path. At most 1000 runs were considered for each subject.

In the evaluation, the execution time of DSE was dominated by the time spent to execute and monitor the subjects, and constraint solving. The time spent by the search strategies, including Fitnex, to maintain the nodes and select the next branching node to flip was negligible. Thus, the number of runs is a good proxy for measuring the effectiveness of search strategies.

The results of the evaluation are shown in Table 2. For each subject, the number of runs needed to cover the designated statements is shown in Columns 3-6 of Table 2 for the four compared strategies. We highlight in bold the subjects that our *Pex with Fitnex* strategy (Column 3) achieves better or same performance than any of the other three strategies. Smaller run numbers are better and the run number of 1000 indicates that some of the designated statements could not be reached. The last row of the table shows the mean improvement factor of each strategy over the *Random* strategy.

From the results, we observe that *Random* and *Iterative Deepening* cannot always produce paths covering the test targets (reflected by their run numbers of 1000); the performance of these strategies serves as our baseline, against which we want to improve. The *Pex without Fitnex* strategy could find paths to cover all test targets. Adding *Fitnex* to Pex's other strategies (i.e., the *Pex without Fitnex* strategy) effectively improves the overall performance: while the *Pex without Fitnex* strategy improves performance on average by a factor of 1.9 over the *Random* strategy, including *Fitnex* (i.e., *Pex with Fitnex*) improves performance on average by a factor of 5.2 over the *Random* strategy.[7]

Compared to the *Random* and *Iterative Deepening* strategies, the *Pex with Fitnex* strategy is more effective in achieving high code coverage faster. The *Pex with Fitnex* strategy also improves the overall performance over Pex's other strategies alone (i.e., the *Pex without Fitnex* strategy). As a result, the *Pex with Fitnex* strategy is most effective in achieving high code coverage faster.

Note that in 8 out of 30 subjects, the *Pex with Fitnex* strategy performs worse than the *Random* strategy. This phenomenon happens when a previously observed fitness gain causes the *Fitnex* strategy to drive the exploration to an eventually unfruitful direction. We use the following example program to illustrate the phenomenon.

```
public void LoopExample(int[] a) {
  int sum = 0;
  for (int i = 0; i < 100; i++)
   { if (a[i] > 0) sum++; }
  if (sum > 100) { ... }
  ...
}
```

The true branch of the last shown conditional is not covered during some initial runs and is considered as one of the target branches. The *Fitnex* strategy will notice that flipping to produce more positive array elements allows to get closer

---

[6] More details of the evaluation subjects and results can be found at `http://ase.csc.ncsu.edu/projects/fitnex/`

[7] Since we capped the number of runs at 1000, which was relevant only for the *Random* and *Iterative Deepening* strategies, these average improvement factors are in fact conservative.

to covering the target branch, but a human observer can easily determine that this target branch cannot ever be covered. Such a factor leads to a bias of the exploration that causes the *Fitnex* strategy to perform worse than the *Random* strategy (when the program contains many more branches elsewhere that are not yet covered and are coverable). However, if the branch would have been `if (sum >= 100) { ... }`, then *Fitnex*'s attempts would have led the exploration to an eventually fruitful direction, as the branch is now coverable.

This example illustrates one situation where the input is first transformed or used in transforming another variable, and branches in later code may depend on the result of the transformation. Not all (combinations of) branches in later code will be reachable, and *Fitnex* may cause the unrolling of loops with the intention to cover a particular branch, but being unable to actually cover it in the end.

Overall, the benefits of integrating *Fitnex* into Pex's other strategies (i.e., the *Pex with Fitnex* strategy) is greatly beneficial: even if *Fitnex* sometimes drives the exploration into a wrong direction, then the fair combination of *Fitnex* with other strategies ensures overall diversity in the exploration; and if *Fitnex* drives the exploration into a beneficial direction, then a substantial advantage in the achieved coverage can be gained. In those cases, the *Fitnex* strategy can often reduce the exponential path explosion to a polynomial problem, as the *Fitnex* strategy always makes progress towards a test target. Such an effect is reflected by the observation that the *Pex with Fitnex* strategy achieved significant improvement over the other strategies for Subjects 25-30 (where the *Random* strategy requires more than 180 runs to achieve the test targets or fails to achieve the test targets within 1000 runs), whereas the *Pex with Fitnex* strategy suffers only marginal loss on the 8 subjects where the *Random* strategy performs better.

6 Related Work

Path exploration strategies in symbolic execution. DART [6] and CUTE [14] perform a depth-first search. SMART [5], an extension of DART, computes method summaries in order to make the analysis modular and thereby more scalable, and it requires a fixed order that explores innermost functions first. EXE [3] uses depth-first search as its default strategy. It also provides a mixture of best-first and depth-first search based on coverage heuristics. SAGE [7] uses a generational search that explores only a very limited horizon, starting from an execution path spawned by a meaningful seed input. CREST [2] uses a search strategy guided by the control flow graph of the program under test and two random strategies. JPF [1] includes strategies of depth-first and breadth-first search in addition to structural heuristics [8]. None of these existing search strategies is strongly guided towards covering test targets in the form of branches. In contrast, our new Fitnex strategy is the first one directly using fitness values to effectively guide the exploration of paths towards covering individual branches. In addition, the Fitnex strategy is integrated with other strategies in Pex to achieve overall effectiveness for programs with various characteristics.

