Approximate Range Searching Using Binary Space Partitions

Mark de Berg*† Micha Streppel*‡

Abstract

We show how any BSP tree $\mathcal{T}_P$ for the endpoints of a set of n disjoint segments in the plane can be used to obtain a BSP tree of size $O(n \cdot \text{depth}(\mathcal{T}_P))$ for the segments themselves, such that the range-searching efficiency remains almost the same. We apply this technique to obtain a BSP tree of size $O(n \log n)$ such that ε-approximate range searching queries with any constant-complexity convex query range can be answered in $O(\min_{\varepsilon>0}\{(1/\varepsilon) + k_\varepsilon\} \log n)$ time, where k_ε is the number of segments intersecting the ε-extended range. The same result can be obtained for disjoint constant-complexity curves, if we allow the BSP to use splitting curves along the given curves.

We also describe how to construct a linear-size BSP tree for low-density scenes consisting of n objects in $\mathbb{R}^d$ such that ε-approximate range searching with any constant-complexity convex query range can be done in $O(\log n + \min_{\varepsilon>0}\{(1/\varepsilon^{d-1}) + k_\varepsilon\})$ time.

Finally we show how to adapt our structures so that they become I/O-efficient.

1 Introduction

Multi-functional data structures and BSP trees. In computational geometry, efficient data structures have been developed for a variety of geometric query problems: range searching, point location, nearest-neighbor searching, etc. The theoretical performance of these structures is often close to the theoretical lower bounds. In order to achieve close to optimal performance, most structures are dedicated to very specific settings; for instance, a structure for range searching with rectangular ranges in a set of line segments will not work for range searching with circular ranges in a set of line segments or for range searching with rectangular ranges in a set of circles. It would be preferable, however, to have a single *multi-functional geometric data structure*: a data structure that can store different types of data and answer various types of queries. Indeed, this is what is often done in practice.

Another potential problem with the structures developed in computational geometry is that they are sometimes rather involved, and that it is unclear how large the constant factors in the query time and storage costs are. Moreover, they may fail to take advantage of the cases where the input objects and/or the query have some nice properties. Hence, the ideal would be to have a multi-functional data structure that is simple and takes advantage of any favorable properties of the input and/or query.

The existing multi-functional data structures can be roughly categorized into *bounding-volume hierarchies (BVHs)* and *space-partitioning structures (SPSs)*. A BVH on a set S of objects is a tree structure whose leaves are in a one-to-one correspondence with the objects in S and where each node ν stores some bounding volume—usually the bounding box—of the objects corresponding to the leaves in the subtree of ν. Note that the size of a BVH is linear in the number of objects stored in it. An SPS is based on a partitioning of the space into cells. With each cell, (pointers to) all objects intersecting that cell are stored. Moreover, there is some search structure that

*Department of Computer Science, TU Eindhoven, P.O.Box 513, 5600 MB Eindhoven, the Netherlands.

†MdB is supported by the Netherlands Organisation for Scientific Research (N.W.O.) under project no. 639.023.301.

‡MS is supported by the Netherlands Organisation for Scientific Research (N.W.O.) under project no. 612.065.203.

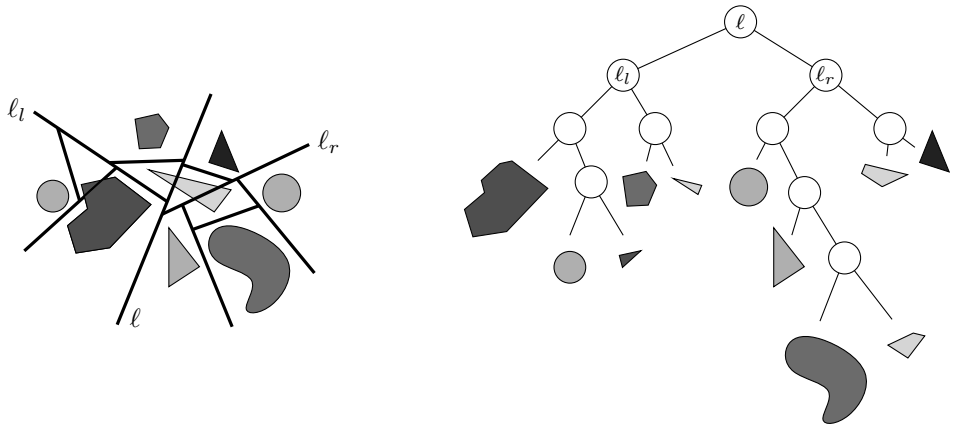

Figure 1: A BSP in the plane, and the corresponding tree. With the leaves we have shown the fragments inside the corresponding cell, although normally just a pointer to the object is stored.

makes it possible to quickly find the cells in the partitioning intersecting a query range. Often the partitioning is done in a hierarchical fashion, and the search structure is a tree whose leaves are in one-to-one correspondence with the cells in the partitioning. Because objects can potentially intersect many cells, the size of an SPS is not necessarily linear. Both BVHs and SPSs can in principle store any type of input object, they can be used to perform range searching with any type of query range—this implies they can also be used for point location—and they can be used to do nearest neighbor searching. The main challenge is to construct the BVH or SPS in such a way that queries are answered efficiently and, for SPSs, that their size is small. In this paper we study this problem for SPSs or, more precisely, for binary space partition trees.

A *binary space partition tree*, or *BSP tree*, is an SPS where the subdivision of the underlying space is done in a hierarchical fashion using hyperplanes (that is, lines in case the space is 2D, planes in 3D, etc.) The hierarchical subdivision process usually continues until each cell contains at most one (or maybe a small number of) input object(s)—see Fig. 1 for an example in 2D. BSP trees are used for many purposes. For example, they are used for range searching [3], for hidden surface removal with the painter's algorithm [19], for shadow generation [15], for set operations on polyhedra [23, 31], for visibility preprocessing for interactive walkthroughs [28], for cell decomposition methods in motion planning [9], and for surface approximation [6].

In some applications—hidden-surface removal is a typical example—the efficiency is determined by the size of the BSP tree, as the application needs to traverse the whole tree. Hence, several researchers have proposed algorithms to construct small BSP trees in various different settings [4, 5, 11, 12, 25, 26, 30]. For instance, Paterson and Yao [25] proved that any set of n disjoint line segments in the plane admits a BSP tree of size $O(n \log n)$. Recently Tóth [29] showed that this is close to optimal by exhibiting a set of n segments for which any BSP tree must have size $\Omega(n \log n / \log \log n)$. Paterson and Yao also proved that there are sets of n disjoint triangles in $\mathbb{R}^3$ for which any BSP tree must have quadratic size, and they gave a construction algorithm that achieves this bound. Since a quadratic-size BSP tree is useless in most practical applications, de Berg [11] studied BSP trees for so-called uncluttered scenes. He proved that uncluttered scenes admit a BSP tree of linear size, in any fixed dimension. Unfortunately, his BSP tree can have linear depth, so it is not efficient for range searching or point location. However, by constructing additional data structures that help to speed up the search in the BSP tree, he showed it is possible to perform point location in $O(\log n)$ time in uncluttered scenes. Range searching in low-density scenes can be done as well in $O(\log n)$ time—again using some additional structures—but only if the diameter of the query range is about the same as the diameter of the smallest object in the scene. Surprisingly, we do not know of any papers that investigate the efficiency of BSP trees, without any additional data structures as in [11], for range searching on non-point objects.

