Drawing uniformly at random in dynamic sets of paths

Frédéric Voisin, M.-C Gaudel

Drawing uniformly at random in dynamic sets of paths

F. Voisin and M.-C. Gaudel

LRI, Univ. Paris-Sud, CNRS, CentraleSupélec
Université Paris-Saclay
Orsay, France 91405,
fv@lri.fr, marieclaude.gaudel@gmail.com
https://www.lri.fr/~fv https://www.lri.fr/~mcg

Abstract. In the case of coverage biased random testing of programs, random generation is used to first draw a set of paths from the control flow graph of the program. Then, some solver is used for trying to derive input values that leads the program to traverse these paths at run time. A well-known problem is that not all paths of the control flow graph correspond to feasible runs. Such paths must be rejected and other paths must be drawn. This is a severe limitation in the case of programs with a high ratio of infeasible paths.
We propose a new technique that uses the information about the infeasible prefixes already detected to prevent any of their extensions from being drawn. Based on uniform drawing from all the paths, our drawing algorithm remains uniform among the paths that do not have a known infeasible prefix. As the number of infeasible paths is often large, their elimination from the subsequent drawings is a substantial improvement w.r.t. the classical rejection method. Preliminary experiments are reported and commented.

Keywords: structural random testing · uniform drawing of paths · infeasible paths.

1 Introduction

We consider the problem of drawing uniformly at random paths of a bounded length from a given graph, avoiding some finite set of infeasible paths. This set is dynamically defined: after each drawing the set of infeasible paths may be extended to exclude additional paths from future drawings. The set of infeasible paths is not known beforehand but evolves with the drawings.

Our motivation comes from the domain of computer program testing. In structural testing, a piece of program code is represented by its Control Flow Graph (CFG) and the quality of a test set for this program is expressed as some covering of structural elements (vertices, edges, subsets of paths, etc) of its CFG. A set of paths is selected that satisfies a given coverage criterion.

Then for each path the program is processed by symbolic execution to obtain a logical formula, called the *path condition*, that states the constraints between the input parameters that must hold for a program run to follow that path. A path condition has the form $\Phi_1 \wedge \Phi_2 \cdots \wedge \Phi_q$. Finding a tuple of values that satisfy this formula, usually with some SMT solver, provides a test case.

In structural random testing [28], this set of paths is selected by random uniform drawing in the set of all paths of length less that some bound fixed by the tester [2, 11]. In contrast to other approaches (input-based random testing or fuzzy testing [23, 16], concolic testing [30, 15, 27, 9]), all paths have the same probability to be drawn. It offers a natural alternative when exhaustiveness is out of reach. Moreover, it provides a probabilistic measure of the quality of a test set (see for instance [11]).

A well-known drawback of testing methods based on selection of paths, being randomised or not, is that not all paths in the CFG correspond to actual runs of the program : *infeasible paths* traverse contradictory conditions and no tuple of input values can make the execution follow these paths. Infeasibility appears during symbolic execution: when traversing a branch of a conditional statement, a new conjunct Φ representing the condition for traversing the edge at that point in the path is added to the current path condition, possibly falsifying it. The path condition is built step-by-step and one can discard a path as soon as its shortest infeasible prefix is detected.

This leads to the following classical rejection method [11] where drawing is performed uniformly at random among all the paths of the CFG, discarding an infeasible path as it comes and drawing a new one hoping for a better chance. However, it is very likely for a piece of code to have infeasible paths and their number can greatly outnumbers the feasible ones. In such cases, the efficiency of the rejection method is severely affected. Moreover, as the ratio feasible/infeasible paths is usually unknown before drawing, it is difficult to anticipate the problem.

In this paper, we present a new method, which uses the information about infeasible prefixes for the drawing itself, precluding the drawing of any path that extends one of the infeasible prefixes that have been already detected. This method provides uniform drawing among all paths without known infeasible prefixes. When the number of infeasible paths is large, this yields a substantial improvement w.r.t. the classical rejection method.

A by-product of this new method is that adding feasible paths already drawn to the set of forbidden prefixes makes it possible to draw uniformly at random without replacement paths of bounded length, presenting some similarities with the work described in [22] for non redundant generation for weighted context-free grammars. In the case of program testing, this gives a way to enumerate all feasible paths of given length when their number is not too large.

The paper is organised as follows: Section 2 illustrates our approach to structural random testing and recalls the recursive method for generating combinatorial structures; Section 3 describes our new drawing algorithm and gives some

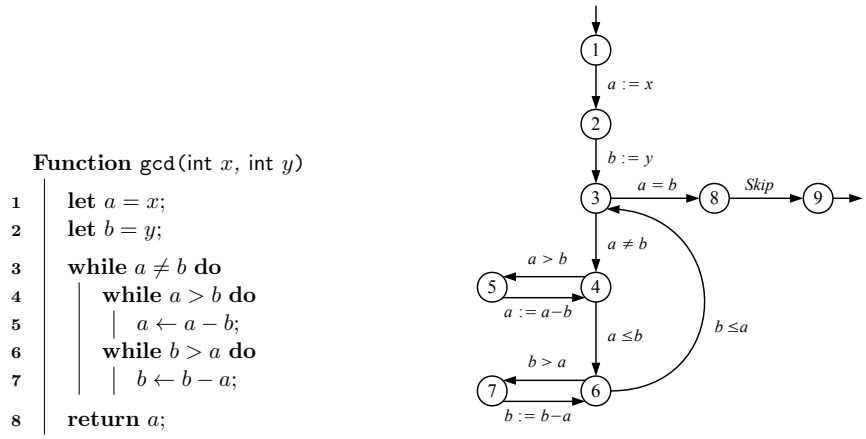

Fig. 1: The *gcd* program and its associated CFG

complexity results; Section 4 reports results of a set of experiments. Section 5 contains a brief comparison with related works and concludes.

