Multifunctional geometric data structures

Document status and date:
Published: 01/01/2007

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Download date: 23. Jun. 2025

MULTIFUNCTIONAL
GEOMETRIC DATA STRUCTURES

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 29 mei 2007 om 16.00 uur

door

Michaël Wilhelmus Albertus Streppel

geboren te Nijmegen

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. M.T. de Berg

Copromotor:
dr. H.J. Haverkort

promotor: Prof. dr. Mark T. de Berg,
               faculteit Wiskunde & Informatica,
               Technische Universiteit Eindhoven

co-promotor: dr. Herman J. Haverkort,
               faculteit Wiskunde & Informatica,
               Technische Universiteit Eindhoven

Cover design: Doki Tops

Printed by: Printservice Technische Universiteit Eindhoven

Contents

Chapter 1

Introduction

Computational geometry is an area of research on algorithms involving geometric objects, such as points, lines and polygons. These geometric objects may represent objects in the physical world, such as streets and rivers in a car navigation system, or a robot which has to be moved between obstacles. Geometric objects can also be abstract, such as an entry in a database consisting of the age, year of employment and salary of a person. This entry can be represented by a point in three dimensions. There are many application areas — for instance geographical information systems, computer graphics, robotics, databases, and astrophysics — where problems can be defined in terms of geometric queries on a set of objects. In computer graphics rendering all objects in the set can be very time consuming. One wants to render only those objects which are visible for the viewer. The geometric query for this example could be: "Which objects are within the viewing volume?". An interactive country map might offer the possibility to retrieve information on a city by selecting it with the mouse. The geometric query asked in this example could be: "Which city (or, more generally, area of the map) is underneath the hot spot of the mouse?".

Many data structures have been developed to answer a wide range of geometric queries. Some of the data structures are very specialized and can answer only a single type of query, for instance finding points contained in a halfplane. Such specialized data structures are usually unable to answer seemingly similar queries efficiently. A data structure for reporting all points contained in a halfplane usually cannot efficiently report all points contained in a box (which is defined by a set of $2d$ halfplanes) or in a disk. Similarly, many data structures can only store one type of objects. For instance, they can store line segments but not disks, or disks but not line segments. Other data structures are able to answer several types of geometrical queries and can store several types of objects. These data structures — we call them *multifunctional geometric data structures* — are often used in practice. However they have not been studied extensively from a theoretical point of view. This is the topic of this thesis: to perform a fundamental study of the efficiency of multifunctional geometric data structures.

The flexibility which is offered by multifunctional geometric data structures comes at a cost of a (theoretical) loss in *query efficiency*. In Section 1.1 we mention some important queries that multifunctional geometric data structures can answer. How the theoretical query efficiency of a data structure is determined is briefly discussed in Section 1.2.1. The theoretical analysis of the query efficiency often considers the worst possible input which can be given to the data structure. This input might not resemble the input observed in practice, so the theoretical analysis of an algorithm might be too pessimistic. In order to obtain a realistic efficiency estimate for an algorithm, assumptions can be made on the input stored in the data structures and the queries asked. These assumptions are captured by *realistic input models*, which are explained in Section 1.2.2.

The existing multifunctional geometric data structures can roughly be classified in two classes, *space partition structures* (SPS) and *bounding-volume hierarchies* (BVH). The first class of multifunctional geometric data structure recursively partitions the space into smaller parts such that there is one or only a few objects in each part of the space. In this thesis we are mainly interested in a special subclass of the space partitioning structures, the *binary space partition* (BSP). The second class of multifunctional geometric data structures builds a hierarchy of objects by joining objects recursively into larger groups until all objects are in one group. With each group we store a *bounding volume* that encloses all objects in the group. The binary space partition and the bounding-volume hierarchy are introduced in more detail in Section 1.3.

We conclude this introductory chapter with an overview of the results obtained in this thesis.

1.1   Geometric queries

There are several types of geometric queries on a set S of n objects in $\mathbb{R}^d$ which can all be answered using a single multifunctional geometric data structure. In this section we give a short introduction to the most important types of queries; for an excellent survey on geometric queries we refer the reader to the survey by Agarwal and Erickson [3]. Geometric queries can be answered exactly or approximately. The theoretical bounds on the query times in the former case are usually bad. In the latter case the theoretical bounds are much better, but one has to tolerate an error in the answer. The error can be specified for every query separately and the theoretical bound depends on that error.

1.1.1   Exact queries

We are interested in the following two types of exact geometrical queries, *intersection queries* and *proximity queries*. Answering these queries is quite similar for both classes of multifunctional geometric data structures. Both BVHs and SPSs are structured as a tree $\mathcal{T}$. A node $\nu \in \mathcal{T}$ is associated with some part of the
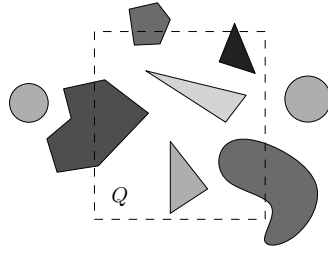
Figure 1.1: A range searching query asks for all objects intersecting the query range. All objects except the two circles intersect the query range Q.

space, which we denote by $R(\nu)$. We only need the tree structure and the regions associated with the nodes to answer the queries presented in this section.

Intersection queries

Intersection queries are queries where one wants to find (or count, or compute some function on) all objects intersecting the query. Intersection queries are for instance used in geographical information systems and computer graphics. The following queries are examples of intersection queries.

Range searching queries.   Compute the subset of all objects in a set S intersecting the query range Q, see Figure 1.1. More formally, we want to compute the set $S^* := \{o \in S : o \cap Q \neq \emptyset\}$. Data structures have been developed for many types of query ranges, such as for axis-parallel queries, halfplane queries, simplex queries and disk queries. For answering range searching queries we start at the root of a tree $\mathcal{T}$ and recursively visit its children whose regions intersect Q; at a leaf we return the object o stored there if o intersects Q.

Range aggregate queries.   Suppose each object $o_i \in S$ has a weight $\omega(o_i)$. The task is to compute an associative and commutative function $\bigoplus_{o \in S^*} \omega(o)$ where S^* is the set of objects in S for which $o \cap Q \neq \emptyset$. To let the query time be independent of the number of objects intersecting Q, one stores at every node ν the result of that function on the objects below ν. We say that a function is *duplicate-insensitive* if $x \oplus x = x$ for any x. For example MAX is a duplicate-insensitive functions while $+$ is not. If the construction of the multifunctional geometric data structure can not guarantee that an object is stored at exactly one leaf then only duplicate-insensitive functions can be computed efficiently with a range aggregate query. For answering a range aggregate query we start from the root of $\mathcal{T}$ and traverse the tree as in a range searching query, while keeping a running aggregate. As opposed to a range searching query we do not recursively visit the children of some node μ if $R(\mu)$ is completely within Q. Instead, we
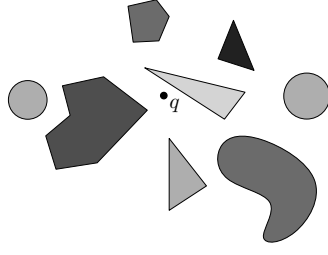
Figure 1.2: A nearest-neighbor query asks for an object which is closest to the query. The light gray triangle is the object which is closest to the query point q.

apply $\oplus$ to the running aggregate and the (pre-computed) outcome of $\bigoplus_{o \in S^*} \omega(o)$ stored at μ, where S^* is the set of objects in the subtree rooted at μ. A range aggregate query can, for instance, be used to determine the maximum weight of an object in Q or to count the number of points within the query.

Point (location) queries or inverse range queries.   Compute the set of objects in S containing a query point q. This is basically a degenerate case of the range searching query, but due to its importance it deserves to be mentioned separately. Answering a point query is done similar to answering a range searching query.

Proximity queries

The second type of exact queries are proximity queries. The answer of this type of query involves the distance from the query to the objects in S. Proximity queries are for instance used in motion planning [14] and pattern recognition [35]. The following two queries are proximity queries.

Nearest-neighbor queries.   Find an object in S closest to the query Q, or more formally, return an object o for which $\delta(o, Q) = \min_{o' \in S} \delta(o', Q)$, where $\delta(A, B)$ is the distance between A and B for some distance measure, see Figure 1.2. The nearest-neighbor query can be generalized to the case that the k closest objects should be returned. The query is then called a k-*nearest-neighbor query*. For answering a nearest-neighbor query we use a priority queue $\mathcal{Q}$ in which we store some nodes of the tree $\mathcal{T}$. The priority of a node is inversely proportional to the minimum distance between its region and the query. We first add the root of $\mathcal{T}$ to $\mathcal{Q}$, set the current minimum distance δ to infinity and continue as follows. Let ν be the node in $\mathcal{Q}$ with the highest priority. While $\mathcal{Q}$ is not empty and the distance from the region of ν to the query is smaller than δ we extract ν from $\mathcal{Q}$. If ν is a leaf we determine the distance between Q and the object(s) stored at ν and if necessary update the shortest distance δ and the nearest-neighbor found until now. Otherwise we add each child of ν to $\mathcal{Q}$.

Furthest-neighbor queries. Find an object in S furthest from the query Q, or more formally, return an object o for which $\delta(o, Q) = \max_{o' \in S} \delta(o', Q)$ where $\delta(A, B)$ is the distance between A and B for some distance measure. As in a nearest-neighbor query we use a priority queue $\mathcal{Q}$ to store some nodes of the tree. The priority of a node ν in the priority queue is in this case proportional to the maximum distance between the region of ν and the query. We add the root of $\mathcal{T}$ to $\mathcal{Q}$ and set the current maximum distance δ to 0. Again let ν be the node in $\mathcal{Q}$ with the highest priority. While $\mathcal{Q}$ is not empty and the distance from the furthest point in the region of ν to the query is larger than δ we extract ν from $\mathcal{Q}$. If ν is a leaf we determine the distance between Q and the object(s) stored at ν and if necessary update the largest distance δ and the furthest neighbor found until now. Otherwise we add each child of ν to $\mathcal{Q}$.

It can be shown that the time to answer an Euclidian nearest-neighbor query on an SPS or BVH is the same, up to an $O(\log n)$ factor, as the time needed to answer a range query with an empty disk. For an Euclidian furthest-neighbor query on an SPS or BVH it can be shown that the cost is the same, up to an $O(\log n)$ factor, as the time needed to answer a aggregate range query with a disk. Hence we will concentrate on analyzing the time for range searching in most chapters of this thesis.

1.1.2 Approximate queries

Exact range searching either uses non-linear storage or incurs super-logarithmic query time [30]. It is therefore natural to seek for approximate solutions. The concept *ε-approximate range searching* was introduced by Arya and Mount [13]. Let $\mathrm{diam}(Q)$ be the diameter of a query range Q. In ε-approximate range searching one considers, for a parameter $\varepsilon > 0$, the ε-extended query range Q_ε, which is the locus of points lying at distance at most $\varepsilon \cdot \mathrm{diam}(Q)$ from Q, see Figure 1.3. For a set S of n objects in $\mathbb{R}^d$, approximate queries similar to the exact queries of the previous section can be defined.

