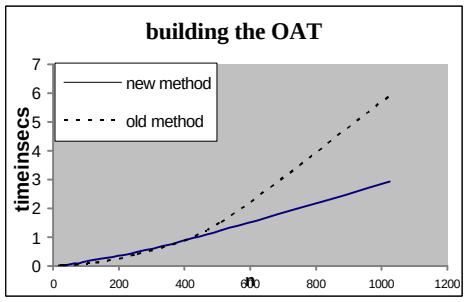
Building Optimal Alphabetic Trees Recursively

Ahmed A. Belal[1], Mohamed S. Selim[2], Shymaa M. Arafat[3]

Department of Computer Science & Automatic Control, Faculty of Engineering, Alexandria University, Egypt.

[2]Department of Computer Science & Automatic Control, Faculty of Engineering, Alexandria University, Egypt.
Department of Computer Science, Faculty of Computer&Information Sciences, Ain Shams University, Egypt.

Abstract. Optimal alphabetic binary trees (OATs) have a wide variety of applications in computer science and information systems. Algorithms for building such trees in O(n log n) time and O(n) space do exist. In this paper, we introduce a new simpler method for solving the same problem. An earlier algorithm, for merging two optimal alphabetic binary trees into one optimal alphabetic binary tree in linear time, is used to build the OAT recursively in a divide-and-conquer manner. Although the resulting algorithm has the same order complexity as previously known algorithms, the new method considerably outperforms the other implementations . The analysis in the paper justifies the improvement. The figure below compares the new implementation with the Hu-Tucker O(nlogn) implementation.



1 Introduction

Binary trees have received a considerable attention in computer science research. Some tree building algorithms assume an equally weighted node tree, in which case an optimal tree means a balanced one. However, when the nodes of the tree, both internal and external, have different access frequencies, it becomes natural to assign a different weight for each node. In this case, an optimal tree is the one with minimal cost, where the cost is the summation of the products of the node weight by the node level over all nodes. Optimal binary trees of this kind can be built in $O(n^2)$ time complexity [11]. For the simpler case where only external nodes have weights, the optimal tree can be found in time O(n log n) . An example is the Huffman tree [9] which is widely used in coding and information theory. A more constrained kind is the binary alphabetic tree, also called an insertion tree, where nodes must appear in their original order in the final tree [6,7].

In the last few years more results on optimal binary alphabetic trees (OAT) were reported in the literature. The equivalence of the OAT to optimal binary search trees was reported in [2]. The use of the 1-dimensional O(n log n) algorithm for 2-dimensional information retrieval was considered in [1,3]. The search for sub O(nlogn) algorithms for the OAT problem was also recently reported [10,12]. It was recently shown in [8] that linear time algorithms for building optimal binary alphabetic trees are possible for some special weight sequences.

An O(n) time algorithm to merge two OATs with n_1,n_2 nodes into an $(n=n_1+n_2)$ nodes OAT is presented in [4]. In this paper, we use this algorithm to build an OAT for a given weight sequence in a divide-and-conqueer manner.

Section 2 is a preface that gives a brief description of the Hu-Tucker algorithm. Then Section3 gives an outline of the linear time merging algorithm along with an explanatory example. Section4 shows how the merging algorithm could be applied successively. Then, section5 shows how to build the OAT recursively using successive merging. Finally section6 concludes the paper.

2 Preface-The Hu-Tucker Algorithm

The algorithm proceeds in three phases

Phase1 : Combination

This is where most of the work is done. Every node before combining is a square node also called external node. If we let $\mathbf{q_i}$ denotes the weight of the node or the node itself then when two square nodes $\mathbf{q_i},\mathbf{q_j}$ combine they form a circular node also called internal node with weight $\mathbf{q_i+q_j}$ occupying the position of the left child.

