Qatar Univ. Sci. J. (2000), 20 : 7-15

A RANDOM BINARY TREES GENERATION METHOD

By
Saud M.A. Maghrabi
Scientific Research Institute
Umm Al-Qura University
P.O. Box 6648
Makkah, Saudi Arabia.

طريقة لإنشاء الأشجار الثنائية بشكل عشوائي

سعود محمد عبد الله مغربي
قسم العلوم الرياضية - كلية العلوم التطبيقية - جامعة أم القرى
ص . ب : ٦٦٤٨ - مكة المكرمة - المملكة العربية السعودية

هناك نوعان من التوزيعات العشوائية والتي تستخدم على نطاق واسع، هما: توزيع شجرة البحث الثنائية، والتوزيع المتماثل. بتحليل أداة الخوارزميات (الطرق) التي حلت مشكلة إنشاء الأشجار الثنائية بشكل عشوائي، وجد أنه من المحتمل أن يُقَوَّمَ إنجاز الخوارزمية باستعمال متوسط حالة إنجازها. يعطي متوسط حالة إنجاز الخوارزمية في الغالب انعكاساً مفيداً لمدى ملاءمة الخوارزمية أكثر مما يعطيه أسوأ حالات إنجازها، أسوأ حالات وقت التنفيذ لأي خوارزمية يمثل مدى التعقيد في هذا الوقت، ولكن متوسط الحالة لوقت التنفيذ يمثل تعقيداً متعدد الحدود في وقت. من مثل هذه الحالات، إذا رغب أحدنا في أن يُقوِّم أداء خوارزمية على مجموعة مثالية من الأشجار المنتظمة من حيث الخواص المستعملة بإنشاء الأشجار بشكل عشوائي، ومن أمثلة هذه الخواص خاصية العمق، وكذلك من حيث السمات المؤثرة في وقت تنفيذ الخوارزمية. إن الهدف من هذا البحث هو تصميم وتطبيق خوارزمية لإنشاء الأشجار الثنائية بشكل عشوائي بحيث يعتمد تَقْوِيم أداء هذه الخوارزمية على متوسط حالة وقت التنفيذ. وتختلف هذه الخوارزمية عن التوزيعين المذكورين أعلاه: التوزيع المتماثل، وتوزيع شجرة البحث الثنائي، ولقد تمت دراسة وتحليل خوارزمية هذا البحث.

KEY WORDS : Binary Trees Generation, Data Structures, Binary Trees Search and Analysis of Algorithms.

ABSTRACT

There are two widely used random distributions on binary trees, namely the binary search tree distribution and the uniform distribution. By analyzing the performance of algorithms that manipulate binary trees generated at random, it is possible to assess the average case performance of the algorithm. This average case performance is often a more useful reflection of the algorithm's suitability than the worst-case performance. An algorithm may have a worst-case run time that is of exponential time complexity, but an average-case run time that is of polynomial time complexity. In such cases, one wishes to assess the performance of an algorithm on a 'typical' range of tree structures and thus relate properties of randomly generated trees, such as depth, to aspects affecting the algorithm's run-time. The purpose of this paper is to design and implement a random binary trees generation algorithm that considers the average case performance. The algorithm is differ from both the uniform distribution and the binary search tree distribution. Analysis of the behaviour of the algorithm is given.

1. INTRODUCTION

Two commonly used random distributions of binary trees are the uniform distribution, escrimed in [1, 4, 6, 7], and the conceptually simpler binary search tree distribution as described in [2, 3, 5]. When we used the uniform distribution to generate binary trees, every *2n-1* node tree as equally likely to be generated, whereas the binary searc tree distribution is a biased distribution. The uniform distribution is biased towards producing trees that are extremely unbalanced in terms of the number of leaves in the left and right sub-trees. For the uniform distribution, the probability that a randomly generated tree $T_n = L \circ R$, where $\circ$ denotes a single node of a tree; $L \circ R$, means that a tree with root node$\circ$, left sub-tree L, and right sub-tree R, is as follows.

$$\text{Probability } [|L| = i] = \frac{|T_i| \cdot |T_{n-i}|}{|T_n|}$$

With referenc to [4], it can be shown that

$$T_n = \frac{1}{4n-2}\binom{2n}{n} \approx \frac{4^n}{n^{\frac{3}{2}}}$$

$$\text{Probability } [|L| = i] \approx \frac{|T_i|}{4^i} \rightarrow 0 \text{ as } i \rightarrow \infty$$

For the binary search tree distribution, the probability that a randomly generated tree having a particular left sub-tree size is constant with respect to the number of leaves in the tree.