Range searching in the approximate setting returns a set S^* such that $\{o \in S : o \cap Q \neq \emptyset\} \subseteq S^* \subseteq \{o \in S : o \cap Q_\varepsilon \neq \emptyset\}$ and the approximate range aggregate query computes $\bigoplus_{o \in S^*}$. Proximity queries can also be answered in the approximate setting. An approximate nearest neighbor query returns, for a query Q, an object $o \in S$ such that $\delta(o, Q) \leq (1 + \varepsilon)\delta(o^*, Q)$, where o^* is the true nearest neighbor of Q and $\delta(A, B)$ is the minimum distance between A and B for some distance measure.

1.2 Algorithm analysis

Usually the efficiency of an algorithm is not stated in milliseconds running time. The stated running time would be outdated quickly, since the hardware gets faster
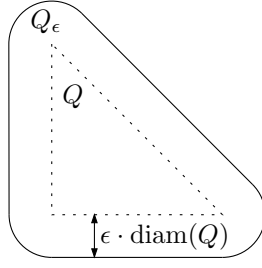
Figure 1.3: The approximate query range Q_ε of the query Q.

all the time. Therefore a different measure is used to express the running time of an algorithm, the number of elementary operations it performs. The number of elementary operations depends on the input size n, output size k and possibly on some property of the input. Usually the efficiency of an algorithm is given for the *worst-case*, the largest number of elementary operations needed before the algorithm terminates.

The running time of algorithms is analyzed asymptotically. We want to know how the running time roughly is affected by a change in the input size, output size or possibly the property of the input. Suppose the running time of an algorithm A only depends on the input size. The asymptotic running time for A is denoted by $O(f'(n))$ if there are positive constants c, n_0 such that $f(n)$, the number of elementary operations performed by A, is at most $c \cdot f'(n)$ for any $n \geq n_0$. When analyzing an algorithm the asymptotic running time of an algorithm is expressed as close to the actual running time as possible and in as little terms depending on n. For instance if the actual running time is given by $f(n) = 12n^2 - 4n + 3$ then the asymptotic running time is denoted by $O(n^2)$ since for $c = 12$ and $n_0 = 1$ the relation $f(n) \leq c \cdot f'(n)$ holds.

Models of computation define the elementary operations used in the analysis of an algorithm. Fixing a computational model allows one to compare two algorithms objectively. Input models are used to describe properties of the input to the algorithm. Without assumptions on the properties the worst possible input has to be taken into account during the analysis, even when this input is never encountered in practice. Using the assumptions on the input or the queries makes it possible to develop more efficient data structures or the assumptions might explain why theoretically bad data structures perform well in practice.

1.2.1 Models of computation

The two computational models which are used in this thesis are introduced briefly in this section. For a discussion on other computational models we refer the reader to the survey by Van Emde Boas [39].
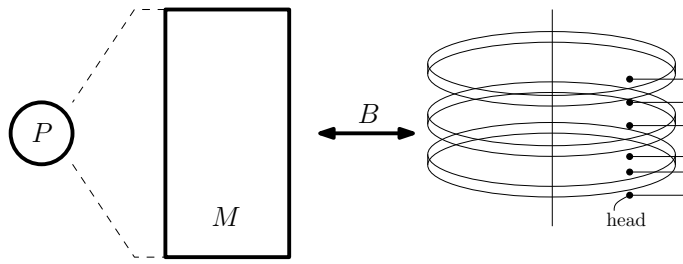
Figure 1.4: In the external-memory model the processor P can only do computations on data in internal memory of size M. If the data is not in internal memory it is transferred from disk to internal memory in blocks of size B.

Random Access Memory model

The internal memory algorithms are analyzed in the *Random Access Memory* model [9, 39], or RAM-model for short, with *uniform cost criterion*. The RAM-model with uniform cost criterion is also called *real RAM-model*.

In the RAM-model there is an unlimited amount of memory available. The algorithm is able to access an arbitrary part of the memory at unit cost. In the RAM-model the following arithmetic instructions are considered to be elementary operations: addition, subtraction, multiplication and division. The uniform cost criterion states that the cost of an operation does not depend on the number of bits used for storing the operands. This implies that arithmetic operations on real numbers can be handled with infinite precision in constant time.

External-memory model

The external-memory model introduced by Aggarwal and Vitter [8] has become the standard model for external-memory algorithms. In this model, a computer has an internal memory of size M and an arbitrarily large external memory (disk), see Figure 1.4. In external memory, data is stored in blocks of size B. Whenever an algorithm wants to work on data not present in internal memory, the block(s) containing those data are read from external memory. Writing data to external memory is also done in blocks. The complexity of an algorithm in this model, the *I/O-complexity*, is measured in terms of the number of I/O-operations—reading or writing a block from or to external memory—it performs. Any computation in the internal memory is for free and therefore does not influence the I/O-complexity. This is justified by the fact that accessing data on a disk takes several orders of magnitude longer than accessing data in internal memory.

An overview of I/O-efficient algorithms and techniques in computational geometry can be found in the survey by Breimann and Vahrenhold [28]. As usual, we assume that $B \ll M < N$, where N is the input size.

1.2.2   Models of input

Many data structures developed in computational geometry are developed with a worst-case input in mind. However, this worst-case input might not occur in practice at all. Input models can be used to capture a property of the input, which can be exploited in the development of data structures. For instance, an input model can stipulate that the objects in the input have a restriction on their shape (such as *fatness*, see below) or that they are distributed in a certain way (such as *density*, see below). In general, the models assign a parameter to the input that describes how well the input satisfies the model – how fat they are, or how dense the scene is. The analysis of the algorithm is then done in terms of the input size, n, the output size k, and this parameter.

In this section the models of input occurring in this thesis are briefly described. Other models include unclutteredness [19], simple cover complexity [64] and dispersion [73, 89].

Fatness

Fatness is probably the best known realistic input model. There are many definitions for fatness [5, 57, 64, 81]. We will use the following definition [24], see Figure 1.5. Let β be a constant with $0 < \beta \leq 1$. Let B be a ball whose center is inside an object o and which intersects the boundary of o. The object o is β-fat if for any such ball B the intersection of o and B covers at least a fraction β of the ball, or more formally, $\mathrm{vol}(o \cap B) \geq \beta \cdot \mathrm{vol}(B)$, where vol denotes the volume. Sometimes the fatness-parameter β is omitted and an object is called fat if it is β-fat for some not too small constant β. A set S is said to be β-fat if all objects in S are β-fat.
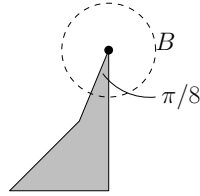
Figure 1.5: This object o is $1/16$-fat, but not β-fat for any $\beta > 1/16$.

Low stabbing number

Input objects are sometimes disjoint or they do not overlap too much. This property is quantified by the stabbing number. The stabbing number of a set S of objects is a measure for how many objects in S intersect in a single point of the space, see Figure 1.6. The stabbing number $\sigma(p, S)$ of a point p is the number of objects from S it intersects. A set S is said to have stabbing number $\sigma(S)$ if

no point in the space has stabbing number larger than σ for the objects from S, or more formally $\sigma(S) = \max_{p \in \mathbb{R}^d} \sigma(p, S)$. A set of disjoint objects thus has a stabbing number of one.
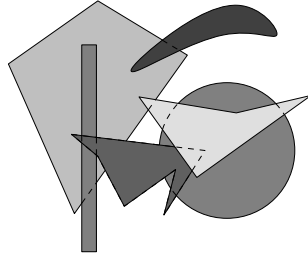
Figure 1.6: This scene has a stabbing number of 3, since there is no point intersected by more than three objects.

Low density

The notion of low density was introduced by Van der Stappen [81, 82]. 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 $\rho(o) \geqslant \rho(B)$ [24], where $\rho(o)$ is the radius of the smallest enclosing ball of o. If S has density λ, we call S a λ-low-density scene.
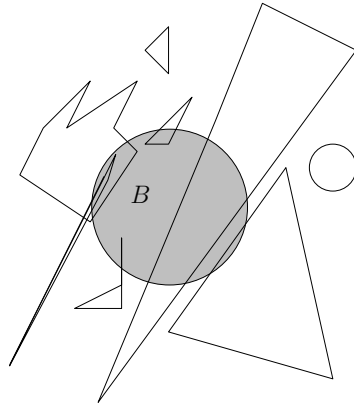
Figure 1.7: This scene is a 4-low-density scene. The ball B intersects 4 objects for which $\rho(o) \geqslant \rho(B)$ holds.

1.3   Multifunctional geometric data structures

In the database community multifunctional geometric data structures are often called *access methods*, or *index structures* [61, 67]. In the past decades many index structures have been proposed, to organize the spatial objects stored in the database so that a geometrical query can be answered efficiently. See the survey by Gaede and Günther [44] and the book by Samet [78].

Existing multifunctional geometric data structures can be roughly categorized into *bounding-volume hierarchies* (BVHs) and *space-partitioning structures* (SPSs). In this thesis we are interested in a specific class of SPS, the *binary space partition* (BSP) which is introduced in the next section. BVHs are introduced in Section 1.3.2.

1.3.1   Binary space partitions

A *binary space partition tree*, or BSP *tree*, for a set S of n objects in $\mathbb{R}^d$ 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 $\mathbb{R}^2$, planes in $\mathbb{R}^3$, etc.) A BSP tree is a binary tree $\mathcal{T}$ with the following properties — see Figure 1.8 for an example in $\mathbb{R}^2$.

- Every (internal or leaf) node ν corresponds to a subset $\mathrm{R}(\nu)$ of $\mathbb{R}^d$, which we call the *region* of ν. These regions are not explicitly stored with the nodes. When ν is a leaf node, we sometimes refer to $\mathrm{R}(\nu)$ 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 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 $\mathrm{R}(\nu) \cap h(\nu)^-$, where $h(\nu)^-$ denotes the half-space on one side of $h(\nu)$, and the right child corresponds to $\mathrm{R}(\nu) \cap h(\nu)^+$, where $h(\nu)^+$ is the half-space on the other side of $h(\nu)$. The hyperplane $h(\nu)$ is said to induce a *balanced* split when both children of ν store at most $\beta|S_\nu|$ objects, for some $0.5 \le \beta < 1$, where S_ν is the set of objects stored in the subtree rooted at ν.

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

Note that we do not place a bound on the number of objects stored with a leaf. In the BSP trees discussed in this thesis, however, this number will be constant.

