RAPPORT DE RECHERCHE

L R I

RANDOM EXPLORATION OF MODELS

OUDINET J

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

06/2010

Rapport de Recherche N° 1534

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

Noname manuscript No.
(will be inserted by the editor)

Random exploration of models

Johan Oudinet

Received: date / Accepted: date

Abstract This article presents optimizations of a randomized method that generates paths while ensuring a good coverage of the model, regardless its topology. The optimizations aim at diminishing the required memory, thus allowing the generation of longer paths. Pure random exploration generally leads to a bad coverage of the model. Methods, based on counting and uniform drawing in combinatorial structures, can ensure a good coverage of paths. Due to memory consumption, such methods can neither explore very large models nor generate very long paths. In this paper, we leverage the limitation of path lengths by using new algorithms with better space complexity. Experimental results show significant improvement over previous randomized approaches. This work opens new perspectives to efficiently explore models for simulation, random testing and model-checking purposes.

Keywords uniform random generation · model exploration · floating point arithmetic

1 Introduction

This article presents probabilistic methods to explore finite non-probabilistic models. Combinatorial explosion is a well-known problem ; systems are becoming increasingly complex and models grow exponentially. These occur, for example, when the system is represented by the combination of several models. In such systems, an exhaustive exploration is generally impracticable. Random exploration of large models is a way of fighting the combinatorial explosion problem.

Pure random exploration generally leads to a bad coverage of the model. In the literature (Clarke et al 1989), there are several coverage criteria (states, transitions, MC/DC, paths) based on the properties sought to verify. Here, we focus on the uniform coverage of paths.

The random exploration of a model is a classical approach in simulation, but is also used for testing (Dwyer et al 2007) and model-checking (Grosu and Smolka 2005;

J. Oudinet
Univ Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405;
CNRS, Orsay, F-91405.
E-mail: johan.oudinet@lri.fr

Hérault et al 2004). To randomly explore a model represented by a graph, the most natural approach is to use random walks. To perform a random walk on a graph $\mathcal{G}$, it suffices to know all the system states and their successors, and to be able to assign a probability to each of these successors such that the sum of probabilities of all successors of a vertex is equal to 1. In the case of an *isotropic* random walk, all successors are equally likely. Algorithm 1 generates a path of length less than or equal to n.

Algorithm 1 Isotropic random walk in a graph

Require: A graph $\mathcal{G}$, an initial vertex s_0 and a length n
Ensure: A path σ such that $|\sigma| \leq n$
  $i \leftarrow 0$
  $s \leftarrow s_0$
  while $i \neq n$ and $succ(s) \neq \emptyset$ **do**
    Choose a vertex s' uniformly among the successors of s ($succ(s)$)
    $\sigma \leftarrow \sigma \cup t$ with t, the transition $s \rightarrow s'$
    $i \leftarrow i + 1$
    $s \leftarrow s'$
  end while
  return σ

Needing no other knowledge than the current state and the list of its successors, an isotropic random walk uses little memory; hence, it seems to be the ideal candidate to explore very large models. Unfortunately, the induced probability distribution is difficult to determine because it depends on the topology graph. Occasionally an isotropic random walk is completely ineffective, as the following example.

Consider the graph in Figure 1, the expected number N of isotropic random walks to perform before obtaining n distinct paths (of length n) is:

$$\begin{aligned} E(N) &= E(N_1) + E(N_2) + \cdots + E(N_n) \\ &= \frac{1}{p_1} + \frac{1}{p_2} + \cdots + \frac{1}{p_n} \\ &= 1 + 2 + \cdots + 2^{n-1} \\ &= 2^n - 1 \end{aligned} \tag{1}$$

where $E(N_i)$ (resp. p_i) denotes the expectation (resp. probability) to obtain a new path after performing $i-1$ separate isotropic random walks. In other words, the equation (1) shows that using isotropic random walks, it takes, on average, an exponential amount of time to cover the paths of the graph in Figure 1.

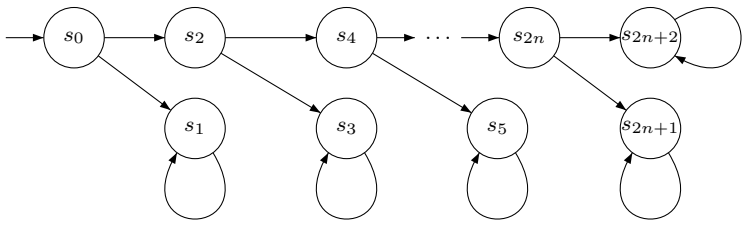

Fig. 1 An example of pathological graph for isotropic random walks

There is a simple explanation why an isotropic random walk is inefficient to cover such kind of graphs. At each state, the random walk must make a choice between either a state that leads to one path, or a state from which there are an exponential number of paths. But in the case of an isotropic step, these two states have the same probability of being fired, thus supporting the likelihood of a path at the expense of a very large number of paths. If we were able to determine the number of paths that start from each state, then we could guide the random walk in order to balance the likelihood of all paths and obtain at best a uniform distribution of paths.

