Construction of Optimal Binary Split Trees in the Presence of Bounded Access Probabilities

J. H. Hester

D. S. Hirschberg

L. L. Larmore

Department of Information and Computer Science

University of California, Irvine

Irvine, California 92717

Abstract

A binary split tree is a search structure combining features of heaps and binary search trees. The fastest known algorithm for building an optimal binary split tree requires $\Theta(n^4)$ time if the keys are distinct and $O(n^5)$ time if the keys are non-distinct. $\Theta(n^3)$ space is required in both cases. A modification is introduced which reduces a factor of n^2 in the asymptotic time to a factor of $n^\alpha \lg(n)$ when the smallest probability of access of any key is $\Theta(n^{-\alpha})$. This yields an asymptotic improvement any time the smallest access probability is greater than $\Theta(n^{-2})$. The space requirement is not affected. For example, when access probabilities are distinct and the smallest access probability is $\Theta(1/n)$, the modified algorithm requires only $O(n^3 \lg n)$ time (as opposed to the $O(n^4)$ result mentioned above). Several natural refinements to the basic modification are presented which should improve the time within the same asymptotic order.

Introduction

A binary split tree (BST) is a structure for storing records on which searches will be performed, assuming that the probabilities of access are known in advance. For every subtree T in a BST, the record with the highest access probability of all records in T is stored in the root of T. The remaining records are distributed among the left and right subtrees of T such that the keys of all records in the left subtree are less than the keys of all records in the right subtree. Each node in a BST contains the *key value* of the record in that node and a *split value* which lexically divides the values of the keys in the left and right subtrees. A simple split value is the value of the largest key in the left subtree.

Under the assumption of distinct access probabilities and no failed searches, for any given set of n records, the key to be put in the root is predetermined but the split value for the root may be chosen to divide the remaining $n-1$ records between the left and right subtrees in any of n possible ways. If failed searches are considered, the split value may be any of $n+2$ possibilities.

Binary split trees were introduced by Sheil [6], who conjectured that the arbitrary removal of nodes with high access probabilities from the lexicographic ordering (for placement in roots of higher subtrees) made the normal dynamic programming techniques inapplicable. However, Huang and Wong [2] and Perl [5] noted that the keys missing from any given range must be the keys with the largest access probabilities in that range of keys, thus allowing a representation of the set of keys in a subtree by specifying a range of keys and a count of the number of keys missing from that range. This led to $\Theta(n^5)$ time and $\Theta(n^3)$ space dynamic programming algorithms to construct optimal BSTs in a manner similar to Knuth's algorithm [4] for constructing optimal binary search trees. These algorithms did not handle the case of non-distinct access probabilities, but Perl outlined a method to handle this case requiring exponential time. Hester, et. al.

[1] presented an improvement which reduced the time required by a factor of $\Theta(n)$, resulting in a total time of $\Theta(n^4)$ for distinct access probabilities. Their algorithm also constructed optimal trees in the presence of non-distinct access probabilities in $O(n^5)$ time.

Huang and Wong [3] defined generalized binary split trees (GBSTs), which relax the constraint that the key in the root of any subtree be the key with highest access probability of the (remaining) keys in that subtree, and presented an $O(n^5)$ algorithm which construct optimal GBSTs. They noted that BSTs, as well as binary search trees, are subsets of GBSTs, and showed that optimal GBSTs may have smaller expected search cost than optimal BSTs (which have already been shown may be much better than optimal binary search trees). They noted, however, that the difference in expected cost between optimal BSTs and optimal GBSTs appears to be small.