2 Preliminaries

2.1 Introducing infeasibilities

We first illustrate the problem on a simple example. Then we recall some characteristics of infeasible paths and sub-paths.

Consider the program from [1] for computing the greatest common divisor: its pseudocode and associated CFG are given in Figure 1.

The CFG contains three groups of infeasible paths:

- paths that enter the external loop but not any of the internal ones,
- paths that traverse the first internal loop, do not enter the second one and finally enter the external loop again,
- paths that exit the external loop, enter it again and do not enter the first internal loop.

Here, the dependencies between the three loops are simple: the set of feasible paths of `gcd` is a regular language [1]. In the CFG there are 15478 paths of maximal length $l = 30$ but only 792 are feasible ones. Although this program is fairly small and the sources of infeasibility of its paths are simple, it illustrates pretty well the problem of infeasible paths for white-box testing: when drawing a path uniformly in the CFG, its probability of being feasible would only approximately be 0.05.

[1] Here finite-state automata techniques or abstraction techniques as those in [1] allow to build a (larger) graph with only feasible paths. See Fig. 3.

This ratio decreases for larger values of l (0.003 and 0.000003, for $l = 50$ and 100 respectively). This illustrates a well-know fact in structural testing: the longer the path, the more likely it is infeasible [31], [26].

We leave to some external procedure to decide whether a path is acceptable or not, keeping our drawing routine generic: it is only notified about the prefixes of paths to exclude. Infeasibility, as characterized above, is merely a specialisation that fits program testing. In the sequel, we use the word infeasible prefix/path for any prefix/path that must no longer be drawn, whatever the reason. The only property of path infeasibility we use is that it is prefix-based: a path is infeasible because it starts with a prefix declared as such and any extension of that prefix is infeasible. The set of infeasible prefixes can be extended after each drawing, but it would make no sense to declare some prefix as infeasible after having allowed some of its extensions: we rely on the fact that notifications report shortest infeasible prefixes.

2.2 Our prototype for structural random testing

Drawing paths in CFG using a specialisation of the classical recursive method for random generation of combinatorial structures [Wilf, Flajolet, and many others] has been first published in [10]. The work in [11] describes coverage-based random exploration of large models described as graphs (CFG, Extended FSM, etc) and [24] investigates several methods of drawing paths uniformly at random in very large models. The method is implemented in the `Rukia` package [2] yielding effective drawing of paths of several hundred edges in graphs with billion of vertices. Related works are reported in [4], [6] and [7].

`Rukia` is a tool for drawing paths in any kind of graphs and does not take into account the semantics of this graph. For structural program testing we have developed the `Auguste` prototype that relies on `Rukia` for the drawing.

In its former version `Auguste` works as follows: in a preamble it constructs the CFG (or some variants of it), translating it to `Rukia`. Then it asks `Rukia` for some number of paths, checks each of them for feasibility with the `Z3` SMT solver [3], discards all infeasible paths, and iterates until reaching the desired number of feasible paths. `Rukia` simply draws the requested number of paths, each one independently from the others, before returning the resulting set.

In the new prototype, `Auguste` and `Rukia` have a client/server relationship: `Rukia` has been modified to react to a set of commands: loading a graph, drawing paths in the current graph, handling a new infeasible prefix or providing the current number of paths in the graph. `Auguste` uses the same preamble as before but asks paths one at a time: whenever it detects that a drawn path is infeasible, it notifies `Rukia` with the shortest infeasible prefix in that path as a request to exclude extensions of that prefix from future drawings. `Rukia` drawing routine has been modified to handle infeasible prefixes, still being uniform on all paths without known infeasible prefixes. The system is depicted in Fig. 2.

[2] http://rukia.lri.fr
[3] https://github.com/Z3Prover/z3

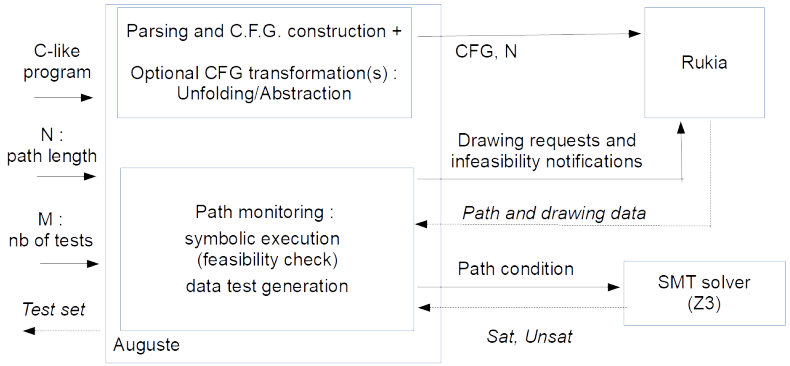

Fig. 2: The `Auguste` prototype for program testing

2.3 Drawing paths with the classical recursive method

We consider a graph $\mathcal{G}$ with (unique) root s_0 and (unique) final vertex s_f. We assume that s_0 has no incoming edge and s_f has no outgoing edge and we want to draw uniformly at random path of length n from s_0 to s_f. In the recursive method, before any drawing `Rukia` computes a table $f(s,l)$, where s is a vertex and l a length, that stores the number of paths from s to s_f of length l. In particular, $f(s_0,n)$ is the number of paths of length n from s_0 to s_f. The length of a path is defined as its number of edges. Note that paths are never empty and always start with s_0.

The computation of f is based on the following relations:

$$f(s_i,j) = \sum_{s_i \to s_k \in \mathcal{G}} f(s_k, j-1) \qquad f(s,0) = 0 \text{ for } s \neq s_f \qquad f(s_f,0) = 1 \quad (1)$$

Given $\mathcal{G}, n, s_0, s_f$ and the table f, Algorithm 1 draws a path p of length n from s_0 to s_f uniformly at random.

