

CS 337 Project 1: Minimum-Weight Binary Search Trees

September 6, 2006

1 Preliminaries

Let X denote a set of keys drawn from some totally ordered universe U , such as the set of all integers, or the set of all real numbers. Let n denote $|X|$ and assume that the elements of X are $x_1 < x_2 < \dots < x_n$. Suppose we would like to process queries of the following form: Given a key u in U , determine whether u belongs to X . A standard data structure for supporting such search queries is the familiar binary search tree (BST). For $n > 1$ there is more than one BST containing the set of keys X . For example, for $n = 2$, there are exactly two such BSTs: one with x_1 at the root and x_2 at the right child of the root, and one with x_2 at the root and x_1 at the left child of the root.

The n keys in X partition U into $2n + 1$ sets, as follows: the set of keys less than x_1 , which we denote U_0 ; the singleton set of keys $\{x_1\}$, which we denote U_1 ; the set of keys greater than x_1 and less than x_2 , which we denote U_2 ; the singleton set of keys $\{x_2\}$, which we denote U_3 ; \dots ; the set of keys greater than x_n , which we denote U_{2n} . Given a BST T containing X , we search for a key u in U by performing a sequence of 3-way comparisons, beginning at the root. (A 3-way comparison between two keys u and v determines whether $u = v$, $u < v$, or $u > v$.) The sequence of 3-way comparisons performed in such a search is completely determined by the index i of the unique set U_i to which u belongs. For any index i , let $w(T, i)$ denote the number of 3-way comparisons used to search T for a key in U_i . Further assume that for any index i , the frequency with which we search for a key in U_i is known and is equal to f_i . Then we define the weight of BST T , denoted $w(T)$, as follows:

$$w(T) = \sum_{0 \leq i \leq 2n} w(T, i) \cdot f_i$$

In this assignment, we explore several efficient methods for computing a minimum-weight BST.

It is worth remarking that the particular values of the x_i 's have no impact on the structure of the minimum-weight BST. Thus, we can represent an instance of the minimum-weight BST problem by an odd-length sequence of nonnegative frequency values $\langle f_0, \dots, f_{2n} \rangle$.

2 Dynamic Programming

There is some similarity between the minimum-weight BST problem and the optimal prefix code problem discussed in the lectures. Accordingly, one might hope to devise a greedy

algorithm, similar to Huffman’s algorithm, for solving the minimum-weight BST problem. However, no such greedy algorithm is known. In this section, we instead consider somewhat more complex algorithms based on a technique called *dynamic programming*.

2.1 An $O(n^3)$ Algorithm

The key observation underlying the dynamic programming approach to the minimum-weight BST problem is as follows. Suppose we are given an instance I equal to $\langle f_0, \dots, f_{2n} \rangle$ of the minimum-weight BST problem. For all integers i and j such that $0 \leq i \leq j \leq n$, let $I(i, j)$ denote the instance of the minimum-weight BST problem given by length- $(2(j - i) + 1)$ sequence of frequency values $\langle f_{2i}, f_{2i+1}, f_{2i+2}, \dots, f_{2j} \rangle$. We define the *size* of an instance $I(i, j)$ as $j - i$, since the solution to instance $I(i, j)$ is a BST containing $j - i$ nodes. Our plan will be to solve the given instance $I = I(0, n)$ by successively solving all of the instances $I(i, j)$, in nondecreasing order of size.

But how do we go about solving a general instance $I(i, j)$? If $i = j$, the instance is trivial to solve, so let us assume that $i < j$. The key observation underlying our approach is as follows. Suppose that instance $I(i, j)$ admits a minimum-weight BST T in which the left subtree T_0 of the root contains n_0 nodes and the right subtree T_1 of the root contains n_1 nodes. (Note that $j - i = n_0 + n_1 + 1$, since the size of instance $I(i, j)$ dictates that T contains $j - i$ nodes.) Then we claim that T_0 is a minimum-weight solution to instance $I(i, i + n_0)$ and T_1 is a minimum-weight solution to instance $I(j - n_1, j)$. We will argue the correctness of this claim in the lecture associated with this programming assignment, and also in the discussion sections. The basic idea is that if either T_0 or T_1 is not minimum-weight, then T could be improved by replacing T_0 and T_1 with minimum-weight solutions to $I(i, i + n_0)$ and $I(j - n_1, j)$, respectively; such an improvement contradicts our assumption that T is a minimum-weight solution to instance $I(i, j)$.

Programming Exercise 1 (35 points): Use the foregoing claim to develop an algorithm for the minimum-weight BST problem with running time $O(n^3)$. Implement your algorithm in Java. Your program should be contained in a single source file named `CubicBST.java`. The input to your algorithm is a sequence of minimum-weight BST instances, one per line. The line of input encoding a particular minimum-weight BST instance consists of a nonnegative integer n followed by $2n + 1$ nonnegative integers representing the sequence of frequencies $\langle f_0, \dots, f_{2n} \rangle$. The input is terminated by a line consisting of a single negative integer.