The method is this paper is differ from both the uniform distribution and the binary search tree distribution is significant ways. The algorithm of thispaper is nonuniform; i.e. node ever *2n-1* is equally like to be generated, unlike the algorithm of references [1, 4, 6, 7]. Also the algorithm behaves differently from the binary search tree distribution. The binary search tree distribution is generated as the output of a recursive random algorithm, whereas thisalgorithm is iterative. We will prove that the method of this paper has the same distribution as the binary search tree distribution.

The aim of the method of this paper is to analyze these differences byconsidering the average case behavior of the following standard measures on trees:

- **Depth :** the depth of a tree is the length of the longest path from a leaf to the root node.

- **Width :** the kth level of a binary tree is the set of nodes at depth k from the root node. The width of a binary tree is the maximum number of nodes on a single level. The widthis at lest 2 and at most $n/2$ for an n node binary tree.

- **Number of twigs :** a twig is a node in the tree, both of whose predecessor nodes are leaves.

- **Sub-tree sizes :** the probability that a randomly generated tree has a left sub-tree size of *2k-1*, thus the distribution of sub-tree sizes for each value of k between 1 and *n-1*. This probability distribution of sub-tree sizes has been widely studied using the uniform distribution and the binary search tree distribution.

2. MODULARIZATION

An algorithm for random binary tree generation is supplied. Let *range* = {1,2,3,..., n}. The initial steps of the algorithm are:

```
while next_node < 2n
  {
    select 2 random nodes from range
    remove these two nodes from range
    add next_node to range
    update the set of edges. E
    next_node: = next_node + 1
  }
```

This is re-written to deal with the problem of selecting the same node twice, as

```
while next_node < 2n
  {
    select a node randomly from range
    remove this node from range
    select another node from range
    remove this node from range by adding next_node
    update the set of edges. E
    next_node: = next-node + 1
  }
```

This is then expanded as follows:

while next_node < 2n

{

1. (a) select a position in range to take a node from
   (b) assign the value at this position to *u*

2. Move the value stored at the position of the number of choices into this position

3. (a) select a position in *range* to take a node from using *n-1* choices
   (b) Assign th value stored at this position to *v*

4. Replace the value at this position with *next_node*

5. Update the set of edges, *E*

6. *next_node := node + 1*

}

The final form of the random binary tree generation algorithm is as follows:

Input: n ∈ N.

Output: Random n leaf, 2n-1 node binary tree T(V,E).

Method:

Range: = {1 2 3,..., n};

E: = 0;

V: = range;

next-node: = n + 1;

while next_node < 2n

{

(u,v): = Random pair of distinct nodes from Fringe;

range: = range - {u,v};

range: = range ∪ {next_node};

V: = V ∪ {next_node};

E: = E ∪ {{U,next_node}, {v,next_node}};

next_node: = next_node+1;

}

return T(V,E);

The algorithm provides enough information for a top-down program design and implementation to be performed. A procedure called *random_tree* has been written toimplement this algorithm.

3. DATA STRUCTURES

3.1 *Range* Structure

The data structure *range* represents a list of integers, which take the values from *1,2,..., n.* This structure is represented by a one-dimensional array of integers. By considering the operations to be performed on this structure, it is noticed that the algorithm requires the repeated selection of two random elements from this set of numbers to be 'connected' to the value *next_node.* Once these two elements had been selected, they were never to be chosen again in order that a node in the three and exactly zeroor two successors. This fact lead one to choose a linked list structure rather than an array, as elements that are selected could be deleted from the list and new ones added. The drawback with using a linked list to represent this structure would be that it would make the execution of the program slightly more difficult to follow. It was therefore decided to use the one-dimensional array to represent *range*, and in order to overcome the possibility of reselection of nodes, the order in which the nodes in the list were stored could be manipulated.

For example, supposing we start with an input of *n=5* to produce a 5 leaf tree; at the start of the program's execution, the structure *range* would look something like this:

1	2	3	4	5

Once the first element had been selected, and assigned to the variable *u*, this element is not to be selected again. The element is replaced by the element at the position of the number of choices and the next selectio is made from the *n-1* elements in the list. If the element number three in the array was chosen first, then this wouldbe replaced by the element number 5, and selection would then continue using only the first four numbers. In the example above, the array would now he of the form:

1	2	5	4	5

and the next selection would be taken from the elements numbered 1, 2, 5 and 4. Similarly, when the second selection has been made, the *next_node* element, which is then added to this list, is placed in the position where the last selection was

taken from. For example, selecting element number 2 would cause element number 6 to replace it giving:

1	6	5	4	5

This would then continue throughout the execution of the loop, each time making a choice from one fewer elements until the constraint oncontinuingthe loop no longer holds.

