Rank Selection in Multidimensional Data*

Amalia Duch, Rosa M. Jiménez, and Conrado Martínez

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Barcelona, Spain
{duch,jimenez,conrado}@lsi.upc.edu

Abstract. Suppose we have a set of K-dimensional records stored in a general purpose spatial index like a K-d tree. The index efficiently supports insertions, ordinary exact searches, orthogonal range searches, nearest neighbor searches, etc. Here we consider whether we can also efficiently support search by rank, that is, to locate the i-th smallest element along the j-th coordinate. We answer this question in the affirmative by developing a simple algorithm with expected cost $\mathcal{O}(n^{\alpha(1/K)} \log n)$, where n is the size of the K-d tree and $\alpha(1/K) < 1$ for any $K \geq 2$. The only requirement to support the search by rank is that each node in the K-d tree stores the size of the subtree rooted at that node (or some equivalent information). This is not too space demanding. Furthermore, it can be used to randomize the update algorithms to provide guarantees on the expected performance of the various operations on K-d trees. Although selection in multidimensional data can be solved more efficiently than with our algorithm, those solutions will rely on ad-hoc data structures or superlinear space. Our solution adds to an existing data structure (K-d trees) the capability of search by rank with very little overhead, and it can be easily adapted to other spatial indexes as well. The simplicity of the algorithm makes it easy to implement, practical and very flexible; however, its correctness and efficiency are far from self-evident.

1 Introduction

Selection is one of the fundamental computing tasks: given a collection A of n items drawn from a totally ordered domain, and a *rank* i, $1 \leq i \leq n$, the goal is to retrieve the i-th smallest item from A. The selection problem can be trivially solved in time $\mathcal{O}(n \log n)$ by sorting A, but it can be solved more efficiently in either expected linear time [1] or worst-case linear time [2].

Suppose that the collection A is stored in some balanced (or unbalanced) binary search tree. Then we can dynamically mantain the collection, supporting both updates and searches in (expected) time $\mathcal{O}(\log n)$, but we can also support selection in (expected) time $\mathcal{O}(\log n)$ quite easily. We will only need to augment the data structure so that each node stores the size of the subtree rooted at that

* This research was supported by the Spanish Min. of Science and Technology project TIN2006-11345 (ALINEX).

node. This is a very modest price to pay. In fact, the information about subtree sizes can be used advantageously to balance the tree, either probabilistically [3] or deterministically [4]. Hence, we might argue that adding the capability of searching or deleting items by rank comes at no cost.

When dealing with multidimensional point data, we also face frequently the need to sort the data according to one of the coordinates or to find the i-th smallest element along some given coordinate. Those problems can be solved like in the unidimensional case in time $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$, respectively. But it is natural to question if we can do better when the collection of multidimensional points is stored in a data structure like a K-d tree [5] or a quadtree [6] (see also [7, 8] for background on multidimensional data structures).

Here we show that we can select the i-th point along a given coordinate j, $0 \leq j < K$, in expected sublinear time, when the collection of K-dimensional points is stored in a K-d tree. More specifically, for a collection of n points, we can find the answer in expected time $\mathcal{O}(n^{\alpha(1/K)} \log n)$, where $\alpha(x)$ is a function that depends on the type of K-d tree we use. Furthermore, the exponent $\alpha(x) = 1 - x + \phi(x) < 1$ for all $x \in (0, 1)$, with $\alpha(x) \to 1$ as $x \to 0$ (that is, when $1/x = K \to \infty$). Although better performance for rank search in multidimensional data can be easily obtained (using more than linear space, for instance), we stress here that our solution adds efficient rank search to general purpose multidimensional data structures like K-d trees or quadtrees, with only a modest increase of space, namely, storing the size of the subtree rooted at each node. Thus the total space comsumption remains linear in n. Like in the case of ordinary "unidimensional" binary search trees, the information about subtree sizes can be used to randomize the insertion and deletion in K-d trees, thus guaranteeing the expected time bounds of several operations like ordinary search, partial match search, orthogonal range search and nearest neighbor search, even when the dynamic updates are not random [9, 10].

