A Subquadratic algorithm for constructing approximately optimal binary search trees

Lawrence L. Larmore

Technical Report # 86-03

February 1986

A Subquadratic algorithm for constructing approximately optimal binary search trees

Lawrence L. Larmore
University of California, Irvine

Abstract

An algorithm is presented which constructs an optimal binary search tree for an ordered list of n items, and which requires subquadratic time if there is no long sublist of very low frequency items. For example, $time = O(n^{1.6})$ if the frequency of each item is at least ϵ/n for some constant $\epsilon > 0$.

A second algorithm is presented which constructs an approximately optimal binary search tree. This algorithm has one parameter, and exhibits a tradeoff between speed and accuracy. It is possible to choose the parameter such that $time = O(n^{1.6})$ and $error = o(1)$.

1. Introduction

A common method for storing information is the *binary search tree.* We are given items B_1, ... B_n ordered by some key value, and $2n+1$ frequencies β_1, ... β_n and α_0,α_1 ... α_n, where β_i is the frequency of encountering B_i, and α_i is the frequency of encountering an item X such that $B_i < X < B_{i+1}$. It is convenient to create fictitious items B_0 and B_{n+1}, with frequencies $\beta_0 = \beta_{n+1} = 0$, whose key values are respectively lower and higher than all actual items.

A binary search tree for the items B_1, ... B_n is a binary tree T with n interior nodes, labeled in inorder B_1, ... B_n, and $(n+1)$ leaves, labeled (B_0,B_1), ... (B_n,B_{n+1}). Let b_i be the depth of B_i in T, and let a_i be the depth of the leaf (B_i,B_{i+1}). To retrieve an item X, $b_i + 1$ comparisons are needed if $X = B_i$, while a_i comparisons are needed if $B_i < X < B_{i+1}$. The *weighted path length* of T, defined to be $\Sigma_{1\le i\le n}\beta_i(b_i+1) + \Sigma_{0\le i\le n}\alpha_i a_i$ is the expected number of comparisons needed to determine whether X is a member of the list.

Knuth has found an algorithm which requires $O(n^2)$ time and $O(n^2)$ space to find an optimal binary search tree, that is, a binary search tree with minimum weighted

path length. We describe that algorithm in detail in the section 2, since it will be used as a subalgorithm of the the two algorithms presented here.

There have been a number of publications dealing with approximately optimal binary trees which can be constructed in subquadratic time. ([1],[6],[7]) If T_{approx} is any approximately optimal binary search tree, whose weighted path length is P_{approx}, we define the *error* to be the difference $P_{\text{approx}} - P_{\text{opt}}$, where P_{opt} is the weighted path length of an optimal binary search tree. The best subquadratic approximation to date is the min-max tree, defined by Bayer [2]. The definition of a min-max tree is top-down: the root is chosen so as to minimize the maximum of the weights of the resulting left and right subtrees. Bayer has shown that the error for a min-max tree is $O(\log P_{\text{opt}})$, and Fredman [3] has exhibited an algorithm for constructing a min-max tree in $O(n)$ time.

An interesting lower bound result by Allen [1] is that none of the existing subquadratic algorithms produce approximately optimal trees with $O(1)$ error. These existing approximation algorithms use a top-down approach. The algorithms presented in this paper, on the other hand, use a bottom-up approach, as does Knuth's original algorithm.

Summary of Results. We introduce, in this paper, two algorithms, which we call the *Basic Algorithm*, and the *Approximation Algorithm*. Let $\phi = (1 + \sqrt{5})/2 \approx 1.6180339$, the "Golden Ratio." In the theorems below, we assume that the frequencies have been normalized, i.e., $\Sigma\alpha_i + \Sigma\beta_i = 1$.

Theorem 1.1: Let $l < n$ be an integer, and let $0 < \lambda < 1$ be a real number such that $\alpha_i + \beta_{i+1} + \alpha_{i+1} + \ldots + \beta_j + \alpha_j \geq \lambda$ for all i, j such that $j - i > l$. Then the Basic Algorithm computes an optimal binary search tree in $O(n(l + \lambda^{-\log_{\phi}2}\log n))$ time.

