TO: CORRESPONDING AUTHOR

AUTHOR QUERIES – TO BE ANSWERED BY THE AUTHOR

The following queries have arisen during the typesetting of your manuscript. Please answer these queries by marking the required corrections at the appropriate point in the text.

Query No.	Nature of Query	Response
Q0	Please check if affiliations for **all** the contributors are correct as per the format given below and provide the missing details, if any: **Name, Department, University/ Company, City, State/Province, Country:** **Author:** Wojciech Rytter **Affn:** New Jersey Institute of Technology, Newark, New Jersey and Warsaw University, Warsaw, Poland.	
Q1	Please check missing citation for Figure 15.5	
Q2	Please provide venue of conference in all conference references.	
Q3	Please provide location details for the references [18,37].	
Q4	Please provide missing details in Ref. [45].	
Q5	Refs. [46–52] are not cited. Please suggest suitable place for its citation in the text.	

15

Trees with Minimum Weighted Path Length

Wojciech Rytter
New Jersey Institute of Technology
and Warsaw University

Q0

15.1 Introduction

The concept of the "weighted path length" is important in data compression and searching. In case of data compression lengths of paths correspond to lengths of code-words. In case of searching they correspond to the number of elementary searching steps. By a *length of a path* we mean usually its number of edges.

Assume we have n weighted items, where w_i is the non-negative *weight* of the ith item. We denote the sequence of weights by $S = (w_1 \ldots w_n)$. We adopt the convention that the items have unique names. When convenient to do so, we will assume that those names are the positions of items in the list, namely integers in $[1 \ldots n]$.

We consider a binary tree T, where the items are placed in vertices of the trees (in leaves only or in every node, depending on the specific problem). We define the minimum weighted path length (cost) of the tree T as follows:

$$cost(T) = \sum_{i=1}^{n} w_i level_T(i)$$

where $level_T$ is the *level function* of T, that is, $level_T(i)$ is the level (or depth) of i in T, defined to be the length of the path in T from the root to i.

In some special cases (lopsided trees) the edges can have different lengths and the path length in this case is the sum of individual lengths of edges on the path.

In this chapter we concentrate on several interesting algorithms in the area:

- Huffman algorithm constructing optimal prefix-free codes in time $O(n \log n)$, in this case the items are placed in leaves of the tree, the original order of items can be different from their order as leaves;
- A version of Huffman algorithm which works in $O(n)$ time if the weights of items are sorted
- Larmore-Hirschberg algorithm for optimal height-limited Huffman trees working in time $O(n \times L)$, where L is the upper bound on the height, it is an interesting algorithm transforming the problem to so called "coin-collector," see [1].

- Construction of optimal binary search trees (OBST) in $O(n^2)$ time using certain property of monotonicity of "splitting points" of subtrees. In case of OBST every node (also internal) contains exactly one item. (Binary search trees are defined in Chapter 3.)
- Construction of optimal alphabetic trees (OAT) in $O(n \log n)$ time: the Garsia-Wachs algorithm [2]. It is a version of an earlier algorithm of Hu-Tucker [3,4] for this problem. The correctness of this algorithm is nontrivial and this algorithm (as well as Hu-Tucker) and these are the most interesting algorithms in the area.
- Construction of optimal lopsided trees, these are the trees similar to Huffman trees except that the edges can have some lengths specified.
- Short discussion of parallel algorithms

Many of these algorithms look "mysterious," in particular the Garsia-Wachs algorithm for optimal alphabetic trees. This is the version of the Hu-Tucker algorithm. Both algorithms are rather simple to understand in how they work and their complexity, but correctness is a complicated issue.

Similarly one can observe a mysterious behavior of the Larmore-Hirschberg algorithm for height-limited Huffman trees. Its "mysterious" behavior is due to the strange reduction to the seemingly unrelated problem of the *coin collector*.

The algorithms relating the cost of binary trees to shortest paths in certain graphs are also not intuitive, for example the algorithm for lopsided trees, see [5], and parallel algorithm for alphabetic trees, see [6]. The efficiency of these algorithms relies on the *Monge property* of related matrices. Both sequential and parallel algorithms for Monge matrices are complicated and interesting.

The area of weighted paths in trees is especially interesting due to its applications (compression, searching) as well as to their relation to many other interesting problems in combinatorial algorithmics.

15.2 Huffman Trees

Assume we have a text x of length N consisting of n different letters with repetitions. The alphabet is a finite set Σ. Usually we identify the i-th letter with its number i. The letter i appears w_i times in x. We need to encode each letter in binary, as $h(a)$, where h is a morphism of alphabet Σ into binary words, in a way to minimize the total length of encoding and guarantee that it is uniquely decipherable, this means that the extension of the morphism h to all words over Σ is one-to-one. The words $h(a)$, where $a \in \Sigma$, are called codewords or codes.

The special case of uniquely decipherable codes are *prefix-free* codes: none of the code is a prefix of another one. The prefix-free code can be represented as a binary tree, with left edges corresponding to zeros, and right edge corresponding to ones.

Let $S = \{w_1, w2, \ldots, w_n\}$ be the sequence of weights of items. Denote by *HuffmanCost(S)* the total cost of minimal encoding (weighted sum of lengths of code-words) and by HT(S) the tree representing an optimal encoding. Observe that several different optimal trees are possible. The basic algorithm is a greedy algorithm designed by Huffman, the corresponding trees and codes are called Huffman trees and Huffman codes.

Example Let *text* = *abracadabra*. The number of occurrences of letters are

$$w_a = 5, w_b = 2, w_c = 1, w_d = 1, w_r = 2.$$

We treat letters as items, and the sequence of weights is:

$$S = (5, 2, 1, 1, 2)$$

An optimal tree of a prefix code is shown in **Figure 15.1**. We have, according to the definition of weighted path length:

$$HuffmanCost(S) = 5*1 + 2*2 + 1*4 + 1*4 + 2*3 = 23$$

The corresponding prefix code is:

$$h(a) = 0, h(b) = 10, h(c) = 1100, h(d) = 1101, h(r) = 111.$$

We can encode the original text *abracadabra* using the codes given by paths in the prefix tree. The coded text is then 01011101100011010101110, that is a word of length 23.

If for example the initial code words of letters have length 5, we get the compression ratio $55/23 \approx 2.4$.
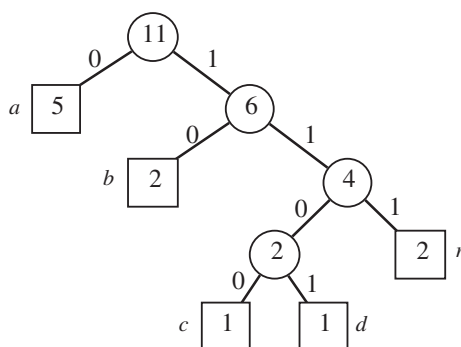
FIGURE 15.1   A Huffman tree T for the items a, b, c, d, r and the weight sequence $S = (5, 2, 1, 1, 2)$. The numbers in internal nodes are sums of weights of leaves in corresponding subtrees. Observe that weighted path length of the tree is the total sum of values in internal nodes. Hence $HuffmanCost(S) = 2 + 4 + 6 + 11 = 23$.

15.2.1 $O(n \log n)$ Time Algorithm

The basic algorithm is the *greedy* algorithm given by Huffman. In this algorithm we can assume that two items of smallest weight are at the bottom of the tree and they are sons of a same node. The *greedy* approach in this case is to minimize the cost *locally*.

Two smallest weight items are *combined* into a single *package* with weight equal to the sum of weights of these small items. Then we proceed recursively. The following observation is used.

Observation Assume that the numbers in internal nodes are sums of weights of leaves in corresponding subtrees. Then the total weighted path length of the tree is the total sum of values in internal nodes.