Approximate range searching. Developing a multi-functional geometric data structure—one that can store any type of object and can do range searching with any type of query range—that provably has good performance seems quite hard, if not impossible. As it turns out, however, such results can be achieved if one is willing to settle for *ε-approximate range searching*, as introduced by Arya and Mount [8]. Here one considers, for a parameter $\varepsilon > 0$, the ε-extended query range Q_ε, which is the set of points lying at distance at most $\epsilon \cdot \mathrm{diam}(Q)$ from Q, where $\mathrm{diam}(Q)$ is the diameter of Q. Objects intersecting Q must be reported, while objects intersecting Q_ε (but not Q) may or may not be reported; objects outside Q_ε are not allowed to be reported. In practice, one would expect that for small values of ε, not too many extra objects are reported.

Arya and Mount [8] showed that approximate range searching can be done very efficiently when the input is a set of n points in $\mathbb{R}^d$. Their BBD-tree can answer range queries with any constant-complexity convex[1] query range in $O(\log n + (1/\varepsilon)^{d-1} + k_\varepsilon)$ time, where k_ε is the number of points inside Q_ε. Later Duncan *et al.* [17, 18] achieved the same results using so-called BAR-trees, which are a special type of BSP trees that use splitting hyperplanes with only a fixed number of orientations. In fact, as observed by Haverkort *et al.* [22], the parameter ε is only needed in the analysis and not in the query algorithm, which can simply report only the objects intersecting Q. This means that a query can be answered in time $O(\log n + \min_{\varepsilon>0}\{(1/\varepsilon)^{d-1} + k_\varepsilon\})$.

Our main objective is to obtain similar results for more general input objects than points, thus providing a truly multi-functional geometric data structure. Some results in this direction have been obtained recently by Agarwal *et al.* [2] and Haverkort *et al.* [22]. Agarwal *et al.* showed how to construct a boxtree—that is, a BVH that uses axis-aligned bounding boxes as bounding volumes—for any set S of n input boxes in $\mathbb{R}^d$, such that a range query with a box as query can be answered in time $\Theta(n^{1-1/d} + k)$, where k is the number of input boxes intersecting the query box. They also showed that this is optimal in the worst case, even if the input boxes are disjoint. Unfortunately, this result is rather limited: even though a boxtree can store any type of object and query with any range, there is only a performance guarantee for a very specific setting: orthogonal range searching in a set of boxes. Moreover, the bound is fairly high. Haverkort *et al.* therefore studied approximate range searching. They presented a BVH that can answer range queries on a set of boxes in $\mathbb{R}^3$, for any constant-complexity query range, in time $O((\lambda/\epsilon)\log^4 n + k_\epsilon)$, where λ is the *density* of the scene.[2] The density of a set of objects is defined as the smallest number λ such that any ball B intersects at most λ objects whose minimal enclosing ball has radius at least $radius(B)$ [13]. When the input objects are disjoint and fat, the density λ is a constant (depending on the fatness). This result is more general than the result of Agarwal *et al.* [2], as it holds for any range, but still there is no performance guarantee for input objects other than boxes. Indeed, even for disjoint fat objects, the query time can be linear for a point location query, because the bounding boxes of the input boxes may all contain a common point.

Our results. In this paper we show that it is possible to construct BSP trees for sets of disjoint segments in the plane, and for low-density scenes in any dimension, whose query time for approximate range searching is as good, or almost as good, as the best known bounds for point data. More precisely, our results are as follows.

In Section 3 we study BSP trees for a set of n disjoint line segments in the plane. We give a general technique to convert a BSP tree $\mathcal{T}_P$ for a set of points to a BSP tree for a set of disjoint line segments, such that the size of the BSP tree is $O(n \cdot \mathrm{depth}(\mathcal{T}_P))$, and the time for range searching remains almost the same. By combining this result with the BAR-tree of Duncan *et al.* [18], we obtain a BSP tree that can answer range queries with any convex constant-complexity query range in time $O(\min_{\varepsilon>0}(1/\varepsilon + k_\varepsilon)\log n)$, where k_ε is the number of segments intersecting the ε-extended query range. This is the first result on approximate range searching for disjoint segments in the plane. This result can be extended to disjoint constant-complexity curves in the plane, if we allow the BSP to use splitting curves (which will be along the input curves) in the

[1] For non-convex ranges, one needs to replace the factor $1/\varepsilon^{d-1}$ by $1/\varepsilon^d$. This holds for all results mentioned below, including ours, so we do not mention this extension in the sequel.

[2] In fact, the result is more general: the bound holds if the so-called slicing number of the scene is λ.

partitioning.

In Section 4 we consider low-density scenes. We prove that any scene of constant density in $\mathbb{R}^d$ admits a BSP of linear size, such that range-searching queries with arbitrary convex ranges can be answered in time $O(\log n + \min_{\varepsilon>0}\{(1/\varepsilon^{d-1}) + k_\varepsilon\})$. (The dependency on the density λ is linear.) This result is more general than the result of Haverkort *et al.* [22], as they only have a performance guarantee for boxes as input. Moreover, our time bound is better by several logarithmic factors, and our result holds in any dimension.

In Section 5 we show that both the general technique for constructing a BSP on a set of disjoint line segments and the BSP for low-density scenes can be adapted to an I/O-efficient variant. The first I/O-efficient BSP answers a convex constant-complexity range query in $O(\min_{\varepsilon>0}(1/\varepsilon + k_\varepsilon)\log_B n)$ operations and the latter in $O((1/B)\min_{\varepsilon>0}\{(\varepsilon^{1-d}) + k_\varepsilon\} + \log_B n)$ operations.

2 Preliminaries

In this section we briefly introduce some terminology and notation that we will use throughout the paper.

A BSP tree for a set S of n objects in $\mathbb{R}^d$ is a binary tree $\mathcal{T}$ with the following properties.

- Every (internal or leaf) node ν corresponds to a subset region(ν) of $\mathbb{R}^d$, which we call the *region* of ν. These regions are not stored with the nodes. When ν is a leaf node, we sometimes refer to region(ν) as a *cell*. Thus the term region can be used both for internal nodes and leaf nodes, but the term cell is strictly reserved for regions of leaf nodes. The root node root($\mathcal{T}$) corresponds to $\mathbb{R}^d$.

- Every internal node ν stores a hyperplane $h(\nu)$, which we call the *splitting hyperplane* (*splitting line* when $d = 2$) of ν. The left child of ν then corresponds to region(ν) $\cap\, h(\nu)^-$, where $h(\nu)^-$ denotes the half-space below $h(\nu)$, and the right child corresponds to region(ν)$\cap h(\nu)^+$, where $h(\nu)^+$ is the half-space above $h(\nu)$.

  A node ν stores, besides the splitting hyperplane $h(\nu)$, a list $\mathcal{L}(\nu)$ with all objects contained in $h(\nu)$ that intersect region(ν). Observe that this list will always be empty when the objects in S are d-dimensional.

- Every leaf node μ stores a list $\mathcal{L}(\mu)$ of all objects in S intersecting the interior of region(μ).

Note that we do not place a bound on the number of objects stored with a leaf. In the BSP trees discussed in this paper, however, this number will be constant. When the input objects are disjoint and convex, this means that we could replace each leaf by a constant-size subtree such that each leaf stores only one object, without affecting the asymptotic bounds on storage and query time.

The *size* of a BSP tree is defined as the total number of nodes plus the total size of the lists $\mathcal{L}(\nu)$ over all (internal and leaf) nodes ν in $\mathcal{T}$. For a node ν in a tree $\mathcal{T}$, we use $\mathcal{T}(\nu)$ to denote the subtree of $\mathcal{T}$ rooted at ν, and we use depth($\mathcal{T}$) to denote the depth of $\mathcal{T}$.

We assume that all our input objects, as well as the query range have constant complexity. This means that we can compute the intersection of an object with a line or with the query range in $O(1)$ time.