Due to the alphabetic constraint, two nodes can only combine if they form a compatible pair. Two nodes in a

sequence form a compatible pair if they are adjacent in the sequence or if all nodes between them are internal nodes. Among all compatible pairs in a weight sequence the one having the minimum weight is called the minimum compatible pair.
To break ties, the Hu-Tucker algorithm uses the convention that the node on the left has a smaller weight.
A pair of nodes $(\mathbf{q_j},\mathbf{q_k})$ is a **Local Minimum Compatible Pair** (*LMCP*) if

$\mathbf{q_i} > \mathbf{q_k}$ for all nodes q_i compatible with q_j

$\mathbf{q_j} \leq \mathbf{q_l}$ for all nodes q_l compatible with q_k

The first phase of the algorithm keeps forming *LMCP*s until a tree is formed.
Phase 2 : Assigning Levels
Uses the tree built in phase1 to find the level of each node.
Phase 3 : Reconstruction
Uses a stack algorithm to construct an alphabetic binary tree based on the node levels.

Both phases 2,3 take time O(n) while phase1 requires O(n logn) time [11].It was shown in [8] that phase1 can also be done in O(n*)* time for some special classes of weight sequences, as for example the increasing weight sequence $q_1 \leq q_2 \leq \ldots \leq q_n$.

3 Merging Optimal Alphabetic Trees

This section gives a brief overview of the merging algorithm introduced in [4]. The problem statement of the algorithm is first presented along with the necessary terminology, then an explanatory example is given.

Problem Statement

The problem can be stated as follows.
Given 2 sequences of LMCPs generated for 2 weight sequences of lengths n_1,n_2, it is required to find, in linear time, the corresponding sequence of LMCPs after concatenating the two weight sequences into one weight sequence of length $n_1 + n_2$.

Terminology

In what follows, we will call the sequence of *LMCP*s for an old tree, ***the old tree list***. Similarly, we will call the sequence of *LMCP*s for the new tree, ***the new tree list***. The set of nodes that needs to be examined to determine each new *LMCP* is called ***the working sequence***.

Old ***LMCP*s**
These are entries of the old tree list.A general form of an old *LMCP* is two compatible nodes that constituted the local minimum compatible pair for its old tree. However, during the course of the algorithm we may face special kinds of old *LMCP*s that will be handled differently.
Blocked *LMCP*s
When an old *LMCP* is to be examined and found to have external nodes separating it from the current working sequence, we call it a blocked *LMCP*. Blocked *LMCP*s are valid *LMCP*s for the new tree, so they are not added to the working sequence and are moved directly to the new tree list. However, although all blocked *LMCP*s will be in ascending order, they will not necessarily appear in their right order in the new tree list, which will make the new tree list not sorted.
Single Nodes
When the two nodes of an old *LMCP* are examined in the working sequence and one of them combines to form the new *LMCP*, the old *LMCP* becomes a broken *LMCP* and must be deleted from all further appearances in the old tree list since it is no longer a valid entry for the new tree. This will result in *LMCP*s with one deleted node and one valid node which we call a single node.

Nodes in the working sequence
The set of nodes to be examined in the working sequence are the nodes of the current old *LMCP* plus the nodes that became valid candidates for the new *LMCP*. Due to the processing of previous *LMCP*s, two kinds of nodes may result.
New Nodes List
At each step, when the working sequence is examined to determine the new *LMCP*, if the new *LMCP* is different from the old one, a new node will result. This new node will participate in further combinations and thus should be considered in the working sequence. Since, new nodes will be generated frequently, we will put them successively in a list called the **new nodes list**. Due to the rules of *LMCP* generation, new nodes will be generated in ascending order and are all compatible with each other. Thus, only the first two nodes of the list are added to the working sequence.
Leftover Nodes
When an old *LMCP* is added to the working sequence, we do not examine the next one till one of its nodes is chosen in the new *LMCP*. If only one is chosen, then the other node remains in the working sequence and becomes a valid candidate for next *LMCP*s. We will call this node a leftover node.
Since the companion of the leftover node formed a new node, the leftover node is compatible with all nodes in the new nodes list. A second leftover node cannot result till the first one combines. Thus, there is at most one leftover node in the working sequence for each old tree.