Section 2 briefly summarizes the standard K-d trees and several of its variants, the probabilistic model that will be used in the sequel, and recalls a few important previous results, e. g., the expected cost of partial match search in K-d trees. Then we describe in Section 3 the main contribution of this paper, the algorithm to find the i-th smallest element of a K-d tree T along the j-th coordinate. The following section, Section 4, is devoted to the analysis of the expected cost of the algorithm, and we prove there that this cost is sublinear for any K. Section 5 reports the results of several experiments that we have conducted. The results match very well the predictions of the theoretical analysis in Section 4.

2 Preliminaries

A K-dimensional search tree T (K-d tree, for short) of size $n \geq 0$ stores a set of n K-dimensional records, each holding a key $x = (x_0, \ldots, x_{K-1}) \in D$, where $D = D_0 \times \cdots \times D_{K-1}$, and each D_j is a totally ordered domain. The K-d tree T is a binary tree such that

- Either it is empty and $n = 0$, or
- Its root stores a record with key x and has a discriminant j, $0 \leq j < K$, and the remaining $n - 1$ records are stored in the left and right subtrees of T, say L and R, in such a way that both L and R are K-d trees; furthermore, for any key $u \in L$, it holds that $u_j \leq x_j$, and for any key $v \in R$, it holds that $x_j < v_j$.

We will assume without loss of generality that $D = [0,1]^K$. We will also use the notation $\langle x, j \rangle$ to refer to a node that contains the key x and the discriminant j.

A K-d tree of size n induces a partition of the domain D into $n+1$ regions, each corresponding to a leaf in the K-d tree. The *bounding box* of a node z is the region of the space associated to the leaf replaced by z when it was inserted into the tree. Thus, the bounding box of the root $\langle x, j \rangle$ is $[0,1]^K$, the bounding box of the left subtree's root is $[0,1] \times \cdots \times [0, x_j] \times \cdots \times [0,1]$, and so on.

Different variants of K-d trees have been proposed so far; many differ in the way the discriminants are assigned to nodes. In the original or *standard* K-d trees by Bentley [5], the root of the tree gets discriminant 0, the nodes in the first level get discriminant 1, and so on, in a cyclic fashion. Notice that, since there is a fixed, data-independent rule to assign discriminants to nodes, there is no need to explicitly store the discriminants. Much later Duch et al. [9] proposed *relaxed* K-d trees, where each node is assigned a random discriminant, uniform and independently drawn from $\{0, \ldots, K-1\}$. The *squarish* K-d trees of Devroye et al. [11] try to get a more balanced partition of the space by discriminating along the coordinate for which the bounding box of the node is more elongated. We will consider along the paper the three variants mentioned above, as representative variants of K-d trees.

Because of their definition, the insertion and search algorithms for K-d trees are straightforward, and we will not give here the details. Insertions work identically in the three variants, except in the way discriminants are assigned to newly inserted nodes. The search algorithm is the same for all variants. We also mention here two other algorithms, common to all variants of K-d trees. In *partial match search* we are given a pattern $q = (q_0, \ldots, q_{K-1})$ where $q_j \in [0,1]$ or $q_j = \perp$, for $0 \leq j < K$. Coordinates such that $q_j \neq \perp$ are called *specified*, otherwise they are called *unspecified*; we assume that the number s of specified coordinates satisfies $0 < s < K$. The goal of the partial match search is to retrieve all points in the K-d tree that match the pattern q, that is, the points x such that $x_j = q_j$ whenever $q_j \neq \perp$. To perform a partial match, the K-d tree is recursively explored. First, we check whether the root matches or not the pattern, to report it in the former case. Then, if the root discriminates with respect to an unspecified coordinate, we make recursive calls in both subtrees. Otherwise, if the root containing x discriminates with respect to a specified coordinate j we continue recursively in the appropriate subtree, depending on whether $q_j \leq x_j$ or $x_j < q_j$. The other algorithm is *orthogonal range search*. The input to the algorithm is a K-d tree T and a K-dimensional rectangle $Q = [\ell_0, u_0] \times \cdots \times [\ell_{K-1}, u_{K-1}]$, and the goal is to retrieve all the points in T that lie within Q. The algorithm is very similar to partial match search; the recursion proceeds into one of the subtrees if the

root stores $\langle x, j\rangle$ and $x_j < \ell_j$ or $u_j < x_j$; otherwise we have to make recursive calls in both subtrees and check if x does actually fall inside Q or not.

