Pruning the Search Space in Path-based Test Generation

Sébastien Bardin Philippe Herrmann
CEA, LIST
Software Reliability Lab
Boîte 65, Gif-sur-Yvette, F-91191 France
E-mail: first.name@cea.fr

Abstract

Recent advances in path-based test (data) generation open the way to the systematic testing of large scale programs. However, these technologies are still suffering from two major bottlenecks: efficient constraint solving and the path explosion phenomenon. We focus in this paper on the second issue and propose three complementary heuristics geared toward lowering path explosion. All these heuristics are both easy to implement and lightweight, and each one deals with a distinct source of path explosion. We provide theoretical and experimental evidence that they can achieve a significant reduction in the search space.

Keywords: *test data generation, symbolic execution, path explosion, lightweight heuristics*

1. Introduction

Context. There are currently two well established frameworks to achieve automatic test (data) generation from source code: constraint-based testing [8, 9, 11, 25, 28] (CBT) and search-based testing [13, 14, 17, 18, 19] (SBT). While the former approach focuses on translating (part of) a program into a logical formula whose solutions are relevant test data, the later approach is based on exploring the input space of the program using optimisation-like methods to guide the search toward relevant test data. SBT can be local or global, see [17] for an overview. CBT also can be global, translating the whole program in a formula [8, 9], or local (path-based) [11, 25, 28], focusing on a single path.

We focus in this paper on path-based CBT, referred as path-based testing. Obviously such a local analysis is not sufficient to prove correctness of the whole system, however it is sufficient to derive a test data exercising the given path at runtime. Then, iterating the process on many different paths allows to automatically build test suites achieving a given structural coverage objective, e.g. branch coverage.

Recent and impressive progress in constraint solvers as well as the combination of both concrete and symbolic execution (referred as *concolic execution* [11, 25] or *mixed execution* [6]) make it possible to perform automatic path-based testing on large scale programs. Concolic path-based testing tools have literally blossomed up recently [3, 5, 6, 7, 12, 27]. Path-based testing, once coupled with a partial but automatic oracle (e.g. assertion or contract checking, runtime error monitoring), opens the way to the systematic massive testing of programs before their release. However, path-based methods still suffer from two major bottlenecks: first, constraint solving along a single path can still be expensive; second, the total number of (bounded) paths to explore may be exponential in the size of the bound.

Path explosion phenomenon. The path explosion happens mainly because of nested calls, loops and conditions. Moreover, path-based methods often require a bound k on the length of paths to explore, and the number of paths may increase considerably if k is overestimated. This is all the more problematic in programs where some parts of the code can be reached only with very long paths, which is a common situation (e.g. reactive systems).

Item coverage and irrelevant paths. Covering all paths of a program is not the primarily objective of current testing practices. Even in critical systems, it is only require to fully cover a class of structural artifacts of the program code source such as instructions, branches or atomic predicates. In the rest of the paper, we denote by *items* these three classes of artifacts. There is an obvious mismatch between path-based approaches and such item coverage goals: while each new test data does cover a new path, it may hit no new item. It then turns out that path-based testing methods coupled with item coverage objectives tend to waste a lot of time trying to compute *irrelevant test data*, i.e. test data exercising no new item. This problem is even worse in a unit testing framework, where only the function under test has to be covered.

Our goal. We focus in this paper on the path explosion issue in path-based testing with item coverage objectives. Our aim is to provide heuristics to discard irrelevant paths as much as possible, thus reducing the number of solver calls and the whole computation time. Moreover, we are interested in heuristics which are both simple to implement into a basic path-based framework and cheap to execute, yet providing significant reduction in the path space.

Results. We propose three heuristics for three distinct common sources of path explosion. These heuristics can be combined altogether to get better performance.

- The Look-Ahead heuristic (LA) stops the current path exploration when no uncovered item can be reached from the current control location.
- The Max-CallDepth heuristic (MCD) prevents the search from backtracking in deep function calls, reducing the path explosion due to nested calls.
- The Solve-First heuristic (SF) modifies the standard DFS procedure in order to solve shorter path prefixes (and simpler constraints) early.

LA and MCD heuristics work both for symbolic execution and concolic execution, while SF makes sense only in a concolic setting. LA and SF perform a sound pruning in the sense that they discard only irrelevant paths, while MCD trades a possible loss in item coverage for a significant reduction of the number of paths.

We present the three heuristics as enhancements of a (bounded) depth-first search (DFS) path-based procedure, either purely symbolic or concolic. Original path-based testing techniques were based on DFS [11, 25, 28], while some recent works advocate using other search strategies [7, 12, 22]. LA, MCD and SF are easy to adapt to these frameworks as well.

Contributions. There are three main contributions in this paper. First, we describe three novel heuristics (LA, MCD and SF) to attack the path explosion issue in path-based testing with item coverage objective. Each technique addresses a particular cause of path explosion, and they can be combined together for greater efficiency. We show that all these techniques are easy to implement in the standard DFS framework and we discuss integration in symbolic and concolic settings. Second, we provide theoretical evidence that these heuristics can achieve huge savings on particular examples, and that better performances may be obtained when the techniques are combined. Third, these techniques have been implemented in an existing path-based testing tool [5] and experiments have been carried out to assess their efficiency. We provide experimental evidence that they do offer a significant increase in performance.

Considering the ease of implementation, the low computational overhead and the effective path pruning, we believe that integration of these techniques should be considered in any path-based testing tool.

