Analysis of Algorithms I: Optimal Binary Search Trees

Xi Chen

Columbia University

Given a set of n keys $K = \{k_1, \ldots, k_n\}$ in sorted order:

$$k_1 < k_2 < \cdots < k_n$$

we wish to build an optimal binary search tree with keys from K to minimize the expected number of comparisons needed for each search operation. We consider the following setting slightly simpler than the one discussed in Section 15.5 of the textbook. Assume we know in advance that for each search operation in the future, the key k to search for always comes from K and satisfies

$k = k_i$ with probability p_i, for each $i = 1, 2, \ldots, n$;

This implies that

$$\sum_{i=1}^{n} p_i = 1$$

Let T be a binary search tree with keys $k_1, \dots, k_n$. So T has n nodes and each node is labelled with a key k_i. We use $\mathrm{depth}_T(k_i)$ to denote the depth of the node labelled with k_i **plus one** (so if k_r is the key at the root, then we set $\mathrm{depth}_T(k_r) = 1$ instead of 0). It is clear that if the key we search for is $k = k_i$, then the number of comparisons needed is exactly $\mathrm{depth}_T(k_i)$.

Thus, the expected number of comparisons is

$$\sum_{i=1}^{n} p_i \cdot \mathrm{depth}_T(k_i)$$

and we will refer to it as $\mathrm{cost}(T)$, the cost of tree T. The goal is then to find an optimal binary search tree T for $K = \{k_1, \ldots, k_n\}$ with the minimum cost. As usual, we start by describing a dynamic programming algorithm that computes the minimum cost. It can be then used to construct an optimal BST.

Note the difference between this problem and Huffman trees. In the latter we only need to build a tree (instead of a binary search tree) in which each leaf is labelled with a character. Also in Huffman trees, the cost of a tree is the expected depth of leaves. Here the cost is the expected depth over all nodes.

Again, we use dynamic programming. To this end, we first need to figure out the optimal substructure of the problem, which will then lead us to a recursive formula that reduces the problem to smaller subproblems. Here the first choice to make, when constructing a binary search tree for K, is which key to put at the root of the tree.

Assume the key at the root is k_r. Denote the tree by T, its left subtree by T_L, and its right subtree by T_R. We know T_L has keys $k_1, \ldots, k_{r-1}$ and T_R has keys $k_{r+1}, \ldots, k_n$. Then for any k_i, $i < r$:

$$\text{depth}_T(k_i) = \text{depth}_{T_L}(k_i) + 1$$

For any k_j, $j > r$, we have:

$$\text{depth}_T(k_j) = \text{depth}_{T_R}(k_j) + 1$$

As a result, we have

$$
\begin{aligned}
&\text{cost}(T) \\
&= \sum_{i=1}^{n} p_i \cdot \text{depth}_T(k_i) \\
&= 1 \cdot p_r + \sum_{i<r} p_i \cdot \big(\text{depth}_{T_L}(k_i) + 1\big) + \sum_{j>r} p_j \cdot \big(\text{depth}_{T_R}(k_j) + 1\big) \\
&= \text{cost}(T_L) + \text{cost}(T_R) + \sum_{i=1}^{n} p_i = 1 + \text{cost}(T_L) + \text{cost}(T_R)
\end{aligned}
$$

This equation implies the following: Denote by c_{r-1} the cost of an optimal BST with keys $k_1, \ldots, k_{r-1}$, and by c'_{r+1} the cost of an optimal BST with keys $k_{r+1}, \ldots, k_n$. Then the minimum cost of a binary search tree for K, that has k_r as its root, is exactly

$$1 + c_{r-1} + c'_{r+1}$$

This gives us the following recursive algorithm.

Naive Optimal Binary Search Tree:

1. For $r = 1$ to n do
2. make a recursive call to compute the cost of an optimal BST for $\{k_1, \ldots, k_{r-1}\}$; store it in $C[r-1]$
   Note when $r = 1$, the cost of an empty BST is 0
3. make a recursive call to compute the cost of an optimal BST for $\{k_{r+1}, \ldots, k_n\}$; store it in $C'[r+1]$
   Note when $r = n$, the cost of an empty BST is 0
4. output

$$1 + \min_{r=1,\ldots,n} \left[C[r-1] + C'[r+1] \right]$$

However, the worst-case running time of this naive recursive algorithm is exponential. Note that there are only about $\Theta(n^2)$ subproblems we make recursive calls to solve in its recursion tree. The reason why its recursion tree is huge is because we solve the same subproblem over and over.

Again, we use dynamic programming to give a more efficient algorithm: maintain a table to store solutions to subproblems already solved; and solve all the subproblems one by one, using the recursive formula we found earlier, in an appropriate order.

For this purpose, we introduce the following notation. We let

$$p_{i,j} = \sum_{i \le k \le j} p_k, \quad \text{for any } 1 \le i \le j \le n$$

Given $p_1, \ldots, p_n$, we can compute all $p_{i,j}$'s in $\Theta(n^2)$ time (how?).

Given $i, j : 1 \leq i \leq j \leq n$, we use $c_{i,j}$ to denote the minimum cost of an optimal binary search tree with keys $k_i, k_{i-1}, \ldots, k_j$. For any $i \in [n]$, we also set $c_{i,i-1} = 0$ for convenience (meaning that an empty binary search tree has cost 0). Then to obtain $c_{i,j}$, we have:

1. If the root is k_i, then the minimum cost is

$$0 + p_i + (c_{i+1,j} + p_{i+1,j}) = c_{i,i-1} + c_{i+1,j} + p_{i,j} \quad \text{as } c_{i,i-1} = 0$$

2. If the root is k_j, then the minimum cost is

$$(c_{i,j-1} + p_{i,j-1}) + p_j + 0 = c_{i,j-1} + c_{j+1,j} + p_{i,j} \quad \text{as } c_{j+1,j} = 0$$

3. If the root is k_r, where $r : i < r < j$, the minimum cost is

$$(c_{i,r-1} + p_{i,r-1}) + p_r + (c_{r+1,j} + p_{r+1,j}) = c_{i,r-1} + c_{r+1,j} + p_{i,j}$$

To summarize, we get the following recursive formula:

$$c_{i,j} = p_{i,j} + \min_{i \le r \le j} \left[c_{i,r-1} + c_{r+1,j} \right]$$

This gives us the following algorithm:

1. compute $p[i, j]$ for all $i, j : 1 \leq i \leq j \leq n$
2. create a table $c[i, j]$, where $i, j : 1 \leq i \leq j + 1 \leq n$
3. set $c[i, i-1] = 0$ and $c[i, i] = p_i$ for all $i \in [n]$
4. for k from 1 to $n - 1$ do
5. for i from 1 to $n - k$ do
6. set $j = i + k$ and set

$$c[i, j] = p[i, j] + \min_{i \leq r \leq j} \Big[c[i, r-1] + c[r+1, j] \Big]$$

7. output $c[1, n]$

Here we fill up the table in the following order. At the beginning, all entries with $j - i = -1$ (empty tree) and $j - i = 0$ (tree with one single node) are ready. Then we work on entries with

$$j - i = 1,\ j - i = 2,\ \ldots,\ j - i = n - 1.$$

Every time we work on an entry $c[i, j]$ with $j - i = k$, we know that all the entries $c[i', j']$ with $j' - i' < k$ have already been computed. Note that the recursive formula we use to compute $c[i, j]$ only involves entries $c[i', j']$ with $j' - i' < k$. So they are all ready, and we can compute $c[i, j]$ in time $O(j - i)$. It is easy to check that the total running time is $\Theta(n^3)$.