ELSEVIER Discrete Applied Mathematics 91 (1999) 305–311

DISCRETE APPLIED MATHEMATICS

Note

Optimal binary trees with order constraints

András Sebő[a,*,1], Zeev Waksman[b]

[a] *CNRS, Laboratoire Leibniz-IMAG, 46, Avenue Félix Viallet, F-38000 Grenoble, France*
[b] *Department of Mathematics and Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105 Israel*

Received 17 April 1997; received in revised form 3 March 1998; accepted 10 August 1998

Abstract

Given a sequence of numbers $a_1,\dots,a_q$, find a binary tree with q leaves minimizing $\max\{h_1+a_1,\dots,h_q+a_q\}$, where h_i is the distance from the ith leaf to the root, $i=1,\dots,q$. This problem is solved by means of a $O(q)$ algorithm and a tight upper bound for the minimum is given by an explicit formula. The task is equivalent to finding a binary tree of minimum height having q subtrees of heights $a_1,\dots,a_q$ whose leaves partition the leaves of the tree. This question seems to be of general interest. In particular, it arises in the problem of the optimal decomposition of a tree into chains (Waksman, Tech. Report FC 95-06, August 1995). © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Binary tree; Optimal decomposition

1. Introduction

A *binary tree* is a tree $T=(V,E)$ with a root $r\in V$ where, by orienting the edges so that every node is reachable from r, the outdegree of every node is 2 or 0; so the degree of the root is 2 (or 0 if $E=\emptyset$) and all other nodes are of degree 1 or 3. Nodes of T whose degree is 1 are called *leaves*. Usually, one of the two outgoing edges is assigned to be the *left* and the second to be the *right*; leaving a node u, they enter the nodes called $l(u)$ and $r(u)$. This introduces, in particular, a linear *left to right* order on the leaves. The *height* $h(v)$ of a node v is its distance (number of edges in the path) from the root. The *height* $h(T)$ of a tree T is the maximum height of its leaves. A tree is *complete* if all its leaves have the same height. Clearly, a complete binary tree of height h has 2^h leaves.

* Corresponding author. E-mail: andras.sebo@imag.fr.

[1] My friend and coauthor Zeev (BOBA) Waksman died in November 1997. Our article also reminds me of his lively personality and the pleasant time I had the opportunity of spending with him.

The number of leaves in T will be denoted by $\|T\|$. The quantity (clearly non-negative)

$$ex(T) := h(T) - \lceil \log_2 \|T\| \rceil \tag{1}$$

is called the *excess* of the tree T. Leaves of a tree T are refered here as $v_1, \dots, v_q$, $q := \|T\|$, and their heights as h_i: $h_i := h(v_i)$. We treat the leaves in this list in the left to right order.

We present here an algorithm solving the following problem: given a vector $\boldsymbol{a} := (a_1, \dots, a_q)$, find a binary tree T with q leaves so that $T(\boldsymbol{a}) := \max\{h_1 + a_1, \dots, h_q + a_q\}$ is minimum. The components of $\boldsymbol{a}$ are also indexed from left to right. If renumbering of the components is forbidden – which can be called the *order constraint* – this minimum will be denoted by $f(\boldsymbol{a})$. If renumbering is allowed (that is each one of all q permutations of the numbers $a_1, \dots, a_q$ – with corresponding renumbering – is permissible), we denote it by $g(\boldsymbol{a})$. We will refer to these two versions of the problem as "f-version" and "g-version". The computation of $g(\boldsymbol{a})$ will be straightforward, whereas the computation and estimation of $f(\boldsymbol{a})$ is the main result of this note.

