Efficient Searching with Linear Constraints

(Extended Abstract)

Pankaj K. Agarwal*[†] Lars Arge*[‡] Jeff Erickson*[§]

Paolo G. Franciosa[¶] Jeffrey Scott Vitter*[‖]

Abstract

We show how to preprocess a set S of points in $\mathbb{R}^d$ into an external memory data structure that efficiently supports *linear-constraint* queries. Each query is in the form of a linear constraint $\mathbf{a} \cdot \mathbf{x} \leq b$; the data structure must report all the points of S that satisfy the constraint. Our goal is to minimize the number of disk blocks required to store the data structure and the number of disk accesses (I/Os) required to answer a query. For $d = 2$, we present the first near-linear size data structure that can answer linear-constraint queries using an optimal number of I/Os. We also present a linear-size data structure that can answer queries efficiently in the worst case. We combine these two approaches to obtain tradeoffs between space and query time. Finally, we show that some of our techniques extend to higher dimensions.

*Center for Geometric Computing, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129; {pankaj,large,jeffe,jsv}@cs.duke.edu; http://www.cs.duke.edu/~{pankaj,large,jeffe,jsv}.

[†]Supported in part by National Science Foundation research grant CCR–93–01259, by Army Research Office MURI grant DAAH04–96–1–0013, by a Sloan fellowship, by a National Science Foundation NYI award and matching funds from Xerox Corporation, and by a grant from the U.S.-Israeli Binational Science Foundation.

[‡]Supported in part by U.S. Army Research Office MURI grant DAAH04–96–1–0013.

[§]Supported in part by National Science Foundation grant DMS–9627683 and by U.S. Army Research Office MURI grant DAAH04–96–1–0013.

[¶]Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Via Salaria 113, 00198, Rome; pgf@dis.uniroma1.it; http://www-ae.dis.uniroma1.it/ae/people/franciosa/. Supported in part by EU Project no. 20244 (Alcom-IT). Part of this work was done while visiting Duke University under the Short Term Mobility program of CNR.

[‖]Supported in part by U.S. Army Research Office MURI grant DAAH04–96–1–0013 and by National Science Foundation grant CCR–9522047.

1 Introduction

In order to be successful, any data model in a large database requires efficient external memory (secondary storage) support for its language features. Range searching and its variants are problems that often need to be solved efficiently. In relational database systems and in SQL, for example, one-dimensional range search is a commonly used operation [30, 40]. A number of special cases of two-dimensional range searching are important for the support of new language features, such as constraint query languages [30] and class hierarchies in object-oriented databases [30, 39]. In spatial databases such as geographic information systems (GIS), range searching obviously plays an extremely important role, and a large number of external data structures for answering such queries have been developed (see, for example, [42, 37]). While most attention has focused on *isothetic* or *orthogonal* range searching, where a query is a d-dimensional axis-aligned hyper-rectangle, the importance of non-isothetic queries has also been recognized, most recently in [17, 22]. Many of the proposed data structures can be used to answer non-isothetic queries.

In this paper we develop efficient data structures for what in the computational geometry community is called *halfspace range searching*, where a query is a *linear constraint* of the form $\mathbf{a} \cdot \mathbf{x} \leq b$ and we wish to report all input points that satisfy this constraint. Our goals are to minimize the number of disk blocks required to store the data structure and to minimize the number of disk accesses required to answer a query. Halfspace range searching is the simplest form of non-isothetic range searching and the basic primitive for more complex queries.

1.1 Problem statement

The *halfspace range searching* problem is defined as follows:

> Preprocess a set S of N points in $\mathbb{R}^d$ into a data structure so that all points satisfying a query constraint $\mathbf{a} \cdot \mathbf{x} \leq b$ can be reported efficiently.

Note that a query corresponds to reporting all points below a query hyperplane h defined by $\mathbf{a} \cdot \mathbf{x} = b$. An example of a simple query that can be interpreted as a halfspace range query is the following [22]: Given a relation

$$\mathit{Companies}(\mathit{Name}, \mathit{PricePerShare}, \mathit{EarningsPerShare}),$$

retrieve the names of all companies whose price-earnings ratio is less than 10. In SQL the query can be expressed as follows:

SELECT *Name* FROM *Companies*
WHERE (*PricePerShare* $-$ 10 $*$ *EarningsPerShare* < 0)

If we interpret each ordered pair (*EarningsPerShare*, *PricePerShare*) as a point in the plane, the result of the query consists of all such points that satisfy the linear constraint line $y - 10x \leq 0$. Several complex queries can be viewed as reporting all points lying within a given convex query region. Such queries can in turn be viewed as the intersection of a number of halfspace range queries.

As our main interest is minimizing the number of disk blocks used to store the points and the number of disk accesses needed to answer a halfspace range query, we will consider the problem in the standard external memory model. In this model it is assumed that each disk access transmits in a single *input/output operation* (or *I/O*) a contiguous block of B units of data. The efficiency of a data structure is measured in terms of the amount of disk space it uses (measured in units of disk blocks) and the number of I/Os required to answer a halfspace range query. As we are interested in solutions that are *output sensitive*, our query I/O bounds are not only expressed in terms of N, the number of points in S, but also in terms of T, the number of points reported by the query. Note that the minimum number of disk blocks we need to store N points is $\lceil N/B \rceil$. Similarly, at least $\lceil T/B \rceil$ I/Os are needed to report T output points. We refer to these bounds as "linear" and introduce the notation $n = \lceil N/B \rceil$ and $t = \lceil T/B \rceil$.

1.2 Previous results

In recent years tremendous progress has been made on non-isothetic range searching in the computational geometry community; see the recent surveys [3, 34] and the references therein. As mentioned, halfspace range searching is the simplest form of non-isothetic range searching and thus the problem has been especially extensively studied. Unfortunately, all the results are obtained in main memory models of computation where I/O efficiency is not considered. Most of the developed data structures are inefficient when mapped to external memory.

