Faster Construction of Optimal Binary Split Trees

J. H. Hester
D. S. Hirschberg

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

S.-H. S. Huang

Department of Computer Science
University of Houston
Houston, Texas 77004

C. K. Wong

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Abstract

A binary split tree is a search structure combining features of heaps and binary search trees. Building an optimal binary split tree was originally conjectured to be intractable due to difficulties in applying dynamic programming techniques to the problem. However, two algorithms have recently been published which generate optimal trees in $O(n^5)$ time, for records with distinct access probabilities. An extension allowing non-distinct access probabilities required exponential time. These algorithms consider a range of values when only a single value is possible. A dynamic programming method for determining the correct value is given, resulting in an algorithm which builds an optimal binary split tree in $O(n^5)$ time for non-distinct access probabilities and $\Theta(n^4)$ time for distinct access probabilities.

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. For optimal BSTs, the number of possible divisions is $n-2$ if failed searches are not considered and n if failed searches are considered. This is due to the easily proven fact that, if there are two or more non-zero probabilities (access probabilities or failure probabilities) in any optimal subtree X, then at least one non-zero probability must be in each of the two subtrees of X.

Binary split trees were introduced by Sheil [5], 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 [1] 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 a $\Theta(n^5)$ time and $\Theta(n^3)$ space dynamic programming algorithm to construct optimal BSTs in a manner similar to Knuth's algorithm [3] for constructing optimal binary search trees.

Shortly after Huang and Wong's first paper, Perl [4] presented an independently derived algorithm similar to Huang and Wong's which had the same time and space bounds, but which also took into account probabilities of failed searches. Perl showed that the technique used by Knuth to reduce the asymptotic time complexity of his optimal binary tree constructer by a factor of n could not be applied to the construction of BSTs. This paper also presented an algorithm which allowed non-distinct access probabilities by including some top-down decision making which resulted in an exponential time algorithm.

Unfortunately, the algorithms (for distinct access probabilities) presented by Huang and Wong and by Perl pick the minimum weight of subtrees resulting from considering values of a variable over a range, when only one value could be correct. This adds a factor of $O(n)$ time to the algorithm, and raises the question of whether the algorithms might proffer a minimum cost value of this variable which is not attainable. If this is so, these errors may be made independently for every subtree, and the algorithms may result in a structure that is not a valid split tree. Some quick thought on the problem and a little testing gave us reason to believe that the cost corresponding to the correct value of the variable will always be better than the costs corresponding to all of the incorrect values, and thus the correct value will always be picked. We have not proven this, since the results of this paper render the question academic.

Huang and Wong [2] proposed a more generalized split tree, which relaxes the constraint that the record with the highest access probability must be in the root. They presented an algorithm for constructing trees of this nature, which works even in the case of non-distinct access probabilities, but requires $\Theta(n^5)$ time.

This paper is the result of a merger of two parallel collaborations. Huang and Wong modified their earlier algorithm to remove the superfluous loop and discussed extending their new algorithm to handle failed accesses in a manner similar to the method given by Perl [4]. Independently, Hester and Hirschberg pointed out the possible error of constructing an incorrect tree and developed an analogous algorithm, but directly incorporated Perl's method of handling failed accesses. They also extended their algorithm to the problem of constructing an optimal binary split tree in polynomial time in the presence of non-distinct access probabilities.

We thus present an algorithm which calculates the value to be used (without the extra loop), resulting in construction of an optimal BST for records with distinct access probabilities in $\Theta(n^4)$ time, while still using only $\Theta(n^3)$ space. This algorithm also constructs an optimal BST for records with non-distinct access probabilities in $O(n^5)$ time by saving some extra $\big($still $\Theta(n^3)\big)$ information to allow postponing top-down decisions until sufficient constraints are accumulated to eliminate the exponential cost of these decisions.

0. Definitions and Data Structures

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 $i < j$. If the records are not so ordered, we can prepend a sort to the algorithm without adversely affecting its asymptotic costs. 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 definition of the $q(i)$s above in a uniform fashion and to simplify the algorithms, we define $Key(0) = -\infty$, $Key(n+1) = \infty$, and $p(0) = p(n+1) = 0$.

For the following definitions, assume that access probabilities are distinct. The next section gives modifications for enabling the use of non-distinct access probabilities.

Define a *range* of records i to j to be the 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 to be placed in the root of some higher subtree. Perl gives 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.

