A Comparison of Random Binary Tree Generators

Article *in* The Computer Journal · June 2002

DOI: 10.1093/comjnl/45.6.653 · Source: DBLP

2 authors, including:

Erkki Mäkinen
Tampere University
168 PUBLICATIONS **1,873** CITATIONS

SEE PROFILE

A Comparison of Random Binary Tree Generators

JARMO SILTANEVA[1] AND ERKKI MÄKINEN[2]

[1]*Information Technology Center, City of Tampere, Lenkkeilijänkatu 8, Finn-Medi 2, FIN-33520 Tampere, Finland*
[2]*Department of Computer and Information Sciences, PO Box 607, FIN-33014 University of Tampere, Finland*
Email: em@cs.uta.fi, Jarmo.Siltaneva@tt.tampere.fi

This paper empirically compares five linear-time algorithms for generating unbiased random binary trees. More specifically, we compare the relative asymptotic performance of the algorithms in terms of the numbers of various basic operations executed on average per tree node. Based on these numbers a ranking of the algorithms, which depends on the operation weights, can be deduced. A definitive ranking of the algorithms, however, hardly exists.

Received 1 January 2002; revised 15 May 2002

1. INTRODUCTION

Binary trees are essential in various branches of computer science [1]. From time to time, there is a need to generate random binary trees. For example, when testing or analysing a program that manipulates binary trees, it is advantageous to have an efficient method to generate random binary trees with a given number n of nodes.

In this paper we only consider algorithms that assign equal probability to all members of the family of trees with n nodes, i.e. we always use the uniform distribution. Mathematically, there are no problems at all in generating random binary trees. Namely, there exist algorithms to enumerate binary trees. Simply choose a random natural number i from the interval $[1 \ldots C_n]$, where the nth Catalan number

$$C_n = \binom{2n}{n}\frac{1}{n+1}$$

gives the number of binary trees with n nodes, and output the ith binary tree from the enumeration with an unranking algorithm.

Computationally the problem is more complicated. As Martin and Orr [2] observe, C_{5000} needs over 2000 digits in decimal notation. It is preferable that algorithms generating random binary trees use only integers of polynomial size on n or, if possible, of size $O(n)$. Notice, however, that we also have to use probabilities; that is, real numbers from the interval $[0 \ldots 1]$.

The problem of exponential numbers in generating unbiased random binary trees can be tackled through the use of binary tree codings. Instead of directly generating binary trees, the algorithms actually generate random code words which are in one-to-one correspondence with binary trees. This requires an efficient way to travel between trees and code words. All the coding schemes considered here allow efficient transformation algorithms. These transformations are obvious when the code words are given, and are not considered in this paper. For further details concerning these transformations, the reader is referred to the original articles introducing the methods [2, 3, 4, 5, 6].

The rest of this paper is organized as follows. In Section 2 we recall the algorithms. Section 3 describes the organization of our tests and introduces the results in several data tables. In Section 4 we sum up the computational costs given in Section 3. Finally, in Section 5 we draw our conclusions. Sections 3–5 are based on [7].

2. THE ALGORITHMS

There are several linear-time random binary tree generators (for a survey, see [8]). In this section we recall five such algorithms. These algorithms can produce code words for random binary trees without any preliminary computations, while there is a sixth algorithm introduced by Johnsen [9] requiring a preprocessing phase which takes $O(n^2)$ time and space. Johnsen's algorithm is not considered here since, besides the need of a preprocessing phase, it uses integers of exponential size on n. Algorithms for structures related to binary trees are also presented in the literature, see e.g. [10].

2.1. Arnold and Sleep

Strings of balanced parentheses are well-known to be in one-to-one correspondence with binary trees. The set of all balanced parentheses can be generated by the grammar with productions

$$S \to \{S\}S,\ S \to \lambda,$$

where λ stands for the empty string. (For notational clarity we use strings of curly brackets { and }.)