Corollary 1.2: Suppose $\beta_i + \alpha_i \geq \epsilon/n$ for all i, where $\epsilon > 0$ is constant. Then the Basic Algorithm computes an optimal binary search tree in $O(n^{\sigma}(\log n)^{\tau})$ time, where $\sigma = 1 + 1/(1+\log\phi) \approx 1.59023$ and $\tau = 1 - 1/(1+\log\phi) \approx 0.40977$.

Theorem 1.3: For any choice of real $0 < r < 1$, the Approximation Algorithm computes a binary search tree T_{approx} such that

$$error = P_{\text{approx}} - P_{\text{opt}} \leq n(n^r/\log n)^{-(1+\log\phi)}$$

where P_{opt} is the weighted path length of the optimal binary search tree, and P_{approx} is the weighted path length of T_{approx}. The time for the Approximation Algorithm is $O(n^{1+r})$.

Corollary 1.4: For any choice of $r > 1/(1+\log\phi) \approx 0.59023$, the Approximation Algorithm computes a binary search tree whose weighted path length differs from that of the optimal binary search tree by $o(1)$, in time $O(n^{1+r})$.

Note that it is pointless to choose $r < 1/(1 + \log\phi)$, since in that case the weighted path length of the tree constructed by the Approximation Algorithm would exceed the weighted path length of an almost complete binary tree, which can be constructed in linear time.

2. Knuth's Monotonicity Lemma and the Quadratic Algorithm

The Traditional Algorithm. The traditional bottom up dynamic programming approach to finding an optimal binary search tree requires cubic time. The method is to compute the optimum binary search tree for each sublist, in order of length. The method depends on the observation that each subtree of an optimal binary search tree is also an optimal binary search tree. Knuth [5] uses a monotonicity property of the optimal binary search trees for sublists to speed this algorithm up by an entire order of magnitude.

The k^{th} step of Knuth's algorithm is to compute the optimal binary search trees on each sublist of length k, given that optimal binary search trees have been computed for each sublist of length $k-1$. The roots of these trees form a monotone increasing sequence of length $n-k+2$.

The correctness of Knuth's algorithm depends on his *Monotonicity Theorem* (named simply "Theorem" on page 18 of [5]), which is a simple corollary of the *Quadrangle Lemma* introduced below, which is similar to the quadrangle condition introduced by Yao [8]. We will need the Quadrangle Lemma for our Approximation Algorithm.

Notation. After Knuth, we let $W_{i,j}$ and $P_{i,j}$ be the total weight and weighted path length of an optimal binary search tree $T_{i,j}$ for all items lying strictly between B_i and B_j, when $i<j$. From [5] we have:

1. $P_{i,i+1} = 0$
2. $W_{i,i+1} = \alpha_i$
3. For $i < j$, $W_{i,j+1} = W_{i,j} + \beta_j + \alpha_j$
4. For $i < j-1$, $P_{i,j} = \min_{i<k<j}(P_{i,k} + P_{k,j}) + W_{i,j}$. Furthermore, $P_{i,j} = P_{i,k} + P_{k,j} + W_{i,j}$ if and only if k could be chosen to be the root of $T_{i,j}$.

For $i \geq j$, we let $W_{i,j} = P_{i,j} = 0$.

Lemma 2.1 (Quadrangle Lemma): Suppose $i_0 \leq i_1$ and $j_0 \leq j_1$. Then $P_{i_0,j_0} + P_{i_1,j_1} \leq P_{i_0,j_1} + P_{i_1,j_0}$.

Proof: If $i_0 = i_1$ or $j_0 = j_1$, both sides are equal.

If $j_0 \leq i_1$, the algorithm follows from the observation that T_{i_1,j_0} is empty, and that the set of nodes T_{i_0,j_1} contains the sets of nodes of T_{i_0,j_0} and T_{i_1,j_1} as disjoint subsets. For $a \leq b < c \leq d$ let $P_{a,d}\big|_{b,c}$ be the contribution to $P_{a,d}$ of the nodes strictly between B_b and B_c. Let T'_{i_0,j_0} be the tree obtained from T_{i_0,j_1} by deleting all other nodes and promoting enough nodes to restore the tree condition. Let P'_{i_0,j_0} be the weighted path length of T'_{i_0,j_0}. Since no node is demoted, $P'_{i_0,j_0} \leq P_{i_0,j_1}\big|_{i_0,j_0}$. T'_{i_1,j_1} and P'_{i_1,j_1} are defined similarly. Since $j_0 \leq i_1$, $P_{i_1,j_0} = 0$. Thus $P_{i_0,j_0} + P_{i_1,j_1} \leq P'_{i_0,j_0} + P'_{i_1,j_1}$ $\leq P_{i_0,j_1}\big|_{i_0,j_0} + P_{i_0,j_1}\big|_{i_1,j_1} \leq P_{i_0,j_1}$.

