Box-Trees and R-trees with Near-Optimal Query Time*

P. K. Agarwal†, M. de Berg‡, J. Gudmundsson‡, M. Hammar§, H. J. Haverkort‡

† Dept. of Computer Science, Box 90129, Duke University, Durham, NC 27708-0129, USA, pankaj@cs.duke.edu

‡ Institute of Information and Computing Sciences, Utrecht University, PO Box 80.089, 3508 TB Utrecht, {herman,joachim,markdb}@cs.uu.nl

§ Dept. of Computer Science, Lund University, Box 118, 221 00 Lund, Sweden, mikael@cs.lth.se

Keywords: bounding-volume hierarchy, R-tree, cs-box-tree, kd-interval-tree, window query, rectangle-intersection query

Abstract

A box-tree is a bounding-volume hierarchy that uses axis-aligned boxes as bounding volumes. The query complexity of a box-tree with respect to a given type of query is the maximum number of nodes visited when answering such a query. We describe several new algorithms for constructing box-trees with small worst-case query complexity with respect to queries with axis-parallel boxes and with points. We also prove lower bounds on the worst-case query complexity for box-trees, which show that our results are optimal or close to optimal. Finally, we present algorithms to convert box-trees to R-trees, resulting in R-trees with (almost) optimal query complexity.

1 Introduction

Motivation and problem statement. *Window queries* report all objects of a given set that intersect a query d-rectangle, that is, a d-dimensional box. Preprocessing a set S of geometric objects in $\mathbb{R}^d$ for answering such queries is central to many applications and has been widely studied in several areas, including computational geometry, computer graphics, spatial databases, GIS, and robotics [7, 17]. In order to expedite and simplify the data structure, a window query is often answered in two steps. In the first step, called the *filtering* step, each object is replaced by the smallest box containing the object and the query procedure reports the bounding boxes that intersect the query window. (Instead of boxes, other simple shapes such as spheres, ellipsoids, cylinders have also been used.) The second step, called the *refinement step*, extracts the actual objects among these bounding boxes that intersect the query window [4, 19]. A few recent results show that under certain reasonable assumptions on the input objects, the number

*The work by P.A. is supported by Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, by NSF grants ITR–333–1050, EIA–9870724, EIA–997287, and CCR–9732787, and by a grant from the U.S.-Israeli Binational Science Foundation. The work by H.H. is supported by the Netherlands' Organization for Scientific Research (NWO).

of bounding boxes intersecting a query window is not much larger than the number of objects intersecting the window, which makes this approach quite attractive; see the paper by Zhou and Suri [21] and the references therein. There has been much work on the filtering step, and we also focus on this step. More precisely, we wish to preprocess a set S of n d-rectangles in $\mathbb{R}^d$ so that all rectangles of S intersecting a query d-rectangle can be reported efficiently. We will refer to this query as the *rectangle-intersection query*. A related query is the *rectangle-containment* query in which we want to report all rectangles in S that contain a query point.

A number of data structures with good provable bounds for answering rectangle-intersection queries have been proposed. Unfortunately they are of limited practical use because the amount of storage used is rather high: $O(n \log n)$ storage and even $O(n)$ storage with a large hidden constant are often unacceptable. Therefore in practice one usually uses simpler data structures. A commonly used structure for answering rectangle-intersection queries, rectangle-containment queries, and in fact many other types of queries is the *bounding-box hierarchy*, or *box-tree* for short, sometimes also called AABB-tree: this is a tree $\mathcal{T}$, in which each leaf is associated with a rectangle of the input set S, and each interior node ν is associated with the smallest box B_ν enclosing all the rectangles stored at the leaves of the subtree rooted at ν. All the rectangles of S intersecting a query rectangle R are reported by traversing $\mathcal{T}$ in a top-down manner. Suppose the query procedure is visiting a node ν. If $B_\nu \cap R = \emptyset$, there is nothing to do. If $B_\nu \subseteq R$, then it reports all rectangles stored in the subtree rooted at ν. Finally, if $B_\nu \cap R \neq \emptyset$ but $B_\nu \not\subseteq R$, it recursively visits the children of ν. We say that R *crosses* a node ν if $B_\nu \cap R \neq \emptyset$ and $B_\nu \not\subseteq R$. If the fan-out of $\mathcal{T}$ is bounded, then the query time is proportional to the number of nodes of $\mathcal{T}$ that R crosses plus the number of rectangles reported. We define the stabbing number of $\mathcal{T}$ to be the maximum number of its nodes crossed by a rectangle. It is therefore desirable to construct a bounding-box hierarchy with small stabbing number.

In many applications, especially in the database applications, the set S is too large to fit in the main memory, therefore it is stored on disk. In that case, the main goal is to minimize the number of disk accesses needed to answer a window query, and the performance of an algorithm is analyzed under the standard external memory model [2]. This model assumes that each disk access transmits a contiguous block of t units of data in a single *input/output operation* (or *I/O*). The efficiency of a data structure is measured in terms of the amount of disk space it uses (measured in units of disk blocks), the number of I/Os required to answer a query, and the number of I/Os needed to construct the data structure. In the context of bounding-box hierarchies, several schemes have been proposed that construct a tree as above but in which the fanout of each node depends on t. Some notable examples of external-memory bounding-box hierarchies are various variants of R-trees; see the survey paper [11]. We can still define the *crossing* nodes and the *stabbing number* as earlier, and one can argue that the number of I/Os needed to answer a query is proportional to the stabbing number plus the output size.

In this paper we study the problem of constructing bounding-box hierarchies, both in main and external memory, that have low stabbing number, and consequently, low query complexity.

Previous results. As noted above, several efficient data structures have been proposed for answering a rectangle-intersection query. For example, Chazelle [5] showed that a compressed range tree can be used to answer a d-dimensional rectangle-intersection query in time $O(\log^{d-1} n + k)$ using $O(n \log^{d-1} n / \log\log n)$ space (where k is the number of rectangles reported). This data structure is too complex to be practical even in $\mathbb{R}^2$. As for bounding volume hiearchies, we know of only one result on the query complexity of rectangle-intersection queries (besides the results on R-trees discussed later): if one maps each d-rectangle to a point in $\mathbb{R}^{2d}$,

constructs a kd-tree on these points, and converts the kd-tree back to a box-tree, then the query time is known to be $O(n^{1-1/2d} + k)$ [1, 15]. A number of heuristics based on kd-trees have also been proposed to answer rectangle-intersection queries [1, 18]. Several papers [12, 14] describe how to construct bounding-box hierarchies or other bounding-volume hierarchies (for example, using k-DOPs as bounding volumes), but they do not obtain bounds on the worst-case query complexity.[1]

Some of the most widely used external-memory bounding-box hierarchies are the R-tree and its variants. An R-tree, originally introduced by Guttmann [13], is a B-tree, each of whose leaves is associated with an input rectangle. All leaves of an R-tree are at the same level, the degree of all internal nodes except of the root is between t and $2t$, for a given parameter t, and the degree of the root varies between 2 and $2t$. We will refer to t as the *minimum degree* of the tree. To minimize the query complexity, several methods have been proposed [9, 10, 11, 16] for ordering the input rectangles along the leaves—varying from simple heuristics to space filling curves—but none of them guarantee the worst-case performance. In the worst case, a linear number of bounding boxes might intersect a query rectangle even if it intersects only $O(1)$ input rectangles. The only analytical results are by Theodoridis and Sellis [20], who present a model that predicts the average performance of R-trees for range queries, and Faloutos *et al.* [10], but they prove bounds on the query time only in the 1-dimensional case when the input intervals are uniformly distributed and have at most two different lengths. Recently, de Berg *et al.* [6] described an algorithm for constructing an R-tree on rectangles in $\mathbb{R}^2$ so that all k rectangles containing a query point can be reported in $O((\sigma + \log \rho) \log n / \log t)$ I/Os. Here ρ is the ratio of the maximum and the minimum x-lengths of the input rectangles, and σ is the *point-stabbing number* of S, that is, σ is the maximum number of input rectangles containing any point in the plane. For a rectangle-intersection query, the number of I/Os is $O((\sigma + \log \rho + w + k) \log n / \log t)$, where w is the ratio of the x-length of the query rectangle to the smallest x-length of an input rectangle.