This paper presents modifications to Hester, et. al.'s optimal BST algorithm which restrict the number of values of one of the loop variables when knowledge about the application is sufficient to guarantee a lower bound of $\Theta(n^{-\alpha})$ on the smallest probability of access of keys for any data set which may arise in that application. When such a bound can be guaranteed, the modifications reduce a factor of n^2 in the algorithm's asymptotic time requirement to a factor of $n^\alpha \lg(n)$.[†] Thus, for example, if knowledge about a given application is sufficient to show that there exists a constant c such that the smallest access probability cannot be less than c/n (for any data set consistant with that application with sufficiently large n), then the modified algorithm will require only $O(n^3 \lg n)$ time instead of the $O(n^4)$ time required without the modifications. The modifications do not adversely affect the algorithm's asymptotic time if no lower bound on access probability can be guaranteed, thus they may be used on all data sets and, since the value of α is not actually built into the algorithm (it is used only for analysis), asymptotic gain

[†] $\lg(n)$ denotes the logarithm base 2 of n.

in speed will occur for bounded distributions without actually proving or even knowing the bounds. The modifications are also directly applicable to Huang and Wong's optimal GBST algorithm, yielding the same improvement.

1. The Algorithm

Only an outline of the algorithm sufficient to understand our modifications will be presented here. The reader is directed to Hester, et. al. [1] for a full description of the algorithm.

We are given n records indexed from 1 to n. Each record r_i has a key $Key(i)$ such that $Key(i) < Key(j)$ for all $i < j$. Each record r_i also has an *access probability* $p(i)$. In addition, to account for failed searches, we are given *failure probabilities* $q(i)$ for $0 \leq i \leq n$, which are the probabilities of searching for a key K such that $Key(i) < K < Key(i+1)$. To complete the definitions of the probabilities in a uniform fashion and to simplify the algorithms, $p(0) = p(n+1) = 0$ and $Key(0)$ and $Key(n+1)$ are defined to be, respectively, $-\infty$ and ∞.

The following assumes that access probabilities are distinct. The modifications for dealing with non-distinct access probabilities are of no consequence to our results.

Define a *range* of records i to j to be the set of records whose indices are in the range $i+1, i+2, \ldots, j$. Let $\langle i, j, k \rangle$ refer to the sequence of probabilities

$$\{q(i), p(i+1), q(i+1), p(i+2), \ldots, q(j-1), p(j)\}$$

where the largest k access probabilities (p's) are left out of the sequence. Since records are ordered by key value, the records with the k largest access probabilities could be anywhere in the sequence. A subtree T *spans* the sequence $\langle i, j, k \rangle$ if the subtree contains all records whose access probabilities are in $\langle i, j, k \rangle$, and contains no other records. A record r is said to be *missing* from a subtree T if the index of r is in the range i to j and T spans $\langle i, j, k \rangle$, but r's access probability is not in

$\langle i, j, k \rangle$. In other words, r is missing from T if it is in the range of T, but has one of the k highest access probabilities in that range, which causes it to be placed in the root of some higher subtree. Perl [5] gave a simple proof that the keys missing from any subtree T must be the keys with the largest access probabilities that T spans, and that these keys must be stored in an ancestor of the root of T.

As in [1], let $COT[i, j, k]$ be the cost of an optimal subtree spanning $\langle i, j, k \rangle$, which is defined by

$$COT[i,j,k] = W[i,j,k] + \min_{i<l<j} \left\{ \begin{array}{c} COT\left[i, l, GE_L(i,j,k,l)\right] \\ + \\ COT\left[l, j, k+1-GE_L(i,j,k,l)\right] \end{array} \right\}$$

where $W[i, j, k]$ is the weight of a subtree (not counting the k missing nodes) and $GE_L(i, j, k, l)$ is the number of records, in the range of the left subtree of a BST T spanning $\langle i, j, k \rangle$ with a split index of l, which have probability greater than or equal to that of the root of T.

The following is an outline of the main portion of Hester, et. al.'s [1] dynamic programming algorithm for calculating an optimal BST. The interspersed order formulas represent time required by blocks of statements that are of no consequence to our result.

```
begin
    $\Theta(n)$
    for $d \leftarrow 2$ until $n+1$ do
        for $i \leftarrow 0$ until $n+1-d$ do begin
            $j \leftarrow i+d$
            for $k \leftarrow 0$ until $d-1$ do begin
                $\Theta(1)$
                for $l \leftarrow i+1$ until $j-1$ do begin
                    $\Theta(1)$ or $O(n)$
                end
                $\Theta(1)$
            end
            $\Theta(1)$
        end
end
```

Figure 1. Loop outline of split tree algorithm.