The remaining case is that $i_0 < i_1 < j_0 < j_1$, which we prove by induction on $j_1 - i_0$. Let r be the root of T_{i_1,j_0} and s the root of T_{i_0,j_1}. Without loss of generality, $r \leq s$. By the inductive hypothesis, $P_{i_0,r} + P_{i_1,s} \leq P_{i_0,s} + P_{i_1,r}$. Note also that $W_{i_0,j_0} + W_{i_1,j_1} = W_{i_0,j_1} + W_{i_1,j_0}$. Using these two relationships, we have:

$$P_{i_0,j_0} + P_{i_1,j_1} \leq P_{i_0,r} + P_{r,j_0} + W_{i_0,j_0} + P_{i_1,s} + P_{s,j_1} + W_{i_1,j_1} \leq$$
$$P_{i_0,s} + P_{s,j_1} + W_{i_0,j_1} + P_{i_1,r} + P_{r,j_0} + W_{i_1,j_0} = P_{i_0,j_1} + P_{i_1,j_0}$$

This concludes the proof of 2.1.

Knuth's monotonicity lemma states that the root of the optimal binary search tree of a sublist is essentially a monotone function of the sublist. More formally:

Lemma 2.2 (Monotonicity Lemma): Let $i_0 \leq i_1, j_0 \leq j_1$. Let r,s be the roots of T_{i_0,j_0}, T_{i_1,j_1} respectively. Then either $r \leq s$, or T_{i_0,j_0}, T_{i_1,j_1} can be replaced by optimal trees T'_{i_0,j_0}, T'_{i_1,j_1} for the same subproblems, whose roots are s and r, respectively.

Proof: Assume $r > s$. Let T'_{i_0,j_0} be a binary search tree for all items strictly between B_{i_0} and B_{j_0} rooted at s, which has minimum weighted path length for all such trees. Let T'_{i_1,j_1} be a binary search tree for all items strictly between B_{i_1} and B_{j_1} rooted at r, which has minimum weighted path length for all such trees. Let P'_{i_0,j_0} and P'_{i_1,j_1} be the weighted path length of those trees. Let $\lambda = P'_{i_0,j_0} - P_{i_0,j_0} \geq 0$, and let $\mu = P'_{i_1,j_1} - P_{i_1,j_1} \geq 0$. Since the left and right subtrees of P'_{i_0,j_0} must be optimal, we have

$$P'_{i_0,j_0} = P_{i_0,s} + P_{s,j_0} + W_{i_0,j_0}$$
hence $$\lambda = P_{i_0,s} + P_{s,j_0} - P_{i_0,r} - P_{r,j_0}.$$
Similarly $$\mu = P_{i_1,r} + P_{r,j_1} - P_{i_1,s} - P_{s,j_1}.$$

But, by the Quadrangle Lemma, we have:

$$P_{i_0,s} + P_{i_1,r} \leq P_{i_0,r} + P_{i_1,s}$$
and $$P_{sj_1} + P_{r,j_0} \leq P_{rj_1} + P_{s,j_0}$$

It follows easily that $\lambda + \mu \leq 0$, whence $\lambda = \mu = 0$. Thus, T'_{i_0,j_0} and T'_{i_1,j_1} are optimal.

Knuth's Quadratic Algorithm. In the version of the algorithm we present here, arrays *root*[*i*,*j*] and *P*[*i*,*j*] are constructed for all $0 \leq i < j \leq n+1$. *P*[*i*,*j*] will contain the value of $P_{i,j}$, the weighted path length of the optimal tree $T_{i,j}$, while *root*[*i*,*j*] will contain the root of that tree.