Our results. In this paper we first describe several algorithms for constructing box-trees, and we prove lower bounds on the worst-case query complexity of box-trees. The lower bounds actually hold for all bounding volume hierarchies that use convex shapes as bounding volumes.

Our first algorithm, like the approach mentioned earlier, is based on a kd-tree in $\mathbb{R}^{2d}$. By changing the structure slightly and doing a more careful analysis, we are able to obtain $O(n^{1-1/d} + k)$ query complexity for rectangle-intersection queries. We also prove a lower bound showing that this bound is optimal.

For disjoint input in the plane, we show how to construct a box-tree that still has almost optimal query time for rectangle-intersection queries, but much better query times for point queries. In fact, it is already better for point-queries when the point-stabbing number σ of the input is $o(n/\log^4 n)$: the time for rectangle-intersection queries is $O(\sqrt{n} \log n + \sqrt{\sigma} \log^2 n + k)$, and the time for point queries is $O(\sqrt{\sigma} \log^2 n + k)$. We also develop a box-tree with $O((\alpha + \sqrt{\sigma}) \log^2 n + k)$ query time for use with query rectangles with aspect ratio α. One would hope that similar improvements are possible in higher dimensions. One of our lower-bound results shows that this is not possible: in dimensions $d \geq 3$, the $\Omega(n^{1-1/d} + k)$ lower bound on the query complexity holds even for hypercubes as query ranges, and any bounding-box hierarchy

[1] Barequet *et al.* [3] gave an algorithm to construct a bounding-box hierarchy in $\mathbb{R}^2$, and they claimed that if the rectangles in S are pairwise disjoint, then the resulting hierarchy has $O(\log n)$ stabbing number. But the argument presented in the paper has a technical problem.

that achieves this query time cannot have a better worst-case query time for point queries, even when the input consists of disjoint 'almost-unit-hypercubes'.

Finally, we give general methods to convert box-trees with small query complexity into R-trees with small query complexity. When we apply these results to our box-trees, we improve the result of de Berg *et al.* [6]: our query complexity does not depend on the parameter w (which makes their query complexity linear in the worst case), and it is linear in $\sqrt{\sigma}$ instead of in σ. We also introduce the concept of *semi-R-trees*; these are similar to ordinary R-trees—the degree of each internal node, except for the root, is between t and $2t$ for some given parameter t—except that the leaves do not have to be at the same level. We give a general algorithm to convert a box-tree with small query complexity into a semi-R-tree with small query complexity; the bound obtained here is better than that for R-trees. This leads to semi-R-trees with (almost) optimal query complexity.

All box-tree construction algorithms in this paper run in $O(n \log n)$ time, and all box-tree-to-(semi-)R-tree conversion algorithms run in $O(n)$ time.

2 Lower Bounds

In this section we give lower bounds on the query complexity of semi-R-trees of minimum degree t in various settings. Since semi-R-trees are more general than R-trees, the same bounds hold for R-trees. By choosing $t = 2$, we obtain lower bounds for box-trees.

We start with a simple generalization of the 2-dimensional lower bound given by de Berg *et al.* [6].

Theorem 2.1 *For any n and $d \geq 2$, there is a set of n disjoint unit hypercubes in $\mathbb{R}^d$ with the following property: for any semi-R-tree $\mathcal{T}$ of minimum degree t there is a query box not intersecting any box from S such that a query with that box visits $\Omega((n/t)^{1-1/d})$ nodes in $\mathcal{T}$.*

Proof: Consider a set of n unit hypercubes arranged in an $n^{1/d} \times \cdots \times n^{1/d}$ grid, and the following set of query ranges: for each axis, we choose $n^{1/d} - 1$ thin boxes orthogonal to it and separating the 'slices' of the grid from each other. Now any bounding box on t hypercubes intersects at least $d(t^{1/d}-1)$ of the query ranges. Hence, the total number of incidences between the ranges and the bounding boxes is at least $\Omega((n/t) \cdot t^{1/d})$. As there are $O(n^{1/d})$ ranges, there must be one that intersects $\Omega((n/t)^{1-1/d})$ bounding boxes. □

Next we describe a construction that proves lower bounds on rectangle-containment queries and that is also useful for a number of other cases. For any $\varepsilon > 0$, we call a d-rectangle an *ε-hypercube* if the length of each edge is between 1 and $1 + \varepsilon$. We fix a parameter $\mu \geq 1$ and construct a set $S = \{b(0), \ldots, b(n-1)\}$ of n ε-hypercubes in $\mathbb{R}^d$. We also construct two sets of query points Q_1 and Q_2, called *primary* and *secondary* point sets, that lie in the common exterior of the rectangles in S and have the following property: for any semi-R-tree $\mathcal{T}$ on S with minimum degree t, either a point of Q_1 lies in at least μ bounding boxes of $\mathcal{T}$ or a point of Q_2 lies in $\Omega((n/t)/\mu^{1/(d-1)})$ bounding boxes of $\mathcal{T}$. From this we derive the desired lower bounds. We first describe the set S and then construct the point sets.

Let $n_1, \ldots, n_{2d}$ be the outward normals of a d-rectangle. We can pair these normals into d pairs (n_{11}, n_{12}), (n_{21}, n_{22}), …, (n_{d1}, n_{d2}) so that no pair contains opposite normals, that is, $n_{i1} \neq -n_{i2}$ for $1 \leq i \leq d$. Let h_i be the 2-plane spanned by the vectors n_{i1} and n_{i2} and containing the origin. Let b be a d-rectangle containing the origin. Since $n_{i1} \neq -n_{i2}$, the facets f_{i1}, f_{i2} of b normal to n_{i1} and n_{i2}, respectively, share a $(d-2)$-face f_i, which is orthogonal to the 2-plane h_i. The intersection of f_i and h_i is a point c_i. Conversely, by specifying a point c_i on

each h_i, $1 \le i \le d$, we can represent a unique d-rectangle in which c_i lies on the facets normal to n_{i1} and n_{i2}. We will therefore define each rectangle $b(j) \in S$ by a d-tuple $(c_1(j), \ldots, c_d(j))$, where the facets of $b(j)$ whose outward normals are n_{i1} and n_{i2} pass through $c_i(j)$. We next describe how to choose the points $c_i(j)$, for $1 \le i \le d$ and $0 \le j < n$.

On each 2-plane h_i, we choose a line ℓ_i of slope -1; the exact equation of ℓ_i will be specified below. We will refer to h_1 as the *primary* plane, and to h_i, for $i > 1$, as a *secondary plane*. Set $\hat{\mu} = \mu^{1/(d-1)}$. We place n points $p_1(0), \ldots, p_1(n-1)$ on ℓ_1 (sorted along ℓ_1 by ascending n_{i1}-coordinate, and consequently, by descending n_{i2}-coordinate) and set $c_1(j) = p_1(j)$ for every $0 \le j < n$. For each $i > 1$, we place $\hat{\mu}$ points $p_i(0), \ldots, p_i(\hat{\mu} - 1)$ on ℓ_i and assign $c_i(j)$ to these points as follows. Let $\alpha(j) = (\alpha_0(j), \ldots, \alpha_{d-2}(j))$ be the representation of $j \bmod \mu$ in radix $\hat{\mu}$, that is, $\sum_{k=0}^{d-2} \alpha_k(j)\hat{\mu}^k = j \bmod \mu$. For each $i > 1$, we set $c_i(j) = p_i(\alpha_{d-i}(j))$. Note that $n/\hat{\mu}$ points have the same value of $c_i(j)$. We choose ℓ_i and the points on ℓ_i so that each b_j is an ε-hypercube, e.g. by putting all points $p_i(j)$ at a distance of at least $1/2$ and at most $(1+\epsilon)/2$ from the origin, both in their projection on the n_{i1}-axis and on the n_{i2}-axis.