Outline. The remaining part of the paper is structured as follows. Section 2 provides a brief overview of the basic (symbolic/concolic) DFS procedure, describes the set of benchmarks used throughout the paper and gives basic definitions. The novel heuristics are discussed from sections 3 to section 5. In each section, we present the underlying source of path explosion, describe the heuristic, provide theoretical arguments about the potential pruning, discuss implementation details and report experiments that have been conducted. Finally, section 6 gives an overview of related work and section 7 provides a conclusion and directions for future work.

2. Preliminaries

Principles of symbolic execution. Path-oriented and path-based approaches are built upon the central notion of path predicate.

Definition 1 (Path predicate)**.** *Given a program P of input domain D and π a path of P, a path predicate of π is a formula φ_π on D such that if $V \models \varphi_\pi$ then execution of P on V follows the path π.*

The two main ideas behind symbolic execution are that: (1) a solution to a path predicate φ_π for a given program P is actually a test data exercising path π, with potential applications in structural testing; and (2) a path predicate φ_π for a path π can be computed by keeping track of logical relations among variables along the execution, rather than just updating their concrete values. Figure 1 shows how a symbolic execution is performed on a small program.

Loc	Instruction	Symbolic exec
0	`input(y,z)`	new vars Y_0, Z_0
1	`y++`	$Y_1 = Y_0 + 1$
2	`x := y + 3`	$X_2 = Y_1 + 3$
3	`if (x < 2 * z)` *(branch True)*	$X_2 < 2 \times Z_0$
4	`if (x < z)` *(branch False)*	$X_2 \geq Z_0$

Path predicate for path 0→1→2→(3,True)→(4,False)

$Y_1 = Y_0 + 1 \wedge X_2 = Y_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$

Path predicate projected on input

$Z_0 - 4 \leq Y_0 < 2 \times Z_0 - 4$

Figure 1. Symbolic execution along a path

Basic procedure. The basic (purely symbolic) path-based testing procedure consists in choosing a path, computing and solving its path predicate, recording the solution (if any) as a test datum and iterate until all paths have been covered. Typically the tree of all paths of the program, i.e. the unfolding of the Control-Flow Graph (CFG), is explored in a depth-first search manner, each branch in the execution tree corresponding to a choice point in the program (`ite`, `switch`, etc.). The path predicate of each path is built incrementally, reusing the path prefix up to the last choice point in the program. Basic instructions are translated into semantically equivalent formulae, for example $x' = z + 1$ for the statement `x:=z+1`. For conditional, we just force the search to take the *"if"* or *"else"* branch by adding to the current path predicate Φ the condition predicate $cond$ or its negation $\neg cond$.

Procedure 1 presents the basic idea of the test generation procedure. The procedure takes as input the CFG of the program. We assume that CFG nodes can be either blocks (**block**) of basic assignments (like `x:=a+b+5`), conditional statements (**ite**) and static jumps (**goto**). The CFG is given by its nodes with method `.next` to access the next node. Basic instructions are translated into formulae by the procedure `atomic`. The external procedure `solve` returns a solution to a constraint or the `unsat` exception in case of unsatisfiability.

We do not impose any particular theory for expressing path predicates in this paper, the only requirement is to have a correct solver able to generate a solution rather than just answering "satisfiable". Popular theories for path-based testing are the quantifier-free fragments of linear arithmetic and bit-vector theory, possibly extended with uninterpreted function symbols and arrays. The interested reader is referred to [20] for an introduction.

```
procedure GENTEST(node_init)
input : initial node node_init
output: set of test data Tests
 1: Tests := ∅
 2: SEARCH(node_init, ⊤)        /* ⊤ denotes the "true" formula */
 3: return Tests
procedure SEARCH(node, Φ)
input : node, path predicate Φ
output: no result, update Tests
 1: Case node of
 2: | ε →       /* end node */
 3:       try S_p := SOLVE(Φ) ; Tests := Tests + {S_p}
 4:       with unsat → ();
 5:       end try
 6: | block i → SEARCH(node.next, Φ ∧ ATOMIC(i))
 7: | goto tnode → SEARCH(tnode, Φ)
 8: | ite(cond,inode,tnode) →     /*branching*/
 9:       SEARCH(inode, Φ ∧ cond);SEARCH(tnode, Φ ∧ ¬cond)
10: end case
```

Procedure 1: Symbolic Basic DFS Procedure (BP)

Discussion. For the sake of clarity, several details have been omitted in procedure 1. First, the length of paths must be bounded to prevent the procedure from infinite looping. Second, we present the procedure from an all-path coverage perspective while practical needs often require less demanding criteria. In this case, the program must record the set of uncovered items U and update it each time a path predicate is successfully solved. The program stops as soon as U is empty. Moreover, such a procedure will try to cover the main function plus all its callees, while in a unit testing perspective the procedure can stop as soon as the main function is covered.

Adapting to procedure 1 other instructions commonly found in imperative programming languages is straightforward. For example,

switch$(e, (v_1 \rightarrow n_1), \ldots, (v_l \rightarrow n_l), (\textit{default} \rightarrow n_d))$
can be handled by: branch to n_i has the path predicate $(e = v_i) \wedge \Phi$ and the default branch has the path predicate $\bigwedge_i (e \neq v_i) \wedge \Phi$. Functions are usually inlined. This requires to keep track of the call stack, and function calls and returns are managed like jumps plus additional constraints for context binding.