3.2 The Edge Set Structure

The data structure E in the algorithm represents the set of internal nodes of the tree and holds information to discern which of these nodes are connected to which other nodes in the tree, i.e. a set of edges. Each element in the set therefore, has associated with two integers. The edge set is represented as a two dimensional array. The advantage set is represented as a two dimensional array. The advantage of choosing this data structure is that one of the indices of the array could be indexed by the values $n+l,\ldots, 2n-l$ representing the node numbers for the non-leaf nodes of the trees of the tree. Also using this data structure makes it easier to calculate the number of twigs in the tree. The twigs in the tree are the nodes in the range $n+l, \ldots, 2n-l,$ which are connnected to tne nodes numbered less than $n+l$.

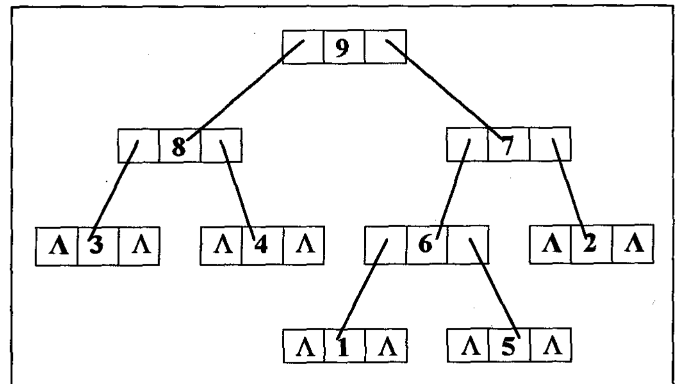
For example, using an input of $n=5$, the data structure E could look something like:

	6	7	8	9
1	1	6	3	8
2	5	2	4	7

nodes numbered 6 and 8 are twigs as they are connected to nodes 1 and 5 and 3 and 4 respectively, all of which are leaf nodes as their node numbers are less than or equal to n.

3.3 The Tree Structure

The structure T represents the tree itself. The program requires a data structure that can be used recursively to form the tree, due to the recursive nature of this definition of a binarytree. The choice of a linked list in this situation to represent the tree is fairly obvious. Each element in the list wouldcontain an integer field to represent the node number and it would alsocontain two pointers to two sub-trees which are themselves binary trees. The pointers would contain NULL values if the node was a leaf of the tree as it would have no successors. In the previous example, the tree structure would look as follows:



Using this implementation wouldmake the calculations of width; dept etc. easier as the tree could be parsed recursively to calculate the relevant data.

4. CALCULATING MEASURES

Once the code had been implemented to generate randomly, a binary tree using the supplied algorithm, it was then necessary to include procedures/functions that would investigate this tree structure to calculate the standard measures described earlier, i.e. width, depth etc.

4.1 Twigs

In order to calculate the number of twigs in the tree, the tree structure itself is not necessary. The data structure, E, is chosen to make this task less complicated. The structure is indexed in one-dimension by the node numbers of all the internal nodes of the tree. If both of the values stored at one of these indices is less than or equal to the number of leaves-input, then the index value must be the node number of a twig.

The function *num_twing* takes the set of edges, E as its parameter and returns the number of twigs as a result. The function searches the two dimensional array through index 1 and the number of twigs is incremented if, for one of these indices, both entries have a number less than the first index position of the array i.e. less than $n+1$.

4.2 Width and Depth

In order to calculate the width and depth of the tree, it was decided that a list of the widths at each of the levels was to be constructed. This makes it not only possible to calculate the greatest width of the tree, but also as a consequence, gives the depth of the tree as well. The procedure called *fill_set* performs this task. It takes three parameters: a one-dimensional array to hold the list of widths at each level, the tree structure, T, and the current level.

The initial call is made with the empty list, *T*, and the level number 1. Each time a call ismade to this procedure, the width at the current level is incremented by 1. If teh left sub-tree pointer does not contain a NULL value, then the procedure calls itself passing the list of widths, *T* and the next level as parameters. As each node is visited only once, on completion of the procedure, thelist contains the width at eachlevel of the tree. A simple function to retrieve the maximum value in this list, *grt_maxim*, reveals the witdh of the tree.