Due to the observation we have for $|S| > 1$:

$$HuffmanCost(S) = HuffmanCost(S - \{u, w\}) + u + w,$$

where u, w are two minimal elements of S. This implies the following algorithm, in which we assume that initially S is stored in a min-priority queue. The algorithm is presented below as a recursive function $HuffmanCost(S)$ but it can be easily written without recursion. The algorithm computes only the total cost.

Theorem 15.1

Huffman algorithm constructs optimal tree in $O(n \log n)$ time

Proof. In an optimal tree we can exchange two items which are sons of a same father at a bottom level with two smallest weight items. This will not increase the cost of the tree. Hence there is an optimal tree with two smallest weight items being sons of a same node. This implies correctness of the algorithm.

The complexity is $O(n \log n)$ since each operation in the priority queue takes $O(\log n)$ time and there are $O(n)$ operations of Extract-Min. ∎

---

function *HuffmanCost(S)*
{Huffman algorithm: recursive version}
{computes only the cost of minimum weighted tree}

1. if S contains only one element u then return 0;
2. u = Extract Min(S); w = ExtractMin(S);
3. insert(u+w, S);
4. **return** HuffmanCost(S) +u+w

The algorithm in fact computes only the cost of Huffman tree, but the tree can be created on-line in the algorithm. Each time we combine two items we create their father and create links son-father. In the course of the algorithm we keep a forest (collection of trees). Eventually this becomes a single Huffman tree at the end.

15.2.2 Linear Time Algorithm for Presorted Sequence of Items

There is possible an algorithm using "simple" queues with constant time operations of inserting and deleting elements from the queue if the items are *presorted*.

Theorem 15.2

If the weights are already sorted then the Huffman tree can be constructed in linear time.

Proof. If we have sorted queues of remaining original items and remaining newly created items (packages) then it is easy to see that two smallest items can be chosen from among 4 items, two first elements of each queue. This proves correctness.

Linear time follows from constant time costs of single queue operations.   ∎

Linear-Time Algorithm {linear time computation of the cost for presorted items}

1. initialize empty queues Q, S;
   total_cost = 0;
2. place original n weights into nondecreasing order into S;
   the smallest elements at the front of S;
3. **while** $|Q| + |S| > 2$ **do** {
         let u, w be the smallest elements chosen from the first two elements of Q and of S;
         remove u, w from $Q \cup S$; insert$(u + w,\ Q)$;
         $total\_cost\ =\ total\_cost\ +\ (u + w);$}
4. **return** total_cost

15.2.3 Relation between General Uniquely Decipherable Codes and Prefix-free Codes

It would seem that, for some weight sequences, in the class of uniquely decipherable codes there are possible codes which beat every Huffman (prefix-free) code. However it happens that prefix-free codes are optimal within the whole class of uniquely decipherable codes. It follows immediately from the next three lemmas.

Lemma 15.1

For each full (each internal node having exactly two sons) binary tree T with leaves $1 \ldots n$ we have:

$$\sum_{i=1}^{n} 2^{-level_T(i)} = 1$$

Proof. Simple induction on n.   ∎

Lemma 15.2

For each uniquely decipherable code S with word lengths $\ell_1, \ell_2, \ldots, \ell_k$ on the alphabet $\{0, 1\}$ we have :

$$\sum_{i=1}^{k} 2^{-\ell_i} \leq 1$$

Proof. For a set W of words on the alphabet $\{0, 1\}$ define:

$$C(W) = \sum_{x \in W} 2^{-|x|}$$

We have to show that $C(S) \leq 1$. Let us first observe the following simple fact.

Observation.

If S is uniquely decipherable then $C(S)^n = C(S^n)$ for all $n \geq 1$.

The proof that $C(S) \leq 1$ is now by contradiction, assume $C(S) > 1$. Let c be the length of the longest word in S. Observe that

$$C(\Sigma^k) = 1 \text{ for each } k, \ C(S^n) \leq C(\{x \in \Sigma^* : 1 \leq |x| \leq cn \}) = cn$$

Denote $q = C(S)$. Then we have:

$$C(S)^n = q^n \leq cn$$

For $q > 1$ this inequality is not true for all n, since

$$\lim q^n/(cn) = +\infty \text{ if } q > 1$$

Therefore it should be $q \leq 1$ and $C(S) \leq 1$. This completes the proof. ∎

Lemma 15.3

[Kraft's inequality] There is a prefix code with word lengths $\ell_1, \ell_2, \ldots, \ell_k$ on the alphabet $\{0, 1\}$ iff

$$\sum_{i=1}^{k} 2^{-\ell_i} \leq 1 \tag{15.1}$$

Proof. It is enough to show how to construct such a code if the inequality holds. Assume the lengths are sorted in the increasing order. Assume we have a (potentially) infinite full binary tree. We construct the next codeword by going top-down, starting from the root. We assign to the i-th codeword, for $i = 1, 2, 3, \ldots, k$, the lexicographically first path of length ℓ_i, such that the bottom node of this path has not been visited before. It is illustrated in Figure 15.2. If the path does not exist then this means that in this moment we covered with paths a full binary tree, and the actual sum equals 1. But some other lengths remained, so it would be :

$$\sum_{i=1}^{k} 2^{-\ell_i} > 1$$

a contradiction. This proves that the construction of a prefix code works, so the corresponding prefix-free code covering all lengths exists. This completes the proof.

∎

The lemmas imply directly the following theorem.

Theorem 15.3

A uniquely decipherable code with prescribed word lengths exists iff a prefix code with the same word lengths exists.

We remark that the problem of testing unique decipherability of a set of codewords is complete in nondeterministic logarithmic space, see [7].
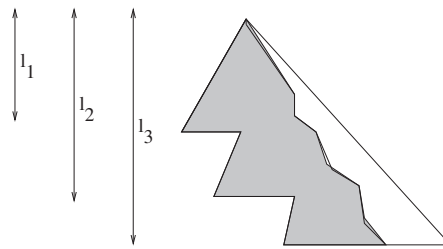

FIGURE 15.2 Graphical illustration of constructing prefix-free code with prescribed lengths sequence satisfying Kraft inequality.

15.2.4 Huffman Codes and Entropy

The performance of Huffman codes is related to a measure of information of the source text, called the *entropy* (denoted by $\mathcal{E}$) of the alphabet. Let w_a be n_a/N, where n_a is the number of occurrences of a in a given text of length N. In this case the sequence S of weights w_a is normalized $\sum_{i=1}^{n} w_i = 1$.

The quantity w_a can be now viewed as the probability that letter a occurs at a given position in the text. This probability is assumed to be independent of the position. Then, the entropy of the alphabet according to w_a's is defined as

$$\mathcal{E}(A) = -\sum_{a \in A} w_a \log w_a.$$

The entropy is expressed in bits (log is a base-two logarithm). It is a lower bound of the average length of the code words $h(a)$,

$$m(A) = \sum_{a \in A} w_a.|h(a)|.$$

Moreover, Huffman codes give the best possible approximation of the entropy (among methods based on coding of the alphabet). This is summarized in the following theorem whose proof relies on the inequalities from Lemma 15.2.

Theorem 15.4

Assume the weight sequence A of weights is normalized. The total cost $m(A)$ of any uniquely decipherable code for A is at least $\mathcal{E}(A)$, and we have

$$\mathcal{E}(A) \leq \textit{HuffmanCost}(A) \leq \mathcal{E}(A) + 1$$

15.2.5 Huffman Algorithm for *t*-ary Trees

An important generalization of Huffman algorithm is to t-ary trees. Huffman algorithm generalizes to the construction of prefix codes on alphabet of size $t > 2$. The trie of the code is then an *almost full t-ary tree*.

We say that t-ary tree is almost full if all internal nodes have exactly t sons, except possibly one node, which has less sons, in these case all of them should be leaves (let us call this one node a *defect* node).

We perform similar algorithm to Huffman method for binary trees, except that each time we select t items (original or combined) of smallest weight.

There is one technical difficulty. Possibly we start by selecting a smaller number of items in the first step. If we know t and the number of items then it is easy to calculate number q of sons of the defect node, for example if $t = 3$ and $n = 8$ then the defect node has two sons. It is easy to compute the number q of sons of the defect node due to the following simple fact.