Finally, we choose a set Q_1 of $n-1$ points on the primary plane h_1 and a set Q_2 of $(d-1)(\hat{\mu}-1)$ points on the secondary planes, as follows. Suppose h_1 is the x_1x_2-plane. For each $1 \le j \le n-1$, we choose the point $q(j) = (x_1(p_1(j-1)), x_2(p_1(j)))$ and add it to Q_1. In other words, if we regard the points on ℓ_1 as the convex corners of a staircase, Q_1 is the set of concave corners of the staircase. To construct Q_2, we repeat the same step for each of the secondary planes, thus obtaining $\hat{\mu} - 1$ points on each of them. These points will be on the boundary of some of the input boxes, but we can shift them a little to make them disjoint from all input boxes.

Lemma 2.2 *Let $\mathcal{T}$ be any semi-R-tree of minimum degree t on the set S constructed above. Then either there is a primary query point contained in $\Omega(\mu)$ bounding boxes stored in $\mathcal{T}$, or one of the secondary query points is contained in $\Omega(n/(t\mu^{1/(d-1)}))$ bounding boxes stored in $\mathcal{T}$.*

Proof: We first prove the lemma for box-trees, which are binary trees. Suppose that all primary query points are contained in less than $\mu/2$ bounding boxes stored in the interior nodes in $\mathcal{T}$. Then the number of incidences between these points and interior nodes' bounding boxes is at most $(n-1)\mu/2$. Since there are $n-1$ interior nodes in $\mathcal{T}$, they store at least $(n-1)/2$ bounding boxes that contain less than μ primary query points. Observe that a bounding box for input boxes $b(j), b(j') \in S$ contains $|j - j'|$ primary query points, because there are that many concave corners in the staircase between corners $c_1(j)$ and $c_1(j')$. We conclude that there are at least $(n-1)/2$ bounding boxes that store boxes $b(j), b(j')$ (and perhaps some more boxes) with $|j - j'| < \mu$. But if $|j - j'| < \mu$ then $j \not\equiv j' \pmod{\mu}$, so $\alpha(j) \neq \alpha(j')$. This implies that there is at least one i with $2 \le i \le d$ such that $c_i(j) \neq c_i(j')$. Hence, the bounding box storing $b(j), b(j')$ will contain one of the secondary query points. So in total we have at least $(n-1)/2$ incidences between secondary query points and bounding boxes, so one of the $(d-1)(\widehat{\mu}-1) = O(\mu^{1/(d-1)})$ secondary query points is contained in $\Omega(n/\mu^{1/(d-1)})$ bounding boxes.

The generalization to semi-R-trees follows easily from the observation that a semi-R-tree of minimum degree t has $\Omega(n/t)$ nodes. If each primary query point is contained in less than $\mu/2$ bounding boxes, we then get $\Omega(n/t)$ nodes whose bounding box contains less than μ primary query points. From that point on, we can basically follow the argument above. □

We can use this lemma to prove lower bounds for several settings. By substituting $\mu = (n/t)^{1-1/d}$, we prove the following lower bound for point queries.

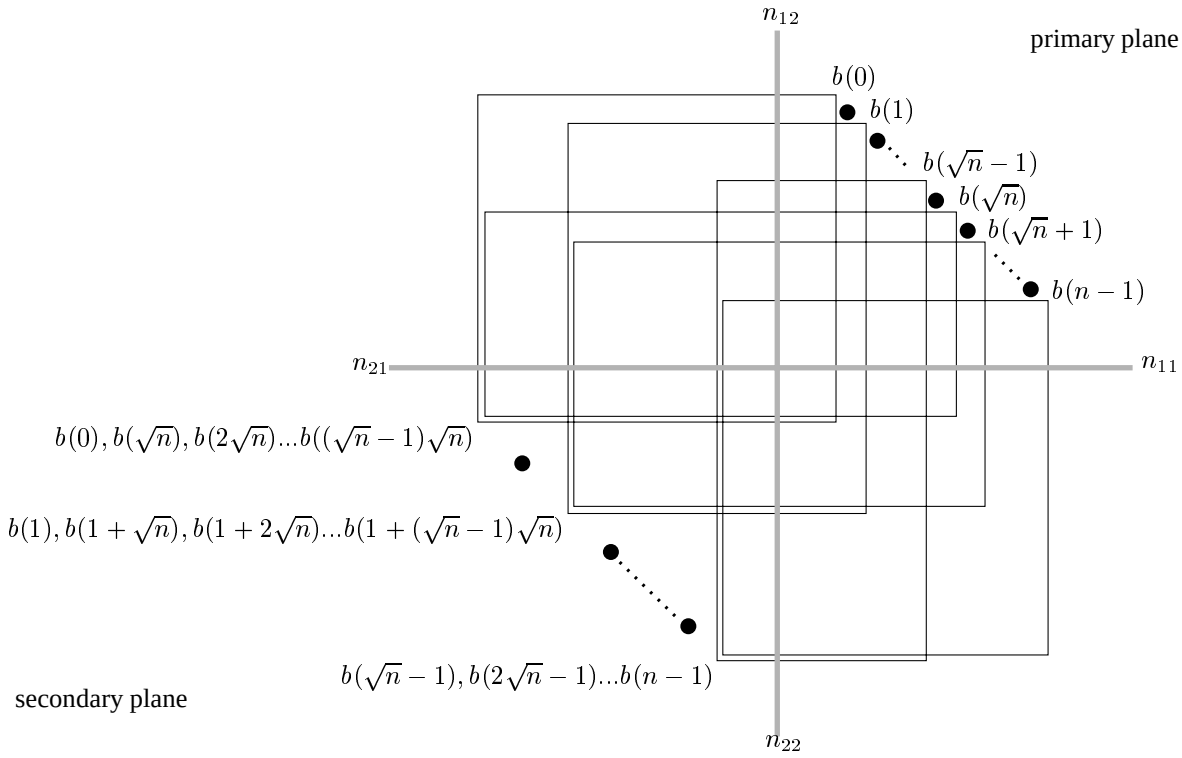

Figure 1: The lower bound construction in two dimensions for $\mu = \hat{\mu} = \sqrt{n}$. In this case, the primary and the secondary plane coincide. Each of the lower left corners is shared by $\sqrt{n}$ boxes (shown slightly displaced for clarity). The black dots indicate the locations of the query points in Q_1 and Q_2.

Theorem 2.3 *For any n, $d \geq 2$, and $\varepsilon > 0$, there is a set S of n ε-hypercubes in $\mathbb{R}^d$ with the following property: for any semi-R-tree $\mathcal{T}$ of minimum degree t there is a point not contained in any box from S such that a query with that point visits $\Omega((n/t)^{1-1/d})$ nodes in $\mathcal{T}$.*

Next, we modify the above construction so that the same bound can be achieved in $d \geq 3$ even if the input consists of a set of n *disjoint* ε-hypercubes and the queries are hypercubes.

Theorem 2.4 *For any n, $d \geq 3$, and $\varepsilon > 0$, there is a set S of n disjoint ε-hypercubes in $\mathbb{R}^d$ with the following property: for any semi-R-tree $\mathcal{T}$ of minimum degree t there is a hypercube not intersecting any box from S such that a query with that hypercube visits $\Omega((n/t)^{1-1/d})$ nodes in $\mathcal{T}$.*

Proof: We apply a variant of the construction above with $\mu = n^{1-1/(d-1)}$ to obtain a set of $(d-1)$-dimensional boxes in the hyperplane $x_1 = 0$. The variation is that we treat all planes on which we put the corners as secondary planes. We use the remaining dimension to make the boxes into d-dimensional ε-hypercubes, and we translate each box into the x_1-direction such that they become disjoint and intersect the x_1-axis in the order $b(1), b(2), \ldots, b(n)$. In between every pair $b(j), b(j+1)$ we put a query point. These $n-1$ query points play the role of the primary query points. The secondary query points are replaced by query ranges which are hypercubes. We can do that in such a way that the intersection of such a range with a secondary plane is a square that misses S and that has one corner coinciding with the secondary query points we had previously. It is easy to see that the bound in Lemma 2.2 still holds. □

Finally, we observe that the proof of the preceding theorem actually shows that in higher dimensions any semi-R-tree with small (say, polylogarithmic) query complexity for points must have large (near-linear) query complexity for ranges. More precisely, it shows the following result.