Suppose we are generating binary trees with n nodes. The corresponding strings of balanced parentheses are of length $2n$. From left to right, we construct a balanced

string by repeatedly choosing between a left parenthesis (corresponding to the production $S \rightarrow \{S\}S$) and a right parenthesis (the production $S \rightarrow \lambda$). The decision probabilities depend only on the number r of unmatched left parentheses produced so far and on the total number k of symbols remaining to be produced. Let $A(r, k)$ denote the number of valid continuations when there are r unmatched left parentheses and k symbols remaining to be produced.

The probabilities to produce left and right parentheses can be expressed in terms of A. Namely, the number of valid continuations starting with a left parenthesis is $A(r+1, k-1)$ and the number of valid continuations starting with a right parenthesis is $A(r-1, k-1)$. Hence, the probability $P(r, k)$ to produce a right parenthesis when there are r unmatched left parentheses and k symbols remaining to be produced is

$$P(r, k) = \frac{A(r-1, k-1)}{A(r, k)}. \quad (1)$$

A string of parentheses can be geometrically represented as a zig-zag line, starting from the origin and consisting of northeast and southeast edges of equal length. Each northeast edge represents a left parenthesis and each southeast edge represents a right parenthesis. A balanced string of parentheses has a drawing in which the line returns to the base line and has no edges below it.

A geometric representation of the situation where there are r unmatched left parentheses and k symbols remaining to be produced is a similar zig-zag line from the point $(0, r)$ to the point $(k, 0)$, not entering the negative region of the plane. Such paths are called *positive paths*. The other paths are *negative paths*. Arnold and Sleep [3] determine the number of positive paths, that is $A(r, k)$, by subtracting the number of negative paths from the total number of paths. This difference is

$$A(r, k) = \frac{2(r+1)}{(k+r+2)} \binom{k}{(k+r)/2}. \quad (2)$$

Based on (1) and (2), the probability $P(r, k)$ can now be written as

$$P(r, k) = \frac{r(k+r+2)}{2k(r+1)}. \quad (3)$$

Note that $r = k$ gives $P(r, k) = 1$.

Equation (3) solves the problem of generating random binary trees: we choose random real numbers from the interval $[0 \ldots 1]$ and compare them to the results obtained by Equation (3) with the present values of r and k.

2.2. Atkinson and Sack

Atkinson and Sack [4] give a divide-and-conquer algorithm to generate random strings of balanced parentheses.

A string of parentheses is said to be *balanced with defect* i if: (1) it contains an equal number of left and right parentheses, i.e. its zig-zag line returns to the base line; and (2) the zig-zag line has precisely $2i$ edges below the base line. Note that the set of balanced strings with defect 0 is the one having a one-to-one natural correspondence with binary trees.

Let B_{ni} stand for the set of balanced strings with defect i and with length $2n$. The sets $B_{n0}, B_{n1}, \ldots, B_{nn}$ are disjoint and their union B_n is the set of all strings of parentheses containing an equal number of left and right parentheses. All the sets B_{ni} have the same size $\binom{2n}{n}(1/(n+1))$ [11]. The algorithm of Atkinson and Sack chooses a random member of B_n and transforms it into the corresponding member of B_{n0}.

If w is a string of parentheses, we denote by $\overline{w}$ the string obtained by replacing each left parenthesis by a right parenthesis and each right parenthesis by a left parenthesis.

Let w be a string (not necessarily balanced) containing an equal number of left and right parentheses. We say that w is *reducible* if it can be written in the form $w = w_1 w_2$, where both w_1 and w_2 are non-empty and contain an equal number of left and right parentheses. Otherwise, w is *irreducible*. If an irreducible string w contains an equal number of left and right parentheses, then either w or $\overline{w}$ is balanced. Moreover, w has a unique factorization $w = w_1 w_2 \ldots w_k$, where each w_i is irreducible [4].

The algorithm of Atkinson and Sack first generates a random combination X of n integers from $\{1, 2, \ldots, 2n\}$. This is possible in linear time (for details see, e.g., [12]). Next, a random string $x = x_1 x_2 \ldots x_{2n}$ of parentheses is constructed by setting $x_i = \{$ provided that $i \in X$; otherwise $x_i = \}$. There are an equal number of left and right parentheses in x, and hence x is in B_n.