Lemma 15.4

If T is a full t-ary tree with m leaves then $m \bmod (t-1) = 1$.

We start the algorithm by combining q smallest items. Later each time we combine exactly t values. To simplify the algorithm we can add the smallest possible number of dummy items of weigh zero to make the tree full t-ary tree.

15.3 Height Limited Huffman Trees

In this section only, for technical reason, we assume that the length of the path is the number of its vertices. For a sequence S of weights the total cost is changed by adding the sum of weights in S.

Assume we have the same problem as in the case of Huffman trees with additional restriction that the height of the tree is limited by a given parameter L. A beautiful algorithm for this problem has been given by Larmore and Hirschberg, see [8].

15.3.1 Reduction to the Coin Collector Problem

The main component of the algorithm is the reduction to the following problem in which the crucial property play powers of two. We call a real number dyadic iff it has a finite binary representation.
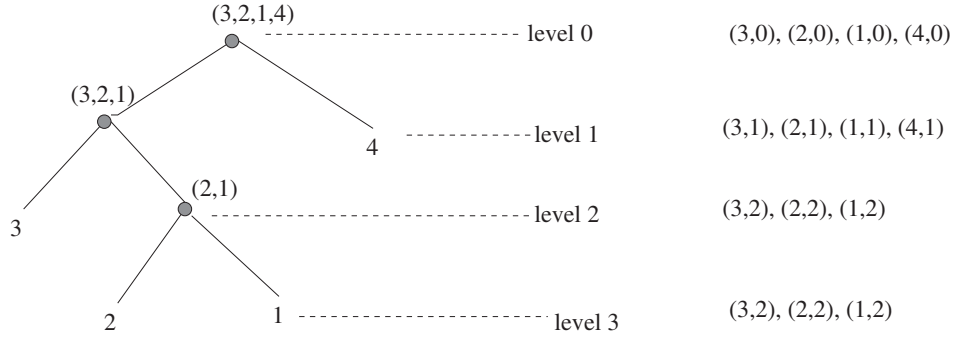
FIGURE 15.3 A Huffman tree T for the items $1, 2, 3, 4$ of height limited by 4 and the corresponding solution to the Coin Collector problem. Each node of the tree can be treated as a *package* consisting of leaves in the corresponding subtree. Assume $weight(i) = i$. Then in the corresponding Coin Collector problem we have $weight(i, h) = i$, $width(i, h) = 2^{-h}$.

Coin Collector problem:

Input: A set I of m items and dyadic number X, each element of I has a *width* and a *weight*, where each width is a (possibly negative) power of two, and each weight is a positive real number.

Output: $CoinColl(I, X)$ - the minimum total weight of a subset $S \subseteq I$ whose widths sum to X.

The following trivial lemma plays important role in the reduction of height limited tree problem to the Coin Collector problem.

Lemma 15.5

Assume T is a full binary tree with n leaves, then

$$\sum_{v \in T} 2^{-level_T(v)} = n + 1$$

Assume we have Huffman coding problem with n items with the sequence of weights weights $W = w_1, w_2, \ldots, w_n$. We define an instance of the Coin Collector problem as follows:

- $I_W = \{(i, l) : i \in [1 \ldots n],\ l \in [1, \ldots L],$
- $width(i, l) = 2^{-l}$, $weight(i, l) = w_i$ for each i, l
- $X_w = n + 1$.

The intuition behind this strange construction is to view nodes as packages consisting of of original items (elements in the leaves). The internal node v which is a *package* consisting of (leaves in its subtree) items $i_1, i_2, \ldots, i_k$ can be treated as a set of coins $(i_1, h), (i_2, h), \ldots (i_k, h)$, where h is the level of v, and $weight(i_j, h) = weight(i_j)$. The total weight of the set of coins is the cost of the Huffman tree.

Example Figure 15.3 shows optimal Huffman tree for $S = (1, 2, 3, 4)$ with height limited by 4, and the optimal solution to the corresponding Coin Collector problem. The sum of widths of the coins is $4 + 1$, and the total weight is minimal. It is the same as the cost of the Huffman tree on the left, assuming that leaves are also contributing their weights (we scale cost by the sum of weights of S).

Hirschberg and Larmore, see [8], have shown the following fact.

Lemma 15.6

The solution $CoinColl(I_W, X_W)$ to the Coin Collector Problem is the cost of the optimal L-height restricted Huffman tree for the sequence W of weights.

15.3.2 The Algorithm for the Coin Collector Problem

The height limited Huffman trees problem is thus reduced to the Coin Collector Problem. The crucial point in the solution of the latter problem is the fact that weights are powers of two.

Denote *MinWidth*(X) to be the smallest power of two in binary representation of number X. For example *MinWidth*(12) = 4 since 12 = 8 + 4. Denote by *MinItem*(I) the item with the smallest width in I.

Lemma 15.7

If the items are sorted with respect to the weight then the Coin Collector problem can be solved in linear time (with respect to the total number |I| of coins given in the input).

Proof. The recursive algorithm for the Coin Collector problem is presented below as a recursive function *CoinColl*(I, X). There are several cases depending on the relation between the smallest width of the item and minimum power of two which constitutes the binary representation of X. In the course of the algorithm the set of weights shrinks as well as the size of X. The linear time implementation of the algorithm is given in References 1. ∎

```
function CC(I, X); {Coin Collector Problem}
    {compute nnimal weight of a subset of I of total width X
    x := MinItem(X); r := width(x);
    if r > MinWidth(X) then
        no solution exists else
    if r = MinWidth(X) then
        return CC(I − {x}, X − r) + weight(x) else
    if r < MinWidth(X) and there is only one item of width r then
        return CC(I − {x}, X) else
            let x, x′ be two items of smallest weight and width r,
            create new item y such that
                width(y) = 2r, weight(y) = weight(x) + weight(x′);
            return CC(I − {x, x′}∪{y}, X)
```

The last two lemmas imply the following fact.

Theorem 15.5

The problem of the Huffman tree for n items with height limited by L can be solved in $O(n \cdot L)$ time.

Using complicated approach of the Least Weight Concave Subsequence the complexity has been reduced to $n\sqrt{L\log n} + n\log n$ in Reference 9. Another small improvement is by Schieber [10]. An efficient approximation algorithm is given in References 11–13. The dynamic algorithm for Huffman trees is given in Reference 14.

15.4 Optimal Binary Search Trees

Assume we have a sequence of $2n + 1$ weights (nonnegative reals)

$$\alpha_0, \beta_1, \alpha_1, \beta_2, \ldots, \alpha_{n-1}, \beta_n, \alpha_n$$

Let Tree($\alpha_0, \beta_1, \alpha_1, \beta_2, \ldots, \alpha_{n-1}, \beta_n, \alpha_n$) be the set of all full binary weighted trees with n internal nodes, where the i-th internal node (in the in-order) has the weight β_i, and the i-th external node (the leaf, in the left-to-right order) has the weight α_i. The in-order traversal results if we visit all nodes in a recursive way, first the left subtree, then the root, and afterwards the right subtree.

If T is a binary search tree then define the *cost* of T as follows:

$$cost(T) = \sum_{v \in T} level_T(v) \cdot weight(v).$$

Let $OPT(\alpha_0, \beta_1, \ldots, \alpha_{n-1}, \beta_n, \alpha_n)$ be the set of trees Tree($\alpha_0, \beta_1, \ldots, \alpha_{n-1}, \beta_n, \alpha_n$) whose cost is minimal.

We use also terminology from Reference 15. Let $K_1, \ldots K_n$ be a sequence of n weighted items (keys), which are to be placed in a binary search tree. We are given $2n + 1$ weights (probabilities): $q_0, p_1, q_1, p_2, q_2, p_3, \ldots, q_{n-1}, p_n, q_n$ where

- p_i is the probability that K_i is the search argument;
- q_i is the probability that the search argument lies between K_i and K_{i+1}.

