ELSEVIER Theoretical Computer Science 290 (2003) 1799–1814

Theoretical Computer Science

www.elsevier.com/locate/tcs

Optimal binary search trees with costs depending on the access paths☆

Jayme L. Szwarcfiter[a,*,1], Gonzalo Navarro[b,2], Ricardo Baeza-Yates[b,2], Joísa de S. Oliveira[c], Walter Cunto[d], Nívio Ziviani[e,3]

[a] *Universidade Federal do Rio de Janeiro, Instituto de Matemática, NCE and COPPE, Caixa Postal 2324, 20001-970 Rio de Janeiro, RJ, Brazil*

[b] *Departamento de Ciencias de la Computación, Universidad de Chile, Blanco Encalada 2120, Santiago, Chile*

[c] *Universidade Federal do Rio de Janeiro, COPPE, Caixa Postal 68511, 21945-970 Rio de Janeiro, RJ, Brazil*

[d] *Depto. de Computación, Universidad Simón Bolívar, Apartado 89000, Caracas, Venezuela*

[e] *Universidade Federal de Minas Gerais, Departamento de Ciência da Computação, 31270-010 Belo Horizonte, MG, Brazil*

Received 9 November 2000; received in revised form 5 December 2001; accepted 5 February 2002
Communicated by G. Ausiello

Abstract

We describe algorithms for constructing optimal binary search trees, in which the access cost of a key depends on the k preceding keys which were reached in the path to it. This problem has applications to searching on secondary memory and robotics. Two kinds of optimal trees are considered, namely optimal worst case trees and weighted average case trees. The time and space complexities of both algorithms are $O(n^{k+2})$ and $O(n^{k+1})$, respectively. The algorithms are based on a convenient decomposition and characterizations of sequences of keys which are paths

☆This paper has been partially supported by project CYTED VII.19 RIBIDI.

*Corresponding author. Tel.: +55-21-598-3155; fax: +55-21598-3156.

E-mail addresses: jayme@nce.ufrj.br (J.L. Szwarcfiter), gnavarro@dcc.uchile.cl (G. Navarro), rbaeza@dcc.uchile.cl (R. Baeza-Yates), joisa@cos.ufrj.br (J.de.S. Oliveira), walter_cunto@mckinsey.com (W. Cunto), nivio@dcc.ufmg.br (N. Ziviani).

[1]Partially supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico, CNPq, and Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro, FAPERJ, Brazil.

[2]Partially supported by Fondecyt Grant 99-0627.

[3]Partially supported by SIAM project grant MCT/FINEP/CNPq/PRONEX 76.97.1016.00 and CNPq grant 520.916/94-8.

PII: S0304-3975(02)00084-1

of special kinds in binary search trees. Finally, using generating functions, we present an exact analysis of the number of steps performed by the algorithms.

Keywords: Algorithms; Optimal binary search trees; Generating functions

1. Introduction

Binary search trees form one of the topics most commonly studied in computer science, probably due to their wide range of applications. Their importance can be assessed by reading the classical books by Knuth [4,5]. The study of optimal weighted binary search trees dates back to the 1950s. A tutorial on this subject has been written by Nagaraj [10], more recently.

Each node of a binary search tree can be assigned an access cost or a weight, where the latter can represent the access probability of the node. With access costs one may be interested in optimizing the worst case or the average case cost, where uniform access probability is assumed. In the second case one assumes that the access cost is uniform, and it is possible to assign probabilities only to internal tree nodes (successful searches), only to external tree nodes (unsuccessful searches) or to both. Finally, we can consider a more general model, where both access costs and weights are included.

An algorithm for constructing an optimal binary search tree has been first described by Gilbert and Moore [1], for the case in which to each key is assigned a weight. The complexity of this algorithm is $O(n^3)$. Knuth [3] considered the model of access probabilities including successful and unsuccessful searches. He proved an elegant monotonicity principle, which decreased the complexity by a factor of $O(n)$. When only unsuccessful searches are relevant a different algorithm can be applied, as described by Hu and Tucker [2]. The complexity of the latter algorithm was $O(n^2)$, but it has been shown to admit an implementation running in $O(n \log n)$ time [5]. Finally, we also mention that the problem of approximating optimal weighted binary search trees has been considered by several authors. See [8,9], for instance.

In this paper, we consider the problems of finding optimal binary search trees in which the access cost to a key x_q depends on the k preceding keys which were reached in the path to x_q. We permit arbitrary access probabilities (independent on the preceding keys) as well. The classical optimal binary search tree construction by Gilbert and Moore [1] and Knuth [3] corresponds thus to the fundamental case $k = 0$. In this work we are concerned with the values $k \geqslant 1$. Two kinds of optimal trees are considered, namely optimal worst case trees and weighted average case trees. The inputs of these problems are a number n of keys, the value k, $1 \leqslant k < n$, and a cost associated to each possible sequence formed by at most $k+1$ keys, all of them distinct. For the weighted average case minimization problem, each key is additionally given a weight. Usually, such a weight would reflect the frequency of accessing the key. Observe that the input size grows exponentially with k, as it is $O(n^{k+1})$.

We describe algorithms for solving the problems of finding the optimal worst case trees and weighted average case trees. The complexity is $O(n^{k+2})$ for both cases. The extra space needed is $O(n^{k+1})$. Time and space complexities are polynomial in the size of the input.

The optimal binary search tree for $k=0$ and with uniform key access costs, as considered in [1,3], is a model for situations in which the keys are in the main memory. Greater values of k and arbitrary access costs could model the cases in which other kind of memories are involved. For example, when all keys are stored in a disk, the access cost to a given key depends on the position on the disk of the key previously accessed. Therefore, finding an optimal tree when all keys are stored in a disk would correspond to the case $k=1$. In this situation, the input size is $O(n^2)$ and the complexity of the proposed algorithm is $O(n^3)$. For example, this is the case for optimal searching strategies on some text indices stored in secondary memory [11]. Similarly, in some motion planning problems, the cost of the next move depends on the previous position. The generic case can also be used where the cost of moving the robot depends on resources used in the last k locations. Besides practical motivations, we believe that some of the concepts presented in this paper might be of interest in the general study of search trees.

