Number 7

OPTIMAL BINARY SEARCH TREES

Suppose we are given a set of n keys, $K_1 < K_2 < \ldots < K_n$, which are to be stored in a binary search tree. After the tree has been constructed, only search operations will be performed — there will be no insertions or deletions. We are also given a probability density function P, where $P(i)$ is the probability of searching for key K_i. There are many different binary search trees in which the n given keys can be stored. For a particular tree T with these keys, the average number of comparisons to find a key, for the given probability density P, is

$$\sum_{i=1}^{n} P(i) \cdot \big(\mathrm{depth}_T(K_i) + 1\big),$$

with $\mathrm{depth}_T(K_i)$ denoting the depth of the node where K_i is stored in T. The problem we would like to solve is to find, among all the possible binary search trees that contain the n keys, one which minimises this quantity. Such a tree is called an *optimal binary search tree.* Note that there may be several optimal binary search trees for the given density function. This is why we speak of an optim*al,* rather than the optim*um,* binary search tree.

A simple way to accomplish this is to try out all possible binary trees with n nodes, computing the average number of comparisons to find a key in each tree considered, and selecting a tree with the minimum average. Unfortunately, this simple strategy is ridiculously inefficient because there are too many trees to try out. In particular, there are $\binom{2n}{n}/(n+1)$ different binary trees with n nodes (if interested in the derivation of this formula, see Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms,* pp. 388–389). Thus, if there are 20 keys, we have to try out 131,282,408,400 different trees. Computing the average number of comparisons in each at the rather astonishing speed of 1 μsec per tree, will still take 2188 hours or approximately 91 days and nights of computing to find the optimal binary search tree (for just 20 keys)! Fortunately, there is a much more efficient, if less straightforward, way to find an optimal binary search tree.

Let T be a binary search tree that contains keys $K_i, K_{i+1}, \ldots, K_j$ for some $1 \le i \le j \le n$. We shall see shortly why it is useful to consider trees that contain subsets of successive keys. We define the *cost* of T, $c(T)$, as

$$c(T) = \sum_{l=i}^{j} P(l) \cdot \big(\mathrm{depth}_T(K_l) + 1\big).$$

Hence, if T contains all n keys (so $i = 1$ and $j = n$), the cost of T is precisely the expected number of comparisons to find a key for the given density function.† Thus, we can rephrase our problem as follows: Given a density function for the n keys, find a minimum cost tree with n nodes.

Before giving the algorithm to find an optimal binary search tree, we prove two key facts.

Lemma 1. *Let T be a binary search tree containing keys $K_i, K_{i+1}, \ldots, K_j$, and let T_L and T_R be the left and right subtrees of T respectively. Then,*

$$c(T) = c(T_L) + c(T_R) + \sum_{l=i}^{j} P(l).$$

† This is not so if T is missing some of the keys, because in that case the probabilities of the keys that are in T do not sum up to 1; then, P is not a proper density function relative to the set of keys in the tree.

PROOF: This is an easy consequence of the definition of cost of a tree. You should prove it on your own. □

Lemma 2. *Let T be a binary search tree that has minimum cost among all trees containing keys $K_i, K_{i+1}, \ldots, K_j$, and let K_m be the key at the root of T (so $i \leq m \leq j$). Then T_L, the left subtree of T, is a binary search tree that has minimum cost among all trees containing keys $K_i, K_{i+1}, \ldots, K_{m-1}$, and T_R, the right subtree of T, is a binary search tree that has minimum cost among all trees containing keys $K_{m+1}, K_{m+2}, \ldots, K_j$.*

PROOF: Proof: We prove the contrapositive. That is, if either the left or right subtree of T fails to satisfy the property asserted in the lemma we show that T does not really have the minimum possible cost among all trees that contain $K_i, K_{i+1}, \ldots, K_j$.

Let T'_L and T'_R be minimum cost binary search trees that contain keys $K_i, K_{i+1}, \ldots, K_{m-1}$ and keys $K_{m+1}, K_{m+1}, \ldots, K_j$ respectively. Then, $c(T'_L) \leq c(T_L)$ and $c(T'_R) \leq c(T_R)$.

Further, let T' be the tree with key K_m in the root, and left and right subtrees T'_L and T'_R respectively. Evidently, T' is a binary search tree that contains keys $K_i, K_{i+1}, \ldots, K_j$. If T_L or T_R do not have the property asserted by the lemma, then either $c(T_L) > c(T'_L)$ or $c(T_R) > c(T'_R)$. This, together with the inequalities stated in the previous paragraph, implies that $c(T_L) + c(T_R) > c(T'_L) + c(T'_R)$. From this and Lemma 1 we have

$$c(T) = c(T_L) + c(T_R) + \sum_{l=i}^{j} P(l) > c(T'_L) + c(T'_R) + \sum_{l=1}^{j} P(l) = c(T').$$

Thus, $c(T) > c(T')$, and T is not a minimum cost binary search tree among all trees that contain keys $K_i, K_{i+1}, \ldots, K_j$. □

