Improving the Performance of Multidimensional Search Using Fingers

AMALIA DUCH and CONRADO MARTÍNEZ
Universitat Politècnica de Catalunya

We propose two variants of K-d trees where *fingers* are used to improve the performance of orthogonal range search and nearest neighbor queries when they exhibit locality of reference. The experiments show that the second alternative yields significant savings. Although it yields more modest improvements, the first variant does it with much less memory requirements and great simplicity, which makes it more attractive on practical grounds.

Categories and Subject Descriptors: E.1 [**Data Structures**]; F.2 [**Analysis of Algorithms and Problem Complexity**]

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Finger search, multidimensional data structures, orthogonal range searching, nearest-neighbors searching, locality, K-d trees, experimental algorithmics

1. INTRODUCTION

The well-known, time-honored aphorism says that "80% of computer time is spent on 20% of the data." The actual percentages are unimportant, but the moral is that we can achieve significant improvements in performance if we are able to exploit this fact. In on-line settings, where requests arrive one at a time and must be attended as soon as they arrive (or after some small delay), we frequently encounter *locality of reference*, that is, for any time frame only a small number of different requests among the possible ones are made or consecutive requests are close to each other in some sense. Locality of reference is systematically exploited in the design of memory hierarchies (disk and memory caches) and it is the rationale for many other techniques like buffering and self-adjustment [Borodin and El-Yaniv 1998; Sleator and Tarjan 1985a, 1985b].

This research was partially supported by the Future and Emergent Technologies programme of the EU under contract IST-1999-14186 (ALCOM-FT) and the Spanish Min. of Science and Technology project TIC2002-00190 (AEDRI II).
Authors' address: Amalia Duch and Conrado Martínez, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, E-08034 Barcelona, Spain; email: {duch,conrado} at lsi.upc.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2005 ACM 1084-6654/05/0001-ART2.4 $5.00

The performance of searches and updates in data structures can be improved by augmenting the data structure with *fingers*, pointers to the hot spots in the data structure where most activity is going to be performed for a while. Thus, successive searches and updates do not start from scratch, but use the clues provided by the finger(s), so that when the request affects some item in the "vicinity" of the finger(s) the request can be attended very efficiently. In 1980, Brown and Tarjan [1980] introduced the idea of *level-linking* to achieve worst-case search time in $\mathcal{O}(\log \Delta)$ for the search of an item which is Δ items away from the finger. Huddleston and Mehlhorn [1982] later gave algorithms to perform insertions and deletions in finger search trees with constant amortized cost.

The early proposals for fingered data structures of Guibas et al. [1977], Kosaraju [1981], and Tsakalidis [1985] show how to achieve worst-case constant time for insertions and deletions, provided that there is a constant number of fixed locations for insertion and/or deletion. [Harel and Lueker 1979; Harel 1980] designed algorithms to perform insertions and deletions in worst-case $\mathcal{O}(\log^* n)$ time with no restrictions on the fingers. More recent work on finger search trees includes the papers by Dietz and Raman [1994], Brodal [1998] and Brodal et al. [2002].

To the best of the authors' knowledge, fingering techniques have not been applied thus to multidimensional data structures in order to improve associative queries. In this paper, we will specifically concentrate in two variants of K-dimensional trees, namely *standard K-d trees* [Bentley 1975] and *relaxed K-d trees* [Duch et al. 1998], but the techniques can easily be applied to other multidimensional search trees and data structures. In general, multidimensional data structures maintain a collection of items or records, each holding a distinct K-dimensional key (which we may assume w.l.o.g. is a point in $[0, 1]^K$). We will also identify each item with its key and use both terms interchangeably. Besides usual insertions, deletions and exact searches, we will be interested in providing efficient answers to questions like which records fall within a given hyper-rectangle (*orthogonal range search*) or which is the closest record to some given point (*nearest neighbor search*) [Samet 1990; Gaede and Günther 1998]. After a brief summary of basic concepts, definitions and previous results in Section 2, we propose two alternative designs that augment K-d trees[1] with fingers to improve the efficiency of orthogonal range and nearest-neighbor searches (Section 3).

While we cannot provide guarantees for improved search time of these associative searches, we have conducted an extensive experimental study of their performance when using fingers, compared to the performance achieved without using them. This experimental study has been made under reasonable models of associative queries, which exhibit locality of reference (Section 4). It seems difficult to improve the performance of multidimensional data structures using self-adjusting techniques (as reorganizations in this type of data structures

[1]They actually apply to any variant of K-d trees, not just to standard and relaxed K-d trees; additional, although minor, modifications are necessary to adapt these fingering schemes to quad trees, K-d tries, etc.

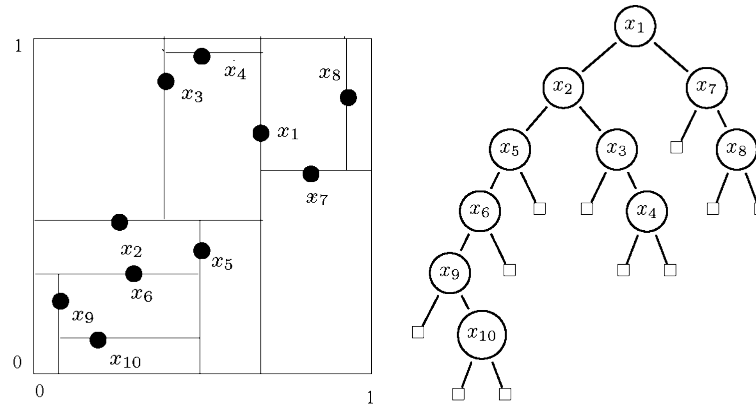

Fig. 1. A standard 2-d tree and the corresponding induced partition of $[0, 1]^2$.

are too expensive); on the contrary, fingering yields significant savings and it is easy to implement. Our experiments show that the second, more complex scheme of m-finger K-d trees better exploits the locality of reference than the simpler 1-finger K-d trees; however, these gains probably do not compensate for the substantial memory that it needs, so that 1-finger K-d trees are more attractive on practical grounds.

A preliminary version of this paper appeared in Duch and Martínez [2004]; the results of this work also appear in Duch [2004].

2. PRELIMINARIES AND BASIC DEFINITIONS

Let $F = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$, $n \geq 0$, be a file of K-dimensional data points, where each $x^{(i)} = (x_0^{(i)}, \ldots, x_{K-1}^{(i)}) \in [0, 1]^K$.

A *standard K-d tree* [Bentley 1975] for a set F of K-dimensional data points is a binary tree in which:

1. Each node contains a K-dimensional data point and has an associated discriminant $j \in \{0, 1, \ldots, K-1\}$ cyclically assigned starting with $j = 0$. Thus, the root of the tree discriminates with respect to the first coordinate ($j = 0$), its sons at level 1 discriminate with respect to the second coordinate ($j = 1$), and, in general, all nodes at level m discriminate with respect to coordinate $j = m \bmod K$;
2. For each node $x = (x_0, x_1, \ldots, x_{K-1})$ with discriminant j, the following invariant is true: any data point y in the left subtree satisfies $y_j < x_j$ and any data point z in the right subtree satisfies $z_j \geq x_j$ (see Figure 1).

Relaxed K-d trees [Duch et al. 1998] are K-d trees where the sequence of discriminants in a path from the root to any leaf is random instead of cyclic. Hence, discriminants must be explicitly stored in the nodes. Other variants of K-d trees use different alternatives to assign discriminants (for instance, the *squarish K-d trees* [Devroye et al. 2000]) or combine different methods to partition the space, not just axis-parallel hyperplanes passing through data

coordinates as in standard and relaxed K-d trees (e.g., the BBD-trees of Arya et al. [1998]).

We say that a K-d tree of size n is *random* if it is built by n insertions where the points are independently drawn from a continuous distribution in $[0, 1]^K$. In the case of random relaxed K-d trees, the discriminants associated to the internal nodes are uniformly and independently drawn from $\{0, \ldots, K-1\}$.

2.1 Orthogonal Range Search

An *(orthogonal) range query* is a K-dimensional hyper-rectangle Q. We shall write $Q = [\ell_0, u_0] \times \cdots \times [\ell_{K-1}, u_{K-1}]$, with $\ell_i \le u_i$, for $0 \le i < K$. Alternatively, a range query can be specified giving its center z and the length of its edges $\Delta_0, \Delta_1, \ldots, \Delta_{K-1}$, with $0 \le \Delta_i \le 1/2$, for $0 \le i < K$. In order to use Theorem 2.1 (see below), we also assume that $-\Delta_i/2 \le z_i \le 1 + \Delta_i/2$ for $0 \le i < K$, so that a range query Q may fall partially outside of $[0, 1]^K$.