The following are some basic definitions.

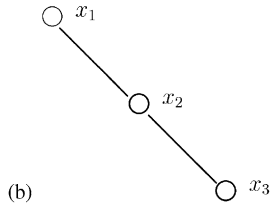
A *binary tree* is a rooted tree T in which every node z, other than the root, is labeled *left child* or *right child*, in such a way that any two siblings have different labels. When z has no siblings it is called an *only child*. A *path* of T is a sequence of nodes $z_1,\ldots,z_t$, such that z_q is the parent of z_{q+1}. In this case, z_1 is an *ancestor* of z_t, while z_t is a *descendant* of z_1. When $z_1 \neq z_t$ they are called *proper ancestor* and *proper descendant*, respectively. A t-path is a path formed by t nodes. The notation $N(T)$ represents the set of nodes of T. For $z \in N(T)$, the binary tree defined in T by all descendants of z is called the *subtree* of T *rooted* at z, and denoted by $T(z)$. The *left subtree* of z is the binary tree formed in T by the left child of z and all of its descendants. Similarly, define the *right subtree* of z. The left and right subtrees of z are represented by $T_L(z)$ and $T_R(z)$, respectively. A binary tree defined in T by a subset of $N(T)$ is called a *partial subtree* of T. A *root path* is a path starting at the root of T, while a *root–leaf path* starts at the root and ends at some leaf of T.

Let $\{x_1,\ldots,x_n\}$ be a set of elements called *keys*, $x_q < x_{q+1}$. A *binary search tree* for $\{x_1,\ldots,x_n\}$ is a binary tree T in which either $N(T)$ is empty, or the left and right subtrees of the root are binary search trees, where all keys in the left subtree are smaller than that of the root, while the keys in the right subtree are greater. A *legal path* is a sequence of keys which is a path in some binary search tree.

The described minimization problems are solved by dynamic programming equations. The corresponding decompositions employ the concepts of legal path and (i,j)-legal paths. The latter means those legal paths leading to a subtree formed by consecutive keys. We then describe characterizations for both legal and (i,j)-legal paths. The algorithms are obtained by combining the decompositions and the characterizations. The decompositions are presented in Section 2 and the characterizations are deferred to Section 3. Section 4 presents an analysis of some parameters of the tree, including the time and space complexity of the algorithms. The analysis is based on generating

(a)

legal paths	x_1	x_2	x_3	x_1x_2	x_1x_3	x_2x_1	x_2x_3	x_3x_1	x_3x_2
key costs	0	0	0	0	2	3	2	3	1



(b)

Fig. 1. Example of an optimal tree with non-optimal subtrees.

functions and enumerates (i,j)-legal paths. Finally, Section 5 presents the conclusions and some additional remarks.

2. The decompositions

Let $k \geqslant 1$ be a given integer value and $\{x_1,\ldots,x_n\}$ a set of keys, $x_q < x_{q+1}$. For each x_q and legal path $y_1,\ldots,y_t$, where $1 \leqslant t \leqslant k+1$ and $x_q = y_t$, it is given a real non-negative *key cost* $c(y_1,\ldots,y_t)$ of y_t *relative* to $y_1,\ldots y_t$. It corresponds to the cost of reaching y_t through the path $y_1,\ldots,y_t$. In addition, each key x_q is given a non-negative real *weight* $w(x_q)$. For a legal path $y_1,\ldots,y_m$, define its *path cost* as

$$C(y_1,\ldots,y_m) = \sum_{1 \leqslant q \leqslant m} c(y_{\max\{1,q-k\}},\ldots,y_q). \tag{1}$$

Let T be a binary search tree for $\{x_1,\ldots,x_n\}$. Denote by x_q^* the root path to key x_q. The values $\max_{1\leqslant q\leqslant n}\{C(x_q^*)\}$ and $\sum_{1\leqslant q\leqslant n} w(x_q)\cdot C(x_q^*)$ are called *worst case tree cost* and *weighted average case tree cost*, respectively. When $N(T)=\emptyset$, the costs of T are defined as zero. The question consists of finding the tree T which minimizes one of these two above costs, as desired. A minimizing tree is called *optimal*.

Observe that subtrees of an optimal tree are not necessarily optimal, for any $k>0$. Consider the example having $k=1$, $n=3$, with key costs as given by Fig. 1(a) and having all weights equal to 1.

The tree of Fig. 1(b) is both worst and average case optimal, but $T(x_2)$ is not optimal in any case. Consequently, the decomposition employed in the dynamic programming solution of the optimal binary search tree problem for $k=0$ does not apply to the present case. However, special kinds of partial subtrees are optimal, making it possible to solve our minimization problems by conveniently decomposing them into smaller subproblems, leading to techniques similar as [1,3]. At this point we need additional notation.

First, introduce k additional keys $\{x_{n+1},\ldots,x_{n+k}\}$, called *dummy keys*, also satisfying $x_q < x_{q+1}$, $n \leqslant q < n+k$. Each of these keys has weight 0. The key costs relative to paths containing dummy keys are defined as follows. Let $y_1,\ldots,y_t$ be a legal path

having at least one dummy key, $1 \leqslant t \leqslant k+1$. Then

$$c(y_1,\ldots,y_t) = \begin{cases} 0, & \text{when } y_1,\ldots,y_t \text{ are all dummy keys} \quad (2)\\ c(y_q,\ldots,y_t), & \text{when } \exists q > 1 \text{ such that } y_1,\ldots,y_{q-1} \text{ are} \\ & \quad \text{dummy keys, but } y_q,\ldots y_t \text{ are not} \quad (3)\\ \infty, & \text{otherwise.} \quad (4) \end{cases}$$

Denote $X = \{x_1,\ldots,x_{n+k}\}$, $X_i^- = \{x_1,\ldots,x_i\}$, $X_i^+ = \{x_{i+1},\ldots,x_{n+k}\}$, $X_{ij} = \{x_{i+1},\ldots,x_j\}$ and $W_{ij} = \sum_{i<q\leqslant j} w(x_q)$.