Knuth's Quadratic Algorithm

```
P[i,i+1] ← 0 for all i
for g ← 2 to n+1 do
        for i ← 0 to n+1−g do
                Compute root[i,i+g] and P[i,i+g]
```

The action of the compute step is to examine each *r* which is a candidate for *root*[*i*,*i*+*g*]. A value of *r* for which $P_{i,r} + P_{r,i+g}$ is minimized is assigned to *root*[*i*,*i*+*g*], and *P*[*i*,*i*+*g*] is assigned the value $P_{i,r} + P_{r,i+g} + W_{i,i+g}$.

The compute step is executed approximately $n^2/2$ times. In principle, all integers in the range [*i*+1, ... *i*+*g*−1] are candidates for *root*[*i*,*i*+*g*]. If all candidates were examined, the combined time for all executions of the compute step would be $O(ng)$ for each *g*, $O(n^3)$ overall. However, if $g > 2$, the monotonicity lemma guarantees that only values in the range [*root*[*i*,*i*+*g*−1], ... *root*[*i*+1,*i*+*g*]] need be considered as candidates. The total number of examinations of candidates for all the compute steps for one value of *g* is thus at most 2*n*−*g*+2. It follows that the time required for each value of *g* is $O(n)$, and thus the entire algorithm requires only $O(n^2)$ time.

3. The Monotone Rowmin Problem

In this section we introduce a divide and conquer algorithm which will be used in the Basic Algorithm of the next section.

The Matrix Rowmin Problem. Given an $n \times m$ matrix *M*, let *Rowmin*[*i*] be the

minimum value of M in the i^{th} row. The Matrix Rowmin Problem is to determine, for each $1 \leq i \leq n$, some $j = Minpos[i]$ for which $M[i,j] = Rowmin[i]$. The obvious algorithm, which is also the best in general, requires examination of every entry of M.

We say that a matrix M is *monotone* if the column position of the minimum entry in each row is a monotone increasing function of the row. Formally, M is monotone if, for any row indices $i_0 < i_1$ and any column indices $j_0 > j_1$ such that $M[i_0,j_0] = Rowmin[i_0]$ and $M[i_1,j_1] = Rowmin[i_1]$, it follows that $M[i_0,j_1] = Rowmin[i_0]$ and $M[i_1,j_0] = Rowmin[i_1]$.

The Rowmin Problem can be solved for a monotone $n \times m$ matrix in time $O(n + m \log n)$ as follows. First, let $k = \lceil n/2 \rceil$, then compute $j = Minpos[k]$ by linear search. Apply the same algorithm recursively to upper left submatrix of size $(k-1) \times j$ and the lower right submatrix of size $(n-k) \times (m-j+1)$. Monotonicity guarantees that correct values of *Minpos* will be found in those restricted ranges. It is a simple recursion exercise to verify the time complexity.

4. The Basic Algorithm

In this section, we show first how the optimal binary search tree problem can be reduced to the classical minimum weight path problem for a directed acyclic graph.

Throughout this section, we assume that the frequencies have been normalized.

Abstract Binary Trees. In any binary tree T, the *abstract position* of a node x is the bit list of descent commands necessary to find x from the root. Let $position_T(x) \in \Sigma^*$ (write $position(x)$ if T is understood) be the abstract position of x in T, where $\Sigma = \{0,1\}$, and where '0' and '1' symbolize left and right descent, respectively. For example, $position(root) = \epsilon$, the empty string, and $position(x) = 110$ if x is the right son of the left son of the left son of the root.

Σ^* is an infinite complete binary tree in the obvious way: the root of Σ is ϵ, and for any $w \in \Sigma^*$, $left(w) = w0$ and $right(w) = w1$.

We sav that a finite subset $t \subset \Sigma^*$ is an *abstract binary tree* if $t = \text{PREF}(t)$, i.e., any ancestor (prefix) of any member of t is also a member of t. To determine that $t = \text{PREF}(t)$, it suffices to check that it contains all of its own parents.

Σ^* can be made into a directed acyclic graph by using an "inorder successor" relation. If $u, v \in \Sigma^*$, we say that $u \Rightarrow v$ if v is the inorder successor of u in some abstract binary tree. Thus, $u \Rightarrow v$ if and only if $v = u10^k$ or $u = v01^k$ for some $k \geq 0$.

