On Complexity of All-Paths Test Generation. From Practice to Theory

Nikolai Kosmatov
CEA LIST
Software Safety Laboratory, PC 94
91191 Gif-sur-Yvette France
Nikolai.Kosmatov@cea.fr

Abstract

Automatic structural testing of programs becomes more and more popular in software engineering. Among the most rigorous structural coverage criteria, all-paths coverage *requires to generate a set of test cases such that every feasible execution path of the program under test is executed by one test case. This article addresses different aspects of computability and complexity of constraint-based all-paths test generation for C programs from the practitioner's point of view, and tries to bridge the gap between mathematical theory and practical computation problems arising in this domain.*

We focus on two particular classes of programs important for practice. We show first that for a class containing the simplest programs with strong restrictions, all-paths test generation in polynomial time is possible. For a wider class of programs in which inputs may be used as array indices (or pointer offsets), all-paths test generation is shown to be NP-hard. Some experimental results illustrating test generation time for programs of these classes are provided.

Keywords: *all-paths test generation, computability, complexity.*

1 Introduction

Testing is nowadays the primary way to improve the reliability of software. Software testing costs may achieve up to 50% of the total cost of software development. Automatic test generation helps to reduce this cost. The increasing demand has encouraged much research on automation of software testing.

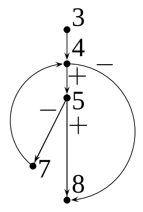
In constraint-based test generation, commonly used since 1990's, the program under test (or its formal model) and the desired coverage criterion are first translated into a constraint solving problem. Then a constraint solver is called to solve the constraints and to provide *a test case*, i.e. input values for all input variables, which may be accompanied by *an oracle,* i.e. the expected behavior of the program on this input.

Among other novel techniques, various combinations of concrete and symbolic execution were developed during the last five years. They were successfully applied in implementation of several testing tools for C programs: PathCrawler [2, 15, 16], DART [5], CUTE [13], EXE [3]. These techniques appeared to be particularly beneficial in *path-oriented* testing, according to the classification of [4]. For example, the *all-paths test coverage criterion* [18] requires to generate a set of test cases such that every possible execution path of the program under test is executed by one test case. The number of possible inputs is assumed to be finite (otherwise the number of paths may be unlimited, *Cf* Section 3). This criterion being very strong and often unreachable, weaker path-oriented criteria were proposed, requiring to cover only paths of limited length, or with limited number of loop iterations, etc. The paths are often explored in depth-first search [5, 13, 15, 16], sometimes in breadth-first search [17] or by mixed heuristics [3]. When path coverage is too strong for the program under test, one may use the *all-statements criterion* (every reachable statement must be executed by some test case) and the *all-branches criterion* (every reachable edge must be executed) [18].

In the context of increasing applications of automatic test generation in industry, the test engineer today is often unable to evaluate the computability and complexity of automatic test generation with a particular test criterion for a given program. The related theoretical results, often difficult to find and to understand for a practitioner, usually consider the most general case and give only negative answers. However, particular programs encountered in practice are seldom as much complicated as "the worst case" considered by the theorists. A detailed study of complexity issues

```
1  #define D 4
2  int atU( int x[D], int y[D], int u ) {
3    int i = 1;
4    while( i < D ) {
5      if( u < x[i] )
6        break;
7      i++; }
8    return y[i-1];
9  }
```

(a)



(b)

Figure 1. (a) Function `atU`, and (b) its control-flow graph

of test generation for different types of programs and criteria, depending on the features used in the program, may seem of little interest to a theorist, but will be extremely useful for a practitioner.

The motivation of this paper is to initiate such a study of computability and complexity of automatic test generation for particular classes of programs that may appear in practice. In this paper, we focus on the all-paths coverage criterion. We present first the depth-first all-paths test generation and illustrate it on a running example (Section 2).

Contributions. Our main contribution is to consider two classes of programs. In the first case (Section 3), under appropriate restrictions on the size, the number and the form of the resulting constraints, we show that all-paths test generation in polynomial time is possible (Theorem 2). This result is based on Pratt's method [11, 12] of solving difference constraints in polynomial time. Formally speaking, we only show that the total time of constraint solving is polynomial, but this phase appears to be the only expensive step in the test generation method in practice. We deduce polynomial complexity of test generation for weaker coverage criteria, such as all-branches and all-statements criteria (Corollary 3).

Next (Section 4), we consider a wider class of programs which may contain in addition input variables used as array indices (or pointer offsets) and constraints with $\neq$. We give a simple sketch of proof by an original reduction from the Hamiltonian cycle problem that all-paths test generation for such programs may be NP-hard (Theorem 5), so all-paths test generation for such programs in polynomial time is not possible (unless P=NP). Section 5 provides experimental results of all-paths test generation for some classes of programs considered in Sections 3 and 4. We finish by a conclusion and future work (Section 6).

To make this paper easily understandable for a specialist in computational complexity theory as well as for a practitioner in software testing, we recall some notions of both domains, give a simplified presentation of the depth-first all-paths test generation and provide in Section 4 a sketch of proof using a simple C program rather than a more formal proof in terms of Turing machines. The reader will find an introduction to the theory of computation in [6]. For more information on constraint-based software testing, we refer the reader to [8] and references in [8].