$R[i, j, k]$, the *root index*, contains the index of the record with the maximum access probability over the probabilities in $\langle i, j, k\rangle$. This gives the index of the record which must be the root of any subtree spanning $\langle i, j, k\rangle$.

$SP[i, j, k]$, the *split index*, gives the index of the record which has the key to be used as the split value for the root of an optimal subtree spanning $\langle i, j, k\rangle$. Our split value is the largest key among the records in the range of the left subtree.

$k_L[i, j, k]$ is the number of keys missing from the left subtree of an optimal BST X which spans $\langle i, j, k\rangle$ and uses a split value of $Key(SP[i, j, k])$. The number of keys missing from the right subtree of X is then $k+1-k_L[i, j, k]$ (the extra $+1$ is to account for the fact that the root of X is missing from one of the two subtrees).

This value is necessary for the final construction of the tree. For example, the root of the left subtree of X is $R\big[i, SP[i,j,k], k_L[i,j,k]\big]$. Note that, for any possible i, j, k and optimal split index l, $k_L[i,j,k]$ is simply the number of records in the range i to l with access probabilities greater than or equal to $p\big(R[i,j,k]\big)$. Therefore, for a fixed split index l, $k_L[i,j,k]$ is a fixed value.

The previous algorithms [1, 4] assumed that the missing records could be distributed in many ways between the left and right subtrees. For each possible i, j, k and split index l, they executed an inner loop (contributing a factor of n to the asymptotic time complexity of the algorithm) to consider values for $k_L[i,j,k]$ corresponding to all distributions of $k+1$ missing records between left and right subtrees. Since this value is needed for the final construction of the tree, it must either be saved or re-calculated. The added calculation slows the final construction by a factor of $O(n)$, but does not affect the overall asymptotic complexity of the algorithm. The previous algorithms did not save this value, but the implications were not obvious since they did not present the final procedure for constructing the tree. Saving the information adds space to the algorithm, not affecting the asymptotic complexity of space required. Thus the decision to save the values is a direct time–space tradeoff, and must be made depending on the time and space constraints of the individual application. Our decision to store the information for later use was not due to a preference to use space for reducing time, but to clarify the use of the same data in two or more sections of the algorithm. All other arrays which were not used in the previous algorithms are a result of the addition of handling non-distinct access probabilities.

We now derive recurrence relations to determine the value of $k_L[i,j,k]$. Let the function $GT_L(i,j,k,l)$ be 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 that of the root of T. Similarly define $EQ_L(i,j,k,l)$ for records which have probability equal to that of the root of T. Finally define $GE_L(i, j, k, l) = GT_L(i, j, k, l) + EQ_L(i, j, k, l)$. Note that, when access probabilities are distinct, $EQ_L(i, j, k, l)$ is either 1 or 0, depending on whether the root of T is in the range of the left or right subtree of T, and that $GE_L(i, j, k, l)$ is the number of records which are missing from the left subtree. In this case, $k_L[i, j, k] = GE_L\big(i, j, k, SP[i, j, k]\big)$.

Fortunately, $GT_L(i, j, k, l)$ and $EQ_L(i, j, k, l)$ can be calculated in terms of $GT_L(i, j, k, l{-}1)$ and $EQ_L(i, j, k, l{-}1)$, so that only the last (in terms of l, the index of a loop) values calculated need to be stored at any given time. If $l = i$, then the left subtree is empty and so $GT_L(i, j, k, l)$ and $EQ_L(i, j, k, l)$ are both zero. Otherwise, consider a subtree X which spans $\langle i, j, k\rangle$ and is split at $l-1$. Since T and X have the same root, moving r_l from the right subtree of X to the left subtree of X (forming T) either has no effect on the two counts (if the probability of r_l is less than that of the root of T) or adds exactly 1 to one of the two counts (depending on whether r_l is greater than or equal to the root of T). This leads to the following recurrence for $GT_L(i, j, k, l)$:

$$GT_L(i, j, k, l) = \begin{cases} 0 & l = i \\ GT_L(i, j, k, l{-}1) & l \neq i, \quad p(l) \leq p\big(R[i, j, k]\big) \\ GT_L(i, j, k, l{-}1) + 1 & l \neq i, \quad p(l) > p\big(R[i, j, k]\big) \end{cases}$$

Substituting 'EQ' for 'GT', '$\neq$' for '$\leq$' and '$=$' for '$>$' in this recurrence gives the recurrence for $EQ_L(i, j, k, l)$. A similar recurrence could also be constructed for $GE_L(i, j, k, l)$, which is simpler and sufficient for distinct access probabilities, but the two separate values are necessary in the next section when we relax the constraint on access probabilities. Since only the previous values (in terms of l) are needed at any given time, our algorithm just uses the scalar variables GT_L and EQ_L within the loops which consider possible values for i, j, k and l, but the more verbose functional definition of $GE_L(i, j, k, l)$ is useful for clarity in the following definitions.