The innermost block requires $\Theta(1)$ time (to look up the costs of optimal subtrees of a subtree spanning $\langle i,j,k\rangle$ and split at index l) when access probabilities are distinct, and $O(n)$ time otherwise (to consider distributions between left and right subtrees of missing records with access probabilities equal to that of the root).

Huang and Wong's [3] optimal GBST algorithm conforms to this outline with the negligible exceptions of variable names, differences of at most one on some of the loop indices, and the innermost block is replaced by two loops which take $O(n)$ time.

2. The Modification

Intuitively, a large range of records cannot be located too low in an optimal tree due to the fact that the weight of those records, that low in the tree, would make the overall expected cost of the tree too high. We note that any subtree spanning $\langle i,j,k\rangle$ must be at least k deep in the tree, since the k missing nodes must be on the path from the root of the tree to the root of the subtree. We bound the values of k which need be considered for a given range of records by eliminating any values which would result in too great a cost being contributed to the tree. In this section we present the basic modifications in simple form and an analysis of the time gained, and then in the next section we discuss some refinements along the same lines.

Sheil [6] defined a median split tree where, by always choosing a split value such that the number of records in the left and right subtrees are equal (or nearly so), an almost complete tree results. Since this is a valid split tree and has a worst case cost of $\lg(n)$ probes, therefore $\lg(n)$ is an upper bound on the expected cost of any optimal split tree. Consequently $\lg(n)$ is an upper bound for the sum of the weighted path lengths (weight of a node multiplied by its depth in the tree) in any optimal split tree. Thus, any subtree whose weight multiplied by the depth of its root is greater than $\lg(n)$ cannot be part of any optimal split tree.

Referring to Figure 1, for a given d $(= j - i)$, we can determine a range of values of k for which any subtree spanning $\langle i, j, k\rangle$ cannot be a part of an optimal BST, and which therefore need not be considered. Assume that, before beginning the algorithm, we search the list of records to determine s, the value of the smallest access probability among the records. The cost contributed to the entire tree by any subtree spanning $\langle i, j, k\rangle$ is at least $W[i, j, k]\cdot$(the depth of the subtree). Since k is a lower bound on the depth of the subtree, and it contains $(d - k)$ nodes each of which weighs at least s, the subtree must contribute a cost of at least $ks(d - k)$ to the split tree. This is greater than $\lg(n)$ for k in the range

$$BADk \quad = \quad \left\{ \frac{d - \sqrt{d^2 - \frac{4}{s}\lg n}}{2} < k < \frac{d + \sqrt{d^2 - \frac{4}{s}\lg n}}{2} \right\}$$

Thus, values of k in the range $BADk$ need not be considered. Note that this bound is only defined for $d \geq \sqrt{(4/s)\lg n}$. For smaller values of d, $BADk$ is the empty range. Removing the $BADk$ range from the "k" loop of the algorithm constitutes the majority of our modification. The only other modification that needs to be made is in the innermost loop, when costs of subtrees are looked up in the COT array. A check must be made to see if these subtrees were not considered, (i.e., if $ks(d-k) > \lg n$). If so, their weight was not previously computed, but should be considered to be infinite so as to eliminate possibility of their use.

The analysis is straightforward. Assume that the access probability of the smallest record is $\Theta(n^{-\alpha})$ (i.e., $s = cn^{-\alpha}$ for some constant value c). Substituting this value into the definition of $BADk$ we find that, for values of d greater than $f(n) = \sqrt{(4/c)n^{\alpha}\lg n}$, the number of values of k which can be ignored is at least $\sqrt{d^2 - (4/c)n^{\alpha}\lg n}$. Thus the number of values of k which need to be considered in this case is no more than $d - \sqrt{d^2 - (4/c)n^{\alpha}\lg n}$. For smaller values of d, the definition of $BADk$ yields no improvement to the old upper bound on the number of values of k which need to be considered, which is d.