BSP trees are used for many purposes. For example, they are used for range searching [3, 25], for hidden surface removal with the painter's algorithm [42], for shadow generation [33], for set operations on polyhedra [65, 87], for visibility preprocessing for interactive walkthroughs [84], for cell decomposition methods in motion planning [15], and for surface approximation [7].
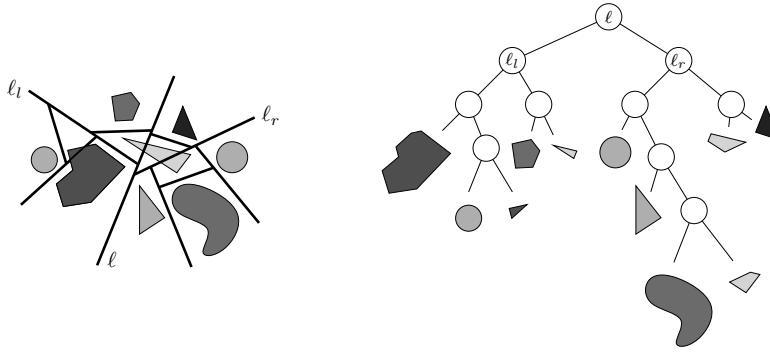
Figure 1.8: A BSP in the plane, and the corresponding tree. With the leaves we have shown the fragments inside the corresponding cell, although normally just a pointer to the object is stored.

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 algorithms have been proposed which construct small BSP trees in various settings [4, 6, 19, 21, 71, 72, 86]. For instance, Paterson and Yao [71] proved that any set of n disjoint line segments in the plane admits a BSP tree of size $O(n \log n)$. Tóth [85] 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.

Next we introduce two special types of BSPs. The first BSP is the kd-tree which can answer exact orthogonal range queries efficiently and the second BSP is the BAR-tree, which can answer approximate range queries efficiently.

kd-trees

The kd-tree was described for the first time in 1975 by Bentley [18]. Usually kd-trees are used to store a set of points in $\mathbb{R}^d$. The kd-tree uses axis-aligned splitting hyperplanes to divide the point set into two (almost) equal sized sets. The resulting sets are split recursively. In the standard kd-tree the dimension along which to split the point set alternates in a round-robin fashion, see Figure 1.9 a) for a partitioning of the plane by a standard kd-tree. So in $\mathbb{R}^3$ the point set is first split along the x-axis, then along the y-axis, then along the z-axis and then again along the x-axis, and so on. A rectangular range query Q in a kd-tree takes $O(n^{1-1/d} + k)$ time where k is the number of points within Q. A rectangular aggregate query can be answered in $O(n^{1-1/d})$ time and a point-location query in $O(\log n)$ time. A nearest-neighbor query takes $O(n)$ time in the worst case, but
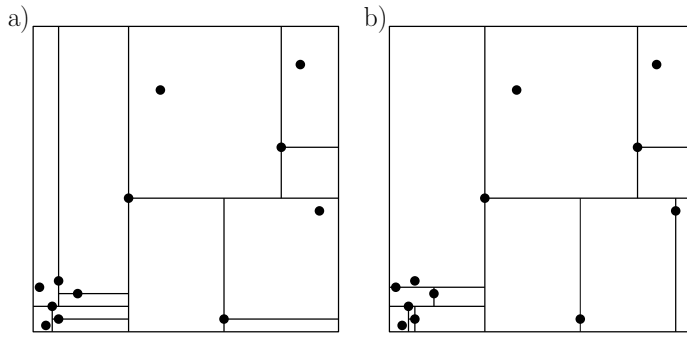
Figure 1.9: A partitioning of the plane by a) a standard kd-tree. b) a LSF-kd-tree. A point on a splitting line ℓ is always considered to be right of ℓ and below ℓ.

Friedman et al. [41] argued that, under certain assumptions, a nearest-neighbor query takes logarithmic expected time.

A variant of the standard kd-tree is the longest-side-first kd-tree (LSF-kd-tree). In the LSF-kd-tree the splitting plane at a node ν is chosen such that it cuts the longest side of the region of ν, where ties are broken arbitrarily. See Figure 1.9 b) for an example of a partitioning of the plane by a LSF-kd-tree. Dickerson et al. [36] showed that the LSF-kd-tree can answer approximate nearest-neighbor queries in $O(\varepsilon^{1-d}\log^d n)$ time and approximate range queries in $O(\varepsilon^{1-d}\log^d n + k)$ time, where k is the number of objects returned.

Robinson [75] introduced an external-memory variant of the kd-tree. The number of I/O-operations needed to answer an axis-aligned range query in his K-D-B-tree on N points is $O((N/B)^{1-1/d} + k/B)$, where B is the number of points which can be stored in a block.

BAR-trees

The BAR-tree was introduced by Duncan et al. [38, 37]. The regions associated with nodes in the BAR-tree have the following properties. The regions are convex, they have aspect ratios bounded by some constant α, and their boundary consists of a constant number of vertical, horizontal, and diagonal line segments, see Figure 1.10 for a partitioning of the plane by a BAR-tree. Duncan et al. [38, 37] proved that under these constraints for any $\alpha \geq 3d$, using two splits, any $R(\nu)$ can be partitioned into 3 cells, such that the number of points in any cell is at most a constant fraction β of the number of points in $R(\nu)$. Thus the height of the tree can be bounded by $O(\log n)$. Note however that some nodes in a BAR-tree may not store a hyperplane which induces a balanced split, since sometimes the first split may have to partition the points in $R(\nu)$ into two subsets with drastically different cardinalities.

A BAR-tree uses linear space, and because of the properties of the regions,
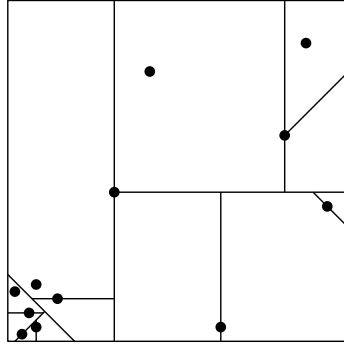
Figure 1.10: A partitioning of the plane by a BAR-tree. A point on a splitting line ℓ is always considered to be right of ℓ and below ℓ.

the number of nodes visited during a query can be effectively bounded using a packing argument [13]. An approximate range query takes $O(\log n + \varepsilon^\gamma + k_\varepsilon)$ time for any query range Q, where $\gamma = 1 - d$ for convex ranges and $\gamma = -d$ otherwise, and k_ε is the number of points inside the extended query range Q_ε. An approximate aggregate query can be answered in time $O(\log n + \varepsilon^\gamma)$, a point-location query in $O(\log n)$ time and a nearest-neighbor query or furthest-neighbor query in $O(\log n + \varepsilon^{1-d} \log(1/\varepsilon))$ time.

As noted by Haverkort et al. [50], the query time of an exact range query in a BAR-tree can be bounded by $O(\log n + \min_\varepsilon\{\varepsilon^\gamma + k_\varepsilon\})$ since Q_ε is only used in the analysis and not by the query algorithm, which only uses Q to visit $\mathcal{T}$ and always reports the correct answers $P \cap Q$. Hence for range searching the notion of approximate range searching in fact just stipulates that we are using ε and k_ε to bound the running time, rather than the traditional "exact" output size $k = |P \cap Q|$.

1.3.2 Bounding-volume hierarchies

The second class of multifunctional geometric data structures is the *bounding-volume hierarchy* (BVH), see Figure 1.11. A BVH on a set S of n 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 constant-complexity bounding volume of the set of objects corresponding to the leaves in the subtree of ν. A BVH has size $O(n)$ by definition, and it can store any kind of object. A query with a range Q is answered by traversing the BVH in a top-down manner, only proceeding to those nodes whose bounding volumes intersect Q. For each leaf that is reached, the corresponding object needs to be checked against Q. In principle one can perform the search with any kind of range Q. To speed up the test whether the range Q intersects the bounding volume of some node, however, the range itself
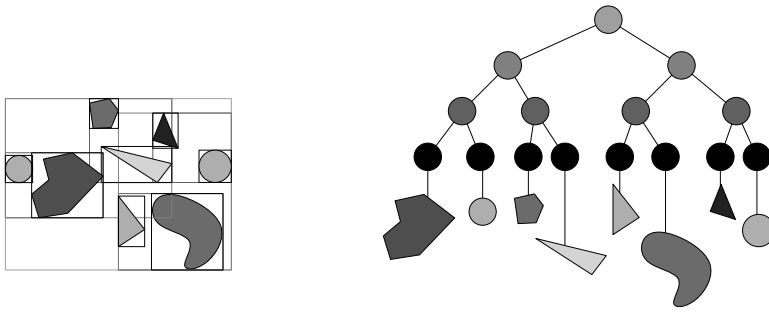
Figure 1.11: A BVH on objects in the plane, and the corresponding tree.

is often also replaced by a bounding volume. Hence, the possibly expensive test with the original range Q only has to be performed with the objects found at the leaf level. Note that there is a trade-off in the type of bounding volume used: simple bounding volumes make the intersection test fast, but they often fit the underlying objects less tightly so that more nodes in the tree are visited.

A very popular bounding volume is the axis-aligned bounding box. The reason for this is that intersection tests between bounding boxes are very fast and easy to implement. BVHs that use bounding boxes as bounding volumes— such BVHs are called box-trees—have been investigated from a theoretical point of view by De Berg et al. [22], Agarwal et al. [2], and Haverkort et al. [50]. Agarwal et al. showed how to construct a box-tree for a set S of n input boxes in $\mathbb{R}^d$ such that a range query with an axis-aligned query box Q can be answered in time $O(n^{1-1/d} + k)$, where k is the number of input bounding boxes intersecting Q. They also showed that this is optimal in the worst case, even if the input boxes are disjoint; this is also implied by the results of Kanth and Singh [55]. For inputs that consist of disjoint axis-aligned bounding rectangles in the plane, Agarwal et al. [2] present another box-tree, which achieves a query time of $O(\sqrt{n} \log n + k)$ for queries with axis-aligned rectangles, and $O(\log^2 n)$ for point queries. In Haverkort's thesis [49], the bound for rectangle queries is improved to the optimal $O(\sqrt{n} + k)$.