2 All-Paths Test Generation in Depth-First Search

In this section, we briefly describe a simplified PathCrawler-like method for generation of all-paths tests for C programs. We consider C programs with integer types, arrays, pointers (where input variables do not appear in indices or offsets), conditionals and loops. Similar methods were used in other tools as DART [5] and CUTE [13]. Let us denote the C function under test by f.

The PathCrawler tool [2, 15, 16] is developed at CEA LIST and contains two main modules. The first one, based on the CIL library [10], translates the instructions of the C source code of f into constraints and creates its *instrumented version* whose execution on any test case traces the execution path in f. Next, the user may modify some default test parameters and provides *a precondition*, i.e. the conditions on the inputs of f for which the behavior of f is defined and must be tested. The second module, *test generator*, is implemented in Prolog language. It reads the constraints of f and the precondition, and generates test cases satisfying the all-paths criterion. The program paths are explored in a depth-first search. The generator is based on an original combination of constraint-based symbolic execution and concrete execution of instrumented code. Symbolic execution translates the test generation problem for a (partial) program path into constraints and calls a constraint solver to generate a test case executing this path. Concrete execution (of compiled instrumented code) allows to quickly find the program path executed by some test case. PathCrawler uses COLIBRI, an efficient constraint solver developed at CEA LIST and shared with GATeL [9] and

(1) Mem.	Constraints
$\mathtt{x}[j] \mapsto X_j$	$\langle$precond$\rangle$
$\mathtt{y}[j] \mapsto Y_j$	
$\mathtt{u} \mapsto U$	
$\pi = \epsilon$	

$\rightsquigarrow$

Test 1:	$U = 2$
$X_0 = 0$	$Y_0 = 19$
$X_1 = 5$	$Y_1 = 17$
$X_2 = 11$	$Y_2 = 5$
$X_3 = 16$	$Y_3 = 15$
$\sigma = 4^+, 5^+$	

$\longrightarrow$

(2) Mem.	Constraints
$\mathtt{x}[j] \mapsto X_j$	$\langle$precond$\rangle, 1 < 4,$
$\mathtt{y}[j] \mapsto Y_j$	$U \geq X_1$
$\mathtt{u} \mapsto U$	
$\mathtt{i} \mapsto 1$	
$\pi = 4^+, 5^-_\star$	

$\rightsquigarrow$

Test 2:	$U = 4$
$X_0 = 1$	$Y_0 = 1$
$X_1 = 4$	$Y_1 = 7$
$X_2 = 9$	$Y_2 = 10$
$X_3 = 14$	$Y_3 = 9$
$\sigma = 4^+, 5^+$	

$\longrightarrow$

(3) Mem.	Constraints
$\mathtt{x}[j] \mapsto X_j$	$\langle$precond$\rangle, 1 < 4,$
$\mathtt{y}[j] \mapsto Y_j$	$U \geq X_1, 2 < 4,$
$\mathtt{u} \mapsto U$	$U \geq X_2$
$\mathtt{i} \mapsto 2$	
$\pi = 4^+, 5^-_\star, 4^+, 5^-_\star$	

$\rightsquigarrow$

Test 3:	$U = 19$
$X_0 = 2$	$Y_0 = 8$
$X_1 = 4$	$Y_1 = 5$
$X_2 = 16$	$Y_2 = 2$
$X_3 = 20$	$Y_3 = 4$
$\sigma = 4^+, 5^+$	

$\longrightarrow$

(4) Mem.	Constraints
$\mathtt{x}[j] \mapsto X_j$	$\langle$precond$\rangle, 1 < 4,$
$\mathtt{y}[j] \mapsto Y_j$	$U \geq X_1, 2 < 4,$
$\mathtt{u} \mapsto U$	$U \geq X_2, 3 < 4,$
$\mathtt{i} \mapsto 3$	$U \geq X_3$
$\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+, 5^-_\star$	

$\rightsquigarrow$

Test 4:	$U = 18$
$X_0 = 1$	$Y_0 = 0$
$X_1 = 11$	$Y_1 = 7$
$X_2 = 15$	$Y_2 = 9$
$X_3 = 18$	$Y_3 = 11$
$\sigma = 4^-$	

$\longrightarrow$

(5) Mem.	Constraints
$\mathtt{x}[j] \mapsto X_j$	$\langle$precond$\rangle, 1 < 4,$
$\mathtt{y}[j] \mapsto Y_j$	$U \geq X_1, 2 < 4,$
$\mathtt{u} \mapsto U$	$U \geq X_2, 3 < 4,$
$\mathtt{i} \mapsto 4$	$U \geq X_3, 4 < 4$
$\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+_\star$	

$\rightsquigarrow$ $\langle$unsatisfiable$\rangle$

$\longrightarrow$

(6) Mem.	Constraints
$\mathtt{x}[j] \mapsto X_j$	$\langle$precond$\rangle, 1 < 4,$
$\mathtt{y}[j] \mapsto Y_j$	$U \geq X_1, 2 < 4,$
$\mathtt{u} \mapsto U$	$U \geq X_2, 3 \geq 4,$
$\mathtt{i} \mapsto 3$	
$\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^-_\star$	

$\rightsquigarrow$ $\langle$unsatisfiable$\rangle$

$\longrightarrow$ $\dots$

Figure 2. Depth-first generation of all-paths tests for the function `atU` of Figure 1

OSMOSE [1] testing tools.