Since the number of values of the inner loop variables i, k, and l are all bounded by the value of the outer loop variable d (i will have no more than $n-d$ values, k will be bounded as described above, and l will have no more than d values), an upper bound on the asymptotic time required by the algorithm can be expressed as a sum of the products of these bounds:

$$\begin{aligned}
&\sum_{d=2}^{f(n)}(n-d)\cdot d\cdot d + \sum_{d=f(n)+1}^{n+1}(n-d)\left(d-\sqrt{d^2-(4/c)n^\alpha \lg n}\right)d \\
&< \sum_{d=2}^{f(n)} nf^2(n) + n\sum_{d=f(n)+1}^{n+1} d\left(d-\sqrt{d^2-(4/c)n^\alpha \lg n}\right) \\
&< nf^3(n) + n\sum_{d=f(n)+1}^{n+1} d\left(d-\sqrt{d^2-\frac{4n^\alpha \lg n}{c}+\left(\frac{2n^\alpha \lg n}{cd}\right)^2}+\sqrt{\left(\frac{2n^\alpha \lg n}{cd}\right)^2}\right) \\
&\le nf^3(n) + n\sum_{d=f(n)+1}^{n+1} d\left(d-\sqrt{\left[d-\frac{2n^\alpha \lg n}{cd}\right]^2}+\sqrt{\left(\frac{2n^\alpha \lg n}{cd}\right)^2}\right) \\
&= nf^3(n) + n\sum_{d=f(n)+1}^{n+1} d\left(d-\left[d-\frac{2n^\alpha \lg n}{cd}\right]+\frac{2n^\alpha \lg n}{cd}\right) \\
&\le nf^3(n) + n\sum_{d=1}^{n+1} \tfrac{4}{c}n^\alpha \lg n \\
&= n\left(\tfrac{4}{c}n^\alpha \lg n\right)^{\frac{3}{2}} + n(n+1)\tfrac{4}{c}n^\alpha \lg n \\
&= O\left(n^{\frac{3}{2}\alpha+1}\lg^{\frac{3}{2}} n + n^{\alpha+2}\lg n\right)
\end{aligned}$$

Since the algorithm for distinct access probabilities never takes more than $O(n^4)$ time, $(3/2)\alpha+1 \le \alpha+2$ for all $1 \le \alpha \le 2$, and the additional loop to handle non-distinct access probabilities adds at most a factor of n to the asymptotic time, the modified algorithm has an upper bound of

$O\left(\min\{n^4, n^{\alpha+2}\lg n\}\right)$ for distinct access probabilities

$O\left(\min\{n^5, n^{\alpha+3}\lg n\}\right)$ for non-distinct access probabilities

Thus the algorithm's asymptotic time is decreased whenever $\alpha < 2$.

Since we ignored any possible improvement on the innermost $O(n)$ loop which deals with non-distinct access probabilities, we can similarly assume no improvement (and certainly no degradation) on the corresponding additional loops in Huang and Wong's [3] optimal GBST algorithm. Thus the same modification and analysis are possible for their algorithm, leading to the same improvement.

3. Natural Refinements

The above modification was the simplest possible. Several natural improvements may be made which may speed the algorithm up by constants.

First, the cost of $\lg(n)$ used as a limit for the cost of the subtrees may be improved by precalculating the expected cost of a median split tree for the given records. A few nodes with high access cost can make this limit much lower than $\lg(n)$, and thus further reduce the number of values of k that need be considered.

The other half of the analysis used a lower bound of $s(d-k)$ for the total weight of a subtree. During the loop through the values of k which have not been ruled out by the lower bound on cost, the true weight of each subtree is known and may be consulted to recognize further subtrees which are too heavy to contribute to any optimal tree. This test should also be substituted in the innermost loop for the test which checks to see if a given subtree's cost was previously calculated.