Computing an Optimal Binary Search Tree

Lemma 2 is the basis of an efficient algorithm to find an optimal binary search tree. Let $T_{i,j}$ denote a binary search tree that has minimum cost among all trees that contain keys $K_i, K_{i+1}, \ldots, K_j$. Lemma 2 then says that $T_{i,j}$ must have as its root the key K_m for some m, and as its left and right subtrees $T_{i,m-1}$ and $T_{m+1,j}$ — minimum cost subtrees containing the keys $K_i, K_{i+1}, \ldots, K_{m-1}$ and $K_{m+1}, K_{m+2}, \ldots, K_j$ respectively. Thus, tree $T_{i,j}$ has the form shown in Figure 1. Since $T_{i,m-1}$ and $T_{m+1,j}$ are "smaller" trees than $T_{i,j}$, this suggests an inductive procedure, starting with small minimum cost trees (each containing just one key) and progressively building larger and larger minimum cost trees, until we have a minimum cost tree with n nodes — which is what we are looking for.

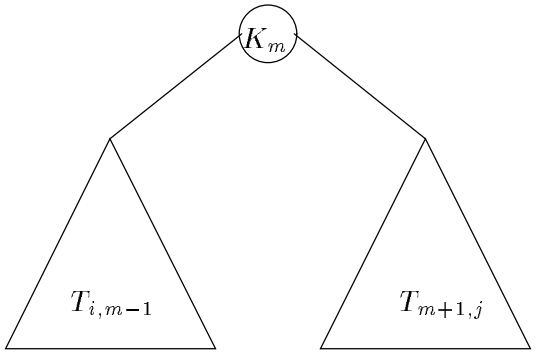

Figure 1

More specifically, we start the induction with minimum cost trees each containing exactly one key, and proceed by constructing minimum cost trees with $2, 3, \ldots, n$ successive keys. Note that there are exactly $n - d + 1$ groups of d successive keys, for $1 \leq d \leq n$. Thus, instead of considering *all* possible trees with n nodes we consider only n (minimum cost) trees with 1 node each, then $n - 1$ (minimum cost) trees with 2 nodes each, and so on, down to 1 minimum cost tree with n nodes; that is, we consider a total of $n(n+1)/2$ trees — much fewer than $\binom{2n}{n}/(n+1)$ trees.

So now the question is how to construct each $T_{i,j}$ by induction on $d = j - i + 1$, the number of keys it contains. The basis of the induction, $d = 1$, is trivial. In this case we have $j = i$ and the minimum cost binary search tree $T_{i,i}$ that stores K_i (in fact the only such tree) is a single node containing K_i. Its cost is $c(T_{i,i}) = P(i)$.

For the induction step, assume that, for some $d > 1$, we have already constructed and computed the costs of all the minimum cost trees with *fewer* than d successive keys. Now we want to construct and compute the cost of all minimum cost trees with exactly d successive keys. Consider such a tree $T_{i,j}$ (hence, $j - i + 1 = d$). Let $T_{i,m,j}$ be the tree with K_m in the root, and left and right subtrees $T_{i,m-1}$ and $T_{m+1,j}$ respectively. Lemma 2 implies that $T_{i,j}$ is the minimum cost tree among the $T_{i,m,j}$'s. Thus we can find $T_{i,j}$ simply by trying out all the $T_{i,m,j}$'s, for $m = i, i+1, \ldots, j$. In fact, Lemma 1 tells us how to compute $c(T_{i,m,j})$ efficiently, so that "trying out" each possible m will not take too long. Since $T_{i,m-1}$ and $T_{m+1,j}$ both have fewer than d keys, we have already (inductively) computed $T_{i,m-1}$ and $T_{m+1,j}$ and their costs, $c(T_{i,m-1})$ and $c(T_{m+1,j})$. Lemma 1 then tells us how to get $c(T_{i,m,j})$ in terms of these. Note that when $m = i$ the left subtree of $T_{i,m,j}$ is $T_{i,i-1}$, and when $m = j$ the right subtree of $T_{i,m,j}$ is $T_{j+1,j}$. We define $T_{i,j}$ to be empty if $i > j$, and we define the cost of an empty tree to be 0.

Figure 2 shows this algorithm in pseudo-code. The algorithm takes as input an array $Prob[1..n]$, which specifies the probability density ($Prob[i] = P(i)$). It computes two two-dimensional arrays, *Root* and $Cost$, where $Root[i,j]$ is the root of $T_{i,j}$, and $Cost[i,j] = c(T_{i,j})$, for all i and j such that $1 \leq i \leq j \leq n$.† To help compute *Root* and $Cost$ the algorithm maintains a third array, $SumOfProb$, where $SumOfProb[i] = \sum_{l=1}^{i} P(l)$ for $1 \leq i \leq n$, and $SumOfProb[0] = 0$. Note that $\sum_{l=i}^{j} P(l) = SumOfProb[j] - SumOfProb[i-1]$.

The algorithm of Figure 2 does not explicitly construct an optimal binary search tree, but such a tree is implicit in the information in array *Root*. As an exercise you should write an algorithm which, given *Root* and an array $Keys[1..n]$, such that $Keys[i] = K_i$, constructs an optimal binary search tree.