The crux of the algorithm is the mapping of x to a unique member of B_{n0}. Formally, we need a map $\Phi : B_n \rightarrow B_{n0}$ defined inductively as follows. When $n = 0$, we have $\Phi_0(\lambda) = \lambda$. For $n > 0$, we express $w \in B_n$ as $w = uv$, where u is non-empty and irreducible and v is of length $s \geq 0$. Now we define Φ_n by setting $\Phi_n(w) = u\Phi_s(v)$, if u is balanced; otherwise $\Phi_n(w) = \{\Phi_s(v)\}\overline{t}$, where $u = \}t\{$. It is possible to prove that Φ_n is bijective on each B_{ni} [4].

The method of Atkinson and Sack has the nice feature that it uses only integers of size at most $2n$ [4].

2.3. Korsh

Korsh [5] introduced a random binary tree generation algorithm based on bit sequence rotations. Korsh's method uses a binary tree coding scheme where the tree is given as a sequence of bit pairs, one pair for each node. The bits indicate whether or not the node has a non-null left and right subtree. Pairs are given in preorder. The code word of a tree with n nodes contains $(n-1)$ 1-bits and $(n+1)$ 0-bits. A *k-rotation* related to the node k of a Korsh's code word is obtained by shifting the first $k-1$ pairs from the front to the end of the code word. Korsh [5] shows that any sequence of bit pairs with $(n-1)$ 1-bits and $(n+1)$ 0-bits is either a valid code word or a k-rotation of a unique code word. Hence, it is sufficient to randomly generate a bit pair sequence of appropriate length (as in the method of Atkinson and Sack) and then find the corresponding Korsh's code word as follows. Suppose that d is an arbitrary sequence of bit pairs with $(n-1)$ 1-bits and $(n+1)$ 0-bits. Find the shortest prefix of d where the number of 0-bits exceeds the

number of 1-bits by two. If this prefix is proper (i.e. differs from d itself), shift the prefix to the end of d and repeat the operation. This process will halt in linear time with the desired result [5].

Like the method of Atkinson and Sack, Korsh's method uses only integers of size at most $2n$ [5].

2.4. Martin and Orr

Consider now the following coding method for binary trees. Each node in the right arm (the path from the root following right child pointers) is labelled with 0. If a node is a left child, its label is $i + 1$ where i is the label of the parent. The label of a right child is the same as the label of its parent. Read the labels in preorder. The code word obtained is called an *inversion table* in [2].

Generating a binary tree is now equivalent to generating a code word $(x_1, x_2, \ldots, x_n)$. If $x_j = i$, Martin and Orr [2] use a cumulative probability distribution function $F(k)$, which gives the probability that $x_{j+1} \in \{0, \ldots, k\}, k \leq i+1$. If a is the number of all valid code words with the prefix so far produced, and b is the number of valid code words with the prefix so far produced augmented with any code item from the set $\{0, \ldots, k\}$, then $F(k) = b/a$. More generally, F is a function of n, the length of the code word, i, the previous code item, j, the position in the code word, and k, the upper bound for the next code item to be determined.

Martin and Orr [2] give the following formula

$$F(n, i, j, k) = \frac{(k+1)(n-j+i+2)!(2n-2j+k)!}{(i+2)(n-j+k+1)!(2n-2j+i+1)!}.$$

We can now choose a random number x from the interval [0, 1), and find the largest m such that $x \geq F(n, i, j, m-1)$. Then m is the next code item.

Let $P(n, i, j, k) = F(n, i, j, k) - F(n, i, j, k-1)$ denote the probability that k is the next code item and let $Q(n, j, k) = P(n, i, j, k-1)/P(n, i, j, k)$. To dispense with the factorials, Martin and Orr [2] derive the formulae

$$Q(n, j, k) = \frac{(k+1)(n-j+k+1)}{(k+2)(2n-2j+k-1)}$$

and

$$P(n, i, j, i+1) = \frac{(i+3)(n-j)}{(i+2)(2n-2j+i+1)}.$$

Because of the fact $P(n, i, j, k-1) = Q(n, j, k)P(n, i, j, k)$, it is now easy to compute the values of P for all necessary k's starting at $i + 1$ (the highest possible value for k) and continuing iteratively towards 0 (the lowest possible value).