The practical need for I/O support has led to the development of a large number of external data structures in the spatial database community. B-trees and their variants [6, 15] have been an unqualified success in supporting one-dimensional range queries. B-trees occupy $O(n)$ space and answer queries in $O(\log_B n + t)$ I/Os, which is optimal. Numerous structures have been proposed for range searching in two and higher dimensions, for example, grid files [36], quad-trees [42, 43, 8], k-d-B-trees and variants [41, 27], hB-trees [19, 31], and R-trees and variants [7, 25, 29, 44, 9]. (More references can be found in the surveys [3, 24, 28, 37].) Although these data structures have good average-case query performance for common geometric searching problems, their worst-case query performance is much worse than the $O(\log_B n + t)$ I/O bound obtained in one dimension using B-trees. One key reason for this discrepancy is the important practical restriction that the structures must use near-linear space. Recently, some progress has been made on the construction of structures with provably good performance for (special cases of) two-dimensional [5, 30, 40, 45] and three-dimensional [46] isothetic range searching.

Even though the practical data structures mentioned above are often presented as structures for performing isothetic range searching, most of them can be easily modified to answer non-isothetic queries and thus also halfspace range queries. However, the query performance often seriously degrades. For example, even though we can answer halfspace range queries for uniformly distributed points in the plane in $O(\sqrt{n}+t)$ I/Os using data structures based on quad trees, the query performance can be as bad as $\Omega(n)$ I/Os even for reasonable distributions. The latter number of I/Os is required, for example, if S consists of N points on a diagonal line ℓ and the query halfplane is bounded by a line obtained by a slight perturbation of ℓ. In this case $\Omega(n)$ nodes of the tree are visited by the query algorithm. Similar performance degradation can be shown for the other mentioned structures. In the internal memory model, a two-dimensional halfspace query can be answered in time $O(\log_2 N+T)$ time using $O(N)$ space [12], but it may require $O(\log_2 N + T)$ I/Os in terms of the external memory model. The only known external memory data structure with provably good query performance works in two dimensions, where it uses $O(n\sqrt{N})$ blocks of space and answers queries using optimal $O(\log_B n + t)$ I/Os [20, 21].

1.3 Our results

In Section 3, we give the first known data structure for answering two-dimensional halfspace range queries in optimal $O(\log_B n + t)$ I/Os using near-linear size. In fact, we present two different structures that achieve this bound. Both structures use $O(n \log_2 n)$ blocks of space and are simple enough to be of use in practice. Both structures are based on the geometric technique called *filtering search* [10, 11, 13].

As mentioned, practical considerations often prohibit the use of more than linear space. In Section 4, we

d	Query I/Os	Space
2	$O(\log_B n + t)$	$O(n \log_2 n)$
	$O(n^{\varepsilon} + t)$	$O(n \log_B n)$
	$O((n/B^a)^{1/2+\varepsilon} + t)$	$O(n \log_2 B)$
	$O(n^{1/2+\varepsilon} + t)$	$O(n)$
3	$O(\log_B n + t)$	$O(N(\log_2 n) \log_B n)$
	$O(n^{2/3+\varepsilon} + t)$	$O(n)$
d	$O(n^{1-1/\lfloor d/2 \rfloor+\varepsilon} + t)$	$O(n \log_B n)$
	$O(n^{1-1/d+\varepsilon} + t)$	$O(n)$

Table 1. Our main results.

present the first linear-size structure with provably good worst-case query performance. Our basic data structure answers two-dimensional queries in $O(n^{1/2+\varepsilon} + t)$ I/Os for any constant $\varepsilon > 0$. It can also report points lying inside a query triangle within the same time bound. The data structure generalizes to answer d-dimensional halfspace range queries in $O(n^{1-1/d+\varepsilon} + t)$ I/Os.

In Section 5, we describe how to trade space for query performance by combining the two previous results. We can answer a query in $O((n/B^a)^{1/2+\varepsilon} + t)$ I/Os, for any constant $a > 0$, using slightly superlinear space $O(n \log_2 B)$, or in $O(n^{\varepsilon} + t)$ I/Os using $O(n \log_B n)$ blocks of space. This last data structure generalizes to answer d-dimensional halfspace queries in $O(n^{1-1/\lfloor d/2 \rfloor+\varepsilon} + t)$ I/Os in the same space bound.

In Section 6, we discuss halfspace range searching in three dimensions. We describe a data structure with optimal query time that uses $O(N(\log_2 n) \log_B n)$ space.

Our main results are summarized in Table 1.

2 Geometric Preliminaries

In order to state our results we need some concepts and results from computational geometry.

2.1 Duality

Duality is a popular and powerful technique used in geometric algorithms; it maps a point in $\mathbb{R}^d$ to a hyperplane in $\mathbb{R}^d$ and vice-versa. We use the following duality transform: The dual of a point $(a_1, \ldots, a_d) \in \mathbb{R}^d$ is the hyperplane $x_d = -a_1 x_1 - \cdots - a_{d-1} x_{d-1} + a_d$, and the dual of a hyperplane $x_d = b_1 x_1 + \cdots + b_{d-1} x_{d-1} + b_d$ is the point $(b_1, \ldots, b_d)$. Let σ^* denote the dual of an object (point or hyperplane) σ; for a set of objects Σ, let $\Sigma^* = \{\sigma^* \mid \sigma \in \Sigma\}$.

An essential property of duality is that it preserves the above-below relationship (see Figure 1):

Lemma 2.1. *A point p is above (resp., below, on) a hyperplane h if and only if the dual hyperplane p^* is above (resp., below, on) the dual point h^*.*

By Lemma 2.1, the points in S that lie below a hyperplane h dualize to hyperplanes in S^* that lie below

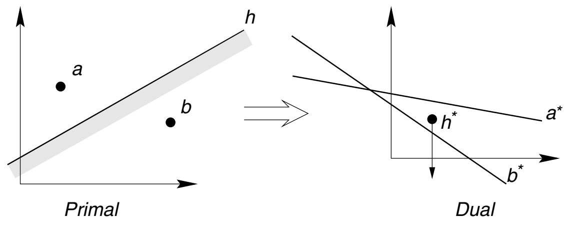

Figure 1. The duality transform in two dimensions.

the point h^*. Hence, the halfspace range searching problem has the following equivalent "dual" formulation:

> Preprocess a set H of n hyperplanes in $\mathbb{R}^d$ so that the hyperplanes lying below a query point p can be reported efficiently.

