Optimal Binary Search Trees

For the third example of dynamic programming, we present material that, while found in the course textbook, is derived from a presentation in Aho, Hopcroft, and Ullman, *The Design and Analysis of Computer Algorithms*. For this example, the problem to be solved is that of constructing a binary search tree that minimizes expected cost of searching given a probability distribution defined over a set of possible queries. More formally, if we are given a set of elements to be stored in a binary search tree, $S = \{s_1, \ldots, s_n\}$ and a sequence σ of MEMBER operations on the binary search tree, we want to construct a binary search tree such that we can process σ using the smallest expected number of comparison. The issue facing us with this problem is that, because ultimate performance depends upon the sequence σ, we need to know the probabilities of certain members being queried. Thus the final tree will not necessarily be a simple balanced tree.

Problem Statement

Let $s_1, \ldots, s_n$ be the set of elements in set S, sorted in increasing order. Let p_i be the probability that MEMBER(s_i, S) is in the sequence σ. We now consider three cases to define the probabilities for when the queried value s is not in S:

1. Let q_0 be the probability that MEMBER(s, S) is in σ, but $s < s_1$.
2. Let q_i be the probability that MEMBER(s, S) is in σ, but $s_i < s < s_{i+1}$.
3. Let q_n be the probability that MEMBER(s, S) is in σ, but $s_n < s$.

Assume we add $n + 1$ "fictitious" leaves to the binary search tree reflecting the elements $U - S$. Call these leaves $0, \ldots, n$ and number them from left to right. If some $s \in S$ is queried, then the number of vertices visited will be equal to $depth(s) + 1$. On the other hand, if some $s \notin S$ is queried and $s_i < s < s_{i+1}$, then the number of vertices visited will be equal to $depth(i)$ for fictitious leaf i. From this, we can compute the cost of a binary search tree given the probabilities defined above as follows:

$$C(T) = \sum_{j=1}^{n} p_j(depth(j) + 1) + \sum_{i=0}^{n} q_i depth(i).$$

Given this cost function, the problem is to construct a binary search tree that minimizes $C(T)$.

Optimal Substructure of Binary Search Trees

Let T_{ij} be a minimum cost binary search tree for a subset of elements of S given by $\{s_{i+1}, \ldots s_j\}$. Let $C(T_{ij})$ be the cost of T_{ij} as defined above. Let w_{ij} be the "weight" of T_{ij}, defined as follows:

$$w_{ij} = q_i + \sum_{k=i+1}^{j} (p_k + q_k).$$

Thus, for a particular subtree T_{ij}, the weight corresponds to the probability that a particular query will end up traversing that subtree. Note that this subtree has root $r_{ij} = s_k$ for some $s_k \in S$ and has two subtrees, $T_{i,k-1}$ and T_{kj}. The optimal substructure requires that these two subtrees already be optimal; therefore, the task is to find k, identifying s_k, that splits the data into the two subtrees such that T_{ij} is also optimal (i.e., of minimal cost).

In T_ij, notice that the depth of every vertex in the subtrees is increased by one from their respective depths in their subtrees. As a result, we can express the cost of T_{ij} as follows:

$$C(T_{ij}) = (w_{i,k-1} + C(T_{i,k-1})) + p_k + (w_{kj} + C(T_{kj})).$$

Recall that $w_{ij} = w_{i,k-1} + p_k + w_{kj}$, so this means we can rewrite the above as $C(T_{ij}) = w_{ij} + C(T_{i,k-1}) + C(T_{kj})$. At the leaves of the tree, we have that $w_{ii} = q_i$ and $C(T_{ii}) = 0$. To construct T_{ij}, we need to select

k such that $C(T_{i,k-1}) + C(T_{kj})$ is minimized. This leads to the following Bellman equation:

$$C(T_{ij}) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k \le j}\{w_{ij} + C(T_{i,k-1}) + C(T_{kj})\} & \text{otherwise} \end{cases}.$$

Constructing the Trees

Given the Bellman equation defined in the previous section, we can compute the value function and find the structure of the tree using the following pseudocode:

Algorithm 1 Value of Binary Decision Tree

```
MinRoot(T)
   for i ← 0 to n do
      w_ii ← q_i
      c_ii ← 0
   for l ← 1 to n do
      for i ← 0 to n − l do
         j ← i + l
         w_ij ← w_{i,j−1} + p_j + q_j
         m ← arg min_{i<k≤j} {c_{i,k−1} + c_{k,j}}
         c_ij ← w_ij + c_{i,m−1} + c_mj
         r_ij ← s_m
```