We now turn our attention to the probabilistic model that we will use later in Section 4, when analyzing the expected performance of our algorithm. We say that a K-d tree built from a given set of n keys is *random* if it is built with identical probability from any of the $n!$ possible input sequences. The discriminants must be assigned according to a fixed rule (standard, squarish K-d trees) or at random (relaxed K-d trees). As a consequence, a K-d tree T of size n is random if and only if it is either empty ($n = 0$), or if its left and right subtrees, L and R, are independent random K-d trees of sizes ℓ and $n-1-\ell$, respectively, with

$$\Pr\left[|L| = \ell \,|\, |T| = n\right] = \frac{1}{n},$$

for any $0 \leq \ell < n$.

There is another equivalent, alternative formulation of the probabilistic model above which is also useful. A random K-d tree of size n is built by n successive random insertions in a initially empty K-d tree. An insertion in a random K-d tree of size n is random if it has the same probability to fail in any of the $n+1$ leaves of the tree. Thus the insertion of n points independently drawn from a continuous distribution in $[0,1]^K$ into an initially empty K-d tree will produce always a random K-d tree.

The probabilistic model for random K-d trees is equivalent, as far as the shape of trees are concerned, to the probabilistic model of binary search trees. It follows then that the expected cost of insertions and the expected cost of exact searches is $\Theta(\log n)$ (see, for instance, [12]).

On the other hand, the expected cost of a partial match search with s random specified coordinates in a random K-d tree of size n is

$$P_n = \beta_q n^{\alpha(s/K)} + \mathcal{O}(1), \qquad 0 < s < K, \tag{1}$$

where β_q is a constant that might depend on the alternance of specified and unspecified coordinates in the pattern q, and $\alpha(x)$ is a function depending on the type of K-d tree that we consider. In all cases, $1 - x \leq \alpha(x) \leq 1$ for $x \in [0,1]$ with $\alpha(x) < 1$ if $x > 0$ and $\alpha \to 1$ as $x \to 0$. For squarish K-d trees $\alpha(x) = 1 - x$ [11], for relaxed K-d trees $\alpha(x) = (\sqrt{9-8x} - 1)/2$ [9, 13] and for standard K-d trees $\alpha(x) = 1 - x + \phi(x)$ [14, 15], where $\phi = \phi(x)$ is the unique solution in $[0,1]$ of

$$(\phi + 3 - x)^x (\phi + 2 - x)^{1-x} - 2 = 0.$$

For instance, for standard K-d trees, $\alpha(1/2) \approx 0.561$, $\alpha(1/3) \approx 0.716$ and $\alpha(1/4) \approx 0.790$. For relaxed K-d trees, we have $\alpha(1/2) \approx 0.618$, $\alpha(1/3) \approx 0.758$ and $\alpha(1/4) \approx 0.823$.

The expected cost of orthogonal range search comes as a combination of partial match costs [16, 17]. The query rectangle Q induces a division of the space

into 2^K regions, which can be indexed with bitstrings of length K. The query rectangle itself is the region $R_{00\ldots0}$. By extending Q along each one of the K coordinates and then substracting Q, we obtain K regions $R_{100\ldots0}, R_{010\ldots0}, \ldots, R_{00\ldots01}$. By extending Q along two coordinates and then substracting Q and all regions of the previous step, we obtain the regions $R_{00\ldots011}, \ldots, R_{110\ldots0}$, and so on. Denoting p_w the probability that a point falls in region R_w when the point is drawn from the continuous distribution in $[0,1]^K$ used to build the random K-d tree, and the center of the query is also drawn using the same distribution, the expected cost of an orthogonal range search is [17]

$$S_n = p_{00\cdots0} \cdot n + 2p_{11\ldots1} \cdot \log n + \sum_{j=1}^{K-1} \sum_{w:w \text{ has } j \text{ ones}} \beta_w p_w n^{\alpha(j/K)} + \mathcal{O}(1), \quad (2)$$

