1-1-1986

Alphabetic Minimax Trees of Degree at Most t*

D. Coppersmith
IBM Thomas J. Watson Research Center

Maria M. Klawe
Harvey Mudd College

Nicholas Pippenger
Harvey Mudd College

SIAM J. COMPUT.
Vol. 15, No. 1 February 1986

ALPHABETIC MINIMAX TREES OF DEGREE AT MOST t[*]

D. COPPERSMITH[†], M. M. KLAWE[‡] AND N. J. PIPPENGER[‡]

Abstract. Problems in circuit fan-out reduction motivate the study of constructing various types of weighted trees that are optimal with respect to maximum weighted path length. An upper bound on the maximum weighted path length and an efficient construction algorithm will be presented for trees of degree at most t, along with their implications for circuit fan-out reduction.

Key words. optimal weighted tree, minimax tree, t-ary tree, fanout reduction, logical circuits

In this paper we consider the problem of constructing, for any list $w_1, \cdots, w_n$ of integers, a tree T with maximum degree at most t (where $t \geqq 2$ is a fixed integer) and leaves $v_1, \cdots, v_n$ in left to right order such that $f_T(w_1, \cdots, w_n) = \max_{1 \leqq i \leqq n} (l_i + w_i)$ is minimized, where l_i denotes the length of the path in T from the root to the leaf v_i. We will call the minimum value $f(w_1, \cdots, w_n) = \min_T f_T(w_1, \cdots, w_n)$ the *minimax weighted path length.*

This work was motivated by the results of Kirkpatrick and Klawe [2] dealing with the analogous problem of constructing t-ary trees, that is, trees in which the degree of every internal vertex is exactly t. As in [2], we obtain a linear algorithm for the case of integer weights and prove a tight upper bound on $f(w_1, \cdots, w_n)$ in terms of $w_1, \cdots, w_n$. Like those in [2], these results can be applied to obtain a circuit fan-out reduction algorithm that preserves size and depth to within constant multiplicative factors without increasing the number of edge crossings. Our relaxation of the constraint on the degrees of internal vertices in the tree results in a smaller multiplicative factor for depth, but a larger multiplicative factor for size. This relaxation also causes some of the proofs to be easier than those in [2]; indeed the ideas in this paper inspired simplifications of both the algorithm and the proof of the upper bound in [2]. Kirkpatrick and Klawe show that an $O(n \log n)$ algorithm for real weights can be obtained from their linear integral weight algorithm, and that the upper bound also applies to the case of real weights. The same methods could be applied to our results to yield analogous results for the case of real weights.

If the leaves of a tree are weighted, we can extend the weighting to the internal vertices of the tree by defining the weight of an internal vertex to be one plus the maximum of the weights of its sons. With this extension, if the leaves $v_1, \cdots, v_n$ have weights $w_1, \cdots, w_n$, then the weight of the root is exactly $f_T(w_1, \cdots, w_n)$. This yields an equivalent formulation of our problem as that of constructing a tree with maximum degree at most t with leaf weights $w_1, \cdots, w_n$ in left to right order such that the weight of the root is minimized. The next lemma gives three modifications which can be made to a list of weights without increasing the minimax weighted path length.

LEMMA 1. *If $w_1, \cdots, w_n$ is a list of weights, then none of the following modifications increase the minimax weighted path length. Define $w_0 = w_{n+1} = \infty$.*

(a) *If $n > 1$ and $w_i \leqq \min(w_{i-1}, w_{i+1})$ for some i with $1 \leqq i \leqq n$, then replace w_i by* $\min(w_{i-1}, w_{i+1})$.

(b) *If* $\min(w_i, w_{i+s+1}) \geqq 1 + \max(w_{i+1}, \cdots, w_{i+s})$ *for some $s \leqq t$ and i with $0 \leqq i \leqq n-s$, then replace the s weights $w_{i+1}, \cdots, w_{i+s}$ by the single weight $1+$* $\max(w_{i+1}, \cdots, w_{i+s})$.