Once this list of widths is created, it is easy to use this to calculate maximum depth. The function called *get_depth* calculates the value for the depth of the tree bycounting the number of entries in the list, which contain a non-zero value.

4.3 Sub-tree Sizes

A function, *get_size*, was also added to calculate the size, in nodes, of any given binary tree. The function takes a tree structure as a parameter and recursively traverses the tree adding one to its size when encountering a non-NULL sub-tree pointer.

All the measures calculated for a particular tree are stored in a record structure and this record structure is returned from the procedure *random_tree*.

5. PROCEDURE FOR TRAVERSES THE TREE

A procedure called *output_tree* is implemented to recursively traversing the tree structure and indicating which nodes of the tree were connected to which other nodes in the tree. The procedure takes as a parameter the tree structure; *T*. The procedure traverses the tree structure recursively outputting the relevant data for each node of the tree. The first node that is visited is the root node of the tree. Data is output using the left and right sub-tree node numbers. The base case is reached when a pointer to the left sub-tree contains a NULL value. Once data have been noted for the node in question, the procedure calls itself using the left sub-tree and then the right sub-tree. As the trees were created at random, this provided a good source of test data for the program implementation. The output from this procedure is as follows:

Node number 9 is connected to nodes 8 and 7

Node number 8 is connected to nodes 3 and 4

Node number 7 is connected to nodes 6 and 2

Node number 6 is connected to nodes 1 and 5

6. THE MAIN PROCEDURE OF THE PROGRAM

The program repeatedly generates random binary trees of the given size. It generates a tree of this size for each of the number of tests. In the case of width, depth and number of twigs, the relevant data are calculated and the values are added together for each of the measures taken. Dividing this value by the number of tests given an averge measure of this size of tree. The program performs these calculations repeatedly for each size of the tree in the given range.

In the case of sub-tree sizes, for each test, the size of the left sub-tree is calculated and the number of times each instance occurs is stored in an array. The probability that a particular sub-tree size occurs is simply the number of times this instance occurs divided by the number of tests.

The program diverts this output to several files. The output is constructed in the form of two columns inorder that the information canbe plotted graphically.

In the case of width, depth and number of twigs, each output file consists of two columns containing the number of nodes for each size of tree in the given range and the average value of the measure taken. For sub-tree sizes, the columns are the *ranges* of possible sub-tree sizes for this input and the probability that each of these instances occurred. Some data has been entered to the program. For each test performed using this random binary tree generation algorithm, graphical representations of this data are produced in Figures 1 - 4.

7. ANALYZING RESULTS

From the graphical representation of the dta, shown in figures 1-6, it can be seen that the number of twigs for a tree of given input size is proportional to the number of nodes in the tree. The width and depth of a tree of given input size seems to increase with some logarithmic function of the number of nodes. This shown more clearly in the case of the depth as it is with respect to the case of width.

The most interesting by far of all the output data is the distribution of sub-tree sizes for a given tree size. On average, the probability of a particular instance of sub-tree size occurring is constant with respect to the number of leaves in the tree. This experimental evidence correlates directly with the behavior of the binary search tree distribution. For the binary search tree distribution, the probability that a randomly generated binary tree having a particular left sub-tree size is constant with respect to the number of leaves in the tree. This pat-

tern of behavior has beets shown to be exhibited by this random binary tree-generating algorithm. For the binary search tree distribution, this probability is not only constant with respect to the number of leaves in the tree; it is of a particular value with respect to the number of leaves in the tree. This is also the case for the algorithm of this paper.

Experimental evidence is now deemed strong enough to construct a mathematical proof that this random binary tree generation algorithm is equivalent to the binary search tree distribution.

8. MATHEMATICAL PROOF OF EVUIVALENCE BETWEEN THIS ALGORITHM AND THE BINARY SEARCH TREE DISTRIBUTION.

8.1 Binary Search Tree Distribution Method

Produce a random n leaf tree, T_n ($L \circ R$), i.e. tree has a root node, $\circ$, a left sub-tree L, a right sub-tree R.

$$\textit{Probability [L has k Leaves]} = \frac{1}{n-1} \; (\forall\, 1 \leq k \leq n-1)$$

This is now a definition of the binary search tree distribution.