If $X = (x_1, \ldots x_n)$ is any list, a *binary search tree* for X is specified by a map $position: X \rightarrow \Sigma^*$ which satisfies the following three conditions:

(4-1) $position(x_1) \in 0^*$

(4-2) $position(x_n) \in 1^*$

(4-3) $position(x_i) \Rightarrow position(x_{i+1})$ for all i

In other words, the image of the function *position* consists of the nodes of a path in Σ^* from the "start set" 0^* to the "final set" 1^*. The following lemma shows that conditions (4-1)-(4-3) suffice to define an binary search tree for X.

Lemma 4.1: Let $(w_1, \ldots w_m)$ be the nodes of a path in Σ^* such that $w_1 \in 0^*$ and $w_m \in 1^*$. Then $W = \{w_1, \ldots w_m\}$ is an abstract binary tree.

Proof: It suffices to prove that $parent(w) \in W$ for all $w \neq \epsilon$. If $W = \{\epsilon\}$ we are done. Otherwise, let w_i be the longest member of W. Without loss of generality, the last symbol of w_i is 0.

If $i = 1$, then $w_1 = 0^k$ for some $k > 0$, and $w_2 = 0^{k-1}$. Otherwise, since neither w_{i-1} nor w_{i+1} can be longer than w_i, we have that $w_i = w10^k$ for some $k > 0$, $w \in \Sigma^*$, that $w_{i+1} = w10^{k-1}$, and that $w_{i-1} = w1$; thus, $w_{i-1} \Rightarrow w_{i+1}$. In either case, $W - \{w_i\}$ is an abstract binary tree by the inductive hypothesis, hence contains its own parents.

Since $parent(w_i) = w_{i+1}$, it follows that W is an abstract binary tree.

The Graph G_d.

For any fixed integer $d \geq 0$, we construct a weighted acyclic directed graph G_d such that the minimum weight of any path in G_d from its source to its sink is the weighted path length P_{opt} of the optimal binary search tree T for $(B_1, \ldots B_n)$.

Let $\Sigma^{(d)} = \{w \in \Sigma^* \mid |w| \leq d\}$. We define $G_d = (V_d, E_d)$, where

$$V_d = \{B_1, \ldots B_n\} \times \Sigma^{(d)} \cup \{source, sink\}$$
$$R_d = \{((B_i,u),(B_{i+1},v)) \mid u,v \in \Sigma^{(d)}, u \Rightarrow v\} \cup \{(source,(B_1,v)) \mid v \in 0^*\} \cup \{((B_n,u),sink) \mid u \in 1^*\}$$
$$S_d = \{((B_i,u),(B_j,v)) \mid i<j; u,v \in \Sigma^{(d)}, u \Rightarrow v, \max\{|u|,|v|\}=d\} \cup \{(source,(B_j,v)) \mid |v|=d\} \cup \{((B_i,u),sink) \mid |u|=d\}$$
$$E_d = R_d \cup S_d$$

The members of R_d we call regular edges, while the members of S_d we call special edges. (Note that $R_d \cap S_d \neq \emptyset$, however.)

We also define the *rank* of each vertex: $rank(B_i,w) = i$, $rank(source) = 0$, and $rank(sink) = n+1$. We define the *span* of an edge to be the difference of the ranks of its end points. An edge is regular if and only if its span is 1.

At variance with the usual practice, we place weights on both vertices and edges of the graph. The *weight* of any path in G_d will be defined to be the total weights of the vertices and edges of the path. Weights will be defined as follows:

for vertices:

$$weight(source) = weight(sink) = 0$$
$$weight(B_i,w) = (|w|+1)\beta_i$$

for regular edges:

$$weight((B_i,u),(B_{i+1},v)) = (\max\{|u|,|v|\}+1)\alpha_i$$
$$weight(source,(B_1,0^k)) = (k+1)\alpha_0$$

$$weight((B_n,1^k),sink) = (k+1)\alpha_n$$

for special edges:

$$weight((B_i,u),(B_j,v)) = P_{i,j} + (d+1)\,W_{i,j}$$
$$weight(source,(B_j,v)) = P_{0,j} + (d+1)\,W_{0,i}$$
$$weight((B_i,u),sink) = P_{i,n+1} + (d+1)\,W_{i,n+1}$$

Lemma 4.2: Let χ be a path in G_d from *source* to *sink*. Then there is a binary search tree T such that $position_T(B_i) = w_i$ for each interior node (B_i,w_i) of χ. Furthermore, the weighted path length of T is the weight of χ.