The following problem has been investigated in a series of papers. Given m functions $f_i(t)$, $i = 1, 2, \dots, m$, $t \in \mathbb{N} := \{0, 1, 2, \dots\}$, each f_i being nondecreasing and "concave", that is the sequence of its increments is nonnegative and nonincreasing:

$$0 \leqslant f_i(t+1) - f_i(t) \leqslant f_i(s+1) - f_i(s) \quad \text{if } t > s, \quad s, t \in \mathbb{N},$$

maximize $\sum_{i=1}^m f_i(x_i)$ under the constraint $\sum_{i=1}^m x_i \leqslant d$. Supposing any value of any function f_i is to be accessible for a constant amount of time (direct access), we have the problem of "optimum distribution of effort", (see [4]). The best known algorithm for this problem is given in [2]. On the other hand, if each of the functions f_i is represented by its *sequentially* accessed sequence of the increments, that is the effort of getting the value of $f_i(t)$ is proportional to t, we have the classical m-way merging problem. The selection tree algorithm in [8] solves it with a $O(m + d \log m)$ effort (in other words, with a $\log m$ per unit effort).

Starting from [9] and until recently [1], certain attention has been paid to the direct access version where the single constraint above is replaced by a set of inequalities $\sum_{i \in H} x_i \leqslant d_H$, $\forall H \in \mathscr{H}$, where $\mathscr{H} \subseteq 2^{\{1, \dots, m\}}$ is defined as follows: if $H_1, H_2 \in \mathscr{H}$, then $H_1 \cap H_2$ is either empty or one of H_1, H_2. In many papers (notably in [1]), the name *nested* (introduced in [9]) is used for such a family, while Hochbaum [5] uses a more apt term *tree family* preserving the name *nested* only for the special case $H_1 \subset H_2 \subset \cdots$ (called in [11] the *chain* case).

The sequential access version of the problem under the tree constraints was considered in [5] and, later, in [11]. Both have obtained (though by completely different means) the asymptotically best ($O(\log m)$-per-unit) effort estimate. In [5], an $O(1)$-per-unit-in-average algorithm is proposed for the nested (chain) case, which exploits heavily the results of [3], and an $O(\log m)$-per-unit algorithm for the tree case, which is based on [10]. In [11], the nested problem is solved by a straightforward $O(1)$-per-unit algorithm and the tree case is reduced to it through an optimal (in a sense) decomposition of trees to chains. The motivation for the present work has been provided by this optimal decomposition problem, which seems to be also of general interest. In particular, Corollary 2.2 below is crucial for the log-per-unit estimate in the tree problem.

2. Excess of a list of trees and its upper bound

Let binary trees $T_1,\dots,T_q$ with roots $r_1,\dots,r_q$ be given. We associate with every binary tree T with q leaves $v_1,\dots,v_q$ another tree, say T^*, by *hanging* $T_1,\dots,T_q$ on its leaves meaning that we identify v_i and r_i for all $i=1,\dots,q$. We want to find a tree T minimizing $h(T^*)$. Clearly, nothing but the heights $(a_1,\dots,a_q)=:\boldsymbol{a}$ of the trees $T_1,\dots,T_q$ is important: it is clear that the minimum height of T^* is $f(\boldsymbol{a})$ or $g(\boldsymbol{a})$ depending on the version we are considering. Since $\|T^*\|$ is equal to $\|T_1\|+\cdots+\|T_q\|$ independently of the choice of the tree $T, h(T^*)-ex(T^*)$ is a constant, and $h(T^*)$ and $ex(T^*)$ are minimized for the same trees T^*. We denote the minimum excess of T^* by $fex(T_1,\dots,T_q)$ or $gex(T_1,\dots,T_q)$ depending on the version of the problem. So,

$$\begin{aligned} fex(T_1,\dots,T_q)&=f(\boldsymbol{a})-\lceil\log_2(\|T_1\|+\cdots+\|T_q\|)\rceil,\\ gex(T_1,\dots,T_q)&=g(\boldsymbol{a})-\lceil\log_2(\|T_1\|+\cdots+\|T_q\|)\rceil. \end{aligned} \tag{2}$$

The trees providing minimum to the functions f and fex (g and gex) are referred here to as *optimal* or *f-optimal* (*g-optimal*, respectively).