We assume that the program under test has at most one instruction per line and one condition per decision. (The first step of the PathCrawler tool transforms multiple conditions into simple ones by introducing additional conditional instructions.) We denote a program path ρ by a sequence of line numbers, e.g.

$$\rho = 3, 4^+, 5^-, 7, 4^+, 5^+, 6, 8$$

is a path for the program of Figure 1. For control points (in conditional or loop statements), the line number is followed by a "$+$" if the condition is true, and by a "$-$" otherwise.

Since a program path is uniquely determined by its decisions, we may abbreviate a path by the sequence of its decisions only, e.g. $\rho = 4^+, 5^-, 4^+, 5^+$. We will mostly use such abbreviated notation. The empty path is denoted by ϵ.

The mark "$\star$" after a decision will indicate that the depth-first search has already completely explored its negation, i.e. the other branch in the tree of all execution paths. For example, the mark "$\star$" in the path $\rho = 4^+, 5^-_\star, 4^+, 5^+$ means that we have already explored all paths of the form $4^+, 5^+, \ldots$ and tried to generate a test case for each of them.

During a test generation session, the generator maintains the following data:

- a table representing the program memory at every moment of symbolic execution. It can be seen as a mapping $Symb \mapsto Val$ which associates a value Val to a symbolic name $Symb$. The symbolic name $Symb$ may denote a variable name or an array element. The value Val may be a constant or a Prolog logical variable.
- a partial program path π in f. If a test case is successfully generated for the partial path π, then σ will designate the remaining part of the complete path it executes.
- a constraint store containing the constraints collected by the symbolic execution of the current partial path π.

We can now describe the test generation method. It contains the following steps:

(Init) Create a logical variable for each input and associate it with the input. Set the initial values of initialized variables. Add constraints corresponding to the precondition. Let the initial partial path π be empty. Continue to (Step 1).

(Step 1) Let σ be empty. The generator symbolically executes the partial path π, that is, adds constraints and updates the memory table according to the instructions in π. If some constraint fails, continue to (Step 4). Otherwise, continue to (Step 2).

(Step 2) The constraint solver is called to generate a test case, that is, concrete values for the inputs, satisfying the current constraints of the constraint store. If it fails, go to (Step 4). Otherwise, continue to (Step 3).

(Step 3) Execute the instrumented version of the program on the test case generated in the previous step to trace the complete execution path. The complete path must start by π (by definition of the constraint solving problem for which the test case was generated). Write the remaining part of the path into σ. Continue to (Step 4).

(Step 4) Let ρ be the concatenation of π and σ. Try to find in ρ the last unmarked decision, i.e. the last decision without a "$\star$" mark. If ρ does not contain any unmarked decision, exit. Otherwise, if $x^\pm$ is the last unmarked decision in ρ, let π be the subpath of ρ before $x^\pm$, followed by $x^\mp_\star$ (i.e. the negation of $x^\pm$ marked as already processed), and continue to (Step 1).

We see that Step 4 chooses the next partial path in a depth-first search. It changes the last unmarked decision in ρ to look for differences as deep as possible first, and marks a decision with a "$\star$" when its negation (i.e. the other branch from this node in the tree of all execution paths) has already been fully explored. For example, if

$$\rho = a^-_\star, b^+, c^+, d^-_\star, e^+_\star,$$

the last $\star$ means that the depth-first search has already processed all paths of the form

$$a^-, b^+, c^+, d^-, e^-, \ldots$$

The previous $\star$ (in $d^-_\star$) means that the depth-first search has already processed all paths of the form

$$a^-, b^+, c^+, d^+, \ldots$$

The last unmarked decision in ρ is c^+, so Step 4 will take the subpath of ρ before this decision $a^-_\star, b^+$, and add $c^-_\star$ to obtain the new partial path $\pi = a^-_\star, b^+, c^-_\star$. Notice that this way to mark conditions with a "$\star$" keeps the information that some shorter partial paths have already been fully explored (here, paths of the form $a^+, \ldots$ are fully explored), and adds this information for the negation of the last condition (here, the paths $a^-, b^+, c^+, \ldots$ are fully explored).

We illustrate this method on the example of function `atU` of Figure 1. This function is the simplest form of interpolation. It takes three parameters, two arrays `x`, `y` (each one with `D` integers) and an integer `u`. Let us define the precondition $\psi_{\texttt{atU}}$ of `atU` as follows:

$$\begin{gathered} \texttt{D} \geq 1, \quad \texttt{x} \text{ contains } \texttt{D} \text{ elements}, \\ \texttt{y} \text{ contains } \texttt{D} \text{ elements}, \\ 0 \leq \texttt{x}[0] < \texttt{x}[1] < \cdots < \texttt{x}[\texttt{D}-1] \leq \mathrm{Max}, \\ \texttt{x}[0] \leq \texttt{u} \leq \texttt{x}[\texttt{D}-1], \\ 0 \leq \texttt{y}[0],\ \texttt{y}[1],\ \ldots,\ \texttt{y}[\texttt{D}-1] \leq \mathrm{Max}. \end{gathered} \qquad (\psi_{\texttt{atU}})$$

The values $\texttt{y}[j]$ are supposed to be the values of some function h at the points $\texttt{x}[j]$, i.e. $h(\texttt{x}[j]) = \texttt{y}[j]$, $0 \leq j \leq \texttt{D}-1$. The function `atU` returns the value of h in the closest point to the left of `u` in which the value of h is known. In other words, it finds the greatest k with $\texttt{x}[k] \leq \texttt{u}$ and returns $\texttt{y}[k]$. Max is a positive constant (for example, the maximal integer MAXINT of the system). For simplicity of our example, we assume $\texttt{D} = 4$ and $\mathrm{Max} = 20$.