The OBST problem is to construct an optimal binary search tree, the keys K_i's are to be stored in internal nodes of this binary search tree and in its external nodes special items are to be stored. The i-th special item K_i' corresponds to all keys which are strictly between K_i and K_{i+1}. The binary search tree is a full binary tree whose nodes are labeled by the keys. Using the abstract terminology introduced above the OBST problem consists of finding a tree $T \in OPT(q_0, p_1, q_1, p_2, \ldots, q_{n-1}, p_n, q_n)$, see an example tree in Figure 15.4.

Denote by $obst(i, j)$ the set $OPT(q_i, p_{i+1}, q_{i+1}, \ldots, q_{j-1}, p_j, q_j)$. Let $cost(i, j)$ be the cost of a tree in $obst(i, j)$, for $i < j$, and $cost(i, i) = q_i$. The sequence $q_i, p_{i+1}, q_{i+1}, \ldots, q_{j-1}, p_j, q_j$ is here the subsequence of $q_0, p_1, q_1, p_2, \ldots, q_{n-1}, p_n, q_n$, consisting of some number of consecutive elements which starts with q_i and ends with q_j. Let

$$w(i, j) = q_i + p_{i+1} + q_{i+1} + \cdots + q_{j-1} + p_j + q_j$$

The dynamic programming approach to the computation of the OBST problem relies on the fact that the subtrees of an optimal tree are also optimal. If a tree $T \in obst(i, j)$ contains in the root an item K_k then its left subtree is in $obst(i, k-1)$ and its right subtree is in $obst(k, j)$. Moreover, when we join these two subtrees then the contribution of each node increases by one (as one level is added), so the increase is $w(i, j)$. Hence the costs obey the following *dynamic programming recurrences* for $i < j$:

$$cost(i, j) = \min\{cost(i, k-1) + cost(k, j) + w(i, j) : i < k \leq j\}.$$

Q1   Denote the smallest value of k which minimizes the above equation by $cut(i, j)$. This is the first point giving an optimal decomposition of $obst(i, j)$ into two smaller (son) subtrees. Optimal binary search trees have the following crucial property (proved in Reference 16, see the figure for graphical illustration)

$$monotonicity\ property: \ i \leq i' \leq j \leq j' \implies cut(i, j) \leq cut(i', j')$$

The property of monotonicity, the cuts and the quadratic algorithm for the OBST were first given by Knuth. The general dynamic programming recurrences were treated by Yao [17], in the context of reducing cubic time to quadratic.
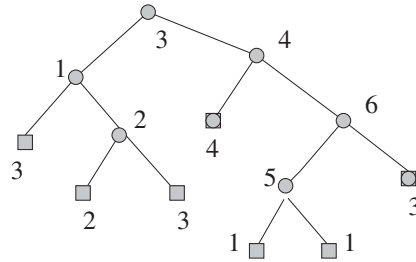

FIGURE 15.4 A binary search tree for the sequences: $\beta = (\beta_1, \beta_2, \ldots, \beta_6) = (1, 2, 3, 4, 5, 6)$, $\alpha = (\alpha_0, \alpha_1, \ldots \alpha_6) = (3, 2, 3, 4, 1, 1, 3)$. We have $cut(0, 6) = 3$.
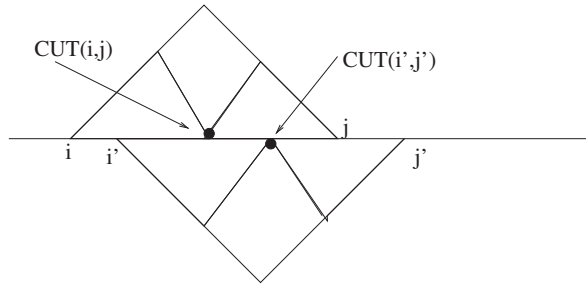

FIGURE 15.5 Graphical illustration of the monotonicity property of cuts.

Theorem 15.6

Optimal binary search trees can be computed in $O(n^2)$ time.

Proof. The values of $cost(i, j)$ are computed by tabulating them in an array. Such tabulation of costs of smaller subproblems is the basis of the *dynamic programming* technique. We use the same name *cost* for this array. It can be computed in $O(n^3)$ time, by processing diagonal after diagonal, starting with the central diagonal.

In case of optimal binary search trees this can be reduced to $O(n^2)$ using additional tabulated values of the cuts in table *cut*. The k-th diagonal consists of entries i, j such that $j - i = k$. If we have computed cuts for k-th diagonal then for (i, j) on the $(k + 1)$-th diagonal we know that

$$cut(i, j - 1) \leq cut(i, j) \leq cut(i + 1, j)$$

Hence the total work on the $(k + 1)$-th diagonal is proportional to the sum of telescoping series:

$$cut(1, k + 1) - cut(0, k) + cut(2, k + 2) - cut(1, k + 1) +$$
$$cut(3, k + 3) - cut(2, k + 2) + \cdots cut(n - k, k) - cut(n - k - 1, k - 1)$$

which is $O(n)$. Summing over all diagonals gives quadratic total time to compute the tables of cuts and costs. Once the table $cost(i, j)$ is computed then the construction of an optimal tree can be done in quadratic time by tracing back the values of *cut*s. ∎

15.4.1 Approximately Optimal Binary Search Trees

We can attempt to reduce time complexity at the cost of slightly increased cost of the constructed tree. A common-sense approach would be to insert the keys in the order of decreasing frequencies. However this method occasionally can give quite bad trees.

Another approach would be to choose the root so that the total weights of items in the left and right trees are as close as possible. However it is not so good in pessimistic sense.

The combination of this methods can give quite satisfactory solutions and the resulting algorithm can be linear time, see [18]. *Average* subquadratic time has been given in Reference 19.

15.5 Optimal Alphabetic Tree Problem

The alphabetic tree problem looks very similar to the Huffman problem, except that the leaves of the alphabetic tree (read left-to-right) should be in the same order as in the original input sequence. Similarly as in Huffman coding the binary tree must be *full*, that is, each internal node must have exactly two sons.

The main difficulty is that we cannot localize easily two items which are to be combined.

Assume we have a sequence of n weighted items, where w_i is the non-negative *weight* of the ith item. We write $\alpha = w_1 \ldots w_n$. The sequence will be changing in the course of the algorithm.

An alphabetic tree over α is an ordered binary tree T with n leaves, where the ith leaf (in left-to-right order) corresponds to the ith item of The optimal alphabetic tree problem (OAT problem) is to find an alphabetic tree of minimum cost.

The Garsia-Wachs algorithm solves the alphabetic tree problem, it is a version of an earlier algorithm by Hu and Tucker, see [4]. The strangest part of the algorithm is that it permutes α, though the final tree should have the order of leaves the same as the order of items in the original sequence. We adopt the convention that the items of α have unique names, and that these names are preserved when items are moved. When convenient to do so, we will assume that those names are the positions of items in the list, namely integers in $[1 \ldots n]$.

15.5.1 Computing the Cost of Optimal Alphabetic Tree

First we show how to compute only the cost of the whole tree, however this computation does not give automatically an optimal alphabetic tree, since we will be permuting the sequence of items. Each time we combine two adjacent items in the current permutation, however these items are not necessarily adjacent in the original sequence, so in any legal alphabetic tree they cannot be sons of a same father.

The alphabetic tree is constructed by reducing the initial sequence of items to a shorter sequence in a manner similar to that of the Huffman algorithm, with one important difference. In the Huffman algorithm, the minimum pair of items are combined, because it can be shown that they are siblings in the optimal tree. If we could identify two adjacent items that are siblings in the optimal alphabetic tree, we could combine them and then proceed recursively. Unfortunately, there is no known way to identify such a pair. Even a minimal pair may not be siblings. Consider the weight sequence (8 7 7 8). The second and the third items are not siblings in any optimal alphabetic tree.