Let i,j be a pair of integers, $0 \leqslant i \leqslant j \leqslant n$. A path $y_1,\ldots,y_k$ is (i,j) *-legal* when there exists a binary search tree T having node set X containing the path $y_1,\ldots,y_k$ and such that either $i=j$ and y_k is a leaf of T, or y_k has a child $x_\ell \in X_{ij}$ satisfying $N(T(x_\ell)) = X_{ij}$. In other words, an (i,j)-legal path is one leading to a subtree containing exactly the keys of X_{ij}, in a tree formed by all keys of X.

Let $y_1,\ldots,y_k$ be an (i,j)-legal path. Denote by $T_{ij}(y_1,\ldots,y_k)$ an optimal subtree formed by the nodes of X_{ij}, where $y_1,\ldots,y_k$ is the path leading to its root. Represent by $C_{ij}(y_1,\ldots,y_k)$ the (optimal) cost of $T_{ij}(y_1,\ldots,y_k)$. That is, $C_{ij}(y_1,\ldots,y_k)$ can be interpreted as the optimal cost to search the subtree X_{ij}, given that $y_1,\ldots,y_k$ is the path leading to it. Note that $T_{ij}(y_1,\ldots,y_k)$ does not contain the nodes of $y_1,\ldots,y_k$, however the cost of it depends on this path. In terms of this notation, a solution to the stated minimization problems is the subtree of $T_{0n}(x_{n+k},x_{n+k-1},\ldots,x_{n+1})$, having as root the child of x_{n+1}. Observe that the path leading to the latter tree is formed solely by dummy keys.

For determining the value of the optimal cost $C_{ij}(y_1,\ldots,y_k)$, we decompose the corresponding problem into the subproblems of finding the optimal costs $C_{i,\ell-1}(y_2,\ldots,y_k, x_\ell)$ and $C_{\ell j}(y_2,\ldots,y_k,x_\ell)$, for each $x_\ell \in X_{ij}$. The key x_ℓ is the child of y_k in the trees. See Fig. 2.

The following dynamic programming equations apply the described decompositions and compute the optimal costs values.

Worst-case minimization

$$C_{ij}(y_1,\ldots,y_k) = \begin{cases} 0, \text{ when } i=j. \text{ Otherwise,} \quad (5)\\ \min\limits_{i<\ell\leqslant j} \{\max\{C_{i,\ell-1}(y_2,\ldots,y_k,x_\ell), \\ \qquad C_{\ell j}(y_2,\ldots,y_k,x_\ell)\} + c(y_1,\ldots,y_k,x_\ell)\} \quad (6) \end{cases}$$

for all $0 \leqslant i \leqslant j \leqslant n$ and (i,j)-legal paths $y_1,\ldots,y_k$, $k \geqslant 1$.

Weighted average-case minimization

$$C_{ij}(y_1,\ldots,y_k) = \begin{cases} 0, \text{when } i=j. \text{ Otherwise,} \quad (7)\\ \min\limits_{i<\ell\leqslant j} \{C_{i,\ell-1}(y_2,\ldots,y_k,x_\ell) + C_{\ell j}(y_2,\ldots,y_k,x_\ell) \\ \qquad + W_{ij}\cdot c(y_1,\ldots,y_k,x_\ell)\} \quad (8) \end{cases}$$

for all $0 \leqslant i \leqslant j \leqslant n$ and (i,j)-legal paths $y_1,\ldots,y_k$, $k \geqslant 1$.

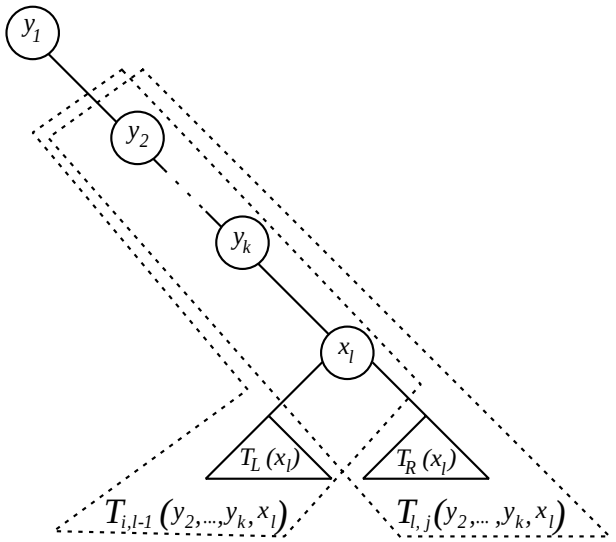


Fig. 2. The decomposition of $T_{ij}(y_1,\ldots,y_k)$.

In order to verify the correctness of the above equations, note that if $y_1,\ldots,y_k$ is an (i,j)-legal path and $i<\ell\leqslant j$ then $y_2,\ldots,y_k,x_\ell$ is both $(i,\ell-1)$-legal and (ℓ,j)-legal. Using this fact, the dynamic programming equations can be obtained by standard induction.

The algorithms for finding optimal worst case and weighted average case binary search trees can now be described.

The input consists of an integer $k>0$, a set $\{x_1,\ldots,x_n\}$ of keys, $x_q<x_{q+1}$, and a key cost $c(y_1,\ldots,y_t)$ for each legal t-path, $1\leqslant t\leqslant k+1$. Alternatively, the input can consist of a function which enables to compute the key costs $c(y_1,\ldots,y_t)$, whenever needed. In the latter case we assume that this computation can be done in constant time. In addition, in the weighted average case problem each key x_q is also given a non-negative weight $w(x_q)$.

The algorithms start by defining the dummy keys $\{x_{n+1},\ldots,x_{n+k}\}$. Using (2)–(4), compute the key costs $c(y_1,\ldots,y_t)$, for each legal t-path $y_1,\ldots,y_t$ with at least one dummy key, $1\leqslant t\leqslant k+1$. Define $w(x_q)=0$ for each $n+1\leqslant q\leqslant n+k$. For each (i,j)-legal t-path $y_1,\ldots,y_t$ and $0\leqslant i\leqslant j\leqslant n$, compute $C_{ij}(y_1,\ldots,y_t)$ by (5)–(6) and (7)–(8), respectively for the worst case and weighted average case problems. All required legal and (i,j)-legal paths are generated using Theorems 1 and 2, respectively. The final solution is $C_{0n}(x_{n+k},\ldots,x_{n+1})$. In the next section we characterize (i,j)-legal paths, using particular ordering schemes.