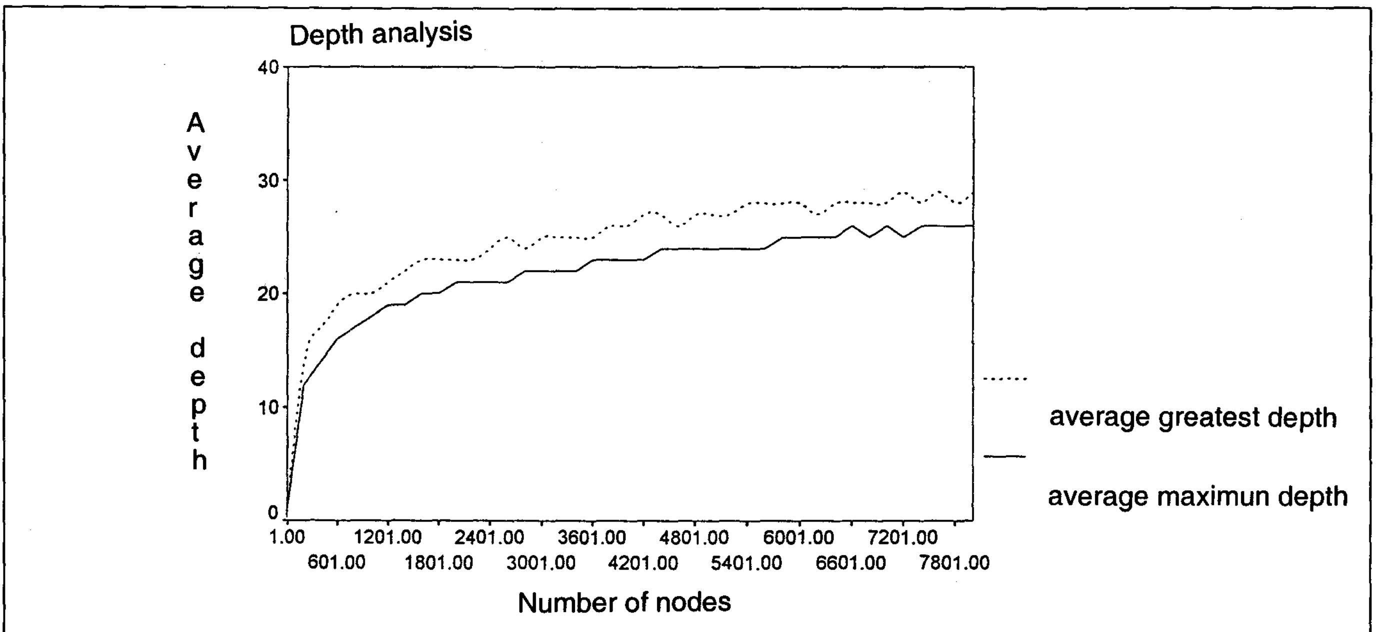


Figure 1 : Graph representing the depth analysis, showing the average maximum depth and the average greatest depth.