2.5. Rémy

Rémy [6] gave the following inductive algorithm to generate a random binary tree with n internal nodes and $n + 1$ leaves:

- suppose that so far we have a binary tree with k internal nodes and $k + 1$ leaves;
- randomly select one of the $2k + 1$ nodes, denote the selected node by v;
- replace v by a new node;
- randomly choose v to be the left or right child of the new node, the other child of the new node is a new leaf, the subtrees of v are kept unchanged;
- repeat the process of inserting nodes until the tree has n internal nodes and $n + 1$ leaves.

The correctness of Rémy's algorithm can be proved by considering binary trees with leaves labelled by numbers $1, \ldots, n + 1$. Namely, it is easy to show by induction (see [6, 13] for details) that the algorithm generates all

$$C_n(n+1)! = \frac{2n!}{n!}$$

binary trees with labelled leaves with probability

$$\frac{n!}{2n!}.$$

We use the implementation of Rémy's algorithm given in [14].

3. THE TESTS

The algorithms recalled in the previous section are all of linear-time complexity. Hence, in order to compare the algorithms we have to use finer methods than the order of magnitude of the time complexities.

A straightforward method is to compare execution times. The results so obtained, however, depend on the environment in which the tests are performed. More general information is obtained if we count the numbers of various types of operations executed. We record the numbers of the following operation types:

- arithmetic operations (abbreviated in the sequel as ADD, MUL and DIV);
- array references (ARR);
- random number generator calls (RAN);
- variable references and assignment statements (LOA, STO);
- arithmetic and logical comparisons (CMP);
- pointer references (PTR);
- recursive procedure calls (REC);
- miscellaneous operations (OTH).

We have to make a few simplifying assumptions concerning the recording. For example, we treat logical expressions as if they were always completely evaluated, thus disregarding situations where the value of an expression becomes known before the end of the evaluation. We count a multiplication by 2 as an addition and not as a bit shift. Furthermore, we disregard type conversions and the size of operands in arithmetic operations. The majority of the ADD operations are additions or subtractions by one, mostly in loop counter variables. We count these operations as normal additions instead of increment operations.

TABLE 1. The average numbers of the operations per node from the random generation of code words by Arnold and Sleep's algorithm.

n	t	LOA	STO	CMP	ADD	MUL	DIV	ARR	RAN	OTH
4	10,000	34.99	10.75	5.00	10.25	2.00	1.00	2.00	1.00	0.00
10	10,000	39.29	11.30	5.45	12.10	3.00	1.50	2.00	1.50	0.00
100	10,000	43.44	11.91	5.93	13.78	3.88	1.94	2.00	1.94	0.00
1000	5000	43.94	11.99	5.99	13.98	3.99	1.99	2.00	1.99	0.00
10,000	1000	43.99	12.00	6.00	14.00	4.00	2.00	2.00	2.00	0.00
100,000	100	44.00	12.00	6.00	14.00	4.00	2.00	2.00	2.00	0.00

TABLE 2. The average numbers of the operations per node from the random generation of code words by Atkinson and Sack's algorithm.

n	t	LOA	STO	CMP	ADD	MUL	DIV	ARR	RAN	REC	OTH
4	10,000	85.28	26.90	12.32	15.91	1.00	0.00	15.35	1.00	0.91	0.91
10	10,000	81.07	25.68	11.34	15.57	1.00	0.00	15.53	1.00	0.57	0.57
100	10,000	76.74	24.47	10.38	15.17	1.00	0.00	15.83	1.00	0.18	0.18
1000	5000	75.52	24.14	10.11	15.04	1.00	0.00	15.95	1.00	0.06	0.06
10,000	1000	75.16	24.04	10.04	15.01	1.00	0.00	15.98	1.00	0.02	0.02
100,000	100	75.06	24.01	10.01	15.01	1.00	0.00	15.99	1.00	0.01	0.01

The correctness of the implementations used in the tests (i.e. the randomness of the trees produced) was verified by using the Chi-square test, as in [14].