Algorithm 1 : drawing uniformly at random a path p of length n

$s = s_0;\ p = s_0;\ l = n;$
while $(l > 0)$ {
  draw s' among the successors s_k of s with probabilities $f(s_k, l-1)/f(s,l)$;
  $s = s';\ p = p.s';\ l = l-1;$
}

The memory space requirement for the table $f(s,l)$ is $O(n \times q)$ where n is the length of the paths and q is the number of vertices of $\mathcal{G}$. The number of

arithmetic operations for its construction is also $O(n \times q)$ [4]. This construction is performed once. Then the time complexity of each drawing is linear in n.

Drawing a path of length $\leq n$ from s_0 to s_f uniformly at random (u.a.r.) is done by adding in $\mathcal{G}$ an edge from s_f to itself, implicitly padding at the end paths that are shorter than n. In the sequel we therefore only consider path with the exact requested length.

In the previous version of our prototype, when an infeasible path is drawn it is simply discarded and a new drawing is performed. This is the classical drawing with replacement model for which numerous mathematical results exist, for instance about the expectation of the number of drawings for collecting the full collection of paths ("Coupon Collector Problem", see for instance [14]).

In the next section we describe a method for drawing u.a.r. paths of length $\leq n$ given some finite set $\mathcal{F}$ of infeasible prefixes where:

- no path with a prefix in $\mathcal{F}$ can be drawn.
- $\mathcal{F}$ can be extended with a new prefix before the next drawing.

3 A new approach: taking into account detected infeasibilities when drawing

The underlying idea of the new drawing algorithm is that instead of using the number of suffixes starting from the last vertex of the prefix currently built, in some cases it uses the number of suffixes of the prefix itself, hence generalizing counting to prefixes. This makes it possible to give a probability zero to draw paths with detected infeasible prefixes.

Suppose that a path of length n is drawn but detected as containing an infeasible prefix $p.s.s'$ [5](with no shorter infeasible prefix). Let l the length of $p.s$ and $K = f(s', n-l-1)$. All K paths with prefix $p.s.s'$ must be excluded from the future drawings. A naive attempt of solution would be to decrement K from the values stored in $f(x, n-|r.x|)$, for all subprefixes $r.x$ of $p.s.s'$, up to the modification of $f(s_0, n)$, as if the corresponding paths never existed. In particular $f(s', n-l-1)$ becomes 0, preventing s' to be chosen to extend $p.s$ in Algo 1. Alas this attempt is incorrect since an entry $f(x, m)$ is shared by all prefixes from s_0 to x of length $n-m$. Counting must now be prefix dependent.

Given $\mathcal{F}$ the current set of infeasible prefixes, we apply this idea by complementing f with a table $f_{\mathcal{F}}$ similar to f but with prefixes instead of vertices as first component. A reasonably efficient implementation is obtained by maintaining a trie data structure along the detections of infeasibilities. More precisely:

[4] If we were to use very large numbers, as `Rukia` can do, we should consider the binary complexity of operations, yielding a $O(n^2 \times q)$ complexity for memory and construction of the table and $O(n^2)$ for the drawing. For program testing we stay beyond such limits as feasibility of very long paths is unlikely [31, 26].

[5] Prefixes are never empty, always starting with s_0. The notation $p.s.s'$, where p can be empty, emphasizes the arc $s.s'$ that can lead to an infeasibility.

- entries in $f_{\mathcal{F}}$ are built lazily, only for prefixes of infeasible prefixes; for other prefixes, the standard f table is accessed with the last vertex of the prefix;
- $f_{\mathcal{F}}$ is implemented with maximal sharing, using a trie ("prefix tree") $\mathcal{C}_{\mathcal{F}}$ where keys are the elements of $\mathcal{F}$ and all their prefixes. The value $f_{\mathcal{F}}(p, l)$ of such a prefix p is the current number of paths from p to s_f of length l. This value takes into account all extensions of p that can have been possibly removed because of the infeasible prefixes already detected.

When an infeasible prefix is notified, a decrease of the current number of paths propagates in the trie from the new infeasible prefix up to s_0 (viewed as a prefix with a single vertex). Hence s_0 will be included in $\mathcal{C}_{\mathcal{F}}$. Elements of $\mathcal{F}$ appears at the leaves of the trie with associated value 0. The propagation can put to 0 the value of existing nodes of the trie, as if the corresponding prefixes were infeasible. This prevents the drawing algorithm from entering "dead ends" whose extensions are all infeasible at some later point.

Initialisation of $\mathcal{C}_{\mathcal{F}}$ Initially $\mathcal{F} = \emptyset$ but for its representation, $\mathcal{C}_{\mathcal{F}}$, we use a trie with the unique key s_0 mapped to value $f(s_0, n)$ so that we can start the drawing with some prefix in $\mathcal{C}_{\mathcal{F}}$.

Drawing in presence of infeasible paths Using f, $\mathcal{C}_{\mathcal{F}}$ and $f_{\mathcal{F}}$, the new drawing algorithm becomes

Algorithm 2 : drawing uniformly at random a path p of length n with $f_{\mathcal{F}}$

let $count(p.x, l) =$ if $p.x \in \mathcal{C}_{\mathcal{F}}$ then $f_{\mathcal{F}}(p.x, l)$ else $f(x, l)$;
$s = s_0;\ p = \varepsilon;\ l = n;$
if $count(s_0, n) == 0$ then fail("no more path to draw");
while $(l > 0)$ {
  draw s' among the successors s_k of s with proba. $count(p.s.s_k, l-1))/count(p.s, l)$
  $s = s';\ p = p.s';\ l = l - 1;$
}

With Algo. 2, as long as the current prefix is drawable and stays within $\mathcal{C}_{\mathcal{F}}$ it is feasible, thanks to the fact that we are notified with shortest infeasible prefixes. In addition to returning a path, `Rukia` now provides the length of its longer prefix included in $\mathcal{C}_{\mathcal{F}}$: feasibility check of any shorter prefix of that path can be skipped [6]. `Rukia` also provides $f_{\mathcal{F}}(s_0, n)$, the current number of paths still drawable, *i.e.* free of any known infeasible prefix.