Let us define the function

$$L:=L(\boldsymbol{a}):=\lceil\log_2(2^{a_1}+\cdots+2^{a_q})\rceil. \tag{3}$$

It is easy to check that

$$f(\boldsymbol{a})\geqslant g(\boldsymbol{a})\geqslant L(\boldsymbol{a}) \tag{4}$$

for the optima in question. Indeed, the inequality $f(\boldsymbol{a})\geqslant g(\boldsymbol{a})$ is clear from the definitions. Suppose a tree T with leaves $v_1,\dots,v_q$, $h_i:=h(v_i)$, is g-optimal, that is, it satisfies (after suitable renumbering of the subtrees T_i from left to right) the equality $g(\boldsymbol{a})=h(T^*)=\max\{h_1+a_1,\dots,h_q+a_q\}$. Expand T^* to a complete binary tree T' of height $g(\boldsymbol{a})$. Denoting by T_i' the (complete) subtree of T' rooted in v_i, $i=1,\dots,q$, we obviously have $h(T_i')\geqslant a_i$ so that, $\|T_i'\|\geqslant 2^{a_i}$, $\|T'\|=\sum\|T_i'\|\geqslant\sum 2^{a_i}$, and

$$g(\boldsymbol{a})=h(T^*)=h(T')=\log_2\|T'\|\geqslant\lceil\log_2(2^{a_1}+\cdots+2^{a_q})\rceil=L(\boldsymbol{a}).$$

In this note we give a simple linear (in q) algorithm constructing an optimal tree and an "almost" explicit formula for $f(\boldsymbol{a})$ (and $g(\boldsymbol{a})$) stated as follows:

Theorem 2.1. *For all $\boldsymbol{a}\in\mathbb{N}^q$ ($q\in\mathbb{N}$),*

$$f(\boldsymbol{a})=L(\boldsymbol{a}) \text{ or } L(\boldsymbol{a})+1, \qquad g(\boldsymbol{a})=L(\boldsymbol{a}). \tag{5}$$

The proof of the theorem will be given below. The following Corollary translates the claims of Theorem 2.1 into upper bounds for fex and gex in terms of $ex(T_i)$, $i=1,\dots,q$. These bounds play an important role in [11].

Corollary 2.2. *For arbitrary binary trees $T_1,\dots,T_q$ ($q \in \mathbb{N}$),*

$$fex(T_1,\dots,T_q) \leqslant 2 + \max_{1\leqslant i\leqslant q} ex(T_i), \qquad gex(T_1,\dots,T_q) \leqslant 1 + \max_{1\leqslant i\leqslant q} ex(T_i) \tag{6}$$

and the bounds are sharp; they are also sharp in the following particular case:

$$ex(T_1) = \cdots = ex(T_q) = 0 \text{ implies } fex(T_1,\dots,T_q) \leqslant 2 \text{ and } gex(T_1,\dots,T_q) \leqslant 1. \tag{7}$$

Proof. By definition (1) and remembering that $a_i := h(T_i)$, we have

$$a_i = \lceil \log_2 \|T_i\| \rceil + ex(T_i) \leqslant 1 + \log_2 \|T_i\| + ex(T_i) \leqslant 1 + \log_2 \|T_i\| + \max_{1\leqslant i\leqslant q} ex(T_i).$$

Taking the corresponding power of two and summing these inequalities for $i=1,\dots,q$, yields $\sum 2^{a_i} \leqslant 2^{1+\max ex(T_i)}(\|T_1\| + \cdots + \|T_q\|)$ or, by taking the logarithm, $L(\boldsymbol{a}) \leqslant 1 + \max ex(T_i) + \lceil \log_2 \sum \|T_i\| \rceil$. Using now $fex(T_1,\dots,T_q) \leqslant 1 + L(\boldsymbol{a}) - \lceil \log_2 \sum \|T_i\| \rceil$, which follows from (2) and from the first claim of Theorem, we get the first bound in (6). The second is now obvious. $\square$

3. Algorithm and bounds

Each time a vector $\boldsymbol{a} := (a_1,\dots,a_q)$ is given, we add, for convenience, $a_0 := a_{q+1} := \infty$.