3 BSPs for segments in the plane

Let S be a set of n disjoint line segments in the plane. In this section we describe a general technique to construct a BSP for S, based on a BSP on the endpoints of S. The technique uses a segment-tree like approach similar to, but more general than, the deterministic BSP construction of Paterson and Yao [25]. The range-searching structure of Overmars *et al.* [24] also uses similar ideas, except that they store so-called long segments—see below—in an associated structure, so they do not construct a BSP for the segments. The main extra complication we face compared to these previous approaches is that we must ensure that we only work with the relevant portions of

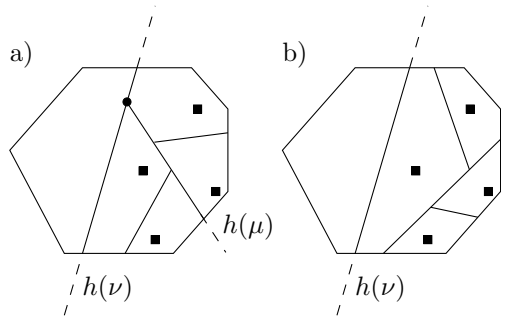

Figure 2: Illustration of the pruning strategy. The black squares indicate endpoints of input segments. a) There is a T-junction on $h(\nu)$: a splitting line in the subtree $h(\mu)$ ends on $h(\nu)$. Pruning $h(\nu)$ would partition the empty part of the region, which might have a negative effect on the query time. b) There is no T-junction on $h(\nu)$, and can therefore be pruned.

the given tree during the recursive construction, and prune away irrelevant portions. The pruning has to be done carefully, however, because too much pruning can have a negative effect on the query time. The following theorem summarizes the result of this section. It will be proved in Sections 3.1–3.3. In Section 3.4 we will discuss an application of this result to approximate range searching.

Theorem 3.1 *Let $\mathcal{R}$ be a family of constant-complexity query ranges in $\mathbb{R}^2$. Suppose that for any set P of n points in $\mathbb{R}^2$, there is a BSP tree $\mathcal{T}_P$ of linear size, where each leaf stores at most one point from P, with the following property: any query Q with a range from $\mathcal{R}$ intersects at most $v(\mathcal{T}_P, Q)$ cells in the BSP subdivision. Then for any set S of n disjoint segments in $\mathbb{R}^2$, there is a BSP tree $\mathcal{T}_S$ such that*

(i) the depth of $\mathcal{T}_S$ is $O(depth(\mathcal{T}_P))$

(ii) the size of $\mathcal{T}_S$ is $O(n \cdot depth(\mathcal{T}_P))$

(iii) any query with a range Q from $\mathcal{R}$ visits at most $O((v(\mathcal{T}_P, Q) + k) \cdot depth(\mathcal{T}_P))$ nodes from $\mathcal{T}_S$, where k is the number of segments intersecting Q.

The BSP tree $\mathcal{T}_S$ can be constructed in $O(n \cdot depth(\mathcal{T}_P))$ time.

3.1 The construction

Let P be the set of $2n$ endpoints of the segments in S, and let $\mathcal{T}_P$ be a BSP tree for P. We assume that $\mathcal{T}_P$ has size $O(n)$, and that the leaves of $\mathcal{T}_P$ store at most one point from P. Below we describe the global construction of the BSP tree for S. Some details of the construction will be omitted here; they will be described when we discuss the time needed for the construction.

The BSP tree $\mathcal{T}_S$ for S is constructed recursively from $\mathcal{T}_P$, as follows. Let ν be a node in $\mathcal{T}_P$. We call a segment $s \in S$ *short* at ν if region(ν) contains an endpoint of s. A segment s is called *long* at ν if (i) s intersects the interior of region(ν), and (ii) s is short at parent(ν) but not at ν. In a recursive call there are two parameters: a node $\nu \in \mathcal{T}_P$ and a subset $S^* \subset S$, clipped to lie within region(ν). The recursive call will construct a BSP tree $\mathcal{T}_{S^*}$ for S^* based on $\mathcal{T}_P(\nu)$. Initially, $\nu = \text{root}(\mathcal{T}_P)$ and $S^* = S$. The recursion stops when S^* is empty, in which case $\mathcal{T}_{S^*}$ is a single leaf.

We will make sure that during the recursive calls we know for each segment (or rather, fragment) in S^* which of its endpoints lie on the boundary of region(ν), if any. This means we also know for each segment whether it is long or short. This information can be maintained during the recursive calls without asymptotic overhead.

Let $L \subset S^*$ be the set of segments from S^* that are long at ν. The recursive call is handled as follows.

Case 1: L is empty.

We first compute $S_l = S^* \cap h(\nu)^-$ and $S_r = S^* \cap h(\nu)^+$. If both S_l and S_r are non-empty, we create a root node for $\mathcal{T}_{S^*}$ which stores $h(\nu)$ as its splitting line. We then recurse on the left child of ν with S_l to construct the left subtree of the root, and on the right child of ν with S_r to construct the right subtree of the root.

If one of S_l and S_r is empty, it seems the splitting line $h(\nu)$ is useless in $\mathcal{T}_{S^*}$ and can therefore be omitted in $\mathcal{T}_{S^*}$. We have to be careful, however, that we do not increase the query time: the removal of $h(\nu)$ can cause other splitting lines, which used to end on $h(\nu)$, to extend further. Hence, we proceed as follows. Define a *T-junction*, see Fig. 2, to be a vertex of the original BSP subdivision induced by $\mathcal{T}_P$; in other words, the T-junctions are the endpoints of the segments $h(\mu) \cap \text{region}(\mu)$, for nodes μ in $\mathcal{T}_P$. To decide whether or not to use $h(\nu)$, we check if $h(\nu) \cap R$ contains a T-junction in its interior, where R is the region that corresponds to the root of $\mathcal{T}_{S^*}$. If this is the case, we do not prune: the root node of $\mathcal{T}_{S^*}$ will store the splitting line $h(\nu)$, one of its subtrees will be empty, and the other subtree will be constructed recursively on the non-empty subset. If $h(\nu) \cap R$ does not contain a T-junction, however, we prune: the tree $\mathcal{T}_{S^*}$ will be the tree we get when we recurse on the non-empty subset, and there will be no node in $\mathcal{T}_{S^*}$ that stores $h(\nu)$.

Case 2: L is not empty.

Now the long segments partition R into $m := |L| + 1$ regions, $R_1, \ldots, R_m$. We take the following steps.

(i) We split $S^* \setminus L$ into m subsets $S_1^*, \ldots, S_m^*$, where S_i^* contains the segments from S^* lying inside R_i.

(ii) We construct a binary tree $\mathcal{T}$ with $m - 1$ internal nodes whose splitting lines are the lines containing the long segments. We call these *free splits* because they do not cause any fragmentation. The leaves of $\mathcal{T}$ correspond to the regions R_i, and will become the roots of the subtrees to be created for the sets S_i^*. To keep the overall depth of the tree bounded by $O(\text{depth}(\mathcal{T}_P))$, we make $\mathcal{T}$ weight-balanced, where the weights correspond to the sizes of the sets S_i^*, as in the trapezoid method for point location [27]. The tree $\mathcal{T}$ is constructed as follows. Let $\ell_i \in L$ separate the regions R_i and R_{i+1}. If $\sum_{i=1}^{r-1} |S_i^*| = 0$ simply build a binary tree on the long segments, otherwise determine the integer r such that $\sum_{i=1}^{r-1} |S_i^*| < |S^* \setminus L|/2$ and $\sum_{i=1}^{r} |S_i^*| \geq |S^* \setminus L|/2$. ℓ_{r-1} is then stored at ν, the root of $\mathcal{T}$, and ℓ_r is stored at the root of the right child μ, see figure 3. The left child of ν, τ_1, and the right child of μ, τ_2 are constructed recursively. Note that both region(τ_1) and region(τ_2) contain each less than $|S^* \setminus L|/2$ short segments. The tree $\mathcal{T}_{S^*}$ then consists of the tree $\mathcal{T}$, with, for every $1 \leqslant i \leqslant m$, the leaf of $\mathcal{T}$ corresponding to R_i replaced by a subtree for S_i^*. More precisely, each subtree T_i is constructed using a recursive call with node ν and S_i^* as parameters.