[6] Checking for feasibility is performed by the system that calls `Rukia`; that system might record itself the longest feasible prefix detected in each drawn path but that information is now provided by `Rukia` at no cost.

Adding an infeasible prefix: Handling a new infeasible prefix mainly amounts to grafting a new branch in the trie using standard textbook algorithms, and propagating the decrease of the value K along that branch up to the root.

Let $\mathcal{F}$, $\mathcal{C}_\mathcal{F}$, $f_\mathcal{F}$ and f as above, and let r a prefix of a path drawn with $\mathcal{C}_\mathcal{F}$ that is detected as infeasible, $\mathcal{F}' = \mathcal{F} \cup \{r\}$ and $\mathcal{C}_{\mathcal{F}'}$ its trie.

We observe that Algo. 2 guarantees that neither r nor one of its subprefixes is in $\mathcal{F}$, otherwise r would not have been drawn at all.

Algorithm 3 : Updating $\mathcal{C}_\mathcal{F}$ to $\mathcal{C}_{\mathcal{F}'}$

let $K = count(r, n - |r|)$;
Graft a branch in $\mathcal{C}_\mathcal{F}$ using the subprefixes of r for keys and labelling nodes $p.x$ with value $f(x, n - |p.x|)$ for prefixes not already in $\mathcal{C}_\mathcal{F}$;
For r and all its subprefixes, substract K from their value in $\mathcal{C}_\mathcal{F}$.

We first add to the trie all the subprefixes of r not already existing, before updating the values of all nodes of the branch for r up to the root. By construction r have associated value 0 in $\mathcal{C}_{\mathcal{F}'}$ and is no longer drawable.

Pruning the trie: The decrease of K upwards in $\mathcal{C}_\mathcal{F}$ can set 0 for value of a prefix not in $\mathcal{F}$ but whose extensions are all infeasible at some later point. Future drawing of such prefix will be now impossible with Algo. 2. Any subtree in $\mathcal{C}_\mathcal{F}$ with a 0 value can be pruned (leaving only the uppermost such vertex), thereby decreasing the size of the trie. When removing feasible paths in addition to infeasible ones, this gives a halting condition for an enumeration of all feasible paths: no vertex left in the trie!

Complexity: With respect to Algo. 1, the additional complexity in memory space and time comes from the management of the trie. We consider, as in our current implementation, the graph to have degree 2, by (unelegantly) representing `switch` statement by cascades of `if`. Otherwise, with a basic implementation, we would have an additional δ factor, where δ is the degree of the graph.

The evolution of the trie, and thus the efficiency of the method is highly dependable on the shape of the graph and on the semantics of the program.

The worst memory requirement for the trie occurs when after each drawing the prefix elimination adds some vertices (at most n) to the trie and no pruning occurs. This happens when enumerating paths in a graph with only feasible paths: each drawing forbids exactly one path that is added (up to some sharing) to the trie. For instance, consider drawing in a full binary tree of height n, with the sequence of drawings reaching one leaf over two in a left-to-right traversal of the tree. The largest trie is obtained after drawing half of the paths, before pruning starts, and it contains all internal nodes and half the leaves of the full binary tree of height n. Hence worst-case complexity is $O(\min(m \times n, \frac{3}{4}2^n - 1))$.

In practice, pruning usually prevents to reach this worst-case, and even maintains low the size of the trie when the killing factors of the prefixes in $\mathcal{G}$ are high. Using "Patricia trees" instead of tries would further reduce memory usage. For program testing, other optimizations could apply, like transmitting to `Rukia` a compressed version of the CFG by collapsing all sequences of arcs with a unique successor. This corresponds to considering, for the drawing, only sequences of branching statements, the only sources of infeasibilities. From a path drawn from `Rukia`, the original path in the CFG can be later recovered. None of these optimizations are currently deemed necessary.

The time complexity of the preprocessing, namely the construction of the table $f(s, l)$ and the initialisation of the trie is unchanged. At each step of the drawing, we have to check if the current prefix has an arc labeled with the newly drawn vertex (pseudo function *count*) before accessing either f or $f_{\mathcal{F}}$. We also have to consider the time complexity for the update of the trie after a notification of an infeasible prefix. Time complexity is $O(n)$.

4 Experiments

To assess our method we have checked its scalability and its efficiency on different examples and different graph representations of the same piece of code.

Scalability of the new drawing method mainly depends on the memory size of the trie structure. Actually, neither the additional lookup for the drawing nor the update of the trie impact the time performance: in [11] the limiting factor was the size of the counting table, which is related to the size of the graph and the length of the paths. Our examples of programs are much smaller than the large models that can be processed by `Rukia`. In all our experiments, time spent in drawing is very negligible with respect to time spent in checking feasibility with SMT solvers. Therefore scalability of the drawing is checked against the maximum size (number of vertices) of the trie. As explained before, adding a new infeasible prefix can in fact shrunk the trie.

Efficiency is to be compared with the drawing with rejection and replacement method of Section 2.3 and may be expressed either as the ratio of feasible paths when drawing a given number of paths, or as the number of drawings needed for reaching a given number of feasible paths. We chose the latter formulation, closer to what is used in program testing where the number of test cases is chosen by the tester. In most experiments, we fixed it in an ad-hoc way in absence of a priori knowledge of the number of feasible paths. In some experiments we triggered the enumeration of all feasible paths. This was hardly doable for programs with a high ratio of infeasible paths when using the drawing with rejection and replacement method. For program testing this is not common practice but the aim was to stress our new method.