Range searching in any variant of K-d trees is straightforward. When visiting a node x that discriminates w.r.t. the jth coordinate, we must compare x_j with the jth range $[\ell_j, u_j]$ of the query. If the query range is totally above (or below) that value, we must search only the right subtree (respectively, left) of that node. If, on the contrary, $\ell_j \le x_j \le u_j$, then both subtrees must be searched; in addition, we must check whether x falls or not inside the query hyper-rectangle. This procedure continues recursively until empty subtrees are reached.

One important concept related to orthogonal range searches is that of *bounding box* or *bounding hyper-rectangle* of a data point. Given an item x in a K-d tree T, its bounding hyper-rectangle $B(x) = [\ell_0(x), u_0(x)] \times \cdots \times [\ell_{K-1}(x), u_{K-1}(x)]$ is the region of $[0, 1]^K$ corresponding to the leaf that x replaced when it was inserted into the tree. Formally, it is defined as follows:

1. If x is the root of T then $B(x) = [0, 1]^K$;
2. If $y = (y_0, \ldots, y_{K-1})$ is the father of x and it discriminates w.r.t. the jth coordinate and $x_j < y_j$ then $B(x) = [\ell_0(y), u_0(y)] \times \cdots \times [\ell_j(y), y_j] \times \cdots \times [\ell_{K-1}(y), u_{K-1}(y)]$;
3. If $y = (y_0, \ldots, y_{K-1})$ is the father of x and it discriminates w.r.t. the jth coordinate and $x_j \ge y_j$ then $B(x) = [\ell_0(y), u_0(y)] \times \cdots \times [y_j, u_j(y)] \times \cdots \times [\ell_{K-1}(y), u_{K-1}(y)]$.

The relation of bounding hyper-rectangles with range search is established by the following lemma.

LEMMA 2.1 [CHANZY ET AL. 2001; DEVROYE ET AL. 2000]. *A point x with bounding hyper-rectangle $B(x)$ is visited by a range search with query hyper-rectangle Q if and only if $B(x)$ intersects Q.*

The cost of orthogonal range queries is usually measured as the number of nodes of the K-d tree visited during the search. It has two main parts: a term corresponding to the unavoidable cost of reporting the result plus an *overwork*, as stated by next theorem.

THEOREM 2.2 [DUCH AND MARTÍNEZ 2002]. *Given a random K-d tree storing n uniformly and independently drawn data points in $[0,1]^K$, the expected overwork $\mathbb{E}[W_n]$ of an orthogonal range search with edge lengths $\Delta_0, \ldots, \Delta_{K-1}$ such that $\Delta_i \to 0$ as $n \to \infty$, $0 \le i < K$, and with center uniformly and independently drawn from $Z_\Delta = [-\Delta_0/2, 1+\Delta_0/2] \times \cdots \times [-\Delta_{K-1}/2, 1+\Delta_{K-1}/2]$ is given by*

$$\mathbb{E}[W_n] = \sum_{1 \le j \le K} \beta_j \cdot n^{\alpha(j/K)} + 2 \cdot (1-\Delta_0) \cdots (1-\Delta_{K-1}) \cdot \log n + \mathcal{O}(1)$$

where

$$\beta_j = \beta(j/K) \cdot \sum_{\substack{w=(w_0,\ldots,w_{K-1}) \in (0+1)^K, \\ \# of\ 1's\ in\ w = j}} \left(\prod_{i:w_i=0} \Delta_i \right) \cdot \left(\prod_{i:w_i=1} (1-\Delta_i) \right)$$

and $\alpha(x)$ and $\beta(x)$ depend on the particular type of K-d trees.

In the case of standard K-d trees $\alpha(x) = 1 - x + \phi(x)$, where $\phi(x)$ is the unique real solution of $(\phi(x)+3-x)^x(\phi(x)+2-x)^{(1-x)} - 2 = 0$; for any $x \in [0,1]$, we have $\phi(x) < 0.07$. For relaxed K-d trees, $\alpha(x) = 1 - x + \phi(x)$, where $\phi(x) = (\sqrt{9-8x} - 3)/2 + x$; for any $x \in [0,1]$, we have $\phi(x) < 0.125$. Of particular interest is the case where $\Delta_i = \Theta(n^{-1/K})$ corresponding to the situation where each orthogonal range search reports a constant number of points, on average. The average cost $\mathbb{E}[R_n]$ is then dominated by the overwork and we have

$$\mathbb{E}[R_n] \sim \mathbb{E}[W_n] = \Theta\left(n^{\phi(1/K)} + n^{\phi(2/K)} + \cdots + n^{\phi(1-1/K)} + \log n\right)$$

2.2 Nearest-Neighbor Search

A *nearest-neighbor query* is a multidimensional point $q = (q_0, q_1, \ldots, q_{K-1})$ lying in $[0,1]^K$. The goal of the search is to find the point in the data structure that is closest to q under a predefined distance measure.

There are several variants for nearest-neighbor searching in K-d trees. One of the simplest, which we will use for the rest of this paper, works as follows. The initial closest point is the root of the tree. We then traverse the tree as if we were inserting q. When visiting a node x that discriminates w.r.t. the jth coordinate, we must compare q_j with x_j. If q_j is smaller than x_j, we follow the left subtree, otherwise we follow the right one. At each step we must check whether x is closer or not to q than the closest point seen so far and update the candidate nearest-neighbor accordingly. The procedure continues recursively until empty subtrees are reached. If the hypersphere, say B_q, defined by the query q and the candidate closest point, is totally enclosed within the bounding boxes of the visited nodes, then the search is finished. Otherwise, we must visit recursively the subtrees corresponding to nodes whose bounding box intersects but does not enclose B_q.

The performance of nearest-neighbor search is similar to the overwork in range search. Given a random K-d tree, the expected cost $\mathbb{E}[NN_n]$ of a nearest-neighbor query q uniformly drawn from $[0,1]^K$ is given by Duch [2004]

$$\mathbb{E}[NN_n] = \Theta(n^\rho + \log n)$$

where $\rho = \max_{0\le s\le K}(\alpha(s/K) - 1 + s/K)$. For standard K-d trees $\rho \in (0.0615, 0.064)$. More precisely, for $K = 2$, we have $\rho = (\sqrt{17}-4)/2 \approx 0.0615536$ and for $K = 3$ we have $\rho \approx 0.0615254$, which is minimal. For relaxed K-d trees $\rho \in (0.118, 0.125)$. When $K = 2$, we have $\rho = \frac{\sqrt{5}}{2} - 1 \approx 0.118$, which is minimal, whereas for $K = 8$, we have $\rho = \frac{1}{8}$, which is maximal.

3. FINGER *K*-D TREES

In this section we introduce two different schemes of fingered K-d trees. We call the first and simpler scheme *one finger K-d tree* (1-finger K-d tree, for short); we augment the data structure with just one finger pointer. The second scheme is called *multiple finger K-d tree* (or m-finger K-d tree, for short). Each node of the new data structure is equipped with two additional pointers or fingers, each pointing to descendent nodes in the left and right subtrees, respectively. The search, in this case, proceeds by recursively using the fingers whenever possible.

3.1 One-Finger *K*-d Trees

Definition 3.1. A *one-finger K-d tree* for a set F of K-dimensional data points is a K-d tree in which:

1. Each node contains its bounding box and a pointer to its parent;
2. There is a pointer called *finger* that points to an arbitrary node of the tree.

The finger is initially pointing to the root, but it is updated after each individual search.

We consider first orthogonal range searches. For 1-finger K-d trees the orthogonal range search algorithm starts the search at some node x pointed to by finger F. Let $B(x)$ be the bounding box of node x and Q the range query. If $Q \subset B(x)$ then (because of Lemma 2.1), all the points to be reported must necessarily be in the subtree rooted at x. Thus, the search algorithm proceeds from x down following the classical range search algorithm. Otherwise, some of the points that are inside the query Q can be stored in nodes which are not descendants of x. Hence, in this situation the algorithm backtracks until it finds the first ancestor y of x such that $B(y)$ completely contains Q. Once y has been found the search proceeds as in the previous case. The finger is updated to point to the node whose bounding box is the smallest one that contains the query rectangle. The idea is that if consecutive queries Q and Q' are close in geometric terms, then either the bounding box $B(x)$ that contains Q also contains Q' or only a limited amount of backtrack suffices to find the appropriate ancestor y to go on with the usual range searching procedure. Of course, the finger is initialized to point to the tree's root before the first search is made. Algorithm 1, herewith, describes the orthogonal range search in 1-finger K-d trees. It invokes the standard range search algorithm (`range_search`) once the appropriate starting point has been found. We use the notation $p \to field$ to refer to the field $field$ in the node pointed to by p.