As an example, we give the detailed code for the Martin–Orr algorithm (cf. Section 2.4) with the recordings of the operations. The procedure generates one random code word. Notation $c(\mathrm{INS}, x)$ stands for incrementing x times the counter corresponding the operation type INS:

```
void main()
{
  int i, j, k, n,
      x[n+1]; // code word 1...n
  double random, sum, p, q;

  x[1] = 0;
         // c(LOA,2); c(ARR,1); c(STO,1);
         // c(LOA,4*n-1); c(STO,n); // for
         // c(CMP,n); c(ADD,n-1); // for

  for (j=1; j<=n-1; j++)
  {
    i = x[j];
    k = i+1;
         // c(LOA,4); c(STO,2);
         // c(ADD,1); c(ARR,1);
    p = double((i+3)*(n-j)) /
         double((i+2)*(2*n-2*j+i+1));
         // c(LOA,12); c(STO,1);
         // c(ADD,8); c(MUL,2); c(DIV,1);
    sum = p;
    random = randomGen();
         // c(RAN,1); c(LOA,1); c(STO,2);
    while (random > sum)
    {
         // c(LOA,2); c(CMP,1); // while
      q = double((k+1)*(n-j+k+1)) /
         double((k+2)*(2*n-2*j+k-1));
         // c(LOA,14); c(STO,1);
         // c(ADD,10); c(MUL,2); c(DIV,1);
      p = q*p;
      sum = sum+p;
      k = k-1;
         // c(LOA,6); c(STO,3);
         // c(ADD,2); c(MUL,1);
    }
       // c(LOA,2); c(CMP,1); // while

    x[j+1] = k;
         // c(LOA,3); c(STO,1);
         // c(ADD,1); c(ARR,1);
  }
}
```

The above algorithm generates code words that represent binary trees according to the coding system by Martin and Orr's algorithm. Because the algorithms use different coding systems, we record the costs in two stages: generating random code words and constructing the binary trees from the code words.

We apply the unit cost principle, i.e. operation costs do not depend on the size of the integers handled. The tests were performed in an environment where this is possible for integers not exceeding $2^{31} - 1$.

We use the random number generator of Park and Miller [15] with parameters $a = 7^5$ and $b = 2^{31} - 1$ giving more than 2×10^9 pseudorandom numbers before the sequence repeats itself.

TABLE 3. The average numbers of the operations per node from the random generation of code words by Korsh's algorithm.

n	*t*	LOA	STO	CMP	ADD	MUL	DIV	ARR	RAN	OTH
4	10,000	103.10	27.41	20.21	18.03	0.75	0.00	15.25	0.75	0.00
10	10,000	105.20	27.34	20.15	19.26	0.90	0.00	16.30	0.90	0.00
100	10,000	106.10	27.13	20.05	19.96	0.99	0.00	16.93	0.99	0.00
1000	5000	106.01	27.03	20.00	19.99	1.00	0.00	16.99	1.00	0.00
10,000	1000	105.33	26.87	19.82	19.86	1.00	0.00	16.95	1.00	0.00
100,000	100	105.95	26.99	19.99	19.99	1.00	0.00	17.00	1.00	0.00

TABLE 4. The average numbers of the operations per node from the random generation of code words by Martin and Orr's algorithm.

n	*t*	LOA	STO	CMP	ADD	MUL	DIV	ARR	RAN	OTH
4	10,000	31.86	7.77	2.26	14.31	3.02	1.26	1.75	0.75	0.00
10	10,000	40.40	9.50	2.65	18.90	4.05	1.65	1.90	0.90	0.00
100	10,000	47.14	10.83	2.96	22.54	4.89	1.96	1.99	0.99	0.00
1000	5000	47.91	10.98	3.00	22.95	4.99	2.00	2.00	1.00	0.00
10,000	1000	47.99	11.00	3.00	23.00	5.00	2.00	2.00	1.00	0.00
100,000	100	48.00	11.00	3.00	23.00	5.00	2.00	2.00	1.00	0.00

TABLE 5. The average numbers of the operations per node from the random generation of code words by Rémy's algorithm.