It is easy to see that the time complexity of this algorithm is dominated by the number of times the innermost (**for** m) loop is executed. This is

$$\sum_{d=1}^{n} d(n-d) = n^2(n+1)/2 - n(n+1)(2n+1)/6 = n(n+1)(n-1)/6$$

which is in $\Theta(n^3)$. A slight modification of this algorithm leads to an algorithm with complexity in $\Theta(n^2)$ (if interested, see D.E. Knuth, "Optimum binary search trees", *Acta Informatica,* Vol. 1 (1971), pp. 14–25.)

† For technical reasons that will become apparent when you look at the algorithm carefully, we need to set $Cost[i, i-1] = 0$ for $1 \leq i \leq n+1$. Recall that $T_{i,i-1}$ is empty and thus has cost 0.

```
OptimalBST(Prob[1..n])

% Initialisation %

SumOfProb[0] := 0

for i := 1 to n do
    SumOfProb[i] := Prob[i] + SumOfProb[i − 1]
    Root[i, i] := i
    Cost[i, i] := Prob[i]
end for

for i := 1 to n + 1 do
    Cost[i, i − 1] := 0
end for

% Compute information about trees with d > 1 successive keys. %

for d := 2 to n do

    % Compute Root[i, j] and Cost[i, j] for each i and j with j − i + 1 = d. %

    for i := 1 to n − d + 1 do
        j := i + d − 1
        MinCost := +∞

        % Find m between i and j so that c(T_{i,m,j}) is minimised. %

        for m := i to j do
            c := Cost[i, m − 1] + Cost[m + 1, j] + (SumOfProb[j] − SumOfProb[i − 1])
            if c < MinCost then
                MinCost := c
                r := m
            end if
        end for
        Root[i, j] := r
        Cost[i, j] := MinCost
    end for
end for
```

Algorithm for Optimal Binary Search Tree

Figure 2

Unsuccessful Searches

In the preceding discussion we have only considered successful searches. However, if we take into account unsuccessful searches, it is possible that the constructed tree is no longer optimal. Fortunately, this problem can be taken care of in a straightforward manner. To find an optimal binary search tree in the case where both successful and unsuccessful searches are taken into account, we must know the probability density for both successful and unsuccessful searches. So, in addition to $P(i)$ we must also be given $Q(i)$ for $0 \leq i \leq n$, where

- $Q(0)$ is the probability of searching for keys less than K_1;
- $Q(i)$ is the probability of searching for keys between K_i and K_{i+1} (exclusive), for $1 \leq i \leq n$;
- $Q(n)$ is the probability of searching for keys greater than K_n.

Thus, Q describes the probability of unsuccessful searches.

In each binary search tree containing $K_1, K_2, \ldots, K_n$, we add $n+1$ external nodes $E_0, E_1, \ldots, E_n$. This is illustrated below, in Figure 3; the external nodes are drawn in boxes, as usual.

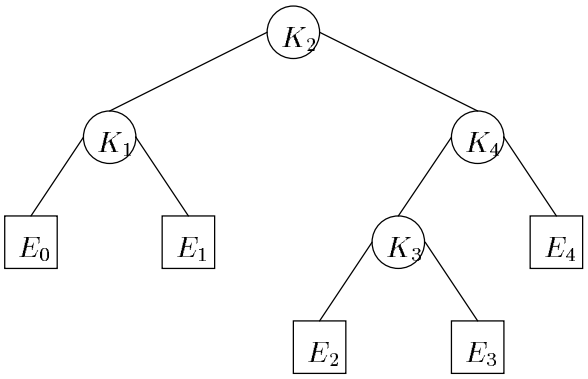

Figure 3

The average number of comparisons for a successful *or* unsuccessful search in such a tree T is

$$\sum_{i=1}^{n} P(i) \cdot \big(\mathrm{depth}_T(K_i) + 1\big) + \sum_{i=0}^{n} Q(i) \cdot \mathrm{depth}_T(E_i).$$

The left term is the average number of comparisons for successful searches, and the right term is the average number of comparisons for unsuccessful searches. Now we want to find a tree that minimises this quantity.

We can proceed exactly as before, except that the definition of the cost of a tree T with keys $K_i, K_{i+1}, \ldots, K_j$ is slightly modified to account for the unsuccessful searches. Namely, it becomes

$$c'(T) = \sum_{l=i}^{j} P(l) \cdot \big(\mathrm{depth}_T(K_l) + 1\big) + \sum_{l=i-1}^{j} Q(l) \cdot \mathrm{depth}_T(E_l).$$

With this cost function, Lemma 1 is slightly different:

Lemma 1′. *Let T be a binary search tree containing keys $K_i, K_{i+1}, \ldots, K_j$, and let T_L and T_R be the left and right subtrees of T respectively. Then,*

$$c'(T) = c'(T_L) + c'(T_R) + Q(i-1) + \sum_{l=i}^{j} \big(P(l) + Q(l)\big).$$

Everything else works out exactly as before. In particular, Lemma 2 is still valid (check this!). As an exercise show how to modify the algorithm in Figure 2 to account for these changes.