[*] Received by the editors May 2, 1983, and in revised form May 21, 1984.

[†] IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

[‡] IBM Research Laboratory, San Jose, California 95193.

(c) *If* $w_i = w_{i+1} = \cdots = w_{i+t-1} \leqq w_{i+t} - 1$ *for some* i *with* $1 \leqq i \leqq n - t + 1$, *then replace the* t *weights* $w_i, \cdots, w_{i+t-1}$ *by the single weight* $1 + w_{i+t-1}$.

Proof. In each case the proof consists of indicating how an optimal tree for the original list of weights can be altered to obtain a tree for the modified list in such a way that the weight of the root is not increased. Let T be a tree that is optimal for $w_1, \cdots, w_n$. In case (a), v_i must have v_{i-1}, v_{i+1} or one of their ancestors as a brother, so that increasing the weight of v_i cannot increase the weight of its father, and hence cannot increase the weight of the root. In case (b), choose a vertex x in T such that all the leaves in the subtree rooted at x are in the set $\{v_{i+1}, \cdots, v_{i+s}\}$ and such that the distance from x to the root is minimal. Note that x must have v_i, v_{i+s+1} or one of their ancestors as a brother. Thus replacing the subtree rooted at x by a single vertex with weight $1 + \max(w_{i+1}, \cdots, w_{i+s})$ and removing any leaves in $\{v_{i+1}, \cdots, v_{i+s}\}$ that are outside the subtree rooted at x cannot increase the weight of the root. Finally, in case (c), there are two possibilities. The first is that the brothers of v_{i+t-1} are precisely $\{v_{i+j}, \cdots, v_{i+t-2}\}$ for some j with $0 \leqq j \leqq t-2$. In this case the change corresponds to replacing the weights of $v_{i+j}, \cdots, v_{i+t-1}$ by the weight of their father which has weight $1 + w_{i+t-1}$, and removing the other leaves $v_i, \cdots, v_{i+j-1}$. In the second possibility v_{i+t-1} has as a brother v_{i+t}, an ancestor of v_{i+t}, or an ancestor of v_{i+j} for some j with $0 \leqq j \leqq t-2$. Thus removing the leaves v_{i+j} for $0 \leqq j \leqq t-2$ and increasing w_{i+t-1} by 1 does not increase the weight of the root. □

We now sketch an algorithm that, given a list $w_1, \cdots, w_n$ of integer weights, constructs an optimal tree. First add dummy weights $w_0 = w_{n+1} = \infty$ to each end of the list. At any stage of execution there will be a list of weights of vertices that have not yet been assigned fathers and a pointer dividing the list into two parts, the left sublist and the right sublist. The algorithm will operate so that the left sublist always forms a nonincreasing sequence from left to right. We call a weight K in the left sublist a *step weight* if K is strictly larger than the weight on its right or if K is the rightmost weight in the left sublist. Initially the pointer is placed so that it points between w_0 and w_1. We now describe the main procedure of the algorithm. Suppose the weights lying immediately to the left and right of the pointer are L and R respectively. If $L \geqq R$, then the algorithm simply moves the pointer past R. Otherwise, let K be the rightmost step weight such that either $K \geqq R$ or there are at least t weights lying strictly between K and R in the list.

First suppose that there are at least t weights between K and R. If at least two of these weights are step weights, find the leftmost such step weight, say K', remove all weights lying between K' and R and insert a new weight equal to K' between K' and the pointer. (We rely here on (b) followed by (a) in Lemma 1 and on the fact that all weights are integers.) If L is the only step weight, remove the t rightmost weights to the left of the pointer and insert a new weight equal to $L+1$ to the right of the pointer. (We rely here on (c) in Lemma 1.) Now suppose that there are less than t weights between K and R, and hence that $K \geqq R$. Remove all weights between K and R and insert a new weight equal to R to the left of the pointer. (We rely again on (b) followed by (a) in Lemma 1.) Note that after applying this procedure the weights to the left of the pointer still form a nonincreasing sequence.