For simplicity, the algorithm assumes that each node stores its bounding box; this implementation requires $\Theta(n)$ additional memory for the parent pointers,

the finger pointer, and bounding boxes, that is, a total of $n+1$ additional pointers and $2n$ K-dimensional points. However, it is possible to modify the algorithm so that only the nodes in the path from the root to F contain this information or to use an auxiliary stack to store the bounding boxes of the nodes in the path from the root to the finger. In addition, the explicit pointers to the parent can be avoided using pointer reversal plus a pointer to finger's parent or using the same stack that stores the bounding boxes in order to recover the followed path. This codification would use $\Theta(\log n)$ additional memory on average.[2]

Algorithm 1 The orthogonal range-search algorithm for 1-finger K-d trees.

```
▷ F: 1-finger K-d tree, Q: query, S: set of keys
function one_finger_range_search (F, Q, S): 1-finger K-d tree
  if (F = nil) then return F;
  B := F → bounding_box;
  if (Q ⊄ B) then return one_finger_range_search(F → parent, Q, S);
  x := F → info;
  j := F → discr;
  if (Q.u[j] < x[j]) then return one_finger_range_search (F → left, Q, S);
  if (Q.l[j] ≥ x[j]) then return one_finger_range_search (F → right, Q, S);
  if (x∈ Q) then
    S := S ∪{x}
  range_search (F→ left, Q, S);
  range_search (F → right, Q, S);
  return F;
end
```

We denote by $G(T, Q)$ the node $x \in T$ such that $B(x) \supseteq Q$ and no other bounding box of a descendant of x contains Q (the range query). In other words, $B(x)$ is the minimal bounding box that completely contains Q, or the first node where a standard orthogonal range search would recursively follow the two branches.

Ideally $G(T, Q)$ is the node to point to with the finger F, since, in this case, the overwork of the search algorithm is minimal. Notice that the one-finger heuristic is to make F point to $G(T, Q')$, where Q' is the previous query and hope for that $G(T, Q)$ should be close to $G(T, Q')$.

All the nodes that belong to the subtree rooted at $G(T, Q)$ and whose bounding box intersects Q must be visited by Algorithm 1 in order to assure its correctness. In fact these nodes are also visited by the classical range search algorithm (by Lemma 2.1). We call these nodes *hits*. More precisely, we say that a node $x \in T$ is a hit if it is a descendant of $G(T, Q)$ such that $B(x) \cap Q \neq \emptyset$.

To understand how the 1-finger orthogonal range algorithm works, we shall now consider three possible situations according to the relation of $G(T, Q)$ and the node pointed to by F—which we will also call F—: (1) F is a descendant of $G(T, Q)$; (2) $G(T, Q)$ is a descendant of F; and (3) neither F nor $G(T, Q)$ is a descendant of the other.

[2]The necessary additional space is proportional to the height of the K-d tree, which, on average, is $\Theta(\log n)$, but can be as much as $\Theta(n)$ in the worst case.

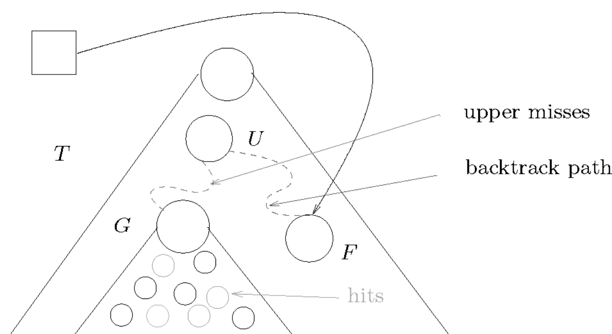

Fig. 2. Special nodes of a 1-finger K-d tree.

Let $U(T, Q, F)$ be the node $x \in T$ with minimum bounding box such that x is an ancestor of F (the finger) and $B(x) \supseteq Q$. In case (1) above, we have $U(T, Q, F) = G(T, Q)$, whereas in case (2), we have $U(T, Q, F) = F$. We say that a node is in the *backtrack path*, if it is in the path from F to $U(T, Q, F)$. Observe that in case (2), since $U(T, Q, F) = F$, the backtrack path is empty.

Note also that $U(T, Q, F)$ is either an ancestor of $G(T, Q)$ or coincides with $G(T, Q)$, and, hence, $U(T, Q, F)$ could have been alternatively defined as the least common ancestor of F and $G(T, Q)$. Figure 2 illustrates all these definitions.

From the previous discussion, it is clear that the 1-finger range-search algorithm visits exactly the following nodes:

1. The nodes in the backtrack path from F to $U(T, Q, F)$; recall that the backtrack path is empty if $G(T, Q)$ is a descendant of F.
2. The nodes in the path from $U(T, Q, F)$ to $G(T, Q)$, which we call *upper misses*; no such nodes exist if F is a descendant of $G(T, Q)$.
3. The descendants of $G(T, Q)$ whose bounding box intersects Q, i.e., the *hits*.

The single finger is exploited for nearest-neighbor searches much in the same vein. Let q be the nearest-neighbor query and x be the node pointed to by finger F. Initially F will point to the root of the tree, but on successive searches it will point to the last closest point reported. The first step of the algorithm is then to calculate the distance d between x and q and to determine the ball with center q and radius d. If this ball is completely included in the bounding box of x then the nearest-neighbor search algorithm proceeds down the tree exactly in the same way as the standard nearest-neighbor search algorithm. If, on the contrary, the ball is not included in $B(x)$, the algorithm backtracks until it finds the least ancestor y whose bounding box completely contains the ball. The algorithm then continues as the standard nearest-neighbor search. Algorithms 2 and 3 describe the nearest-neighbor algorithm for 1-finger K-d trees; notice that they behave just as the standard nearest-neighbor search once the appropriate node to start has been found.

Algorithm 2 The nearest-neighbor search algorithm for 1-finger K-d trees.

```
▷ Precondition: F is not empty
function nearest (F : 1-finger K-d tree, q : query) : key
   one_finger_NN (F, q, ∞ F,);
   return F→ key;
end
```

Algorithm 3 The procedure `one_finger_NN` for the nearest-neighbor search algorithm for 1-finger K-d trees.

```
▷ F: 1-finger K-d tree, q: query, min_dist: distance, nn: 1-finger K-d tree
procedure one_finger_NN (F, q, min_dist, nn): 1-finger K-d tree
  x := F→ info;
  d := dist (q, x);
  B := F→ bounding_box;
  if (d < min_dist) then
    min_dist := d;
    nn := F;
  if (BALL (q,min_dist) ⊄ B) then ▷ Backtrack
    one_finger_NN (F→ parent, q, min_dist, nn);
    j := F → discr;
  if (q[j] < x[j]) then
    one_finger_NN (F → left, q, min_dist, nn);
    other := F → right;
else
    one_finger_NN (F→right, q, min_dist, nn);
    other := F → left;
  if (q[j] − min_dist ≤ x[j] and q[j] + min_dist ≥ x[j]) then
    one_finger_NN (other, q, min_dist, nn);
end
```

3.2 Multiple Finger *K*-d Trees

Definition 3.2. A *multiple-finger K-d tree* (m-finger K-d tree) for a set F of K-dimensional data points is a K-d tree in which:

1. Each node contains its bounding box, a pointer to its parent and
2. Two pointers, `fleft` and `fright`, pointing to two arbitrary nodes in its left and right subtrees, respectively.

Given a m-finger K-d tree T and an orthogonal range query Q, the orthogonal range search in T returns the points in T which fall inside Q, as usual, but it also modifies the finger pointers of the nodes in T to improve the response time of future orthogonal range searches. The algorithm for m-finger search trees recursively applies the 1-finger K-d tree scheme, i.e., jump using the appropriate finger, backtrack if necessary, jump using again a finger, and so on. The fingers of visited nodes are updated as the search proceeds; we have considered that if a search continues in just one subtree of the current node the

finger corresponding to the nonvisited subtree should be reset, because it was not providing useful information. The pseudocode for this algorithm is given as Algorithm 4.

The implementation of m-finger search trees does require $\Theta(n)$ additional memory for the parent pointer, finger pointers, and bounding boxes, that is, a total of $3n$ additional pointers and $2n$ K-dimensional points. This could be a high price for the improvement in search performance which, perhaps, might not be worth paying.