As noted above, a simple bounding volume such as a bounding box may not fit the underlying objects or the query range very well. Suppose, for instance, that we want to find all objects intersecting a query line segment and suppose that the objects are not too large and are distributed on a line ℓ whose slope is 1, see Figure 1.12. The bounding boxes enclose some space in the neighborhood of ℓ. When we query with a segment s whose slope is also close to 1, then, even when s is not approximated with a bounding box, the query has to visit many nodes without returning a single answer. Indeed, the box-trees of Agarwal et al. [2] (or any box-tree, for that matter) cannot give any (sublinear) worst-case guarantees for queries with non-rectangular ranges or in non-rectangular data. The
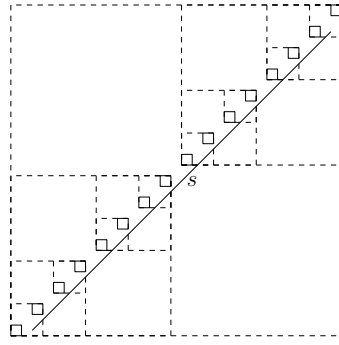
Figure 1.12: A query with a line segment may visit many axis-aligned bounding boxes without intersecting an object.

fact that bounding boxes may not always fit well, inspired research on BVHs that use different, more tightly fitting bounding volumes [40, 46, 58, 80]. The types of bounding volumes suggested include spheres [51, 68], oriented bounding boxes [16, 46], and discretely oriented polyhedra [43, 52, 56, 90].

Below we introduce the R-tree, a well known external-memory BVH.

R-trees

The R-tree, originally introduced by Guttman [47] in 1984, is a height-balanced multi-way tree similar to a B-tree. An R-tree is a bounding-volume hierarchy with axis-aligned boxes as the bounding volume. Let B be the size of a disk block. Every internal node of an R-tree has degree $\Theta(B)$, except for possibly the root whose degree is at least two. Every internal node, including the root, stores for every child a smallest enclosing axis-aligned box. Every leaf contains $\Theta(B)$ boxes, each enclosing an object from the input set, with possibly a pointer to the original object. An R-tree on N objects is stored in $\Theta(N/B)$ (disk) blocks and has height $\Theta(\log_B N)$. R-trees are constructed by *repeated insertion* or by *bulk-loading*.

When R-trees are constructed by repeated insertion, the objects in the input set are inserted in the R-tree one at a time. An update of an R-tree changes the bounding boxes stored in some internal nodes. The query efficiency of an R-tree depends on the amount of overlap between the bounding boxes stored in the tree. Therefore many R-trees have been developed [17, 47, 54, 79] using different heuristics for the updates. The cost of updating an R-tree is usually $O(\log_B N)$. The cost of the construction of an R-tree is thus $O(N \log_B N)$. Bulk-loading algorithms consider the whole set of objects at once during the construction of an R-tree. The cost of most bulk-loading construction algorithms [11, 45, 53, 59] is $O((N/B) \log_{M/B}(N/B))$.

R-trees are heuristic-based structures and have no good guarantees on the

query performance. It was shown [2] that in the worst case, a query has to visit $\Omega((N/B)^{1-1/d} + k/B)$ blocks using *any* R-tree variant built on N points in $\mathbb{R}^d$, where k is the output size. This lower bound is reached by a recently developed R-tree variant, called the PR-tree [11], but the result holds only if both the queries and objects are axis-parallel hyperrectangles.

For an overview of the theory and applications of R-trees the reader is referred to the book by Manolopoulos et al. [60].

1.4 Overview of this thesis

This thesis is divided into two parts. The first part is dedicated to binary space partitions and the second part to bounding-volume hierarchies.

In Chapter 2, we study the use of binary space partitions for queries with so called *discretely oriented polygons* (DOP) in a set of points in $\mathbb{R}^d$. A DOP is a convex polygon whose edges have an orientation from a set of a constant number of predefined orientations. We introduce a new BSP for this family of queries which has approximately the same query time as an axis-aligned range query in a kd-tree. Next we develop a framework that converts a BSP on points to a BSP on line segments in Chapter 3. Using this framework we give a BSP which can answer approximate range-searching queries efficiently. In Chapter 4 we give a BSP of linear size for a set of objects in a low-density scene, which we call the oBAR-tree. This data structure is able to answer approximate range-searching queries as efficiently as the BAR-tree. In the last chapter of this part we extend both the BAR-tree and the oBAR-tree to the I/O-efficient setting.

In the second part we are interested in bounding-volume hierarchies. Usually bounding-volume hierarchies use axis-aligned bounding boxes as bounding volumes. By using DOPs, however, one can get bounding volumes that fit more tightly. Investigating such BVHs, which we call DOP-*trees*, is the topic of the second part. In Chapter 6 we give a lower bound on the query cost in a DOP-tree. We also give an algorithm to construct a DOP-tree achieving this bound. In the same chapter we develop a DOP-tree for scenes with a low stabbing number which can answer a range query with a DOP at almost the (theoretically) optimal cost as an axis-aligned query in a kd-tree. In the second and last chapter of this part we experimentally investigate the influence of using more orientations to describe the underlying objects in the external memory setting. We also compare these external-memory DOP-trees with the PR-tree.

Chapter 8 concludes this thesis with an overview of the results presented and some open problems.

The work presented in this thesis is based on my research of the last four years. The results, except for the experimental study in Chapter 7, have been presented in [23, 26, 27, 83].

Part I

Binary Space Partitions
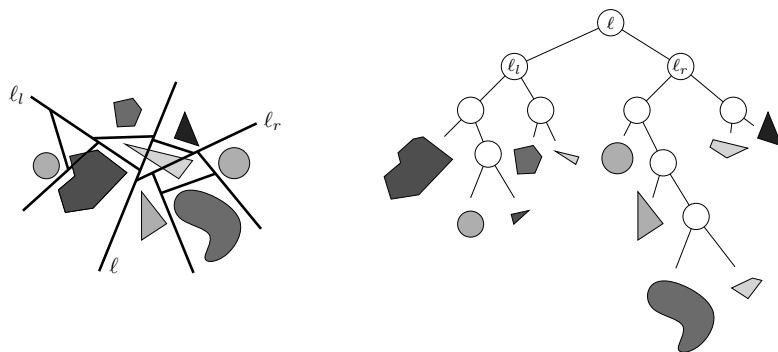
Chapter 2

c-Oriented Range Queries in Point Data

Recall from Section 1.1 that in the range searching problem we want to find the objects from a set S that intersect a given region called the query range. We want to preprocess S into a data structure D such that D can be used repeatedly for determining the objects intersecting a query range efficiently. In practice the overhead in storage used for D should be kept as small as possible. We are therefore interested in data structures which needs linear storage.

Range searching is an important topic in computational geometry. In the past a lot of research has been done on special instances of the range searching problem. These instances include orthogonal range searching [29, 30, 76, 77], simplex range searching [31, 34, 63] and halfspace range searching [32, 62]. In this chapter we are interested in another special instance, convex-polytope range searching for which the orientations of the facets of the polytope come from a pre-determined set of c orientations. We call this problem the c-DOP *range-searching problem*. Note that axis-aligned range searching is an instance of c-DOP range searching.

A c-DOP is more formally defined as follows. Let $\mathcal{C}$ be a set of c non-parallel hyperplanes having orientations $o_1 \ldots o_c$. We say that two hyperplanes have the same orientation if they are parallel. Thus the set $\mathcal{C}$ defines c orientations. We say that a hyperplane, or a $(d-1)$-dimensional facet of a d-dimensional polytope, is *c-oriented* if it has one of the c orientations defined by $\mathcal{C}$. We call a convex polytope a c-DOP if all of its facets are c-oriented. Hence, a c-DOP has at most $2c$ facets. We assume that the set $\mathcal{C}$ is fixed, and terms like c-DOPs, c-oriented, and so on, always refer to this set $\mathcal{C}$. Moreover, when we speak of a DOP, we always mean a c-DOP.

In the first section of this chapter we describe two existing SPSs for which there is a bound on c-oriented range searching. The first SPS has a good bound on the query time when the query is a DOP, but cannot guarantee a better bound

when the query is a point. The second structure is a BSP and has a good bound on the query time with points. The bound for DOP-queries, however, is high. In Section 2.2 we develop a new data structure which has good query times for both point and DOP-queries. It answers point queries in $O(\log n)$ time and DOP-queries in $O(n^{1-1/d+\varepsilon} + k)$ time where k is the number of points in the query and ε is an arbitrary constant larger than 0. This bound holds for any set $\mathcal{C}$ of a constant number of orientations.

2.1 Existing solutions

A c-DOP-query can be answered by decomposing the query in simplices and then query with each simplex. In Section 2.1.1 we briefly describe a data structure by Matoušek [63] which has the best query time for simplex range searching. One would hope that small queries can be answered more efficiently than large queries. The time bound for point queries can be used as an indication for the query time of small queries. Unfortunately this structure does not have a good query bound for point queries (that is, the query time for point queries is not better than the query time for simplex queries).

    Section 2.1.2 discusses how the kd-tree, a well-known data structure for axis-aligned range searching which has a good point query time, can be extended to support c-oriented range queries. The resulting kd-tree, however, does not have a very good DOP-query time.

2.1.1 Partition trees
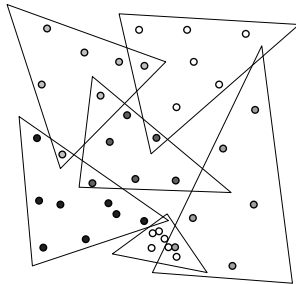
Figure 2.1: A simplicial partition of size 6.

    The partition tree by Matoušek [63] is a data structure with the best known bound for simplex range searching. It is constructed using a recursive application of a *simplicial partitioning*, see Figure 2.1. Let S be a set of n points in $\mathbb{R}^d$ and let r be a positive integer at most n. A simplicial partition of size r for the set S is a set of pairs $\Psi(S) = \{(S_1, \Delta_1), \ldots, (S_r, \Delta_r)\}$, where the S_i form a disjoint

partition of S, each Δ_i is a simplex containing S_i, and $|S_i| \leqslant 2n/r$ for all i. The *crossing number* of $\Psi(S)$ is the maximum number of simplices intersecting any hyperplane. Matoušek [63] has shown that any set of points admits a simplicial partition of size r with crossing number $O(r^{1-1/d})$.

The partition tree answers a simplex-query in $O(n^{1-1/d} \log^{O(1)} n)$ time and, hence, it can be used to answer DOP-queries in the same time bound. However for point queries no better bound is known.