Search-based test generation based on fitness values. Search-based test generation [11], often referred to as evolutionary testing (ET) [17], uses genetic algorithms to find test data to achieve test target coverage. Based on fitness values computed from test outputs or other observations along execution paths, ET selects a subset of test inputs with the best fitness values, and then applies crossover and mutation operations (in a random fashion) on this subset to produce new test inputs. Evacon [9] loosely integrates evolutionary testing [17] and symbolic execution [14] in generating effective method sequences for achieving high structural coverage. Our new Fitnex strategy shares commonality with existing ET approaches in that both use fitness functions to compute fitness values. However, the salient novelty and difference of Fitnex is our novel way of using the fitness values to guide the search process for feasible paths in DSE, as opposed to performing a search (largely randomly) on the test inputs, as usually done in ET. Fitnex uniquely computes fitness gains for branches to help select branching nodes to flip, which involves solving constraint systems, whereas existing ET approaches would randomly apply crossover and mutation operations on test inputs. In addition, our Fitnex strategy is also integrated with other strategies for achieving overall effectiveness.

7 Discussion

Enhancement of fitness functions. We currently assign the worst possible fitness value to a path that does not reach immediately before a test target (such as a branch). Assume that none of the known paths reaches immediately before a test target, then we will have no indication of which paths are more promising to cover a test target. In this case, the Fitnex frontier does not choose any node to flip, but lets other integrated strategies proceed. In future work, we plan to explore using a new type of fitness function [17] by measuring the distances between an already explored path and statically computed paths that can reach immediately before a test target. Then we can assign different fitness values to different paths that do not reach immediately before the test target if these paths have different distances in reaching before the test target.

Guidance from tool users. Sometimes tool users could have insights and knowledge in knowing which test targets to focus on first or formulating a sub-target to focus on first. With the best of our knowledge, no previous test generation tool provides convenient features to allow tool users to guide the tool when the tool cannot effectively accomplish the test generation task. For example, a tool may get stuck in exploring loop iterations that may not help cover a particular test target. The integration of Fitnex with other strategies alleviates the issue to some extent, but there will always be programs for which no fully automatic and effective search strategy exists. To this end, Pex provides a mechanism for allowing tool users to specify annotations for informing Pex which portion of the program under test should be given higher (or lower) priority in path exploration. These annotations are used by several search strategies, including Fitnex. We plan to explore this promising area of cooperation between the tool and tool users in accomplishing testing tasks in future work.

Method-sequence generation. In object-oriented test generation, generating effective method sequences [9] is an important and yet challenging problem. We can reduce the method-sequence generation problem to the path-exploration problem by constructing a test driver such as the one below for testing a `Stack` class:

```
public void TestDriverForSeq(int[] methods,
                   int[] args, int SeqLen){
 Stack s = new Stack();
 for (int i = 1; i <= SeqLen; i++) {
    switch (methods[i])    {
      case 1: s.push(args[i]); break;
      case 2: s.pop();  break;
      default: s.clear(); break;
    }
}}
```

In method-sequence generation, the path explosion problem is aggravated and especially calls for an effective approach such as the one proposed in this paper. We plan to apply our Fitnex strategy and its integration to address the sequence-generation problem in future work.

Complexity of fitness-guided exploration. If a given program is amenable to fitness functions, then Fitnex basically reduces the general problem of exponential path exploration to polynomial complexity, since the fitness-guided exploration always makes progress towards the coverage of the test target, instead of trying all possible combinations as in previous approaches.

8 Conclusion

Dynamic symbolic execution generates test inputs to achieve test target coverage by iteratively exploring paths of the program under test. Although the reachability of a test target is undecidable in general, dedicated search strategies may be effective for certain kinds of programs in finding paths that cover a test target. We have developed a novel search strategy called Fitnex for fitness-guided path exploration in dynamic symbolic execution. Fitnex prioritizes the search by minimizing fitness values, which indicate how close a path is to covering a test target. Fitness-guided exploration is integrated with other search strategies to achieve overall testing effectiveness. We have implemented the Fitnex strategy and integrated it with other strategies in Pex, an automated structural testing tool for .NET developed at Microsoft Research. The evaluation results show that our new approach consistently achieves high code coverage faster than existing search strategies.

Acknowledgments

This work is supported in part by NSF grant CCF-0725190 and ARO grant W911NF-08-1-0443.

References

[1] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java Pathfinder. In *Proc. TACAS*, pages 134–138, 2007.

[2] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443–446, 2008.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proc. ACM CCS*, pages 322–335, 2006.

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

[5] P. Godefroid. Compositional dynamic test generation. In *Proc. POPL*, pages 47–54, 2007.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 75–84, 2005.

[7] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*, pages 151–166, 2008.

[8] A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proc. ISSTA*, pages 12–21, 2002.

[9] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. ASE*, pages 297–306, 2008.

[10] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[11] P. McMinn. Search-based software test data generation: a survey. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[12] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[13] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, 2007.

[14] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[15] N. Tillmann and J. de Halleux. Pex – white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[16] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.

[17] P. Tonella. Evolutionary testing of classes. In *Proc. ISSTA*, pages 119–128, 2004.

[18] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Proc. ISSTA*, pages 73–81, 1998.