The paper is organized as follows. Section 2 explains a method used by Denise *et al.* (Denise et al 2004; Gouraud et al 2001) to draw uniformly at random paths of length less than or equal to n, which is based on a recursive method to count paths (Flajolet et al 1994). Due to its space complexity, this method can neither explore very large models nor draw very long paths. Gaudel et al (2008) have developed modular techniques, which rely on uniform generation of paths in each component, to leverage the limitation of model size. In sections 3 and 4 two optimizations are presented to leverage the limitation of path lengths by reducing the space complexity. The results of initial experiments with these new methods are presented in Section 5.

2 Uniform generation of paths

It is necessary to be able to count the paths that start from a state to obtain a uniform generation on these paths. In this section I summarize the work of Denise *et al.* on the uniform generation of paths of length less than or equal to n. But before that, we need a minimum of formalism.

Models can be represented in different ways depending on the semantics that we want them and the type of analysis that is desired. For the rest of this article, we use the notion of an automaton.

Definition 1 A finite automaton $\mathcal{A}$ is a structure :

$$\mathcal{A} = \langle \mathcal{X}, \mathcal{S}, s_0, \mathcal{F}, \mathcal{T} \rangle$$

where $\mathcal{X}$ is an alphabet of labels, $\mathcal{S}$ a finite set of states, s_0 the initial state, $\mathcal{F} \subseteq \mathcal{S}$ a set of final states, and $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{X} \times \mathcal{S}$ a set of transitions.

Definition 2 A path σ of length n in an automaton $\mathcal{A}$ is a sequence of transitions:

$$\sigma = t_1 t_2 \cdots t_n$$

such that: $\forall i, t_i = s_{i-1} \times a_i \times s_i$ and $s_n \in F$.

Figure 2 shows an automaton with a unique final state (s_5). The path $\sigma = s_0, a, s_1, b, s_2, c, s_5$ is one of the two paths of length 3 in this automaton. There are 6 paths of length ≤ 9.

Consider the automaton $\mathcal{A}$ and an integer n, $\mathcal{P}_n(\mathcal{A})$ (resp. $\mathcal{P}_{\leq n}(\mathcal{A})$) denotes the set of paths of length n (resp. less than of equal to n) in $\mathcal{A}$ from s_0 to any state of $\mathcal{F}$. $\mathcal{P}_n$ (resp. $\mathcal{P}_{\leq n}$) is used when there is no ambiguity about $\mathcal{A}$.

The aim is, given an integer n, to generate uniformly at random one or several paths of length $\leq n$ from s_0 to any state of F. Uniformly means that all paths in $\mathcal{P}_{\leq n}$ have the same probability to be generated. At first, let us focus on a slightly different problem: the generation of paths of length n exactly. We will see further that a slight

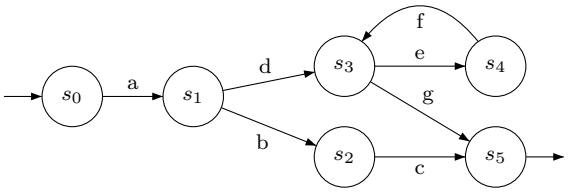

Fig. 2 An example of an automaton

change in the automaton allows to generate paths of length $\leq n$. Remark that generally the number of paths of length n grows exponentially with n.

Suppose that the current step, the random walk is on the state s, which has k successors: $s_1, s_2, \ldots, s_k$, and it remains m steps before obtaining a path of length n. To ensure uniformity of paths of $\mathcal{P}_n$, the random walk must choose the successor s_i (for $1 \leq i \leq k$) according to the following probability:

$$P(s_i) = \frac{l_{s_i}(m-1)}{l_s(m)} \tag{2}$$

where $l_s(m)$ denotes the number of paths of length m that connect s to any state of $\mathcal{F}$.

Computing the numbers $l_s(i)$ for any $0 \leq i \leq n$ and any state s of the graph can be done by using the following recurrence rules:

$$\begin{cases} l_s(0) = 1 & \text{if } s \in \mathcal{F} \\ l_s(0) = 0 & \text{if } s \notin \mathcal{F} \\ l_s(i) = \sum\limits_{s \to s'} l_{s'}(i-1) & \forall i > 0 \end{cases} \tag{3}$$

Thus, after a preprocessing stage of storing the values $l_s(i)$ for all $s \in \mathcal{S}$ and for $0 \leq i \leq n$ in a two dimensional array - called the counting table - paths of length $\leq n$ can be generated. Note that the preprocessing stage must be done only once, whatever the number of paths to be generated.

Theorem 1 *The bit complexity of the preprocessing stage is:*

- *time: in* $\mathcal{O}(n^2 \times d \times S)$,
- *space: in* $\mathcal{O}(n^2 \times S)$.

The bit complexity of the generation stage is:

- *time: in* $\mathcal{O}(n^2 \times d)$,
- *space: in* $\mathcal{O}(n^2 \times S)$.

Proof Since this algorithm is not new, its complexity was already studied and similar results can be found in Flajolet et al (1994), except that we take into account the maximum number of transitions from a state.