2.2 Arrangements

Let H be a set of N hyperplanes in $\mathbb{R}^d$. The *arrangement* of H, denoted as $\mathcal{A}(H)$, is the decomposition of $\mathbb{R}^d$ into cells of dimensions k, for $0 \le k \le d$, each cell being a maximal connected set of points contained in the intersection of a fixed subset of H and not intersecting any other hyperplane of H. For example, a set of lines in the plane induces a decomposition of the plane into vertices, edges, and two-dimensional faces; see Figure 2(i).

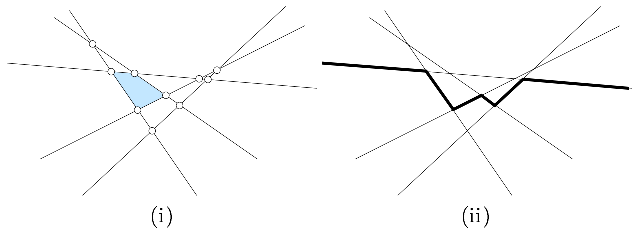

Figure 2. (i) An arrangement of lines, with one cell shaded; (ii) the 2-level of the arrangement.

2.3 Levels

The *level* of a point $p \in \mathbb{R}^d$ with respect to H is the number of hyperplanes of H that lie (strictly) below p. All the points in a single cell of arrangement $\mathcal{A}(H)$ lie above the same subset of hyperplanes of H, so we can define the level of a cell of $\mathcal{A}(H)$ to be the level of any point in that cell. For any $0 \le k < n$, the *k-level* of $\mathcal{A}(H)$, denoted $\mathcal{A}_k(H)$, is the closure of all the $(d-1)$-dimensional cells whose level is k; it is a monotone piecewise-linear $(d-1)$-dimensional surface. For example, the k-level in an arrangement of lines is a x-monotone polygonal chain. Figure 2(ii) depicts the 2-level in an arrangements of lines in the plane. An algorithm by Edelsbrunner and Welzl [18] can compute a two-dimensional level with ν edges in $O(\nu \log_2^2 n)$ time. Their algorithm uses the dynamic convex-hull algorithm by Overmars and van-Leeuwen [38], which in turn uses a two-level red-black tree. Replacing the second level red-black tree by a B-tree, the number of I/Os required to compute the level can be reduced to $O(\nu(\log_2 n)\log_B n)$,

Little is known about the worst-case complexity of the k-level. A recent result of Dey [16] shows that the maximum number of vertices on the k-level in an arrangement of N lines in the plane is $O(Nk^{1/3})$. For $d = 3$, the best known bound on the complexity of $\mathcal{A}_k(H)$ is $O(Nk^{5/3})$ [1]. Neither of these bounds is known to be tight. However, if we choose a random level of $\mathcal{A}(H)$, a better bound can be proven using a result of Clarkson and Shor [14]; see, for example, Agarwal *et al.* [4].

Lemma 2.2. *Let H be a set of N hyperplanes in $\mathbb{R}^d$. For any $1 \le i \le \lfloor N/2 \rfloor$, if we choose a random integer k between i and $2i$, the expected complexity of $\mathcal{A}_k(H)$ is $O(N^{\lfloor d/2 \rfloor}k^{\lceil d/2 \rceil - 1})$.*

3 Query-Optimal Data Structures in 2D

In this section we describe two data structures for halfspace range searching in the plane. Each of them can be stored in $O(n\log_2 n)$ blocks and answers a query using $O(\log_B n + t)$ I/Os. We will describe the data structures in the dual setting, *i.e.*, we wish to preprocess a set L of N lines in the plane so that the lines of L lying below a query point can be reported efficiently. In order to present our data structures we need an additional geometric concept.

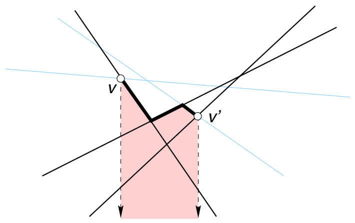

Figure 3. A cluster induced by two vertices of the 2-level

Let $x(v)$ denote the x-coordinate of v. For a point p in the plane, let L_p be the set of lines lying below the point p. For two vertices v and v' on $\mathcal{A}_k(L)$ with $x(v) < x(v')$, we define the *cluster* $C \subseteq L$ induced by v and v' to be the subset $\bigcup_p L_p$, where the union is taken over all points on $\mathcal{A}_k(L)$ between v and v'. In other words, C is the set of lines intersecting the polygon formed by the portion of $\mathcal{A}_k(L)$ between v and v' and the vertical downward rays emanating from v and v'; see Figure 3. We say that C is *relevant* for a point p if $x(v) \le x(p) \le x(v')$. Let $V = \langle v_0, v_1, v_2, \dots, v_u \rangle$ be a subsequence of vertices of $\mathcal{A}_k(L)$, sorted from left to right, where v_0, v_u are the points on $\mathcal{A}_k(L)$ at $x = -\infty, +\infty$, respectively. The *clustering* defined by V is the family $\Gamma = \{C_1, C_2, \dots, C_u\}$, where C_i is the cluster induced by v_{i-1}, v_i. We call $v_0, v_1, \dots, v_k$ the *boundary points* of Γ. Note that a line in L may belong to several clusters C_i. We call Γ a *b-clustering* if every cluster contains b or fewer lines. The *size* of a clustering Γ is just the number of clusters u.

The following lemma is the basis of the data structures described in this section.

Lemma 3.1. *Let L be a set of N lines in the plane. For any $1 \le k < N$, there exists a $3k$-clustering Γ of $\mathcal{A}_k(L)$ of size at most N/k. It can be computed using $O(\nu_k(\log_2 n)\log_B n)$ I/Os, where ν_k is the complexity of $\mathcal{A}_k(L)$.*

Proof: We will use a greedy algorithm to construct Γ. Let $V = \langle v_0, v_1, \dots, v_m \rangle$ be the complete sequence of vertices of $\mathcal{A}_k(L)$, sorted from left to right, where v_0, v_u are the points on $\mathcal{A}_k(L)$ at $x = -\infty, +\infty$. Suppose we have already computed $C_1, \dots, C_{i-1}$ and that C_{i-1} is induced by the vertices v_a and v_b of V. To construct C_i, we simply scan through V starting from v_b. Suppose we are currently scanning a vertex $v_c \in V$. C_i consists of lines that lie below $\mathcal{A}_k(L)$ between v_b and v_c. We process c as follows. If $c = m$, C_i is defined by v_b and v_m, and we are done. Otherwise, let $\ell \in L$ be the line that appears on $\mathcal{A}_k(L)$ immediately after v_c. We check whether ℓ is in C_i. If so, there is nothing to do. If $\ell \notin C_i$ and C_i already has $3k$ lines, we set v_c to be the right boundary point of C_i, start the new cluster C_{i+1} (set v_c to be the left boundary point of C_{i+1}), and add ℓ to C_{i+1}. Finally, if C_i has less then $3k$ lines, we add ℓ to C_i.