n	*t*	LOA	STO	CMP	ADD	MUL	DIV	ARR	RAN	OTH
4	10,000	36.82	16.25	3.83	3.00	2.00	0.00	9.91	2.00	9.91
10	10,000	36.14	15.50	3.89	3.00	2.00	0.00	9.87	2.00	9.87
100	10,000	35.97	15.05	3.98	3.00	2.00	0.00	9.96	2.00	9.96
1000	5000	35.99	15.00	4.00	3.00	2.00	0.00	9.99	2.00	9.99
10,000	1000	36.00	15.00	4.00	3.00	2.00	0.00	10.00	2.00	10.00
100,000	100	36.00	15.00	4.00	3.00	2.00	0.00	10.00	2.00	10.00

TABLE 6. The average numbers of the operations per node from the construction of binary trees from the code words by Arnold and Sleep's and Atkinson and Sack's algorithms.

n	*t*	LOA	STO	CMP	ADD	ARR	PTR	REC	OTH
4	10,000	13.75	4.75	3.00	2.00	2.00	3.50	2.00	0.00
10	10,000	13.90	4.90	3.00	2.00	2.00	3.80	2.00	0.00
100	10,000	13.99	4.99	3.00	2.00	2.00	3.98	2.00	0.00
1000	5000	14.00	5.00	3.00	2.00	2.00	4.00	2.00	0.00
10,000	1000	14.00	5.00	3.00	2.00	2.00	4.00	2.00	0.00

TABLE 7. The average numbers of the operations per node from the construction of binary trees from the code words by Korsh's algorithm.

n	*t*	LOA	STO	CMP	ADD	ARR	PTR	REC	OTH
4	10,000	12.75	5.75	2.00	2.00	2.00	1.50	1.00	2.00
10	10,000	12.90	5.90	2.00	2.00	2.00	1.80	1.00	2.00
100	10,000	12.99	5.99	2.00	2.00	2.00	1.98	1.00	2.00
1000	5000	13.00	6.00	2.00	2.00	2.00	2.00	1.00	2.00
10,000	1000	13.00	6.00	2.00	2.00	2.00	2.00	1.00	2.00

TABLE 8. The average numbers of the operations per node from the construction of binary trees from the code words by Martin and Orr's algorithm.

n	t	LOA	STO	CMP	ADD	ARR	PTR	REC	OTH
4	10,000	9.75	4.25	2.50	0.75	0.75	1.50	1.00	0.75
10	10,000	11.10	4.70	2.80	0.90	0.90	1.80	1.00	0.90
100	10,000	11.91	4.97	2.98	0.99	0.99	1.98	1.00	0.99
1000	5000	11.99	5.00	3.00	1.00	1.00	2.00	1.00	1.00
10,000	1000	12.00	5.00	3.00	1.00	1.00	2.00	1.00	1.00

TABLE 9. The average numbers of the operations per node from the construction of binary trees from the code words by Rémy's algorithm.

n	t	LOA	STO	CMP	ADD	ARR	PTR	REC	OTH
4	10,000	23.25	8.75	6.75	0.00	4.50	4.00	2.25	9.00
10	10,000	21.90	8.30	6.30	0.00	4.20	4.00	2.10	8.40
100	10,000	21.09	8.03	6.03	0.00	4.02	4.00	2.01	8.04
1000	5000	21.01	8.00	6.00	0.00	4.00	4.00	2.00	8.00
10,000	1000	21.00	8.00	6.00	0.00	4.00	4.00	2.00	8.00

TABLE 10. The weights of the operations.

LOA	STO	CMP	ADD	MUL	DIV	ARR	PTR	REC	RAN	OTH
1	1	1.5	1.5	2	2	4.5	2	10	32	2

3.1. Generating code words

In this section we give the observed numbers of the operations for the five methods from the random generation of code words. Tables 1–5 show the number n of nodes in trees, the number t of trees generated, and the average numbers of different operations *per node* (i.e. we sum up the corresponding numbers from all the t trees generated and divide the sums by nt). As a result, the numbers from different test runs are comparable even with differing values of n and t. When n is large enough, these values can be used to estimate and compare the asymptotic performance of the algorithms.

3.2. Constructing the trees

In this section we give the observed numbers of the operations per node from the construction of the resulting binary trees from the code words generated in the previous section. The data shown in Tables 6–9 is collected and presented as in the case of Tables 1–5. Notice that Arnold and Sleep's and Atkinson and Sack's algorithms use the same coding method.