2.1.2 c-oriented kd-tree

One way to extend a kd-tree to support c-DOP-queries is to lift the point set S to c dimensions and then build an axis-aligned kd-tree $\mathcal{T}$ in c dimensions. More precisely let $\mathcal{O}$ be the origin. We map every point $p \in S$ in $\mathbb{R}^d$ to a point $p' = (x_1 \cdots x_c)$ in $\mathbb{R}^c$, where x_i is defined as the signed distance[1] from $\mathcal{O}$ to the projection of p on a line through $\mathcal{O}$ whose orientation is o_i. Let $Slab_i(x_i^-, x_i^+)$ be the slab in $\mathbb{R}^d$ bounded by two hyperplanes with normal o_i at distance x_i^- and x_i^+ from $\mathcal{O}$. Any DOP D is defined by $\bigcap_{i=1}^{c} Slab_i(x_i^-, x_i^+)$ where $Slab_i(x_i^-, x_i^+)$ is the smallest slab enclosing D, see Figure 2.2. In $\mathbb{R}^c$ a query-DOP D now corresponds to an axis-aligned box $D' = \bigcap_{i=1}^{c} [x_i^-, x_i^+]$. Note that any point in $\mathbb{R}^c$ contained in D' corresponds to a point contained in D in $\mathbb{R}^d$ and vice versa. We can thus answer a DOP-query Q in $\mathbb{R}^d$ by mapping it to an axis-aligned box D' in $\mathbb{R}^c$ in constant time and then query the $\mathcal{T}$ with D' in $O(n^{1-1/c})$ time. Similarly a point query q can be answered in $O(\log n)$ time.
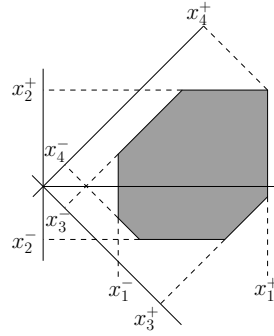
Figure 2.2: A 4-DOP defined by defined by $\bigcap_{i=1}^{4} [x_i^-, x_i^+]$.

Alternatively the set of axis-aligned orientations used in the kd-tree construction can be replaced by the set $\mathcal{C}$. The resulting kd-tree is basically the same as in the first approach (assuming that the order in which the orientations are used

---

[1]The signed distance between two points is defined as the distance between the points with an additional sign. This sign indicates the order of the two points when projected on the x-axis (or y-axis when both points have the same projection on the x-axis). The sign is negative when the projection of the second point has a lower abscissa.

to cut the remaining set is the same), since the mapping preserves the order of the points along each orientation.

2.2   *c*-grid BSP

In this section we show how to construct a c-oriented BSP for S with properties similar to the simplicial partition of Matoušek: every cell in the BSP contains at most n/r points, there are $O(r)$ cells, and every c-oriented hyperplane intersects $O(r^{1-1/d})$ cells.

One important difference with the result of Matoušek is that we only bound the crossing number with respect to c-oriented hyperplanes, not with respect to arbitrary hyperplanes. In this sense, our result is weaker than Matoušek's result. On the other hand, our partitioning is a c-oriented BSP whereas Matoušek uses arbitrary simplices that can even intersect. This means that the crossing number of a point—the maximum number of cells containing the point—is 1 for our partitioning, whereas no better bounds than $O(r^{1-1/d})$ are known for Matoušek's partitioning. An additional advantage is that our partitioning algorithm is much simpler.

Lemma 2.1 *Let S be a set of n points in $\mathbb{R}^d$. For any $r \leq n$ and constant $c \geqslant d$ there exists a c-oriented BSP for S with the following properties:*

   i) each cell in the BSP contains at most n/r points

   ii) the number of cells in the BSP is $O(r)$

   iii) the depth of the corresponding BSP tree is $O(\log r)$

   iv) any hyperplane h with an orientation in $\mathcal{C}$ intersects at most $O(r^{1-1/d})$ cells in the BSP.

The partitioning can be constructed in $O(n \log r)$ time when the points of S are given as c lists $S_1, ..., S_c$, where S_i contains all points of S sorted by their projections on a line orthogonal to the i-th orientation in $\mathcal{C}$.

Proof: The basic idea is the following, see Figure 2.3. Consider the i-th orientation, o_i, in $\mathcal{C}$. Take a set $\mathcal{H}_i$ of $r^{1/d} - 1$ splitting hyperplanes with that orientation, such that there are $n/r^{1/d}$ points from S in each of the slabs defined by the hyperplanes.[2] Do this for each of the c orientations. The hyperplanes from $\mathcal{H}_1 \cup \cdots \cup \mathcal{H}_c$ together define a "grid" with $O(r)$ cells. In the second stage, we partition each cell that contains more than n/r points into cells with at most n/r and at least $n/(2r)$ points. It is not difficult to see that we can get a BSP from this strategy with properties (i)–(iv). The main complication is to achieve the desired running time of the construction. Next we describe the details.

---

[2]For simplicity we assume here and in the sequel that the set of points is in general position. More precisely, we assume that no two points lie on a common line with one of the c orientations.
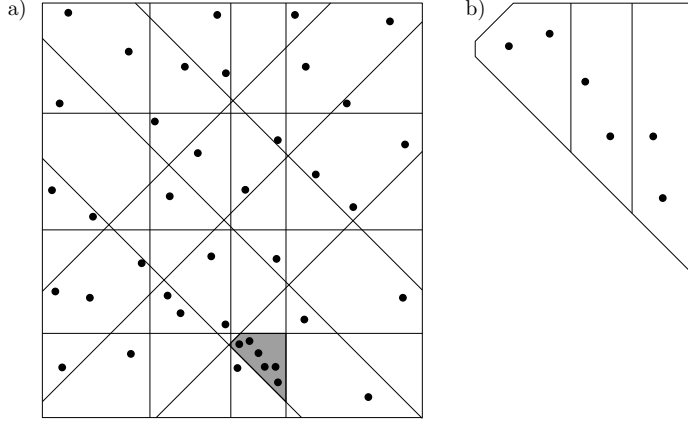
Figure 2.3: An example of the partitioning of $\mathbb{R}^2$ by a c-grid BSP on a set of 40 points with $r = 16$. In a) the grid is shown which is created in the first step of the construction. Some cells in the partitioning are empty and some contain more than $40/16 < 3$ points. A cell C containing too many points is depicted in gray. b) shows the partitioning of C in the second step.

Stage 1. We first scan, for each $i \in \{1, \ldots, c\}$, the list S_i to find an ordered set $\mathcal{H}_i$ of $r^{1/d} - 1$ splitting hyperplanes such that the slabs between two consecutive hyperplanes in $\mathcal{H}_i$ (and the "slabs" before the first and after the last hyperplane in $\mathcal{H}_i$) each contain at most $n/r^{1/d}$ points. We then re-order the hyperplanes from each $\mathcal{H}_i$: we construct a balanced binary tree $\mathcal{T}_i$ on $\mathcal{H}_i$, and output the hyperplanes in the order given by a pre-order traversal of $\mathcal{T}_i$. We still use $\mathcal{H}_i$ to denote the re-ordered set.

Next we start the construction of the BSP. We initialize a BSP tree $\mathcal{T}$ consisting of a single leaf node. We then insert the hyperplanes from $\mathcal{H}_1, \ldots, \mathcal{H}_c$ one by one. (It is not important that all of the hyperplanes from $\mathcal{H}_1$ are inserted before we start with $\mathcal{H}_2$, etc. One can also treat the sets $\mathcal{H}_i$ in a round-robin fashion, as long as the ordering within each $\mathcal{H}_i$ is respected.) To insert a hyperplane h, we traverse $\mathcal{T}$ top-down to locate all the leaf cells that are intersected by h, as follows. Each (internal or leaf) node ν of the BSP corresponds to a region $R(\nu)$ in $\mathbb{R}^d$: the root corresponds to $\mathbb{R}^d$, the left and right child of the root to the half-spaces to the left resp. right of the splitting hyperplane at the root, etc. When we arrive at a node ν during the traversal, we make sure that we have $h \cap R(\nu)$ available. If $h \cap R(\nu)$ does not intersect the splitting hyperplane $h(\nu)$ stored at ν, then we continue the traversal in the appropriate child, otherwise we use $h(\nu)$ to cut $h \cap R(\nu)$ into two pieces and continue the traversal in both children with the appropriate piece. Note that since the regions are c-DOPs, the pieces $h \cap R(\nu)$ have constant complexity, so each node is handled in $O(1)$ time. Each leaf

that we reach is replaced by a node with splitting plane h and two leaves as children.

Stage 2. We first distribute the points of S to the resulting cells by searching in the BSP with each point. When a cell contains more than n/r points—we say that it is *overfull*—we construct a balanced BSP on those points using parallel hyperplanes with an arbitrary orientation from C, until the number of points in each subcell drops below n/r. Note that this implies that each subcell created in this manner has at least $n/(2r)$ points. We let this tree replace the leaf of $\mathcal{T}$ corresponding to the overfull cell.

This BSP has property (i) by construction. Property (ii) follows from the fact that the first stage produces $O(r)$ cells, and the second stage produces only cells with at least $n/(2r)$ points.

For property (iii) consider the tree $\mathcal{T}_i'$ obtained by taking the nodes in the BSP which have a partition plane of orientation $o_i \in \mathcal{C}$ and connecting the nodes if one is a descendant of the other. Since the hyperplanes are inserted according to a pre-order on $\mathcal{T}_i$, any path in $\mathcal{T}_i'$ corresponds to a path in $\mathcal{T}_i$, with maybe some nodes removed from this path. The tree $\mathcal{T}_i'$ thus has at most the same depth as $\mathcal{T}_i$, namely $O(\log r)$. Since this holds for all orientations, the tree resulting after Stage 1 has depth $O(c \log r) = O(\log r)$. The balanced binary trees created in the second step, which replace the leaves of the BSP corresponding overfull cells, also have depth $O(\log r)$ and so the property follows.

To prove property (iv), we note that the number of cells intersected by any c-oriented hyperplane h after the first stage is bounded by the complexity of the arrangement on h induced by the partitioning, which is $O(((c-1)r^{1/d})^{d-1}) = O((c-1)^{d-1}r^{1-1/d})$. It remains to account for the increase in the number of intersected cells due to the second stage. To this end, we observe that the total number of points in the intersected cells is at most $n/r^{1/d}$, since h lies inside one of the slabs induced by the splitting hyperplanes parallel to h. Because the second stage only produces cells with at least $n/(2r)$ points, this implies that there cannot be more than $2r^{1-1/d}$ such cells.

We are now left with proving the construction time of the partitioning. Scanning the sorted lists S_i to obtain the sets $\mathcal{H}_i$, building the c binary trees $\mathcal{T}_i$, and doing the pre-order traversal of each of them takes $O(cn)$ time in total. Initializing the BSP tree $\mathcal{T}$ takes constant time. Since the depth of the BSP tree $\mathcal{T}$ is $O(\log r)$, inserting a hyperplane h takes $O(m_h \log r)$ time, where m_h is the number of leaves where h has to be inserted. Since this number is equal to the total complexity of the arrangement induced on h by the previously inserted hyperplanes, we have $m_h = O(r^{1-1/d})$. Hence, in total we spent $O(r \log r)$ time, so Stage 1 takes $O(n + r \log r)$ time.

The distribution of the points of S to the resulting cells can subsequently be done in $O(n \log r)$ time using the BSP. Let n_j be the number of points in a cell j of the partitioning. The construction of the BSPs for the overfull cells takes $\sum_i O(n_i) = O(n)$ time, where the sum is over all overfull cells. The overall construction time is $O(r \log r + n \log r)$ and since $r \leq n$ the bound follows. □

To obtain a BSP that can answer DOP-queries efficiently, we recursively apply Lemma 2.1 (with a suitable value of r—see below) to each cell in the partitioning that contains more than one point. This is similar to the way simplicial partitions are used to construct partition trees.