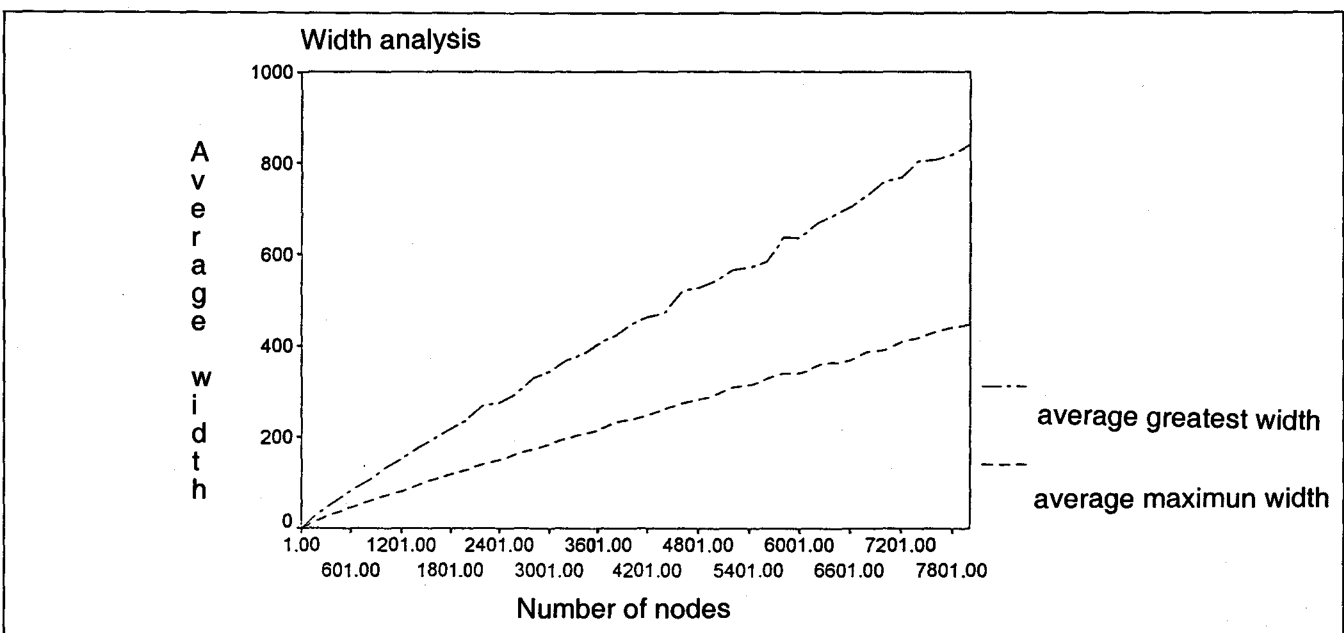


Figure 2 : Graph representing the width analysis, showing the average maximum width and the average greatest width.

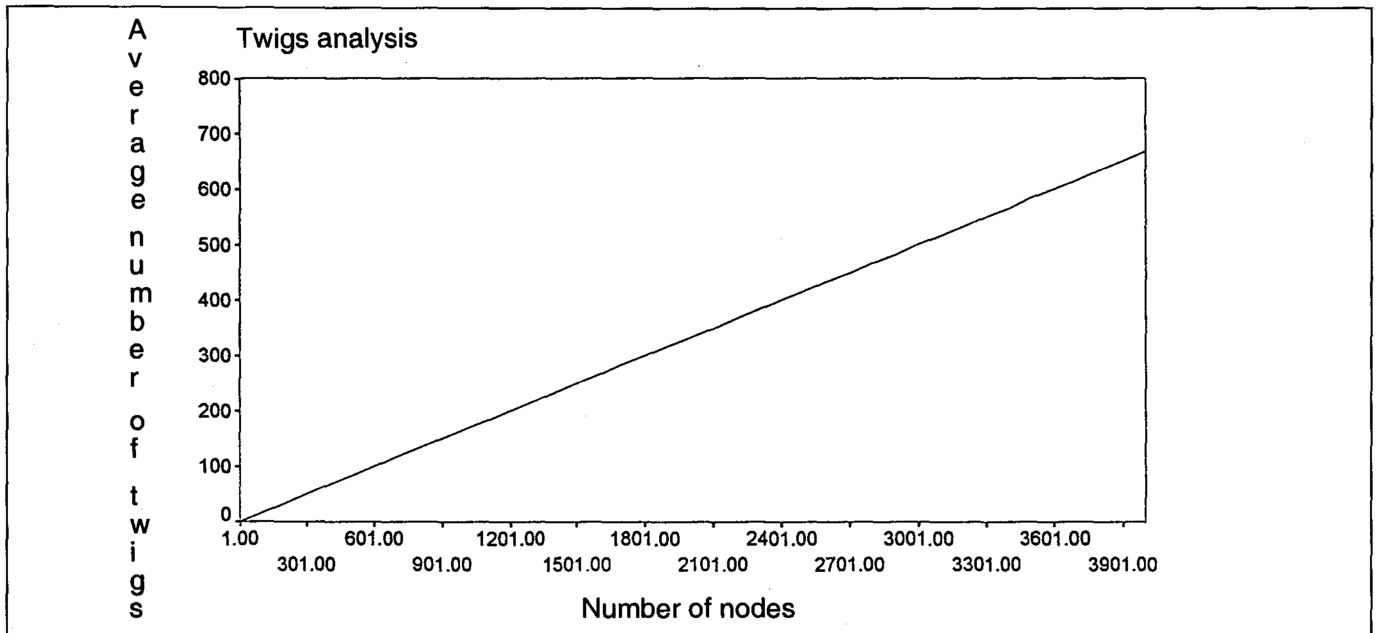


Figure 3 : Graph representing the twigs analysis, showing the average number of twigs for 4000 nodes.