Memory space for tree nodes is allocated from a static pool of n nodes. In Tables 6–9 there is no column for the memory space allocation operation, since these are executed once per node.

4. TOTAL COSTS

So far, we have counted the numbers of different operations the algorithms execute. These numbers are quite different among the algorithms. In order to compare the algorithms, we have to decide weights for different types of operations. The weights used in this paper are shown in Table 10. Naturally, reasonable sets of weights vary from one environment to another.

When the weights are fixed, it is straightforward to give the total costs of the algorithms. Just multiply every number in Tables 1–9 by its corresponding weight and add up the numbers in each row. The final results obtained in this manner are given in Tables 11–15. The column *trees* includes the weighted cost of 36 of allocating memory space for one tree node.

The algorithms include many loops with a fixed number of iterations. In principle, the cost of the control structure of this kind of loop can be avoided, if the body of the loop is duplicated in the program code the known number of times. As an aside, we recorded the amount by which ignoring these costs reduces the total costs. The reduction in the weighted costs of the generation of code words was of the order of 15% for Atkinson and Sack, and Korsh, 5% for Martin and Orr, and Remy, and zero for Arnold and Sleep. However, these results did not affect the ranking order of the algorithms.

TABLE 11. The weighted costs of Arnold and Sleep's algorithm.

n	t	Codes	Trees	Total
4	10,000	115.59	98.00	213.59
10	10,000	142.87	98.90	241.77
100	10,000	167.68	99.44	267.12
1000	5000	170.66	99.49	270.16
10,000	1000	170.97	99.50	270.47
100,000	100	171.00	99.50	270.50

TABLE 12. The weighted costs of Atkinson and Sack's algorithm.

n	t	Codes	Trees	Total
4	10,000	268.49	98.00	366.49
10	10,000	257.86	98.90	356.76
100	10,000	246.90	99.44	346.34
1000	5000	243.81	99.49	343.30
10,000	1000	242.91	99.50	342.41
100,000	100	242.64	99.50	342.14

TABLE 13. The weighted costs of Korsh's algorithm.

n	t	Codes	Trees	Total
4	10,000	281.97	86.50	368.47
10	10,000	295.61	87.40	383.01
100	10,000	303.07	87.94	391.01
1000	5000	303.45	87.99	391.45
10,000	1000	302.00	88.00	390.00
100,000	100	303.40	88.00	391.40

TABLE 14. The weighted costs of Martin and Orr's algorithm.

n	t	Codes	Trees	Total
4	10,000	104.90	72.75	177.65
10	10,000	130.97	76.80	207.77
100	10,000	150.57	79.23	229.80
1000	5000	152.75	79.47	232.23
10,000	1000	152.98	79.50	232.47
100,000	100	153.00	79.50	232.50

TABLE 15. The weighted costs of Rémy's algorithm.

n	t	Codes	Trees	Total
4	10,000	195.72	146.88	342.60
10	10,000	194.14	140.35	334.49
100	10,000	194.25	136.44	330.68
1000	5000	194.45	136.04	330.50
10,000	1000	194.49	136.00	330.50
100,000	100	194.50	136.00	330.50

5. CONCLUSIONS

The level of convergence and monotonicity of the total costs in Tables 11–15 imply that our tests have been sufficiently long to estimate the relative constant factors of the time complexities (possibly excluding Korsh's algorithm, whose total cost did not behave monotonically as n increased). We can conclude that exact asymptotic costs are found at least for Arnold and Sleep's, Martin and Orr's and Rémy's algorithms. The standard deviations of the total costs of the algorithms (again excluding Korsh's algorithm) are very small with $n \geq 1000$.

First we discuss the numbers of different operation types in the algorithms. In most environments a call of a random number generator is far more expensive than the other operation types recorded. In Table 10 we have weighted it 32 times more expensive than the basic operations LOA and STO. A natural measure of the efficiency of the algorithms is then the number of calls of the random number generator per node. Arnold and Sleep's and Rémy's algorithms use two calls per node, while one call is sufficient for the other three algorithms.