Among the triples a_{i-1}, a_i, a_{i+1} ($i=1,\dots,q$) there is always at least one with the property that not all three elements are equal and a_i is the smallest. Two situations are possible for such a triple: either

(A) a_i is the only smallest element of the triple, or

(B) there are two smallest elements in the triple.

The following lemma paves the way for the algorithm.

Lemma 3.1. *Suppose $q \geqslant 2$.*

1° *If* (A) *holds, then $f(\boldsymbol{a}) = f(\boldsymbol{a}')$, where $\boldsymbol{a}' := (a_0,\dots a_{i-1}, a_i + 1, a_{i+1},\dots a_{q+1})$, and the same trees are f-optimal for $\boldsymbol{a}$ and for $\boldsymbol{a}'$.*

2° *If* (B) *holds, so that, say, $a_{i-1} > a_i = a_{i+1}$, then we have $f(\boldsymbol{a}) = f(\tilde{\boldsymbol{a}})$, where $\tilde{\boldsymbol{a}} := (a_0,\dots,a_{i-1}, a_i + 1, a_{i+2},\dots,a_{q+1})$, and if $\tilde{T}$ is an f-optimal tree for $\tilde{\boldsymbol{a}}$, $\|\tilde{T}\| = q-1$, then the tree obtained from $\tilde{T}$ by adding two sons to the ith leaf of $\tilde{T}$ is f-optimal for $\boldsymbol{a}$.*

Proof. Let (A) hold and let T be an f-optimal tree for $\boldsymbol{a}$, that is, $T(\boldsymbol{a}) = f(\boldsymbol{a})$, and, say, $v_i = r(u)$. Then v_{i-1} is a descendant of u, so that $h_{i-1} \geqslant h_i$, and consequently $h_{i-1} + a_{i-1} > h_i + a_i$. $T(\boldsymbol{a}') = T(\boldsymbol{a})$ follows, that is, $f(\boldsymbol{a}') \leqslant T(\boldsymbol{a}') = T(\boldsymbol{a}) = f(\boldsymbol{a}) \leqslant f(\boldsymbol{a}')$, and the equality follows throughout.

Now let the condition in (B) hold. If

$$v_i = l(u) \quad \text{and} \quad v_{i+1} = r(u), \tag{8}$$

i.e. v_i and v_{i+1} are the left and the right sons of u, then we define a new tree $\tilde{T}$ by deleting v_i and v_{i+1} from T, so that u becomes a leaf. Clearly, $\tilde{T}$ is an f-optimal tree for $\tilde{\boldsymbol{a}}$.

We have to show that a tree T, f-optimal for $\boldsymbol{a}$ and satisfying (8), always exists. Let $v_i = l(u)$, so that v_{i+1} is the leftmost node in the subtree rooted in $w := r(u)$. To get the promised tree, rearrange T as follows: delete v_i, identify w with u and equip v_{i+1} with two sons (which become now new v_i and v_{i+1}). If $v_i = r(u)$, then $h_{i-1} \geqslant h_i$. Choose from the leaves v_i, v_{i+1} one with lesser value of h. Let it be v_i, that is, $h_i \leqslant h_{i+1}$. This time delete v_{i+1} from the tree T and add to v_i two new sons. □

The lemma permits, by systematically detecting triples satisfying (A) or (B) to build a sequence $\boldsymbol{a}^1 := \boldsymbol{a}, \boldsymbol{a}^2, \ldots$ with $f(\boldsymbol{a}^1) = f(\boldsymbol{a}^2) = \cdots$ and with decreasing value of q. When $q = 1$, $f(\boldsymbol{a})$ is found and a tree f-optimal for $\boldsymbol{a}$ is constructed.