Improvement: concolic execution. Concolic execution [11, 25] is a recent major improvement to symbolic execution. Basically, a concolic execution is a concrete execution and a symbolic execution running in parallel, the concrete execution collecting relevant information along the execution path to help the symbolic execution. A typical mechanism is the *concretization* of a variable: at some point of the symbolic execution, a logical variable is forced to be equal to the corresponding concrete value over the concrete execution. Concretization permits meaningful approximations on instructions which are either impossible to model in the given path predicate theory or whose modelling is too costly to solve. We can cite for example non-linear arithmetic constraints, calls to native code and multiple levels of pointer dereferencement.

Procedure 2 describes a straightforward concolic extension of procedure 1. The purely symbolic procedure is extended with a concrete memory state (`C`) mapping program variables to their concrete value. The concrete memory state can be updated by an instruction and evaluated against a condition (procedures `update` and `eval`). It is initialised either randomly or at a constant value, typically 0 for each memory cell. The set of test data is initialised with the initial concrete memory state. The major modification compared to procedure 1 is that when a conditional statement is encountered, the concrete memory state is first evaluated to decide which successor must be followed (the corresponding path is feasible since the concrete execution goes through). On backtracking, the alternative path is *immediately solved*. On success, a new test datum exercising the

path prefix plus the new branch is generated. The search then continues through the new branch with a new concrete memory state (derived from the concrete execution of the test datum) consistent with the new branch. Intuitively, the new test guides the search toward the new branch.

```
procedure GENTEST1(node_init)
input : initial node node_init
output: set of test data Tests
 1: C := init_concrete()
 2: Tests := {C}
 3: SEARCH(node_init, ⊤,C)        /* ⊤ denotes the "true" formula */
 4: return Tests
procedure SEARCH(node, Φ, C)
input : node, path predicate Φ, concrete state C
output: no result, update Tests
 1: Case node of
 2: | ε → ()        /* end node */
 3: | block i → SEARCH(node.next, Φ ∧ ATOMIC(i), update(C,i))
 4: | goto tnode → SEARCH(tnode, Φ, C)
 5: | ite(cond,inode,tnode) →
 6:       Case eval(cond,C) of
 7:         | true →
 8:           SEARCH(inode, Φ ∧ cond, C);
 9:           try /* try to find C' compatible with the else branch */
10:             S_p := SOLVE(Φ ∧ ¬cond) ; Tests := Tests + {S_p}
11:             C' := UPDATE_C_FOR_BRANCHING(S_p)
12:             SEARCH(tnode, Φ ∧ ¬cond,C')  /* branching */
13:           with unsat → ()
14:           end try
15:         | false → .....        /* symmetric case */
16:       end case
17: end case
```

Procedure 2: Concolic Extension of BP

Terminology. In the rest of the paper, we use the following abbreviations: BP stands for the standard DFS path-based testing procedure, either purely symbolic like procedure 1 or concolic like procedure 2; UT (unit testing) indicates that the procedure stops as soon as the top-level function is fully covered; LA, MCD and SF denote the three heuristics Look-Ahead, Max-CallDepth and Solve-First. Extensions of BP are written like in BP+LA+UT indicating that the basic procedure is used in a unit testing way and enhanced with the Look-Ahead heuristic. In the rest of the paper, we compare different test generation procedures. One criterion is their covering power.

Definition 2 (Covering power)**.** *Given two path-based testing procedures P_1 and P_2, we say that P_1 and P_2 have the same covering power if for any program $Prog$ and any depth-bound k, P_1 and P_2 executed on $Prog$ with depth-bound k achieve the same item coverage.*

About experiments. All heuristics discussed in this paper have been implemented in the concolic execution tool OSMOSE [5]. This tool is designed to generate test data from machine code and it follows mostly concolic BP. Experiments have been conducted on a bench of six small C programs cross-compiled to the PowerPC 550 architecture (32-bit) and to the Intel 8051 architecture (8-bit). The test suite is taken from [5]. We used the following cross-compilers: `sdcc` for the 8051 and `gcc` for the PowerPC. Characteristics of the different executable files are described in table 1. To avoid any confusion, an `ite(cond,if,then)` statement counts for one instruction and two branches.

program	processor	#I	#Br	#F	CD
check-pressure	8051	59	10	3	1
square 3x3	8051	272	46	1	0
square 4x4	8051	274	46	1	0
hysteresis	8051	91	16	2	1
merge	8051	56	24	3	1
triangle	8051	102	38	5	3
square 4x4	PPC 32-bit	226	30	1	0
hysteresis	PPC 32-bit	76	16	2	1
merge	PPC 32-bit	188	16	3	2
triangle	PPC 32-bit	40	18	3	2

#I : n. of instructions   #Br : n. of branches
#F : n. of functions   CD : maximal call depth

Table 1. Benchmark description

All evaluations are performed on an Intel Pentium M 2Ghz with 1.2 GBytes of RAM running Linux Ubuntu 6.10. The time-out for the solver is set up to 1 minute.

3. Look-Ahead (LA) heuristic

Motivation. Procedure BP is primarily designed to cover all paths of the program up to a given bound. It turns out that while each new iteration tries to generate a test datum exercising a new path, this test datum may hit no new item, wasting all the computation time spent in solving the path predicate. Tracking the set of current uncovered items in procedure 1 is a first solution discarding the worst case, when the procedure still enumerates paths while full item coverage is already achieved.

Nevertheless, in many cases such redundant test data may still be produced. Typically when uncovered items remain but the procedure searches in a part of the program already fully covered, e.g. the example of figure 2. At most three test cases are required to cover all instructions, while there are infinitely many paths going through the *true* condition.