The probabilities p_w will depend on the dimensions $\Delta_0, \ldots, \Delta_{K-1}$ of the query Q and can be though of as the "volumes" of the corresponding regions. For instance, if the data points and the center of the queries are uniformly distributed in $[0,1]^K$ then

$$p_w = \left(\prod_{i:w_i=0} \Delta_i \right) \cdot \left(\prod_{i:w_i=1} (1 - \Delta_i) \right).$$

For the particular case where the query hyperrectangle is a slice $Q = [0,1] \times [0,1] \times \cdots \times [\ell_j, u_j] \times [0,1] \times \cdots \times [0,1]$ the expected cost reduces to

$$S_n = p \cdot n + \beta_{000\ldots1\ldots0} \cdot (1-p) \cdot n^{\alpha(1/K)} + \mathcal{O}(1), \quad (3)$$

since all regions except $R_{00\ldots0} = Q$ and $R_{00\ldots1\ldots0}$ are empty. Here we use p for the probability that a random point falls inside the slice; the first term is thus the expected number of points that fall inside the slice.

3 The algorithm

We present now the algorithm to find the i-th smallest point along the coordinate j, $0 \leq j < K$, in a K-d tree T. The algorithm has three main steps. In the first step, it does a breadth-first traversal of the tree T using a queue Q of pointers to nodes. This first step can also be easily formulated using a recursive preorder traversal of the tree.

During the first step, at any of its iterations, we have a current subtree t and two values *low* and *high* with the guarantee that the j-th coordinate of the sought element is between those two values. The purpose of the first step is either to locate the i-th point along the j-th coordinate—and we would be then done—or to return a reasonably "thin" slice defined by *low* and *high* that must contain the sought element. If the sought element is not found during the first phase, the algorithm performs a convential orthogonal range search to find all the points within the slice $[low, high]$. Finally, the third step finds the sought element using a standard selection algorithm applied to the elements returned by the second step.

Algorithm 1 The first phase of multidimensional selection.

```
kdt kdselect(kdt T, int i, int j) {
    queue<kdt> Q; Q.push(T);
    double low = 0.0;
    double high = 1.0;
    bool found = false;
    kdt t;
    while (not Q.empty() and not found) {
      t = Q.pop(); if (t == NULL) continue;
      if (t -> discr != j) {
        Q.push(t -> left); Q.push(t -> right);
      } else { // t -> discr == j
        double z = t -> key[j];
        if (low <= z and z <= high) {
          int r = below(z, j, T);
          if (i < r) high = z;
          else if (i > r) low = z;
          else found = true;
        }
        if (z <= low) Q.push(t -> right);
        if (z >= high) Q.push(t -> left);
      }
    }
    if (found) return t;
    ...
}
```

We give now a detailed description of the first step. If the current subtree t discriminates with respect to $j' \neq j$, then the sought element could be eventually found in any of its subtrees; therefore, nothing useful can be inferred and both subtrees of t are enqueued for further processing in later iterations. If, on the other hand, the root of t contains $\langle x, j \rangle$, then we have to consider three possibilities. If $x_j < low$ then none of the elements in the left subtree of t can be the sought element; therefore, we enqueue the right subtree of t only. Similarly, if $high < x_j$ then the right subtree of t can be pruned, and we need only to explore (part of) the left subtree of t. Finally, if $low \leq x_j \leq high$ then we compute how many points in the collection, that is, in T, have coordinate j less than or equal to x_j. This is done using the procedure below. Let r be that number. Then if $i = r$ the root of t is the sought element. If $i < r$ then the sought element might be in the left subtree of t but not in its right subtree, thus we push only the left subtree of t into the queue. Furthermore, the j-th coordinate of the sought element must be less than or equal to x_j, hence we set $high := x_j$. If $i > r$ then the sought element cannot be in the left subtree of t, and we enqueue the right subtree of t; additionally, we set $low := x_j$, since the j-th coordinate of the i-th element must be greater than or equal to x_j. This first part of the algorithm is

given in Algorithm 1. Figure 1 illustrates a standard K-d tree with $K = 2$, the partition of $[0,1]^2$ that it induces, and the outcome (the shaded slice) of the first step of kdselect when looking for the 11-th smallest element along coordinate 1 (the y-axis). Note that the number of the items in the figure only indicate their order of insertion.