Next we prove bounds on the size and depth of the tree $\mathcal{T}_S$.

Lemma 3.1 *The size of $\mathcal{T}_S$ is $O(n \cdot \text{depth}(\mathcal{T}_P))$.*

Proof. The tree $\mathcal{T}_S$ has two types of nodes: *partition nodes*, which store splitting lines from nodes in $\mathcal{T}_P$, and *segment nodes*, which store free splits along a long segment.

A segment can only be cut when it has an endpoint in a region of a node in $\mathcal{T}_P$, so it will be cut into $O(\text{depth}(\mathcal{T}_P))$ fragments. Hence, the number of segment nodes is $O(n \cdot \text{depth}(\mathcal{T}_P))$.

For a partition node μ, we either have that both subtrees contain a segment fragment, or $h(\mu) \cap \text{region}(\mu)$ contains a T-junction. The number of partition nodes of the former type is bounded by $O(n \cdot \text{depth}(\mathcal{T}_P))$ because the number of segment fragments is $O(n \cdot \text{depth}(\mathcal{T}_P))$. The number of partition nodes of the latter type is bounded by $O(n)$ due to the size of $\mathcal{T}_P$.

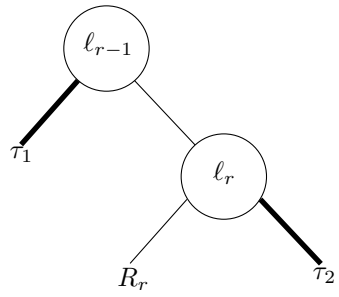

Figure 3: Construction of the weight balanced tree $\mathcal{T}$. Splitting lines ℓ_{r-1} and ℓ_r are two long segments which are stored at respectively the root of $\mathcal{T}$ and at the root of the right child of $\mathcal{T}$, and τ_1 and τ_2 are subtrees where long segments might have to be inserted. The regions associated with τ_1 and τ_2 contain each less than half of the short segments in region($\mathcal{T}$). The root node for R_r must be a partition node.

It follows that the total number of nodes is $O(n \cdot \text{depth}(\mathcal{T}_P))$, as claimed. □

Lemma 3.2 *The depth of $\mathcal{T}_S$ is $O(depth(\mathcal{T}_P))$.*

Proof. This follows more or less from standard arguments [27], but not directly, since our process is not fully recursive in the sense that we must use the given tree $\mathcal{T}_P$. This means that we may "inherit" splitting lines in a subtree that are good splitters in $\mathcal{T}_P$ but not in the subtree. Hence, we give a short proof.

As in the proof of the previous lemma, we distinguish between partition nodes and segment nodes. The number of partition nodes on any path in $\mathcal{T}_S$ is bounded by depth($\mathcal{T}_P$). It remains to bound the number of segment nodes on any path. For this, we will look at the process of replacing a multi-way node with a binary subtree, and establish an invariant that establishes the bound. Call an edge in $\mathcal{T}_S$ from a node ν to a node μ *black* when the number of segments in region(μ) is at most half the number of segments in region(parent(ν)). An invariant that is maintained when we replace a multi-way node by a subtree is then: there cannot be more than two consecutive segment nodes on any path in $\mathcal{T}_S$ without a black edge in between—see Fig. 3 where the black edges are drawn fat. Note that the subtree for R_r must have a partition node as root node. Between two consecutive partition nodes on a path either it crosses one or more black edges or exactly two segment nodes. Since by the definition of a black edge there can be no more than $O(\log n)$ black edges on any path and the number of partition nodes on any path is bounded by depth($\mathcal{T}_P$), the bound on the number of segment nodes is $O(\text{depth}(\mathcal{T}_P))$. □

3.2 Analysis of the query time

A query in the BSP tree with a query range Q visits all nodes ν such that Q intersects region(ν). Next we bound the number of such nodes, in terms of the number of visited nodes in $\mathcal{T}_P$ when we would query $\mathcal{T}_P$ with Q.

Lemma 3.3 *Let Q be a query range, and let $v(\mathcal{T}_P, Q)$ be the number of visited leaves in $\mathcal{T}_P$ when we query $\mathcal{T}_P$ with the range Q. Then the number of visited nodes in $\mathcal{T}_S$ when we query with Q is bounded by $O((v(\mathcal{T}_P, Q) + k) \cdot depth(\mathcal{T}_P))$, where k is the number of segments intersecting Q.*

Proof. We distinguish two categories of visited nodes: nodes ν such that region(ν) is intersected by ∂Q (the boundary of Q), and nodes ν such that region(ν) $\subset Q$.

We first bound the number of leaves of the first category, that is, the number of cells intersected by ∂Q. This number is bounded by the number of intersections of ∂Q and cell boundaries (except for the trivial case where Q lies completely inside a single region). A cell boundary is composed of pieces of splitting lines that were already present in $\mathcal{T}_P$ and fragments of segments in S. Because we made sure that the pruning step in our construction did not cause splitting 'lines' to be extended, the number of pieces of splitting lines in $\mathcal{T}_S$ intersected by ∂Q is not more than the number of cells of the subdivision induced by $\mathcal{T}_P$ that are intersected by ∂Q. Furthermore, the number of segments in S intersected by ∂Q is $O(k)$. Hence, the total number of leaf cells intersected by ∂Q is $O(v(\mathcal{T}_P, Q) + k)$. Because the depth of $\mathcal{T}_S$ is $O(\text{depth}(\mathcal{T}_P))$, the total number of nodes in the first category is $O((v(\mathcal{T}_P, Q) + k) \cdot \text{depth}(\mathcal{T}_P))$.

Nodes in the second category are organized in subtrees rooted at nodes ν such that $\text{region}(\nu) \subset Q$ but $\text{region}(\text{parent}(\nu)) \not\subset Q$. Let $N(Q)$ be the collection of these roots. Note that the regions of the nodes in $N(Q)$ are disjoint. For a node $\nu \in N(Q)$, let $k_s(\nu)$ denote the number of segments that are short at ν, and $k_l(\nu)$ the number of segments that are long at ν. Then the size of the subtree $\mathcal{T}_S(\nu)$ is $O(k_l(\nu) + k_s(\nu) \cdot \text{depth}(\mathcal{T}_S(\nu))$. Hence, the total number of nodes in the subtrees $\mathcal{T}_S(\nu)$ rooted at the nodes $\nu \in N(Q)$ is bounded by

$$\sum_{\nu} O(k_l(\nu) + k_s(\nu) \cdot \text{depth}(\mathcal{T}_S(\nu)) = O(\sum_{\nu} k_l(\nu)) + O(\text{depth}(\mathcal{T}_P) \cdot \sum_{\nu} k_s(\nu)).$$

The first term is bounded by $O(k \cdot \text{depth}(\mathcal{T}_P))$, because a segment is long at $O(\text{depth}(\mathcal{T}_P))$ nodes. The second term is bounded by $O(k \cdot \text{depth}(\mathcal{T}_P))$, because the regions of the nodes in $N(Q)$ are disjoint which implies that a segment will be short at at most two such nodes (one for each endpoint).

Adding up the bounds for the first and the second category, we get the desired bound. □

A bound on the number of visited nodes does not directly give a bound on the query time, because at a node ν we have to test whether Q intersects the regions associated with the children of ν. These regions are not stored at the nodes and, moreover, they need not have constant complexity.