Proof: Let $\chi = (source = x_0, x_1, \ldots x_m = sink)$. Let $i_k = rank(x_k)$. For $0 < k < m$, we write $x_k = (B_{i_k},w_k)$, and $1 \le i_1 < i_2 < \ldots < i_{m-1} \le n$. Note that (x_k,x_{k+1}) is a regular edge if and only if $i_{k+1} = i_k + 1$. By Lemma 4.1, there exists a binary tree T' whose nodes are $\{B_{i_k}\}$, where $position_{T'}(B_{i_k}) = w_k$ for each k. For each $0 \le k < m$, attach the optimal subtree $T_{i_k,i_{k+1}}$ to T' as the left subtree of $B_{i_{k+1}}$ or the right subtree of B_{i_k}; exactly one of those two choices will be possible in each case. Let T be the resulting tree. It is straightforward to verify that the weighted path length of T is the weight of the path χ.

Lemma 4.3: Let T be a search tree (not necessarily optimal), and let P be its weighted path length. Let $\mathcal{X}_d = \{(B_i,w_i) \mid w_i = position_T(B_i),\ |w_i| \le d\}$. Then the elements of $\mathcal{X}_d$ are exactly the interior nodes of some path χ in G_d from *source* to *sink*. Furthermore, $weight(\chi) \le P$.

Proof: We can order the elements of $\mathcal{X}_d$ by rank. Write $\mathcal{X}_d = \{(B_{i_k},w_k)\}_{0<k<m}$, where $i_k < i_{k+1}$ for all k. As in the proof of Lemma 4.3, let $x_0 = source$, $x_m = sink$, $i_0 = 0$, and $i_m = n+1$. Then (x_k,x_{k+1}) is a regular edge if and only if $i_{k+1} = i_k + 1$, and is a special edge otherwise. Thus $\chi = (x_0, x_1, \ldots x_m)$ is a path in G_d. Let T' be the tree constructed from χ by Lemma 4.3, with weighted path length P'. T' and T are identical down to level d, and the subtrees of T' rooted at level $d+1$ are "optimized" versions of the corresponding subtrees of T. Thus, $weight(\chi) = P' \le P$.

The following theorem, which shows the reduction of the optimal binary search tree problem to the minimum weight path problem, is then an immediate corollary of Lemmas 4.3 and 4.4.

Lemma 4.5 (Reduction Lemma): Let T_{opt} be an optimal binary search tree and let P_{opt} be the weighted path length of T_{opt}. Then, for any fixed $d \geq 0$, P_{opt} equals the weight of a minimum weight path in the graph G_d from *source* to *sink*.

Note that this reduction does not yield a subquadratic algorithm by using general graph techniques to find the minimum weight path in G_d, since that graph is too large. In fact, G_d has $n(2^{d+1}-1) + 2$ vertices, $(n-1)(2^{d+2}-2d-4) + 2(d+1)$ regular edges, and $(2^d-1)(n-1)(n-2) + 2(n-1)(d+1)$ special edges which are not regular. The algorithms we introduce will use techniques which take advantage of the special structure of G_d.

The Function f_d. For any d and for any vertex x of G_d, we define $f_d(x)$ to be the least weight of any path in G_d from *source* to x. Recall that we have defined the weight of a path to be the sum of the weights of the vertices and the edges of that path, and that the weight of the last vertex is included. Thus, $f_d(sink) = P_{\text{opt}}$. If no path exists from *source* to x, $f_d(x) = \infty$. For a fixed integer $l \leq n$, let $G_{d,l}$ be the subgraph of G_d consisting of all the vertices and only edges of span not exceeding l. For any vertex x, let $f_{d,l}(x)$ be the least weight of any path in $G_{d,l}$ from *source* to x. Clearly $f_{d,l}(x) \geq f_d(x)$.

Let F_k denote the k^{th} Fibonacci number.

Lemma 4.6: If $W_{i,j} > 2/F_{d+3}$ for all pairs i, j such that $j - i > l$, then $P_{\text{opt}} = f_{d,l}(sink)$.