The test generation session for the function `atU` is shown in Figure 2, where "$\rightsquigarrow$" denotes the application of (Step 2) and (Step 3), and "$\rightarrow$" the application of (Step 4) and (Step 1). First, (Init) creates logical variables X_j, Y_j ($0 \leq j \leq 3$) and U to represent the inputs as shown in (1) of Figure 2. The first two lines of ($\psi_{\texttt{atU}}$) being now satisfied, (Init) adds into the constraint store the $3\texttt{D} + 3$ inequalities corresponding to the last three lines of ($\psi_{\texttt{atU}}$), which are denoted by $\langle$precond$\rangle$ in Figure 2:

$$\begin{gathered}0 \leq X_0,\ X_0 < X_1,\ X_1 < X_2,\ X_2 < X_3,\ X_3 \leq 20,\\ X_0 \leq U,\ U \leq X_3,\\ 0 \leq Y_0,\ \dots\ ,\ 0 \leq Y_3,\\ Y_0 \leq 20,\ \dots\ ,\ Y_3 \leq 20.\end{gathered}$$

The first partial path π being always empty, (Step 1) has nothing to do now. Next, (Step 2) generates the first test case, Test 1. (Step 3) executes Test 1 on the instrumented version of the program and obtains $\sigma = 4^+, 5^+$ (that is an abbreviation for $3, 4^+, 5^+$).

We are now going from (1) and Test 1 to (2) in Figure 2. (Step 4) finds $\rho = 4^+, 5^+$, where 5^+ is the last unmarked decision. Therefore, it sets $\pi = 4^+, 5^-_\star$. Next, (Step 1) symbolically executes the partial path π in constraints, node by node, for unknown inputs. The execution of the assignment 3 adds $\texttt{i} \mapsto 1$ to the memory table. The execution of the decision 4^+ adds the constraint $1 < 4$ trivially true, and the execution of the decision 5^- adds the constraint $U \geq X_1$, after replacing the variables `i`, `u` and $\texttt{x}[1]$ by their current values in the memory table. Next, (Step 2) generates Test 2, and (Step 3) executes it and obtains $\sigma = 4^+, 5^+$.

We are now going from (2) and Test 2 to (3) in Figure 2. (Step 4) finds $\rho = 4^+, 5^-_\star, 4^+, 5^+$ and sets $\pi = 4^+, 5^-_\star, 4^+, 5^-_\star$. (Step 1) symbolically executes π, (Step 2) generates Test 3, and so on. Let us now move from (4) and Test 4 to (5) in Figure 2. (Step 4) finds

$$\rho = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+, 5^-_\star, 4^-$$

and sets $\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+_\star$. The last constraint $4 < 4$ added by symbolic execution at (Step 1) is obviously false, so the generator goes directly to (Step 4), which sets $\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^-_\star$. As shows (6) in Figure 2, the last constraint $3 \geq 4$ added by (Step 1) fails again, so the generator continues to (Step 4). The steps after (6) are not shown in Figure 2. Similarly, the generator will try the partial paths $\pi = 4^+, 5^-_\star, 4^-_\star$ and $\pi = 4^-_\star$, which are also infeasible, and stops. A test case was generated for each of the 4 feasible paths.

In general, if during some execution of (Step 1) or (Step 2), the constraints are unsatisfiable and no test case can be generated, then π is infeasible and the algorithm continues the exploration of other paths normally. If it happens at (Init) or at the very first iteration of (Step 2), that is, the precondition is unsatisfiable, then the algorithm stops at (Step 4) since ρ is empty.

3 All-Paths Test Generation in Polynomial Time

In this section, we give sufficient conditions for a class of all-paths test generation problems to be solvable in polynomial time. It is intuitively clear that all-paths test generation for a program may take much time for (one or several of) the following reasons:

(†) the program has a great number of paths and, therefore, results in a great number of constraint solving problems,

(††) the instructions of the program result in complex constraints which cannot be solved fast,

(†††) the program has very long paths giving rise to problems with too many constraints.

We show that under appropriate restrictions for these three issues, all-paths test generation in polynomial time becomes possible.

In practice, the most expensive step of the all-paths test generation method is constraint solving in (Step 2), so we focus on the constraint solving time. Our experience with the PathCrawler tool shows that the other steps (instrumentation, translation into constraints, symbolic execution etc.) are done very efficiently. For example, the first module of PathCrawler, which instruments the program under test and translates it into constraints, takes less than or about 1 minute for programs with hundreds or thousands of lines. Since the performance of these steps depends on many implementation details and appears quite satisfactory in practice, we do not discuss it here in detail. Notice that test generation time in experiments of Section 5 includes all steps of the test generation process, from source code to generated test cases.