Algorithm Outline

The idea is to emulate the effect of rebuilding the optimal n-node tree using the Hu-Tucker algorithm [7,11] and making use of the information already obtained during the process of building the two previous trees.

The processing starts when the two boundary nodes, the rightmost node of the left tree and the leftmost node of the right tree, appear in their corresponding *LMCP* lists. As long as these two nodes are external, nodes from one tree cannot combine with nodes from the other

tree . Thus, old *LMCP*s formed in both trees before those two boundary nodes appear remain valid *LMCP*s for the new tree.

When the new *LMCP* list is completely formed, these entries that were originally valid *LMCP*s for the old trees, although sorted amongst themselves , will cause the new list of *LMCP*s to be unsorted. A final merging phase is required to merge 3 lists, the list of valid *LMCP*s from each old tree, and the list of newly formed *LMCP*s .
The following example will demonstrate the process.

Example

Fig. 1 is an example of two 6 nodes OATs to be merged; the weight sequences and the old tree lists are shown.

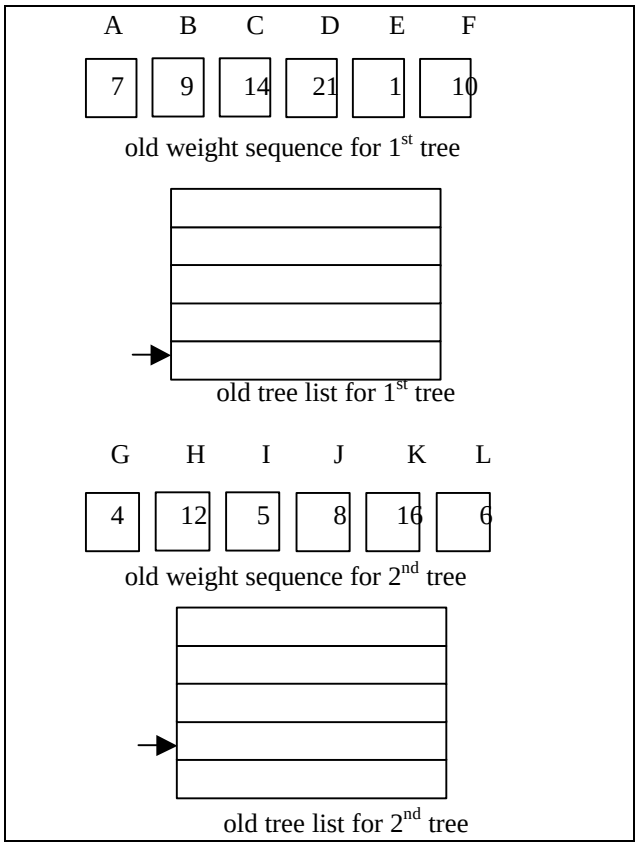


Fig. 1. two OATs to be merged

The old *LMCP* (I,J) is copied since it was formed before the node G combined. The pairs (E,F) and (G,H) constitute the first working sequence, and (E,F) is the chosen *LMCP*. The next one from the 1st tree is looked up (A,B) and found to be blocked, and so is (AB,C). Then the nodes forming the *LMCP* (D,EF) are brought to the working sequence . (EF,G) is the new *LMCP*, D,H become the first leftover nodes and EFG is the first node in the new nodes list. After that, ABC is a single node brought from the 1st tree, and (K,L) are brought from the 2nd tree list. Also, since K is external the minimum internal compatible node, I J, is brought to the working sequence. (K,L) is chosen as the new *LMCP*, and the process continues in the same manner till the root is formed at the last step; different working sequences, the new nodes list, and the new tree list are shown in Fig. 2. Blocked entries from the left tree are marked by (L), and from the right tree by (R).