Theorem 2.2 *Let S be a set of n points in $\mathbb{R}^d$. For any $\varepsilon > 0$, there is a c-oriented BSP tree $\mathcal{T}$ that has size $O(n)$, has depth $O(\log n)$, and can answer DOP-queries in $O(n^{1-1/d+\varepsilon} + k)$ time, where k is the number of reported points. The BSP tree $\mathcal{T}$ can be constructed in $O(n \log n)$ time.*

Proof: Think about the tree not as a BSP, but as a tree with fan-out $O(r)$: the root of the tree has $O(r)$ children, each corresponding to a cell in the partitioning of Lemma 2.1, each of these children has $O(r)$ children corresponding to the cells in the recursively created partitionings, and so on. Since r is a constant—see the end of the proof—the conversion to a binary tree does not influence the asymptotic bounds. By property (i) of Lemma 2.1, this tree has depth $O(\log_r n) = O(\log n)$, and it is easy to see that its size is $O(n)$.

To analyze the query time, we distinguish between nodes whose associated region is contained in the query range and nodes whose region is intersected by the boundary of the range. The number of nodes of the former type is $O(k)$. To bound the number of nodes of the latter type, we formulate a recurrence for $Q(n)$, the number of nodes whose region is intersected by a fixed facet f of the query range; multiplying this number by $2c$ then gives the bound. By Lemma 2.1, the facet f intersects at most $a \cdot r^{1-1/d}$ regions of the children of the root node, for some constant a. Each such region contains at most n/r points, so we have $Q(n) \leqslant 1 + ar^{1-1/d} \cdot Q(n/r)$. For $r = a^{1/\varepsilon}$, this solves to $Q(n) = O(n^{1-1/d+\varepsilon})$.

During the construction of the partitioning as in Lemma 2.1 it is easy to keep the points in the cells in sorted order for all c orientations. The preprocessing step thus has to be performed only once. From the same lemma we know that a partitioning on n_i points can be constructed in $O(n_i \log r)$ time. At each level a point can be in only one node, so the construction time of all partitionings at a level i is $\sum_j O(n_j \log r) \leq O(n \log r)$. Since there are $O(\log_r n)$ levels in the tree the construction time is $O(n \log n)$. $\square$

We call the BSP constructed as described above a c-grid BSP.

Remark 2.3 The recurrence for $Q(n)$ still solves to $O(n^{1-1/d+\varepsilon})$ if any single region of a child of root may contain more than n/r points, but the regions that intersect f together still contain at most $n/r^{1/d}$ in total—that is, when we have $Q(n) \leqslant 1 + \sum_{i=1}^{j} Q(n_i)$ with $j \leqslant ar^{1-1/d}$ and $\sum_{i=1}^{j} n_i \leqslant n/r^{1/d}$. Therefore we can maintain the same query times if, in the partitioning algorithm of Lemma 2.1, we only build the grids, and omit the second stage of subdividing cells containing more than n/r points by means of parallel hyperplanes.

Chapter 3

A Framework for BSPs on Line Segments

In this chapter we describe a general technique to construct a BSP for a set S of n disjoint line segments in the plane, 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 [71]. The range-searching structure of Overmars et al. [70] also uses similar ideas, except that they store segments spanning the entire region of a node in an associated structure, so they do not construct a BSP for the segments.

The following theorem summarizes the result of this chapter. It is proved in Sections 3.1 – 3.3.

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 with a range Q 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 range query 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.

In Section 3.4 we give two instantiations of the framework. The first instantiation yields a BSP on line segments for c-oriented range searching and the second a BSP for approximate range searching.
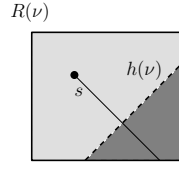
Figure 3.1: The segment s is short in the region R(ν) of a node ν. The splitting line $h(\nu)$ stored at ν cuts s in two. There is no endpoint of s in the new dark gray region and so s is long for that region. In the light gray region s is still short.

3.1   From BSP-trees on points to BSP-trees on line segments

Let S be a set of n disjoint segments in $\mathbb{R}^2$, 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 $\mathcal{T}_S$ on S. Some details of the construction are omitted here; they are 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 $\mathcal{T}_P$ and let R(ν) denote the region associated with ν. We call a segment $s \in S$ *short* at ν if R(ν) contains an endpoint of s, see Figure 3.1. A segment s is called *long* at ν if (i) s intersects the interior of R(ν), and (ii) s is short at parent(ν) but not at ν. Note that a long segment spans the interior of R(ν) that is, it intersects two edges of R(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 R(ν). The subtree rooted at ν is denoted by $\mathcal{T}_P(\nu)$. The recursive call constructs a BSP tree $\mathcal{T}_{S^*}$ for S^* based on $\mathcal{T}_P(\nu)$. Initially, ν is the root of $\mathcal{T}_P$ and $S^* = S$. The recursion stops when S^* is empty, in which case $\mathcal{T}_{S^*}$ is a single leaf.

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

Case 1: L is empty.   Let $h(\nu)$ be the splitting line stored at node ν in $\mathcal{T}_P$. 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.
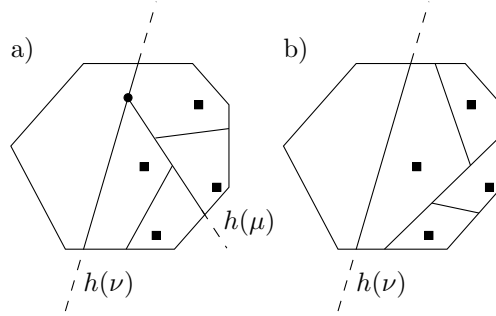
Figure 3.2: Illustration of the pruning strategy. The black squares indicate end-points of input segments. a) There is a T-junction on $h(\nu)$: a splitting line in the subtree $h(\mu)$ ends on $h(\nu)$. Removing $h(\nu)$ would partition the empty part of the region, which might have a negative effect on the query time. b) There is no T-junction on $h(\nu)$, and $h(\nu)$ can therefore be removed.

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 and should therefore be omitted in $\mathcal{T}_{S^*}$, because otherwise the size of the tree becomes too large. 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 Figure 3.2, 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 $h(\mu) \cap \mathrm{R}(\mu)$, 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 $\mathcal{T}_{S^*}$ stores the splitting line $h(\nu)$, one of its subtrees is empty, and the other subtree is constructed recursively on the non-empty subset. If $h(\nu) \cap R$ does not contain a T-junction, however, we prune: the tree $\mathcal{T}_{S^*}$ is the tree we get when we recurse on the non-empty subset, and there is 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 become the roots of the subtrees to be created for the sets S_i^*. To keep the overall depth of the tree bounded by $O(\mathrm{depth}(\mathcal{T}_P))$, we make $\mathcal{T}$ weight-balanced, where the weights correspond to the sizes of
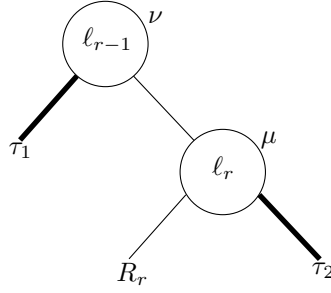
Figure 3.3: Construction of the weight balanced tree $\mathcal{T}$. Splitting lines ℓ_{r-1} and ℓ_r are two long segments which are stored in segment nodes, respectively the root of $\mathcal{T}$ and at the root of the right child of $\mathcal{T}$. In the subtrees τ_1 and τ_2 long segments might have to be inserted. The regions associated with τ_1 and τ_2 contain each at most half of the short segments in $R(\mathcal{T})$. The root node for R_r must be a partition node.

the sets S_i^*, as in the trapezoid method for point location [74]. The tree $\mathcal{T}$ is constructed as follows. Let $\ell_i \in L$ separate the regions R_i and R_{i+1}. If $\sum_{i=1}^{m} |S_i^*| = 0$ simply build a balanced 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 $\sum_{i=1}^{r} |S_i^*| \geq |S^* \setminus L|/2$. The long segment 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 3.3. The left child of ν, τ_1, and the right child of μ, τ_2 are constructed recursively. Note that both $R(\tau_1)$ and $R(\tau_2)$ contain at most $|S^* \setminus L|/2$ 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 T_i for S_i^*. More precisely, each 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 3.2 *The size of $\mathcal{T}_S$ is $O(n \cdot depth(\mathcal{T}_P))$.*

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 is cut into $O(depth(\mathcal{T}_P))$ fragments. Hence, the number of segment nodes is $O(n \cdot depth(\mathcal{T}_P))$.

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

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

Lemma 3.3 *The depth of $\mathcal{T}_S$ is $O(\text{depth}(\mathcal{T}_P))$.*

Proof: This follows more or less from standard arguments [74], 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 $\text{depth}(\mathcal{T}_P)$. It remains to bound the number of segment nodes on any path. There are two cases to consider. In the first case there are no more segments to be inserted and a balanced binary tree is created on the long segments which has depth $O(\log n)$. In the second case we look at the process of replacing a multi-way node with a binary subtree, and establish an invariant that proves the bound. Call an edge in $\mathcal{T}_S$ from a node ν to a node μ *black* when the number of (short) segments in $\text{R}(\mu)$ is at most half the number of (short) segments in $\text{R}(\text{parent}(\nu))$. 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 Figure 3.3 where the black edges are drawn fat. Note that the subtree for R_r must have a partition node as root node. On any path, after two consecutive segment nodes either there is a black edge or the next node on a path is a partition node. 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 $\text{depth}(\mathcal{T}_P) = \Omega(\log n)$, the bound on the number of segment nodes is $O(\text{depth}(\mathcal{T}_P))$. $\quad\square$

3.2 Analysis of the query time

A node in a BSP tree is visited by a query range Q if Q intersects $\text{R}(\nu)$. Next we bound the number of visited nodes, in terms of the number of visited nodes in $\mathcal{T}_P$ when we would query $\mathcal{T}_P$ with Q.