Let us consider first the preprocessing stage. Computing the counting table requires $\mathcal{O}(n \times d \times S)$ additions, where n is the path length, S the number of states in the automaton and d stands for the maximum number of transitions from a state. Each operation is performed using integer arithmetic. However, the two operands that represent numbers of paths, can have a significant size (in $\mathcal{O}(n)$ bits) because the space

needed to represent a number is an order of magnitude compared to logarithmic value and the number of paths of length n can be exponential compared to n. Thus, the cost of each arithmetic operation is in the worst case in $\mathcal{O}(n)$ (Knuth 1997).
Easy computations show that the memory space requirement is $n \times |S|$ integer numbers and each of those numbers has a size in $\mathcal{O}(n)$ bits.

Regarding the generation stage, obtaining a path of length n requires to choose n times a state among at most d successors. Hence, $\mathcal{O}(n \times d)$ comparisons between numbers of size in $\mathcal{O}(n)$ bits.
The memory space requirement is the counting table, $\mathcal{O}(n^2 \times S)$, plus the size of a path of length n, $\mathcal{O}(n)$.

For generating paths of length $\leq n$ instead of exactly n, the only change is the following: Add to the automaton a new state s'_0 which becomes the new initial state, with a transition from s'_0 to s_0 and a loop transition from s'_0 to itself. Label both transitions with a same new letter. Each path of length $n+1$ from s'_0 to a state of F in this new automaton crosses k times the new loop transition for some k such that $0 \leq k \leq n$ and exactly once the one from s'_0 to s_0. With this path we obviously associate a path of length $n-k$ in the previous graph. It is straightforward to verify that any path of length $\leq n$ can be generated in such a way, and the generation is uniform.

The limitations of this method are: memory space for the counting table, which can be very important when one is interested to draw long paths in very large models, and to a lesser extent, time generation can become quadratic in n if the operations are done with integer arithmetic. Actually, there is a known trick to reduce the average complexity of choosing the next state. The idea is to draw a random number between 0 and 1 bit by bit and decide what transition fires as soon as possible. With such procedure, few bits are enough to decide in average and the average complexity of each comparison becomes in $\Theta(1)$ instead of $\mathcal{O}(n)$, but the worst complexity is unbounded. The next two sections describe, respectively, two variants of this method that can be combined to exceed these limits.

3 Uniform generation without storing the counting table

If we look in more detail the creation and use of the counting table in the generation method described in the previous section, we note the following:

- The counting table is filled with the recurrence relation (3), lines 0 to n.
- Then, for each generation of a path of length n, the table is traversed in the opposite direction, namely from line n to line 0. But every step of the random walk, only two lines (lines i and $i-1$) are useful.

For example, considering the counting table in Table 1, the random walk that generates the path of length 5 will need only lines 3 and 4 when it is on the state s_1 to choose with probability 1 state s_3.

Thus, if we were able to calculate, for any s, $l_s(i-1)$ from $l_s(i)$ it would no longer need to remember all the counting table, but only two lines that we would update every step of the random walk. The method becomes:

Preprocessing: Determine the last row (n) of the table with equation (3). Only the last line is stored in memory.

	sommets					
n	s_0	s_1	s_2	s_3	s_4	s_5
0	0	0	0	0	0	1
1	0	0	1	1	0	0
2	0	2	0	0	1	0
3	2	0	0	1	0	0
4	0	1	0	0	0	0
5	1	0	0	0	0	0

Table 1 Counting table associated with the graph of Figure 2, for $n = 5$

Path generation: The principle remains the same, except that each step of the random walk, we calculate the line above the counting table.

Theorem 2 *For any aperiodic automaton, there exists a positive integer n_0 from which every line can be computed from the next one.*

Proof To be able to proof this theorem and to give an algorithm, we need to translate this problem into an equivalent algebra problem. Let $A \in \mathbb{N}^{S\times S}$ be the transition matrix associated with $\mathcal{A}$ and the vector $F \in \mathbb{N}^S$ defined as follows:

$$F[i] = \begin{cases} 1 & \text{if } s_i \in \mathcal{F} \\ 0 & \text{else.} \end{cases} \tag{4}$$

Then, the vector L_n that represents $l_s(n)$ for all $s \in \mathcal{S}$ can be defined recursively:

$$\begin{cases} L_0 = F \\ L_n = AL_{n-1} \end{cases}$$

We are interested by a matrix B such that $L_{n-1} = BL_n$. The problem of inverting the system of recurrences defined by equation (3) is then equivalent to the problem of linear algebra as follows: find an integer n_0 and a matrix $B \in \mathbb{Q}^{S\times S}$ such that

$$\forall i \geq n_0, \quad B.A^{i+1} = A^i. \tag{5}$$

Adding n_0, the integer minimum at which the relationship is valid, is required because there may be a place where it is impossible to return the previous line.