We define *an all-paths test generation problem* as

$$\Phi = (P, f, \psi)$$

where P is a C program, f is a function in P to be tested and ψ is a precondition of f. A solution of Φ is a set of test cases satisfying the all-paths criterion. The precondition may contain information necessary for correct initialization of test generation (e.g. input array sizes, domain of variables, etc.) and any other conditions on the input variables restricting admissible inputs of f. We denote by $L_P^\Phi > 0$ the length of the program P.

The number of possible inputs must be finite (and not only bounded by the available computer memory, which is assumed sufficiently large). Indeed, if the number of inputs is unlimited, the number of paths may be unlimited and test generation of all-paths tests will not terminate. It happens

```
char LastChar(char str[]) {
  while( *(str + 1) != 0 )
    str = str + 1;
  return * str;
}
```

Figure 3. Function `LastChar` returns the last non-zero symbol in a given non-empty string

for the function `LastChar` of Figure 3, which takes a non-empty zero-terminated string and returns the last non-zero symbol.

Therefore, we assume that ψ bounds by some $L_I^\Phi > 0$ the maximal length of admissible inputs for Φ, measured in number of bytes, or up to a constant, of integers.

We define *a system of difference constraints* as a system of constraints of the form

$$x - y \leq c, \quad \text{or} \quad x \leq c, \quad \text{or} \quad x \geq c,$$

where x, y are integer variables and c is an integer. An equality $x - y = c$ can be represented as $x - y \leq c$ and $y - x \leq -c$.

Theorem 1 ([11, 12]) *A system of difference constraints may be solved in polynomial time $g(m, n)$, where $g(X, Y)$ is a polynomial, n is the number of variables and m is the number of constraints.*

The reader will find various estimates g in [12]. We are now ready to state the main result of this section. Notice that the conditions (i), (ii), (iii) precisely correspond to the three reasons (†), (††), (†††) stated above.

Theorem 2 *Let $\mathcal{C}$ be a class of all-paths test generation problems, and $g_1(X, Y)$, $g_2(X, Y)$ two polynomials. Suppose that every problem $\Phi = (P, f, \psi)$ of $\mathcal{C}$ satisfies the following properties:*

(i) *the number of program paths in Φ for which the method will try to generate a test case is bounded by $g_1(L_P^\Phi, L_I^\Phi)$,*

(ii) *symbolic execution of any program path (including the precondition) adds only difference constraints,*

(iii) *symbolic execution of any program path (including the precondition) adds at most $g_2(L_P^\Phi, L_I^\Phi)$ constraints.*

Then there exists a polynomial $g_3(X, Y)$ such that the total constraint solving time in the all-paths test generation for Φ is bounded by $g_3(L_P^\Phi, L_I^\Phi)$.

```
#define D 4
int bsearch(int a[D], int key) {
  int low = 0; int high = D-1;
  while (low <= high) {
    int mid = low + (high-low)/2;
    int midVal = a[mid];
    if (midVal < key)
      low = mid+1;
    else if (midVal > key)
      high = mid-1;
    else
      return mid;
  }
  return -1;
}
```

Figure 4. Function `bsearch` for binary search of a given element `key` in a given sorted array `a` of length `D`

Sketch of proof. Assume without loss of generality that $g(X, Y)$ is monotonic in each argument for $X > 0$, $Y > 0$. Let $\Phi = (P, f, \psi)$ be an all-paths test generation problem of $\mathcal{C}$. By (i), the all-paths test generation method for Φ solves at most $g_1(L_P^\Phi, L_I^\Phi)$ constraint solving problems. By (ii), the constraint solving problem created for any program path (hence, for any partial path) of Φ is a system of difference constraints. By (iii), the number of constraints in the system $m \leq g_2(L_P^\Phi, L_I^\Phi)$. The number of variables n is bounded by L_I^Φ. It follows that the total constraint solving time is bounded by

$$g_1(L_P^\Phi, L_I^\Phi) g(m, n) \leq g_3(L_P^\Phi, L_I^\Phi),$$

where $g_3(X, Y) := g_1(X, Y) g(g_2(X, Y), Y)$. $\square$

We intentionally allow to bound the number of paths and constraints by the length of the program, or the length of the input, or both, because different estimates may be useful in different examples. Notice also that the number of paths mentioned in (i) includes infeasible partial paths like those seen in Section 2, but does not include several infeasible paths starting with the same infeasible partial path. Indeed, the depth-first method adds at most one new constraint to those of a feasible partial path, so it never tries to generate a test case for two longer paths starting with the same infeasible partial path.

Let us apply Theorem 2 to an example. Consider the family of all-paths test generation problems $\Phi_{\mathtt{D}} = (P_{\mathtt{D}}, \mathtt{atU}, \psi_{\mathtt{atU}})$, where $\mathtt{D} > 0$ is a parameter, $P_{\mathtt{D}}$ is the program of Figure 1 containing the function `atU`, and the precondition $\psi_{\mathtt{atU}}$ is defined in Section 2. The number of input variables in $\Phi_{\mathtt{D}}$ is $2\mathtt{D} + 1$, so $L_I^{\Phi_{\mathtt{D}}} = 2\mathtt{D} + 1$. The number of partial paths in $\Phi_{\mathtt{D}}$ for which the method will try to generate a test case is equal to $2\mathtt{D} \leq L_I^{\Phi_{\mathtt{D}}}$ as required by (i). Symbolic execution of paths for $\Phi_{\mathtt{D}}$ adds only difference constraints (hence (ii) is satisfied), with $3\mathtt{D}+3$ constraints for the precondition and $2\mathtt{D}-1$ constraints for the longest path, so $(3\mathtt{D}+3)+(2\mathtt{D}-1) \leq 3L_I^{\Phi_{\mathtt{D}}}$ as required by (iii). The conditions of Theorem 2 are verified, so all-paths test generation in polynomial time is possible for this example.