We assume the SMT solver to be able to decide the satisfiability of any path. In the experiments we have used the `Z3` system, currently restricted to the logic supported by the prototype described in [2] that generates the Control Flow

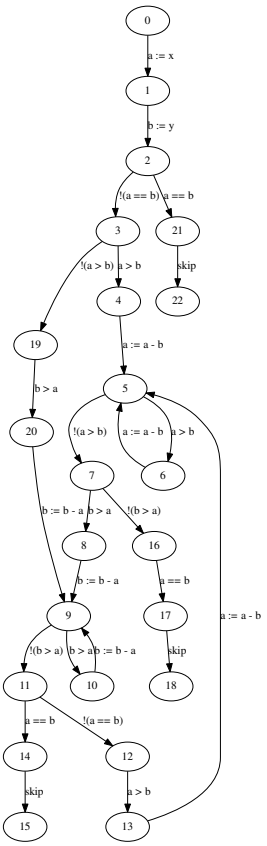

Fig. 3: A graph for the `gcd` that is free of infeasible paths

Graphs in the examples, and also some more elaborated variants of them as described below.

4.1 CFG or finer representations?

A high ratio of infeasible paths in a CFG hinders the application of structural testing, abstract interpretation, or most static analysis methods, by impairing the accuracy of their computations with information associated with paths that do not correspond to any execution. In the line of [19–21], one can use symbolic execution to statically unfold the CFG to obtain a larger graph from which part of the infeasible paths are removed. We have used such a method to carry out experiments with a variety of ratio of infeasibilities. Below, we briefly recall the principles of this method, which was presented in [2].

When the CFG contains no loop, a large-enough unfolding yields a symbolic execution tree with no infeasible paths (see Section 4.4). In the presence of loops, symbolic execution usually results in an infinite tree, so unfolding has to be bounded. Abstraction and subsumption are used to recover some loop structures during the unfolding: these loops are specialized versions (according to a feasible prefix) of the loops in the CFG free of their infeasible paths. For the CFG in Fig. 1, one gets a graph with 23 vertices that is free of infeasible paths: see Fig. 3. When no such "optimal" graph is obtained, the expansion of the CFG is glued to the original CFG at the vertices where the bound of expansion was reached, so that no feasible path is lost and one can still draw paths that are longer than the bound of expansion [3]. Some of the drawn paths stay in the expanded part and are free of infeasible prefixes, other start in the expanded part but end in the original CFG with less guarantee of feasibility. The bound of unfolding is left up to the tester, a trade off between precision and efficiency, but does not limit the length of paths for the drawing.

To summarize: original CFG have minimal size but possibly many infeasible paths; unfolded graphs are much larger, with less sharing but less infeasible paths. It is worth mentioning that the number of feasible paths is the same in all cases. Both kinds of graphs are useful. One may perform drawing in the original CFG for quick selection of many paths, or in its unfolded versions for a more accurate representation of the set of execution paths. It is interesting to see how our method fits these various representations of a program. In this paper the unfolding also provides a simple way to get large but realistic graphs.

4.2 Description of the experiments

For each example, drawing is performed on: the CFG, the optimal representation when it exists, and in partially unfolded graphs. In such a case, it means drawing paths longer than the bound of the unfolding. Each experiment is performed five times and we give the minimum and maximal value of each parameter when it is not constant. We give values for the following parameters:

- l: the length of the paths to draw
- $|G|$: the number of vertices in the graph
- paths: the initial number of paths in G
- drawn: the number of drawings needed for reaching the objective, either a given number of feasible paths or the enumeration of all feasible paths
- K (killing score): the maximum number of paths removed by a single infeasible prefix
- size: the maximum number of vertices in the trie
- saved: the ratio of calls to the SMT solver saved by knowing the longest prefix within $\mathcal{C}_{\mathcal{F}}$ of each path drawn.

The value of parameter K may vary in each of the 5 repetitions of an experiment. In practice we get the same value as long as we draw enough paths: K is based on shortest infeasible prefixes and in experiments we can draw different

set of paths but get the same K if the corresponding shortest prefix appears at least once in each experiment.

For the experiments with our former drawing, with rejection and replacement, infeasible paths are simply discarded from the selected paths before drawing again from the same graph. When drawing without redundancy for enumeration of the feasible paths, duplicates of feasible paths are also discarded. We give the number of drawings for reaching the objective. Some of these experiments yield a crash of the system by memory exhaustion in this version of `Auguste`, due to some internal bookkeeping of infeasible prefixes and feasible paths that is no longer needed in the new method; in such cases this gives a lower bound of the number of drawings performed without having reached the objective.

4.3 Experiments with `gcd`

In this simple example we give results for drawing in the initial CFG of Fig. 1 and its optimal unfolding of Fig. 3. For this last graph `Rukia` provides the number of paths of length at most 30, i.e. the number of feasible paths of that length in both graphs. We mention that our implementation adds an extra vertex: $|G|$ is one more than the number of vertices in the figures. In this experiment we ask for the full collection of feasible paths without duplicates.

	l	$\|G\|$	paths	drawn	K	size	saved
initial CFG	30	10	15478	1152	4672	1796 - 1880	80.1- 80.4 %
optimal CFG	30	24	792	792	1	1358 - 1444	76.0 - 76.2 %

Table 1: Results for the `gcd` example

As expected, in the optimal CFG we need exactly 792 drawings and each drawing removes exactly one path for the future drawings. In the initial CFG, the kind of infeasibility in this example leads to a high killing factor and a rather low number of drawings (with respect to the total number of paths) before obtaining the 792 feasible paths. We also have a high decrease in the number of calls needed to the SMT solver for checking feasibility of drawn paths. With our old drawing method, it requires between 4990 and 8000 drawings in the optimal CFG (because of the duplicates) and more than 100 000 drawings in the initial CFG to get the full collection.