Look-Ahead heuristic. The key idea of the Look-Ahead heuristic (LA) is to perform a reachability analysis (in terms of reachable items in the CFG) to decide whether the current path must be expanded or not. If no new items can be reached, then exploration along the current path is stopped.

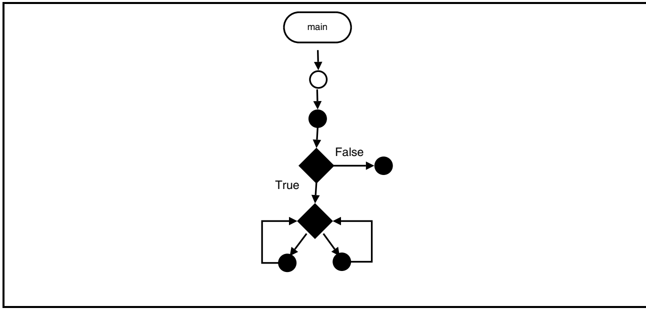

Figure 2: Running example for LA

In this case, we do not prune only a single path, but the whole part of the execution tree following the current CFG node.

A purely symbolic version of BP+LA is described in procedure 3. We show only modifications of BP. The set of uncovered items is denoted by U. This is a global variable, initialized with all items before the first call to SEARCH. The main difference compared to procedure 1 is that at each choice point, we call the `reachUncover(node,`U`)` routine to test whether the set of reachable items from `node` intersects U. If not, current path exploration stops. However, the path prefix may still have to be solved if it goes through an uncovered item. It is why we keep track of all items encountered along the current path in a history variable H. Here, H contains both instructions and branches. The main issue in `reachUncover` is how to compute efficiently the (control) reachability set of `node`. We discuss this point further.

Property 1 (Soundness)**.** *The Look-Ahead heuristic discards only redundant paths w.r.t the coverage objective. Thus BP and BP+LA have the same covering power.*

Proof. The proof comes directly from the fact that pruning is done according to the class of items being covered. Note that the procedure test the intersection of the reachability set with $U - H'$ instead of simply U. Indeed, covering items in H' needs to solve only the current path prefix, not an extended prefix. □

Property 2 (Exponential reduction)**.** *BP+LA needs always less paths to achieve full coverage than BP, and there exist programs for which BP+LA explores exponentially less paths than BP.*

Proof. The first claim comes directly from property 1 and the observation that BP+LA follows mostly a DFS search. For the second claim, let us consider running example of figure 2, a depth bound $k \geq 4$ and instruction coverage, and let us assume that every path of the program is feasible. Then BP needs to explore $\approx 2^k$ paths to achieve full coverage (because of the two nested loops) while BP+LA requires at most 3 paths. □

```
procedure SEARCH(node, Φ, H)
input : node, path predicate Φ, set of items H
output: no result, update Tests
 1: /* U the set of uncovered items (global variable) */
 2: /* initialized with all items before the first call to SEARCH*/
 3: H' := H + {node}
 4: Case node of
 5: | ε →       /* end node */
 6:      try
 7:         S_p := SOLVE(Φ) ; Tests := Tests + {S_p}
 8:         U := U − H'
 9:      with unsat → ();
10:      end try
11: | block i → SEARCH(node.next, Φ ∧ ATOMIC(i), H')
12: | goto tnode → SEARCH(tnode, Φ, H')
13: | ite(cond,inode,tnode) →
14:      if (reachUncover(inode, U − H')) then
15:         SEARCH(inode, Φ ∧ cond, H'+{(node, true)})
16:         else if H' ∩ U ≠ ∅ then SEARCH(ε, Φ, H')
17:         else ()
18:      end if
19:      if (reachUncover(tnode, U − H')) then /* branching */
20:         SEARCH(tnode, Φ ∧ ¬cond, H'+{(node, false)})
21:         else if H' ∩ U ≠ ∅ then SEARCH(ε, Φ, H')
22:         else ()
23:      end if
24: end case
```

Procedure 3: Procedure BP+LA

Implementation details. The main point is to implement both efficiently and precisely the sub-procedure `reachUncover`, and especially the computation of reachability sets. Let us define `computeReachSet` as the subroutine that takes as input the CFG and the call graph of the program as well as a specific (instruction) node, and returns the set of items reachable from that node. We do not use the standard worklist algorithm from control-flow analysis (see for example [23]) because it computes the reachability set of every node at once, spending a lot of time before any test generation is performed. Moreover, we want a precise context-sensitive interprocedural analysis rather than the standard intraprocedural analysis. Our implementation relies on the following principles:

- lazy forward computation: we compute the reachability set of a single node at once, starting from that node and adding its successors in a bread-first manner;
- compact representation of sets of items: we manipulate finite sets of item identifiers (IIDs) and *function identifiers (FIDs)* representing compactly all IIDs from a given function;
- function summaries: a call to function f is summarised by collecting all items reachable from (entry points of) all functions reachable from f in the call graph; it can be easily computed from the call graph and efficiently represented by FIDs;
- computation cache and memoization: all previously

computed reachability sets are recorded in two distinct caches (one for instructions and one for functions) to avoid redundant computations;

- context-sensitive analysis: we keep track of the call stack, so that when a `return` instruction is encountered, we add recursively the reachability set of the first return site (and pop this site from the call stack). It ensures both a precise context-sensitive interprocedural analysis and an efficient cache mechanism, since we still record a reachability set for each instruction rather than for each pair *(instruction, call stack)*.