Similarly, this theorem may be applied for other interpolation functions occurring in practice, or other search functions in an array, such as the function `bsearch` of binary search in a sorted array given in Figure 4. The function `bsearch` performs a classical binary (dichotomic) search of a given element `key` in a given sorted array `a` of length `D`. It returns the index of some occurence of `key` in `a`, or -1 if `key` does not appear in `a`. At first glance, the assignment of line 5 in Figure 4 provides a constraint that is not a difference one. In fact, for any given program path, the right-hand side of the assignments in lines $3, 5, 8, 10$ contain only constants and no input variables, so they only require a direct computation of the new value of a variable and do not add a constraint on input variables to the constraint store during symbolic execution of a partial program path.

Since all-paths coverage criterion subsumes other criteria such as all-branches coverage or all-statements coverage [18], the following result immediately follows from Theorem 2.

Corollary 3 *Let $\mathcal{C}$ be a class of all-paths test generation problems satisfying the conditions of Theorem 2. Then the total constraint solving time necessary to generate tests for all-branches (or all-statements) coverage is polynomial.*

4 All-Paths Testing with Internal Aliases is NP-hard

In this section we consider a wider class of all-paths test generation problems, where constraints with $\neq$ are allowed, and array indices (or pointer offsets) may depend on input variables. Presence of unknown indices (or offsets) during symbolic execution with unknown inputs leads to the problem of *internal aliases*, as we called them in [7]. Indeed, if `j` is an input variable or was assigned a value that depends on some input variable, the expression `a[j]` is a non-trivial alias for one of the elements of `a`. Using input variables `p[j]` as indices in the array `G` in lines 26, 28 of Figure 5a is another example of internal aliases. All-paths test generation for programs with internal aliases was considered in [7] and an extension of the method of Section 2 for such programs was proposed.

We will use the well-known Hamiltonian cycle problem. *A Hamiltonian cycle* in a directed graph G is a cycle which visits each vertex of G exactly once and returns to the starting vertex. For example, Figure 5b represents a directed graph with five vertices $\{0,1,2,3,4\}$ which has a Hamiltonian cycle. We may identify a directed graph with its *adjacency matrix*. Its element $G(i,j)$ is 1 if G has an arc from i to j, and 0 otherwise. We prefer here the mathematical notation $G(i,j)$, $p(i)$ (in roman font) to the C notation `G[i][j]`, `p[i]` (written in TrueType). In lines 5–11 of Figure 5a, G is the graph of Figure 5b (with $N=5$ vertices) defined by its adjacency matrix. The first loop in the function HC checks that the elements of p are in $\{0,1,\dots,N-1\}$ and are all different (lines 15–23). It means that p is a bijection of $\{0,...,N-1\}$ onto itself, or *a permutation of* $\{0,...,N-1\}$. HC returns 1 if p is a permutation of vertices of G and

$$p(0) \to p(1) \to \cdots \to p(N-1) \to p(0)$$

is a Hamiltonian cycle in G, and 0 otherwise (lines 25–32). The precondition ψ_{HC} is defined as:

$$\begin{array}{c} p \text{ contains } N \text{ elements,} \\ 0 \leq p(j) \leq MAXINT. \end{array} \qquad (\psi_{\mathrm{HC}})$$

We assume the following Conjecture 4, a consequence of the famous $P \neq NP$ conjecture strongly believed to be true, and state the main result of this section.

Conjecture 4 ([6, Section 10.4.4]) *There is no algorithm deciding in polynomial time if a given directed graph has a Hamiltonian cycle.*

Theorem 5 *There exists no polynomial-time algorithm for all-paths test generation problems for programs with internal aliases.*

Sketch of proof. Assume the contrary. Let A be a polynomial-time algorithm that, given an all-paths test generation problem $\Phi = (P, f, \psi)$, generates a list

$$(t_1, \rho_1), \dots, (t_k, \rho_k)$$

where t_i is a test case, ρ_i is the execution path activated by executing f on t_i, and $\rho_1, \dots, \rho_k$ are all feasible paths of f. The "polynomial-time algorithm" means that there exist $K, m > 0$ such that the number of steps of A is always bounded by the polynomial $K(L_P^{\Phi} + L_I^{\Phi})^m$, where L_P^{Φ} is the length of P and L_I^{Φ} is the maximal input size.