Algorithm 4 The orthogonal range-search algorithm in a m-finger K-d tree.

```
▷ F: m-finger K-d tree, Q: query, S: set of keys
function multiple_finger_range_search (F, Q, S): m-finger K-d tree
    if (F = nil) then return F;
    B:= F → bounding_box;
    if (Q ⊄ B) then ▷ Backtrack
        F → fleft:= F → left;
        F → fright:= F → right;
        return multiple_finger_range_search (F → parent, Q, S );
    x := F → info;
    if (Q.u[j] < x[j]) then
        F→ fright:= F → right;
        F→ fleft : = multiple_finger_range_search (F → fleft, Q, S);
        if (F → fleft = nil) then returnF;
        if (F → fleft = F ) then F → fleft := F → left;
        returnT → fleft;
    if (Q.l[j] ≥ x[j]) then
        F → fleft := F → left;
        F → fright := multiple_finger_range_search (F → fright, Q, S);
        if (F → fright = nil) then return F;
        if (F → fright = F ) then F → fright := F → right;
        return T → fright;
    if (x∈ Q) then S := S ∪ {x};
    F → fleft := multiple_finger_range_search (F→ fleft, Q', S);
    if (F → fleft= nil) then return F;
    if (F → fleft = F ) then F → fleft := F → left;
    F → fright := multiple_finger_range_search (F→ fright, Q", S);
    if (F→ fright = nil) then return F;
    if (F → fright = F ) then F → fright := F → right;
    return F;
end
```

The multiple finger is not defined for nearest-neighbor search. The reason is that while the multiple finger algorithm for orthogonal range search is based on successive decompositions of the (range) query, in the case of nearest neighbor we did not find a suitable way to exploit the decomposition of the hypersphere delimited by the nearest-neighbor query and its so-far closest neighbor. Neither is it clear how fingers can be used to aid in partial match or more general half-plane search queries, with either of the two fingering schemes considered here.

4. LOCALITY MODELS AND EXPERIMENTAL RESULTS

Both 1-finger and m-finger K-d trees try to exploit locality of reference in long sequences of queries, so one of the main aspects of our work was to devise meaningful models on which we could establish how the proposed fingered schemes can improve orthogonal range and nearest-neighbor queries.

The programs used in the experiments described in this section have been written in C, using the GNU compiler `gcc-2.95.4`. The experiments themselves have been run in a computer with Intel Pentium IV CPU at 2.8 GHz with 1 GB of RAM and 512 KB of cache memory.

4.1 The Models

In the case of orthogonal range search, given a size n, and a dimension K, we generate $T = 1000$ sets of n K-dimensional points drawn uniformly and independently at random in $[0, 1]^K$. Each point of each set is inserted into two initially empty trees, so that we get a random standard K-d tree T_s and a random relaxed K-d tree T_r of size n containing both the same information. For each pair (T_s, T_r), we generate $S = 300$ sequences of $Q = 100$ orthogonal range queries and make the corresponding search with the standard and the fingered variants of the algorithm, collecting the basic statistics on the performance of the search.

We have performed experiments with fixed- and variable-size queries, with up to $n = 50{,}000$ elements per tree. For fixed-size queries the length of the K edges of each query was $\Delta = 0.01$. For variable-size queries the length of the edges was dependent of the number of nodes in the tree. The length of each of the K edges was set to $\Delta = \sqrt[K]{1/n}$.

To model locality, we introduced the notion of δ-close queries. We propose two different models: the first one relative to the length size side of the query and the second one independent of the query size.

Definition 4.1. Given two orthogonal range queries Q and Q' with identical edge lengths $\Delta_0, \Delta_1, \ldots, \Delta_{K-1}$, we say that Q and Q' are *δ-close in relative terms* if their respective centers z and z' satisfy $z - z' = (d_0, d_1, \ldots, d_{K-1})$ and $|d_j| \leq \delta \cdot \Delta_j$, for any $0 \leq j < K$.

Definition 4.2. Given two orthogonal range queries Q and Q' with identical edge lengths $\Delta_0, \Delta_1, \ldots, \Delta_{K-1}$, we say that Q and Q' are *δ-close in absolute terms* if their respective centers z and z' satisfy $z - z' = (d_0, d_1, \ldots, d_{K-1})$ and $|d_j| \leq \delta$, for any $0 \leq j < K$.

The sequences of δ-close queries were easily generated by choosing the initial center z_0 uniformly at random in $[-\Delta/2, 1 + \Delta/2]^K$ and setting each successive center $z_{m+1} = z_m + d_m$ for some randomly generated vector d_m; in particular, the ith coordinate of d_m is generated uniformly at random in $[-\delta \cdot \Delta_j, \delta \cdot \Delta_j]$ in the relative model and in $[-\delta, \delta]$ in the absolute one.

The real-valued parameter δ is a simple way to capture into a single number the degree of locality of reference. For instance, in the relative model, if $\delta < 1$ then δ-close queries must overlap at least a fraction $(1 - \delta)^K$ of their volume.

When $\delta \to \infty$ (in fact, it suffices to set $\delta = \max\{\Delta_i^{-1}\}$) there is no locality of reference.

For nearest-neighbor searches, the experimental setup was pretty much the same as for orthogonal search; for each pair (T_s, T_r) of randomly built K-d trees, we perform nearest-neighbor search on each of the $Q = 100$ queries of each of the $S = 300$ generated sequences. As for orthogonal range, we propose two models of locality: the relative model and the absolute model.

Definition 4.3. Successive nearest-neighbor queries q and q' are said to be *δ-close in relative terms* (relative to the number of nodes in the tree), if, $q - q' = (d_0, d_1, \ldots, d_{K-1})$ and $|d_j| \leq \delta \cdot \sqrt[K]{1/n}$, for any $0 \leq j < K$.

Definition 4.4. Successive nearest-neighbor queries q and q' are said to be *δ-close in absolute terms* if $q - q' = (d_0, d_1, \ldots, d_{K-1})$ and $|d_j| \leq \delta$, for any $0 \leq j < K$.

In absolute terms, nearest-neighbor queries are δ-close if the L_∞-norm $\|q - q'\|_\infty$ is smaller than δ. As such, only values in the range $[0, \sqrt{K}]$ are meaningful, although we find it convenient to say $\delta \to \infty$ to indicate that there is no locality of reference.

4.2 The Experiments

4.2.1 *Range Queries.* We show in this section some representative results of the performance of orthogonal range search in relaxed and standard K-d trees, with the model of relative locality and for both fixed- and variable-length queries.

To facilitate the comparison between the standard algorithm and its fingered counterparts, we use the ratio of the respective overworks; namely, if $W_n^{(1)}$ denotes the overwork of 1-finger search, $W_n^{(\mathrm{m})}$ denotes the overwork of m-finger search and $W_n^{(0)}$ denotes the overwork of standard search (no fingers), we will use the ratios $\omega_n^{(m)} = W_n^{(1)}/W_n^{(0)}$ and $\omega_n^{(m)} = W_n^{(\mathrm{m})}/W_n^{(0)}$. Recall that the overwork is the number of visited nodes during a search minus the number of nodes (points) that satisfied the range query. The graphs of Figures 3 and 4 depict $\omega_n^{(m)}$ (in blue) and $\omega_n^{(m)}$ (in red) for $\delta = 0.25$ and $\delta = 2$, respectively, for fixed-size queries (we recall that we have used $\Delta = 0.01$ in all experiments with queries of fixed edge length).

Figure 6 (see later) depicts $\omega_n^{(m)}$ (in blue) and $\omega_n^{(m)}$ (in red) for the case of variable-size queries, locality parameters $\delta = 0.25$ and $\delta = 0.50$ and dimensions $K = 2$ and $K = 3$.

The performance of both 1-finger K-d trees and m-finger K-d trees heavily depends on the locality parameter δ, a fact that is well illustrated by Figures 5 and 7 that show the plot of the ratios $\omega_n^{(m)}$ and $\omega_n^{(m)}$ of relaxed 1-finger and m-finger K-d trees for various values of δ and dimension $K = 2$, for fixed- and variable-size queries, respectively. In particular, when the dimension increases, we shall expect big differences in the savings that fingered search yield as δ varies; for lower dimensions, the variability of $\omega_n^{(m)}$ and $\omega_n^{(m)}$ with δ is not so "steep," but we do not give here a plot showing the phenomenon.