Theorem 2.5 *For any n, $d \geq 3$ and $\varepsilon > 0$, there is a set S of n disjoint ε-hypercubes in $\mathbb{R}^d$ with the following property: for any semi-R-tree $\mathcal{T}$ of minimum degree t, if the number of nodes visited by any point query is μ, then there is a hypercube not intersecting any box from S such that a query with that hypercube visits $\Omega(n/(t\mu^{1/(d-1)}))$ nodes in $\mathcal{T}$.*

3 From kd-trees to Box-Trees

In this section we describe and analyze several methods to construct box-trees using kd-trees. For convenience we will allow our box-trees to have nodes of degree up to $2d + 3$—it is easy to convert these trees to binary trees without affecting the asymptotic bounds on the query complexity. Query ranges (other than points) will be assumed to be open, while input boxes, bounding boxes and cells in space decompositions are closed.

3.1 The configuration-space approach

The basic method. Let S be a set of n arbitrary, possibly overlapping, d-rectangles in $\mathbb{R}^d$, which we call the *workspace*. As noted in the introduction, we can represent a d-rectangle $b = \prod_{i=1}^{d}[x_i^-(b), x_i^+(b)]$ by a point $(x_1^-(b), x_2^-(b), ..., x_d^-(b), x_1^+(b), x_2^+(b), ..., x_d^+(b))$ in $\mathbb{R}^{2d}$, which we call the *configuration space*. We build a $2d$-dimensional kd-tree on these points.

A kd-tree is a binary space decomposition tree, which is used to index points. Every node in a $2d$-dimensional kd-tree is associated with a cell, which is a $2d$-rectangle, and an axis-parallel splitting hyperplane. The splitting plane divides the cell into two rectangular subcells, one for each child of the node.

The root cell is chosen large enough to contain all input points. The tree is then built recursively by determining splitting planes for all cells. The orientations of the splitting planes depend on the level in the tree, in such a way that all possible orientations ($2d$ in this case) take turns in a round-robin fashion on any path down into the tree. The location of each splitting plane is chosen such that the numbers of input points in the resulting subcells differ by at most one. When a cell contains only one input point, we make it a leaf of the tree and do not split it further.

To transform the kd-tree in configuration space into a box-tree in workspace, proceed as follows. Replace the representative point in each leaf by the corresponding input box. Then, going bottom-up, store in each internal node the bounding box of its children. We call the resulting box-tree a *configuration-space box-tree*, or *cs-box-tree* for short.

In the introduction we pointed out that it can be used to do rectangle-intersection queries in $O(n^{1-1/2d} + k)$ time; in this paper we will show how to improve the upper bound to $O(n^{1-1/d} + k)$.

For the analysis of the range query complexity of the cs-box-tree, we need the following fact about kd-trees, given here without proof.

Lemma 3.1 *The number of cells at depth i in a d-dimensional kd-tree that intersect an axis-parallel f-flat ($0 \leq f \leq d$) is $O(2^{if/d})$.*

A kd-tree and, hence, our box-tree has the following property: the number of objects stored in the two subtrees of any given node differ by at most one. We call such trees *perfectly balanced*.

The perfect balance in our box-tree will be advantageous when we will convert it to an R-tree. We can now analyze the range query complexity of a cs-box-tree.

Lemma 3.2 *Let S be a set of n possibly intersecting boxes in the plane. There is a perfectly balanced box-tree for S such that the number of nodes at level i that are visited by a range query with an axis-aligned box is $O(2^{i(1-1/d)} + k)$, where k is the number of boxes in S intersecting the query range. The box-tree can be built in $O(n \log n)$ time.*

Proof: Let $Q = \prod_{i=1}^{d} (x_i^-(Q), x_i^+(Q))$ be a query range. We can restrict our attention to the interior nodes visited, since the number of visited leaves is at most one more. We distinguish two types of visited interior nodes ν. The first type is where at least one of the input boxes stored in the subtree of ν intersects Q. Obviously there are only $O(k)$ such nodes at a given level i. The second type is where all input boxes in the subtree of ν are disjoint from Q. The interior of any input box disjoint from Q must be separated from Q by a hyperplane through a facet of Q. Not all input boxes are separated from Q by the same hyperplane, otherwise the bounding box of ν would not intersect Q and ν would not be visited. Hence, there are at least two such hyperplanes separating Q from an input box in the subtree of ν.

Assume w.l.o.g. that $x_i = x_i^-(Q)$ is one of these separating hyperplanes, and let b be the input box it separates from Q. Then we must have $x_i^+(b) \leq x_i^-(Q)$. But there must also be a box b' with $x_i^+(b') > x_i^-(Q)$, otherwise the bounding box of ν would not intersect Q. We conclude that the points representing b and b' in the configuration space lie on or on opposite sides of the hyperplane $x_i^+ = x_i^-(Q)$. Consequently, the hyperplane $x_i^+ = x_i^-(Q)$ intersects the cell in configuration space of the node in the kd-tree corresponding to ν.

We can apply the same argument to the second hyperplane separating Q from an input box (the hyperplane $x_j = x_j^+(Q)$, for example), to show that there is a hyperplane in configuration space with points on or on opposite sides ($x_j^- = x_j^+(Q)$ in the example).

We can conclude the following. Suppose Q visits a node ν of the second type. Then in configuration space there is a pair of hyperplanes, both of the form $x_i^+ = x_i^-(Q)$ or $x_i^- = x_i^+(Q)$ and both intersecting the cell in configuration space of the kd-tree node corresponding to ν. But then the cell is also intersected by the $(2d-2)$-flat that is the intersection of these two hyperplanes. By Lemma 3.1 there are only $O(2^{i(2d-2)/2d}) = O(2^{i(1-1/d)})$ such nodes at level i.

For the building time, see section 3.4. □

This leads directly to the following theorem.

Theorem 3.3 *Let S be a set of n possibly intersecting boxes in the plane. There is a perfectly balanced box-tree for S such that the number of nodes visited by a range query with an axis-aligned box is $O(n^{1-1/d} + k \log n)$, where k is the number of boxes in S intersecting the query range. The box-tree can be built in $O(n \log n)$ time.*

Proof: From Lemma 3.2 we get a bound for the stabbing number on each level in the tree. Since a kd-tree has height $\lceil \log n \rceil$, so has a cs-box-tree, and summation over all levels yields a total query complexity of $\sum_{i=0}^{\lceil \log n \rceil} O(2^{i(1-1/d)} + k) = O(n^{1-1/d} + k \log n)$. □

Improving the query time. We now show how to reduce the $O(k \log n)$ term in the query complexity to $O(k)$. The idea is the same as in a priority search tree [7]: input elements (boxes in our case) that have a high chance of being reported are pushed to high levels in the tree. In

our case, the boxes that extend farthest in one of the x_i-directions are stored high in the tree. More precisely, the construction of the tree $\mathcal{T}$ for a set S of boxes in $\mathbb{R}^d$ is as follows.

If $|S| = 1$, then $\mathcal{T}$ consists of a single leaf node storing the input box in S. Otherwise we make a node ν storing the bounding box B_ν of all boxes in S, and proceed as follows.

For each of the $2d$ inner normals of the facets of B_ν, take the box from S that extends farthest in the direction of that normal. This results in a set S^* of at most $2d$ boxes. Each box in S^* is put in a so-called *priority leaf*, which is an immediate child of ν.

If the set $S \setminus S^*$ of remaining boxes contains less than two boxes, then this box (if it exists) is put as a leaf child of ν. If two or more boxes remain, we split the set of boxes into two (almost) equal-sized subsets with an axis-parallel hyperplane in configuration space. Like in a normal kd-tree, the orientation of the splitting plane depends on the level in the tree, so that all $2d$ orientations take turns in a round-robin fashion on any path from the root down into the tree.

The subset of boxes whose representative points lie to one side of the cutting hyperplane are stored recursively in one subtree of ν. The subset of boxes whose representative points lie to the other side of the cutting hyperplane are stored recursively in another subtree of ν.

Next we analyze the query complexity of the tree resulting from this construction, which we call a *cs-priority-box-tree*. In our analysis we bound the number of visited nodes of a given weight, where the weight of a node is defined as the number of input boxes stored in its subtree. This will be useful when we convert this box-tree into a semi-R-tree.