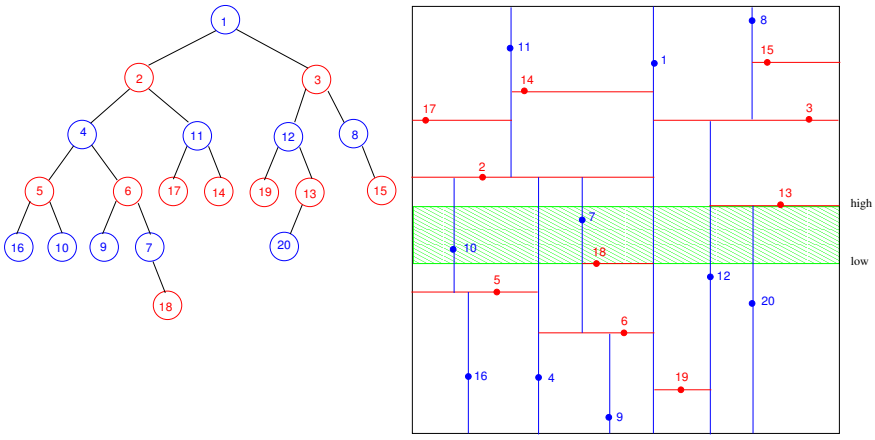

Fig. 1. An example of K-d tree and the execution of Algorithm 1.

To complete our description of the first step, we now draw our attention to the procedure below. It is a simple variation of partial match (see Algorithm 2). The algorithm uses the sizes of subtrees that we store at each node. Using this stored information is essential to avoid computation and thus to achieve a reasonable expected performance. We assume, for convenience, that each node stores its rank relative to its bounding box and the discriminating coordinate, that is, the size of its left subtree plus one.

If the tree is empty we return 0. Otherwise, if the root of T discriminates with respect to a coordinate $\neq j$, we count recursively how many points there are below the given line in both subtrees. We also add one if the root itself meets the condition $x_j \leq z$. If the root of T discriminates with respect to j then we have to continue counting recursively in only one of the subtrees. Note that if $x_j \leq z$ then we count how many points are below z in the right subtree, as all the points in the left subtree and the root itself are below z. We avoid making any traversal of the left subtree since this size is stored at the root of the tree.

4 Analysis

For our analysis of `kdselect`, we will consider that the input K-d tree is random, that the given rank i is random, namely, uniformly distributed in $\{1, \ldots, n\}$, and that the given coordinate j is also uniformly chosen from $\{0, \ldots, K-1\}$.

Algorithm 2 below counts how many points in T have coordinate j less than z

```
int below(double z, int j, kdt T) {
    if (T == NULL) return 0;
    if (T -> discr != j) {
       int c = (T -> key[j] <= z) ? 1 : 0;
       return below(z, j, T -> left) +
              below(z, j, T -> right) + c;
    } else {
       if (z < T -> key[j])
         return below(z, j, T -> left);
       else
         return T -> rank + below(z, j, T -> right);
    }
}
```

The analysis of the expected performance of kdselect is based upon the following ingredients, which we will later prove formally:

1. The number of visited nodes in the main loop of kdselect is at most the number of nodes that we would visit in an orthogonal range search to locate the points that lie within the slice defined by $[low, high]$.
2. The expected cost of a call to below is that of a partial match query with a single specified coordinate (the j-th).
3. The expected number of calls to below is $\mathcal{O}(\log n)$.
4. If the i-th smallest element along coordinate j discriminates along coordinate j, it will be found during the first step; otherwise, the first step will report the smallest slice $[low, high]$ that contains the i-th element along the j-th coordinate and no interior point discriminating along coordinate j.
5. If the first step of kdselect does not find the sought element, then the expected number of points in $[low, high]$ is $\Theta(1)$.