For each minimum-weight BST instance, your program should produce a single line of output, indicating the weight of a minimum-weight BST. Note: To simplify the coding task, we are not asking you to output a minimum-weight BST, just the weight of such a BST.

2.2 An $O(n^2)$ Algorithm

For all integers i and j such that $0 \leq i < j \leq n$, let $T(i, j)$ denote the set of all minimum-weight solutions to instance $I(i, j)$, and let $\ell(i, j)$ denote the minimum, over all T in $T(i, j)$, of the number of nodes in the left subtree of the root of T . We claim that the following lemma holds. (The proof of the lemma is somewhat challenging, and beyond the scope of this course.)

Lemma 1 *For all integers i and j such that $0 \leq i, j \leq n$, and $j - i \geq 2$, we have*

$$\ell(i, j - 1) \leq \ell(i, j) \leq \ell(i + 1, j)$$

Paper & Pencil Exercise 1 (5 points): Make use of Lemma 1 to justify the correctness of an improved version of the $O(n^3)$ algorithm developed in Section 2.1. The improved version should run in $O(n^2)$ time.

Programming Exercise 2 (10 points): Implement your $O(n^2)$ algorithm in Java, using a single source file named `QuadraticBST.java`. The input-output behavior of `QuadraticBST.java` should be the same as that of `CubicBST.java`.

3 A Greedy Algorithm for a Special Case

In this section we develop algorithms for the special case of the minimum-weight BST problem in which $f_i = 0$ for all odd indices i . In other words, we are now focusing on the special case in which all searches are unsuccessful. In this assignment we refer to this problem as the minimum-weight UBST problem, where the letter ‘U’ is intended to remind us that all searches are unsuccessful. The minimum-weight UBST problem turns out to be closely related to the optimal prefix code problem discussed in the lectures, and which we have seen how to solve using Huffman’s algorithm. Recall that the input to Huffman’s algorithm is a sequence of symbols with associated frequencies, and the output is a prefix code. Assume that the symbols to be encoded are drawn from some totally ordered set. Then we define such a prefix code to be *order-preserving* if the i th largest symbol is mapped to the i th largest codeword, where codewords are ordered lexicographically (i.e., in “dictionary order”). It is not hard to see that the minimum-weight UBST problem is equivalent to the problem of determining a minimum-weight order-preserving prefix code for a set of $n + 1$ symbols with associated nonnegative frequencies $\langle f_0, f_2, f_4, \dots, f_{2n} \rangle$.

Paper & Pencil Exercise 2 (5 points): Let s_0 and s_1 be distinct sequences of symbols, where the sequences are drawn from some totally ordered universe. Assume that s_0 lexicographically precedes s_1 . Let t_0 (resp., t_1) denote the bit string obtained by encoding s_0 (resp., s_1) using an order-preserving prefix code. Prove that t_0 lexicographically precedes t_1 .

3.1 Minimum-Weight Order-Preserving Prefix Codes

In this section we describe a Huffman-like greedy algorithm, due to Hu and Tucker, for computing a minimum-weight order-preserving prefix code. Recall that, in the case of Huffman’s algorithm, the input is a bag of n nonnegative frequency values. In the order-preserving version of the problem, the frequency values are given in the form of a sequence, where the order of the sequence is determined by a given total order on the symbols. For example, if the symbols are ‘A’ through ‘Z’, in alphabetic order, then the sequence begins with the frequency value for ‘A’, followed by the frequency value for ‘B’, et cetera.

The Hu-Tucker algorithm consists of two main phases. The first phase determines the length of the codeword to be assigned to each input symbol. This phase uses a greedy strategy that is similar to Huffman’s algorithm, though somewhat more complicated.

The second phase of the Hu-Tucker algorithm determines the specific codeword to be assigned to each symbol. It is not hard to convince yourself that the codewords are uniquely determined by the lengths computed in the first phase. Furthermore, it is relatively straightforward to compute the codewords from these lengths. In this assignment, you will not need

to implement the second phase of the Hu-Tucker algorithm, because you are only being asked to compute the weight of a minimum-weight order-preserving prefix code.

We now describe the first phase of the Hu-Tucker algorithm in greater detail. Like Huffman's algorithm, this phase proceeds by iteratively merging two of the frequencies into a single frequency (by adding them) until there is only one frequency left. Once all of the merging is complete, the length of the codeword associated with a particular symbol can be deduced by computing the number of merges associated with the frequency value of that symbol. In this assignment, you will not need to determine the codeword length for each symbol, since you are only being asked to compute the weight of a minimum-weight solution, a quantity which can be easily computed as a by-product of the merging process. In what follows, we describe the merging rule in greater detail.