Instead, the HT and GW algorithms, as well as the algorithms of [6,20–22], operate by identifying a pair of items that have the same level in the optimal tree. These items are then combined into a single "package," reducing the number of items by one. The details on how this process proceeds differ in the different algorithms. Define, for $1 \leq i < n$, the ith *two-sum*:

$$TwoSum(i) = w_i + w_{i+1}$$

A pair of adjacent items $(i, i+1)$ is a *locally minimal pair* (or *lmp* for short) if

$$
\begin{aligned}
TwoSum(i-1) &\geq TwoSum(i) && \text{if} \quad i > 1 \\
TwoSum(i) &< TwoSum(i+1) && \text{if} \quad i \leq n-2
\end{aligned}
$$

A locally minimal pair which is currently being processed is called the *active pair*.

The Operator *Move*. If w is any item in a list π of weighted items, define $RightPos(w)$ to be the predecessor of the nearest right larger or equal neighbor of w. If w has no right larger or equal neighbor, define $RightPos(w)$ to be $|\pi| + 1$.

Let $Move(w, \pi)$ be the operator that changes π by moving w w is inserted between positions $RightPos(w) - 1$ and $RightPos(w)$. For example

$$Move(7, (2, 5, 7, 2, 4, 9, 3, 4) = (2, 5, 2, 4, 7, 9, 3, 4)$$

```
function GW(π); {π is a sequence of names of items}
    {restricted version of the Garsia-Wachs algorithm}
    { computing only the cost of optimal alphabetic tree }
    if π = (v) (π consists of a single item)
    then return 0
      else
        find any locally minimal pair (u, w) of π
        create a new item x whose weight is weight(u) + weight(w);
        replace the pair u, v by the single item x;
        { the items u, v are removed }
        Move(v, π);
        return GW(π) + weight(v);
```

Correctness of the algorithm is a complicated issue. There are two simplified proofs, see [23,24] and we refer to these papers for detailed proof. In Reference 23 only the rightmost minimal pair can be processed each time, while [24] gives correctness of general algorithm when any minimal pair is processed, this is important in parallel computation, when we process simultaneously many such pairs. The proof in Reference 24 shows that correctness is due to *well-shaped* bottom segments of optimal alphabetic trees, this is expressed as a *structural theorem* in Reference 24 which gives more insight into the global structure of optimal alphabetic trees.

For $j > i + 1$ denote by $\pi_{i,j}$ the sequence π in which elements $i, i+1$ are moved just before left of j.

Theorem 15.7

[Correctness of the GW algorithm]
Let $(i, i+1)$ be a locally minimal pair and $RightPos(i, i+1) = j$, and let T' be a tree over the sequence $\pi_{i,j}$, optimal among all trees over $\pi_{i,j}$ in which $i, i+1$ are siblings. Then there is an optimal alphabetic tree T over the original sequence $\pi = (1, \ldots n)$ such that $T \cong T'$.

Correctness can be also expressed as equivalence between some classes of trees.

Two binary trees T_1 and T_2 are said to be **level equivalent** (we write $T_1 \cong T_2$) if T_1, and T_2 have the same set of leaves (possibly in a different order) and $level_{T_1} = level_{T_2}$.

Denote by $OPT(i)$ the set of all alphabetic trees over the leaf-sequence $(1, \ldots n)$ which are optimal among trees in which i and $i+1$ are at the same level. Assume the pair $(i, i+1)$ is locally minimal. Let $OPT_{moved}(i)$ be the set of all alphabetic trees over the leaf-sequence $\pi_{i,j}$ which are optimal among all trees in which leaves i and $i+1$ are at the same level, where $j = RightPos(i, i+1)$.

Two sets of trees OPT and OPT$'$ are said to be *level equivalent*, written OPT $\cong$ OPT$'$, if, for each tree $T \in$ OPT, there is a tree $T' \in$ OPT$'$ such that $T' \cong T$, and vice versa.

Theorem 15.8

Let $(i, i+1)$ be a locally minimal pair. Then

1. $\text{OPT}(i) \cong \text{OPT}_{moved}(i)$.
2. $\text{OPT}(i)$ contains an optimal alphabetic tree T.
3. $\text{OPT}_{moved}(i)$ contains a tree T' with $i, i+1$ as siblings.

15.5.2 Construction of Optimal Alphabetic Tree

The full Garsia-Wachs algorithm first computes the level tree. This tree can be easily constructed in the function $GW(\pi)$ when computing the cost of alphabetic tree. Each time we sum weights of two items (original or newly created) then we create new item which is their father with the weight being the sum of weights of sons.

Once we have a level tree, the optimal alphabetic tree can be constructed easily in linear time. Figures 15.6 through 15.8 show the process of construction the level tree and construction an optimal alphabetic tree knowing the levels of original items.

Lemma 15.8

Assume we know level of each leaf in an optimal alphabetic tree. Then the tree can be constructed in linear time.

Proof. The levels give the *"shape"* of the tree, see Figure 15.8.

Assume $l_1, l_2, l_3, \ldots, l_n$ is the sequence of levels. We scan this sequence from left-to-right until we find *the first* two levels l_i, l_{i+1} which are the same. Then we know that the leaves i and $i+1$ are sons of a same father, hence we link these leaves to a newly created father and we remove these leaves, in the level sequence the pair l_i, l_{i+1} is replaced by a single level $l_i - 1$. Next we check if $l_{i-1} = l_i - 1$, if not we search to the right. We keep the scanned and newly created levels on the stack. The total time is linear. ∎

There are possible many different optimal alphabetic trees for the same sequence, Figure 15.9 shows an alternative optimal alphabetic tree for the same example sequence.

Theorem 15.9

Optimal alphabetic tree can be constructed in $O(n \log n)$ time.

Proof. We keep the array of levels of items. The array *level* is global of size $(2n - 1)$. Its indices are the names of the nodes, that is, the original n items and the $(n-1)$ nodes ("packages") created during execution of the algorithm. The algorithm works in quadratic time, if implemented in a naive way. Using priority queues, it works in $O(n \log n)$ time. Correctness follows directly from Theorem 15.7. ∎

15.5.3 Optimal Alphabetic Trees for Presorted Items

We have seen that Huffman trees can be constructed in linear time if the weights are presorted. Larmore and Przytycka, see [21] have shown that slightly weaker similar result holds for alphabetic trees as well:

assume that weights of items are sortable in linear time, then the alphabetic tree problem can be solved in $O(n \log \log n)$ time.

Open problem Is it possible to construct alphabetic trees in linear time in the case when the weights are sortable in linear time?

15.6 Optimal Lopsided Trees

The problem of finding optimal prefix-free codes for unequal letter costs consists of finding a minimal cost prefix-free code in which the encoding alphabet consists of unequal cost (length) letters, of lengths α and β, $\alpha \leq \beta$. We restrict ourselves here only to binary trees. The code is represented by a *lopsided tree*, in the same way as a Huffman tree represents the solution of the Huffman coding problem. Despite the similarity, the case of unequal letter costs is much harder then the classical Huffman problem; no polynomial time algorithm is known for general letter costs, despite a rich literature on the problem, for example, [25,26]. However there are known polynomial time algorithms when α and β are integer constants [26].