As mentioned above, we can construct $\mathcal{A}_k(L)$ using $O(\nu_k(\log_2 n)\log_B n)$ I/Os. In order to check $\ell \in C_i$ using one I/O, we maintain a bit $b(\ell)$ for each line $\ell \in L$; $b(\ell)$ is 1 if $\ell \in C_i$ and 0 otherwise. After we found the right endpoint of C_i, we reset the bits of all lines in C_i to 0 using at most $|C_i|$ I/Os. Thus the scan can be performed using $O(\nu_k)$ I/Os.

To finish the proof, it suffices to show that there are at least k lines in each cluster $C_i \in \Gamma$ that do not belong to any other cluster C_j with $j > i$. Fix a cluster C_i induced by v_a and v_b. For each line $\ell \in C_i$, define its *exit point* a_ℓ to be the rightmost point of ℓ between v_a and v_b whose level is at most k. If ℓ lies below v_b, then a_ℓ lies on the vertical line passing through v_b, otherwise a_ℓ is a vertex of $\mathcal{A}_k(H)$. Let $h \in C_i$ be a line so that the exit points of at least $2k$ lines in C_i lie to the right of a_h. There are k such lines. We claim that h lies above $\mathcal{A}_k(L)$ to the right of a_h.

Let $g \in C_i$ be another line whose exit point lies to the right of a_h; see Figure 4. If a_h lies below g, then g intersects h to the left of a_g and lies below h to the right of their intersection point (because $a_g \in \mathcal{A}_k(L)$ and h lies above $\mathcal{A}_k(L)$ at $x = x(a_g)$). Since only k lines of L lie below a_h and at least $2k$ exit points lie to the right of a_h, at least k lines of C_i lie below h to the right of v_b.

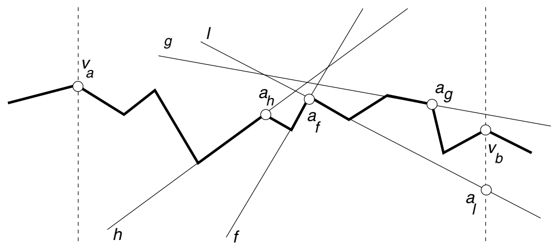

Figure 4. Proof of Lemma 3.1.

Let ℓ be the line that appears on $\mathcal{A}_k(L)$ immediately after v_b. By construction, $\ell \notin C_i$, and ℓ also intersects h to the left of v_b and lies below h after the intersection point. This implies that at least $k+1$ lines lie below h, and therefore h lies above $\mathcal{A}_k(L)$ to the right of v_b, as claimed. □

In the following we present two data structures based on Lemma Lemma 3.1. In Section 6 we will show that the technique used in the second of our structures extends to $\mathbb{R}^3$.

3.1 The layered structure

Let $\beta = B \log_B n$ and $m = \lfloor \log_2(N/\beta) \rfloor$. For all $0 \leq i \leq m-1$, choose a random integer λ_i between $\beta 2^i$ and $\beta 2^{i+1} - 1$, let $\lambda_m = N$, and compute $\mathcal{A}_{\lambda_i}(L)$. Intuitively, the plane is partitioned by $\mathcal{A}_{\lambda_0}(L), \ldots, \mathcal{A}_{\lambda_m}(L)$ into a series of *layers* of exponentially increasing size; see Figure 5. Our approach is first to find the layer containing the query point and then to report all lines of L lying below the query point. Similar ideas have been used previously to build range searching data structures in internal memory [4, 13] and external memory [46]. How exactly we do this efficiently is described next.

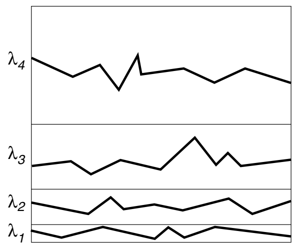

Figure 5. Layers of levels

For each i, we compute a $3\lambda_i$-clustering Γ_i of $\mathcal{A}_{\lambda_i}(L)$ of size at most N/λ_i, as described by Lemma 3.1. By Lemma 2.2, the expected number of vertices in $\mathcal{A}_{\lambda_i}(L)$ is $O(N)$, so Γ_i can be computed using $O(N(\log_2 n)\log_B n)$ expected number of I/Os. Since we can store each of the N/λ_i clusters of Γ_i in $\lceil 3\lambda_i/B \rceil$ blocks, we use $O(n)$ blocks to store Γ_i. We also build and store a B-ary tree (namely, a B^+-tree) on the x-coordinates of the boundary points of Γ_i. The space needed for this tree is $O(N/(\lambda_i B))$ blocks, so the total space needed to store a layer is $O(n)$. Thus the total space used on all layers is $O(n \log_2 n)$. The expected number of I/Os required to construct the data structure is $O(N(\log_2^2 n)\log_B n)$. In the worst case, since $\mathcal{A}_{\lambda_i}(L)$ has $O(N\lambda_i^{1/3})$ vertices, Γ_i can be computed using $O(N\lambda_i^{1/3}(\log_2 n)\log_B n)$ I/Os. Hence, the number of I/Os required to construct all layers is $O(N^{4/3}(\log_2 n)\log_B n)$ in the worst case.

Let p be a query point. To report all the lines below p, we visit the layers in increasing order $i = 0, 1, 2, \ldots$, until we find a cluster that contains all the lines of L that lie below p. For each layer i, we determine which cluster $C_i \in \Gamma_i$ is relevant for p; this can be accomplished in $O(\log_B n)$ I/Os using the B-ary tree on the boundary points of Γ_i. Next we count the lines in C_i that lie below p. If there are fewer than λ_i such lines, we report them and halt; since p lies below $A_{\lambda_i}(L)$ in this case, every line in L that lies below p is actually reported. Otherwise, we repeat the same step for the next layer.