The solution to equation (5) is composed of two steps:

1. Find the smallest i such that the rank of A^{i+1} is equal to the rank of A^i, the i is actually n_0[1].
2. As the rank of A^{n_0} is equal to the rank of A^{n_0+1}, let f be the linear mapping associated to A and f_{n_0} that associated to A^{n_0}, the restriction of f to the image of f_{n_0} is an isomorphism (ie., the corresponding matrix C is invertible). Just then calculate the matrix C, inverse it and return to the original space to obtain the matrix B sought.

[1] As the rank of A^{i+1} is always positive or zero and less than or equal to that of A^i, n_0 always exist and it is at most S. We could choose directly $n_0 = S$ but the size of the automaton is large, it is better to try to find a n_0 much smaller.

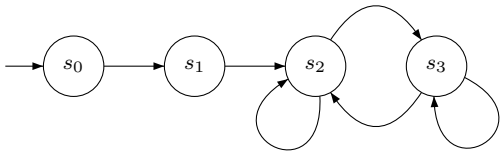

Fig. 3 A simple graph used as an example to reverse the recurrences

Example 1 Let see an example to clarify the process. Figure 3 shows a simple graph; its associated transition matrix is:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

The first step is to find n_0 such that the rank r of A^{n_0} is equal to A^{n_0+1}. In this example, we got $n_0 = 3$ and $r = 1$. Note that after achieving s_2, it is impossible to go back in one of the first two states. Hence, n_0 should be greater than 2.

As soon as n_0 is known, the matrix B can be computed and the result is:

$$B = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

If we have the number of paths of length 5 that connect any state to any state, it is equivalent to $L_5 = A^5 \times \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}^t = \begin{pmatrix} 8 & 16 & 32 & 32 \end{pmatrix}^t$. We can verify that $L_4 = B \times L_5 = \begin{pmatrix} 4 & 8 & 16 & 16 \end{pmatrix}^t$.

Now the generation scheme is as follows:

- Preprocessing stage 1: Compute n_0 and B from the automaton.
- Preprocessing stage 2: Compute and store the first n_0 rows of the counting table, then compute the next rows until the last row is reached. Store the last row, L_n.
- Generation stage: calculate the penultimate row from the last one and choose the successor of s_0 according to these two lines. Repeat this process until you need to draw the last n_0 transitions then use the counting table to generate the end of the path.

Theorem 3 *The bit complexity of the preprocessing stages is:*

- *time: in* $\mathcal{O}(n_0^2 \times d \times S^2 + r^3 \times n_0 + n^2 \times d \times S)$ *where* r *is the rank of* A^{n_0},
- *space: in* $\mathcal{O}(n_0 \times n \times S)$.

The bit complexity of the generation stage is:

- *time: in* $\mathcal{O}(n^2 \times d \times S)$,
- *space: in* $\mathcal{O}(n_0 \times n \times S)$.

Proof Let us consider first the complexity of preprocessing stage 1. Obtaining A^{n_0} and its rank requests successive multiplications of sparse matrices of integers. Each integer is bounded by $\mathcal{O}(d^{n_0})$; its size is in $\mathcal{O}(n_0)$. Thus, the first step is done in $\mathcal{O}(S^2 \times d \times n_0^2)$ bits. The most time consuming operation of the second step is the inversion of the matrix C, which is of size $r \times r$ (r being the rank of A^{n_0}). Indeed, the second step takes $\mathcal{O}(r^3 \times n_0)$ bits.

Then, the second preprocessing stage is equivalent to the preprocessing step of the original generation method (i.e., in $\mathcal{O}(n^2 \times d \times S)$) except that it requires less memory, $\mathcal{O}(n_0 \times n \times S)$ bits only.

Finally, the generation stage keeps n_0+1 lines of the counting table in memory and uses 2 extra lines for every decision step. As each line is composed of S big numbers, the generation stage requires $\mathcal{O}(n_0 \times n \times S)$ bits only. However, the cost of generating a path of length n has increased because the calculation of the previous line of the counting table using the recurrence relations defined by B requires $\mathcal{O}(S \times d)$ multiplications of a rational by a large integer. Considering the complexity of those multiplications is in $\mathcal{O}(n)$ each, the bit complexity of the generation step is in $\mathcal{O}(n^2 \times d \times S)$.

Thus, the inversion of a system of recurrences is polynomial in the size of the system, S. Nevertheless, as for the preprocessing when using the counting table, this inversion need be done only once whatever the number of paths generated then. The generation is less efficient, but we must take into account the gain in memory size is very important when one wants to generate very long paths.

4 Floating-point calculus

A possible alternative (Denise and Zimmermann 1999) is to avoid the prohibitive cost of each operation, replacing the integer arithmetic in a floating-point arithmetic. We recall that a floating-point number x is generally represented by 3 numbers, a sign s, a mantissa m and an exponent e as follows:

$$x = (-1)^s \cdot m \cdot 2^e$$

Using floating-point numbers provides computing time for each operation in $\mathcal{O}(b)$ where b is the size of the mantissa, well below n; the cost of each operation is independent of n and is considered as a constant value. The calculations are approximate, but with the use of libraries such as MPFR (Fousse et al 2007) , the result is guarantee to be the closest value to the exact value, representable with the precision defined.