The algorithm operates by repeating this procedure until exactly three weights are left in the list. As the ∞ weights are never removed, the final list is of the form ∞, w, ∞. Interpreting modifications of types (b) and (c) in the obvious manner of making the new weight the weight of the father of the vertices whose weights were removed from the list, it is clear that this algorithm constructs an optimal tree and that the minimax weighted path length is w.

To implement the algorithm efficiently, it is only necessary to maintain the position of the pointer and (in a doubly-linked list) the step weights and their positions in the left sublist. In any execution of the main procedure, all but the leftmost of the step weights examined will no longer be step weights at the end of the procedure. From this and by examining the other operations in the procedure it is easy to see that the running time of the algorithm is at most linear in the total number of vertices in the tree (which is at most $2n-1$), with a coefficient that is independent of t.

We now prove an upper bound on the minimax weighted path length for the case of integer weights.

LEMMA 2. *If $w_1, \cdots, w_n$ are integers, then*

$$f(w_1, \cdots, w_n) < 1 + \log_t 2 + \log_t \left(\sum_{1 \leqq i \leqq n} t^{(w_i)} \right).$$

Proof. For W the list of weights $w_1, \cdots, w_n$, define $g(W) = \sum_{1 \leqq i \leqq n-1} t^{\max(w_i, w_{i+1})}$. Then it is easy to verify that if W' is any list obtained by modifying W according to (a), (b) or (c) of Lemma 1, then $g(W') \leqq g(W)$. Suppose w is the weight of the root of the tree constructed by our algorithm and suppose the weights of the sons of the root are $x_1, \cdots, x_s$. Let X be the list $x_1, \cdots, x_s$. By iterating the observation above, $g(X) \leqq g(W)$. Combining this with the obvious inequalities $t^w \leqq tg(X)$ and $g(W) < 2\sum_{1 \leqq i \leqq n} t^{(w_i)}$ and taking logarithms completes the proof. □

Remark 3. The corresponding upper bound in [2] for t-ary trees is $2 + \log_t (\sum_{1 \leqq i \leqq n} t^{(w_i)})$.

We now describe the application to circuit fan-out reduction in more detail, in order to compare the effect of using various tree constructions. Suppose G is an acyclic directed graph with fan-in bounded by s. In [1], an algorithm is given that constructs a new graph G' with fan-out at most t, by replacing each vertex of G that has fan-out greater than t with a tree connecting that vertex to its sons. By choosing trees that minimize the increase in depth while having degrees bounded by t, it can be proved that $\text{Size}(G') \leqq (1 + (s-1)/(t-1)) \text{Size}(G) + (q-1)/(t-1)$ and $\text{Depth}(G') \leqq (1 + \log_t s) \text{Depth}(G) + \log_t q$, where q is the number of outputs of G. Unfortunately, however, using trees that minimize the increase in depth will generally increase the number of edge crossings.

In [2] it is observed that using alphabetic minimax trees avoids the increase in edge crossings in exchange for a poorer bound on the depth of the new graph. Thus, although the size bound remains the same, the depth bound becomes $\text{Depth}(G') < (2 + \log_t s) \text{Depth}(G) + \log_t q$. Finally, using the trees described in this paper also avoids the increase in edge crossings with a better depth bound than that of [2] but a poorer size bound. More precisely, if our algorithm is used, the bounds become $\text{Depth}(G') \leqq (1 + \log_t (2s)) \text{Depth}(G) + \log_t q$ and $\text{Size}(G') \leqq s \text{Size}(G) + q - 1$. We will see, however, that the size bound can be improved to $\text{Size}(G') \leqq (1 + (s-1)/(t-1) + (s+1)(t-2)/3(t-1)) \text{Size}(G) + (q+1)/(t-1) + (q+1)(t-2)/3(t-1)$, by adding an extra phase to our algorithm to reduce the size of the optimal tree.