The above query procedure uses $O(\log_B n + \lambda_i/B) = O(2^i \log_B n)$ I/Os at the ith layer. Suppose we visit layers 0 through μ. Then the total number of I/Os used to answer the query is $O(2^\mu \log_B n)$. If $\mu = 0$ the query takes $O(\log_B n)$ I/Os. If $\mu > 0$, the query point must lie above at least $\lambda_{\mu-1}$ lines, since otherwise the algorithm would have stopped at layer $\mu - 1$, and hence the output size $T \geq \lambda_{\mu-1} \geq 2^{\mu-1} B \log_B n$. Therefore the number of I/Os required in this case is $O(T/B) = O(t)$.

Theorem 3.2. *Let S be a set of N points in the plane. We can store S in a data structure that uses $O(n \log_2 n)$ blocks so that a halfspace range query can be answered in $O(\log_B n + t)$ I/Os. The expected number of I/Os used to construct the data structure is $O(N(\log_2^2 n) \times \log_B n)$.*

3.2 The binary tree structure

Our second data structure is constructed recursively as follows. If the number of lines in L is less than $\beta = B \log_B n$, we simply store the lines. Otherwise, we choose a random integer λ between β and 2β and construct a 3λ-clustering Γ of the level $A_\lambda(L)$. We also build a B-ary tree on the boundary points of Γ, as above. We then partition L into two equal-sized subsets L_1 and L_2, using a method to be described shortly, and recursively build data structures for those subsets. The overall structure is a balanced binary tree with depth $\log_2 n$, where any node at depth i is associated with a subset of $N/2^i$ lines. We use $O(n/2^i)$ blocks to store the clustering in a node at depth i in this tree, or $O(n)$ blocks for clusterings in all nodes at depth i. Thus, the overall disk space used by the data structure is $O(n \log_2 n)$.

We say that the partition $L = L_1 \cup L_2$ is *balanced* if, for any point $p \in \mathcal{A}_\lambda(L)$, the sets $L_p \cap L_1$ and $L_p \cap L_2$

each contain at least $\lceil \lambda/4 \rceil$ lines. (Recall that L_p denotes the set of λ lines that lie below p.) This condition is necessary to guarantee optimal query time. To obtain a balanced partition, we compute a random permutation $\langle \ell_{\pi(1)}, \ell_{\pi(2)}, \ldots, \ell_{\pi(N)} \rangle$ of the lines in L and split it down the middle, setting $L_1 = \{\ell_{\pi(1)}, \ldots, \ell_{\pi(\lfloor N/2 \rfloor)}\}$ and $L_2 = \{\ell_{\pi(\lfloor N/2 \rfloor+1)}, \ldots, \ell_{\pi(N)}\}$, and test whether the resulting partition is balanced. The test can be performed in time proportional to the complexity of $\mathcal{A}_\lambda(L)$ by scanning through the vertices of the level, keeping a count of how many lines of L_1 and L_2 lie below the level. Since $\lambda = \Omega(\ln N)$, Chernoff's bound [35, page 86] implies that this partition is balanced with high probability (that is, with probability $O(1 - n^{-c})$ for some constant $c > 0$). In the unlikely event that the partition is unbalanced, we try again with a new random partition; the expected number of iterations before we obtain a balanced partition is less than 2. (See the remark below.) Thus the expected number of I/Os used on each level of the tree is $O(N(\log_2 n)\log_B n)$, resulting in an total expected construction time of $O(N(\log_2^2 n)\log_B n)$ I/Os.

Given a query point p, we use our data structure to find the lines in L that lie below p as follows: We first perform a B-ary search to find the cluster $C \in \Gamma$ that is relevant for p, using $O(\log_B n)$ I/Os. We then count the lines in C that lie below p, using $O(\lambda/B) = O(\log_B n)$ I/Os. If the number of lines below p is less than λ, we report them and halt. Since p lies below $A_\lambda(L)$ in this case, every line in L that lies below p is reported. Otherwise, we recursively query both L_1 and L_2. At the leaves of the binary tree, we simply scan through the list of at most β lines, using $\beta/B = O(\log_B n)$ I/Os.

Suppose the query point p is above at least λ lines in its relevant cluster $C \in \Gamma$. Then it also lies above $A_\lambda(L)$, and since the partition $L_1 \cup L_2$ is balanced, it must lie above at least $\lambda/4 \geq \beta/4$ lines in both L_1 and L_2. Thus, whenever the query algorithm reaches a node v in the binary tree, except possibly when v is the root, the query point lies above at least $\beta/4$ of the lines associated with v. It follows that the query algorithm visits at most $O(T/\beta)$ nodes besides the root. Since $O(\log_B n)$ I/Os are performed at every visited node, we conclude that the total number of I/Os needed to answer a query is $O((T/\beta + 1)\log_B n) = O(\log_B n + t)$. Thus, our binary tree data structure also satisfies Theorem 3.2.

Remark. In practice, when we build the data structure it suffices to take the first random partition, without testing whether it is balanced. The number of I/Os required to answer a query using the resulting data structure is still $O(\log_B n + t)$ with high probability. With this modification, the number of I/Os needed to construct the data structure is $O(N\beta^{1/3}(\log_2 n)\log_B n) = O(NB^{1/3}(\log_2 n)\log_B^2 n)$ in the worst case, an improvement of $O(n^{1/3}/\log_B n)$ over the layered structure.

4 A Linear-Size Data Structure in 2D

In this section we present a halfspace range-searching data structure that uses only $O(n)$ disk blocks. We will describe the data structure in the primal setting. Let S be a set of N points in $\mathbb{R}^2$. A *simplicial partition* of S is a set of pairs $\Pi = \{(S_1, \Delta_1), (S_2, \Delta_2), \ldots, (S_r, \Delta_r)\}$, where S_i's are disjoint subsets of S, and each Δ_i is a triangle containing the points in S_i. Note that a point of S may lie in many triangles, but it belongs to only one S_i; see Figure 6. The size of Π, here denoted r, is the number of pairs. A simplicial partition is *balanced* if each subset S_i contains between N/r and $2N/r$ points.