Discussion. The `reachUncover` test could be performed at each step of the procedure rather than only on branching. There is of course a trade-off between computational overhead and total path pruning. Testing reachability only on choice points seems a good compromise since testing on other instructions can shorten the length of paths but cannot prune more paths.

We now discuss several adaptations of the heuristic. LA is easy to adapt to unit testing: reachability sets are neither computed nor tested in callees, and the `computeReachSet` procedure is modified to stop on instructions `return` and ignore instructions `call` (treated as a `skip`). The heuristic is also easy to adapt to concolic execution. Actually the implementation is even easier in this case, since the current prefix is always already covered. Hence, lines 16 and 21 of procedure 3 as well as the history variable H are useless.

Evaluation. We have led two distinct sets of experiments on the benchmark described in section 2.

First, we evaluate the influence of LA on the overall performances of the tool. Results are summarised in table 2. Computation times are given in seconds and include the computation time for reachability sets (for BP+LA). We report the number of tests found by the tool. We believe it is a rather fair indicator of the number of explored paths, especially because BP and BP+LA are both DFS procedures. Both heuristics achieve the same coverage, as predicted by property 1.

BP+LA shows substantial gains both in terms of the number of generated tests and in terms of computation time.

Actually, the number of (generated) tests is reduced on average by 43% (57% for the total amount of tests), with a minimal reduction of 0% and a maximal reduction of 85%. The computation time is reduced on average by 37% (57% for the total amount of time), from an increase of 4% to a maximal reduction of 80%. We have also evaluated the heuristic when testing reachability on each branch (not only backtracking) and on every instruction. No more paths were pruned on these examples.

		BP		BP+LA	
program	cover	time	# tests	time	# tests
check-pressure 8051	100	5,4	4	+4%	0%
square 3x3 8051	100	24	43	-60%	**-70%**
square 4x4 8051	100	61	123	**-66%**	**-85%**
hysteresis 8051	100	5,6	35	-8%	-9%
merge 8051	100	2,7	70	-48%	-43%
triangle 8051	100	13	15	0%	-20%
square 4x4 ppc	82	68	125	**-64%**	**-84%**
hysteresis ppc	100	82	251	-46%	-44%
merge ppc	100	2,5	2	0%	0%
triangle ppc	100	65	19	**-80%**	-47%
mean				-37%	-43%
total				-57%	-57%

mean : the average value of all gains
total : the gain w.r.t. the total amount of time/tests (weighted mean value)

Table 2. Experimental results for LA

Second we evaluate the computational overhead of LA. We carried out experiments with a modified version of BP+LA, where `testReachSet` is computed normally but always return true, pruning no path. We then compare running times of BP and (BP+LA)'. Results are summarised in table 3. The `LA=1` and `LA=2` columns indicate respectively that reachability sets are computed only on backtrack or on each branch. Time overhead is reported w.r.t. the computation time of BP. It is clear from the table that the computational overhead of LA is negligible.

program	BP+LA' (time overhead)	
	LA=1	LA=2
check-pressure 8051	+0%	+7%
square 3x3 8051	+0%	+5%
square 4x4 8051	+0%	+4%
hysteresis 8051	+0%	+0%
merge 8051	+0%	+0%
triangle 8051	+1%	+4%
square 4x4 ppc	+0%	+5%
hysteresis ppc	+0%	+0%
merge ppc	+0%	+0%
triangle ppc	+0%	+1%
mean	+0.1%	+2,6%
total	0%	+2,4%

Table 3. Overhead of Look-Ahead

4. Max-CallDepth (MCD) heuristic

Motivation. Function calls, and especially nested function calls, are a major source of path explosion. It is all the more embarrassing when only the top-level function is of interest. For example, it may be the case that the procedure explores alternative (long) paths due to backtrack in deep

callees while a simple backtrack at top-level would be sufficient. Running example of figure 3 gives such a behaviour.

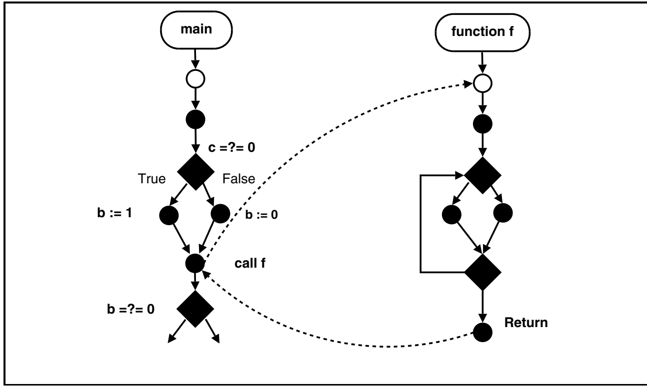

Figure 3: Running example for MCD

MCD heuristic. The principle of the Max-CallDepth heuristic (MCD) is to prevent backtracking in deep nested calls, hoping such a deep decision is not mandatory to cover the function under test. Implementation is straightforward: it is sufficient to add to the `REC` procedure a parameter `mcd` and a counter h recording the call stack height (updated on `call` and `return` instructions), then backtrack is allowed only when $h \leq mcd$.

It is clear that this heuristic makes sense only in unit testing. Moreover, contrary to LA, MCD may discard relevant paths and prevent the full coverage of the function under test. On the other hand, on some programs MCD can discard many paths and still achieve full coverage. These results are summarised in the next property.