Although our algorithm constructs an optimal tree, it does not necessarily construct the optimal tree with the smallest number of vertices. In the worst case, which occurs for sequences of the form $2j-1, 2j-3, \cdots, 3, 1, 2, 4, \cdots, 2j-2, 2j$, our tree has $n-1$ internal vertices, although there is an optimal tree with $\lceil (n-1)/(t-1) \rceil$ internal vertices. Although we have been unable to find a linear algorithm which produces the smallest optimal tree, by applying a simple linear "compaction" algorithm to our optimal tree we obtain a tree in which the number of internal vertices is at most $\lfloor (n-1)/(t-1) +$

$(t-2)(n+1)/3(t-1)\rfloor$. We will also give an example showing that there are sequences for which the smallest optimal tree has this many internal vertices.

A *leaflet* is defined to be an internal vertex that has only leaves as sons and has degree less than t. The object of the compaction algorithm is to produce a tree satisfying the following three conditions.

(1) Each internal vertex either has degree t or is a leaflet.

(2) No two adjacent leaves are the sons of different leaflets.

(3) Each leaflet has degree at least 2.

It is not hard to design a linear algorithm that accomplishes this. Condition (1) can be met by a phase that processes the vertices in preorder (or any other order that visits each vertex before its sons) and raises the degree of nonleaflet internal vertices by making grandsons into sons. Condition (2) can be met by a phase that, whenever two "offending" adjacent leaves are located, moves sons from the leaflet with smaller weight to the leaflet with larger weight until either the first leaflet has degree 1 and can be collapsed (i.e. replaced by its son), or the second has degree t and is no longer a leaflet. Finally, condition (3) can be met by collapsing leaflets with only one son.

We shall show that any tree satisfying the three conditions above has at most $\lfloor (n-1)/(t-1)+(t-2)(n+1)/3(t-1)\rfloor$ internal vertices.

LEMMA 4. *If T is tree with n leaves satisfying conditions* (1), (2) *and* (3), *then t has at most* $\lfloor (n-1)/(t-1)+(t-2)(n+1)/3(t-1)\rfloor$ *internal vertices.*

Proof. Let k be the number of leaflets and let p be the number of leaves that are sons of leaflets. Obviously, $p \geqq 2k$, by condition (3). Thus the number of internal vertices is at most $k+(n-k-1)/(t-1)$, since if we remove all leaves that are sons of leaflets from T, the remaining tree is a t-ary tree with $n-p+k$ leaves and so its number of internal vertices is exactly $(n-p+k-1)/(t-1)$. Finally, it is easy to see that $k \leqq (n+1)/3$, by conditions (2) and (3), which yields the stated bound. □

We conclude our paper with examples which show that even after compaction our optimal tree is not necessarily the smallest optimal tree, and also that there are lists of weights for which the number of internal vertices in the smallest optimal tree attains the bound in the preceding lemma. Let $W(n)$ be the list $w_1, \cdots, w_n$, where $w_i = 1$ for $i = 0 \bmod 3$ and $w_i = 0$ otherwise. For $n = 9$ and $t = 3$, the balanced ternary tree is optimal and has only four internal vertices. Our algorithm, however, begins by pairing the three pairs of 0 weights, and it can easily be checked that no matter how the compaction algorithm is implemented, the resulting compacted tree will have five internal vertices. On the other hand, if $n = \lceil 3t^k/2 \rceil$ for some $k \geqq 1$ and $t \geqq 3$, then there is only one optimal tree for $W(n)$, and it has $\lfloor (n-1)/(t-1)+(t-2)(n+1)/3(t-1)\rfloor$ internal vertices.

REFERENCES

[1] H. J. HOOVER, M. M. KLAWE AND N. J. PIPPENGER, *Bounding fan-out in logical networks*, J. Assoc. Comput. Mach., 31 (1984), pp. 13-18.

[2] D. G. KIRKPATRICK AND M. M. KLAWE, *Alphabetic minimax trees*, this Journal, 14 (1985), pp. 514-526.