Proof: By Lemma 4.5, $P = f_d(sink) \leq f_{d,l}(sink)$. Now suppose $f_{d,l}(sink) < f_d(sink)$. The path in $G_{d,l}$ corresponding to T_{opt} must therefore use some edge of $G_{d,l}$ not available in g_d, which would have to be a special edge of span greater than l. This implies that P_{opt} has a subtree $T_{i,j}$ for $j-i > l$ rooted at depth d, which,

by [4, Th. 2] implies that the weight of $T_{i,j}$ cannot exceed $2/F_{d+3}$, contradiction.

We now present the Basic Algorithm, in top-down form. The output of the algorithm is the value $f_{d,l}(sink)$, which will equal F_{opt}.

Basic Algorithm

```
Choose d, l
Compute f_{d,l}(B_i,0^d) for all i
w ← 0^d
while w ≠ 1^d do
    begin
        w ← the inorder successor of w in Σ^(d)
        Compute f_{d,l}(B_i,w) for all i
    end
Compute f_{d,l}(sink)
```

Detail of the Basic Algorithm and Time Analysis. For any vertex y, $f_{d,l}(y) = weight(y) + \min\{f_{d,l}(x) + weight(x,y)\}$, where the minimum is taken over all edges (x,y) of the graph $G_{d,l}$. The classic minimum weight path algorithm examines all edges. The Basic Algorithm examines all regular edges, but only a small subset of the special edges.

The first step of the Basic Algorithm is to find the smallest integers d, l such that $l = \lceil 2^d \log\ n \rceil$ and $W_{i,j} > 2/F_{d+3}$ for all pairs i, j such that $j - i > l$. It requires $O(n)$ time to determine whether a particular candidate value for d is suitable, since it suffices to check $W_{i,i+l}$ for all i, and there are at most $\log n \log\log n$ values of d to check. Thus, the step "Compute d, l" requires $O(n \log n \log\log n)$ time.

For any i, there is at most one edge to $(B_i,0^d)$, and that edge is from *source*. It thus takes $O(n)$ time to compute $f_{d,l}(B_i,0^d)$ for all i.

Suppose that $w \neq 0^d$, and that $f_{d,l}(B_i,u)$ has been computed for all $u \in \Sigma^{(d)}$ such that $u \Rightarrow w$. Let w' be the inorder predecessor of w in $\Sigma^{(d)}$. We define an $n \times n$ matrix

M as follows: $M[j,i] = f_{d,l}(B_i,w') + P_{i,j} + (d+1)\,W_{i,j}$ provided $0 < j-i \leq l$, and $M[j,i] = \infty$ otherwise. Then the minimum value in the j^{th} row of M is precisely the minimum weight of any path in $G_{d,l}$ from *source* to (B_j,w) where the last edge of that path is a special edge. By the Quadrangle Lemma (Lemma 2.1), M is monotone, and thus finding those minimum row values is an instance of the Monotone Rowmin Problem introduced in section 2. The minimum row values are found in $O(n \log n)$ time using the divide and conquer algorithm. Each regular edge is then examined once to possibly find even lower values for $f_{d,l}(B_j,w)$. Thus, the total time for the main loop of the algorithm is $O(2^d n \log n)$.

The final step of the Basic Algorithm is to compute $f_{d,l}(sink)$, which involves examining each of the $l + d$ edges to *sink*. This step takes linear time.

We conclude that the Basic Algorithm takes $O(nl + 2^d n \log n) = O(nl)$ time. Correctness of the Basic Algorithm is implied by Lemma 4.6.

5. The Approximation Algorithm

In this section we describe a modification of the Basic Algorithm which executes in subquadratic time, and yields an approximately optimal binary search tree. There is a tradeoff between accuracy and speed. Again, we assume that the frequencies are normalized, i.e., $\Sigma\alpha_i + \Sigma\beta_i = 1$.

The Approximation Algorithm has one real parameter, r. Although r could be chosen to be any value in the range $0 < r < 1$, it is pointless to choose $r < 1/(1 \log \phi) \approx 0.59023$, since weighted path length of the tree constructed by the algorithm would exceed that of an almost complete binary tree.

The worst case for the Basic Algorithm is the case where there are a large number of consecutive items with very low frequency. The idea of the Approximation Algorithm is to delete such runs of low frequency items, then apply the Basic Algorithm to construct an optimal binary search tree on the remaining items, then to attach the

low frequency nodes to that tree in a manner which does not add much to the weighted path length.