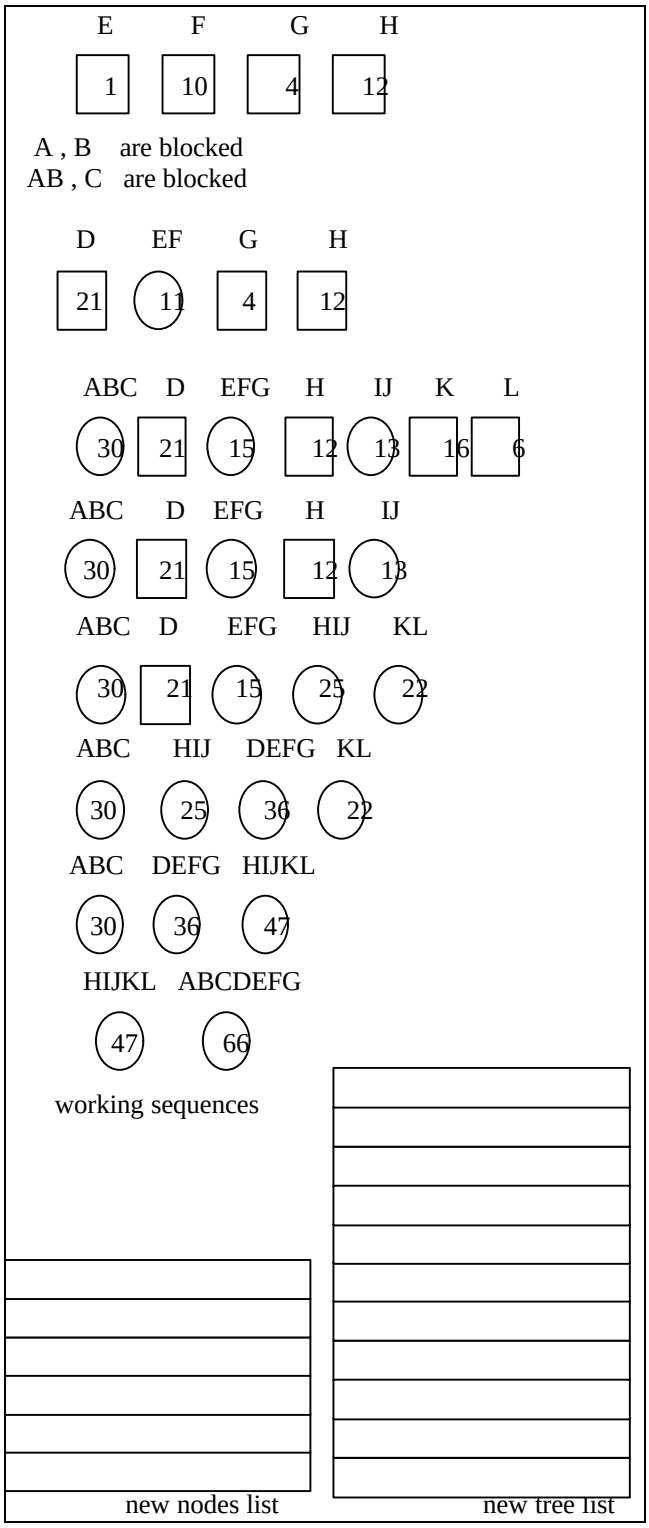


Fig. 2. different working sequences and final lists

4 Successive Merging

In this section we show how the merging algorithm can be applied successively.

Problem statement

In order to be able to apply the previous merging algorithm successively, the new tree list produced at each stage must contain all the information needed by the algorithm, that is the same information contained in the old tree list as a result of running the original Hu-Tucker algorithm. However, the merging algorithm as described in [4], cannot get the left and right borders of each node correctly. This is because some *LMCP*s are out of order; their borders, and the borders of other *LMCP*s as well depend on their right position in the new tree list.

To illustrate the problem, consider as an example the following weight sequence of the left tree at an intermediate step of the merging algorithm; for simplicity, borders are assigned to the *LMCP* pair instead of each node.

... A ..... x ..... y ..... B ..... C D E

Nodes C,D,E are assumed internal.

Let the old tree list contain the following two entries.

. . .
(B, *)
(x,y)
. . .

(x,y) is a blocked *LMCP*, so it will be copied to the new tree list with borders (A,B) (**A**,(x,y),**B**)

Then B will be added to the working sequence

B will combine with say C forming the entry (B,C)

Then (D,E) will combine after that

The new tree list will contain the following entries (with other entries of course), where (x,y) is out of order.