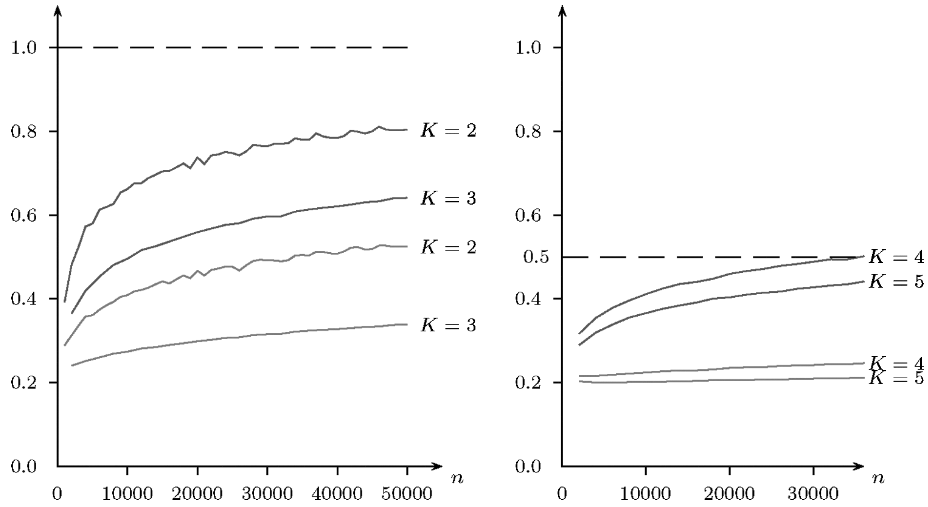

Fig. 3. Overwork ratios for relaxed 1-finger K-d trees (in blue) and m-finger K-d trees (in red), with relative 0.25-close queries, $\Delta = 0.01$ and $K = 2, 3, 4, 5$.

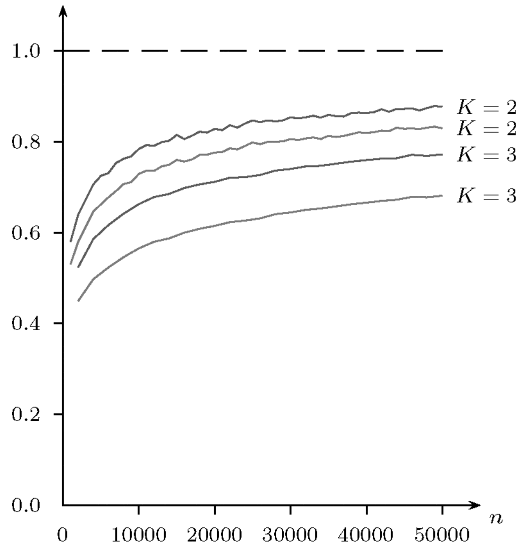

Fig. 4. Overwork ratios for relaxed 1-finger K-d trees (in blue) and m-finger K-d trees (in red), with relative 2-close queries, $\Delta = 0.01$ and $K = 2, 3$.

All these plots confirm that significant savings in the number of visited nodes can be achieved thanks to the use of fingers; in particular, m-finger K-d trees do much better than 1-finger K-d trees for all values of K and δ. As δ increases, the savings with respect to nonfingered search decrease, as we already expected. This is well exemplified by Figure 8, which shows the variation of ω^{m}_{50000} and $\omega^{\mathrm{m}}_{50000}$ as functions of δ for relaxed 1-finger (in blue) and m-finger K-d trees (in red), for $K = 2$ and $K = 3$.

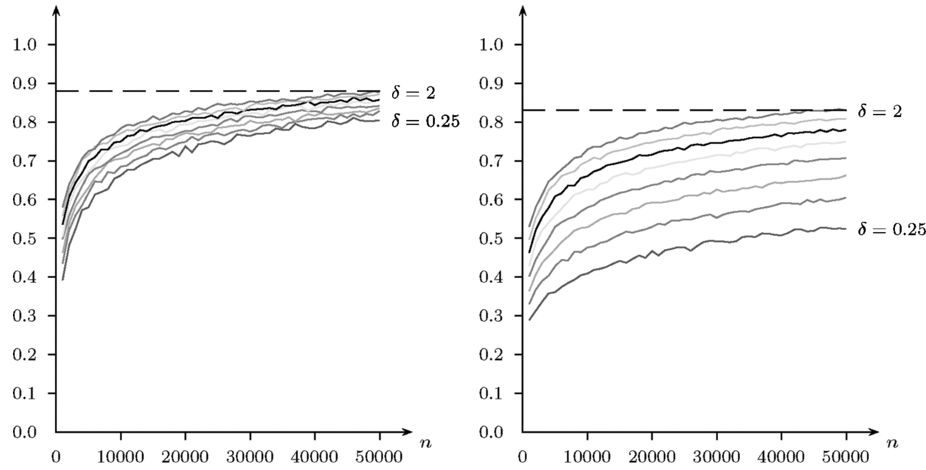

Fig. 5. Overwork ratio in relaxed 1-finger 2-d trees (left) and relaxed m-finger 2-d trees (right), for several values of the locality parameter δ and $\Delta = 0.01$.

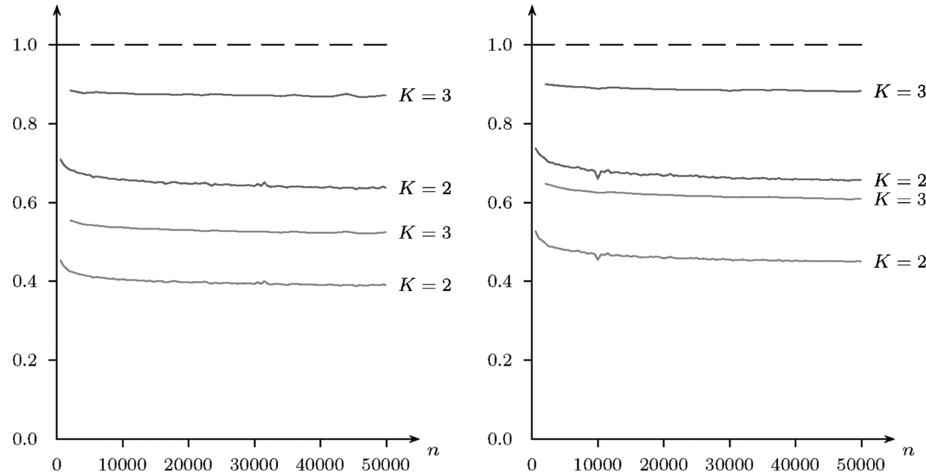

Fig. 6. Overwork ratios for relaxed 1-finger K-d trees (in blue) and m-finger K-d trees (in red), with relative δ-close queries, for $\delta = 0.25$ and $\delta = 0.5$, $\Delta = \sqrt[K]{1/n}$ and $K = 2, 3$.

On the other hand, Figure 9 shows the obvious fact that sufficiently large n variable-length queries should perform better than fixed-length queries.

Last but not least, Figure 10 shows the same data as Figure 3, but for standard K-d trees. Figure 11 compares the performance of fingered standard and relaxed K-d trees. The figure shows that fingering schemes yield similar benefits for standard and relaxed K-d trees, with a slight advantage for standard K-d trees. Both standard orthogonal and fingered range search in standard K-d trees is more efficient than in relaxed K-d trees, as predicted by the available theoretical results and common sense; it is more difficult to intuitively explain why the fingered/nonfingered overwork ratios are also better in standard K-d trees than in relaxed K-d trees.

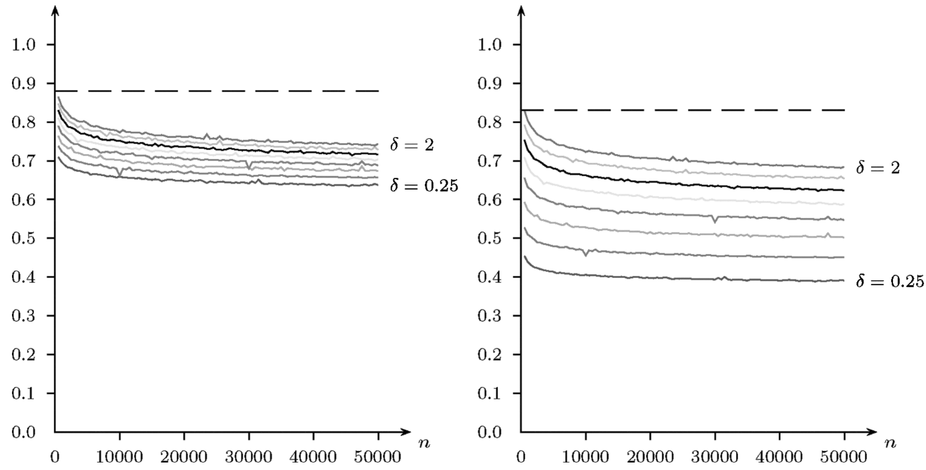

Fig. 7. Overwork ratio in relaxed 1-finger 2-d trees (left) and relaxed m-finger 2-d trees (right), for several values of the locality parameter δ and $\Delta = \sqrt{1/n}$.