Lemma 3.4 *Let Q be a constant-complexity 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. When we query with Q the number of visited nodes in $\mathcal{T}_S$ is bounded by $O((v(\mathcal{T}_P, Q) + k)\cdot \text{depth}(\mathcal{T}_P))$, where k is the number of segments intersecting Q.*

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

We first bound the number of leaves of the first category, that is, the number of cells intersected by ∂Q. Note that the number of cells intersected by ∂Q is at most one more than the number of intersections between cell boundaries and ∂Q. We made sure that the pruning step in our construction did not cause splitting

'lines' to be extended. All cell boundaries intersected by ∂Q, except for new cell boundaries due to the inserted segments, were thus already intersected in the subdivision induced by $\mathcal{T}_P$. The number of segments in S intersected by ∂Q is $O(k)$ and 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 $\text{R}(\nu) \subset Q$ but $\text{R}(\text{parent}(\nu)) \not\subset Q$. 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 $O(k_l(\nu) + k_s(\nu) \cdot \text{depth}(\mathcal{T}_S(\nu)))$ since a short segment is split into at most $O(\text{depth}(\mathcal{T}_S(\nu)))$ fragments and a long segment by construction is not split at all. Hence, the total number of nodes in the subtrees $\mathcal{T}_S(\nu)$ rooted at the nodes $\nu \in N(Q)$ is bounded by

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

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 is 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. $\qquad\square$

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

Assuming that Q is convex, an alternative is to store with each node ν some additional information. In particular, we store two lines that are tangent to $\text{R}(\nu)$ at the two endpoints of $h(\nu) \cap \text{R}(\nu)$. When we come to a node ν such that $\text{R}(\nu)$ intersects Q, we can then proceed as follows. If Q intersects $h(\nu) \cap \text{R}(\nu)$—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. With this approach, the query time for convex ranges is asymptotically the same as the number of visited nodes. The storage requirements of the BSP tree increases by about a factor two, however.
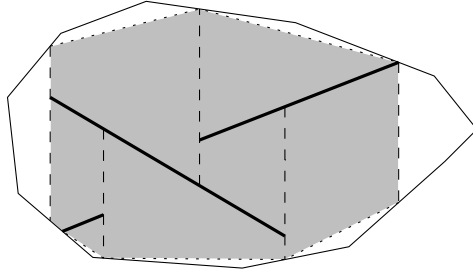
Figure 3.4: Example of a trapezoidal decomposition of the segments clipped to a region. The trapezoids inside the region are shaded.

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 $O(n \cdot \mathrm{depth}(\mathcal{T}_P))$ time. 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 $O(n \cdot \mathrm{depth}(\mathcal{T}_P))$ time in total.

Secondly, we maintain a trapezoidal decomposition of the current region R induced by the segments in S^*, where S^* is the current subset we are dealing with. We clip this trapezoidal decomposition to lie within the current region. Note that the region boundary does not play a role in the trapezoidal 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 Figure 3.4. Computing the initial trapezoidal decomposition takes $O(n \log n)$ time. It is stored as its dual. The dual of the trapezoidal decomposition is a graph $\mathcal{G}(S^*)$ whose nodes correspond to trapezoids in the trapezoidal decomposition. There are arcs between neighboring trapezoids. Maintaining the dual of the trapezoidal 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 trapezoidal decompositions is $O(\sum(|S^*|))$, where the sum is over the sets S^* arising in all recursive calls.

Recall that $L \subseteq S^*$ is the set of segments from S^* that are long at ν. 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 $\mathrm{R}(\nu)$ in this case, we account for this time

by charging $O(1)$ to each endpoint in $\mathcal{T}_P(\nu)$. This way an endpoint is charged $O(\text{depth}(\mathcal{T}_P))$ times, so the overall time is $O(n \cdot \text{depth}(\mathcal{T}_P))$.

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 of the long segments. Delete all arcs crossing the long segments and label each node incident to such an arc by the region containing it. 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 labeled by the region in which it is contained, we obtain 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 [74] 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 3.5 *The construction of the* BSP *tree* $\mathcal{T}_S$ *from the tree* $\mathcal{T}_P$ *can be done in* $O(n \cdot \text{depth}(\mathcal{T}_P))$ *time.*

3.4   Instantiations

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 and Welzl [48] as underlying structure we can get a BSP on a set of n disjoint line segments whose query time is $O(n^{2/3+\varepsilon} + k \log n)$.

In this section we focus on the following two applications. The first application is for c-oriented range searching in a set of line segments in the plane. For

this application we use the c-grid BSP developed in Section 2.2. The second application is approximate range searching in a set of line segments in the plane. For this we use a BAR-tree [38].

Both trees are BSPs, where all splitting lines have orientations that come from a fixed set of predefined possibilities. For example, a BAR-tree on points in the plane uses as orientations the horizontal, vertical, and the two diagonal directions ($45°$ and $135°$). Hence, the regions in both BSPs have constant complexity. This also holds for the regions we get when we build either of these BSPs on the endpoints of a set of segments and transform it into a BSP for the segments; such regions can only be twice as complex as the original regions since the boundary of the cell cannot have two consecutive edges which are on an input segment.

We can extend the result for both BSPs to get BSPs for 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 there are only a constant possible orientations, a curve is cut at most into a constant number of pieces.

3.4.1 c-grid BSP

The c-grid BSP developed in Section 2.2 answers DOP range queries in $O(n^{1/2+\varepsilon})$ time and has depth $O(\log n)$, see Theorem 2.2. The number of cells of the c-grid BSP subdivision intersected by the boundary of a DOP-query can trivially be bounded by $O(n^{1/2+\varepsilon})$ as well. Using the c-grid BSP in the general framework we thus obtain the following result.

Corollary 3.6 *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 for S of size $O(n \log n)$ and depth $O(\log n)$ such that any DOP range query Q can be answered in time $O(n^{1/2+\varepsilon} + k \log n)$, where k is the number of curves intersecting Q.*

3.4.2   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 ensures that a BAR-tree has excellent query time for approximate range queries—see Duncan's thesis [37] 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 $O(1 + (1/\varepsilon) + k_\varepsilon)$, where k_ε is the number of points inside the extended query range. The extended query range Q_ε is the locus of points lying at distance at most $\varepsilon \cdot \mathrm{diam}(Q)$ from Q, where $\mathrm{diam}(Q)$ is the diameter of Q. As noted by Haverkort et al. [50], Q_ε is only used in the analysis of the approximate range

query time and not by the query algorithm. The analysis therefore holds for any $\varepsilon > 0$ and we obtain the following corollary.

Corollary 3.7 *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 an exact 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.*

Chapter 4

The oBAR-tree: a BSP on General Objects

The lower bound of $\Omega(n \log n / \log \log n)$ by Tóth [85] on the size of a BSP on line segments in the plane shows that for general objects not all input sets admit a linear-size BSP. In practice even a super-linear storage might be too large, but still BSPs are widely used. Hence, despite the super-linear lower bound, BSPs in practice seem to have linear size. Realistic input models, see Section 1.2.2, can account for this gap between theory and practice. In this chapter we develop a BSP using a realistic input model.

De Berg [19] 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 in $O(\log n)$ time as well [19]—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. In this chapter we improve these results.

Recall from Section 1.2.2 that the *density* of a set S of objects is the smallest number λ such that the following holds: any ball B is intersected by at most λ objects $o \in S$ with $\rho(o) \geqslant \rho(B)$ [24], where $\rho(o)$ is the radius of the smallest enclosing ball of an object o. If S has density λ, we call S a λ-low-density scene.

In this chapter we develop a linear-size BSP, based on the BAR-tree described in Section 1.3.1, for λ-low-density scenes without any additional structures, which supports approximate range searching in $O(\log n + \varepsilon^{1-d} + k_\varepsilon)$ time and point location in $O(\log n)$ time where k_ε is the number of objects intersecting the extended query range. The extended query range Q_ε is the locus of points lying at distance at most $\varepsilon \cdot \operatorname{diam}(Q)$ from Q, where $\operatorname{diam}(Q)$ is the diameter of Q. Our overall
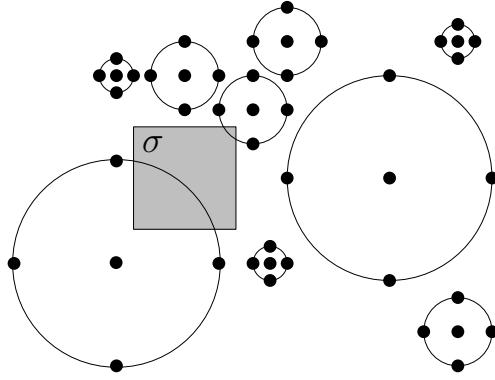
Figure 4.1: A 4-guarding set of size $5n$ for a set of n disjoint discs against squares.

strategy, also used by de Berg [19], is to compute a suitable set of points that will guide the construction of the BSP-tree. The construction of a suitable set of points is the topic of Section 4.1. How this set is then used to construct a BSP-tree is explained in Section 4.2.

4.1   A guarding set against BAR-tree cells

A guarding set [20] for a collection of objects can be seen as a set of points that approximates the distribution of the objects. More precisely, let S be set of objects in $\mathbb{R}^d$, let $\mathcal{R}$ be a family of subsets of $\mathbb{R}^d$ called ranges (for instance hypercubes). A set $\mathcal{G}$ of points is called a κ-guarding set for S against $\mathcal{R}$ for some positive integer κ, if any range from $\mathcal{R}$ not containing a point from $\mathcal{G}$ intersects at most κ objects from S. See Figure 4.1 for an example of a guarding set against squares for disjoint disks in the plane.

In the next section we construct a BAR-tree on a suitable guarding set of S to obtain a linear-size BSP whose leaves store a constant number of objects and can answer an approximate range query efficiently. We cannot use the bounding-box vertices of the objects in S as a guarding set as done by De Berg [19], 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* [20] against BAR-tree cells, and we call the points in G *guards*.

First we describe how to construct a guarding set for λ-low-density scenes in $\mathbb{R}^d$. The constants in this construction are however large: $\kappa = (5d)^d\lambda$, 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 in Section 4.1.2.
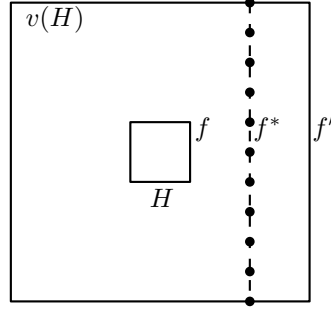
Figure 4.2: The guards for a facet f of a hypercube H in $\mathbb{R}^2$.

4.1.1 Construction of a guarding set

We base our guarding set on the following construction by De Berg et al. [20] against β-fat convex ranges.

Let H be a hypercube. The *vicinity* $v(H)$ of H is the hypercube obtained by scaling H by a factor of 5 around its center, see Figure 4.2. Let f be a facet of H and f' be the corresponding facet of $v(H)$. Let f^* be a translated copy of f' inside $v(H)$ such that it is midway between f and f'. The facet f^* is subdivided into a regular grid G of M^{d-1} cells where

$$M = 1 + \left\lceil \frac{5^{d-1}(d-1)(2^{d+1}-2)(3/2)^d}{\omega_d\beta} \right\rceil$$

and ω_d a constant such that the volume of a hypersphere of radius r in $\mathbb{R}^d$ is $\omega_d r^d$. The guarding set for f are the $(M+1)^{d-1}$ vertices of the grid on f^*. The guarding set G_H for H is the union of all $2d$ guarding sets of its facets. De Berg et al. prove the following lemma, which gives an upper bound on the fatness of a convex region R which does not contain a guard from G_H and which is not fully contained in $v(H)$.