(D,E)
(B,C)
(x,y)

There are 3 possibilities for the sorted order, each will result in different borders

... A ..... x ..... y ..... B ..... C D E

(**A**,(D,E),-)
(**A**,(B,C),-)
(**A**,(x,y),**B**)

(**A**,(D,E),-)
(**A**,(x,y),-)
(**y**,(B,C),-)

(**A**,(x,y),-)
(**y**,(D,E),-)
(**y**,(B,C),-)

Thus, there is no way of knowing the borders of these *LMCP*s before we sort the list.

Solution

The only way to adjust the borders and make the new tree list valid for successive use, is to *maintain the sorted order of the new tree list at each step*. This can be done by keeping blocked *LMCP*s in separate lists (or keeping pointers to them in their original place in the old tree lists) and insert them in the right order; at each step, the selected *LMCP* is compared with the smallest blocked *LMCP* and the smaller of them is inserted.

When each *LMCP* is inserted exactly in its right order, it is much easier to adjust the borders of each node. This depends on the fact that only borders of nodes compatible with the working sequence may change, borders of blocked *LMCP*s will not change unless they became compatible when they are in the right order (like the last two possibilities for (x,y) position in the preceding example), otherwise they are copied with their old borders.

To adjust the borders of compatible nodes, we maintain two linked lists, of external nodes to the left and right of the working area that did not combine yet. For example, in the merging algorithm, the *left borders list* will start with all external nodes in the left tree, and the *right borders list* contains those of the right tree.

Whenever an external node combines (blocked or not blocked), it is deleted from its corresponding borders list (deletion is done in a constant time by maintaining a pointer from each external node to its representing node in the borders list). At any step, the borders of the working area are the heads of the two lists, then the borders of any node inside the working area can be adjusted in constant time.

Example

Here, we repeat the merging example in section 5 and show how the borders of each node can be adjusted.

We start with the two borders lists as follows

Left borders list: F E D C B A

Right borders list: G H I J K L

Thus, the left and right borders are F,G.

At first (I,J) is kept then when (E,F) is chosen, they are compared and (E,F) is found smaller and is inserted in the new tree list; E,F are deleted from the left borders list. (I,J) remains kept in the old tree list (not added to the new tree list), (A,B) and (AB,C) are kept too since they are blocked. Then when (E,FG) is chosen, it is compared to both (I,J) and (A,B); i.e. the smallest blocked *LMCP* from each side. (I,J) is the smaller so it is inserted with its old borders since it is still incompatible; I,J are deleted from the right borders list.

The rest of the steps follow the same procedure; the chosen *LMCP* with its borders, and the corresponding left and right borders lists for each step of the algorithm are shown in Table 1.

Chosen *LMCP* pair	**Left borders list**	**Right borders list**
(**D**,(E,F),**G**)	D C B A	G H I J K L
(**H**,(I,J),**K**)	D C B A	G H K L
(**D**,(EF,G),**H**)	D C B A	H K L

(-,(A,B),**C**)	D C	H K L
(H,(K,L),D)	D C	H
(**D**,(H,IJ),-)	D C	-
(-,(AB,C),**D**)	D	-
(-,(D,EFG),-)	-	-
(-,(HIJ,KL),-)	-	-
(-,(ABC,DEFG),-)	-	-
(-,(ABCDEFG,HIJKL),-)	-	-

Table 1. : the chosen *LMCP*, and the left and right linked lists at each step

5 Building the optimal alphabetic tree

In this section, we use the merging algorithm described above to build an optimal alphabetic tree in a recursive way that is expected to have less runtime, although it has the same order complexity. The tree is constructed by repeated merging; a procedure similar to that of the merge-sort algorithm.