Property 3. *The following relationships hold between BP+UT, BP+UT+LA and BP+UT+MCD:*

(i) (under-approximation) The covering power of BP+UT+MCD is strictly less than the one of BP+UT.

(ii) (exponential reduction) there exist programs for which BP+UT+MCD achieves the same coverage as BP+UT+LA, while exploring exponentially less paths.

(iii) (complementarity) there exist programs for which BP+UT+LA+MCD achieves the same coverage as BP+UT+LA (resp. BP+UT+MCD), while exploring exponentially less paths.

Proof. (i) Figure 4 shows a small program where BP+UT achieves full branch coverage of `main` (with $k \geq 8$). On the other hand, BP+UT+MCD with $mcd = 0$ and any value of k cannot cover one of the two branches of `main`, since a backtrack in sub-function `f` is necessary. *(ii)* Consider the program of figure 3, take $mcd = 0$ and let us assume that sub-function `f` does not affect variable `b`. Then both BP+LA+UT and BP+UT+MCD achieve full branch coverage. However, BP+UT+MCD explores only the two paths of function `main`, while BP+UT+LA explores $\approx 2^k$ paths. *(iii)* BP+UT+LA+MCD performs better than BP+UT+LA in the example of figure 3, and it outperforms BP+UT+MCD in the example of figure 2. □

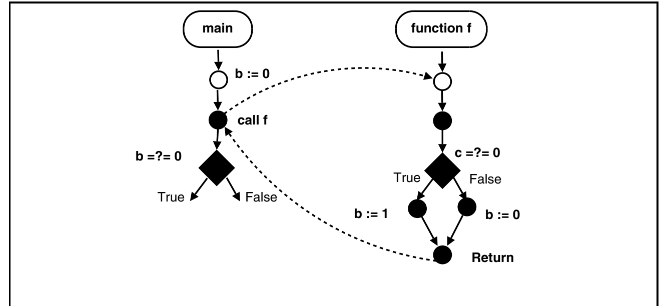

Figure 4: Example for property 3-1

Note that property *(iii)* shows that LA and MCD are complementary and do not address the same source of path explosion.

Discussion. MCD is different from simply concretizing function calls above a certain call depth. Indeed, such a technique would set both the path to follow in the callees and concrete values of input and output, while MCD fixes only the path in callees. We discuss this point further in related work (section 6).

Evaluation. Experiments have been performed on all the programs from our benchmark with callees, except the `hysteresis` program whose single sub-function has no choice point. test generation was launched in unit testing mode, and the `mcd` parameter was set to 0 (no backtrack in any sub-function). On these examples, BP+UT+MCD always achieves full branch coverage. Results are summarized in table 4.

BP+UT+MCD shows dramatic gains both in terms of the number of tests generated and in terms of computation time.

Actually, the number of tests is reduced on average by 54% (72% for the total amount of tests), with a minimal reduction of 0% and a maximal reduction of 84%. The computation time is reduced on average by 49% (85% for the total amount of time), with a minimal reduction of 0% and a maximal reduction of 97%.

We have also evaluated the heuristic with `mcd=1`. In this case, BP+UT+MCD behaves like BP+UT, except for the `check_pressure` program where performances are better than those of BP+UT.

		BP+UT		BP+UT+MCD (mcd=0)	
program	CD	time	# tests	time	# tests
check-pressure 8051	1	3,5	7	-20%	-43%
merge 8051	1	7	37	-57%	**-84%**
triangle 8051	3	13	12	**-75%**	**-66%**
merge ppc	2	2,5	4	-0%	-0%
triangle ppc	2	67	19	**-97%**	**-80%**
mean				-49,8%	-54,6%
total				**-85%**	**-72%**

CD: call depth

Table 4. Experimental results for MCD

5. Solve-First (SF) heuristic

Motivation. As usual, DFS path exploration lowers memory consumption compared to other searches since only one path at a time has to be maintained. However, DFS has at least two drawbacks in path-based testing. First, in a realistic setting where testing budget is not sufficient to cover all items (says the number of tests we can run is limited), DFS may focus only on a very deep and narrow portion of the program under test and achieve only a poor global coverage. Second, DFS-based procedures explore and try to solve first the longest path prefixes, while shorter prefixes with probably simpler constraints may have covered the same items. Hence, DFS suffers from a slow initial coverage-speed and may waste resources on unduly complex paths.

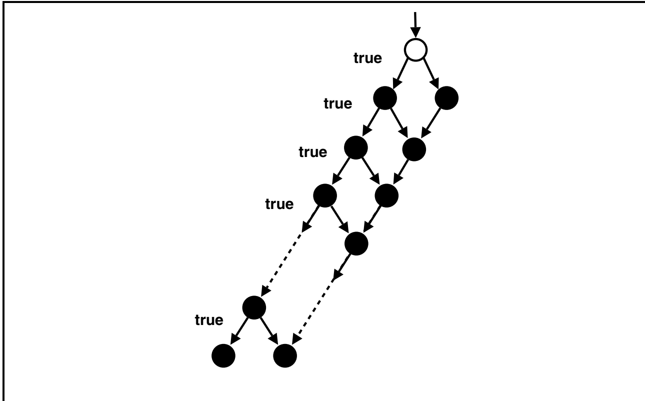

Figure 5: Running example for SF

SF heuristic. We propose the Solve-First heuristic (SF) to partially address both issues described above, while keeping most of the advantages of DFS. SF is a DFS with the slight following modifications. When the procedure reaches a choice point for the first time, the first successor node is chosen as usual but *all alternative successors are immediately resolved*. Test data are derived from the solutions and executed in concrete mode to update coverage information. All solutions found are stored in a cache. On backtrack, once the next successor is chosen, the corresponding test datum is retrieved from the cache. The test datum is then relaunched and the procedure proceeds as usual in the concolic case.