Then we can construct another algorithm B which, given the adjacency matrix G of a directed graph and its number of vertices N,

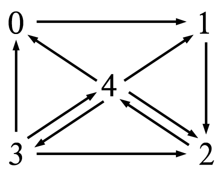
$B1)$ constructs a program P_G similar to that of Figure 5a with the given N and G,

$B2)$ executes A on the problem $\Phi_G = (P_G, \mathrm{HC}, \psi_{\mathrm{HC}})$,

$B3)$ says "yes" if A has generated a test for the path returning 1, and "no" otherwise.

```
#define N 5 // number of vertices in graph G
typedef int graph[N][N];
typedef int perm[N];
// graph G is defined by its adjacency matrix :
graph G = {
  0,1,0,0,0,
  0,0,1,0,0,
  0,0,0,0,1,
  1,0,1,0,1,
  1,1,1,1,0
};

int HC(perm p){
  int i, j;
  for( i = 0; i < N; i++ ){
    if( p[i] < 0 )
      return 0;
    if( p[i] > N-1 )
      return 0;
    for( j = i+1; j < N; j++ )
      if( p[i] == p[j] )
        return 0;
  }
// we checked that p is a permutation of {0,...,N-1}
  for( i = 1; i < N; i++ )
    if( G[ p[i-1] ][ p[i] ] != 1 )
      return 0;
  if( G[ p[N-1] ][ p[0] ] != 1 )
    return 0;
// we checked that p defines the Hamiltonian cycle
// p(0) -> p(1) -> ... -> p(N-1) -> p(0) in G
  return 1;
}
```



(a) **(b)**

Figure 5. a) For the graph G with N vertices, statically defined by its adjacency matrix, the function HC checks if p is a permutation of vertices defining the Hamiltonian cycle $p(0) \to p(1) \to \cdots \to p(N-1) \to p(0)$. b) The graph G has the Hamiltonian cycle $0 \to 1 \to 2 \to 4 \to 3 \to 0$.

The size of the input (N, G) of B is proportional to N^2:

$$|G| \le |(N,G)| \le 2|G|, \ |G| \sim N^2.$$

We claim that B is a polynomial-time algorithm. Indeed, for some constants $K_j > 0$, Part $B1$ copies some pieces of text whose length is $\le K_1 N^2$. Part $B2$ executes the algorithm A on the problem Φ_G with $(L_P^\Phi + L_I^\Phi) \le K_2 N^2$, so it takes $\le K(K_2 N^2)^m$ steps. The function HC has $\le K_3 N^2$ feasible paths, and the length of each path is $\le K_4 N^2$. Part $B3$ reads the list of generated tests which may contain $\le K_3 N^2$ couples (t_i, ρ_i) each of length $\le K_5 N^2$. So B makes $\le K_6 N^{2m} + K_7 N^4 + K_8 N^2$ steps.

It is clear that the path returning 1 in the function HC of P_G is feasible if and only if G has a Hamiltonian cycle. Therefore B is a polynomial-time algorithm deciding if a given directed graph G has a Hamiltonian cycle. The contradiction with Conjecture 4 finishes the proof. $\square$

Theorem 5 is true even for the simplest programs, such as that of Figure 5a, where

- the number of paths may be bounded by a polynomial in program size, so we have a restriction for (†),

D	atU		bsearch	
	tests	time	tests	time
4	4	0.50 s	9	0.49 s
10	10	0.52 s	21	0.56 s
50	50	1.38 s	101	2.91 s
100	100	6.06 s	201	12.10 s
500	500	5 m 39 s	1001	6 m 50 s
1000	1000	30 m 46 s	2001	32 m 47 s

Figure 6. Results of all-paths test generation for functions `atU` of Figure 1 and `bsearch` of Figure 4 for different array sizes `D`.

N	HC		
	paths	tests	time
4	15	38	0.66 s
5	21	140	2.10 s
6	28	747	19.75 s
7	36	5 075	4 m 33 s
8	45	40 364	26 m 58 s
9	55	362 934	4 h 2 m 24 s

Figure 7. Results of all-paths test generation for function HC of Figure 5a (with complete graphs $\mathrm{G_N}$).

- the path length may be bounded by a polynomial in program size as well, so we have a restriction for $(\dagger\dagger\dagger)$,
- the function under test f contains only integers, arrays, conditionals, assignments and loops with a fixed number of iterations,
- f contains no function calls and no *external aliases* (which appear when f contains pointer inputs and some memory location is reachable in two different ways from the inputs, see [7]).

In this example, the complexity of all-paths test generation is due to $(\dagger\dagger)$, i.e. the form of the constraints, which include internal aliases and $\neq$.

Remark. We have actually shown that the all-paths test generation problem is NP-hard, i.e. at least as difficult as the Hamiltonian cycle problem or any other NP-complete problem. A specialist in complexity theory will notice that our sketch of proof may be transformed into a strict proof, since a computer may be simulated by a Turing machine in polynomial time, and vice versa [6, Section 8.6]. An appropriate representation for N and p will solve the problem of big values overpassing the word length on our computer.

5 Experiments

In this section we provide some experiments with the PathCrawler tool to illustrate the results of all-paths test generation for some classes of programs considered in Sections 3 and 4. The experiments were made on an Intel Core 2 Duo laptop with 1Gb RAM.

Figure 6 shows the experimental results for two functions, `atU` of Figure 1 and `bsearch` of Figure 4, for different values of parameter D. For each value of parameter `D`, the columns "tests" and "time" show the number of generated test cases and test generation time. Here, PathCrawler generates exactly one test for each feasible path, so the number of feasible paths is equal to the number of tests.