Problem Definition

It is required to find the optimal alphabetic tree for a given set of weights. Here, we propose a different implementation method based on the merging algorithm of the previous section. Applying phases 2,3 of the Hu-Tucker algorithm to the *LMCP* list is a linear time process that is similar for all implementation methods. Thus, we are going to limit the discussion, whether in the analysis or in the comparison between different methods, to finding the *LMCP* list for the tree. Hence the problem can be defined as follows

Given a weight sequence of n nodes, it is required to find the LMCP list for the optimal alphabetic tree of that weight sequence.

For simplicity of recursion, n is assumed to be a power of 2; a condition that can be easily waived without loss of generality.

Algorithm

The steps of the algorithm can be summarized as follows
The set of weights is divided into subsets of length 2, where the *LMCP* list contains a single entry, the existing pair of nodes with the node with smaller index on the left (i.e. an *LMCP* entry reserves the relative order between its two nodes).
The n/2 sublists are then grouped into n/4 pairs, where each pair contains two adjacent sublists. The two *LMCP* lists of each group are merged, using the prescribed merging algorithm, giving the *LMCP* list for the subtree of the 4 nodes in the group.
The process is repeated, at each step the existing n/k sublists each containing k nodes are merged into n/2k sublists each containing 2k nodes. In the last step, two trees are merged, each of length n/2, to get the final tree.

Fig.3 shows a pseudo code of the algorithm, where it is defined in a recursive manner. An optimal tree of length *n* is obtained by finding the optimal tree of its left and right parts, each of length *n/2*, recursively using the same algorithm, then merging them.

```
    Construct (1,n)

{ a procedure to construct an OAT of the weight
sequence in the array'tree', the two parameters are the
indices of the beginning and the end of the array, the
procedure returns an LMCP list}

  Begin
      If (n > 2) Then
          Construct(1, n/2)
          Construct(n/2+1,n)
          Merge(1,n/2,n/2+1,n)
{ a procedure for merging two sublists, the parameters
represent the indices of the beginning and the end in the
array tree}
      Else
          LMCP pair = (tree[1],tree[2])

  End
```

Fig. 3: pseudo code of the algorithm

Complexity Analysis

For the space complexity, the total length of *LMCP* lists at each step of the algorithm never exceeds n. Thus the algorithm has a linear space complexity.

As for the time complexity of the algorithm T(n), it can be described by the following recurrence relation

$T(n) = O(n) + 2\,T(n/2)$
$T(n) = O(n \log n)$

The algorithm has the same order complexity as the well known method for constructing the Hu-Tucker tree. However, it is expected to have a smaller constant factor for its simplicity and since the constant of linearity for the merging algorithm is small, while the known implementation is rather complex and involves the use of many data structures [5]. Specifically, there are two factors that favor the new algorithm.

The first is that the new algorithm has *more locality of reference.* The same set of data (memory locations) is processed by each merging task, then other sets are added gradually at higher levels. On the other hand, the old implementation processes different sets of data to find the *LMCP* and adjust the priority queues. Add to this, the fact that the merged data is stored in arrays (contiguous memory locations), while the dynamic data structures in the other implementations store the data in scattered memory locations.

The second factor is due to *copied LMCPs* in our algorithm and which involves almost no processing. The rules of compatibility imposed on alphabetic trees, cause many of the old *LMCP*s to be blocked and thus valid for the new tree. For example, it is a rare situation

when an *LMCP* formed at the extreme left of the first tree will be broken due to the merging process.

Experimental Results

Sample runs were made to compare runtimes of the proposed algorithm with that of the known implementation. The resulting curves for the average and worst case runtimes are shown in Figs. 4 , 5 respectively, where the proposed method has a considerably smaller runtime.
Also, sample runs showed that the number of copied *LMCP*s exceeds half the entries in the old tree list (by copied *LMCP*s we mean blocked *LMCP*s from both trees in a merging step, and *LMCP*s before the two boundary nodes appear in the old tree lists; i.e. those *LMCP*s that are not brought to the working sequence), Fig.6.