Lemma 3.4 *The number of nodes of weight at least w visited by a query with a query box Q is $O((n/w)^{1-1/d} + k)$.*

Proof: Let $Q = \prod_{i=1}^{d}(x_i^-(Q), x_i^+(Q))$. We can restrict our attention to the visited nodes of weight at least $2d$, as the total number of visited nodes is at most a constant times larger than this number. Let ν be such a visited node of weight at least $2d$. There are two cases.

The first case is where one of the priority leaves directly below ν stores a box intersecting Q. Clearly there are at most k such nodes.

The second case is when all priority leaves directly below ν store boxes disjoint from Q. Thus each such box's interior is separated from Q by a hyperplane through a facet of Q. We claim that not all boxes can be separated by the same hyperplane. Suppose for a contradiction that there is a facet f whose containing hyperplane separates all boxes of the priority leaves from Q. Then in particular it would separate the box that extends farthest in the direction of the inner normal of the facet f, contradicting that Q intersects the bounding box stored at ν. So we have two distinct hyperplanes through facets of Q separating a box in the subtree of ν from Q.

The box-tree that we have constructed basically corresponds to a kd-tree in configuration space, as before. The priority leaves make that the tree in configuration space is strictly speaking not a kd-tree, but it is easy to see that Lemma 3.1 still holds. Moreover, there is still a one-to-one correspondence between nodes of the box-tree and nodes of the kd-tree in configuration space. Hence, we can use the fact that there are two distinct hyperplanes through facets of Q separating a box in the subtree of ν from Q in the same way as in the proof of Lemma 3.2: it implies that there is a $(2d-2)$-flat in configuration space (defined by a pair of facets of Q) intersecting the cell in the kd-tree corresponding to ν. It follows that the total number of nodes ν to which the second case applies at a given level i is $O(2^{i(1-1/d)})$.

To finish the proof, observe that nodes at the lowermost $\lfloor \log(w/(2d)) \rfloor$ levels have weight less than w. Adding the bounds for the second case on the remaining levels, we get $\sum_{i=0}^{\lceil \log n \rceil - \lfloor \log(w/(2d)) \rfloor} O(2^{i(1-1/d)}) = O((n/w)^{1-1/d})$.

For the building time, see section 3.4. □

The following theorem follows directly.

Theorem 3.5 *Let S be a set of n possibly intersecting boxes in $\mathbb{R}^d$. There is a box-tree for S such that the number of nodes that are visited by a range query with an axis-aligned box is $O(n^{1-1/d} + k)$, where k is the number of boxes in S intersecting the query range.*

3.2 The kd-interval-tree approach

The cs-box-tree of the previous section has optimal query complexity for point queries (and range queries) if the input consists of arbitrary, intersecting boxes. Unfortunately, if the input boxes are disjoint then the query complexity for point queries does not improve. In this section we develop a different box-tree, the *kd-interval tree*, whose query complexity is much better if σ, the point-stabbing number of the input set S, is small. The query complexity for range queries increases only slightly. This approach only works in the plane; Theorem 2.5 states that a similar result in more than two dimensions cannot be obtained.

The basic idea behind kd-interval trees is again to use a kd-tree, but this time in the workspace (which is now the plane). Since the objects in the workspace are rectangles, not points, many of them may intersect the cutting line. These boxes are taken out and handled separately, like in an interval tree. To make kd-interval trees more efficient, we introduce priority leaves, like in the previous section.

The 1-dimensional case. First we describe how a set S of boxes all intersecting a given line ℓ are handled. With a slight abuse of terminology, we call a tree for this case a 1-dimensional kd-interval tree.

If $|S| = 1$, then $\mathcal{T}$ consists of a single leaf node storing the input rectangle in S. Otherwise we make a node ν storing the bounding box B_ν of all rectangles in S, and proceed as follows.

For each of the 4 inner normals of the edges of B_ν, take the rectangle from S that extends farthest in the direction of that normal. This results in a set S^* of at most 4 rectangles. Each rectangle in S^* is put in a *priority leaf*.

Consider the set of intersections of the edges of the remaining rectangles with ℓ. Let p be the median of these intersection points. The rectangles in $S \setminus S^*$ containing p are stored in a subtree of ν that is a 2-dimensional cs-priority-box-tree as described in the previous section. The rectangles in $S \setminus S^*$ completely to one side of p are stored recursively as a 1-dimensional kd-interval tree in a second subtree of ν. The rectangles in $S \setminus S^*$ completely to the other side of p are stored recursively in another subtree of ν.

We call the nodes in the main 1-dimensional kd-interval tree *1D-nodes*. Such a node corresponds to an interval on the defining line ℓ. We call the nodes of the 2-dimensional cs-priority-box-trees *cs-nodes*.

We start by analysing the query complexity when we query with a segment on the line ℓ.

Lemma 3.6 *If we query a 1-dimensional kd-interval tree storing a set S of n rectangles with a line segment on the defining line ℓ, then we visit at most $O(\log n + k)$ nodes, where k is the number of rectangles to be reported.*

Proof: Observe that the query segment s intersects a rectangle (or bounding box) if and only if it intersects the intersection of that rectangle (or bounding box) with ℓ.

Consider a 1D-node that is visited when we query with s. When the interval corresponding to this node is completely contained in s, then by the above observation all rectangles in the

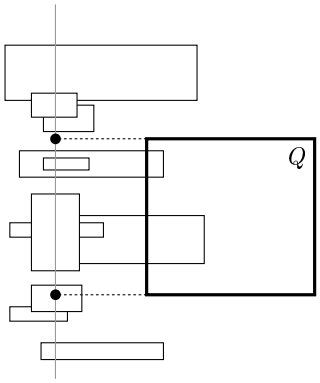

Figure 2: Querying a 1-dimensional kd-interval tree with a box Q.

subtree intersect s. Hence, there cannot be more than $O(k)$ such nodes. When the interval is not completely contained in s, then it contains an endpoint of s, and there are only $O(\log n)$ such nodes.

Now consider a cs-node ν that is visited. Let p be the point on ℓ common to all rectangles in the subtree of ν. Assume w.l.o.g. that ℓ is vertical and p lies inside or above s. Then the rectangle in the subtree extending farthest downward must intersect s. This rectangle is stored in a priority node directly below ν, so we can charge the visit of ν to this answer. □

Next we analyze the query complexity when we query with a box.

Lemma 3.7 *(i) If we query a 1-dimensional kd-interval tree storing a set S of n rectangles with a query box Q, then we visit at most $O(\sqrt{\sigma/w}\log n + k)$ nodes of weight at least w, where k is the number of rectangles to be reported.*
(ii) If σ is $O(\log n)$, then the query time reduces to $O(\log n + k)$.
(iii) If the projection of Q onto the line ℓ that stabs the rectangles in S contains the intersections of all rectangles with ℓ, then the query time reduces to $O(k)$.

Proof: (i) See Figure 2. If Q intersects ℓ then the query is equivalent to querying with $Q \cap \ell$, so the result follows from the previous lemma. Otherwise, assume w.l.o.g. that ℓ is vertical and that Q lies to the right of ℓ. Consider a 1D-node ν that is visited when we query with Q. When the interval corresponding to this node is completely contained in the projection of Q onto ℓ, then the rectangle in the subtree extending farthest to the right must be intersected. This rectangle is stored in a priority leaf immediately below ν, to which we can charge the visit of ν. Hence, there can be at most k such nodes. When the interval is not completely contained in the projection of Q, then it contains an endpoint of the projection of Q, and there are only $O(\log n)$ such nodes.