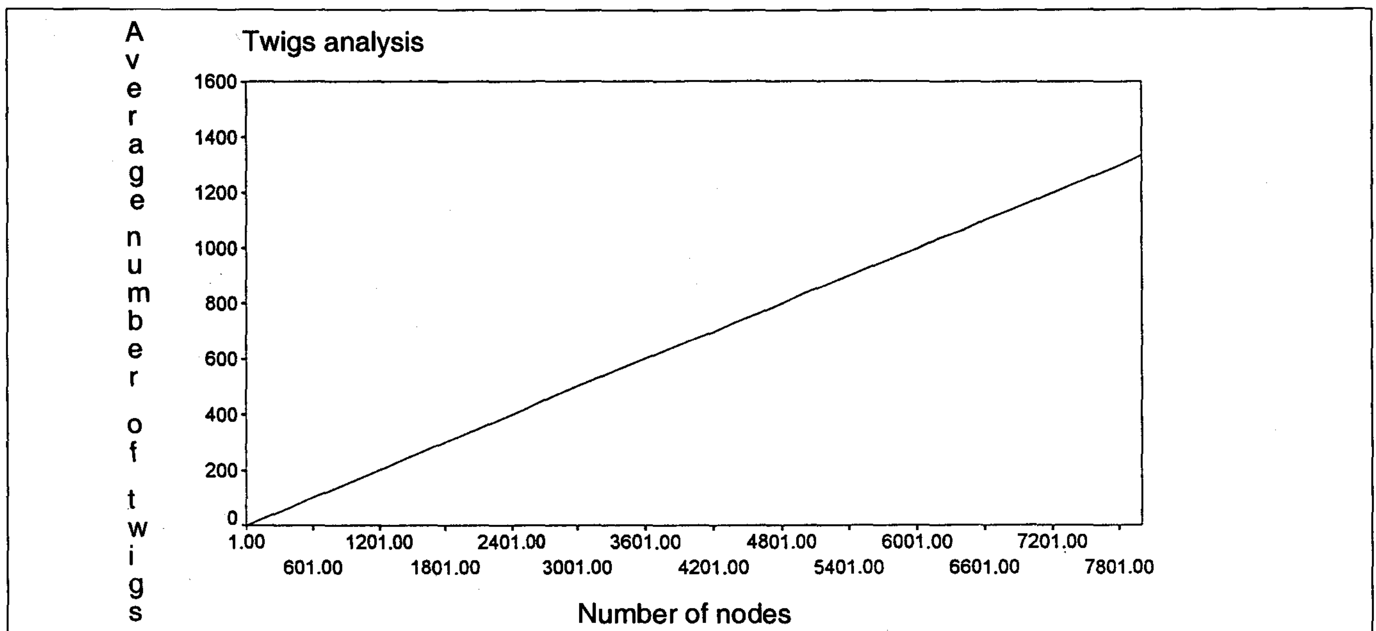


Figure 4 : Graph representing the twigs analysis, showing the average number of twigs for 8000 nodes.

8.2 Range Method

Let S(n) be defined as the number of different 'structures' that can arise, starting witn *n* leaves.

Level 0: *n elements in* ***range*** *gives* $\binom{n}{2}$ *ways of starting.*

Level 1 : *n-1 elements in* ***range*** *gives* $\binom{n-1}{2}$ *ways of continuing*

Level 2 : *n-2 elements in* ***range*** *gives* $\binom{n-2}{2}$ *ways of continuing*

. . . . .

. . . . .

Level k : *n-k elements in* ***range*** *gives* $\binom{n-k}{2}$ *ways of continuing*

. . . . .

. . . . .

Level n-2 : *2 elements in* ***range*** *gives* $\binom{2}{2}$ *ways of completing.*

Therefore:

$$S(n) = \prod_{i=0}^{n-2} \binom{n-i}{2}$$

$$= \prod_{i=0}^{n-2} \left(\frac{(n-i) \cdot (n-i-1)}{2} \right)$$

$$= \frac{\prod_{i=0}^{n-2} (n-i) \cdot \prod_{i=0}^{n-2} (n-i-1)}{2^{n-1}}$$

$$= \frac{\prod_{i=2} (n-i) \cdot \prod_{i=0} (i)}{2^{n-1}}$$

$$= \frac{\prod_{i=2}^{n} (i) \cdot \prod_{i=1}^{n-1} (i)}{2^{n-1}}$$

$$S(n) = \frac{n!\,(n-1)!}{2^{n-1}}$$

Let R (n, k) be defined as the number of different 'structures' that correspond to binar tree, $T_n = (L \circ R)$, in which *L* has *k* leaves and *R* has *n - k* leaves.