It is simple to modify the algorithms to avoid computations with dummy keys. An idea is to impose that whenever x_p and x_q are dummy keys and x_p is a proper ancestor of x_q then $p>q$.

One could wonder whether is it possible to improve this algorithm. The *monotonicity principle* by Knuth [3] made it possible to decrease the number of iterations from $O(n^3)$

to $O(n^2)$, for constructing an optimal binary search tree. Unfortunately, the principle does not hold for $k>0$, as shown by the following example. Let $\{x_1,\ldots,x_{k+2}\}$ be the given set of keys, all with uniform weights. The costs are defined as follows:

$$c(x_{k+1},\ldots,x_1)=c(x_{k+1},\ldots,x_2)=\cdots=c(x_{k+1})=0,$$

$$c(x_1,\ldots,x_k,x_{k+2})=c(x_1,\ldots,x_k)=\cdots=c(x_1)=0,$$

$$c(x_2,\ldots,x_k,x_{k+2},x_{k+1})=0,$$

while any other key cost is equal to 1. The solution of both minimization problems for the keys $\{x_1,\ldots,x_{k+1}\}$ is the tree formed by the single path $x_{k+1},\ldots,x_1$. When adding the key x_{k+2}, the optimal tree for $\{x_1,\ldots,x_{k+2}\}$ is the path $x_1,\ldots,x_m,x_{m+2},x_{m+1}$, meaning that the principle does not apply for $k>0$. In fact, it does not hold also for $k=0$ under non uniform key costs.

Finally, it would be worth mentioning that the proposed model can also handle unsuccessful searches. Basically, to the existing $n+k$ keys of the tree, we add $n+k+1$ new nodes. These are called *gaps* and correspond to the external nodes, i.e., unsuccessful searches. To each gap it is given an arbitrary weight, as for keys. The key costs of a key or gap y_t are redefined, so as to satisfy the following conditions. If $y_1,\ldots,y_t$ are all keys then the value $c(y_1,\ldots,y_t)$ is exactly as explained in this section. That is, either taken from the input or computed by (2–4). Otherwise (i) $c(y_1,\ldots,y_t)=\infty$, whenever any among $y_1,\ldots,y_{t-1}$ is a gap, or (ii) $c(y_1,\ldots,y_t)=0$, in case that y_t is a gap and all $y_1,\ldots,y_{t-1}$ are keys. Then we apply the algorithms just described.

3. Characterizing legal paths

In this section we describe characterizations for legal and (i,j)-legal paths. That is, for sequences of keys which are paths in some binary tree, and which lead to subtrees formed by consecutive keys, respectively. The following definition is useful.

Let $Y\subset X$. An ordering $y_1,\ldots,y_m$ of the keys of Y is called *min–max* when each y_q is either minimal or maximal in $\{y_q,\ldots,y_m\}$. In this case, label each y_q, $1\leqslant q\leqslant m$, as *min* or *max*, respectively.

The following characterizes legal paths.

Theorem 1. *A path is legal if and only if it is a min–max ordering.*

Proof. Let $y_1,\ldots,y_m$ be a legal path. Then there exists a binary search tree T, such that $y_1,\ldots,y_m$ is a path of T. If it is not a min–max ordering there exists a key y_i which is neither the minimal nor the maximal of $\{y_i,y_{i+1},\ldots,y_m\}$, $i\leqslant m-2$. If y_{i+1} is a left child in T then $y_i>y_{i+1},\ldots,y_m$, implying that y_i is a max key. Similarly, y_{i+1} cannot be a right child, because it would imply that y_i is a min key. The contradiction implies that $y_1,\ldots,y_m$ is a min–max ordering.

Conversely, let $y_1,\ldots,y_m$ be a min–max ordering. We construct a binary tree T such that $y_1,\ldots,y_m$ is a path of it. For each i, $1<i\leqslant m$, let y_i be either the left or right child

of y_{i-1} in T, according to whether y_i is a min or max key, respectively. It follows that T is a binary search tree. Consequently, $y_1,\ldots,y_m$ is a legal path. □

The following ordering is also of interest.

For $Y \subset X$ and $0 \leqslant i < j \leqslant n+k$, an ordering Y' of Y is called (i,j)-*inc-dec* when:

- $Y \subset X_i^- \cup X_j^+$;
- the keys of $Y \cap X_i^-$ are in increasing ordering in Y', while those of $Y \cap X_j^+$ are in decreasing ordering;
- $Y \cap X_i^- \neq \emptyset \Rightarrow x_i \in Y$; and
  $Y \cap X_j^+ \neq \emptyset \Rightarrow x_{j+1} \in Y$.

Lemma 1. *A (i,j)-inc-dec ordering is necessarily a min–max ordering.*

Proof. Label the keys of $Y \cap X_i^-$ as min, and as max those of $Y \cap X_j^+$. □

The next theorem characterizes (i,j)-legal paths.

Theorem 2. *For $i=j$ a path is (i,j)-legal if and only if it is a min–max ordering. For $i<j$, a path is (i,j)-legal if and only if it is a (i,j)-inc-dec ordering.*

Proof. When $i=j$ the results follows from Theorem 1. Consequently, we assume throughout the proof that $i<j$.

($\Rightarrow$): The idea of the proof of necessity is simple. Considering an arbitrary (i,j)-legal path, we show that it satisfies the definition of an (i,j)-inc-dec ordering.