Now consider a 2-dimensional cs-priority-box-tree that is visited. Suppose the interval of the 1D-node that is the parent of this subtree is completely contained in Q. Then we can argue again (using the priority leaves) that we can charge all the visited nodes to rectangles intersecting Q. If the interval of the 1D-node that is the parent of this subtree is not completely contained in the projection of Q, we argue as follows. First observe that the interval must then contain an endpoint of the projection of Q, so there are only $O(\log n)$ such parent nodes. In the 2-dimensional configuration-space box-tree below such a parent, we apply Lemma 3.4 to bound the number of visited nodes of weight w by $O(\sqrt{n'/w} + k')$, where n' is the number of boxes stored in the cs-priority-box-tree and k' is the number of answers reported in this subtree. Note

that $n' \leq \sigma$, since the cs-box-trees are used only to store sets of boxes that share a single point. Hence, the overall number of cs-nodes visited is $O(\sqrt{\sigma/w}\log n + k)$, finishing the proof of part (i) of the lemma.

(ii) For the proof of part (ii), we analyze the number of cs-nodes visited in a different way. Note that cs-nodes in a single cs-priority-box-tree share a single point on ℓ. If this point is contained in the projection of Q onto ℓ, then we can use the priority nodes to charge all nodes visited in this cs-box-tree to rectangles intersecting Q.

If the defining point of a cs-prority-box-tree lies outside the projection of Q onto ℓ, then each cs-node ν visited in this cs-box-tree must have at least one rectangle that contains an endpoint of the projection of Q. For each such node ν, the rectangle in its subtree which extends farthest into (or beyond) the projection of Q, is stored in a priority node directly below ν, to which we can charge the visit of ν. In all cs-box-trees together, at most 2σ priority nodes can contain one of the two endpoints; therefore, at most $O(\sigma)$ cs-nodes with defining points outside the projection of Q can be visited.

In total, we find a bound of $O(\log n + \sigma + k)$, which reduces to $O(\log n + k)$ if σ is $O(\log n)$.

(iii) If the projection of Q onto ℓ contains the intersections of all rectangles with ℓ, it also contains all intervals corresponding to the nodes in the box-tree. Therefore, we can use the priority leaves again to charge all the visited nodes to rectangles intersecting Q. □

The 2-dimensional case. Our kd-interval tree for a general set S of rectangles in the plane is defined as follows.

If $|S| = 1$, then $\mathcal{T}$ consists of a single leaf node storing the input box in S. Otherwise we make a node ν storing the bounding box B_ν of all boxes in S, and proceed as follows.

For each of the 4 inner normals of the edges of B_ν, take the rectangle from S that extends farthest in the direction of that normal. This results in a set S^* of at most 4 rectangles. Each rectangle in S^* is put in a *priority leaf*, which is an immediate child of ν.

If the set $S \setminus S^*$ of remaining rectangles contains less than two rectangles, then this rectangle (if it exists) is put as a leaf child of ν. If two or more rectangles remain, we split the cell corresponding to ν using a vertical or horizontal line (depending on the level ν in the tree). This splitting line ℓ is chosen such that the number of rectangles in $S \setminus S^*$ lying completely to either side of ℓ is at most $\lfloor |S \setminus S^*|/2 \rfloor$. The rectangles in $S \setminus S^*$ lying to one side of ℓ are stored recursively in one subtree of ν. The rectangles in $S \setminus S^*$ lying to the other side of ℓ are stored recursively in another subtree of ν. The rectangles in $S \setminus S^*$ intersecting ℓ are stored in a 1-dimensional kd-interval tree, as explained above. We call the nodes of the main tree, which correspond to 2-dimensional cells, *2D-nodes*. Next we analyze the performance of the kd-interval tree.

Lemma 3.8 *The number of nodes of weight at least w that are visited by a range query with an axis-aligned box is $O(\sqrt{n/w}\log n + \sqrt{\sigma/w}\log^2 n + k)$, where k is the number of reported answers. The number of such nodes visited by a point query is $O(\sqrt{\sigma/w}\log^2 n + k)$. If σ is $O(\log n)$, we may omit the $\sqrt{\sigma/w}$ factor.*

Proof: Consider a 2D-node that is visited when we query with an axis-aligned rectangle Q. We distinguish four different types of such nodes (see Figure 3). We bound their number and the number of nodes visited in 1-dimensional kd-interval-subtrees for each type separately.

Inner nodes: These are 2D-nodes whose bounding boxes lie completely inside Q. The number of inner nodes is easy to bound, since all rectangles in the subtree of such a node intersect Q.

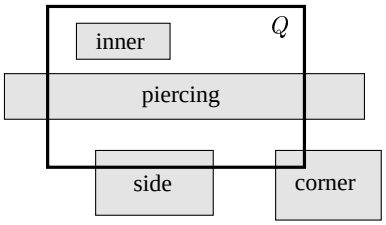

Figure 3: Four different types of 2D-nodes with respect to a query range Q.

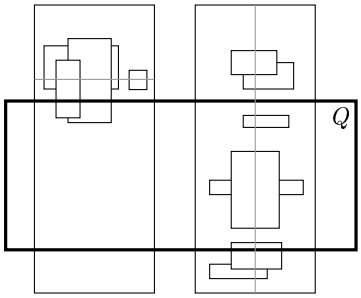

Figure 4: Piercing nodes with parallel splitting lines (to the left) and orthogonal splitting lines (to the right).

Hence, the total number of such nodes, or nodes in their 1-dimensional associated kd-interval trees, is $O(k)$.

Side nodes: These are 2D-nodes whose bounding boxes cut exactly one edge of Q. In this case the rectangle that extends farthest into the direction of the inner normal of this edge must intersect Q. This rectangle is stored in a priority leaf immediately below the node. The same reasoning applies to their 1-dimensional associated kd-interval trees. Hence, the total number of side nodes or nodes in their associated kd-interval trees is $O(k)$.

Piercing nodes: These are 2D-nodes that cut two opposing edges of Q, but do not contain any corners of Q. From Lemma 3.1 and the fact that all nodes at the lowermost $\lfloor \log(w/(2d)) \rfloor$ levels of the tree must have weight less than w, we conclude that the number of 2D-nodes with weight at least w that intersect any edge of Q must be bounded by $\sum_{i=0}^{\lceil \log n \rceil - \lfloor \log(w/(2d)) \rfloor} O(2^{i/2}) = O(\sqrt{n/w})$. Now there are two cases—see Figure 4: the splitting line used at such a node ν is orthogonal to the intersected edges, or it is parallel to them. In the former case we can apply Lemma 3.6 to obtain a $O(\log n + k')$ bound on the number of nodes visited in the 1-dimensional kd-interval tree associated with ν, where k' is the number of reported answers. In the latter case we can apply Lemma 3.7(iii) to get a bound of $O(k')$. Hence, we get a grand total of $O(\sqrt{n/w} \log n + k)$.

Corner nodes: These are 2D-nodes that contain one or more corners of Q. There are $O(\log n)$ such nodes. To obtain the total number of visited nodes in the associated 1-dimensional kd-interval trees, we have to multiply this by the bound of Lemma 3.7, leading to a total of $O(\sqrt{\sigma/w} \log^2 n + k)$ in the general case, or $O(\log^2 n + k)$ if σ is $O(\log n)$.

There are no other types of nodes whose bounding boxes intersect Q. Adding up the number of nodes for all four cases gives the desired bound for box-queries. Note that in the case of

point queries, we only have corner nodes. For the building time, see section 3.4. □

This leads to the following theorem.

Theorem 3.9 *Let S be a set of n possibly intersecting boxes in the plane, such that no single point is contained in more than σ boxes. There is a box-tree for S such that the number of nodes visited by a range query with an axis-aligned box is $O(\sqrt{n}\log n + \sqrt{\sigma}\log^2 n + k)$, where k is the number of boxes in S intersecting the query range. The number of nodes visited by a point query is $O(\sqrt{\sigma}\log^2 n + k)$. If σ is $O(\log n)$, this reduces to $O(\log^2 n)$. The box-tree can be built in $O(n \log n)$ time.*

3.3 The longest-side-first approach

Recall that a kd-interval tree is basically a modified kd-tree, where each node is split by a line. The orientations of these lines depend on the level in the tree in such a way, that orientations take turns in a round-robin fashion on any path from the root down into the tree. An interesting variation of the kd-interval tree arises when we replace the round-robin splitting strategy by the longest-side splitting rule as suggested by Dickerson et al. [8]. In such a longest-side-first kd-interval tree, the number of nodes whose corresponding cell is pierced by a query rectangle is small if the query rectangle is fat. We use this to prove the following lemma.