An efficient way of finding a triple satisfying (A) or (B) is to find the smallest i with $a_i \leqslant a_{i+1}$. ($1 \leqslant i \leqslant q$, of course.) If $a_i < a_{i+1}$ then (A) holds, whereas in case $a_i = a_{i+1}$ (B) is true. To translate Lemma 3.1 into an algorithm of complexity $O(q)$, one has only to assign in case (A) the value $\min\{a_{i-1}, a_{i+1}\}$ to a_i.

The situation in the g-version is much simpler: let $a_1 \leqslant a_2$ be the two smallest coordinates of $\boldsymbol{a}$. Now depending on whether $a_1 < a_2$ or $a_1 = a_2$ we can apply 1° or 2° (to $i = 1$). Then do the same again. Paradoxically, despite the straightforwardness of the g-version, here the complexity of the problem will be $O(q \log q)$ instead of $O(q)$, because of the sorting we need at the start (we need to know the two smallest components of $\boldsymbol{a}$ in each step).

Algorithm

Input: $\boldsymbol{a} := (a_1, \ldots, a_q)$ (We keep the convention $a_0 := a_{q+1} := \infty$), $X \in \{0, 1\}$.
Output: A tree T with q leaves, f-optimal if $X = 0$ and g-optimal if $X = 1$.
Initialization: If $X = 1$, the execution below supposes that $a_1 \geqslant \cdots \geqslant a_q$. Hence, a sorting algorithm should be invoked if needed. After that, the algorithm works independently of X.

Let a forest T be initialized by q trees, each tree T_i is associated with the value a_i and consists only of the root v_i, $i = 1, \ldots, q$.

Start: If $q = 1$, then STOP.

1. Find the smallest i with $a_i \leqslant a_{i+1}$ (clearly, $1 \leqslant i \leqslant q$).

2. If $a_i < a_{i+1}$,
assign the value $\min\{a_{i-1}, a_{i+1}\}$ to a_i (this minimum is always finite since $q \geqslant 2$). Leave the forest and the value of q unchanged. GO TO **1**.

3. If $a_i = a_{i+1}$,
update $q, \boldsymbol{a}$, and the forest as follows. Reduce q by one. Replace the component a_i in $\boldsymbol{a}$ by $a_i + 1$ and delete a_{i+1}. Build the tree with T_i and T_{i+1} being the left and the right subtrees of the root, assign it to T_i and delete T_{i+1}. Renumerate the rest of trees

and components of $\boldsymbol{a}$, that is, assign $T_j := T_{j+1}$ and $a_j := a_{j+1}$ for $j = i+1, \dots, q$. GO TO START.

It is immediate that in the f-version the execution follows Lemma 3.1, so that after termination the forest becomes an f-optimal tree and the only component of $\boldsymbol{a}$ is equal to $f(\boldsymbol{a})$. The same holds in the g-version, which may be shown in the same way as in Lemma 3.1. It means, in fact, that sorting is always an optimal renumeration in the g-version.

Clearly, once $\boldsymbol{a}$ has been sorted the overall execution of algorithm needs $O(q)$ time. However, the sorting needed for initialization in the g-version requires $O(q \log q)$ time.

Lemma 3.1 and its algorithmic proof are quite natural for solving the problem in question. The most tricky single step of this note is the following:

Proof of Theorem 2.1. The inequality $L(\boldsymbol{a}) \leqslant f(\boldsymbol{a})$ is contained in (4). The inequality $f(\boldsymbol{a}) \leqslant L(\boldsymbol{a}) + 1$ seems to be less trivial. It is equivalent to $2^{f(\boldsymbol{a})} < 4 \times 2^{L(\boldsymbol{a})}$, and we prove a slightly sharper bound for $q \geqslant 2$: $2^{f(\boldsymbol{a})} \leqslant M := 2\sum_{i=1}^{q-1} \max\{2^{a_i}, 2^{a_{i+1}}\}$. First, this inequality is obviously valid if $q = 2$. Second, during the execution of the algorithm, $f(\boldsymbol{a})$ does not change whereas M remains unchanged if (A) occurs and cannot increase if (B) occurs. So, the bound follows by induction.