$W[i,j,k]$ is the weight of a subtree spanning $\langle i,j,k\rangle$, which is defined as

$$W[i,j,k] = \sum_{p(l)\in\langle i,j,k\rangle} p(l) + \sum_{l=i}^{j-1} q(l)$$

$COT[i,j,k]$ is the cost of an optimal subtree spanning $\langle i,j,k\rangle$, which is defined as

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

1. Non-Distinct Access Probabilities

The definitions in the previous section permit design of a $\Theta(n^4)$ time and $\Theta(n^3)$ space dynamic programming algorithm for constructing optimal BSTs similar to Knuth's algorithm [3] for constructing optimal binary search trees. We now present extensions of these definitions which lead to an algorithm that allows non-distinct access probabilities with the same space complexity and requires at most an extra factor of $O(n)$ time. Thus the algorithm requires $O(n^5)$ time, but this is only an upper bound based on large numbers of equal access probabilities. When access probabilities are distinct, this algorithm requires only $\Theta(n^4)$ time.

The major problem when there may be non-distinct access probabilities is that, during the calculation of COT, SP and k_L, the root of a given subtree may be unknown, since it could be any one of a set of non-missing records with maximal access probabilities. It may even be unknown which records are in this set, i.e. which of the records with access probabilities equal to the root are not missing. The previous algorithm [4] shifted to a top-down approach at this point, which resulted in an exponential time complexity. We note that the only pieces of information needed during the calculation of COT, SP and k_L are the *weights* of the subtrees, and that these weights are not dependent on which one of the records with equal access probabilities is the root. The only problem is in predicting from which subtree the root, that eventually will be picked, is missing. This is *not* fixed, so we check all possible distributions of potential roots between the subtrees without ever committing to exactly which record is the root of the current subtree. The final decisions will be made in a top-down fashion when the tree is constructed, at which time only the optimal subtrees are considered, and thus the exponential work is avoided.

For the following definitions required by our algorithm, $OBST$, let T be any tree spanning $\langle i, j, k \rangle$ and let P be the access probability of any key that might be the root of T. When we compare records, saying one is greater, less, etc. than another, we are referring to the access probabilities of those records. This also applies when we compare a record to P.

We refine the definition of $R[i, j, k]$ to be the index of the *rightmost* possible root of T. This gains only a minor savings in time, but calculation of the following arrays supplies this information at no additional cost.

Let $EQ[i, j, k]$ be the number of records with indices in the range i to j which are equal to P (recall that P is determined, in part, by k). Similarly, let $LT[i, j, k]$ be the number of records which are less than P. We define the function $GT(i, j, k) = j - i - \big(LT[i, j, k] + EQ[i, j, k]\big)$ as the number of records in the range of T which are greater than P. Also, we define $EQ_m(i, j, k) = k - GT(i, j, k)$ as the number of records with indices in the range i to j which are equal to P and missing from T (note that k includes all records which are greater than P and possibly some that are equal to P). Since $GT(i, j, k)$ and $EQ_m(i, j, k)$ can be calculated from other known values, they are not stored by the algorithm, but are used for notational convenience.

Note that, although the value of $GE_L(i, j, k, l)$ $\big(= GT_L + EQ_L\big)$ was equal to the number of records missing from the left subtree when only distinct access probabilities were considered, this is not true in $OBST$, since some unknown number of the records counted in EQ_L may not be missing. We define EQ_{mL} to be the number of records counted in EQ_L which are missing from the left subtree of T. The number of missing records in the left subtree thus is represented in $OBST$ by the value of $GT_L + EQ_{mL}$, which eventually will be stored in $k_L[i,j,k]$ after the optimal value of EQ_{mL} is found.