Denise and Zimmermann (1999) already studied the error propagation done when computing the $l_s(i)$ with a floating-point arithmetic. Since their results are for all classes of decomposable structure, we adapt their results for the error propagation done when computing the counting-table presented in Section 2, which correspond to the class of regular languages.

Theorem 4 *If the counting table is computed using floating-point arithmetic with mantissa of size b, then the approximation $\widetilde{l_s}(n)$ of $l_s(n)$ is such that:*

$$|\widetilde{l_s}(n) - l_s(n)| \leq n \cdot d \cdot 2^{1-b} \cdot l_s(n)$$

Hence, the error is bounded by $\mathcal{O}(n \times d \times 2^{-b})$.

Proof This theorem can be proof by induction. The floating-point addition $\oplus$ has the following property: If $\tilde{a} = a(1+\delta_a)$ and $\tilde{b} = b(1+\delta_b)$, then:

$$|(\tilde{a} \oplus \tilde{b}) - (a+b)| \leq |a+b|(\max(\delta_a, \delta_b) + 2^{1-b})$$

Thus, by assuming all $l_s(0)$ are exactly represented in floating-point numbers, one obtains:

$$|(\widetilde{l_s}(1)) - l_s(1))| \leq l_s(1)(d \cdot 2^{1-b})$$

And, by induction, one gets:

$$|(\widetilde{l_s}(k)) - l_s(k))| \leq l_s(k)(k \cdot d \cdot 2^{1-b})$$

Note that in practice, both in Denise and Zimmermann's experiences and those presented in Section 5, the measured error is much lower than the bound.

Proposition 1 *The bit complexities of the algorithm with counting table using floating-point arithmetic are as follows.*

The bit complexity of the preprocessing stage is:

- *time: in* $\mathcal{O}(n \times b \times d \times S)$,
- *space: in* $\mathcal{O}(n \times b \times S)$.

The bit complexity of the generation stage is:

- *time: in* $\mathcal{O}(n \times b \times d)$,
- *space: in* $\mathcal{O}(n \times b \times S)$.

Proposition 2 *The bit complexities of the algorithm without counting table using floating-point arithmetic are as follows.*

The bit complexity of the preprocessing stages is:

- *time: in* $\mathcal{O}(n_0^2 \times d \times S^2 + r^3 \times n_0 + n \times b \times d \times S)$ *where* r *is the rank of* A^{n_0},
- *space: in* $\mathcal{O}(n_0 \times b \times S)$.

The bit complexity of the generation stage is:

- *time: in* $\mathcal{O}(n \times b \times d \times S)$,
- *space: in* $\mathcal{O}(n_0 \times b \times S)$.

Table 2 summarizes time-complexity and space-complexity for each version: with or without counting-table, using integer or floating-point arithmetic. In this table, d, b, r and n_0 values are fixed.

5 Experimental results

In this section, we present results of comparing elapsed time and memory consumption for all versions. Moreover, we studied the quality of the floating-point calculus by measuring the relative error.

5.1 Implementation and methodology

The system of inverse recurrences, which is needed for the generation of paths without the counting-table, is made by the following process. A SAGE script (a free computer algebra (Stein 2007)) is automatically created from the model graph. This script begins by building the matrix transitions (using a sparse matrix representation), then it computes the minimum index from which the inversion is possible and finally it calculates the system of inverse recurrences using the SAGE function `solve_left`.

We did all our experiments on a dedicated server whose hardware is composed of an Intel Xeon 3GHz processor with 16GB memory. Each graph used for our experiments comes from the VLTS (Very Large Transition Systems (Garavel and Descoubes 2003)) benchmark suite. These models correspond to real industrial systems and their associated transition matrix is sparse (i.e., d is much smaller than S). Each model name is of the form *vasy_X_Y*, where X is the number of states divided by 1000, and Y is the number of transitions divided by 1000.

All versions of the uniform generation of paths are available via a common interface written in *C++* (source code is in my website). This generic interface makes easy to switch from one version to another. I used several tools and libraries: `bcg_io` command from CADP toolbox (Garavel et al 2001) to convert graphs from BCG to `GraphViz` format.; BOOST.GRAPH library to handle graphs; BOOST.RANDOM (Maurer et al 2000) library to generate random numbers; GMP (Granlund 2007) and MPFR (Fousse et al 2007) for integer and floating-point calculus, respectively.

5.2 Memory consumption

Figure 4 shows memory usage for each of the four variants: *tbl_exact* for the original algorithm, namely the calculation of the counting table with infinite precision; *tbl_float* for the version with the counting table but using floating-point arithmetic with 64-bit precision; *inv_exact* and *inv_float* for versions that do not keep the counting table in memory and where calculations are done with infinite precision and with 64-bit precision, respectively. For each of the 3 models, the four variants have been implemented to generate paths whose length varies between 200 and 40000. The 100000 MiB is fictive and means that the method required more memory than the 16 GiB available. Thus, it is impossible to generate paths of length 10000 or more in *vasy_5_9* with the original method.