Lemma 3.10 *The number of nodes of weight at least w that are visited by a range query with an axis-aligned box is $O((\alpha + \sqrt{\sigma/w})\log^2 n + k)$, where k is the number of reported answers. The number of such nodes visited by a point query is $O(\sqrt{\sigma/w}\log^2 n + k)$. If σ is $O(\log n)$, the $O(\sqrt{\sigma/w})$ factor can be omitted from the bounds.*

Proof: In the analysis in the previous subsection, the piercing nodes were responsible for the $O(\sqrt{n/w}\log n)$ term in the query complexity. This term arose because in a normal kd-tree there can be $O(\sqrt{n/w})$ piercing nodes, and in each of the associated 1-dimensional kd-interval trees, $O(\log n)$ nodes could be visited.

In the longest-side-first kd-tree, however, the number of disjoint cells that cut opposing sides of a query rectangle of aspect ratio α is $O(\alpha \log n)$ [8]. As before, we have two types of piercing nodes: those with splitting lines that are orthogonal to the intersected edges of Q, and those with parallel splitting lines. For the first case, observe that such splitting lines separate two disjoint cells that cut opposing sides of the query rectangle. This implies that there can be at most $O(\alpha \log n)$ piercing nodes with orthogonal splitting lines, each of which can have a 1-dimensional kd-interval tree in which $O(\log n + k')$ nodes are visited. For the second case, observe that the total number of piercing nodes on all levels in the tree is at most $O(\alpha \log^2 n)$, and each of them can have a 1-dimensional kd-interval tree in which $O(k')$ nodes are visited. Hence, we get a grand total of $O(\alpha \log^2 n + k)$ for both types of piercing nodes.

Since the other cases in the analysis of the original kd-tree still go through, the lemma follows. □

Theorem 3.11 *Let S be a set of n boxes in the plane with stabbing number σ. There is a box-tree for S such that the number of nodes that are visited by a range query with a rectangular range of aspect ratio α is $O((\alpha+\sqrt{\sigma})\log^2 n+k)$, where k is the number of boxes in S intersecting the query range. The number of such nodes visited by a point query is $O(\sqrt{\sigma}\log^2 n + k)$. If σ is $O(\log n)$, the $O(\sqrt{\sigma})$ factor can be omitted from the bounds. The box-tree can be built in $O(n \log n)$ time.*

3.4 Building the box-trees

All boxtrees mentioned in this section, can be built in $O(n \log n)$ time. Since the construction algorithms are very similar, we will explain them together.

We start by sorting all input boxes by x_i^--coordinate and x_i^+-coordinate for all dimensions $1 \leq i \leq d$. This costs $O(n \log n)$ time. Using suitable list structures and cross-pointers, we can now do the following operations:

- in $O(1)$ time, selecting a box with an extreme value for one of the $2d$ coordinates and removing it from the $2d$ sorted lists;
- in $O(1)$ time, determine the bounding box of the set (and, if necessary, determine the dimension in which the bounding box is largest);
- in $O(n)$ time, splitting the set of boxes in two, such that all boxes whose value for a particular coordinate is smaller than the median for that coordinate go in one list, while the remaining boxes go in the other list, and at the same time splitting the $2d$ sorted lists in sorted lists for each of the two subsets.
- in $O(n)$ time, splitting the set of boxes in three subsets S^-, S^0 and S^+ with respect to some discriminating dimension i, such that there is a value x_i^0 such that all boxes in S^- are on one side of the hyperplane $x_i = x_i^0$, all boxes in S^+ are on the other side, and all boxes in S^0 intersect the plane, $|S^-| \leq n/2$ and $|S^+| \leq n/2$—and at the same time, splitting the $2d$ sorted lists in sorted lists for each of the three subsets.

  This operation can be implemented by choosing x_i^0 to be the median value of the union of the x_i^-- and x_i^+-coordinates. Using the lists ordered by these two coordinates, we can find the median value in $O(n)$ time. By definition, at most n coordinate values can be smaller than the median and at most n coordinate values can be greater than the median. This implies that at most $n/2$ input boxes can be completely on one side of the median hyperplane, and at most $n/2$ can be completely on the other side. After we have found the median, we can just check all boxes to see on which side they are, assign them to one of the three subsets, and then split the sorted lists accordingly.

The boxtrees can now be built top-down recursively, following the descriptions in the previous subsections. First we make a root for a tree that has to store all boxes, we calculate how to divide these boxes among its children, and then we split the set of boxes, giving each child its own subset. With the above operations we can do this for cs-box-trees, cs-priority-box-trees, kd-interval trees as well as for longest-side-first kd-interval trees in $O(n)$ time, where n is the number of boxes that has to be stored in the tree rooted at this node.

Then we construct the childrens' subtrees recursively, spending $O(n)$ time in total for each level in the tree. Since all box-trees constructed in this section have height $O(\log n)$, the total time for division and construction is $O(n \log n)$.

Adding the time needed for sorting to the time needed for division and construction, we get a total building time of $O(n \log n)$.

4 From box-trees to R-trees

In the previous section we described several algorithms to construct box-trees with good query complexity. In this section we give general theorems to convert them to (semi-)R-trees.

We start with a general theorem that converts any box-tree to an R-tree. Recall that the *weight* of a box-tree node is the number of input boxes stored in its subtree.

Theorem 4.1 *Let $\mathcal{T}$ be a box-tree for a set of n boxes in $\mathbb{R}^d$ such that any query with a range of a given type visits at most $f(w)$ nodes of weight w or more. Then $\mathcal{T}$ can be converted in $O(n)$ time to an R-tree of minimum degree t where every query with a range of the same type visits at most $O(f(t)\log n/\log t)$ nodes.*

Proof: We simply read out the leaves from $\mathcal{T}$ in order, and then construct an R-tree where the boxes occur in the same order in the leaves. We can build this R-tree bottom-up, level by level. First we construct the R-tree nodes just above leaf level by repeatedly taking $2t$ leaves from the list and giving them a new R-tree node as their parent. We continue doing this until less than $4t$ leaves are without parent: these leaves are then divided into two groups (if there are more than $2t$) or made children of a single parent (if there are no more than $2t$ leaves left). Next, we consider the new parent nodes just constructed as leaves, and construct the next level of the tree, and so on, until we reach the level where only one node is constructed (the root). In this way, we spend $O(1)$ time for each node to connect it to a parent node, thus getting a total running time of $O(n)$.

Consider a bounding box B stored in the R-tree. It is the bounding box for some input boxes that were stored in consecutive leaves in the box-tree $\mathcal{T}$. Let $\nu(B)$ be the lowest common ancestor of these leaves. Since the minimum degree in the R-tree is t, the weight of $\nu(B)$ is t or more. Furthermore, the nodes $\nu(B)$ for the bounding boxes B stored at a fixed level in the R-tree must be distinct, because their defining sets form a partition of the leaves in $\mathcal{T}$ into consecutive sequences. Hence, we can charge the visited nodes of the R-tree to visited nodes of weight t or more in $\mathcal{T}$, in such a way that a node in $\mathcal{T}$ does not get charged more than once from nodes at a fixed level in the R-tree. Since the depth of the R-tree is $O(\log n/\log t)$, the bound follows. □

The construction of Theorem 4.1 results in losing a logarithmic factor in the query complexity. Next we show how to improve this result for perfectly balanced box-trees. Recall that a box-tree is called perfectly balanced if for any node the weight of its left and right child differ by at most one.

Theorem 4.2 *Let $\mathcal{T}$ be a perfectly balanced box-tree for a set of n boxes in $\mathbb{R}^d$ such that any query with a range of a given type visits at most $f(i)$ nodes at level i in $\mathcal{T}$. Then $\mathcal{T}$ can be converted in $O(n)$ time to an R-tree of minimum degree t where every query with a range of the given type visits at most $O(\sum_{i=0}^{(\log n/\log t)-1} f(i\log t))$ nodes.*

Proof: We first prove that any perfectly balanced tree has the following property: the weights of all nodes at a fixed level in the tree differ by at most one. The proof is by induction on the level. The statement is trivially true at level zero (the level of the root). Now assume all nodes at a given level have weight w or $w+1$. Then the balancing condition guarantees that the nodes at the next level have weight $w/2$ or $w/2+1$ (in case w is even) or they have weight $(w+1)/2-1$ or $(w+1)/2$ (in case w is odd). So in both cases the weights at the next level differ by at most one.