One way to handle this is to maintain the vertices of the regions $\text{region}(\nu)$ of all visited nodes in a red-black tree in order along the boundary. From $\text{region}(\nu)$ and the splitting line $h(\nu)$, we can then compute the regions of the left and right child of ν in $O(\log |\text{region}(\nu)|)$ time, where $|\text{region}(\nu)|$ denotes the number of vertices of $\text{region}(\nu)$. If Q is polygonal and of constant complexity, we can then also test whether Q intersects $\text{region}(\nu)$ in $O(\log |\text{region}(\nu)|)$ time; if Q has constant complexity but is not polygonal we can do the test in $O(|\text{region}(\nu)|)$ time.

An alternative is to store with each node ν some additional information. In particular, we store two lines that are tangent to $\text{region}(\nu)$ at the two endpoints of $h(\nu) \cap \text{region}(\nu)$. When we come to a node ν such that $\text{region}(\nu)$ intersects Q, we can then proceed as follows. If Q intersects $h(\nu) \cap \text{region}(\nu)$—this segment can be computed from $h(\nu)$ and the two additional lines stored at ν—then we recurse on both children. Otherwise we can use the two additional lines to decide which child need not be visited, assuming that Q is convex. With this approach, the query time for convex ranges will be asymptotically the same as the number of visited nodes. The storage requirements of the BSP tree will increase by about a factor three, however.

3.3 An efficient construction algorithm

Next we discuss how the construction described in Section 3.1 can be performed efficiently.

To do the construction efficiently, we need to maintain some extra information during the recursive calls. First of all, we need to know for the partition lines whether they contain T-junctions. To this end we compute all T-junctions in $\mathcal{T}_P$ during preprocessing in $O(n \cdot \text{depth}(\mathcal{T}_P))$ time, and we maintain which T-junctions lie in the current region during the recursive calls. We can do the maintenance by spending time linear in the number of T-junctions during each recursive call, giving $O(n \cdot \text{depth}(\mathcal{T}_P))$ time in total.

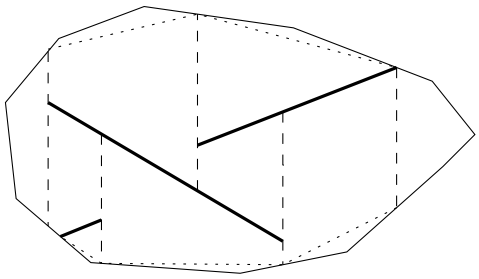

Figure 4: Example of a vertical decomposition of the segments clipped to a region. The boundary of the trapezoids inside the region are drawn dotted.

Secondly, we maintain a vertical decomposition of the segments in S^*, where S^* is the current subset we are dealing with. We clip this vertical decomposition to lie within the current region. Note that the region boundary does not play a role in the vertical decomposition besides that it is used to shorten the vertical extensions; in particular the boundary is replaced by a single segment between intersection points of the vertical extensions of segments in the region and its boundary, see Fig. 4. Computing the initial vertical decomposition takes $O(n \log n)$ time. It will be stored as its dual. The dual of the vertical decomposition is a graph $\mathcal{G}(S^*)$ whose nodes correspond to trapezoids in the vertical decomposition. There are arcs between neighboring trapezoids. Maintaining the dual of the vertical decomposition during a recursive call boils down to splitting the dual with a line, which we can do in linear time. Hence, the total time for maintaining the vertical decompositions is $O(\sum(|S^*|)$, where the sum is over the sets S^* arising in all recursive calls.

We consider the two cases of the algorithm:

Case 1: L is empty.
Computing S_l and S_r takes $O(|S^*|)$ time. Since any segment in S^* has an endpoint in region(ν) in this case, we account for this time by charging $O(1)$ to each endpoint in $\mathcal{T}_P(\nu)$. This way an endpoint will be charged $O(\text{depth}(\mathcal{T}_P))$ times, so the overall time will be $O(n \cdot \text{depth}(\mathcal{T}_P))$.

The other time consuming part is the test whether $h(\nu) \cap R$ contains a T-junction. Recall that we maintain the T-junctions lying inside the current region R. It remains to check whether any one of them lies on $h(\nu)$, which can be done brute-force in time linear in the number of T-junctions, which does not increase the time bound asymptotically.

We conclude that the total time taken to perform this case over all recursive calls is $O(n \cdot \text{depth}(\mathcal{T}_P))$.

Case 2: L is not empty.

(i) We have to determine the order of the long segments on the splitting line $h(\text{parent}(\nu))$, and we have to distribute the short segments over the regions induced by the long segments. By traversing $\mathcal{G}(S^*)$ we know the order on the long segments. Delete all arcs crossing the long segments and label their nodes by the appropriate region. Now we compute the connected components which corresponds exactly to the regions induced by the long segments. Since at least one node in each component is labelled by the region in which it is contained, we obtained for each region the set of short segments and the graph $\mathcal{G}(S_i^*)$. This whole procedure takes linear time.

(ii) Construct the weight balanced tree. The previous step gives us the weights needed to organize the long segments into a weight-balanced tree. It can be argued [27] that then the construction of the weight-balanced trees over all recursive calls can be done in $O(n \cdot \text{depth}(\mathcal{T}_P))$ in total.

We conclude that the time for this case over all recursive calls is bounded by $O(\sum(|S^*|)+n\cdot \text{depth}(\mathcal{T}_P))$, where the sum is over all recursive calls. This is bounded by $O(n\cdot\text{depth}(\mathcal{T}_P))$.

The following lemma summarizes the discussion above, and finishes the proof of Theorem 3.1.

Lemma 3.4 *The construction from the BSP tree $\mathcal{T}_S$ from the tree $\mathcal{T}_P$ can be done in $O(n\cdot depth(\mathcal{T}_P))$ time.*

3.4 Application to approximate range searching

Several of the known data structures for range searching in point sets are actually BSP trees. For example, if we use the partition tree of Haussler en Welzl [21] as underlying structure we can get a BSP on a set of n disjoint line segments whose query time is $O(n^{2/3+\varepsilon}+k\log n)$. Here we focus on the application using BAR-trees [18], as it gives good bounds for approximate range searching for line segments in the plane. The BAR-tree is a BSP tree, where all splitting lines have orientations that come from a fixed set of predefined possibilities. In the plane for a corner-cut BAR-tree e.g., the orientations that are used are horizontal, vertical, and the two diagonal directions (45° and 135°). Hence, the regions in a BAR-tree have constant complexity. This also holds for the regions we get when we transform a BAR-tree on the endpoints of a set of segments to a BSP tree for the segments; such regions can only be twice as complex as the regions in a BAR-tree.

The main strength of a BAR-tree is that it produces regions with bounded aspect ratio: the smallest enclosing circle of a region is only a constant times larger than the largest inscribed circle. This makes that a BAR-tree has excellent query time for approximate range queries—see Duncan's thesis [17] for details. In the plane one can construct BAR-trees with logarithmic depth, such that the number of leaves visited by a query with a convex query range Q is bounded by $O((1/\varepsilon)+k_\varepsilon)$, where k_ε is the number of points inside the extended query range Q_ε.

By applying theorem 3.1, we can thus obtain a BSP for segments with a query time of $O((\varepsilon^{-1}+k_\varepsilon)\log n)$. We can even extend this result to disjoint constant-complexity curves in the plane, if we allow the BSP to use splitting curves in the partitioning. For the construction algorithm to work we have to ensure that any splitting line can intersect a curve only once. We do this by cutting the curve at every point where the orientation of its tangent is one of the possible orientations of the splitting lines. These pieces are then used in the construction of $\mathcal{T}_S$. Since the curves have constant-complexity and BAR-tree splitting lines have only four possible orientations, a curve is cut at most into a constant number of pieces.