4.4 Experiments with `tcas`

This example is a derived version of the program `tcas` from the Siemens benchmark for testers [12]. The original program consists of several auxiliary functions and a `main` function that reads input parameters before calling the function of interest. Documentation mentions the presence of an infeasible path. None of the functions contains a loop or a recursive call and the code of most of them reduces to a unique expression.

For this experiment, we gathered the code in a unique function that takes the input values as parameters, processing auxiliary functions as macros to obtain some textual in-lining. The resulting piece of code has no loop but it contains many lazy boolean operators, resulting in a complex flow of control: in the CFG the flow of control associated with the lazy operators is made explicit by introducing vertices for each atomic expression and adding the needed edges. This increases the size of the CFG with respect to the size of the source code. In this case the resulting CFG has 88 vertices.

This CFG can be unfolded in a symbolic evaluation tree with only feasible paths: exactly 123 paths of length at most 47. We test our method again this corner case, by asking for all paths of length 47, without duplicates. We also compare the drawing effort when using either an incomplete unfolding or an approximate length. Table 2 shows the results for the following experiments:

- tcas-opt: Draw all 123 feasible paths of length 47 in the Symbolic Execution Tree
- tcas-CFG: Draw all 123 feasible paths of length 47 in the initial CFG
- tcas-40: Draw all 123 feasible paths of length 50 after unfolding the CFG up to depth 40 instead of the optimal 47. The resulting graph is quite large and contains infeasible paths
- tcas-40 (60) and CFG-40 (60): Draw 60 feasible paths (allowing duplicates) of length 50 in tcas-40 and in the CFG. These requests are closer to what would be actual test requests.

	l	$\|G\|$	paths	drawn	K	size	saved
tcas-opt	47	1677	123	123	1	282 - 322	58.6 - 60.0 %
tcas-CFG	47	88	179720	752 - 758	7562	1729 - 1812	87.4 - 87.9 %
tcas-40	50	1232	386	298	4	1017 - 1174	79.0 - 80.1 %
tcas-40 (60)	50	1232	386	161 - 171	4	1051 - 1149	68.1 - 71.7 %
CFG-40 (60)	50	88	181512	579 - 601	7562	1697 - 1787	85.7 - 85.9 %

Table 2: Results for the `tcas` example

For tcas-opt: as we have only feasible paths, each drawing reaches a target but removes only one path ($K = 1$). In tcas-CFG, the killing score K is much higher leading to a reasonable number of drawings for collecting the 123 feasible paths. The unfolded graph used in row 3 of Table 2 is much larger than the CFG, with the same number of feasible paths. Here 175 infeasible paths were drawn in addition to the 123 targets. The last two rows give the number of drawings for collecting only 60 of the 123 targets. We observe that the number of drawings is not too high, even when one draws in the raw CFG with 123 targets disseminated among 181512 paths. `Rukia` also provides the number of remaining paths in the graph (if they were all feasible): this gives an hint on how far we are from the full collection of feasible paths. For row 4 there are around 140 such paths and

for row 5 it varies between 185 and 242. In this last case, much of the cleaning in the set of paths is already performed.

In practice, for this small example, it takes less time to draw in the initial CFG, with its many infeasible paths, than to first build the optimal representation before drawing paths in it. In addition to this positive result, we also observe the important ratio of calls to the SMT solver that can be saved, a very important gain in performance for the overall system.

Below are the corresponding drawing numbers with our previous method: without infeasible prefixes and duplicates elimination. All other parameters from Table 2 are either the same or do not apply to the old method.

- tcas-opt: between 491 and 906 drawings, depending on the experiment, to find all 123 paths
- tcas-CFG: crash after 130 400 drawings with only 62 distinct feasible paths found
- tcas-40 (60): 194 drawings for finding 60 feasible paths among the 386;
- CFG-40 (60): 83 301 drawings for finding 60 feasible paths among all paths.

One clearly notes the benefit of our method when the ratio of feasible paths is low, even when the objective is only 60 feasible paths.

4.5 Experiments with `bsearch`

This experiment is inspired by an example from the `Pathcrawler` tutorial [29] (a tool for concolic testing [5]): it is a dichotomic search of an element in a sorted array. As `Auguste` does not currently handle formulae for stating that an arrays is sorted, our program performs a kind of dichotomic walk in the array between two bounds given as input parameters. Depending on the value of the current element, it explores the left sub-array or the right sub-array. Drawing is performed in the CFG and in an unfolded version. Here we chose an unfolding bound of 30 and a maximal path length of 50. The resulting graph becomes about 50 times larger but with only one third of the initial set of paths. We first ask for 300 different feasible paths, then for 3000 such paths, not knowing the number of feasible paths.

	l	$\|G\|$	paths	drawn	K	size	SMT saved
CFG-300	50	17	21247	831 - 883	4607	6495 - 6632	73.5 - 74.6 %
b30-300	50	855	8148	738 - 817	31	6061 - 6523	71.8 - 73.2 %
CFG-3000	50	17	21247	4883	4607	10841 - 11022	85.7- 85.9 %
b30-3000	50	855	8148	4787	31	10732 - 10950	85.6 - 85.8 %

Table 3: Results for the `bsearch` example

In row 3 of Table 3 when asked for 3000 different feasible paths, the system stops after having collected 2594 feasible paths and the additional information

(from the trie) that there are no more paths to draw. We now know that there exist only 2594 different feasible paths.