The problem of finding the minimum cost tree in this case was first studied by Karp [27] in 1961 who solved the problem by reduction to integer linear programming, yielding an algorithm exponential in both n and β. Since that time there has been much work on various aspects of the problem such as; bounding the cost of the optimal tree, Altenkamp and Mehlhorn [28], Kapoor
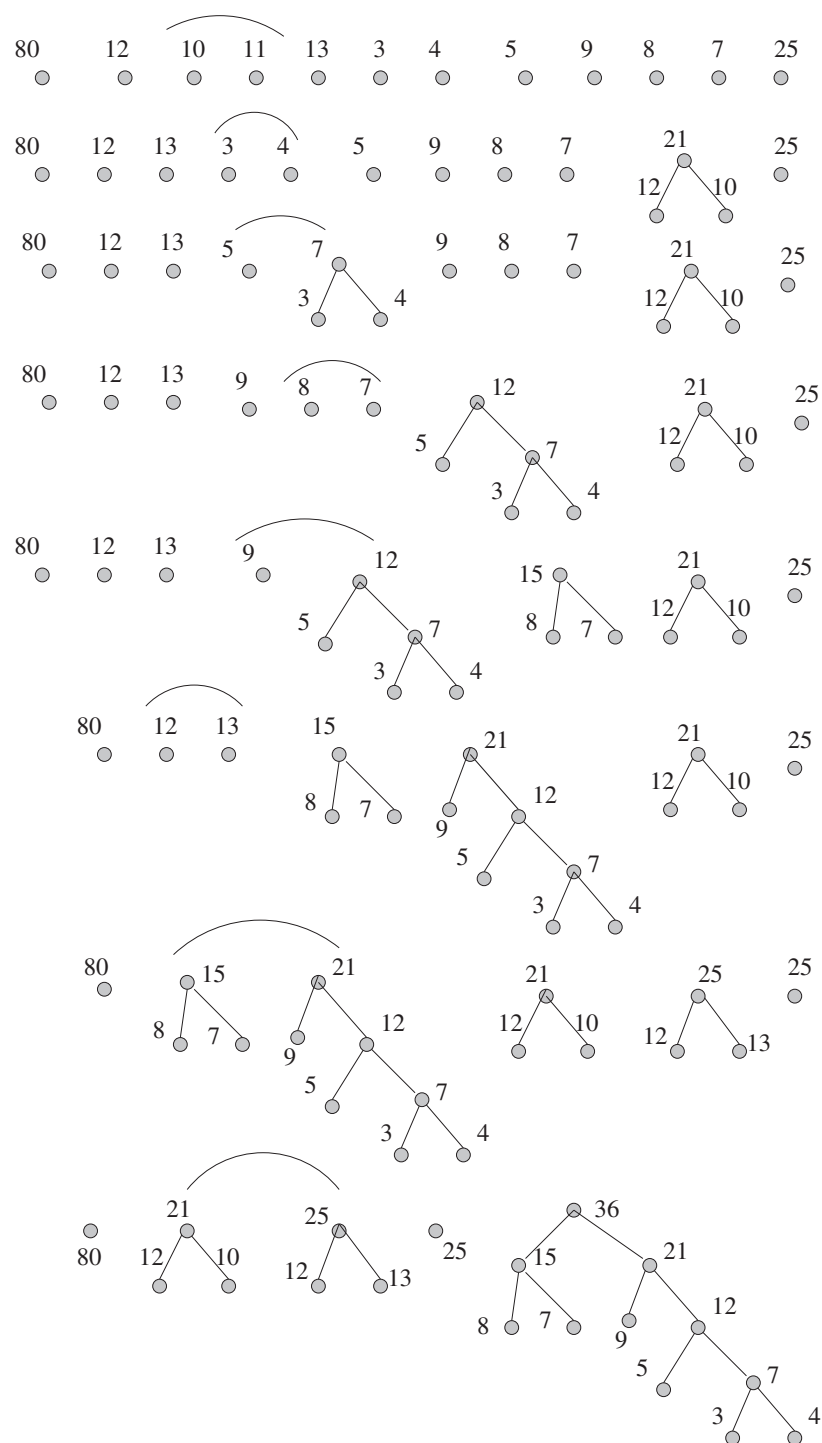
FIGURE 15.6 The first 7 phases of Garsia-Wachs algorithm.

and Reingold [29] and Savari [30]; the restriction to the special case when all of the weights are equal, Cot [31], Perl Gary and Even [32], and Choi and Golin [33]; and approximating the optimal solution, Gilbert [34]. Despite all of these efforts it is still, surprisingly, not even known whether the basic problem is polynomial-time solvable or in *NP*-complete.

Golin and Rote [26] describe an $O(n^{\beta+2})$-time dynamic programming algorithm that constructs the tree in a top-down fashion.
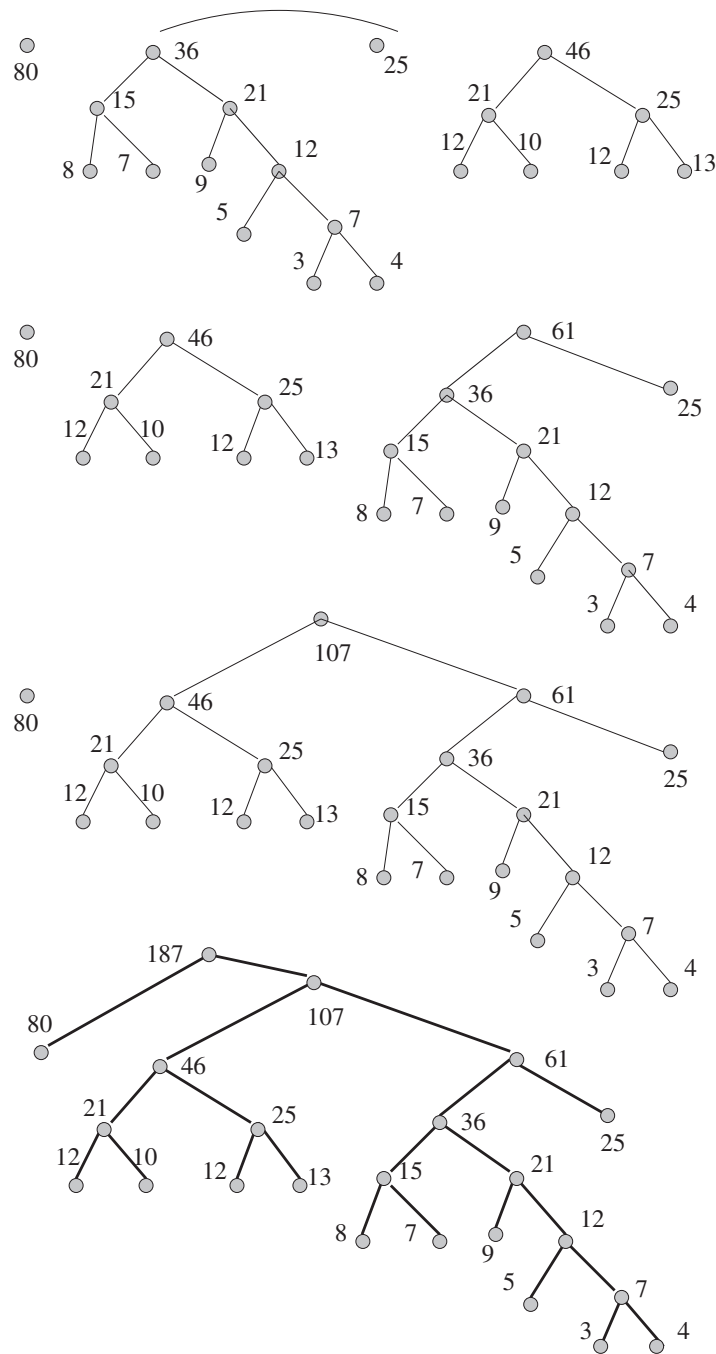
FIGURE 15.7 The last phases of Garsia-Wachs algorithm. The final tree is the level tree (but not alphabetic tree). The total cost (sum of values in internal nodes equal to 538) and the levels of original items are the same as in optimal alphabetic tree. The level sequence for the original sequence (80, 12, 10, 11, 13, 3, 4, 5, 9, 8, 7, 25) is: $\mathcal{L} = (1, 4, 4, 4, 4, 7, 7, 6, 5, 5, 3)$.

This has been improved using a different approach (monotone-matrix concepts, for example, the *Monge property* and the SMAWK algorithm [35].