....
1: | ite(cond,inode,tnode) →
2: **Case** eval(cond,C) **of**
3: | true →
4: **try** */* try to find C' compatible with the else branch */*
5: S_p := SOLVE($\Phi \wedge \neg cond$) ; Tests := Tests + $\{S_p\}$
6: C' := UPDATE_C_FOR_BRANCHING(S_p)
7: $tnode.cache := C'$
8: `execute_concretely_and_update_cover` (S_p)
9: **with** unsat → tnode.cache := None
10: **end try**
11: SEARCH(inode, $\Phi \wedge$ cond, C);
12: `if` tnode.cache = None `then` ()
13: `else` SEARCH(tnode, $\Phi \wedge \neg$cond,tnode.cache)
14: | false → ... */* symmetric case */*
15: **end case**
....

Procedure 4: Procedure BP+SF

The main interests of this modified version are twofold. (1) Along a path, shorter and potentially simpler prefixes are resolved before longer ones. (2) Some paths of the programs very distant from the first path (w.r.t. DFS ordering) are resolved quickly, allowing for potential faster initial coverage.

Property 4. *The following properties hold:*

(i) BP+SF has the same covering power as BP;

(ii) for some programs, BP+LA+SF explores linearly less path than BP+LA to achieve full coverage.

Proof. *(i)* The property follows directly from the fact that BP+SF performs essentially the same exploration as BP, the only difference being that some path prefixes are resolved in advance. *(ii)* Considering the program in figure 5 parametrized by its number of nodes $2n+1$, and assuming that all paths are feasible and that the first concrete path goes through every *true* branch, BP+LA needs to explore $n+1$ paths to achieve full instruction coverage while BP+LA+SF needs only to explore 2 paths. ☐

Discussion. The depth bound for the concrete execution may be much larger than the one for concolic execution, allowing to cover cheaply more items. Note that SF is concolic in essence and has no interest in a symbolic setting.

Evaluation. Experiments have been performed on all the programs from our benchmark. On all programs but both versions of `square 4x4`, BP+LA+SF achieves the same coverage as BP. On `square-4x4`, BP+LA+SF achieves a smaller branch coverage (75%) than BP, seemingly due to a problem in the solver. Results are summarized in table 5.

	BP		BP+LA+SF	
program	time	# tests	time	# tests
check-pressure 8051	-52%	-50%	-48%	-50%
square 3x3 8051	-58%	**-65%**	0%	+15%
hysteresis 8051	+40%	**-80%**	+60%	**-78%**
merge 8051	+120%	**-83%**	+227%	**-71%**
triangle 8051	+75%	+50%	+75%	+100%
hysteresis ppc	**-86%**	**-98%**	**-72%**	**-96%**
merge ppc	+80%	+30%	+80%	+30%
triangle ppc	**-86%**	+15%	-30%	+130%
mean	+4%	-35%	+36%	-1%
total	**-61%**	**-80%**	-14%	**-65%**

Table 5. Experimental results for B+LA+SF

Values for `square-4x4` are not reported since they cannot be fairly compared to those of BP. Gains in both time and number of tests vary greatly from a program to another, and the average gain of BP+LA+SF is smaller than the one of BP+LA.

However, BP+LA+SF offers the greater gain on the more demanding examples. It turns out that when considering the total amount of time and tests, BP+LA+SF outperforms largely BP and is significantly better than BP+LA.

6. Related Work

Recent works have addressed the path explosion issue in path-based testing with item coverage objective in various ways. We first discuss methods closer to our work, then we sketch other approaches.

Look-Ahead. The RWSet [4] and the path subsumption [2] heuristics consist in pruning a path exploration when the current state is considered similar to a previously encountered state. This is complementary to LA in the sense that these approaches cut a path according to its execution history while LA cuts a path according to its potential future. The idea of cutting a path according to its potential future is present in the SYNERGY approach [10], mixing concolic execution and abstract reachability set computation. LA can be seen as a special instance with very low computational overhead, where the abstract reachability set records only control and forget anything about data.

Max-CallDepth. Ideas similar to MCD can be found in both the Variably Interprocedural Program Analysis heuristic [26] and concretization of function calls [11, 25]. The Variably Interprocedural Program Analysis heuristic consists in abstracting a function call when the call stack is too high, like MCD. However, the function body is not explored at all: the effect of the abstract function is to return directly an unconstrained value or an unconstrained data store (to model arbitrary side-effects) depending on the user choice. This prevents the procedure from exploring a (single) potentially deep path in the callees like MCD does, however the resulting path predicate is too loose, and a solution is not ensured anymore to exercise the path at runtime. Function call concretization [11, 25] can be seen as a special instance of MCD, where not only backtrack is forbidden in the callees, but input and output values are fixed. The two heuristics can be used in case of calls to native code functions (i.e. libraries) which is not the case for MCD. These three heuristics could all be combined in the same procedure for different situations, depending of the function under consideration, the current call depth or the precision required for the analysis.