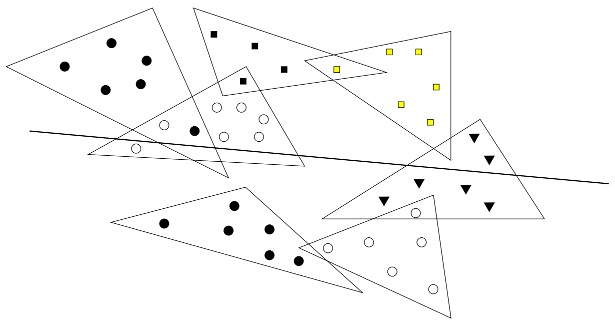
Figure 6. A balanced simplicial partition of size 7.

Theorem 4.1 (Matoušek [32]). *Let S be a set of N points in the plane, and let $1 < r \leq N/2$ be a given parameter. For some constant α (independent of r), there exists a balanced simplicial partition Π of size r, so that any line crosses at most $\alpha\sqrt{r}$ triangles of Π.*

We use this theorem to build a range-searching data structure for S called a *partition tree* T. Partition trees are one of the most commonly used internal memory data structures for geometric range searching [3, 26, 32, 47]; our construction closely follows the one by Matoušek [32]. Each node v in a partition tree T is associated with a subset $S_v \subseteq S$ of points and a triangle Δ_v. For the root u of T, we have $S_u = S$ and $\Delta_u = \mathbb{R}^2$. Let $N_v = |S_v|$ and $n_v = \lceil N_v/B \rceil$. Given a node v, we construct the subtree rooted at v as follows. If $N_v \leq B$, then v is a leaf and we store all points of S_v in a single block. Otherwise, v is an internal node of degree r_v, where $r_v = \min\{cB, 2n_v\}$, for some constant $c \geq 1$ to be specified later. We compute a balanced simplicial partition $\Pi_v = \{(S_1, \Delta_1), \ldots, (S_{r_v}, \Delta_{r_v})\}$ for S_v, as described in Theorem 4.1, and then recursively construct a partition tree T_i for each subset S_i. For each i, we store the vertices of Δ_i and a pointer to T_i; the root of T_i is the ith child of v, and it is associated with S_i and Δ_i. We need $O(c) = O(1)$ blocks to store any node v. Since r_v was chosen to be $\min\{cB, 2n_v\}$, every leaf node contains $\Theta(B)$ lines. Thus the total number of nodes in the tree is $O(n)$, so the total size of the partition tree is $O(n)$.

To find all points below a query line ℓ, we visit T in a top down fashion. Suppose we are at a node v. If v is a leaf, we report all points of S_v that lie below ℓ. Otherwise we test each triangle Δ_i of Π_v. If Δ_i lies above ℓ, we ignore Δ_i; if Δ_i lies below ℓ, we report all points in S_i by traversing the ith subtree of v; finally, if ℓ crosses Δ_i, we recursively visit the ith child of v. Note that each point is reported only once.

To bound the number of I/Os used to perform a query, first note that if Δ_v lies below ℓ, then all points of S_v lie below ℓ and we spend $O(n_v)$ I/Os to report these points. Since each point is reported only once, $\sum_v n_v \leq t$, where the sum is taken over all nodes v visited by the query procedure for which Δ_v lies below ℓ. Suppose μ is the number of nodes v visited by the query procedure for which Δ_v intersects ℓ, or in other words, the number of recursive calls to the query procedure. Since we require $O(1)$ I/Os at each such node, the overall query procedure requires $O(\mu + t)$ I/Os.

What remains is to bound the value of μ. Let $\Sigma(N_v)$ be the maximum number of descendants of v (including v) that are recursively visited by the query procedure. If v is a leaf, only v itself is visited. If v is an interior node, then by Theorem 4.1, we recursively visit at most $\alpha\sqrt{r_v}$ children of v. Since each child of v is associated with at most $2N_v/r$ points, we obtain the following recurrence:

$$\Sigma(N_v) \leq \begin{cases} 1 + \alpha\sqrt{r_v}\,\Sigma(2N_v/r_v) & \text{if } N_v > B, \\ 1 & \text{if } N_v \leq B. \end{cases}$$

By choosing $c = c(\varepsilon)$ sufficiently large in the definition of r_v, we can prove by induction on N_v that $\mu = \Sigma(N) = O(n^{1/2+\varepsilon})$ for any constant $\varepsilon > 0$. The constant of proportionality depends on ε [32].

Theorem 4.2. *Given a set S of N points in the plane and a parameter $\varepsilon > 0$, we can preprocess S into a data structure of size $O(n)$ blocks so that a halfspace range query can be answered using $O(n^{1/2+\varepsilon} + t)$ I/Os.*

Remarks.

(i) In practice, the query bound will be better than $O(n^{1/2+\varepsilon} + t)$ I/Os because a query line will not cross $\alpha\sqrt{r}$ triangles at all nodes visited by the query procedure.

(ii) The same data structure can also be used to report points lying inside a query *polygon* with m edges in $O(mn^{1/2+\varepsilon} + t)$ I/Os.

(iii) Theorem 4.1 can be extended to higher dimensions. That is, for any set S of n points in $\mathbb{R}^d$ for any r, we can construct a balanced partition Π of size r so that any hyperplane intersects $O(r^{1-1/d})$ simplices of Π [32]. Thus our data structure can be modified to work in higher dimensions. For any fixed d, a set of N points in $\mathbb{R}^d$ can be preprocessed into a data structure of size $O(n)$ so that a halfspace range query can be answered using $O(n^{1-1/d+\varepsilon} + t)$ I/Os.

(iv) In our algorithm, we used brute force to determine which triangles at each node cross the query line ℓ. If we use a more sophisticated procedure to determine these triangles, we can choose $r_v = N_v^\beta$ for some constant $\beta < 1/2$. The query bound improves to $O(\sqrt{n}\log^{O(1)} n + t)$, and the disk space used by the data structure remains $O(n)$.