Below are the corresponding numbers with the old drawing method:

- CFG-300: 2726 drawings;
- b30-300: 944 drawings, with 632 infeasible and 12 feasible paths drawn more than once;
- CFG-3000: Crash after 130 300 drawings with 2590 feasible paths found
- b30-3000: 83 369 drawings with 56 854 infeasible, other ones were duplicates. We ask for 2594, not 3000, feasible paths otherwise the system would not stop, searching forever additional feasible paths that do not exist.

5 Conclusions, related works and perspectives

In this paper, we present a significant improvement of the structural random testing method presented in [10] and [11]. The aim of this method is to ensure a probabilistic uniform coverage of the paths of the program (or model) under test. It has been successfully implemented via some specialisation of the classical recursive method for random generation of combinatorial structures. However, as for any method based on control flow graphs of programs, a serious problems comes from the existence of infeasible paths: such paths are detected after drawing, using SMT solvers, and are useless for defining test data.

We have developed an algorithm that enriches the data structures used by the classical recursive method in such a way that detected infeasibilities are collected and ignored in the following drawings. This opens the way to practical applications of the method, avoiding lot of useless drawings, and in some cases masses of them. In order to confirm the practical interest of our technique by realistic experiments, we are currently working at the extension of the considered programming language to a more significant subset of C and at the integration of our system into the `Frama-C` platform.

As far as we know, in the area of uniform generation of combinatorial structures, the closest piece of work is [22] where non redundant drawing is performed in languages described by weighted context-free grammars using a tree of prefixes similar to the trie structure we use.

Combinatorial methods are more and more used in the area of random testing of programs or random exploration of models. One can cite: [4] on the uniform sampling of timed automata and [6] for networks of automata; or [7] where the authors provide a way of approximating vertex coverage via some sampling.

None of these works address the issue of infeasibility. An interesting perspective is to study how our approach to uniform sampling with bounded length is transposable to other notions considered in [6], namely Boltzmann sampling (see Appendix A) and Parry sampling, which are relevant in Monte Carlo model checking [17, 25]: it would open interesting perspectives for program model checking [18]. Besides, a very recent paper [8] addresses the issue of drawing uniformly

behaviours from parallel compositions where synchronisations introduce forbidden traces in the product automaton. The problem presents some similarities with infeasibilities, but the proposed technique is very different from ours: another interesting perspective is to compare their domains of application.

Acknowlegdment We are indebted to Alain Denise, Danièle Gardy and Yann Ponty for interesting discussions.

References

1. Aissat, R.: Détection de Chemins Infaisables : un Modèle Formel et un Algorithme. Phd thesis, Université Paris-Saclay (2017)
2. Aissat, R., Gaudel, M.C., Voisin, F., Wolff, B.: A Method for Pruning Infeasible Paths via Graph Transformations and Symbolic Execution. In: 2016 IEEE Int. Conf. on Software Quality, Reliability and Security, QRS (2016), https://hal.archives-ouvertes.fr/hal-01655414
3. Aissat, R., Voisin, F., Wolff, B.: Infeasible Paths Elimination by Symbolic Execution Techniques: Proof of Correctness and Preservation of Paths. Archive of Formal Proofs (Aug 2016), https://hal.archives-ouvertes.fr/hal-01764577
4. Barbot, B., Basset, N., Beunardeau, M., Kwiatkowska, M.: Uniform sampling for timed automata with application to language inclusion measurement. In: Quantitative Evaluation of Systems - 13th International Conference, QEST 2016. Lecture Notes in Computer Science, vol. 9826, pp. 175–190. Springer (2016)
5. Bardin, S., Kosmatov, N., Marre, B., Mentré, D., Williams, N.: Test case generation with PathCrawler/LTest: How to automate an industrial testing process. In: Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV. pp. 104–120 (2018)
6. Basset, N., Mairesse, J., Soria, M.: Uniform sampling for networks of automata. In: 28th International Conference on Concurrency Theory, CONCUR 2017. LIPIcs, vol. 85, pp. 36:1–36:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
7. Bernard, J., Héam, P., Kouchnarenko, O.: An approximation-based approach for the random exploration of large models. In: Tests and Proofs - 12th International Conference, TAP 2018. Lecture Notes in Computer Science, vol. 10889, pp. 27–43. Springer (2018)
8. Bodini, O., Dien, M., Genitrini, A., Peschanski, F.: The combinatorics of barrier synchronization. In: Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11522, pp. 386–405. Springer (2019)
9. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. pp. 443–446. ASE '08, IEEE Computer Society (2008)
10. Denise, A., Gaudel, M.C., Gouraud, S.D.: A generic method for statistical testing. In: IEEE Int. Symp. on Software Reliability Engineering (ISSRE). pp. 25–34 (2004)
11. Denise, A., Gaudel, M.C., Gouraud, S.D., Lassaigne, R., Oudinet, J., Peyronnet, S.: Coverage-biased random exploration of large models and application to testing. STTT, International Journal on Software Tools for Technology Transfer **14**(1), 73–93 (2012)

12. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Softw. Eng. **10**(4), 405–435 (2005). https://doi.org/10.1007/s10664-005-3861-2
13. Duchon, P., Flajolet, P., Louchard, G., Schaeffer, G.: Random Sampling from Boltzmann principles. In: 29th International Colloquium on Automata, Languages and Programming - ICALP'2002. Lecture Notes in Computer Science, vol. 2380, pp. 501–513. Springer (2002)
14. Flajolet, P., Gardy, D., Thimonier, L.: Birthday paradox, coupon collectors, caching algorithms and self-organizing search. Discrete Applied Mathematics **39**(3), 207–229 (1992)
15. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. PLDI '05, ACM (2005)
16. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012)
17. Grosu, R., Smolka, S.A.: Monte Carlo model checking. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 3440, pp. 271–286. Springer Berlin Heidelberg (2005)
18. Havelund, K., Visser, W.: Program model checking as a new trend. STTT **4**(1), 8–20 (2002)
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. 29th ACM Symp. on Principles of Programming Languages. pp. 58–70. ACM (2002)
20. Jaffar, J., Murali, V.: A path-sensitively sliced control flow graph. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 133–143. ACM (2014)
21. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: TRACER: A symbolic execution tool for verification. In: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. pp. 758–766 (2012)
22. Lorenz, W.A., Ponty, Y.: Non-redundant random generation algorithms for weighted context-free grammars. Theor. Comput. Sci. **502**, 177–194 (2013)
23. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (Dec 1990)
24. Oudinet, J.: Approches combinatoires pour le test statistique à grande échelle. Ph.D. thesis, Université Paris-Sud XI (2010)
25. Oudinet, J., Denise, A., Gaudel, M.C., Lassaigne, R., Peyronnet, S.: Uniform Monte-Carlo model checking. In: Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, vol. 6603, pp. 127–140. Springer Berlin Heidelberg (2011)
26. Papadakis, M., Malevris, N.: A symbolic execution tool based on the elimination of infeasible paths. In: 5th Int. Conf. on Soft. Eng.Advances, ICSEA 2010. pp. 435–440. IEEE (2010)
27. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 263–272. ESEC/FSE-13, ACM (2005)
28. Thévenod-Fosse, P., Waeselynck, H.: An investigation of statistical software testing. Softw. Test., Verif. Reliab. **1**(2), 5–25 (1991)

29. Williams, N., Kosmatov, N.: Automated structural testing with pathCrawler. a tutorial, examples (2012), http://pathcrawler-online.com:8080/tutorial/tutorial2012examples.pdf
30. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In: Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3463, pp. 281–292. Springer (2005)
31. Yates, D.F., Malevris, N.: Reducing the effects of infeasible paths in branch testing. In: Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification, TAV 1989, Key West, Florida, USA, December 13-15, 1989. pp. 48–54. ACM (1989)

A Appendix: Transposition of the method to Boltzmann samplers

The method can be used to enrich other drawing methods than the classical recursive one considered in this paper, for instance Boltzmann samplers. Such samplers exist in two varieties, the ordinary version and the exponential version. In this appendix we explain how our method is applicable to ordinary Boltzmann sampling [13]. The same approach is doable for exponential Boltzmann sampling.

A.1 Ordinary Boltzmann samplers in a nutshell

Instead of using equation (1), which is based on the number of paths of a given length from a given vertex, Boltzmann samplers use recursive equations on generating functions of the sets of all paths from a given vertex. In contrast with the recursive method, which allows to uniformly draw objects of a fixed size (or in our context paths of bounded lengths), such samplers return objects of random sizes and ensure uniformity for the objects of the same size. It's possible to tune these samplers to favour objects of a size in the vicinity of a given value. When drawing paths, using such samplers would avoid to introduce a bound on their lengths.

Let consider the set of paths starting from a given vertex s as a language noted $\mathcal{L}_s$. The ordinary generating function of parameter z of this language is:

$$L(z)_s = \Sigma_{m \in \mathbb{N}} l_m z^m$$

where z is a complex variable and l_m the number of words of length m. Such a function is defined for values $0 < z < \rho_L$ where ρ_L is called the convergence radius of L.

Given a value of z such that $0 < z < \rho_L$, an ordinary Boltzmann sampler draws a path p with probability $z^{|p|}/L(z)$. The choice of the value of z determines the distribution on the lengths of the paths. For more details and complexity results, see [13] and [6].

A.2 Taking into account infeasibilities in a Control Flow Graph

The generating functions for the set of vertices of $\mathcal{G}$ satisfy the following equation:

$$L_s(z) = \big(z \sum_{s \to t \in \mathcal{G}} L_t(z)\big) + 1_{s=s_f} \tag{2}$$

Similarly to what was done on the base of equation (1) in section 2.3, equation (2) can be used to define recursively a drawing algorithm: if $s = s_f$, since s_f has no successors, the random generation stops, otherwise a successor t of s is drawn with probability $zL_t(z)/L_s(z)$, then the drawing algorithm is called recursively with t.

We observe that the enrichment of $\mathcal{G}$ with an edge from s_f to itself, introduced in subsection 2.3 for drawing paths of lengths less or equal to bound n, is no more necessary: as said above, Boltzmann samplers return paths of different lengths whose distribution, i.e average value and variance, is adjustable.

One can note that when the aim is to draw only paths of an exact length, the sampler can be tuned via an adequate choice of z to a distribution of the lengths with small variance, and combined with a rejection method to filter unwanted paths, at the cost on a slightly increased complexity.

As noted in [13], a Boltzmann sampler requires as input the value of the parameter z, and the finite collection of the values at z of the generating functions used in the specification. These values need only to be computed once. In our case, it is the vector $\mathbb{L}(z) = (L_s(z))_{s \in \mathcal{G}}$. It means that the memory requirement is significantly lower than for the recursive method $O(q)$ instead of $O(n \times q)$.

After detecting the infeasibility of some prefix $p.s.s'$, the probability of drawing s' after $p.s$ must become 0, and the probabilities of the vertices of $p.s$ must be decreased, taking into account the size of $\mathcal{L}_{s'}$, i.e. $\mathbb{L}(z)_{s'} = L_{s'}(z)$. This can be achieved by complementing vector $\mathbb{L}(z)$ by the same trie data structure as in Section 3. The contents of the vector and of the trie are different since the lengths of the paths, and therefore of the suffixes of the infeasible prefixes, are no more taken into account. But the principles of algorithms for drawing with the trie, updating it, and pruning it remain unchanged.

We cannot report experiments with such samplers due to the lack of easily available and well documented implementations for the choice of z and the computation of $\mathbb{L}(z)$.