The equality $g(\boldsymbol{a}) = L(\boldsymbol{a})$ follows in a similar way: since $a_1 \geqslant \cdots \geqslant a_q$, it is easy to see that both $g(\boldsymbol{a})$ and $L(\boldsymbol{a})$ here are unchanged during the execution and are equal at the end. □

In conclusion, we remark that the problem under discussion belongs to the following, apparently not yet identified, class of problems. Let a vector $\boldsymbol{a} = (a_1, \dots, a_q)$ be given and a binary operation, say $x \circ y$, be specified. It defines a function, say $\boldsymbol{a}_T$, on the node set of any binary tree T with $\|T\| = q$ as follows: $\boldsymbol{a}_T(v_i) := a_i$, where v_i is the ith leaf of T, and $\boldsymbol{a}_T(u) := \boldsymbol{a}_T(l(u)) \circ \boldsymbol{a}_T(r(u))$ for any inner node u. Specifying in addition a functional Φ over the set of such functions, we have the following minimization problem (in both f- and g-versions): find $\min \Phi(\boldsymbol{a}_T)$ over all trees T with $\|T\| = q$. For $x \circ y := 1 + \max\{x, y\}$ and $\Phi(\boldsymbol{a}_T)$ being the value of $\boldsymbol{a}_T$ at the root of T, we have the problem above. For $x \circ y := x + y$ and $\Phi(\boldsymbol{a}_T)$ equals to the sum of $\boldsymbol{a}_T(u)$ over all nodes u of T, we have – in g-version – the Huffman's problem of building a minimum redundancy code (see [7]). The f-version of this problem is solved in [6] with an $O(q \log q)$ algorithm. These two examples provoke various questions in the general case. For example, the characterization of $(\circ, \Phi)$ with a greedy solution algorithm or an $O(q)$ algorithm.

References

[1] M.E. Dyer, A.M. Frieze, On an optimization problem with nested constraints, Discrete Appl. Math. 26 (1990) 159–173.

[2] G.N. Frederickson, D.B. Johnson, The complexity of selection and ranking in $X + Y$ and matrices with sorted columns, J. Comput. System Sci. 24 (1982) 197–208.

[3] H.N. Gabow, R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J. Comput. System Sci. 30 (1985) 209–221.
[4] Z. Galil, M. Meggido, A fast selection algorithm and the problem of optimum distribution of effort, J. ACM 26 (1979) 58–64.
[5] D.S. Hochbaum, Lower and upper bounds for the allocation problem and other nonlinear optimization problems, Math. Oper. Res. 19 (1994) 390–409.
[6] T.C. Hu, A.C. Tucker, Optimal computer search trees and variable-length alphabetical codes, SIAM J. Appl. Math. 21 (1971) 514–532.
[7] D.A. Huffman, A method for the construction of minimum redundancy codes, Proc. Inst. Radio Engrs. 40 (1952) 1098–1101.
[8] D.E. Knuth, The Art of Computer Programming, Addison-Wesley, Reading, MA, 1973.
[9] A. Tamir, Further remarks on selection problems with nested constraints, Department of Statistics Report, Tel-Aviv University, Tel-Aviv, 1979.
[10] R.E. Tarjan, Application of path compression on balanced trees, J. Assoc. Comp. Mash. (1979) 690–715.
[11] Z. Waksman, More on optimization under nested constraints, Ben-Gurion University of the Negev, Tech. Report FC 95-06, August 1995.