We can now construct an R-tree from $\mathcal{T}$ as follows. From the leaf level of the box-tree, walk up the tree until a level i is encountered where all nodes have weight at least t. Thus there must be at least one node with weight at most $t-1$ on the level just below i, and therefore, by the perfect-balance property, no node on that level has weight more than t. This implies that the

weight of nodes at level i cannot exceed $2t$. Hence, each subtree rooted at a node at this level can be compressed in a single leaf (which will be a node in the R-tree). Recurse on the new tree. The recursion ends when there are less than t leaves, which are compressed to a single node which will form the root of the R-tree. It is immediately clear that this construction can be done in $O(n)$ time.

The bound on the query complexity immediately follows from the construction. □

Finally, we can show that that we can also improve Theorem 4.1 for the general case if we are willing to settle for semi-R-trees instead of real R-trees. Recall that the difference between a semi-R-tree and an R-tree is that in the former we do not require all leaves to be at the same depth.

Theorem 4.3 *Let $\mathcal{T}$ be a box-tree for a set of n boxes in $\mathbb{R}^d$ such that any query with a range of a given type visits at most $f(w)$ nodes of weight w or more. Then $\mathcal{T}$ can be converted in $O(n)$ time to a semi-R-tree of minimum degree t where every query with a range of the same type visits at most $O(f(t))$ nodes.*

Proof: We start by converting the binary box-tree to a forest of at least 1 and at most $t-1$ semi-R-trees. This is done recursively as follows. If the box-tree is just a leaf, we leave it as it is. Otherwise, we convert the left and the right subtree separately, getting two forests of at least 2 and at most $2(t-1)$ semi-R-trees in total. We distinguish two cases:

- The total number of semi-R-trees is less than t. In this case, we are done immediately.
- The total number of semi-R-trees is at least t. In this case, we combine the semi-R-trees in the two forests into a single semi-R-tree by making the semi-R-trees in the forests the children of a new root node. Note that the new root node has between t and $2(t-1)$ children. The descendant leaves of this new root node are exactly the descendant leaves of the box-tree node which is being converted, so the associated bounding box is exactly the same; no new bounding box is introduced.

In the end we get a forest of at least 1 and at most $t-1$ semi-R-trees. If it is not a single tree, we combine the trees in the forest into one tree by adding a root node.

Clearly each node in the box-tree will be processed exactly once and will be processed in O(1) time if the forest operations are implemented suitably. Therefore, the conversion of a complete box-tree takes $O(n)$ time.

No new bounding boxes are introduced, no bounding box in the boxtree appears more than once in the semi-R-tree, and no internal nodes with weight less than t are constructed. This is easily seen to result in a semi-R-tree with the desired bound on the query complexity. □

By applying the conversion algorithms of the theorems above to the structures from the previous section, we obtain the following results.

Corollary 4.4 *Let S be a set of n boxes in $\mathbb{R}^d$ with stabbing number σ.*

(i) There is an R-tree for S of minimum degree t such that the number of nodes visited by any box query is $O((n/t)^{1-1/d} + k \log n / \log t)$, where k is the number of reported answers.

(ii) There is an semi-R-tree for S of minimum degree t such that the number of nodes visited by any box query is $O((n/t)^{1-1/d} + k)$.

(iii) When $d = 2$, there is a semi-R-tree for S of minimum degree t such that the number of nodes visited by any box query is $O(\sqrt{n/t}\log n + \sqrt{\sigma/t}\log^2 n + k)$, and the the number of nodes visited by any point query is $O(\sqrt{\sigma/t}\log^2 n + k)$. In both bounds, k is the number of reported answers. If σ is $O(\log n)$, the $O(\sqrt{\sigma/t})$ factor can be omitted from the bounds.

(iv) When $d = 2$, there is a semi-R-tree for S of minimum degree t such that the number of nodes visited by any query with a rectangle of aspect ratio α is $O((\alpha + \sqrt{\sigma/t})\log^2 n + k)$, where k is the number of reported answers. If σ is $O(\log n)$, the bound reduces to $O(\alpha \log^2 n + k)$.

(v) For the cases mentioned under (iii) and (iv) there is also an R-tree of minimum degree t for which the number of visited nodes is $O(\log n/\log t)$ times the number of visited nodes in the semi-R-tree.

All R-trees can be constructed in $O(n \log n)$ time.

Proof: Part (i) follows from Theorem 4.2 and Lemma 3.2. Part (ii) follows from Theorem 4.3 and Theorem 3.5, and part (iii) follows from Theorem 4.3 and Lemma 3.8. Part (iv) follows from Theorem 4.3 and Theorem 3.10. To obtain part (v), we use Theorem 4.1 instead of Theorem 4.3. □

5 Conclusions

We have developed now algorithms to construct box-trees (bounding-volume hierarchies using axis-aligned boxes as bounding volumes) and we analyzed the complexity of rectangle-intersection queries and point-containment queries for these structures. We also proved lower bounds showing that our results are optimal or almost optimal. Finally, we gave algorithms to convert our box-trees to (semi-)R-trees with optimal or almost optimal query complexity.

The bounds that we get, except for the case of fat ranges in the plane, are rather disappointing—even though they are optimal. In practice, one would hope for much better performance. It would be interesting to see under which conditions one can obtain better bounds for, say, box-queries in $\mathbb{R}^3$. We also would like to see how our trees behave in practice—the lower-bound constructions are rather contrived—and to compare them experimentally against trees constructed by known heuristics.

In many applications it is important to support fast insertions and deletions, and it would be interesting to develop box-trees or R-trees that support fast insertion and deletion, while still guaranteeing close to optimal query complexity.

References

[1] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.

[2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] G. Barequet, B. Chazelle, L. J. Guibas, J. S. B. Mitchell, and A. Tal. BOXTREE: A hierarchical representation for surfaces in 3D. In *Computer Graphics Forum*, volume 15, pages 387–396, 1996.

[4] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 197–208, 1994.

[5] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal of Computing*, 17:427–462, 1988.

[6] M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars. On R-trees with low stabbing number. In *Proc. 8th European Symposium on Algorithms, LNCS*, volume 1879, pages 167–178, 2000.

[7] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[8] M. Dickerson, C. Duncan, and M. Goodrich. K-D trees are better when cut on the longest side. In *Proc. 8th European Symposium on Algorithms*, volume 1879 of *LNCS*, pages 179–190, 2000.

[9] C. Faloutos and I. Kamel. Packed R-trees using fractals. Report CS-TR-3009, University of Maryland, College Park, 1992.

[10] C. Faloutos, T. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 426–439, 1987.

[11] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–205, 1998.

[12] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: a hierarchical structure for rapid interference detection. In *ACM Computer Graphics Proceedings*, pages 171–180, 1996.

[13] A. Guttmann. R-trees: a dynamic indexing structure for spatial searching. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[14] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[15] U. Lauther. 4-dimensional binary search trees as a means to speed up associative searches in design rule verification of integrated circuits. *Journal of Design Automation and Fault-Tolerant Computing*, 2(3):241–247, 1978.

[16] S. Leutenegger, M. A. Lopez, and J. Edington. STR: A simple and efficient algorithm for R-tree packing. In *Proc. 13th IEEE International Conference on Data Engineering*, pages 497–506, 1997.

[17] Y. Manolopoulos, Y. Theodoridis, and V. Tsotras. *Advanced Database Indexing*. Kluwer Academic Publishers, 1999.

[18] 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 Geographic Information Systems*, volume 1340 of *LNCS*, pages 153–198. 1997.

[19] J. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 343–352, 1990.

[20] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proc. Annual Symposium on Principles of Database Systems*, pages 161–171, 1996.

[21] Y. Zhou and S. Suri. Analysis of a bounding box heuristic for object intersection. In *Proc. 10th Annual Symposium on Discrete Algorithms (SODA)*, 1999.