5 Trading Space for Query Time in 2D

We can combine Theorem 4.2 with Theorem 3.2 in order to improve the query time at the expense of space. The idea is to use the same recursive procedure to construct a tree as in the previous section, but stop the recursion when $N_v \leq B^a$ for some constant $a > 1$. In that case we preprocess S_v into a data structure of size $O(n_v \log_2 n_v) = O(aB^a \log_2 B)$ using Theorem 3.2. The total size of the data structure is $O(an \log_2 B)$. A query is answered as in the previous section except that when we reach a leaf of the tree, we use the query procedure described in Section 3. The query procedure now visits $O((n/B^{a-1})^{1/2+\varepsilon})$ nodes of the tree.

Theorem 5.1. *Given a set S of N points in the plane and constants $\varepsilon > 0$ and $a > 1$, we can preprocess S into a data structure of size $O(n \log_2 B)$ blocks so that a halfspace range query can be answered using $O((n/B^{a-1})^{1/2+\varepsilon} + t)$ I/Os.*

If we allow $O(n \log_B n)$ space, the query time can be improved considerably, at least theoretically. In this abstract we only outline the basic approach. A line ℓ is called *k-shallow* with respect to S, for $k < N$, if at most k points of S lie below ℓ.

Theorem 5.2 (Matoušek [33]). *Let S be a set of n points in the plane, and let $1 < r \leq n/2$ be a given parameter. For some constant $\beta > 1$ (independent of r), there exists a balanced simplicial partition Π of S so that any (N/r)-shallow line crosses at most $\beta \log_2 r$ triangles of Π.*

Using this theorem we construct a so-called *shallow partition tree* Ψ essentially as in the previous section (with the same values of r), except that at each node v of Ψ, we also construct a (non-shallow) partition tree T_v on S_v and store it as a secondary structure of v. Since we need $O(n_v)$ blocks to store T_v, the total size of the shallow partition tree is $O(n \log_B n)$.

A query is answered by traversing Ψ in a top down fashion. Suppose we are at an interior node v. As previously, we check which of the triangles of Π_v cross the query line ℓ. If more than $\beta \log_2 r$ triangles cross ℓ, we conclude that ℓ is not (N_v/r)-shallow with respect to S_v, and we use the secondary structure T_v to report all points of S_v lying below ℓ using $O(n_v^{1/2+\varepsilon} + t_v)$ I/Os, where t_v is the number of points of S_v lying below ℓ. Since $t_v \geq N_v/r \geq n_v/c$, we have that $O(n_v^{1/2+\varepsilon} + t_v) = O(t_v)$. Otherwise, if at most $\beta \log_2 r$ triangles of Π_v cross ℓ, we report all points in triangles in Π_v that lie below ℓ, by traversing the corresponding subtrees, and recursively visit all triangles in Π_v that cross ℓ.

We say that a node v visited by the query procedure is *shallow* if ℓ is (N_v/r)-shallow with respect to S_v. If the query visits μ shallow nodes, then the query procedure requires $O(\mu + t)$ I/Os. Let $\Sigma(N_v)$ be the maximum number of shallow descendants of a shallow node v (including v). Then, as in the previous section, we obtain the recurrence

$$\Sigma(N_v) \leq \begin{cases} 1 + \beta \log_2 r \Sigma(2N_v/r) & \text{if } N_v > B, \\ 1 & \text{if } N_v \leq B. \end{cases}$$

Solving this recurrence gives us the bound $\mu = \Sigma(N) = O(n^\varepsilon)$ for any constant $\varepsilon > 0$, provided we choose $c = c(\varepsilon)$ sufficiently large.

Theorem 5.3. *Given a set S of N points in the plane and a constant $\varepsilon > 0$, we can preprocess S into a data structure of size $O(n \log_B n)$ blocks so that a halfspace range query can be answered using $O(n^\varepsilon + t)$ I/Os.*

Like our earlier partition tree data structure, shallow partition trees can be generalized to higher dimensions. For any $d > 2$, we obtain a data structure requiring $O(n \log_B n)$ blocks that can answer halfspace queries in $O(n^{1-1/\lfloor d/2 \rfloor + \varepsilon} + t)$ I/Os. The corresponding internal-memory data structure is described by Matoušek [33].

6 Halfspace Range Searching in 3D

In this section we sketch our three-dimensional data structure. Details will appear in the full version of this paper. Our structure closely follows the binary tree structure described in Section 3.2. As in that section, we solve the problem in the dual; given a set P of n planes, we build a data structure that lets us quickly compute all the planes lying below an arbitrary query point.

We first need to describe a slightly weaker three-dimensional analogue of Lemma 3.1. Let P be a set of N planes, and let k be an integer between 1 and N. The level $\mathcal{A}_k(P)$ is a monotone piecewise-linear surface, or a polyhedral terrain, with several convex polygonal facets. Let ν_k denote the number of facets of $\mathcal{A}_k(P)$, and for each facet ϕ, let P_ϕ be the set of k planes that lie below (any point in) ϕ. A *clustering* of $\mathcal{A}_k(P)$ consists of a partition of the facets into $O(\nu_k/k)$ families, such that for each family Φ, its associated *cluster* $\bigcup_{\phi \in \Phi} P_\phi$ contains $O(k)$ planes. The entire clustering can be stored using $O(\nu_k/B)$ blocks.

Lemma 6.1 (Agarwal *et al.* [2]). *Let P be a set of N planes. For any $1 \leq k \leq N$, a clustering of $\mathcal{A}_k(P)$ can be constructed using $O(Nk^{5/3})$ expected I/Os.*

We construct our data structure for P as follows. Let $\beta = cB \log_B n$ for some constant $c > 1$. If the number of planes in P is less than β, we simply store the planes in $O(\log_B n)$ blocks. Otherwise, we choose a random integer λ between β and 2β and build a clustering Γ of level $A_\lambda(P)$, as described above. By Lemma 2.2, the expected complexity of $\mathcal{A}_\lambda(P)$ is $O(N\beta)$, and we can store Γ in $O(n\beta)$ expected number of blocks. (If we are unlucky and the clustering is too large, then we start over with another random integer λ.) We also construct an external point-location data structure for the planar map obtained by projecting $A_\lambda(P)$ vertically onto the xy-plane; this will allow us to compute the facet of $A_\lambda(P)$ directly above or below a query point in $O(\log_B n)$ time [23]. We then partition the planes in P into two subsets P_1 and P_2 of equal size, by repeatedly choosing a random permutation of P and splitting the resulting sequence in the middle, until the resulting partition is balanced. (As in the two-dimensional case, a random partition is balanced with high probability.) Finally, we recursively construct a data structure for the subsets P_1 and P_2. The overall data structure is a binary tree requiring $O(n\beta \log_2 n) = O(N(\log_2 n) \log_B n)$ blocks of storage.