A general iteration of the merging process starts with a sequence of $n \geq 2$ ordered pairs $(b_0, g_0), \dots, (b_{n-1}, g_{n-1})$, where the b_i 's are boolean values and the g_i 's are frequency values. In the first iteration, all of the b_i 's are initialized to `true`, and the g_i 's represent the input frequencies. We define an *index-pair* as an ordered pair of integers (i, j) such that $0 \leq i < j < n$. The *value* of such an index-pair is defined as the integer triple $(g_i + g_j, i, j)$. An index-pair (i, j) is defined to be *good* if there is no integer k such that $i < k < j$ and b_k is true; otherwise, it is *bad*. Let G denote the set of all good index-pairs. Note that G is guaranteed to be nonempty, since we have assumed that $n \geq 2$. Let (u, v) denote the index-pair in G with the lexicographically least associated value. Then the Hu-Tucker algorithm removes the pair (b_v, g_v) from the current sequence, and replaces the pair (b_u, g_u) with the pair $(\text{false}, g_u + g_v)$.

Programming Exercise 3 (35 points): Write a Java program based on the Hu-Tucker algorithm to compute the weight of a minimum-weight order-preserving prefix code. All of your source code should be provided in a single file named `HuTucker.java`. The input to your algorithm is a sequence of instances, one per line. The line of input encoding a particular instance consists of a positive integer n followed by n nonnegative integers representing the ordered sequence of symbol frequencies. The input is terminated by a line containing a single zero. For each instance, your program should produce a single line of output, indicating the weight of a minimum-weight order-preserving prefix code. For full credit, your algorithm should run in $O(n^2)$ time.

3.2 An $O(n \log n)$ Algorithm

A mergeable priority queue data structure supports the usual priority queue operations, and also provides efficient support for merging two priority queues. There exist mergeable priority queues with the following performance guarantees: (1) the usual priority queue operations `insert` and `delete-min` run in $O(\log n)$ time, where n denotes the number of items in the priority queue; (2) given two priority queues Q_0 and Q_1 containing m and n items, respectively, one can combine Q_0 and Q_1 into a single priority queue Q containing all $m + n$ items in $O(\log(m + n))$ time.

Paper & Pencil Exercise 3 (10 points): Describe how to use a mergeable priority queue data structure to perform the same computation as the program `HuTucker.java`, but with $O(n \log n)$ running time per instance.

4 Turning in the Project

This project has three programming exercises and three paper & pencil exercises. The programming exercises are due at 8pm on Wednesday, September 27, and should be turned in electronically as described in the “project protocol” document available in the projects section of the course website. The file `readme.txt` should contain a separate section for each of the three programs. It is highly recommended that you turn in all three programming exercises by the deadline. However, since you will be working on three separate programs, it is possible that you might complete some of them by the deadline, and still be working on some others. In that case we will only assess a lateness penalty on the programs that you turn in late. In the event that you opt to turn in one or more of the programs late, please avoid resubmitting your other programs, to minimize your lateness penalty.

The TA in charge of grading this project is Indrajit Roy. If you turn in all of your programs at once, you should turn them in using the command `turnin -submit indrajit Project1 CubicBST.java QuadraticBST.java HuTucker.java readme.txt`.

The paper & pencil exercises are also due on Wednesday, September 27. It is highly recommended that you turn in your solutions (as a paper document) at the lecture on Wednesday, September 27. However, your paper solutions will be accepted without a lateness penalty up to 8pm on Wednesday, September 27. If you decide to turn in your solutions outside the class period, please drop them off in Greg Plaxton’s office, TAY 3.132. (If no one is there when you come by, just slide your solutions under the door.) Unlike the programming portion of the assignment, please note that the 8pm deadline for turning in the paper & pencil exercises is a *hard deadline*. If by the deadline you haven’t been able to complete all of the paper & pencil exercises, just turn in whatever you have been able to complete.

5 Questions and Project Clarifications

This project description is somewhat terse. We will be discussing the project in greater detail during the lecture period on Monday, September 11. This project will also be the primary topic of discussion at the upcoming discuss sections and office hours. Please take advantage of these opportunities to get your questions answered.

You may also use the project newsgroup `utexas.class.cs337` to ask for clarifications on the project. The primary advantage of the newsgroup is that it allows you to pose a question that might be answered by another member of the class. For this project, the newsgroup will be loosely monitored by one of the TAs, Indrajit Roy. If you urgently require a timely response to a question that comes up outside of the regularly scheduled lectures, discussion sections, and office hours, then you should not simply post it to the newsgroup, because it might not get answered for a day or more. Instead, you should send an email to the course instructor or to one of the TAs. But you should try to avoid this, i.e., to the extent possible, you should try to get your questions answered at one of the regularly scheduled meetings.

Finally, instructor clarifications regarding the project will be posted on the project web page. You are responsible for any modifications found there. Sample inputs and outputs will also be posted on the project web page.