Notice that the identification of r_{ij} in the innermost loop identifies the root of T_{ij}. An additional data structure would be required to maintain the index of the root (corresponding to the k value), but notice that the index simply indexes into the original set S. After computation of the cost function and the tree, the actual tree can be recovered as follows:

Algorithm 2 Building the Tree

```
BuildTree(i, j)
   Create(v_ij)  // v_ij is the root of the subtree
   v_ij ← r_ij
   m ← index(r_ij)  // position of r_ij in S
   if i < m − 1 then
      v_ij, left ← BuildTree(i, m − 1)
   if m < j then
      v_ij.right ← BuildTree(m, j)
```

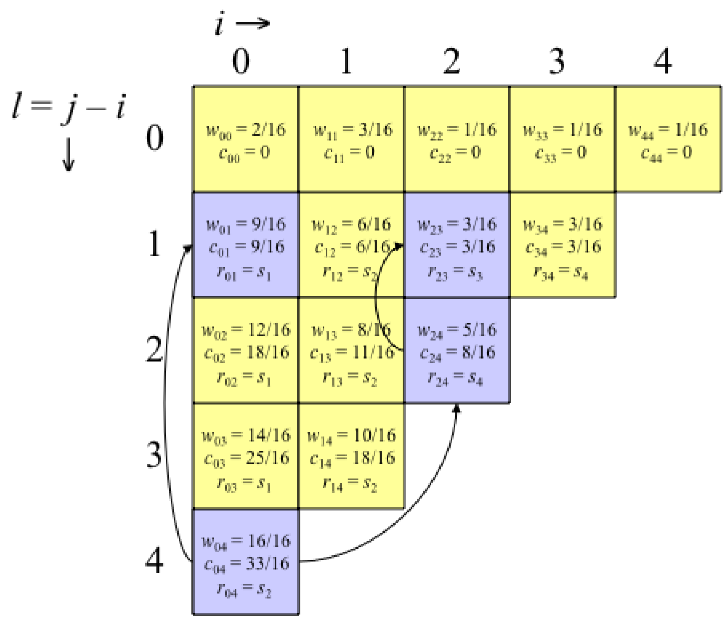
This function would be called with BuildTree$(0, n)$. At the start of the function, a node is created holding the root of the associated subtree. The limits i and j are then compared to the index of that element. If strict inequalities hold, then additional subtrees need to be constructed; otherwise, we have reached a leaf in the tree.

Example

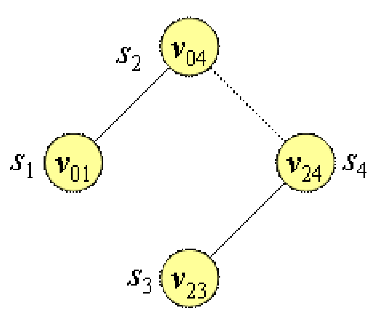
To illustrate the construction of an optimal binary search tree, suppose we have $s = \{s_1, s_2, s_3, s_4\}$ with the following probabilities for the various queries:

$$\begin{aligned} q &= \left\{\frac{1}{8}, \frac{3}{16}\frac{1}{16}, \frac{1}{16}\frac{1}{16}\right\} \\ p &= \left\{\frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16}\right\}. \end{aligned}$$

Then the MinRoot function will construct the cost table c and identify the appropriate tree structure as follows.



In this diagram, we start with $c(0, 4)$ to find the root of the tree. Here, we find that the root r_{04} is s2; therefore, the left subtree will have cost $c(0, 1)$, and the right subtree will have cost $c(2, 4)$. Considering the left subtree, r_{01} points to s_1. Since the costs of its left and right subtrees are $c(0, 0)$ and $c(1, 1)$ respectively, the base case is hit in both cases. For the right subtree, r_{24} points to s_4. The cost of its right subtree is $c(4, 4)$, so that is another base case. The left subtree has cost $c(3, 4)$, so we look for r_{23} and find it is s_3. As with the other branches, the tree terminates here because the left and right subtrees are base cases. The final constructed tree is as follows:



Analysis

So far, we have not considered the computational complexity of any of the dynamic programming algorithms. We provide an analysis here as a sample and point out that the other analyses would be similar.

Theorem: Constructing an optimal binary search tree requires $O(n^3)$ time using the algorithm presented.

Proof: First, given we have computed the optimal roots r_{ij}, we can construct the tree in $O(n)$ time since there are only n calls to BuildTree (one for each root), each requiring constant time. The expensive part of the algorithm comes from MinRoot. The "arg min" part of the algorithm requires $O(j-i)$ time since we are scanning over $j-i$ possible roots. All other steps require constant time. This step falls in the middle of two nested loops. The outer loop is executed n times, and the inner loop is executed at most $n-1$ times. In combination, this leads to $O(n^3)$ performance. QED