Corollary 3.1 *Let S be a set of n disjoint constant-complexity curves in $\mathbb{R}^2$. In $O(n\log n)$ time one can construct a BSP tree for S of size $O(n\log n)$ and depth $O(\log n)$ such that a range query with a constant-complexity convex range can be answered in time $O(\min_{\varepsilon>0}\{(1/\varepsilon)\log n+k_\varepsilon\log n\})$, where k_ε is the number of curves intersecting the extended query range Q_ε.*

4 BSPs for low-density scenes

Let S be a set of n objects in $\mathbb{R}^d$. For an object o, we use $\rho(o)$ to denote the radius of the smallest enclosing ball of o. Recall from the introduction that the *density* of a set S is the smallest number λ such that the following holds: any ball B is intersected by at most λ objects $o\in S$ with $\rho(o)\geqslant\rho(B)$ [13]. If S has density λ, we call S a λ-low-density scene. In this section we show how to construct a BSP tree for S that has linear size and very good performance for approximate range searching if the density of S is constant. Our method combines ideas from de Berg [11] with the BAR-tree of Duncan *et al.* [18]. We will call this BSP an *object BAR-tree*, or oBAR-tree for short. The following theorem summarizes our result. Its proof follows directly from the results in Subsections 4.1–4.3.

Theorem 4.1 *Let S be a λ-low-density scene consisting of n objects in $\mathbb{R}^d$. There exists a BSP tree $\mathcal{T}_S$ for S such that*

(i) *the depth is* $O(\log n)$

(ii) *the size is* $O(\lambda n)$

(iii) *a query range with a convex range* Q *takes* $O(\log n + \lambda \cdot \min_{\varepsilon>0}\{(1/\varepsilon) + k_\varepsilon\})$ *time, where* k_ε *is the number of objects intersecting the extended query range* Q_ε.

The BSP tree can be constructed in $O(\lambda n \log n)$ *time.*

4.1 The construction

Our overall strategy, also used by de Berg [11], is to compute a suitable set of points that will guide the construction of the BSP tree. Unlike in [11], however, we cannot use the bounding-box vertices of the objects in S for this, because that does not work in combination with a BAR-tree. What we need is a set G of points with the following property: any cell in a BAR-tree that does not contain a point from G in its interior is intersected by at most κ objects from S, for some constant κ. We call such a set G a *κ-guarding set* [10] against BAR-tree cells, and we call the points in G *guards*.

The algorithm is as follows.

1. Construct a κ-guarding set G for S, as explained below in Subsection 4.3. The construction of the guarding set is done by generating $O(1)$ guards for each object $o \in S$, so that the guards created for any subset of the objects will form a κ-guarding set for that subset. We will use object(g) to denote the object for which a guard $g \in G$ was created.

2. Create a BAR-tree $\mathcal{T}$ on the set G using the algorithm of Duncan *et al.* [18], with the following adaptation: whenever a recursive call is made with a subset $G^* \subset G$ in a region R, we delete all guards g from G^* for which object(g) does not intersect R. This pruning step, which was not needed in [11], is essential to guarantee a bound on the query time.

3. Search with each object $o \in S$ in $\mathcal{T}$ to determine which leaf cells are intersected by o. Store with each leaf the set of all intersected objects. Let $\mathcal{T}_S$ be the resulting BSP tree.

Lemma 4.1 *The depth of* $\mathcal{T}_S$ *is* $O(\log |G|)$*, its size is* $O(\kappa \cdot |G|)$*, and it can be constructed in* $O(\kappa \cdot |G| \log |G|)$ *time.*

Proof. The first two statements follow immediately from the construction and the bounds on BAR-trees. As for the construction time, the guarding set G can be constructed in linear time, and the construction of the BAR-tree takes $O(|G| \log |G|)$ time [18]; the pruning of guards whose objects do not intersect the current region R can also be done within this bound. The time to search with an object from S is $O(\log |G|)$ times the number of intersected leaf cells. Since there are $O(\kappa \cdot |G|)$ cell-object incidences, the total time is $O(\kappa \cdot |G| \log |G|)$. □

4.2 Analysis of the query time

A query in the BSP tree $\mathcal{T}_S$ is done by maintaining the regions of the visited nodes, as described in Section 3.4. The next lemma states the query time of our structure.

Lemma 4.2 *A range query in* $\mathcal{T}_S$ *with a constant-complexity convex query range* Q *takes* $O(\log n + \kappa \cdot \min_{\varepsilon>0}\{(1/\varepsilon) + k_\varepsilon\})$ *time, where* k_ε *is the number of objects intersecting the extended query range* Q_ε.

Proof. Fix some $\varepsilon > 0$. Since $\mathcal{T}_S$ is a BAR-tree, its regions have $O(1)$ complexity, so the total query time is linear in the number of visited nodes. As before, we distinguish two categories of visited nodes: nodes ν such that region(ν) is intersected by ∂Q (the boundary of Q), and nodes ν such that region$(\nu) \subset Q$.

From the analysis of the BAR-tree it follows that the number of leaves in the first category is $O(1/\varepsilon^{d-1})$ and that the number of internal nodes is $O(\log n + (1/\varepsilon^{d-1}))$. (This proof is based on a packing argument, and not influenced by our pruning of guards.) At each leaf we spend $O(\kappa)$ time to check the objects, so in total we spend $O(\log n + (\kappa/\varepsilon^{d-1}))$

Nodes in the second category are organized in subtrees rooted at nodes ν such that region$(\nu) \subset Q$ but region(parent$(\nu)) \not\subset Q$. Let $N(Q)$ be the collection of these roots. For a node $\nu \in N(Q)$, let $k(\nu)$ denote the number of objects in the subtree of ν. Because we delete guards of objects not intersecting a region during the recursive construction, $k(\nu)$ is linear in the number of guards in region(ν), which is linear in the number of nodes in the subtree. Since the regions of the nodes in $N(Q)$ are disjoint, an object has guards in only $O(1)$ regions. This means that the overall number of nodes in the second category is $O(k_\varepsilon)$. Since we have to check at most κ objects at each leaf, the total time spent in these nodes is $O(\kappa \cdot k_\varepsilon)$.

Since the analysis above holds for any $\varepsilon > 0$, the bound follows. □

4.3 Constructing the guarding set

De Berg *et al.* [10] proved that any set S that has a small guarding set against hypercubes also has a small guarding set against arbitrary fat ranges. Since the bounding-box vertices of a λ-low-density scene form a λ-guarding set against hypercubes, this implies that low-density scenes admit linear-size guarding sets against fat ranges, as stated more precisely in the following lemma.

Lemma 4.3 [10] *Let S be a λ-low-density scene consisting of n objects in $\mathbb{R}^d$. Then there is an $O(\lambda)$-guarding set for S of size $O(n/\beta)$ against β-fat ranges.*

Since BAR-tree cells are fat, this implies there exists a guarding set for λ-low-density scenes such that any BAR-tree cell without guards in its interior intersects at most $\kappa = O(\lambda)$ objects. Unfortunately the constants in the construction are large: $\kappa = 14^d\lambda$, and the dependency on d in the size is more than $2^{3d(d-1)}$. Using the special properties of (corner-cut) BAR-tree cells, we give a much smaller guarding set against BAR-tree cells for the planar case.

Now let S be a λ-low-density scene in $\mathbb{R}^2$. Our guarding set G has 12 guards for each object $o \in S$, which are generated as follows. Let σ be a bounding square of o of minimal size. For each edge e of σ, we place a square of the same size as σ that shares e but lies on the opposite side. The 12 guards for o are the vertices of the five squares—see Fig. 5.