A search in this data structure is similar to the two-dimensional case. First we perform a point-location query to determine the facet ϕ of $A_\lambda(P)$ directly above or below the query point p. If p lies above ϕ, we recursively query the two children. Otherwise, we find the the cluster C associated with the family of facets containing ϕ and report all the planes in C that lie below p. At the leaves, we simply scan through the β or fewer planes. We spend $O(\log_B n)$ I/Os at each visited node. By the same analysis as in Section 3.2, the total number of I/Os required to answer a query is $O(\log_B n + t)$.

Theorem 6.2. *Let S be a set of N points in $\mathbb{R}^3$. We can preprocess S into a data structure using $O(N \times (\log_2 n) \log_B n)$ blocks, so that a halfspace range query can be answered in $O(\log_B n + t)$ I/Os. The expected number of I/Os used to construct the data structure is $O(NB^{5/3}(\log_2 n) \log_B^{5/3} n)$.*

References

[1] P. K. Agarwal, B. Aronov, T. Chan, and M. Sharir. On levels in arrangements of lines, segments, planes, and triangles. *Discrete Comput. Geom.*, 1998, in press.

[2] P. K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf. Constructing levels in arrangements and higher order Voronoi diagrams. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 67–75, 1994.

[3] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Discrete and Computational Geometry: Ten Years Later*, 1998, to appear.

[4] P. K. Agarwal, M. van Kreveld, and M. Overmars. Intersection queries in curved objects. *J. Algorithms*, 15:229–266, 1993.

[5] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 560–569, 1996.

[6] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 322–331, 1990.

[8] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. ACM Sympos. Principles of Database Systems*, pages 78–86, 1997.

[9] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for higher dimensional data. In *Proc. 22th International Conference on Very Large Databases*, pages 28–39, 1996.

[10] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comput.*, 15:703–724, 1986.

[11] B. Chazelle, R. Cole, F. P. Preparata, and C. K. Yap. New upper bounds for neighbor searching. *Inform. Control*, 68:105–124, 1986.

[12] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[13] B. Chazelle and F. P. Preparata. Halfspace range search: An algorithmic application of k-sets. *Discrete Comput. Geom.*, 1:83–93, 1986.

[14] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[15] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11:121–137, 1979.

[16] T. Dey. Improved bounds for k-sets and k-th levels. In *Proc. 38th Annu. IEEE Sympos. Found. Comput. Sci.*, pages, 156–161, 1997.

[17] F. Dumortier, M. Gyssens, and L. Vandeurzen. On the decidability of semi-linearity for semi-algebraic sets and its implications for spatial databases. In *Proc. ACM Symp. Principles of Database Systems*, pages 68–77, 1997.

[18] H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM J. Comput.*, 15:271–284, 1986.

[19] G. Evangelidis, D. Lomet, and B. Salzberg. The hB^{Π}-tree: a multi-attribute index supporting concurrency, recovery and node consolidation. *The VLDB Journal*, 6:1–25, 1997.

[20] P. Franciosa and M. Talamo. Time optimal halfplane search on external memory. Unpublished manuscript, 1997.

[21] P. G. Franciosa and M. Talamo. Orders, k-sets and fast halfplane search on paged memory. In *Proc. Workshop on Orders, Algorithms and Applications*, volume 831 of *Lecture Notes Comput. Sci.*, pages 117–127. Springer-Verlag, 1994.

[22] J. Goldstein, R. Ramakrishnan, U. Shaft, and J.-B. Yu. Processing queries by linear constraints. In *Proc. ACM Symp. Principles of Database Systems*, pages 257–267, 1997.

[23] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.

[24] R. Güting. An introduction to spatial database systems. *VLDB Journal*, 4:357–399, 1994.

[25] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 47–57, 1985.

[26] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete Comput. Geom.*, 2:127–151, 1987.

[27] A. Henrich. Improving the performance of multi-dimensional access structures based on *kd*-trees. In *Proc. 12th IEEE Intl. Conf. on Data Engineering*, pages 68–74, 1996.

[28] E. G. Hoel and H. Samet. A qualitative comparison study of data structures for large line segment databases. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 205–214, 1992.

[29] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings 20th International Conference on Very Large Databases*, pages 500–509, 1994.

[30] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences*, 52:589–612, 1996.

[31] D. Lomet and B. Salzberg. The hB-tree: A multi-attribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, 1990.

[32] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.

[33] J. Matoušek. Reporting points in halfspaces. *Comput. Geom. Theory Appl.*, 2(3):169–186, 1992.

[34] J. Matoušek. Geometric range searching. *ACM Comput. Surv.*, 26:421–461, 1994.

[35] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, New York, NY, 1995.

[36] J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9:257–276, 1984.

[37] J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*. Springer-Verlag, 1997. Lecture Notes in Computer Science Vol. 1340.

[38] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane, 23:166–204, 1981.

[39] S. Ramaswamy and P. Kanellakis. OOBD indexing by class division. In *A.P.I.C. Series, Academic Press, New York*, 1995.

[40] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. Principles of Database Systems*, pages 25–35, 1994.

[41] J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 10–18, 1984.

[42] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison Wesley, MA, 1989.

[43] H. Samet. *The Design and Analyses of Spatial Data Structures*. Addison Wesley, MA, 1989.

[44] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-tree: A dynamic index for multi-dimensional objects. In *Proc. IEEE International Conf. on Very Large Databases*, 1987.

[45] S. Subramanian and S. Ramaswamy. The p-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995.

[46] D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proc. of the 28th Annual ACM Symposium on Theory of Computing*, pages 192–201, 1996.

[47] D. E. Willard. Polygon retrieval. *SIAM J. Comput.*, 11:149–165, 1982.