Description of the Approximation Algorithm. Let $r < 1$ be the parameter. Let $l = \lceil n^r \rceil$, and let $\delta = (\log n)^{\log\phi}/l^{1+\log\phi}$. Let $D = \{i \le i \le n | \alpha_{i-1}+\beta_i+\alpha_i < \delta\}$. Use the Basic Algorithm to construct an optimal binary search tree T' for the list obtained by deleting both B_i and (B_i, B_{i+1}) for all $i \in D$. We now organize those deleted items into almost complete binary trees and attach them to T', as follows. Write D as the disjoint union of maximal runs, i.e., $D = [i_1 .. j_1] \cup \ldots \cup [i_m .. j_m]$ where $i_{k+1} > j_k + 1$. For each k, let T_k be the almost complete binary search tree for the items strictly between B_{i_k-1} and B_{j_k+1}. We now form T, a binary search tree for all the nodes, by removing from T' each external node (B_{i_k-1}, B_{i_k}) and replacing it with T_k.

Time Complexity. After the deletion, the weight of any run of items of length l is at least $l\delta = (\log n/l)^{\log\phi}$. The time to run the Basic Algorithm is thus $O(nl) = O(n^{1+r})$. The remaining parts of the Approximation Algorithm require only linear time.

Analysis of the Error. Let P, P' be the weighted path lengths of T, T', respectively. Let P_{opt} be the weighted path length of the optimal binary search tree. Obviously $P' \le P_{\text{opt}}$, since P' is an optimal tree constructed after deletetion of 0 or more items. Let h be the height of T'. By [4, Th. 2], $h = O(\log n)$. The sum of the weights of the T_k cannot exceed $n\delta$, and their heights cannot exceed $\log n$. The amount that is added to the weighted path length by attaching the T_k must therefore be $O(n\,\delta\,\log n)$. It follows that:

$$error = P - P_{\text{opt}} \le P - P' = O(n\,\delta\,\log n) = O(n^{1-r(1+\log\phi)}(\log n)^{1+\log\phi}).$$

6. Open Questions

The results of this paper leave open the problem of whether a subquadratic algorithm can be found for constructing an optimal binary search tree in all cases.

However, we do believe the following, weaker, "Las Vegas" version to be true:

Conjecture. Suppose the values of the α_i and the β_i are chosen at random, from two given distributions; then normalized by dividing by $\Sigma\alpha_i + \Sigma\beta_i$. Then there exists an agorithm to construct the optimal binary search tree, which runs in subquadratic expected time.

Note that the Basic Algorithm of this paper answers the conjecture affirmatively, unless the distributions are such that all but very few of the values are very small. Since we expect that few distributions occuring in applications would be extremely skewed, the Basic Algorithm should construct the optimal binary search tree in subquadratic time in most practical situations.

References

[1] Allen, B., "Optimal and Near-Optimal Binary Search Trees," *Acta Informatica* **18** (1982) pp. 255-263.

[2] Bayer, P.J., "Improved bounds on the costs of optimal and balanced binary search trees," *Project MAC Technical Memorandum 69* MIT Cambridge MA (1975)

[3] Fredman, M.L., "Two applications of a probabilistic search technique: Sorting X + Y and building balanced search trees," *7th ACM Symposium on Theory of Computing*, Albuquerque (1975), pp. 240-244.

[4] Hirschberg, D.S., Larmore, L.L. and Melodowitch, M., "Subtree Weight Ratios for Optimal Binary Search Trees", Tech Rept. 86-02, ICS Dept, University of California Irvine (1986). Also submitted for publication to Acta Informatica.

[5] Knuth, D.E. "Optimum binary search trees," *Acta Informatica* **1** (1971), pp. 14-

25.

[6] Mehlhorn, K. "Nearly optimal binary search trees," *Acta Informatica* **5** (1975), pp. 287-295.

[7] Unterauer, K. "Dynamic Weighted Binary Search Trees," *Acta Informatica* **11** (1979), 341-362.

[8] Yao, F.F. "Efficient dynamic programming using quadrangle inequalities," *Proc. 12th Annual ACM Symp. on Theory of Comput.* (1980), pp. 429-435.