Lemma 4.4 *Let C be a corner-cut BAR-tree cell that intersects an object o, but does not contain any guard from $G(o)$ in its interior. The radius of the largest inscribed circle L in C is at most $\sqrt{2}\rho(o)$.*

Proof. Assume without loss of generality that the bounding square σ used in the construction of $G(o)$ has edge length 1. This means that $\rho(o) \geqslant 1/2$, so we have to prove that L has a radius of at most $\frac{1}{2}\sqrt{2}$. Let ξ be the cross-shaped polygon having the guards of o as vertices, see Fig 5a).

If the center of L is contained in ξ, then the radius of L is at most $\frac{1}{2}\sqrt{2}$ since L would otherwise contain at least one guard of o. For the rest of this proof let the center of L be outside ξ. Since BAR-tree cells are convex, there is an edge e of ξ that it is intersected by two edges of C, c_1 and c_2, whose extensions l_1 and l_2 enclose L. Note that when l_1 and l_2 are not parallel they form with (a part of) e an isosceles right triangle, because a BAR-tree only uses splitting lines that are horizontal, vertical or diagonal. There are three possibilities, see Fig. 5c):

1. *l_1 and l_2 are parallel.* The radius of L is in this case at most $1/2$ since the distance between l_1 and l_2 is at most 1.

2. *l_1 and l_2 intersect inside ξ.* In order to intersect o the cell C also has to intersect an edge f of the bounding square containing o. Both f and e are edges of one of the squares attached to the bounding square. No isosceles right triangle can stick out of a square when there is an edge of the triangle which is completely on an edge of the square. Hence, this case can actually not occur, since we assume that C intersects o.

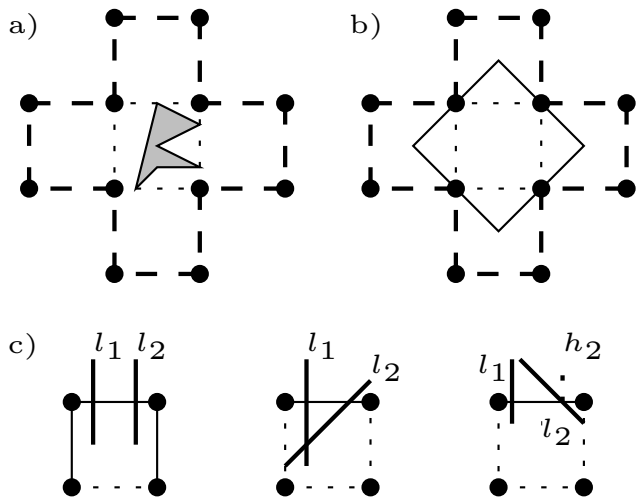

Figure 5: Creating a guarding set against BAR-tree cells. a) An object o with its guards. The bounding square $\sigma(o)$ of o is drawn with a dotted line. The union ξ of the squares placed at each edge of the bounding square is drawn with a dashed line. b) An example of a cell whose largest inscribed ball has radius $\sqrt{2}\rho(o)$. c) The three possible ways a cell can intersect ξ. For each possibility one of the dotted edges of the square is an edge of $\sigma(o)$.

3. *l_1 and l_2 intersect outside ξ.* Let h_1 and h_2 be the lines perpendicular to e intersecting e at the same point as l_1 resp. l_2. The isosceles right triangle formed by l_1, l_2 and e is contained between these two lines and therefore L is also contained between them. Since the distance between h_1 and h_2 is at most 1 the radius of L is at most $1/2$.

□

In Fig. 5b) a BAR-tree cell whose largest inscribed circle has radius $\sqrt{2}\rho(o)$ is given. So the given bound on the radius of the largest inscribed circle is tight.

Using the previous lemma and the bounded aspect ratio of BAR-tree cells, we can prove that G is a good guarding set.

Theorem 4.2 *Let S be a λ-low-density scene of n objects in $\mathbb{R}^2$. One can compute in $O(n)$ time a 72λ-guarding set of size $12n$ for S against corner-cut BAR-tree cells.*

Proof. Any cell C in a BAR-tree has bounded aspect ratio: for any $\alpha \geq 3d = 6$, we can ensure that $\rho(C)/\rho_I(C) \leqslant \alpha$, where $\rho(C)$ is the radius of the smallest enclosing circle of C and $\rho_I(C)$ is the radius of the largest inscribed circle of C [18]. Using the previous lemma, we get the following for any object o intersecting a cell C:

$$\rho(C) \leqslant \alpha \cdot \rho_I(C) \leqslant \alpha(\sqrt{2}\rho(o)) = 6\sqrt{2}\rho(o).$$

We now cover C by 72 disks of radius $\rho(C)/(6\sqrt{2})$. Any object o intersecting C will intersect one of them, and $\rho(o)$ will be at least the radius of that disk. Because S has density λ, this implies that the number of objects intersecting C is bounded by 72λ. □

This approach is generalizable to $\mathbb{R}^3$ but the constant in the construction gets quite large, so to obtain a guarding set in $\mathbb{R}^d$ for $d \geq 3$ we employ Lemma 4.3. The problem of finding a small guarding set for a λ-low-density scenes with a small constant remains open in $\mathbb{R}^d$ for $d \geq 3$.

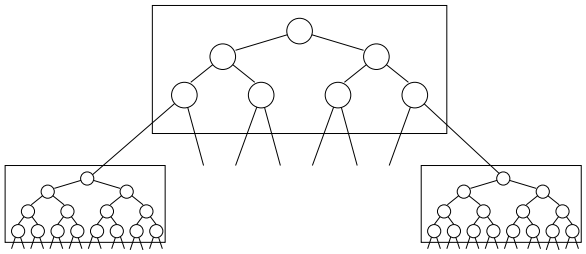

Figure 6: Grouping nodes of a BSP-tree into blocks.

5 External memory

A BSP of a large set of objects might be too large to fit into main memory in which case it needs to be stored in external memory. The access-time to external memory is far larger than the time needed for operations in main memory. Aggarwal and Vitter [1] introduced the *parallel disk model* which tries to model this behavior. In this model only the number of I/O-operations is considered; any operation in memory is free. In one I/O-operation a block consisting of B items per disk are read from or written to the external memory. We consider the case that there is only one disk. An overview of external memory algorithms and techniques in computational geometry can be found in [14]. As usual, we will assume that the main memory can store M items and $B \ll M \ll n$.

In Fig. 6 it is depicted how the nodes of a BSP $\mathcal{T}$ are grouped into blocks. The first B nodes considered in a breadth first search of $\mathcal{T}$ are grouped in one block b. The children of the nodes in b who are not in b themselves are roots of subtrees. These subtrees are recursively grouped together in blocks (each child in its own block). The B-BAR-tree of Duncan [17] uses the same blocking strategy.

The query algorithm in this I/O-efficient tree is almost the same as usual except when it wants to visit a node ν outside the current block. In that case the visit to that node is postponed by placing the block containing ν as a node in a queue. The algorithm continues searching in the current block and will visit the node only when the block containing the node has been fetched from the queue.

The blocking scheme described above has empty blocks at the bottom of the tree, called *leaf-blocks*. To ensure that the size of the tree is $O(n/B)$, we group one or more adjacent leaf-blocks together, such that at least every other leaf-block (in the left-to-right ordering) contains $B/2$ items.

Corollary 5.1 *Let $\mathcal{T}$ be a BSP in main memory for either disjoint segments in the plane or for objects in a low-density scene of depth $O(\text{depth}(\mathcal{T}))$. There is then an I/O-efficient BSP tree of depth $O(\text{depth}(\mathcal{T})/\log B)$ that uses $O(n/B)$ blocks.*

Proof. The blocking described above ensures that at least $O(\log B)$ nodes on any path in $\mathcal{T}$ are grouped in one block. So there are $O(\text{depth}(\mathcal{T})/\log B)$ blocks necessary to block any path in $\mathcal{T}$. □

For any query range Q the number of leaves whose cell is intersected by ∂Q is not changed using this blocking scheme, since we cannot ensure that the visited leaves are grouped with other visited leaves or nodes in a subtrees whose region is completely inside Q_ε. Therefore the number of visited leaf-blocks is equal to the number of leaves. A subtree whose region is completely inside Q_ε can be reported in a linear number of I/O-operations since each block containing a part from the subtree contains $\Theta(B)$ nodes of the subtree. Using these observations and the results obtained earlier we get the following two corollaries.