By hypothesis, $y_1,\ldots,y_m$ is a (i,j)-legal path. Then there is a binary search tree T, having X as its node set, where $y_1,\ldots,y_m$ is a path of it, y_m the father of some $x_\ell \in X_{ij}$ and the subtree $T(x_\ell)$ contains exactly the keys of X_{ij}. Let $Y=\{y_1,\ldots,y_m\}$. We prove that $y_1,\ldots,y_m$ satisfies the three above conditions for an (i,j)-inc-dec ordering. First, clearly $Y \subset X_i^- \cup X_j^+$. Second, suppose there exists a key $y_q \in X_i^- \cap Y$ such that $y_q > y_{q+1}$ for some $1 \leqslant q < m$. Since T is a binary search tree, it follows that y_{q+1} is a key of the left subtree of y_q. Since y_q is an ancestor of y_m, we know that x_ℓ also belongs to this subtree, contradicting $x_\ell > y_q$, implied by $y_q \in X_i^-$. Hence no such q can exist. Consequently, the keys of $X_i^- \cap Y$ are in increasing ordering in $y_1,\ldots,y_m$. Similarly, we prove that those of $X_j^+ \cap Y$ form a decreasing ordering. Third, suppose that $X_i^- \cap Y \neq \emptyset$ and $x_i \notin Y$. Denote by y_t the maximal key of $X_i^- \cap Y$. Clearly, $y_t < x_i$. We try to locate key x_i in T. Suppose that x_i is a descendant of y_t. Then x_i belongs to the right subtree R of y_t. Consequently, $T(x_\ell)$ is also in R. If $t=m$ then $x_i \in T(x_\ell)$, a contradiction. When $t<m$ we know that $y_t,\ldots,y_m$ is a path of R. Because T is a binary search tree and the maximality of y_t in X_i^- it follows that $y_{t+1},\ldots,y_m \in X_j^+$. Consequently, because the keys of $X_j^+ \cap Y$ are in decreasing ordering in $y_1,\ldots,y_m$, we conclude that y_{t+1} is a right child, but $y_{t+2},\ldots,y_m,x_\ell$ are all left children. Because $x_i < y_{t+1},\ldots,y_m,x_\ell$ it follows that x_i must belong to $T(x_\ell)$. The latter contradicts again the fact that $T(x_\ell)$ contains exactly X_{ij}. Hence x_i is not a descendant of y_t. Neither can x_i be an ancestor of y_t. Because in this case, y_t belongs to the left subtree L of x_i, implying that $x_\ell > x_i$ belongs to L, a contradiction. The remaining possibility is

that x_i is neither a descendant nor an ancestor of y_t. In this case, let z be the nearest common ancestor of x_i and y_t. Denote by L and R the left and right subtrees of z, respectively. If x_i is in L then y_t must be in R, contradicting $y_t < x_i$. The other case is x_i in R and y_t in L, making it impossible the assumption $x_i < x_{i+1}$. Therefore the alternative that x_i is neither a descendant nor an ancestor of y_t can also not occur. Consequently, $X_i^- \cap Y \neq \emptyset$ implies $x_i \in Y$. The proof that $X_j^+ \cap Y \neq \emptyset$ implies $x_{j+1} \in Y$ is similar. Consequently, $y_1, \ldots, y_m$ is an (i,j)-inc-dec ordering, settling the proof of necessity.

($\Leftarrow$): The idea of the proof of the converse is as follows. Consider a set of keys X and a subset of it forming an (i,j)-inc-dec ordering $\{y_1, \ldots, y_m\}$. We construct a binary search tree T for X such that $\{y_1, \ldots, y_m\}$ is an (i,j)-legal path of T.

Suppose that $y_1, \ldots, y_m$ is an (i,j)-inc-dec ordering, $0 \leqslant i < j \leqslant n + k$. Construct a binary tree T' as follows. The sequence $y_1, \ldots, y_m$ is a path of T', such that y_p is a left or right child of y_{p-1}, according to whether $y_p < y_{p+1}$ or $y_p > y_{p+1}$, respectively. T' also contains a subtree $T'(x_\ell)$, having an arbitrary root $x_\ell \in X_{ij}$, and satisfying the following property: $T'(x_\ell)$ is a binary search tree containing exactly the keys of X_{ij}. Finally, make x_ℓ the left or right child of y_m, according to whether $y_m \in X_j^+$ or $y_m \in X_i^-$, respectively. The construction of T' is completed. Let $Y = \{y_1, \ldots, y_m\}$. Since $y_1, \ldots, y_m$ is an (i,j)-inc-dec ordering, it follows that $Y \cap X_{ij} = \emptyset$. Hence the path $y_1, \ldots, y_m$ and $T(x_\ell)$ are disjoint. The latter completes the argument to show that T' is a binary tree. Moreover, we will conclude that it is in fact a binary search tree. With this purpose, let z_1, z_2 be keys of T', z_1 belonging to the left subtree L of z_2. Consider the possibilities:

Case 1: $z_1, z_2 \in Y$.

Since $y_1, \ldots, y_m$ is an (i,j)-inc-dec ordering, by Lemma 1 it is a min–max ordering. By Theorem 1 it must be a legal path. Hence z_1 being in L implies $z_1 < z_2$.

Case 2: $z_1 \in X_{ij}$ and $z_2 \in Y$.

Suppose $y_m = z_2$. Then x_ℓ must be the left child of y_m. By the construction of T', we conclude that $z_2 \in X_j^+$. Hence $z_1 < z_2$. Suppose now $z_2 \neq y_m$. By Case 1, we conclude that $y_m < z_2$. Suppose $y_m \in X_j^+$. Then $z_1 < y_m$, implying $z_1 < z_2$. Alternatively, consider $y_m \in X_i^-$. In this case, if $z_2 \in X_i^-$ then z_2, y_m must appear in increasing ordering, because $y_1, \ldots, y_m$ is an (i,j)-inc-dec ordering. Hence $z_2 < y_m$, a contradiction. Consequently, $z_2 \in X_j^+$. That is, $z_1 < z_2$.

Case 3: $z_1 \in Y$ and $z_2 \in X_{ij}$.

This case cannot occur, because it implies that z_2 is a descendant of z_1. This contradicts z_1 belonging to the left subtree of z_2.

Case 4: $z_1, z_2 \in X_{ij}$.

Since $T(x_\ell)$ is a binary search tree, z_1 being in L implies $z_1 < z_2$.