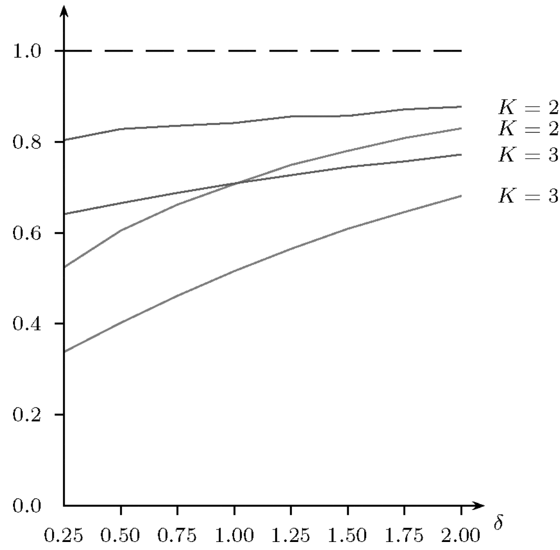

Fig. 8. Overwork ratios for relaxed 1-finger 2-d trees (in blue) and m-finger 2-d trees (in red), with $\Delta = 0.01$, $n = 50000$ and $K = 2, 3$.

Taking into account the way the algorithm works and the theoretical results of Duch and Martínez [2002], we conjecture that 1-finger search reduces by a constant factor the logarithmic term in the overwork. Thus, if standard search has overwork $W_n^{(0)} \approx \sum_{0<j<K} \beta_j n^{\phi(j/K)} + \xi \log n$ then

$$W_n^{(1)} \approx \sum_{0<j<K} \beta_j n^{\alpha(j/K)} + \xi' \log n \qquad (1)$$

with $\xi' = \xi'(\delta)$, which is consistent with the results that we have obtained. However, since the α's and β_j's are quite small, it is rather difficult to disprove this hypothesis on an experimental basis.

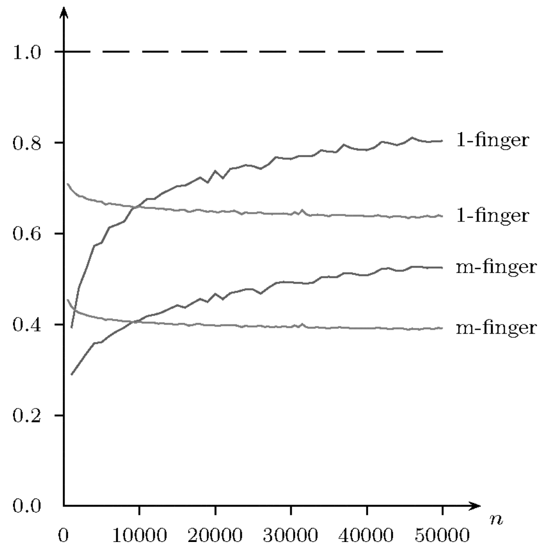

Fig. 9. Overwork ratios for relaxed 1-finger and m-finger 2-d trees, with relative 0.25-close queries, for fixed $\Delta = 0.01$ (blue) and for variable $\Delta = \sqrt{1/n}$ (red).

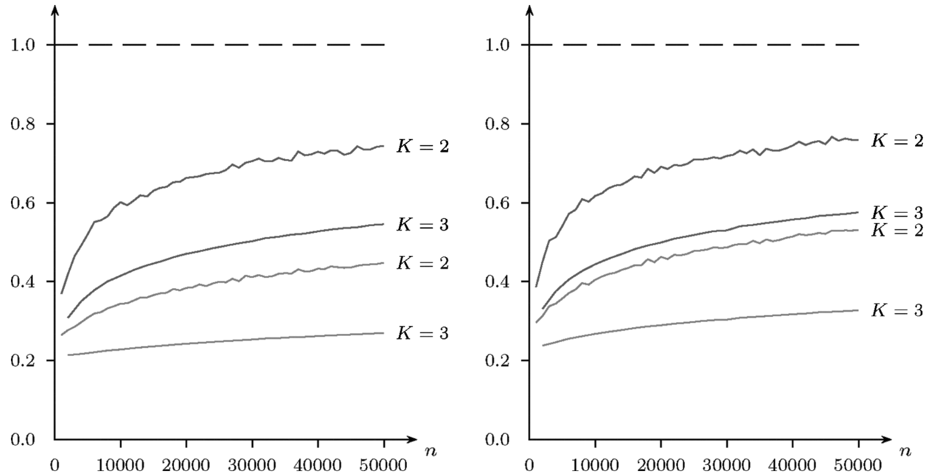

Fig. 10. Overwork ratios for standard 1-finger K-d trees (in blue) and m-finger K-d trees (in red), with relative 0.25-close and 0.5-close queries, $\Delta = 0.01$ and $K = 2, 3$.

On the other hand, and again, following our intuitions on its *modus operandi*, we conjecture that the overwork of m-finger search is equivalent to skipping the initial logarithmic path and then performing a standard-range search on a random tree whose size is a fraction of the total size, say n/x, for some $x > 1$ (basically, the m-finger search behaves as the standard search, but skips more or long intermediate chains of nodes and their subtrees). In other words, we

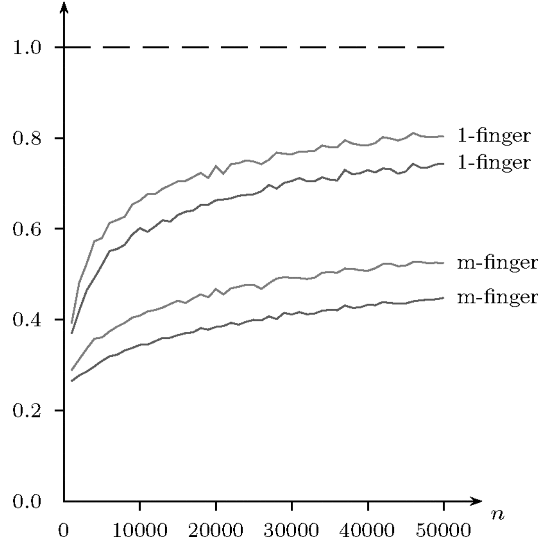

Fig. 11. Overwork ratios for standard 1-finger and m-finger K-d trees (in blue) and relaxed 1-finger and m-finger K-d trees (in red), with relative 0.25-close, $\Delta = 0.01$ and $K = 2$.

Table I. Best-fit β and ξ for Relaxed 2-d Trees

	No finger		1-finger		m-finger	
δ	β	ξ	β	ξ	β	ξ
0.25	0.037	1.912	0.040	0.746	0.028	0.338
0.50	0.037	1.912	0.040	0.836	0.031	0.450
0.75	0.037	1.912	0.040	0.902	0.034	0.550
1.00	0.037	1.912	0.040	0.955	0.035	0.647
1.25	0.037	1.912	0.040	0.999	0.037	0.733
1.50	0.037	1.912	0.040	1.054	0.038	0.824
1.75	0.037	1.912	0.040	1.103	0.039	0.908
2.00	0.037	1.912	0.040	1.151	0.039	0.986

would have

$$W_n^{(\mathrm{m})} \approx \sum_{0<j<K} \beta'_j n^{\alpha(j/K)} + \xi' \log n \qquad (2)$$

for some β'_j's and ξ', which depend on δ (but ξ' here is not the same as for 1-finger search). In this case we face the same problems in order to find experimental evidence against the conjecture.

Table I summarizes the values of $\beta \equiv \beta_1$ and ξ that we obtain by finding the best-fit curve for the experimental results of relaxed 2-d trees. It is worth recalling that the theoretical analysis in Duch and Martínez [2002] predicts for the overwork $W_n^{(0)}$ of standard search with $\Delta = 0.01$ in relaxed 2-d trees the following values: $\phi(1/2) = (\sqrt{5}-1)/2 \approx .618033989$, $\beta = 4\Delta(1-\Delta)\frac{\Gamma(2\alpha+1)}{(\alpha+1)\alpha^3\Gamma^3(\alpha)} \approx 0.03828681802$ and $\xi = 2(1-\Delta)^2 = 1.9602$.

For every experiment, we count the number of nodes visited by the algorithms during backtrack. Since we want to know the amount of the overwork that corresponds to backtrack, we take the respective ratios; that is, if $B_n^{(1)}$ denotes

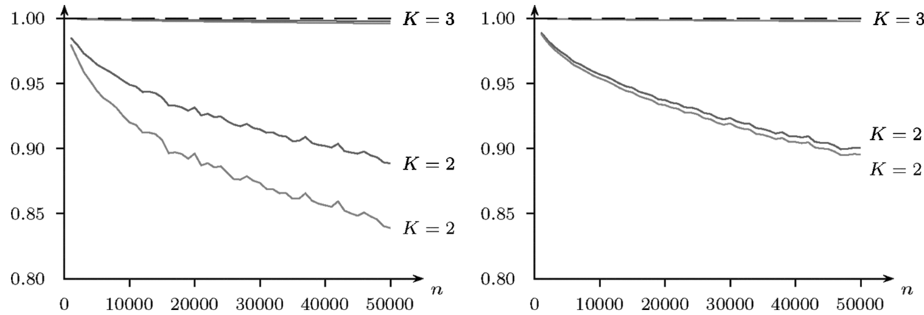

Fig. 12. Backtrack ratios for relaxed 1-finger K-d trees (in blue) and m-finger K-d trees (in red), with relative δ-close queries, for $\delta = 0.25$ (left) and $\delta = 2$ (right), and $K = 2, 3$.

the number of nodes visited in the backtrack path of 1-finger search and B_n^{m} denotes the number of nodes visited in backtrack paths of m-finger search, we will use the ratios $\beta_n^{(1)} = B_n^{(1)}/W_n^{(1)}$ and $\beta_n^{(\mathrm{m})} = B_n^{\mathrm{m}}/W_n^{(\mathrm{m})}$. The graphs of Figure 12 depict $\beta_n^{(1)}$ (in blue) and $\beta_n^{(\mathrm{m})}$ (in red) for $\delta = 0.25$ and $\delta = 2$.