The lower bound on the expected cost contributed by a subtree can be further refined by noting that, for a subtree at depth (at least) k, there must be k nodes along the path from the root of the tree to the root of the subtree. These nodes all have weight at least that of the root of the subtree and, since they must lie on a path from the root of the tree to the root of the subtree, their contributed cost is at least (and possibly much more than) $p\big(R[i,j,k]\big)k(k+1)/2$ $\big($where $R[i,j,k]$ is the index of the root of the subtree spanning $\langle i,j,k\rangle\big)$. A more accurate value may be obtained by taking the k largest weights in the range and calculating the exact cost they would contribute (the heaviest would be at depth 1, the next heaviest at depth 2, etc.). All such sums may be computed in a dynamic programming fashion which will increase time and space only by constant factors.

Finally, for any subtree spanning d nodes there are $n-d$ nodes in the rest of the tree which are all at a depth of at least one, and therefore contribute an expected cost of at least $s(n-d)$. A more accurate bound, which is no more difficult to calculate, is $1-$ (the weight of the subtree and all nodes in the path from the root of the subtree to the root of the tree).

These refinements lead to the formula

$$kW[i,j,k] + \sum_{x=1}^{k} xp\bigl(R[i,j,x-1]\bigr) + \bigl(1 - W[i,j,0]\bigr) \le C_m$$

where $p\bigl(R[i,j,x]\bigr)$ is the weight of the root of an optimal subtree spanning $\langle i,j,k\rangle$ and C_m is the expected cost of a median split tree for the records. An analysis, using the lower bounds $W[i,j,k] \ge s(d-k)$ (as before) and $p\bigl(R[i,j,x]\bigr) \ge s$, indicates no asymptotic improvement.

From a practical standpoint, the first and third terms of the left side of the inequality above are inexpensive and can only decrease the expected number of subtrees which are considered. Although the sum on the left and C_m on the right both appear to contribute strong bounds, they each require sufficient additional work $\bigl(O(n^2)$ to calculate costs of all paths from subtree roots to the root of the tree, and $O(n \lg n)$ to calculate the cost of a median split tree$\bigr)$ that it may be better to use the less accurate but less expensive bounds of $R[i,j,k]k(k+1)/2$ and $\lg(n)$. These decisions are probably best made empirically.

Conclusions and Open Questions

A modification to an algorithm for constructing optimal binary split trees has been presented which reduces a factor of n^2 in the asymptotic time to a factor of $n^\alpha \lg(n)$ when the smallest access probability of records is $\Theta(n^{-\alpha})$. In the best case, when the smallest access probability is known to be $O(1/n)$, a factor of n in the required time is reduced to a factor of $\lg(n)$. These bounds on probabilities need not be known to achieve the improvement: the algorithm may be run without asymptotic loss of speed (relative to the unmodified algorithm) on all data sets, and improvements will occur whenever the data sets *are* bounded. It is likely that, for unbounded data, the algorithm will still prune enough loops to offset the constant factor of extra work it performs.

It would be interesting to determine, either theoretically or empirically, what effect the refinements have on the run-time of the algorithm.

It seems unlikely that the lower bound on the algorithm is different from the upper bound we provided, but this is yet to be shown.

Other methods of expressing our bounds may lead to more accurate analyses Similarly, different bounds may lead to different (and better) modifications (or algorithms).

References

1. J. H. Hester, D. S. Hirschberg, S.-H. S. Huang and C. K. Wong, Faster construction of optimal binary split trees, *J. Algorithms*, to appear.

2. S.-H. S. Huang and C. K. Wong, Optimal binary split trees, *J. Algorithms* **5** (1984) 69–79.

3. S.-H. S. Huang and C. K. Wong, Generalized binary split trees, *Acta Informatica* **21** (1984) 113–123.

4. D. E. Knuth, "The Art of Computer Programming," Vol. 3, "Sorting and Searching," pp. 433–439, Addison–Wesley, Reading, Mass., 1973.

5. Y. Perl, Optimum split trees, *J. Algorithms* **5** (1984) 367–374.

6. B. A. Sheil, Median split trees: A fast lookup technique for frequently occurring keys, *Comm. ACM* **21** (1978) 947–958.