		Preprocessing		Generation	
Table	Arith	Time	Space	Time	Space
with	integer	$\mathcal{O}(n^2 \times S)$	$\mathcal{O}(n^2 \times S)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2 \times S)$
without	integer	$\mathcal{O}(n^2 \times S + S^2)$	$\mathcal{O}(n \times S)$	$\mathcal{O}(n^2 \times S)$	$\mathcal{O}(n \times S)$
with	float	$\mathcal{O}(n \times S)$	$\mathcal{O}(n \times S)$	$\mathcal{O}(n)$	$\mathcal{O}(n \times S)$
without	float	$\mathcal{O}(n \times S + S^2)$	$\mathcal{O}(S)$	$\mathcal{O}(n \times S)$	$\mathcal{O}(S)$

Table 2 Summary of bit complexities both in time and in space for all versions. n means the maximal length of paths and S the number of states in the automaton. To clarify the complexities, we have considered as constant values the following values: the maximal output degree d of the automaton A, the value n_0 from which recurrences can be reversed, the rank r of A^{n_0}, and the mantissa b chosen for representing floating-point numbers

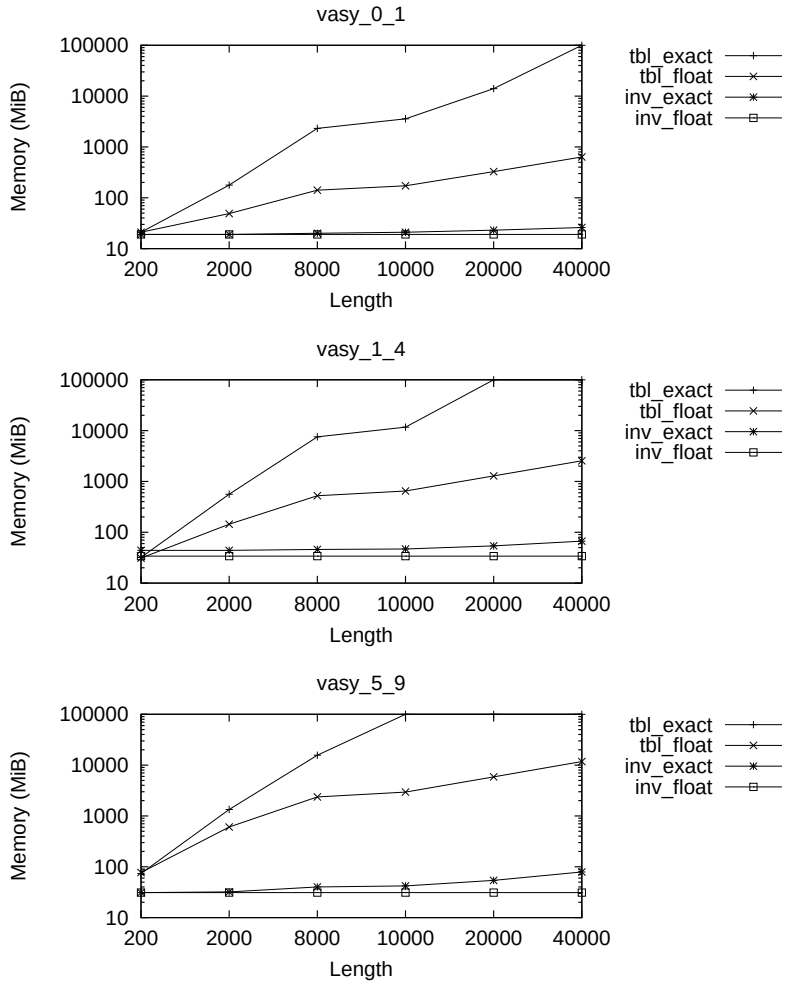

Fig. 4 Comparison of memory consumption of the four variants depending on the length of paths. There are three different models

Each time, the floating-point version requires less memory than its counterpart in the exact calculation. Regarding the versions that do not keep in memory the counting table, they still consume much less memory than alternatives with a counting-table, except for very short lengths where the storage of the matrix B for reverse recurrences may be more expensive than the counting table itself.

5.3 Running time

Figure 5 summarizes the time taken by each of the four variants to draw 100 paths in the 3 same models as before. The generation time is longer for variants without counting table; one path has been generated with these variants and the generation

time was multiplied by 100 to get the generation time of 100 paths. In addition, the fictive value of 100 million seconds means that the experiment lasted more than 24 hours.

The methods without counting-table can generate very long paths using little memory. However, when it is possible to keep the counting table in memory, it is preferable to use the alternatives with table that can generate paths more quickly.

The floating-point methods are faster and consume less memory than their counterparts in exact calculation. We might think it is always preferable to use floating-point performance, but care must be taken to the possible deviation from the uniformity which could be problematic. That is the study of the next section.