Theorem 15.10: [5]

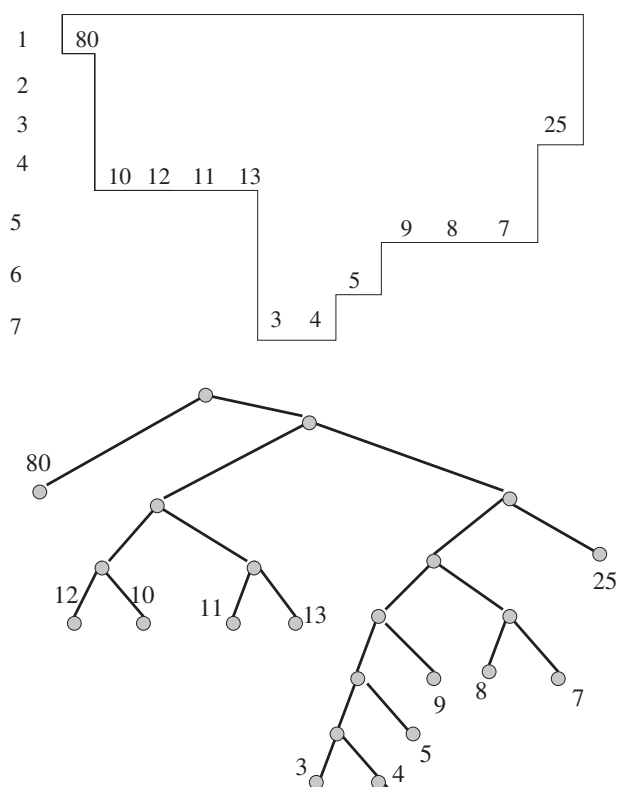Optimal lopsided trees can be constructed in $O(n^{\beta})$ time.

FIGURE 15.8 The shape of the optimal tree given by the level sequence and the final optimal alphabetic tree (cost = 538) corresponding to this shape and to the weight sequence (80, 12, 10, 11, 13, 3, 4, 5, 9, 8, 7, 25).
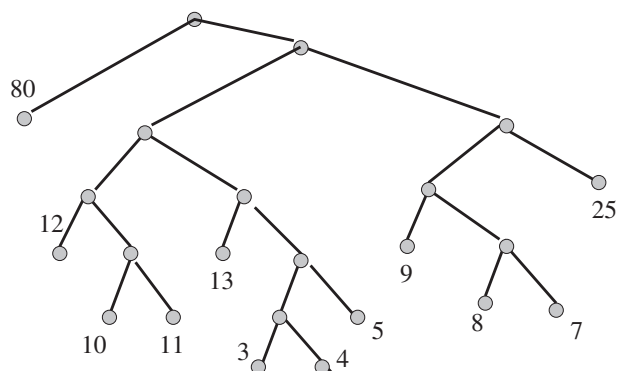

FIGURE 15.9 An alternative optimal alphabetic tree (cost = 538) for the same weight sequence.

This is the the most efficient known algorithm for the case of small β; in practice the letter costs are typically small (*e.g.,* Morse codes).

Recently a scheme of an efficient approximating algorithm has been given.

Theorem 15.11: [36]

There is a polynomial time approximation scheme for optimal lopsided trees.

15.7 Parallel Algorithms

As a model of parallel computations we choose the *Parallel Random Access Machines* (PRAM), see [37]. From the point of view of parallel complexity two parameters are of interest: parallel time (usually we require polylogarithmic time) and total work (time multiplied by the number of processors).

The sequential greedy algorithm for Huffman coding is quite simple, but unfortunately it appears to be inherently sequential. Its parallel counterpart is much more complicated, and requires a new approach. The global structure of Huffman trees must be explored in depth.

A full binary tree T is said to be *left-justified* if it satisfies the following properties:

1. the depths of the leaves are in non-increasing order from left to right,
2. let u be a left brother of v, and assume that the height of the subtree rooted at v is at least l. Then the tree rooted at u is full at level l, which means that u has 2^l descendants at distance l.

15.7.1 Basic Property of Left-Justified Trees

Let T be a left-justified binary tree. Then, T consists of one leftmost branch and the subtrees hanging from this branch have logarithmic height.

Lemma 15.9

Assume that the weights w_1, w_2, ..., w_n are pairwise distinct and in increasing order. Then, there is Huffman tree for $(w_1, w_2, \ldots, w_n)$ that is left-justified.

The left-justified trees are used together with efficient algorithm for the CLWS problem (the *Concave Least Weight Subsequence* problem, to be defined below) to show the following fact.

Theorem 15.12: [38]

The parallel Huffman coding problem can be solved in polylogarithmic time with quadratic work.

Hirschberg and Larmore [8] define the *Least Weight Subsequence* (LWS) problem as follows: Given an integer n, and a real-valued *weight* function $w(i, j)$ defined for integers $0 \le i < j \le n$, find a sequence of integers $\overline{\alpha} = (0 = \alpha_0 < \alpha_1 < \ldots < \alpha_{k-1} < \alpha_k = n)$ such that $w(\overline{\alpha}) = \sum_{i=0}^{k-1} w(\alpha_i, \alpha_{i+1})$ is minimized. Thus, the LWS problem reduces trivially to the minimum path problem on a weighted directed acyclic graph. The *Single Source* LWS problem is to find such a minimal sequence $0 = \alpha_0 < \alpha_1 < \ldots < \alpha_{k-1} < \alpha_k = m$ for all $m \le n$. The weight function is said to be *concave* if for all $0 \le i_0 \le i_1 < j_0 \le j_1 \le n$,

$$w(i_0, j_0) + w(i_1, j_1) \le w(i_0, j_1) + w(i_1, j_0). \tag{15.2}$$

The inequality (15.2) is also called the *quadrangle inequality* [17].

The LWS problem with the restriction that the weight function is concave is called the *Concave Least Weight Subsequence* (CLWS) problem. Hirschberg and Larmore [8] show that the LWS problem can be solved in $O(n^2)$ sequential time, while the CLWS problem can be solved in $O(n \log n)$ time. Wilber [39] gives an $O(n)$-time algorithm for the CLWS problem.

In the parallel setting, the CLWS problem seems to be more difficult. The best current polylogarithmic time algorithm for the CLWS problem uses concave matrix multiplication techniques and requires $O(\log^2 n)$ time with $n^2 / \log^2 n$ processors.

Larmore and Przytycka [40] have shown how to compute efficiently CLWS in sublinear time with the total work smaller than quadratic. Using this approach they showed the following fact (which has been later slightly improved [41,42].

Theorem 15.13

Optimal Huffman tree can be computed in $O(\sqrt{n} \log n)$ time with linear number of processors.

Karpinski and Nekrich have shown an efficient parallel algorithm which *approximates* optimal Huffman code, see [43].

Similar, but much more complicated algorithm works for alphabetic trees. Again the CLWS algorithm is the main tool.

Theorem 15.14: [6]

Optimal alphabetic tree can be constructed in polylogarithmic time with quadratic number of processors.

In case of general binary search trees the situation is more difficult. Polylogarithmic time algorithms need huge number of processors. However sublinear parallel time is easier.

Theorem 15.15: [44,45]

The OBST problem can be solved in **(a)** polylogarithmic time with $O(n^6)$ processors, **(b)** in sublinear time and quadratic total work.