Before going on with a formal proof of each of the statements above, we discuss now how they affect the overall expected performance of kdselect. The expected cost of the first phase will have two contributions, one coming from the calls to below, the other from the main loop. From the items 2 and 3 above, and since the expected cost of a partial match (Eq. (1)) is $\Theta(n^{\alpha(1/K)})$, it follows that the first contribution is $\mathcal{O}(n^{\alpha(1/K)} \log n)$. For the second contribution, we deduce from items 1 and 5 and Eq. (3) that it is $\Theta(n^{\alpha(1/K)})$. In total, the first step of the algorithm has expected cost $\mathcal{O}(n^{\alpha(1/K)} \log n)$. The second and third steps are only necessary if the i-th element has not been found (this happens[1] with probability $(K-1)/K$, when it does not discriminate with respect to j). The second step is an orthogonal range search for points falling in the slice

[1] Actually, for variants of K-d trees such as standard and relaxed K-d trees; in general, for any variant which does not exhibit a bias in the distribution of the coordinates assigned to the discriminants.

$[low, high]$ and has expected cost $\Theta(n^{\alpha(1/K)})$. The third and last step is an ordinary selection algorithm applied to the points found in the previous step; since the expected number of points within the slice is $\Theta(1)$ (item 5), this part has expected cost $\Theta(1)$. Summing up everything we conclude with the following theorem.

Theorem 1. *The expected cost to select the i-th smallest element along coordinate j in a K-d tree of size n is $\mathcal{O}(n^{\alpha(1/K)} \log n)$, where $\alpha(x)$ is a function depending on the variant of K-d tree used, such that $1 - x \leq \alpha(x) \leq 1$ for all $x \in [0,1]$. Furthermore, $\alpha(x) < 1$ for all $x > 0$, and $\alpha(x) \to 1$ as $x \to 0$.*

We now prove the key five statements above. For the first statement, relating the number of iterations of the first phase and the cost of an orthogonal range search, the proof relies in the fact that a node x in a K-d tree is visited during an orthogonal range search with query Q if and only if Q and the bounding box of x intersect $[16, 17]$. Let $\ell_0 = 0$, ℓ_1, $\ldots$, ℓ_r be the sequence of values assigned to the variable low along the execution of Algorithm 1 and similarly, $h_0 = 1$, h_1, $\ldots$, $h_{r'}$ for the values of $high$. Suppose $t = \langle x, j' \rangle$ is the current node, and that its bounding box intersects $[\ell_r, h_{r'}]$. Suppose also that at that iteration $low = \ell_m$ and $high = h_{m'}$. If $j' \neq j$ both subtrees of t will be visited (if they are non-empty) and their corresponding bounding boxes intersect $[\ell_r, h_{r'}]$. If $j' = j$ and $x_j < low = \ell_m$ then only the right subtree of t will be visited. Since $\ell_m \leq \ell_r$, the bounding box of the right subtree of t does intersect $[\ell_r, h_{r'}]$, whereas the bounding box of the left subtree of t does not. For the case where $high = h_{m'} < x_j$, we have that the left subtree is visited and its bounding box intersects $[\ell_r, h_{r'}]$, and the right subtree is not visited and its bounding box does not intersect $[\ell_r, h_{r'}]$. Finally, if $low \leq x_j \leq high$ we will update either low or $high$ (or finish because we find the sought element). If we have not yet finished, the new current $[low, high]$ contains the final $[low, high]$ slice, that is, $[\ell_r, h_{r'}]$, and we apply the same reasoning as above. The basis of this inductive proof is provided by the root of the tree, whose bounding box $[0,1]^K$ obviously intersects the slice $[\ell_r, h_{r'}]$.

The second statement, that the cost of below is that of a partial match is also very easy to prove. The algorithm below behaves exactly as a partial match with a query pattern $q = (\perp, \perp, \ldots, z, \perp \ldots)$, where only the j-th coordinate of q is specified.

For the third statement, where we claim that the expected number of calls to below is $\mathcal{O}(\log n)$ we reason as follows. Consider the points whose j-th coordinate is smaller than or equal to that of the element for which we set the final value of low. Of those, only the points whose bounding box intersects $[low, high]$ will be visited. Furthermore, only a fraction $1/K$ (on average, see the remarks in the footnote of the previous page) of them discriminate with respect to j, so eventually a call to below will be made when visiting them. Since the K-d tree is random, the sequence of j-th coordinates of these points will form a random permutation of $[1, \ldots, N]$, where $N \leq n$ is the number of points discriminating with respect to j, whose bounding box intersects $[low, high]$ and such that its

j-th coordinate is less than or equal to low. Each call to below to update the value of low corresponds to a left-to-right maxima in that permutation, and it is well-known (see for instance [12]) that the expected number of left-to-right maxima in a random permutation of size N is $\Theta(\log N)$. Analogously, each call to below to update the value of $high$ corresponds to a left-to-right minima in the random permutation induced by the sequence of j-th coordinates larger or equal to $high$, for visited points discriminating with respect to j.