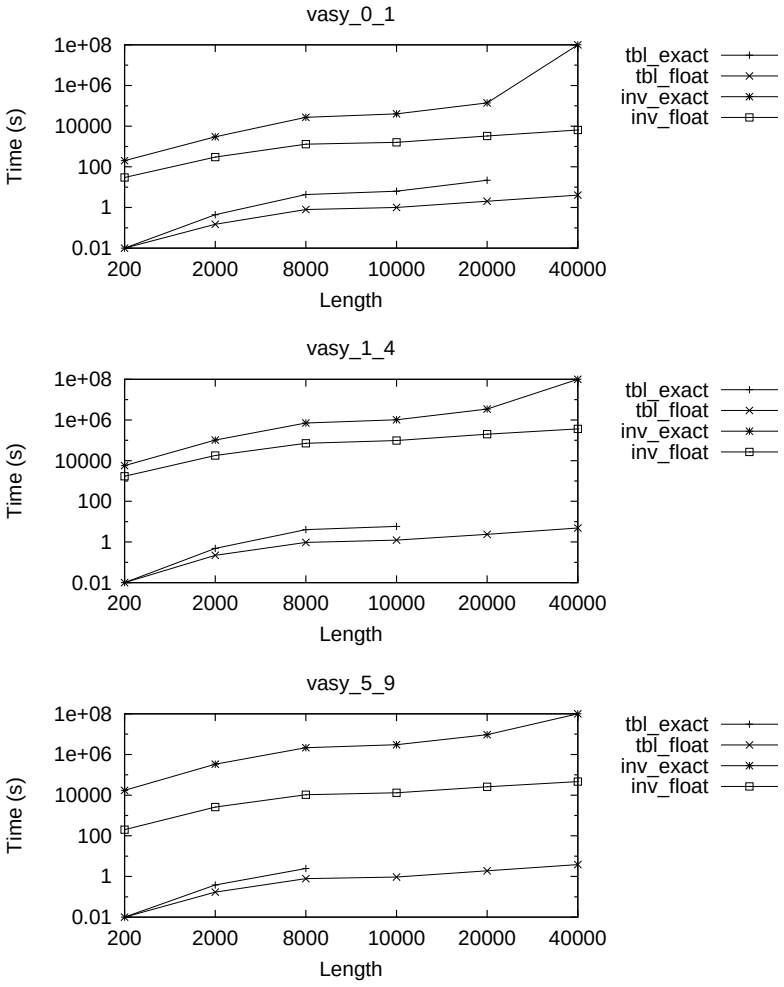

Fig. 5 Time taken by each of the four variants to draw 100 paths in all 3 models

5.4 Deviation from the uniformity

Using the same notation $l_s(i)$ as defined in Section 2, computed with infinite precision, and using $\widetilde{l_s}(i)$ for the same value but computed in a floating-point arithmetic, one can measure the maximum relative error by the following formula:

$$Err = \max_{\substack{s \in S \\ 0<i\leq n}} \frac{|l_s(i) - \widetilde{l_s}(i)|}{l_s(i)} \tag{6}$$

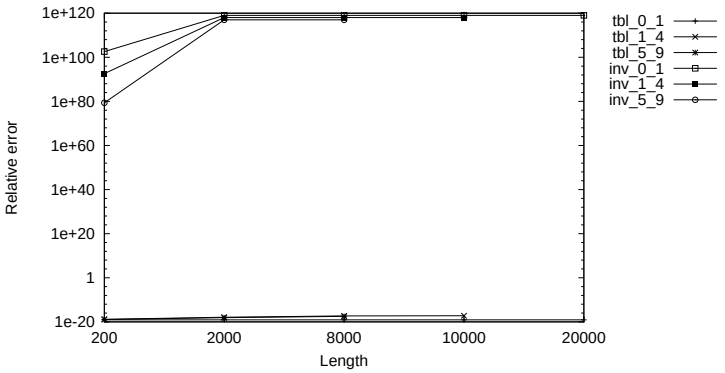

Fig. 6 Measurement of the maximum relative error, defined by Equation (6), for the two variants that use floating-point (*inv* and *tbl*) on the 3 models

Figure 6 shows the Err value of the two methods using 64-bit floating-point arithmetic executed on the 3 models. In each experiment, the algorithm with counting table is numerically stable with a maximum relative error of less than 10^{-17}. However, the version without the table is numerically unstable. This numerical instability is due to the presence of negative values in the matrix B which can cause a phenomenon of cancellation when subtracting two close numbers.

We can give more details about the error propagation in the case of the algorithm without counting-table. Remember that reverse recurrences have the following shape:

$$l_s(i) = \sum_{s'} r_{s,s'} \times l'_s(i+1) \quad \text{with } r_{s,s'} \in \mathbb{Q}$$

If ε_i^s is the error of $l_s(i)$, then the error propagation is as follows:

$$\begin{aligned}
\varepsilon_{n-1}^s &= \sum_t \varepsilon_n^t \times |r_{st}| \\
\varepsilon_n &= \max_s \varepsilon_n^s \\
\varepsilon_{n-1} &\leq \max_s \sum_t \varepsilon_n |r_{st}| \\
\varepsilon_{n-1} &\leq \left[\max_s \sum_t |r_{st}| \right] \varepsilon_n
\end{aligned}$$

Thus, the error propagation follows a geometric distribution with parameter $\max_{s,s'} |r_{s,s'}|$. In other word, the algorithm without counting-table is numerically unstable as soon as there is a value greater than 1 in B, which is the case in the 3 models studied.