There may be many possible values of EQ_{mL}, which correspond to decisions about whether the roots of T and subtrees of T are chosen from the left or right of their respective subtrees. When looking for optimal splits, we bound the possible values of EQ_{mL} and then check all values within our bounds. This does not determine a root for T, but provides constraints which are used during the final top-down construction of the tree to ensure that the root picked is consistent with the remainder of the tree. Define $EQ_R = EQ[i,j,k] - EQ_L$ and $EQ_{mR} = EQ_m(i,j,k) + 1 - EQ_{mL}$. Since these values can be calculated from other known values, they are not stored by our algorithm, but are calculated as needed. They are defined here for clarity in the following bounds.

Bounds on EQ_{mL}:

(1) (number of keys missing from left subtree)

$\leq$ (number of keys missing from both subtrees)

(2) $GT_L + EQ_{mL} \leq k + 1$ rewriting (1)

(3) $EQ_{mL} \leq EQ_L$

(∗) $EQ_{mL} \leq \min\{EQ_L, k + 1 - GT_L\}$ from (2) and (3)

(4) $EQ_R \geq EQ_{mR}$

(5) $EQ_R \geq EQ_m(i,j,k) + 1 - EQ_{mL}$ rewriting (4)

(6) $EQ_{mL} \geq 0$

(∗∗) $EQ_{mL} \geq \max\{0, EQ_m(i,j,k) + 1 - EQ_R\}$ from (5) and (6)

Thus, for any l splitting T (and the values of EQ_L and GT_L corresponding to that split), (∗) and (∗∗) give us

$$\max\{0, EQ_m(i,j,k) + 1 - EQ_R\} \leq EQ_{mL} \leq \min\{EQ_L, k + 1 - GT_L\}$$

Note that, any time there is only one possible root, these bounds restrict EQ_{mL} to either 1 or 0, depending on whether the single possible root of T is in the left or right subtree of T. Thus, the extra factor of n on the time of this algorithm is only an upper bound; the algorithm is $o(n^5)$ $\big($approaching $\Theta(n^4)\big)$ when there are few records with equal access probabilities, and is $\Theta(n^4)$ when records have distinct access probabilities.

The calculation of $W[i,j,k]$ is complicated by the fact that, when a record with index j such that $p(j)\!=\!P$ is being considered by the dynamic programming processes, it is sometimes unclear whether record j is present or missing from the subtree. It is simple enough when $EQ_m(i,j,k) = 0$, since record j must be present if $j\!=\!P$ and no records equal to P are missing from the subtree. When $EQ_m(i,j,k) > 0$, we do not know which of the records that are equal to P are missing, but we *do* know their weight and how many of them there are. Thus we avoid making any decision about whether record j is missing by subtracting $p(j)\text{·}EQ_m(i,j,k)$ (= the total weight of the records which are equal to P and missing from the subtree) from $W\big[i,j,k\!-\!EQ_m(i,j,k)\big]$ (= the weight of the subtree with *none* of the records equal to P missing).

We construct the tree in a natural top-down fashion based on the values of R, SP, k_L and Key as before, but the choice of the root for each subtree is made in postorder, after the subtrees below it have been fully constructed. A global array of flags is used to indicate which records have been allocated as roots so far, and the choice of the root for a subtree is restricted to any record in the range of the subtree which has the correct access probability and has not already been allocated as a root of some lower subtree. We search backwards from the rightmost possible root in the range, which may save a little time, but still yields an $O(n)$ search for each root, making the time required to construct the tree (after the arrays have been set up) $O(n^2)$. Thus the total time for the algorithm is dominated by the $O(n^5)$ time required to calculate COT, SP, and k_L.