References

1. L. L. Larmore and D. S. Hirschberg, A fast algorithm for optimal length–limited Huffman codes, *Journal of the ACM* 37, 1990, 464–473.
2. A. M. Garsia and M. L. Wachs, A New algorithm for minimal binary search trees, *SIAM Journal of Computing* 6, 1977, 622–642.
3. T. C. Hu, A new proof of the T-C algorithm, *SIAM Journal of Applied Mathematics* 25, 1973, 83–94.
4. T. C. Hu and A. C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM Journal of Applied Mathematics* 21, 1971, 514–532.
5. P. Bradford, M. Golin, L. Larmore, W. Rytter, Optimal prefix-free codes for unequal letter costs and dynamic programming with the monge property, *Journal of Algorithms,* 42 (2), February 2002, 277–303.
6. L. L. Larmore, T. M. Przytycka, W. Rytter, "Parallel Construction of Optimal Alphabetic Trees," in *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, 1993, pp. 214–223.
7. W. Rytter, The space complexity of the unique decipherability problem, *IPL* 16 (4), 1983.
8. D. S. Hirschberg and L. L. Larmore, "The Least Weight Subsequence Problem," in *Proc. 26th IEEE Symp. on Foundations of Computer Science, Portland Oregon*, Oct. 1985, pp. 137–143. Reprinted in *SIAM Journal on Computing* 16, 1987, 628–638.
9. A. Aggarwal, B. Schieber, T. Tokuyama, Finding a Minimum-weight k-link path graphs with the concave Monge property and applications, *Discrete & Computational Geometry* 12, 1994, 263–280.
10. B. Schieber, Computing a Minimum Weight k-Link Path in Graphs with the Concave Monge Property. 204–222.
11. R. L. Milidiu and E. Laber, The warm-up algorithm: A Lagrangian construction of length limited Huffman codes, *SIAM J. Comput.* 30 (5), 2000, 1405–1426.
12. R. L. Milidi and E. S. Laber, Linear time recognition of optimal L-restricted prefix codes (extended abstract), *LATIN* 2000, 227–236.
13. R. L. Milidi and E. S. Laber, Bounding the inefficiency of length-restricted prefix codes, *Algorithmica* 31 (4), 2001, 513–529.
14. J. S. Vitter, Dynamic huffman coding, *ACM Trans. Math. Software* 15, June 1989, 158–167.
15. D. E. Knuth, *The Art of computer programming*, Addison–Wesley, 1973.
16. D. E. Knuth, Optimum binary search trees, *Acta Informatica* 1, 1971, 14–25.
17. F. F. Yao, "Efficient Dynamic Programming Using Quadrangle Inequalities," in *Proceedings of the 12th ACM Symposium on Theory of Computing*, 1980, pp. 429–435.
18. K. Mehlhorn, *Data structures and algorithms*, vol. 1, Springer, 1984.
19. M. Karpinski, L. Larmore, W. Rytter, "Sequential and parallel subquadratic work constructions of approximately optimal binary search trees," in *the 7th ACM Symposium on Discrete Algorithms, SODA'96.*
20. M. M. Klawe and B. Mumey, "Upper and Lower Bounds on Constructing Alphabetic Binary Trees," in *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, 1993, pp. 185–193.
21. L. L. Larmore and T. M. Przytycka, "The Optimal Alphabetic Tree Problem Revisited," in *Proceedings of the 21st International Colloquium, ICALP'94, Jerusalem, LNCS 820*, Springer-Verlag, 1994, pp. 251–262.
22. P. Ramanan, Testing the optimality of alphabetic trees, *Theoretical Computer Science* 93, 1992, 279–301.
23. J. H. Kingston, A new proof of the Garsia-Wachs algorithm, *Journal of Algorithms* 9, 1988, 129–136.
24. M. Karpinski, L. l. Larmore, W. Rytter, Correctness of constructing optimal alphabetic trees revisited, *Theoretical Computer Science,* 180 (1–2), 10 June 1997, 309–324.
25. J. Abrahams, "Code and Parse Trees for Lossless Source Encoding," in *Sequences '97*, 1997.

Q2

Q3

26. M. Golin and G. Rote, "A Dynamic Programming Algorithm for Constructing Optimal Prefix-Free Codes for Unequal Letter Costs," in *Proceedings of the 22nd International Colloquium on Automata Languages and Programming (ICALP '95)*, July 1995, pp. 256–267.

27. R. M. Karp, Minimum-redundancy coding for the discrete noiseless channel, *IRE Transactions on Information Theory* 7, 1961, 27–39.

28. D. Altenkamp and K. Mehlhorn, Codes: Unequal probabilities unequal letter costs, *Journal of the Association for Computing Machinery* 27 (3), July 1980, 412–427.

29. S. Kapoor and E. Reingold, Optimum lopsided binary trees, *Journal of the Association for Computing Machinery* 36 (3), July 1989, 573–590.

30. S. A. Savari, Some notes on varn coding, *IEEE Transactions on Information Theory,* 40 (1), Jan. 1994, 181–186.

31. N. Cot, "A Linear-Time Ordering Procedure with Applications to Variable Length Encoding," in *Proc. 8th Annual Princeton Conference on Information Sciences and Systems*, 1974, pp. 460–463.

32. Y. Perl, M. R. Garey and S. Even, Efficient generation of optimal prefix code: Equiprobable words using unequal cost letters, *Journal of the Association for Computing Machinery* 22 (2), April 1975, 202–214.

33. S.-N. Choi, M. Golin, "Lopsided trees: Algorithms, analyses and applications," Automata, Languages and Programming, *Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming (ICALP 96)*.

34. E. N. Gilbert, Coding with digits of unequal costs, *IEEE Transactions on Information Theory*, 41, 1995,

35. A. Aggarwal, M. Klawe, S. Moran, P. Shor, R. Wilber, Geometric applications of a matrix-searching algorithm, *Algorithmica* 2, 1987, 195–208.

36. M. Golin and G. Rote, "A Dynamic Programming Algorithm for Constructing Optimal Prefix-Free Codes for Uneq ual Letter Costs," in *Proceedings of the 22nd International Colloquium on Automata Languages and Programming (ICALP '95)*, July 1995, pp. 256–267.Expanded version to appear in *IEEE Transactions on Information Theory*.

37. A. Gibbons, W. Rytter, *Efficient parallel algorithms*, Cambridge Univ. Press, 1997.

38. M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, S-H. Teng, "Constructing Trees in Parallel," in *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 499–533.

39. R. Wilber, The Concave least weight subsequence problem revisited, *Journal of Algorithms* 9, 1988, 418–425.

40. L. L. Larmore and T. M. Przytycka, Constructing Huffman trees in parallel, *SIAM Journal on Computing* 24 (6), 1995, 1163–1169.

41. M. Karpinski, L. Larmore, Y. Nekrich. A work efficient algorithm for the construction of length-limited Huffman codes, to appear in Parallel Processing Letters.

42. C. Levcopulos and T. Przytycka, A work-time trade-off in parallel computation of Huffman trees and concave least weight subsequence problem, *Parallel Processing Letters* 4 (1–2), 1994, 37–43.

43. P. Berman, M. Karpinski, M. Nekrich, "Approximating Huffman codes in parallel," in *Proc. 29th ICALP, LNCS vol. 2380*, Springer, 2002, pp. 845–855.

44. M. Karpinski and W. Rytter, On a Sublinear time parallel construction of optimal binary search trees, *Parallel Processing Letters* 8 (3), 1998.

Q4   45. Fast parallel computations for some dynamic programming problems, Theoretical Computer Science, 1988.

Q5   46. D. A. Huffman, A Method for the constructing of minimum redundancy codes, *Proc. IRE* 40, 1952, 1098–1101.

47. R. Güttler, K. Mehlhorn, W. Schneider, Binary search trees: Average and worst case behavior, *Electron. Informationsverarb Kybernet,* 16, 1980, 41–61.

48. D. G. Kirkpatrick and T. M. Przytycka, "Parallel construction of binary trees with almost optimal weighted path length," in *Proc. 2nd Symp. on Parallel Algorithms and Architectures*, 1990.

49. D. G. Kirkpatrick and T. M. Przytycka, "An Optimal Parallel Minimax Tree Algorithm," in *Proc. 2nd IEEE Symp. of Parallel and Distributed Processing*, 1990, pp. 293–300.

50. L. L. Larmore and T. M. Przytycka, "Parallel Construction of Trees with Optimal Weighted Path Length," in *Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures*, 1991, pp. 71–80.

51. L. Larmore and W. Rytter, Optimal parallel algorithms for some dynamic programming problems, *IPL* 52, 1994, 31–34.

52. W. Rytter, Efficient parallel computations for some dynamic programming problems, *Theoretical Computer Science* 59, 1988, 297–307.