6 Conclusion and perspectives

We developed methods that allow the efficient exploration of very large models. They perform a random exploration while ensuring a good coverage of paths whatever the model topology, which is not the case with an isotropic random walk. Note that these methods could be extended to cover states and transitions in accordance with the principles given in (Gaudel et al 2008; Gouraud et al 2001).

A limit that one could accuse the existing version of this method was the necessity of the counting table. This table requires having a memory space proportional to the product of number of states of the automaton (S) by the length of paths (n), i.e. in $\mathcal{O}(n^2 \times S)$, which is considerable when one is interested to generate long paths in very large models (Gaudel et al 2008). With the improvements introduced in sections 3 and 4, memory space is now in $\mathcal{O}(S)$. This is excellent because we recall that in the case of a modular exploration S denotes the size of the component and not the overall system.

In addition, experiments conducted in Section 5 showed that variants with floating-point computation use less memory and are faster than their counterparts with exact calculation. These benefits are to the detriment of the accuracy of uniformity on paths (although in practice no difference was found for the variant with counting-table). The version without counting-table is useful if resources do not allow to store the counting table, but the generation is slower. These methods offer attractive alternatives to the use of isotropic random walks to explore models, either for simulation, random testing, or model-checking.

The choice of the upper bound n on the path length can be difficult to define, but depending on the desired coverage criterion, it is often easy to determine. For example, if the criterion is to consider the paths that cross at most once in each loop, n is equal to the length of the longest elementary path. In the field of model-checking, this bound depends on the property to check, but often the diameter is used. However, for cases where we do not know this or is simply too large, n can be quite arbitrarily chosen, often from the number of states in the system. The choice of this integer has similarities with the choice of an objective function, as it is the case in all metaheuristics that are currently the alternatives to the isotropic random walks. A new perspective that would overcome the selection of this bound is to use Boltzmann sampling (Flajolet et al 2007) whose parameter sets the average length of paths obtained, instead of having a fixed length.

Acknowledgements I thank Matthias Krieger for his valuable assistance which help solving the linear algebra problem defined in Equation (5), Alain Denise and Marie-Claude Gaudel for their active support during the redaction of this article, and I am also very graceful to Andreas Enge and Paul Zimmermann for fruitful discussions about error calculus.

References

Clarke L, Podgurski A, Richardson D, Zeil S (1989) A formal evaluation of data flow path selection criteria. IEEE Transactions on Software Engineering 15(11):1318–1332

Denise A, Zimmermann P (1999) Uniform random generation of decomposable structures using floating-point arithmetic. TCS 218:233–248

Denise A, Gaudel MC, Gouraud SD (2004) A generic method for statistical testing. In: ISSRE, pp 25–34

Dwyer M, Elbaum S, Person S, Purandare R (2007) Parallel randomized state-space search. In: ICSE, pp 3–12

Flajolet P, Zimmermann P, Cutsem BV (1994) A calculus for the random generation of labelled combinatorial structures. TCS 132:1–35

Flajolet P, Fusy É, Pivoteau C (2007) Boltzmann sampling of unlabelled structures. In: ANALCO, SIAM Press, vol 126, pp 201–211

Fousse L, Hanrot G, Lefèvre V, Pélissier P, Zimmermann P (2007) MPFR: A multiple-precision binary floating-point library with correct rounding. ACM Transactions on Mathematical Software 33(2):13:1–13:15, URL http://doi.acm.org/10.1145/1236463.1236468

Garavel H, Descoubes N (2003) Very large transition systems. http://tinyurl.com/yuroxx

Garavel H, Lang F, Mateescu R (2001) An overview of cadp 2001. Tech. Rep. 254, INRIA

Gaudel MC, Denise A, Gouraud SD, Lassaigne R, Oudinet J, Peyronnet S (2008) Coverage-biased random exploration of large models. In: MBT

Gouraud SD, Denise A, Gaudel MC, Marre B (2001) A new way of automating statistical testing methods. In: ASE, pp 5–12

Granlund T (2007) GNU MP: The GNU Multiple Precision Arithmetic Library, version 4.2.4. http://gmplib.org/manual/

Grosu R, Smolka S (2005) Monte carlo model checking. In: TACAS, pp 271–286

Hérault T, Lassaigne R, Magniette F, Peyronnet S (2004) Approximate probabilistic model checking. In: VMCAI, pp 73–84

Knuth DE (1997) Seminumerical Algorithms, The Art of Computer Programming, vol 2, 3rd edn. Addison-Wesley, Boston, MA, USA

Maurer J, Abrahams D, Dawes B, Rivera R (2000) Boost random number library. http://www.boost.org/libs/random/

Stein W (2007) SAGE Mathematics Software. The SAGE Group, http://www.sagemath.org