$$\textit{Probability [L has k leaves]} = \frac{R\,(n,k)}{S\,(n)} \quad (\forall 1 \leq k \leq n-1)$$

$$R\,(n,k) = \frac{1}{2} \left[\begin{matrix} \textit{number of ways} \\ \textit{of choosing a} \\ \textit{subtree with L} \\ \textit{having a partiular} \\ \textit{set of k leaves} \end{matrix} \cdot \begin{matrix} \textit{number of} \\ \textit{structures} \\ \textit{with this} \\ \textit{set of k} \\ \textit{leaves} \end{matrix} \cdot \begin{matrix} \textit{number of ways} \\ \textit{of choosing a} \\ \textit{sub — tree for R} \\ \textit{given L has k} \\ \textit{leaves} \end{matrix} \cdot \begin{matrix} \textit{number of} \\ \textit{structures} \\ \textit{with this} \\ \textit{set of n - k} \\ \textit{leaves} \end{matrix} \right]$$

Where, the number of ways of choosing *L* having a particular set of k leaves is $\binom{n}{k}$, the number of structures with this set of *k* leaves is *S(k)*, the number of ways of choosing a sub-tree for R given L has k leaves is $\binom{n-2}{k-1}$, the number of structures for *R* given *L* has k leaves is *S(n-k)*. Only half the number is required as 'mirror images' are created and therefore counted twice, so:

$$R(n,k) = \frac{1}{2} \left[\binom{n}{k} \cdot S(k) \cdot \binom{n-2}{k-1} \cdot S(n-k) \right]$$

8.3 Proof of Equivalence

The algorithm of this paper should generate random binary trees in the same way as the binary search tree distribution, if,

$$\frac{R(n,k)}{S(n)} = \frac{1}{n-1} (\forall\ 1 \leq k \leq n-1)$$

Proof :

$$\frac{R(n,k)}{S(n)} = \frac{\frac{1}{2}\left[\binom{n}{k} \cdot S(k) \cdot \binom{n-2}{k-1} \cdot S(n-k)\right]}{S(n)}$$

$$= \frac{\frac{1}{2}\left[\left(\frac{n!}{k!(n-k)!}\right) \cdot \left(\frac{k!(k-1)!}{2^{k-1}}\right) \cdot \left(\frac{(n-2)!}{(k-1)!\,(n-k-1)!}\right) \cdot \left(\frac{(n-k)!\,(n-k-1)!}{2^{n-k-1}}\right)\right]}{\frac{n!(n-1)!}{2^{n-1}}}$$

$$= \frac{\frac{1}{2}\left[\frac{n!(n-2)!}{2^{k-1} \cdot 2^{n-k-1}}\right]}{\frac{n!(n-1)!}{2^{n-1}}}$$

$$= \frac{(n-2)! \cdot 2^{n-2}}{2^{k-1} \cdot 2^{n-k-1} \cdot (n-1)!}$$

$$= \frac{1}{n-1}$$

Therefore :

$$\frac{R\,(n,k)}{(S\,(n)} = \frac{1}{n-1}$$

From this mathematical proof, it can be seen that the algorithm of this paper generates random binary trees in the same way as the binary search tree distribution, therefore, the method of this paper is equivalent to the binary search tree distribution.

9. CONCLUSION

In this paper, a method for generating random binary trees has been described. The method is differ from both the uniform distribution and the binary search tree distribution. These differences have been analyzed by considering the average cases of the following: the depth of the tree, the width of the tree, the number of the nodes in the tree, and the sub-tree sizes. The experimental evidence, along with the mathematical proof, is conclusive to say that for the case of full binary trees, the random binary tree generation algorithm of this paper and the binary search tree distribution are in fact the same distribution.

REFERENCES :

[1] **Atkinson, M. D., and Sack, J. R., 1992,** "Generating Binary Trees at Random", Information Processing Letters, 41, 21-23.

[2] **Devroye, L., and Reed, B., 1996.** "On the variance of the Height of Random Binary Search Trees,", Siam Journal on Computing, 24, 1157-1162.

[3] **Devroye, L., and Robson, J. M., 1997.** "On the Generation ofRandom Binary Search Trees", SIAM Journal on Computing, 28, 1141-1162.

[4] **Dunne, P. E., Gittings, C. j., and Leng, P. H., 1995.** "Multiprocessor Simulation Strategies with Optimal Speed-up". Information processing Letters, 54, 23-33.

[5] **Kim, S. K., 1997.** "Logarithmic width, Linear Area Upward Drawing of AVL Trees", Information Processing Letters, 64, 303-307.

[6] **Korsh, J. F., 1993.** "Counting and Randomly Generating Binary Tree', Information Processing Letters, 45, 291-294.

[7] **Martin, H. W. and Orr, B. J., 1990.** "A Random Binary Tree Generator". In: Computing Trends in the 1990's, ACM Seventeenth Computer Science Conf., Louisville, KY (ACM, New York, 1990), 33-38.