Corollary 5.2 *Let S be a set of n disjoint segments (or constant complexity curves) in $\mathbb{R}^2$. There exists an I/O-efficient BSP for S such that a range query with a constant-complexity convex range can be answered using $O(\min_{\varepsilon>0}\{(\varepsilon^{-1} + k_\varepsilon)\log_B n\})$ I/O-operations, where k_ε is the number of segments intersecting the extended query range Q_ε.*

Corollary 5.3 *Let S be a set of n objects in a λ-low-density scene in $\mathbb{R}^d$. There exists an I/O-efficient BSP for S such that a range query with a convex range can be answered using $O(\log_B n + \min_{\varepsilon>0}\{\lceil\lambda/B\rceil\varepsilon^{1-d} + \lambda\lceil k_\varepsilon/B\rceil\})$ I/O-operations, where k_ε is the number of objects intersecting the extended query range Q_ε.*

6 Conclusions

We have presented a general method to convert a BSP on the endpoints of a set of line segment into a BSP on the segments themselves, in such a way that the time for range searching queries remains almost the same and the size of the BSP is $O(n \log n)$. We used this to obtain a BSP with $O(\min_{\varepsilon>0}\{(1/\varepsilon) + k_\varepsilon\}\log n)$ query time for approximate range searching with arbitrary ranges in internal memory and $O(\min_{\varepsilon>0}\{(\varepsilon^{-1} + k_\varepsilon)\log_B n\})$ query time in external memory. Furthermore we showed how to generalize these results to curves.

We also presented a linear-size BSP for approximate range searching in low-density scenes in any dimension. Its query time is $O(\log n + \min_{\varepsilon>0}\{(1/\varepsilon^{d-1}) + k_\varepsilon\})$. Thus we obtain the same bounds as for point data, but for much more general objects. This improves the previously best known bounds for approximate range searching in $\mathbb{R}^3$ [22] by several logarithmic factors, and generalizes the results to higher dimensions as well. For the external memory variant the query time is $O(\log_B n + \min_{\varepsilon>0}\{\lceil\lambda/B\rceil\varepsilon^{1-d} + \lambda\lceil k_\varepsilon/B\rceil\})$ operations.

Our structures are the first pure BSP structures with guarantees on the query time. This is attractive because of the simplicity of such structures, and the fact that they are quite versatile. E.g., they can easily be used for (approximate) nearest-neighbor searching, and one can readily insert new objects into the structure (although then the query time may deteriorate). Moreover, the query times for approximate range searching are quite good, and our methods are surprisingly simple. Unfortunately, some of the constants (especially in the BSP for low-density scenes) can be fairly large in theory. We expect that this bad behavior will not show up in real applications, and that our structures will be quite competitive in practice, but this still needs to be verified experimentally.

References

[1] A. Aggarwal, J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):11161127, September 1988.

[2] P.K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, H.J. Haverkort, Box-trees and R-trees with near-optimal query time, Discrete Comput. Geom. 28 (2002) 291–312.

[3] P.K. Agarwal and J. Erickson. Geometric range searching and its relatives. In: B. Chazelle, J. Goodman, and R. Pollack (Eds.), *Advances in Discrete and Computational Geometry*, Vol. 223 of *Contemporary Mathematics*, pages 1–56, American Mathematical Society, 1998.

[4] P.K. Agarwal, E. Grove, T.M. Murali and J.S. Vitter. Binary space partitions for fat rectangles. *SIAM J. Comput.* 29:1422-1448, 2000.

[5] P.K. Agarwal, T.M. Murali and J.S. Vitter. Practical techniques for constructing binary space partition for orthogonal rectangles. In *Proc. 13th ACM Symp. of Comput. Geom.*, pages 382–384, 1997.

[6] P.K. Agarwal and S. Suri. Surface Approximation and Geometric Partitions. *SIAM J. Comput.* 19: 1016-1035, 1998.

[7] L.A. Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms In *Proc. 4th Workshop Algorithms Data Struct*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 1995.

[8] A. Arya, D. Mount, Approximate range searching, Comput. Geom. Theory Appl. 17 (2000) 135–152.

[9] C. Ballieux. Motion planning using binary space partitions. Technical Report Inf/src/93-25, Utrecht University, 1993.

[10] M. de Berg, H. David, M. J. Katz, M. Overmars, A. F. van der Stappen, and J. Vleugels. Guarding scenes against invasive hypercubes. *Comput. Geom.*, 26:99–117, 2003.

[11] M. de Berg. Linear size binary space partitions for uncluttered scenes. *Algorithmica* 28:353–366, 2000.

[12] M. de Berg, M. de Groot, and M. Overmars. New results on binary space partitions in the plane. In *Proc. 4th Scand. Workshop Algorithm Theory*, volume 824 of *Lecture Notes Comput. Sci.*, pages 61–72. Springer-Verlag, 1994.

[13] M. de Berg, M.J. Katz, A. F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 294–303, 1997.

[14] C. Breimann and J. Vahrenhold. External Memory Computational Geometry Revisited. In: U. Meyer, P. Sanders and J.F. Sibeyn (Eds.), *Algorithms for Memory Hierarchies* Vol. 2625 of *Lecture Notes in Computer Science*, pages 110–148, Springer, 2003.

[15] N. Chin and S. Feiner. Near real time shadow generation using bsp trees. In *Proc. SIGGRAPH'89*, pages 99–106, 1989.

[16] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer and E. Ramos. Randomized External-Memory Algorithms for Some Geometric Problems. In *Symposium on Computational Geometry*, pages 259–268, 1998.

[17] C.A. Duncan, Balanced Aspect Ratio Trees, Ph.D. Thesis, John Hopkins University, 1999.

[18] C.A. Duncan, M.T. Goodrich, S.G. Kobourov, Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees, In *Proc. 10th Ann. ACM-SIAM Sympos. Discrete Algorithms*, pages 300–309, 1999.

[19] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980. Proc. SIGGRAPH '80.

[20] L. J. Guibas and F. F. Yao. On Translating a Set of Rectangles. *Proceedings of the twelfth annual ACM symposium on Theory of computing*, 154–160, 1980.

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

[22] H.J. Haverkort, M. de Berg, and J. Gudmundsson. Box-Trees for Collision Checking in Industrial Installations. In *Proc. 18th ACM Symp. on Computational Geometry*, pages 53–62, 2002

[23] B. Naylor, J. A. Amanatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. *Comput. Graph.*, 24(4):115–124, August 1990. Proc. SIGGRAPH '90.

[24] M.H. Overmars, H. Schipper, and M. Sharir. Storing line segments in partition trees. *BIT*, 30:385–403, 1990

[25] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.

[26] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *J. Algorithms*, 13:99–113, 1992.

[27] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[28] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. *Comput. Graph.*, 25(4):61–69, July 1991. Proc. SIGGRAPH '91.

[29] C.D. Tóth. A Note on Binary Plane Partitions. *Discrete Comput. Geom.* 20:3–16, 2003.

[30] C.D. Tóth. Binary Space Partitions for Line Segments with a Limited Number of Directions. *SIAM J. Comput.* 32:307–325, 2003.

[31] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Comput. Graph.*, 21(4):153–162, 1987. Proc. SIGGRAPH '87.