As expected, the amount of backtrack for relaxed m-finger K-d trees is higher than this ratio for relaxed 1-finger K-d trees in dimension $K = 2$. This amount augments as the locality parameter and the dimension increase. In fact, for dimensions ($K \geq 3$), it seems that the amount of backtrack is the whole cost of the overwork. This surprising situation can be explained by the tiny size of the query in higher dimensions; this means that orthogonal range search is almost equivalent to unsuccessful exact search. Therefore, the cost of backtrack dominates the logarithmic cost of reaching a leaf.

Concerning the time of CPU, it turns out that the total amount of CPU time required to answer the sequences of proposed queries in relaxed 1-finger K-d trees is less than the one required for plain relaxed K-d trees, which, in turn, is smaller than the one for m-finger K-d trees.

We use as in our study of overwork and backtrack, the ratios of average CPU times. We use $\tau_n^{(1)} = T_n^{(1)}/T_n^{(0)}$ to denote the ratio of the average CPU times of 1-finger search ($T_n^{(1)}$) and standard search ($T_n^{(0)}$), and $\tau_n^{(\mathrm{m})} = T_n^{(\mathrm{m})}/T_n^{(0)}$ for the ratio of the average CPU times of m-finger search ($T_n^{(\mathrm{m})}$) and standard search. The graphs of Figure 13 depict $\tau_n^{(1)}$ and $\tau_n^{(\mathrm{m})}$ for $\delta = 0.25$. These experiments clearly indicate that m-finger K-d trees are not a good alternative in practice, since they incur too much overhead in memory space and CPU time. These huge requirements of CPU time are because of the high number of pointer updates and other bookkeeping that the m-finger algorithm performs at each recursive step; in other words, there are less visited nodes but each visit is very costly. A finely tuned implementation (our implementation is not) can noticeably reduce the average CPU time per visited node, but it is not likely that this fine tuning would suffice to yield m-finger interesting enough in practice.

4.2.2 *Nearest-Neighbor Queries.* The plots in Figure 14 show the performance of relaxed 1-finger K-d trees when the absolute locality model is used. There, we plot the ratio of the cost using 1-finger nearest-neighbor search to the cost using no fingers. For each dimension K ($K = 2, 3, 4, 5$), the curve

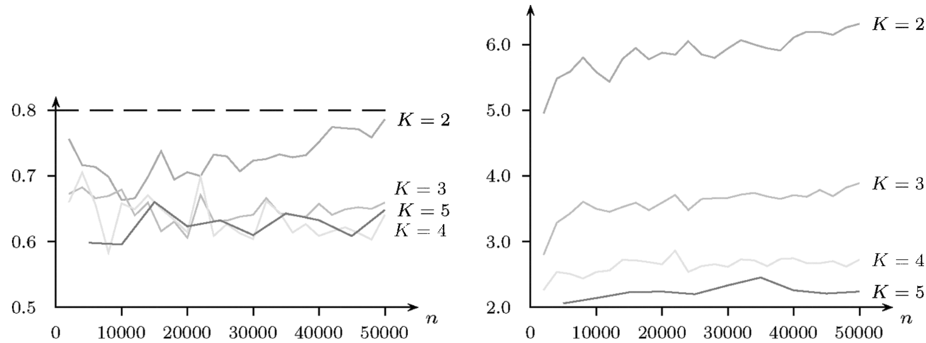

Fig. 13. Time ratios for relaxed 1-finger K-d trees (left) and m-finger K-d trees (right), with relative 0.25-close queries, $\Delta = 0.01$ and $K = 2, 3, 4, 5$.

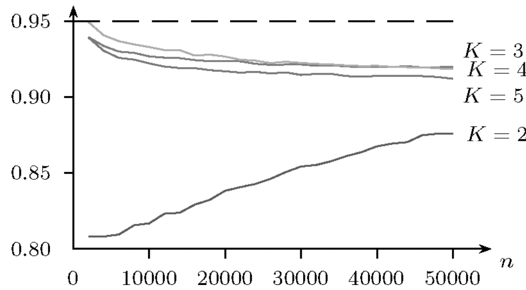

Fig. 14. Nearest-neighbor queries in relaxed 1-finger K-d trees for absolute 0.005-close queries, and $K = 2, 3, 4, 5$.

corresponds to nearest neighbor search with $\delta = 0.005$. The same plot with $\delta = 0.01$ is shown in Figure 15.

It is not a surprise that when we have better locality of reference (a smaller δ) the performance improves. It is more difficult to explain why in this absolute model the variability on δ is smaller as the dimension increases. The qualitatively different behavior for $K = 2$, $K = 3$, and $K > 3$ is also surprising. For $K = 2$ the ratio of the costs increases as n increases until it reaches some stable value (e.g., roughly 90% when $\delta = 0.005$). For $K = 3$ we have rather different behavior when we pass from $\delta = 0.005$ to $\delta = 0.01$. For $K = 4$, $K = 5$, and $K = 6$, we have the same qualitative behavior in all cases[3]: a decrease of the ratio as n grows until the ratio reaches a limit value. A similar phenomenon might occur for $K = 2$ and $K = 3$ provided that δ is even smaller than 0.005. In other words if $\delta \approx \sqrt[K]{1/n}$ or smaller, we then will find the same qualitative behavior as in the case for $K = 3, 4, 5$; but if $\delta \gg \sqrt[K]{1/n}$, we then find a rather different qualitative behavior well represented here by the experiments for $K = 2$.

The plots in Figure 16 show the performance of relaxed 1-finger K-d trees when the relative locality model is used. There, we plot the ratio of the cost using 1-finger nearest-neighbor search to the cost using no fingers. For each

[3]The plot for $K = 6$ is not shown in the figure.

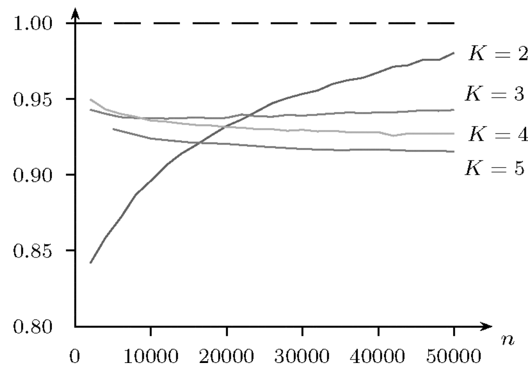

Fig. 15. Nearest-neighbor queries in relaxed 1-finger K-d trees for absolute 0.01-close queries, and $K = 2, 3, 4, 5$.

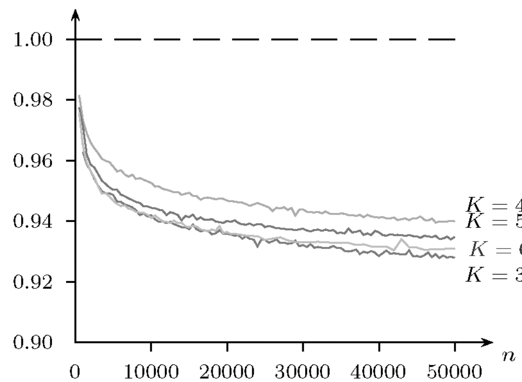

Fig. 16. Nearest-neighbor queries in relaxed 1-finger K-d trees for relative 0.25-close queries, and $K = 3, 4, 5, 6$.

dimension K ($K = 2, 3, 4, 5$), the curves correspond to nearest-neighbor search with $\delta = 0.25 \sqrt[K]{1/n}$. The behavior of nearest-neighbor search for the relative locality model is completely different than for the absolute one. In this case, the ratios decrease as the dimension is augmented, and, for the fixed dimension, they seem to tend to a constant value.