2. The Algorithm OBST

We now present the algorithm $OBST$ for calculating an optimal BST when there may be non-distinct access probabilities. The output of $OBST$ is the variable *Tree*, which points to the root of an optimal BST for the given input. The input value n and input functions Key, p and q are global to all procedures. The internal arrays R, W, COT, SP, k_L, EQ, LT and $FLAG$ are also global to all procedures.

```
$OBST(n, Key, p, q, Tree)$:
                              /* calculate optimal BST for non-distinct access probabilities */
begin
    $InitR()$
    $InitW()$
    $Compute()$
    for $i \leftarrow 1$ until $n$ do
        $FLAG[i] \leftarrow$ 'free'
    $Tree \leftarrow Build\_Tree(0, n+1, 0)$
end


$InitR()$:                                                  /* initialize $R$, $EQ$, and $LT$ */
for $i \leftarrow 0$ until $n-1$ do begin
    $R[i, i+1, 0] \leftarrow i+1$ ,      $R[i, i+1, 1] \leftarrow 0$
    $EQ[i, i+1, 0] \leftarrow 1$ ,       $EQ[i, i+1, 1] \leftarrow 0$
    $LT[i, i+1, 0] \leftarrow 0$ ,       $LT[i, i+1, 1] \leftarrow 0$
    for $j \leftarrow i+2$ until $n+1$ do begin
        if $p(j) > p\big(R[i, j-1, 0]\big)$ then begin                 /* new root */
            $R[i, j, 0] \leftarrow j$
            $EQ[i, j, 0] \leftarrow 1$
        end else if $p(j) = p\big(R[i, j-1, 0]\big)$ then begin   /* rightmost root */
            $R[i, j, 0] \leftarrow j$
            $EQ[i, j, 0] \leftarrow EQ[i, j-1, 0] + 1$
        end else begin                                          /* less than root */
            $R[i, j, 0] \leftarrow R[i, j-1, 0]$
            $EQ[i, j, 0] \leftarrow EQ[i, j-1, 0]$
        end
        $LT[i, j, 0] \leftarrow j - i - EQ[i, j, 0]$
        for $k \leftarrow 1$ until $j - i - 1$ do
            $CheckR(i, j, k)$
    end
end
```

```
$CheckR(i, j, k)$:                       /* check general conditions for $R$, $EQ$, and $LT$ */
if  $p(j) > p\big(R[i, j-1, k-1]\big)$  then begin                                /* missing */
        $R[i, j, k] \leftarrow R[i, j-1, k-1]$
        $EQ[i, j, k] \leftarrow EQ[i, j-1, k-1]$
        $LT[i, j, k] \leftarrow LT[i, j-1, k-1]$
end else if  $p(j) > p\big(R[i, j-1, k]\big)$  then begin                         /* new root */
        $R[i, j, k] \leftarrow j$
        $EQ[i, j, k] \leftarrow 1$
        $LT[i, j, k] \leftarrow LT[i, j-1, k] + EQ[i, j-1, k]$
end else if  $p(j) = p\big(R[i, j-1, k]\big)$  then begin                   /* rightmost root */
        $R[i, j, k] \leftarrow j$
        $EQ[i, j, k] \leftarrow EQ[i, j-1, k] + 1$
        $LT[i, j, k] \leftarrow LT[i, j-1, k]$
end else begin                                                        /* less than root */
        $R[i, j, k] \leftarrow R[i, j-1, k]$
        $EQ[i, j, k] \leftarrow EQ[i, j-1, k]$
        $LT[i, j, k] \leftarrow LT[i, j-1, k] + 1$
end


$InitW()$:                                                              /* initialize $W$ */
begin
        $W[n, n+1, 0] \leftarrow q(n)$
        $W[n, n+1, 1] \leftarrow q(n)$
        for  $i \leftarrow 0$  until  $n-1$  do begin
                $W[i, i+1, 0] \leftarrow q(i) + p(i+1)$
                $W[i, i+1, 1] \leftarrow q(i)$
                for  $j \leftarrow i+2$  until  $n+1$  do begin
                        $W[i, j, 0] \leftarrow W[i, j-1, 0] + q(j-1) + p(j)$
                        for  $k \leftarrow 1$  until  $j-i$  do
                                if  $p(j) > p\big(R[i, j-1, k-1]\big)$  then               /* missing */
                                        $W[i, j, k] \leftarrow W[i, j-1, k-1] + q(j-1)$
                                else if  $p(j) < p\big(R[i, j-1, k]\big)$  or  $EQ_m(i, j, k) = 0$  then
                                                                                /* less than root */
                                                                 /* or no records equal to root missing */
                                        $W[i, j, k] \leftarrow W[i, j-1, k] + q(j-1) + p(j)$
                                else                                 /* equal to root and maybe missing */
                                        $W[i, j, k] \leftarrow W\big[i, j, k - EQ_m(i, j, k)\big] - p(j) \cdot EQ_m(i, j, k)$
                end
        end
end
```

```
$Compute()$:                                   /* initialize $COT$, $SP$, and $k_L$ */
begin
    for $i \leftarrow 0$ until $n$ do begin
        $COT[i, i{+}1, 0] \leftarrow W[i, i{+}1, 0]$
        $COT[i, i{+}1, 1] \leftarrow W[i, i{+}1, 1]$
    end
    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
                $Find\_Min(i, j, k)$
            $COT[i, j, d] \leftarrow W[i, j, d]$
        end
end
```

```
$Find\_Min(i, j, k)$:              /* find optimal $COT$, $SP$, and $k_L$ given $i, j, k$ */
begin
    $GT_L \leftarrow 0$
    $EQ_L \leftarrow 0$
    $minc \leftarrow \infty$
    for $l \leftarrow i + 1$ until $j - 1$ do begin
        if $p(l) > p\big(R[i, j, k]\big)$ then
            $GT_L \leftarrow GT_L + 1$
        if $p(l) = p\big(R[i, j, k]\big)$ then
            $EQ_L \leftarrow EQ_L + 1$
        for $EQ_{mL} \leftarrow \max\{0, EQ_m(i, j, k) + 1 - EQ_R\}$
                                            /* recall that $EQ_R = EQ[i, j, k] - EQ_L$ */
        until $\min\{EQ_L, k + 1 - GT_L\}$ do begin
            $try \leftarrow COT[i, l, GT_L + EQ_{mL}] + COT\left[l, j, k{+}1-(GT_L + EQ_{mL})\right]$
            if $try < minc$ then begin
                $minc \leftarrow try$
                $minl \leftarrow l$
                $mink \leftarrow GT_L + EQ_{mL}$
            end
        end
    end
    $COT[i, j, k] \leftarrow minc + W[i, j, k]$
    $SP[i, j, k] \leftarrow minl$
    $k_L[i, j, k] \leftarrow mink$
end
```

```
$BUILD\_TREE(i,j,k)$:
                    /* return pointer to root of optimal subtree spanning $\langle i,j,k\rangle$ */
begin
    if $i=n$ or $k=j-i$ or $R[i,j,k]=0$ or $(j=n+1$ and $k=j-i-1)$ then
        $node \leftarrow$ null pointer
    else begin
        $node \leftarrow$ pointer to a new tree node
        $node.SPLIT \leftarrow Key\big(SP[i,j,k]\big)$
        if $i \geq j-1$ then begin
            $node.LEFT \leftarrow NULL$
            $node.RIGHT \leftarrow NULL$
        end
        else begin
            $node.LEFT \leftarrow BUILD\_TREE\big(i, SP[i,j,k], k_l[i,j,k]\big)$
            $node.RIGHT \leftarrow BUILD\_TREE\big(SP[i,j,k], j, k+1-k_l[i,j,k]\big)$
        end
        $x \leftarrow R[i,j,k]$
        while $p(x) \neq p\big(R[i,j,k]\big)$ or $FLAG[x] \neq$ 'free' do
            $x \leftarrow x-1$
        $FLAG[x] \leftarrow$ 'used'
        $node.KEY \leftarrow Key(x)$
    end
    return $node$
end
```