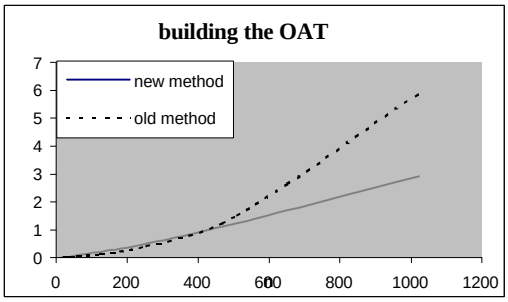


Fig. 4. average runtime for both methods

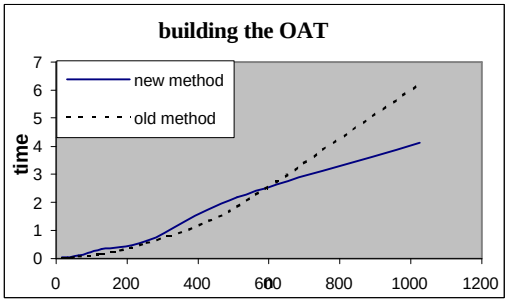


Fig. 5. : worst case runtime for both methods

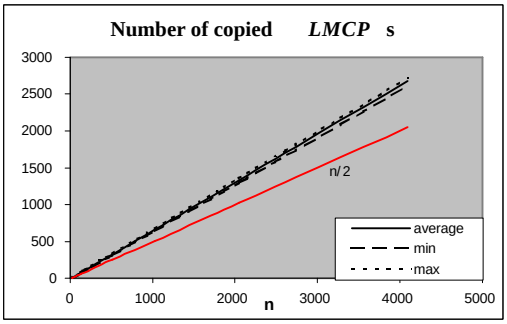


Fig. 6: number of copied *LMCP*s in a merging step

All programs were implemented with Borland C++ programming language, and the random sequences were generated using the language built-in random number generator.

6 Conclusion

An algorithm for merging two optimal alphabetic trees in linear time is used to develop a faster technique to build optimal alphabetic trees. The new algorithm runs in time O(n log n) but with a smaller constant than existing implementations.

References

[1] Ahmed M.A., Belal A.A. and Ahmed K.M., "Optimal insertion in two-dimensional arrays", *International Journal of Information Sciences*, 99(1/2) : 1-20, June 1997.

[2] Andersson A., "A note on searching in a binary search tree", *Software-Practice and Experience*, 21(10) : 1125-1128, 1991.

[3] Belal A.A., Ahmed M.A., Arafat S.M., "Limiting the search for 2-dimensional optimal alphabetic trees", *Fourth International Joint Conference on Information Sciences*, North Carolina-USA, October 1998.

[4] Belal A.A., Mohamed S. Selim, Arafat S.M., "Merging optimal alphabetic trees in linear time", *First International Conference on Intelligent computing and Information Systems*, Cairo-Egypt, June 2002.

[5] Davis S.T., "*Hu-Tucker algorithm for building optimal alphabetic binary search trees*", Rochester Institute of Technology CS Dept., A master thesis, Dec. 1998.

[6] Garcia A.M. and Wachs M.L., "A new algorithm for minimum cost binary trees", *SIAM Journal on Computing*, 6(4): 622-642, 1977.

[7] Hu T.C. and Tucker A.C., "Optimal computer search trees and variable-length alphabetic codes", *SIAM Journal on Applied Mathematics*, 21(4): 514-532, 1971.

[8] Hu T.C., Morgenthaler J.D., "Optimum alphabetic binary trees", Combinatorics and Computer Science, 8th Franco-Japanese and 4th Franco-Chinese Conference, Vol. 1120 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp.234-243.

[9] Huffman D.A., "A method for the construction of minimum redundancy codes", *Proceedings of the IRE*, 40: 1098-1101, 1952.

[10] Klawe M.M. and Mumey B., "Upper and lower bounds on constructing alphabetic binary trees", *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, page 185-193, 1993.

[11] Knuth D.E., "*The Art of Computer Programming, Volume 3: Sorting and Searching*", Addison-Wesley, Reading, MA, 1973.

[12] Przytycka T.M. and Larmore L.L., "The optimal alphabetic tree problem revisited", *Journal of Algorithms*, 28(1): 1-20, June 1997.