The fourth statement says that if the sought element discriminates with respect to the given coordinate j, then it will be found; otherwise, the first phase of the algorithm will terminate returning the slice $[low, high]$ that contains the sought element. The last part follows by design of the algorithm: the invariant of the iteration guarantees that the sought element lies within the slice $[low, high]$. On the other hand, if the sought element discriminates with respect to j and since its bounding box intersects $[low, high]$, sooner or later it will be visited and its rank will be computed using below.

The last statement establishes that the expected number of points within the slice is $\Theta(1)$, when the sought element is not found by the first step of kdselect. By item 4,the sought element does not discriminate with respect to j. Moreover, low and $high$ are the j-th coordinates of two points, say u and v, that discriminate with respect to j; all points properly falling within $[low, high]$ have been visited but do not discriminate with respect to j. The expected number of points in the slice is the number of points that we see when we start from the j-th coordinate of the sought element and go towards $low = u_j$, plus the number of points that we see when we go towards $high = v_j$. These points that we count must not discriminate with respect to j. Since the probability that a point does not discriminate with respect to j is $(K-1)/K$, the expected number of points within the slice in either direction is K, including the two points discriminating with respect to j that define the boundaries of the slice. In total, the expected number of points is $2K+1$.

5 Experiments

To corroborate the analysis of kdselect we have performed a preliminary set of experiments in two diferent variants of K-d trees (standard and relaxed).

For every dimension going from $K = 2$ to $K = 4$, we generate M K-d trees of size n, with n going from 1000 to 50000 with a step of 1000 elements. In each tree we look for the i-th element (with i going from 1 to n with a step of $n/100$) in each of the K possible coordinates (going from 0 to $K-1$). For each tree we count the total number of visited nodes in the main loop of kdselect, the number of calls to function below and the number of points lying in interval $[low, high]$ and take the corresponding averages.

Figure 2 contains the experimental results regarding the total number of visited nodes in the main loop of kdselect. In particular, we plot the ratio of the number of visited nodes to $n^{\alpha(1/K)}$, so the figures exhibit the convergence of the ratio to a constant factor as n grows.

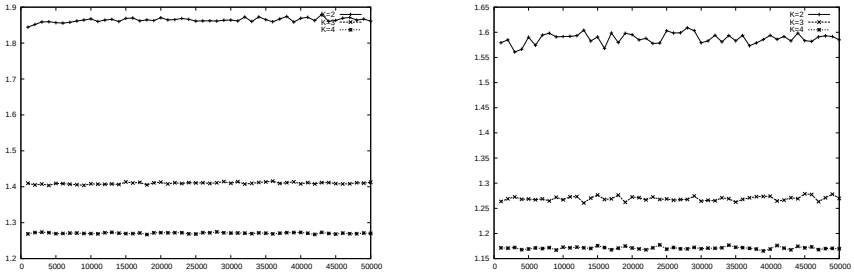

Fig. 2. Number of visited nodes in the main loop of Algorithm 1 divided by $n^{\alpha(1/K)}$, for standard (left) and relaxed (right) K-d trees.

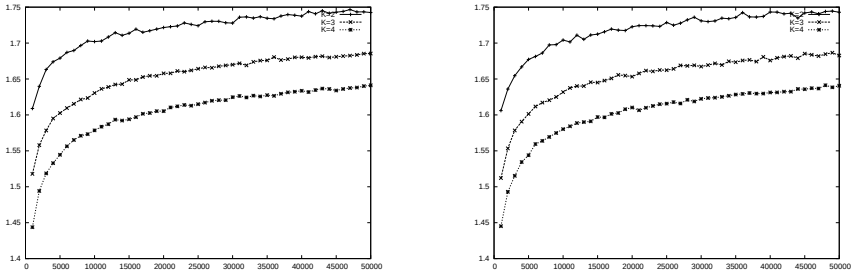

Fig. 3. Number of calls to Algorithm 2 divided by $\log n$ in standard (left) and relaxed (right) K-d trees.