With respect to array references, Arnold and Sleep's and Martin and Orr's algorithms are clearly the best ones. They need only two array references per node while the other algorithms need at least 10 array references. Moreover, the two algorithms with the lowest numbers of array references also make their references in consecutive array positions.

Tables 11–15 suggest the following overall order of the algorithms:

1. Martin and Orr;
2. Arnold and Sleep;
3. Rémy;
4. Atkinson and Sack;
5. Korsh.

However, as we let n increase from 10^5 up to 10^6, run-time errors occurred with Arnold and Sleep's and Martin and Orr's algorithms because of integer overflow. These errors do not affect the results reported in this paper. Moreover, they could be avoided if we followed the algorithms less literally, i.e. rearranged the order of evaluation of arithmetic expressions with the help of floating-point calculations. This is achieved with only a few additional operations. The extra cost incurred is not crucial since on modern computers floating-point arithmetic, built into hardware, can be quite close to integer arithmetic in speed.

Execution times for the generation of code words (measured in the VAX 7000-830 environment) with average size trees ($n = 10^4$) gave the order

1. Martin and Orr;
2. Atkinson and Sack;
3. Korsh;
4. Arnold and Sleep;
5. Rémy.

In summary, Martin and Orr's and Arnold and Sleep's algorithms performed best in terms of weighted costs, but this is offset by the fact that they use larger integers than the other algorithms (see also [8]). Because the algorithms do not differ very much in performance, their overall order, however, may vary depending on the weights, i.e. on the implementation of the operations. In any case, weighted costs based on different operation weights can easily be recalculated from Tables 1–9.

ACKNOWLEDGEMENT

This work of the second author was supported by the Academy of Finland (project 35025).

REFERENCES

[1] Knuth, D. E. (1997) *The Art of Computer Programming. Vol. 1, Fundamental Algorithms* (3rd edn). Addison-Wesley, Reading, MA.

[2] Martin, H. W. and Orr, B. O. (1989) A random binary tree generator. In *Proc. ACM 17th Annual Computer Science Conf.*, Louisville, KY, 21–23 February, pp. 33–38. ACM Press, New York.

[3] Arnold, D. B. and Sleep, M. R. (1980) Uniform random generation of balanced parenthesis strings. *ACM Trans. Program. Lang. Syst.*, **2**, 122–128.

[4] Atkinson, M. D. and Sack, J.-R. (1992) Generating binary trees at random. *Inf. Process. Lett.*, **41**, 21–23.

[5] Korsh, J. F. (1993) Counting and randomly generating binary trees. *Inf. Process. Lett.*, **45**, 291–294.

[6] Rémy, J. L. (1985) Un procédé itératif de dénombrement d'arbres binaires et son application à leur génération aléatoire. *RAIRO Inform. Théor.*, **19**, 179–195.

[7] Siltaneva, J. (2000) *Random Generation of Binary Trees*. Master's Thesis, Department of Computer and Information Sciences, University of Tampere, Finland. (In Finnish). Also available at http://www.cs.uta.fi/research/theses/masters/.

[8] Mäkinen, E. (1999) Generating random binary trees—a survey. *Inf. Sciences*, **115**, 123–136.

[9] Johnsen, B. (1991) Generating binary trees with uniform probability. *BIT*, **31**, 15–31.

[10] Pallo, J. M. (1994) On the listing and random generation of hybrid binary trees. *Int. J. Comput. Math.*, **50**, 135–145.

[11] Feller, W. (1968) *Introduction to Probability and Its Applications* (3rd edn). Vol. 1, Wiley, New York.

[12] Reingold, E. M., Nievergelt, J. and Deo, N. (1977) *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Englewood Cliffs, NJ.

[13] Alonso, L. and Schott, R. (1995) *Random Generation of Trees*. Kluwer, Boston, MA.

[14] Mäkinen, E. and Siltaneva, J. (2001) A note on Rémy's algorithm for generating random binary trees. *Missouri J. Math. Sci.*. To appear. Also available at http://www.cs.uta.fi/reports/r2000.html.

[15] Park, S. K. and Miller, K. W. (1988) Random number generators: good ones are hard to find. *Commun. ACM*, **31**, 1192–1201.