From the above cases, we can conclude that z_1 belonging to $T_L(z_2)$ implies that $z_1 < z_2$, for any $z_1, z_2 \in Y \cup X_{ij}$. Similarly, it can be proved that z_1 belonging to $T_R(z_2)$ implies $z_1 > z_2$. Consequently, T' is a binary search tree containing the keys $N(T') = Y \cup X_{ij}$. Let $X' = X \setminus N(T')$. We now include in T' each key of X', as follows. If $Y \cap X_i^- = \emptyset$ and $i > 0$ then include $x_i \in X'$ in T' so as y_1 becomes the right child of

x_i. Similarly, if $Y \cap X_j^+ = \emptyset$ and $j < n + k$ then $x_{j+1} \in X'$ is included in T' in such a way that y_1 is the left child of x_{j+1}. Note that the above two conditions cannot occur simultaneously. Next, for each key of X' not yet included in the tree, include it according to the rules of binary search tree insertion. Let T be the final tree so obtained. Since T' is a binary search tree, T is so. Also, T' is a partial subtree of T. Clearly $N(T) = X$ and $y_1, \ldots, y_k$ is a path of T'. Consequently, in order to show that $y_1, \ldots, y_m$ is (i,j)-legal, it remains only to prove that $T'(x_\ell) = X_{ij}$. Equivalently, that $T(x_\ell) = T'(x_\ell)$. Suppose the contrary. Then $T(x_\ell)$ necessarily contains some key $z \in X'$. Suppose $z \in X_i^-$. The following alternatives exist.

Case 1: $Y \cap X_i^- \neq \emptyset$.

By the definition of (i,j)-inc-dec ordering, it follows that $x_i \in Y$. That is, x_i is a proper ancestor of x_ℓ in T. Hence, $z \neq x_i$. Since x_i is the maximal key of X_i^-, it follows $z < x_i$. Then the binary search tree insertion procedure would not include z in the right subtree of x_i. On the other hand, x_ℓ belongs to the right subtree of x_i, as $x_i < x_\ell$. Hence $z \notin N(T(x_\ell))$.

Case 2: $Y \cap X_i^- = \emptyset$.

If $i = 0$ then $X_i^- = \emptyset$, contradicting $z \in X_i^-$. When $i > 0$, y_1 is the right child of x_i, by the construction of T. Hence $z < x_i$, implying that the binary search tree insertion again could not include z in $T_R(x_i)$. However $x_\ell \in N(T_R(x_i))$. That is, $z \notin N(T(x_\ell))$.

Consequently, $z \in X_i^-$ implies that z is not in $T(x_\ell)$. Similarly, we prove that $z \in X_j^+$ also implies that z cannot be in $T(x_\ell)$. Therefore $T(x_\ell)$ is formed exactly by the keys of X_{ij}. Hence $y_1, \ldots, y_m$ is an (i,j)-legal path, completing the proof of Theorem 2. □

4. Analytical results

In this section we compute some measures related to the problem. We start by computing a couple of general measures and later use them to deduce some parameters important for the problem: number of steps performed by the algorithm, space complexity, size of the input and number of (i,j)-legal paths. We first compute the above measures exactly and later give an easier to grasp approximation. The final result is that we pay $O(n^{k+2})$ time and $O(n^{k+1})$ space.

We make heavy use of generating functions to obtain our results. Generating functions represent a sequence $\{a_n\}_{n \geqslant 0}$ as a complex-valued function $a(z) = \sum_{n \geqslant 0} a_n z^n$ (this operation is also called the z-transform). Recurrences, that is, equations that define $\{a_n\}$ by relating the values of a_n for different n, are transformed on both sides so as to obtain an equation that defines $a(z)$. After solving for $a(z)$, the transformation is reversed and the value of a_n is obtained. It is possible to transform multi-index sequences into multivariate generating functions. For more details refer to the book by Sedgewick and Flajolet [12].

Rethink legal paths this way: instead of considering a sequence of y_q min–max values, consider that the interval to work on, initially $[1,n]$, is reduced k times, by either incrementing its left limit (min value) or decrementing its right limit (max value). Hence, we have a sequence of increments and a sequence of decrements, where

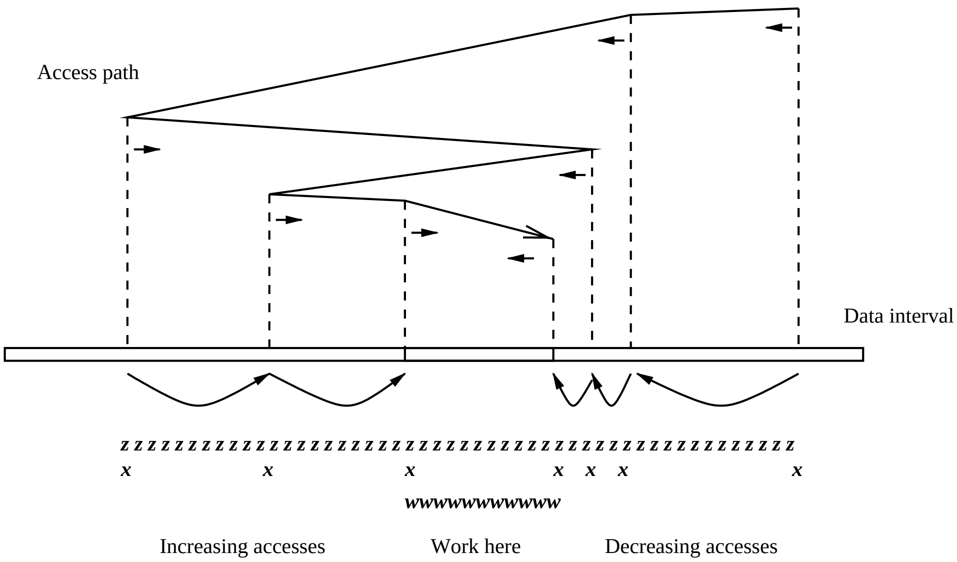


Fig. 3. Interpreting legal paths. Variables z, x and w correspond to the quantities to be counted.

the sum of the steps is k. We can identify the legal path with the pair of sequences (accounting also for the form in which they are mixed). If we are interested in the amount of work to do, we consider that after the k steps are done, we work in time proportional to the size of the interval left. See Fig. 3.