We see that test generation time for these functions with the PathCrawler tool grows rather slowly (clearly under-exponentially) with the parameter `D`. So, as it can be expected by Section 3, all-paths test generation remains tractable for such programs with hundreds and even thousands of input variables, and PathCrawler provides an efficient test generation method for these programs.

On the other hand, Figure 7 shows experimental results for the function HC of Figure 5a with the complete ordered graph G_N with N vertices (i.e. in which there is an arc from i to j for any vertices i, j). The column "paths" shows the number of feasible paths. Here, in presence of internal aliases, PathCrawler generates superfluous test cases. (Basically, it introduces additional choice points for aliases and explores all possible combinations when it tries to cover a program path which it cannot cover otherwise, see [7]).

As predicted in Section 4, we see that test generation time grows extremely fast (nearly factorially), and all-paths test generation becomes intractable already for $N > 10$. For incomplete graphs G, the number of tests and generation time are different, but their growth remains over-exponential.

We also tried a very similar example from a piece of industrial code with several hundreds of lines of C code (that we cannot describe here in detail due to intellectual property issues), where inputs were also used as indices in a two-dimensional array. We obtained similar results: all-paths test generation becomes intractable already for programs with about 20 input variables.

6 Conclusion and Future Work

All-paths test generation is often believed intractable, but neither a characterization of programs for which it would be tractable, nor a description of program features that can make test generation intractable were really provided. It seems important to be able to answer these questions.

This paper addresses the problem of evaluation of potential complexity of all-paths test generation for various classes of programs. Using Pratt's result on solving difference constraints [11], we proved a theorem providing sufficient conditions for a class of programs to allow all-paths test generation in polynomial time. It shows for the first time that, contrary to what is often believed, all-paths testing can be tractable for some classes of programs occurring in practice.

We also showed by an original reduction from the Hamiltonian cycle problem that all-paths test generation for a wider class of programs, where array indices and pointer offsets may depend on the inputs, is intractable (NP-hard). It gives a concrete example of program features that can prevent all-paths test generation to be feasible. We met this situation in an industrial example. These results were illustrated by some experiments using the PathCrawler testing tool.

Future work includes more detailed analysis of the effect of various features of programming languages on test generation under different restrictions appearing in practice. The existence of polynomial-time algorithms for solving other types of constraints (such as range constraints) [12, 14] may provide other positive results for test generation that will help to better understand the applicability of all-paths testing in practice.

We can expect that exhaustive all-paths testing will be intractable in various cases, but test generation for other test coverage criteria (all statements, all branches, etc.) may be easier. We believe that this study will help test engineers to anticipate the computability and complexity of test generation and choose an appropriate test coverage criterion for a particular code.

Acknowledgments. The author thanks Sébastien Bardin, Bernard Botella, Mickaël Delahaye, Philippe Herrmann, Bruno Marre and Nicky Williams for useful discussions.

References

[1] S. Bardin and P. Herrmann. Structural testing of executables. In *the First IEEE International Conference on Software Testing, Verification, and Validation (ICST'08)*, pages 22–31, Lillehammer, Norway, April 2008.

[2] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of C programs: Experience with PathCrawler. In *the Fourth International Workshop on the Automation of Software Test (AST'09)*, Vancouver, Canada, May 2009.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *the 13th ACM Conference on Computer and Communications Security (CCS'06)*, pages 322–335, Alexandria, Virginia, USA, November 2006.

[4] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[5] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223, Chicago, IL, USA, June 2005.

[6] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, November 2000.

[7] N. Kosmatov. All-paths test generation for programs with internal aliases. In *the 19th International Symposium on Software Reliability Engineering (ISSRE'08)*, pages 147–156, Redmond, WA, USA, November 2008.

[8] N. Kosmatov. *Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects*, chapter XI: Constraint-Based Techniques for Software Testing. Advances in Intelligent Information Technologies Book Series. IGI Global, 2009. ISBN: 1605667587.

[9] B. Marre and A. Arnould. Test sequences generation from Lustre descriptions : GATeL. In *the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 229–237, Grenoble, France, September 2000.

[10] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *the 2002 International Conference on Compiler Construction (CC'02)*, pages 213–228, Grenoble, France, Apr. 2002.

[11] V. Pratt. Two easy theories whose combination is hard. Technical report, MIT, Cambridge, Massachusetts, USA, September 1977.

[12] G. Ramalingam, J. Song, L. Joskowicz, and R. E. Miller. Solving systems of difference constraints incrementally. *Algorithmica*, 23(3):261–275, 1999.

[13] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *the 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272, Lisbon, Portugal, September 2005.

[14] Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theor. Comput. Sci.*, 345(1):122–138, 2005.

[15] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-paths tests for C functions : towards the automation of grey-box testing. In *the 19th IEEE International Conference on Automated Software Engineering (ASE'04)*, pages 290–293, Linz, Austria, September 2004.

[16] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *the 5th European Dependable Computing Conference (EDCC'05)*, pages 281–292, Budapest, Hungary, April 2005.

[17] Z. Xu and J. Zhang. A test data generation tool for unit testing of C programs. In *the 6th International Conference on Quality Software (QSIC'06)*, pages 107–116, Beijing, China, October 2006.

[18] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.