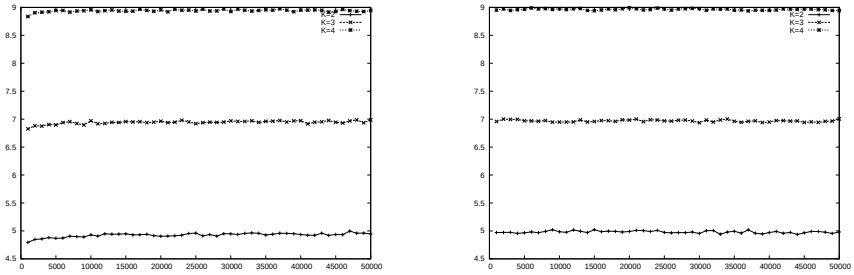

Fig. 4. Number of points within interval $[low, high]$ in in standard (left) and relaxed (right) K-d trees.

The number of calls to below can be found in Figure 3; we actually plot the ratio between the number of calls to below and $\log n$, which converges to a constant factor depending on K.

Finally, Figure 4 shows the number of points contained in the interval $[low, high]$. The experiments confirm very well the predicted value $2K + 1$, that does not depend on the variant of K-d trees considered.

References

1. Hoare, C.A.R.: FIND (Algorithm 65). Comm. ACM **4** (1961) 321–322
2. Blum, M., Floyd, R., Pratt, V., Rivest, R., Tarjan, R.: Time bounds for selection. J. Comp. Syst. Sci. **7** (1973) 448–461
3. Martínez, C., Roura, S.: Randomized binary search trees. J. Assoc. Comput. Mach. **45**(2) (1998) 288–323
4. Roura, S.: A new method for balancing binary search trees. In Orejas, F., Spirakis, P.G., van Leeuwen, J., eds.: Proc. of the 28$^{\text{th}}$ Int. Col. on Automata, Languages and Programming (ICALP). Volume 2076 of Lecture Notes in Computer Science., Springer-Verlag (2001) 469–480
5. Bentley, J.L.: Multidimensional binary search trees used for associative retrieval. Comm. ACM **18**(9) (1975) 509–517
6. Bentley, J., Finkel, R.: Quad trees: A data structure for retrieval on composite keys. Acta Informatica **4** (1974) 1–9
7. Samet, H.: The Design and Analysis of Spatial Data Structures. Addison-Wesley (1990)
8. Gaede, V., Günther, O.: Multidimensional access methods. ACM Computing Surveys **30**(2) (1998) 170–231
9. Duch, A., Estivill-Castro, V., Martínez, C.: Randomized k-dimensional binary search trees. In Chwa, K.Y., Ibarra, O., eds.: Proc. of the 9$^{\text{th}}$ Int. Symp. on Algorithms and Computation (ISAAC). Volume 1533 of Lecture Notes in Computer Science., Springer-Verlag (1998) 199–208
10. Duch, A., Martínez, C.: Updating relaxed k-d trees. ACM Trans. on Algorithms (2008) Accepted for publication.
11. Devroye, L., Jabbour, J., Zamora-Cura, C.: Squarish k-d trees. SIAM J. Comput. **30** (2000) 1678–1700
12. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching. 2nd edn. Volume 3. Addison–Wesley (1998)
13. Martínez, C., Panholzer, A., Prodinger, H.: Partial match queries in relaxed multidimensional search trees. Algorithmica **29**(1–2) (2001) 181–204
14. Flajolet, P., Puech., C.: Partial match retrieval of multidimensional data. J. Assoc. Comput. Mach. **33**(2) (1986) 371–407
15. Chern, H.H., Hwang, H.K.: Partial match queries in random k-d trees. SIAM J. Comput. **35**(6) (2006) 1440–1466
16. Chanzy, P., Devroye, L., Zamora-Cura, C.: Analysis of range search for random k-d trees. Acta Informatica **37** (2001) 355–383
17. Duch, A., Martínez, C.: On the average performance of orthogonal range search in multidimensional data structures. J. Algorithms **44**(1) (2002) 226–245