The generating function to be used has three variables z, x, w. Let the variable z count the total size of the array (n), x count the total number of accesses (k) and w the total amount of work. Our generating function is thus

$$F(z,x,w) = \sum_{n,k,r \geqslant 0} F_{n,k,r} z^n x^k w^r$$

such that in an array of n elements there are $F_{n,k,r}$ different legal paths of k steps which lead to an interval of size r (which costs $O(r)$).

To keep count of the size of the array (in z) and the number of steps (in x) at the same time, we consider the number of elements "skipped" in the consecutive increments (see Fig. 3). A single increasing step is represented by the function

$$I(z,x) = \frac{xz}{1-z} = xz + xz^2 + xz^3 + \cdots$$

that is, one access is performed (x) after skipping over one or more elements of the array (z's). There is at least one element, which is the array element compared. A sequence of zero or more increasing accesses is represented by

$$I^*(z,x) = \frac{1}{1-I(z,x)} = 1 + I(z,x) + I(z,x)^2 + I(z,x)^3 + \cdots$$

and the same formulas hold for $D(z,x)=I(z,x)$ and $D^*(z,x)=I^*(z,x)$. A sequence of intermingled increasing and decreasing accesses corresponds to

$$ID^*(z,x)=\frac{1}{1-(I(z,x)+D(z,x))}=\frac{1}{1-2xz/(1-z)}$$

and the final sequence of elements of the set where we have to work is represented by

$$\frac{1}{1-wz}=1+wz+w^2z^2+w^3z^3+\cdots$$

(where we count one unit of work in w and one element of the array in z). On the other hand, a sequence of elements where we do not have to work is simply $1/(1-z)=1+z+z^2+\cdots$.

We are still missing some border conditions. If we are interested in the total number of access paths that start with the complete array and end up at a given (i,j) interval (i.e. inc-dec orderings), then we have all the elements to express the final formula, which is

$$F_0(z,x,w)=\frac{1}{1-2xz/(1-z)}\frac{1}{1-wz}$$

which counts a number of increasing or decreasing steps plus a final central segment. Since we sum z's along all this process, we have in z the length of the resulting array. We add an x per step so we have in x the number of steps. Finally, we have in w the size of the final segment. At the end, we select those processes which turn out to have n elements (z^n), k steps (x^k), and lead to an array of size $|j-i|+1$ ($w^{|j-i|+1}$).

However, this is not the correct formula if we are interested in the time or space complexity. The reason is that we have to compute the above measures not only if we start with the original array, but also for any possible original subinterval.

There are two important cases here. First, if an interval has increasing and decreasing components, then we do not have to perform a different computation for all the possible original subintervals. For instance, suppose that $n=100$ and $k=2$. The legal path given by $[25,75]$ for example, that yields the subinterval $[26,74]$ to work on does not depend on the original interval $[1,100]$. The final subinterval $[26,74]$ would not need to be recomputed if the original interval was $[10,90]$ instead. If, on the other hand, both accesses at 25 and 75 are increasing then the final subinterval is $[76,100]$, which certainly depends on the initial interval $[1,100]$. Hence, we must sum over all legal paths with no regard to the initial subintervals, except for those which have only increasing or only decreasing components.

We are now ready to state the general formula for the complexities. Since we are disregarding the initial and final ends of the array, we represent the sequence of accesses just by $ID^*(z,x)$. However, for the case of only increasing or decreasing elements we have to subtract what we have added and replace it by a formula that allows to consider all the possible initial right extremes (for I) and all possible initial left extremes (for D). In the case of increments (the decrements are similar), this is obtained by subtracting $I^*(z,x)$ from $ID^*(z,x)$ and then adding $I^*(z,x)/(1-z)$, since this allows to add an arbitrary number of z's to the right, accounting for all possible positions of the

sequence inside the array. Finally, after a sequence of increasing and decreasing steps, there is a final central segment on which we work. The formula is

$$F(z,x,w) = \left(ID^*(x,y) - I^*(z,x) + I^*(z,x)\frac{1}{1-z} - D^*(z,x) + D^*(z,x)\frac{1}{1-z} \right) \frac{1}{1-wz}$$

which is equal to

$$F(z,x,w) = \left(\frac{1}{1-2xz/(1-z)} + \frac{2z/(1-z)}{1-xz/(1-z)} \right) \frac{1}{1-wz}.$$

4.1. Time complexity

To count the total amount of work to do, we consider that each different subinterval (i,j) of the array reached through a different legal path must be processed. To process such interval, we must consider all its positions from i to j, and compute the worst-case or expected-case cost at each position. To compute such cost, we need the cost of some subintervals. Given that those subintervals are already computed, we work $O(|j-i|+1)$ to solve the subinterval (i,j) given a previous legal path of length k. Hence, what we have to compute is the sum of $|j-i|+1$ for all $i \leqslant j$ for all legal paths of length k which lead to the subinterval (i,j).

Therefore the total amount of work is the coefficient of $z^n x^k$ in the function

$$T(z,x) = \frac{\partial F}{\partial w}(z,x,1) = \sum_{n,k,r \geqslant 0} rF_{n,k,r} z^n x^k.$$

This is correct, since $rF_{n,k,r}$ is the total amount of work to do on an array of size n and legal paths of length k.

We derive the above formula with respect to w and evaluate it at $w=1$, to obtain

$$T(z,x) = \frac{z}{(1-z)^2} \left(\frac{1}{1-2xz/(1-z)} + \frac{2z/(1-z)}{1-xz/(1-z)} \right).$$

To find the coefficient that corresponds to x^k in $T(z,x)$, notice that the coefficient for $1/(1-ax)$ is a^k. Hence

$$T_k(z) = \frac{z}{(1-z)^2} \left(\frac{2^k z^k}{(1-z)^k} + \frac{2z^{k+1}}{(1-z)^{k+1}} \right)$$

and to obtain the coefficient that corresponds to z^n in $T_k(z)$, notice that the coefficient of $1/(1-z)^{m+1}$ is $\binom{n+m}{m}$, and that the coefficient of z^n in $zf(z)$ is that of z^{n+1} in $f(z)$. Consequently, the total amount of work is exactly

$$T_{k,n} = 2^k \binom{n}{k+1} + 2\binom{n}{k+2}$$

which for instance shows that for $k=1$ the amount of work is $T_{1,n}=n^3/3-n/3$. To obtain a more easy to handle formula we point out that $T(k,n)=\Theta(n^{k+2})$. More precisely,

$$\frac{2n^{\underline{k+2}}}{(k+2)!} \leqslant T_{k,n} \leqslant \frac{2^{k+1}n^{\underline{k+2}}}{(k+1)!}$$

holds for $0 \leqslant k < n-1$. This can be checked by induction on k. We recall that $n^{\underline{k}}=n(n-1)(n-2)\ldots(n-k+1)$.