The behavior of the backtrack is shown in the graphs of Figure 17. The plots represent the ratios of the number of nodes visited during backtrack divided by the number of nodes visited by the whole overwork. From the graphs, it can be observed that the amount of backtrack decreases as the dimension increases. For fixed dimension, the amount of backtrack seems to tend to a constant quantity. For the performed experiments, it goes from approximately a 20% for $K = 2$ to a 3% for $K = 5$.

Taking the number of visited nodes as a measure of the cost of the nearest-neighbor algorithm, we did not find significant improvements of 1-finger search with respect to standard search in none of our experiments, in particular, the cost of 1-finger nearest-neighbor search was not below 90% of the standard cost, even for large dimensions and small (but not unrealistic) δ's. However, it is worth to observe that the time of CPU required for 1-finger search is less

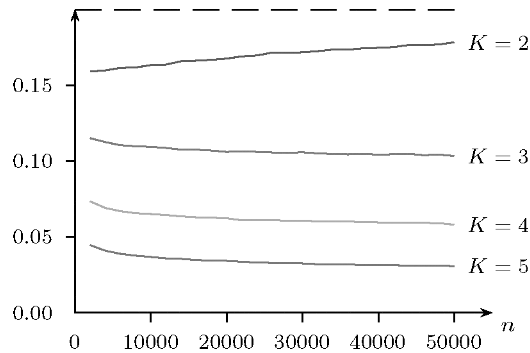

Fig. 17. Backtrack ratios in nearest-neighbor queries for relaxed 1-finger K-d trees for absolute 0.005-close queries, and $K = 2, 3, 4, 5$.

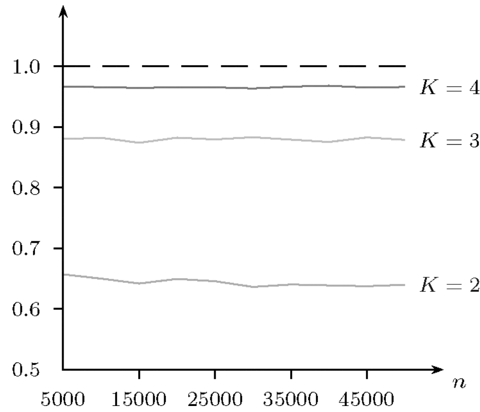

Fig. 18. Time ratios for nearest neighbor search in relaxed 1-finger K-d trees, with relative 0.25-close queries, and $K = 2, 3, 4$.

than the time required for standard search. In this case, the improvements are significant improvements, in particular, for $K = 2$ (see Figure 18). The improvements in CPU time decrease as the dimension increases and they are most likely because of the good memory usage of 1-finger search: starting the search at the point where the previous one finishes, seems to make good use of the memory cache when there is locality of reference. Figure 18 shows the plots of the total CPU time for $K = 2$, 3, and 4 when the relative locality model is used with $\delta = 0.25\sqrt[K]{1/n}$.

5. FINAL REMARKS

In this work, we have introduced two fingering schemes to exploit locality of reference in associative queries on multidimensional data structures. In particular, we have applied these schemes to K-d trees. More precisely, we proposed 1-finger K-d trees and m-finger K-d trees together with meaningful locality models to conduct the experimental study of the performance of orthogonal range and nearest-neighbor queries using these fingering schemes.

We showed that the application of this technique results in a significative amelioration of the performance of hierarchical multidimensional data

structures, since the presented algorithms can be easily applied to other multidimensional data structures, not just K-d trees.

A natural extension of this work would be, for instance, to study the improvements that this technique might yield for nearest-neighbor queries in metric data structures like Burkhard–Keller trees, vantage point trees, GNATs, GHTs, etc. (see Chávez et al. [2001] and the references therein).

Another interesting line of research is to study a generalization of the two schemes considered here; in this more general scheme, a subset of the nodes of a K-d tree are equipped with fingers, while the remaining nodes are not; for instance, all nodes up to some level could have fingers, while the others do not. When a search reaches one of those nodes, the algorithm would follow the fingers associated to the corresponding branches, rather than following the branches themselves. The update of fingers is performed in the same manner as in the 1-finger and m-finger K-d trees. The 1-finger K-d trees correspond to the case where there is a fictitious root node with key $+\infty$, whose left child is the actual root of the tree, and which is the only node that has a finger, namely, pointing to a node on its left subtree. In the opposite end of the spectrum, all nodes of a m-finger K-d tree have fingers to their both subtrees.

ACKNOWLEDGMENTS

We want to thank the anonymous referees for their useful comments and insights to the preliminary conference version of this paper and the submitted manuscript.

REFERENCES

ARYA, S., MOUNT, D., NETANYAHU, N., SILVERMAN, R., AND WU, A. 1998. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM 45*, 6, 891–923.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative retrieval. *Communications of the ACM 18*, 9, 509–517.

BORODIN, A. AND EL-YANIV, R. 1998. *On line Computation and Competitive Analysis*. Cambridge University Press, Cambridge, MA.

BRODAL, G. S. 1998. Finger search trees with constant insertion time. In *Ninth Annual ACM–SIAM Symposium on Discrete Algorithms (SODA)*. 540–549.

BRODAL, G. S., LAGOGIANNIS, G., MAKRIS, C., TSAKALIDIS, A., AND TSICHLAS, K. 2002. Optimal finger search trees in the pointer machine. In *Thirty-Fourth Annual ACM Symposium on Theory of Computing (STOC)*. 583–591.

BROWN, M. R. AND TARJAN, R. E. 1980. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing 9*, 3, 594–614.

CHANZY, P., DEVROYE, L., AND ZAMORA-CURA, C. 2001. Analysis of range search for random k-d trees. *Acta Informatica 37*, 355–383.

CHÁVEZ, E., NAVARRO, G., BAEZA-YATES, R., AND MARROQUÍN, J. 2001. Searching in metric spaces. *ACM Computing Surveys 33*, 3, 273–321.

DEVROYE, L., JABBOUR, J., AND ZAMORA-CURA, C. 2000. Squarish k-d trees. *SIAM Journal on Computing 30*, 1678–1700.

DIETZ, P. F. AND RAMAN, R. 1994. A constant update time finger search tree. *Information Processing Letters 52*, 147–154.

DUCH, A. 2004. Design and analysis of spatial data structures. Ph.D. thesis, Dept. Llenguatges i Sistemes Informàtics, Univ. Politècnica de Catalunya.

DUCH, A. AND MARTÍNEZ, C. 2002. On the average performance of orthogonal range search in multidimensional data structures. *Journal of Algorithms 44*, 1, 226–245.

DUCH, A. AND MARTÍNEZ, C. 2004. Fingered multidimensional search trees. In *Proceeding of the Third International Workshop on Experimental and Efficient Algorithms (WEA)*, C. C. Ribeiro and S. L. Martins, Eds. LNCS, vol. 3059. Springer–Verlag, New York. 228–242.

DUCH, A., ESTIVILL-CASTRO, V., AND MARTÍNEZ, C. 1998. Randomized K-dimensional binary search trees. In *Int. Symposium on Algorithms and Computation (ISAAC '98)*, K.-Y. Chwa and O. H. Ibarra, Eds. LNCS, vol. 1533. Springer-Verlag, New York. 199–208.

GAEDE, V. AND GÜNTHER, O. 1998. Multidimensional access methrods. *ACM Computing Surveys 30*, 2, 170–231.

GUIBAS, L. J., MCCREIGHT, E. M., PLASS, M. F., AND ROBERTS, J. R. 1977. A new representation for linear lists. In *Symp. on the Theory of Computing (STOC)*.

HAREL, D. 1980. Fast updates of balanced search trees with a guaranteed time bound per update. Tech. Rep. 154, Univ. California, Irvine, CA.

HAREL, D. AND LUEKER, G. S. 1979. A data structure with movable fingers and deletions. Tech. Rep. 145, Univ. California, Irvine, CA.

HUDDLESTON, S. AND MEHLHORN, K. 1982. A new data structure for representing sorted lists. *Acta Informatica* 17, 157–184.

KOSARAJU, S. R. 1981. Localized search in sorted lists. In *13th. Ann. ACM Symp. on Theory of Computing (STOC)*. 62–69.

SAMET, H. 1990. *The Design and Analysis of Spatial Data Structures*. Addison–Wesley, Reading, MA.

SLEATOR, D. AND TARJAN, R. E. 1985a. Amortized efficiency of list update and paging rules. *Communications of the ACM 28*, 2, 202–208.

SLEATOR, D. AND TARJAN, R. E. 1985b. Self-adjusting binary search trees. *J. ACM 32*, 3, 652–686.

TSAKALIDIS, A. K. 1985. AVL-trees for localized search. *Information and Computation 67*, 173–194.

Received February 2005; revised December 2005; accepted February 2006