Solve-First. Low coverage speed of DFS test generation has already been pointed out in [7, 12, 22] and solutions have been proposed. Hybrid testing [22] consists in breaking the regularity of a DFS with random test generation: during the DFS search, a test data is “sometimes” generated at random and the DFS goes on from the corresponding path. Best-first search [7] is mostly a DFS enhanced with breadth-first aspects: occasionally all active choice points are ranked according to some internal heuristic and the *best* branch is expanded. Generational search [12] computes all potential new paths from a given execution, and all potential new paths from all active executions are ranked, then the best one is expanded.

SF is mostly a DFS and only one path is active at a time like in [22]. All path successors are computed at once like in [12]. However, SF does not involve any ranking heuristic between potential successors.

Other path-pruning methods. Other lines of work address the problem of path explosion in rather different ways than ours. Path explosion due to thread interleaving in concurrent programs is addressed in [24, 29]. The first work is inspired by *partial orders* methods from (explicit) model checking, while the second work is more *ad hoc*.

Going back to nested function calls, some works have been conducted on *modular test generation* in order to avoid function inlining. Recent approaches are based on function summaries, which may be either discovered during the analysis [1, 15] or provided manually [21]. Functions can also be handled lazily [1].

7. Conclusion

Path-based testing with item coverage objective is a powerful paradigm allowing to automatically build test suites for real size programs. One of the main drawback of this technique is the path explosion phenomenon. However

many paths are irrelevant for full item coverage and can be discarded. This paper discusses three heuristics to address this issue in common cases of irrelevancy. All these heuristics are lightweight, both in terms of implementation cost and computational overhead, and they are all complementary since each one tackles a particular source of path explosion. We provided experimental evidence that they have a significant impact on the number of paths considered and overall performances. The three techniques have been presented in a DFS framework. However, they are easy to adapt to other path-based frameworks. Considering their ease of implementation and their effective pruning power, we believe that their integration should be considered in any path-based testing tool.

There are several directions for future work. First, one can wonder how heuristics developed here combine with heuristics from the literature. For example, the following combinations seem promising: Hybrid Testing [22] and Solve-First to increase initial covering speed, RWSet [4] and Look-Ahead to cut early the path exploration and Variably Interprocedural Program Analysis [26] and Max-CallDepth to alleviate the cost of deep function calls. Second, it would be interesting to study whether the methods developed here for the constraint-based paradigm can be helpful in a search-based setting, and how they compare with existing work [16].

References

[1] S. Anand, P. Godefroid and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *TACAS 2008*. Springer.

[2] S. Anand, C. S. Pasareanu and W. Visser. Symbolic Execution with Abstract Subsumption Checking. In *SPIN 2006*. Springer.

[3] S. Anand, C. S. Pasareanu and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *TACAS 2007*. Springer.

[4] P. Boonstoppel, C. Cadar and D. R. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *TACAS 2008*. Springer, 2008.

[5] S. Bardin and P. Herrmann. Structural Testing of Executables. In *IEEE ICST 2008*. IEEE.

[6] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN 2005*. Springer.

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

[8] A. Gotlieb, B. Botella and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *ISSTA 1998*. ACM.

[9] A. Gotlieb, B. Botella and M. Watel. Inka: Ten years after the first ideas. In *ICSSEA 2006*.

[10] B. Gulavani, T. A. Henzinger, Y. Kannan, A. Nori and S. K. Rajamani. Synergy: A new algorithm for property checking. In *FSE 2006*, ACM.

[11] P. Godefroid, N. Klarlund and K. Sen. DART: Directed Automated Random Testing. In *PLDI'2005*. ACM.

[12] P. Godefroid, M. Y. Levin and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS 2008*.

[13] N. Gupta, A. P. Mathur and M. L. Soffa. Automated Test Data Generation Using an Iterative Relaxation Method. In *FSE 1998*.

[14] N. Gupta, A. P. Mathur and M. L. Soffa. UNA Based Iterative Test Data Generation and its Evaluation. In *ASE 1999*. IEEE.

[15] P. Godefroid. Compositional dynamic test generation. In *POPL 2007*. ACM.

[16] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn and J. Wegener . The Impact of Input Domain Reduction on Search-Based Test Data Generation. In *FSE 2007*. ACM.

[17] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA 2007*.

[18] B. Korel. Automated Software Test Data Generation. In *IEEE TSE*. IEEE, 1990.

[19] B. Korel. Automated Test Data Generation for Programs with Procedures. In *ISSTA 1996*.

[20] D. Kroening and O. Strichman. Decision Procedures: An Algorithmic Point of View. Springer, 2008.

[21] P. Mouy, B. Marre, N.Williams and P. Le Gall. Generation of All-Paths Unit Test with Function Calls. In *ICST 2008*. IEEE, 2008.

[22] R. Majumdar and K. Sen. Hybrid Concolic Testing. In *ICSE 2007*. IEEE.

[23] F. Nielson, H. R. Nielson and C. Hankin. Principles of Program Analysis. Springer, 1999.

[24] K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *FASE 2006*. Springer.

[25] K. Sen, D. Marinov and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE'2005*. ACM.

[26] A. Tomb, G. P. Brat and W. Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA 2007*. ACM, 2007.

[27] N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In *TAP 2008*. Springer.

[28] N. Williams, B. Marre and P. Mouy. On-the-Fly Generation of K-Path Tests for C Functions. In *ASE 2004*. IEEE.

[29] C. Wang, Z. Yang, V. Kahlon and A. Gupta. Peephole Partial Order Reduction. In *TACAS 2008*. Springer.