Notice that we have left aside the case of zero-length sequences, where both ends of the initial subinterval must be considered (not only the rightmost or leftmost). Hence, the previous analysis does not apply to $k=0$. In this case we have

$$F_0(z,w)=\frac{1}{1-z}\frac{1}{1-wz}\frac{1}{1-z}$$

which gives, $T_{0,n}=n^3/6+n^2/2+n/3$.

4.2. Space complexity and size of the input

We consider space now. We have to store one cell for each different access path. Hence, instead of being interested in the size of the final central segments, we just count their number. This is equivalent to

$$S(z,x)=F(x,z,1)=\sum_{n,k,r\geqslant 0} F_{n,k,r}z^n x^k$$

which is

$$S(z,x)=\frac{1}{1-z}\left(\frac{1}{1-2xz/(1-z)}+\frac{2z/(1-z)}{1-xz/(1-z)}\right)$$

which gives

$$S_{k,n}=2^k\binom{n}{k}+2\binom{n}{k+1}$$

which is $\Theta(n^{k+1})$. More precisely,

$$\frac{2n^{\underline{k+1}}}{(k+1)!} \leqslant S_{k,n} \leqslant \frac{2^k n^{\underline{k+1}}}{k!}$$

holds for $1 \leqslant k < n-1$.

The size of the input problem has exactly the same complexity. For each possible legal path of length k or less, we have an access cost.

4.3. Inc-dec orderings

Finally, we compute the total number of (i,j)-inc-dec orderings in an array of n elements. In this case, our original interval starts at the root, and hence the $F_0(x,z,1)$

defined before is appropriate, instead of $F(x,z,1)$. Using the same techniques as above, we find $O_{n,k}$, which is the total number of inc-dec orderings of k steps.

However, there is one final problem. When we considered the legal paths leading to each (i,j) interval, each path was counted twice. The reason is that the last comparison could be a *min* or a *max* component of the sequence. This was correct in the previous section because both cases lead to different final intervals to work on. Since we are interested in the number of paths here, we divide the total by two (except when $k=0$). The result, valid for $k>0$, is

$$O_{n,k} = 2^{k-1}\binom{n}{k} = \frac{2^{k-1}n^{\underline{k}}}{k!}$$

(and $O_{n,0}=1$), while if we are not interested in k, we have

$$O_n = \sum_{k=0}^{n} O_{n,k} = (3^n+1)/2.$$

5. Concluding remarks

We have described algorithms for finding optimal binary search trees for a given set $\{x_1,\ldots,x_n\}$ of keys when the cost of each key x_q depends on the $(k+1)$-path leading to x_q. The parameter k is a given arbitrary integer in the range $1\leqslant k<n$. The optimality refers to a tree having either minimal worst case or weighted average case cost. The complexity of both algorithms is $O(n^{k+2})$. It should be noted that although the complexity is an exponential in k, it is polynomial in the input size, in fact $O(n)$ times the input size.

As we pointed out with an example at the end of Section 2, the monotonicity principle of Knuth [3] does not hold for the case $k>0$. Therefore, it seems difficult to improve this algorithm. On the other hand, as mentioned in the same section, the algorithm can be extended without problems to handle also unsuccessful searches.

An open problem is to devise good on-line approximation algorithms, as is done for the case $k=1$ in [11], where a linear time algorithm with a constant average approximation ratio is achieved. Motivated by [11], additional results on this problem have been obtained recently [6,7].

References

[1] E.N. Gilbert, E.F. Moore, Variable-length binary encoding, Bell Syst. Tech. J. 38 (1959) 933–968.
[2] T.C. Hu, A.C. Tucker, Optimal computer search trees and variable-length alphabetical codes, SIAM J. Appl. Math. 21 (1971) 514–532.
[3] D.E. Knuth, Optimum binary search trees, Acta Inform. 1 (1971) 14–25.
[4] D.E. Knuth, The Art of Computer Programming 1: Fundamental Algorithm, Addison-Wesley, Reading, MA, 1968. (2nd Edition, 1973)
[5] D.E. Knuth, The Art of Computer Programming 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

[6] E.S. Laber, R.L. Milidiú, A.A. Pessoa, Strategies for searching with different access costs, in: Proc. ESA'99, Lecture Notes in Computer Science, vol. 1645, Springer, Berlin, 1999, Theoret. Comput. Sci., to appear.
[7] E.S. Laber, R.L. Milidiú, A.A. Pessoa, On binary searching with non-uniform costs, Proc. 12th ACM-SIAM Annu. Symp. Discrete Algorithms 2001, pp. 855–864.
[8] L.L. Larmore, A subquadratic algorithm for constructing approximately optimal binary search trees, J. Algorithms 8 (1987) 579–591.
[9] C. Levcopoulos, A. Lingas, J.R. Sack, Heuristics for optimum binary search trees and minimum weight triangulation problems, Theoret. Comput. Sci. 66 (1989) 181–203.
[10] S.V. Nagaraj, Optimal binary search trees, Theoret. Comput. Sci. 188 (1997) 1–44.
[11] G. Navarro, E. Barbosa, R. Baeza-Yates, W. Cunto, N. Ziviani, Binary searching with non-uniform costs and its applications to text retrieval, Algorithmica 27 (2000) 145–169.
[12] R. Sedgewick, P. Flajolet, Analysis of Algorithms, Addison-Wesley, Reading, MA, 1996.