Conclusions and Open Questions

An algorithm has been presented for constructing optimal binary split trees in $\Theta(n^4)$ time when access probabilities are distinct, and $O(n^5)$ time when access probabilities are non-distinct. Taking into account the added complexity of choosing split values and assuming the necessity of an extra $O(n)$ time to allow non-distinct access probabilities, the efficiency of this algorithm is comparable to that of Knuth's $O(n^3)$ algorithm for finding an optimal binary search tree. Since Perl [4] showed that the technique used by Knuth [3] to obtain an $O(n)$ speedup for optimal binary search trees (reducing the time to $O(n^2)$) cannot be applied to optimal BSTs, an open question arises as to whether or not there is some other technique (perhaps similar to Knuth's) that *can* be applied to BSTs to reduce the time of the algorithm presented here. One technique worth consideration is the Quadrangle Inequality presented by Yao [6].

Another open question is that of the relative value of the algorithm presented here with that of the $\Theta(n^5)$ algorithm presented by Huang and Wong [2] for generalized split trees (which handles equi-probable keys). They show by simulation that optimum generalized split trees can have faster expected search times than optimum split trees, but that the difference appears never to be great. However, there is no theoretic evidence to that effect.

References

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

2. S.-H. S. Huang and C. K. Wong, Generalized Binary Split Trees, *Acta Informatica* **21** (1984) 113–123.

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

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

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

6. F. F. Yao, Efficient Dynamic Programming Using Quadrangle Inequalities, *The 12th Annual ACM Symposium on Theory of Computing* Los Angeles, Calif., (April 28–30, 1980) 429–435.