Lemma 4.1 [20] *Let H be a hypercube in $\mathbb{R}^d$, let $v(H)$ be its vicinity, and let G_H be the set of guards defined for H as described above. Let R be a convex $\beta' \leq \beta$-fat range intersecting H and not fully contained in $v(H)$. Then*

$$\beta' \leq \frac{5^{d-1}(d-1)(2^d-2)(3/2)^d}{\omega_d M d}.$$

Using this lemma we can now prove the following theorem.

Theorem 4.2 *Let S be a λ-low-density scene of n objects in $\mathbb{R}^d$. In $O(n)$ time one can compute a $(5d)^d\lambda$-guarding set of size $2d(M+1)^{d-1}n$ for S against* BAR-*tree cells.*

Proof: Let $H(o)$ be the smallest hypercube enclosing an object o and let $G_{H(o)}$ be the set of $2d(M+1)^{d-1}$ guards for $H(o)$ obtained as described above, where for β we use the minimum fatness of a BAR-tree cell. $G_{H(o)}$ can be constructed in constant time for constant d. The guarding set for S is obtained by taking the union of $G_{H(o)}$ for each $o \in S$. The bound on the construction time and the size of the guarding set follow immediately.

By the choice of M the fatness of any BAR-tree cell is larger than the upper bound of Lemma 4.1 and therefore a BAR-tree cell C, that intersects o but does not contain a guard from $G_{H(o)}$, is completely contained within $v(H(o))$, and thus we get:

$$\rho(C) \leq \rho(v(H(o))) \leq 5\rho(H(o)) \leq 5\sqrt{d}\rho(o).$$

We cover C with balls having radius $\rho(C)/5\sqrt{d}$. We overestimate the number of balls we need by covering the smallest hypercube H_B that contains C by the largest hypercube H_b that is contained in a ball of radius $\rho(C)/5\sqrt{d}$. The edge length of H_B is $r_B = 2\rho(C)$ and $r_b = 2\rho(C)/5d$ is the edge length of H_b. We need at most $(r_B/r_b)^d = (5d)^d$ hypercubes. Any object o intersecting C will intersect one of the balls, and $\rho(o)$ will be at least the radius of that ball. Because S has density λ, this implies that the number of objects intersecting C is bounded by $(5d)^d\lambda$. $\qquad\square$

In the next section we describe for $\mathbb{R}^2$ a smaller guarding set against BAR-tree cells.

4.1.2   A small guarding set in the plane

Let S be a λ-low-density scene in $\mathbb{R}^2$. Our guarding set G has 12 guards for each object $o \in S$ (denoted G_o), 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 Figure 4.3 a).

Lemma 4.3 *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 it suffices 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 Figure 4.3 a).

If the center of L is contained in ξ, then the radius of L is at most $\frac{1}{2}\sqrt{2}$ since otherwise L would contain at least one guard of o. For the remainder of this proof let the center of L be outside ξ. Since BAR-tree cells are convex, there is an edge e of ξ which 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, together with (a part of) e, form an isosceles right triangle, because a BAR-tree only uses splitting
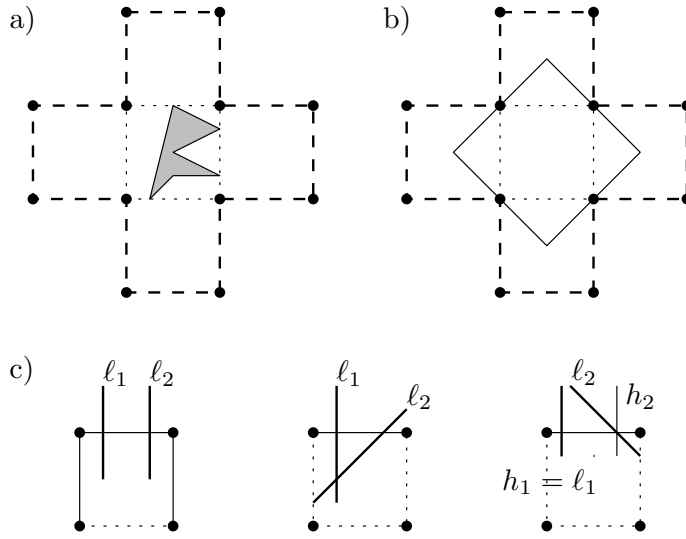
Figure 4.3: Creating a guarding set against BAR-tree cells. a) An object o with its guards. The bounding square $\sigma(o)$ of o is drawn with a dotted line. The union ξ of the squares placed at each edge of the bounding square is drawn with a dashed line. b) An example of a cell whose largest inscribed ball has radius $\sqrt{2}\rho(o)$. c) The three possible ways a cell can intersect the boundary of ξ. For each possibility one of the dashed edges of the square is an edge of $\sigma(o)$.

lines that are horizontal, vertical or diagonal. There are three possibilities, see Figure 4.3 c):

1. ℓ_1 *and* ℓ_2 *are parallel.* The radius of L is at most $1/2$ in this case 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 cannot actually occur, since we assume that C intersects 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 and ℓ_2, respectively. The isosceles right triangle formed by ℓ_1, ℓ_2 and e lies between these two lines and therefore L also lies between them. Since the distance between h_1 and h_2 is at most 1 the radius of L is at most $1/2$.

$\square$
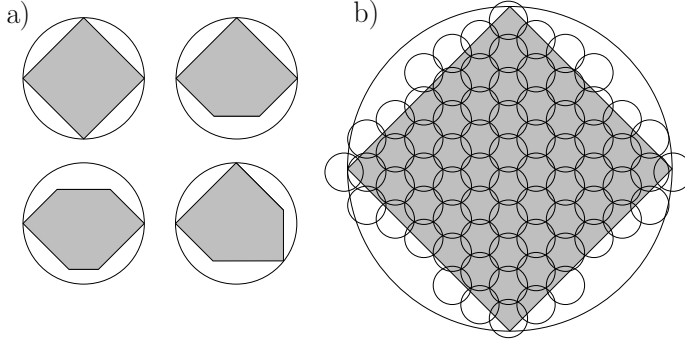
Figure 4.4: a) The possible shapes of the BAR-tree cells fitting in a disk D of radius ρ. The square shaped cell C is the largest cell possible in D. b) An example of a covering of C by 72 disks of radius $\rho/(6\sqrt{2})$. The largest BAR-tree cell C has edge length $\sqrt{2}\rho$. The largest enclosing square in a disk of radius $\rho/(6\sqrt{2})$ has edge length $\rho/6$, so we need 72 such squares to cover C.

In Figure 4.3 b) 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 Lemma 4.3 and the bounded aspect ratio of BAR-tree cells, we can prove that G is a good guarding set.

Theorem 4.4 *Let S be a λ-low-density scene of n objects in $\mathbb{R}^2$. In $O(n)$ time one can compute 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 [38]. Using the previous lemma, we get the following, if we choose $\alpha = 6$, for any object o intersecting a cell C not containing a guard from G_o:

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

We now cover C by 72 disks of radius $\rho(C)/(6\sqrt{2})$, see Figure 4.4. Any object o intersecting C will intersect at least one of the 72 disks, 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. □

4.2   A linear-size BSP on objects

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 [19] with the BAR-tree of Duncan et al. [38]. We will call our BSP-tree an *object* BAR-*tree,* or OBAR-tree for short. The following theorem summarizes our result.

Theorem 4.5 *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) *an approximate range query with a convex range Q takes $O(\log n + \lambda \cdot \{(1/\varepsilon)^{d-1} + k_\varepsilon\})$ 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.

We begin by describing the construction algorithm.

1. Construct a κ-guarding set G for S, as explained in Section 4.1, where $\kappa = 72\lambda$ for $d = 2$ and $\kappa = (5d)^d\lambda$ for $d \geq 3$. The construction of the guarding set is done by generating 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. [38], 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 the paper by De Berg [19], 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.

We will first bound the depth, size, query time and the construction time in terms of the size of the guarding set G. In section Section 4.1 we showed how to construct a guarding set of size $O(n)$ such that Theorem 4.5 follows.

Lemma 4.6 *Let $\mathcal{T}_S$ be an OBAR-tree on a set S of n objects in $\mathbb{R}^d$ built as described above. 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, see Theorems 4.2 and 4.4, and the construction of

the BAR-tree takes $O(|G| \log |G|)$ time [38]; 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 $O(\kappa \cdot |G| \log |G|)$. $\qquad\square$

We are now left with proving a bound on the query time.

Lemma 4.7 *An approximate range query in $\mathcal{T}_S$ with a constant-complexity convex query range Q takes $O(\log |G| + \kappa \cdot \{(1/\varepsilon)^{d-1} + k_\varepsilon\})$ time, where k_ε is the number of objects intersecting the extended query range Q_ε.*

Proof: 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. Let $\mathrm{R}(\nu)$ be the region associated with the node ν. We distinguish two categories of visited nodes: nodes ν such that $\mathrm{R}(\nu)$ is intersected by ∂Q (the boundary of Q), but where $\mathrm{R}(\nu) \not\subset Q_\varepsilon$, and nodes ν such that $\mathrm{R}(\nu) \subset Q_\varepsilon$.

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 $O(\log |G| + 1/\varepsilon^{d-1})$. (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 $O(\log |G| + \kappa/\varepsilon^{d-1})$ time.

Nodes in the second category are organized in subtrees rooted at nodes ν such that $\mathrm{R}(\nu) \subset Q_\varepsilon$ but $\mathrm{R}(\mathrm{parent}(\nu)) \not\subset Q_\varepsilon$. 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. The number of nodes in the subtree is linear in the number of guards in $\mathrm{R}(\nu)$. Because we delete guards of objects not intersecting a region during the recursive construction, the number of guards in $\mathrm{R}(\nu)$ is $O(k(\nu))$. 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 $O(\kappa \cdot k_\varepsilon)$. $\qquad\square$

Chapter 5

BAR-B-tree: an I/O-efficient BAR-tree

In spatial databases building efficient spatial indexes for range searching is a central problem. The R-tree, see Section 1.3.2, has been a popular spatial index thanks to its simplicity, ability to answer various queries, and the flexibility to store spatial objects of different shapes. However, the R-tree is known to be a heuristic-based structure and no guarantees can be made on its query performance. Also many theoretical index structures have been proposed to provide worst-case guarantees, but they either use non-linear storage or incur super-logarithmic query costs.

In this chapter, two disk-based indexes for approximate range searching are introduced which are as versatile as the R-tree, use linear storage and at the same time provide good guarantees on the query performance, albeit in the approximate sense. More precisely, our first index, the BAR-B-*tree*, stores a set of N points in $\mathbb{R}^d$. It answers an approximate range search query Q by accessing $O(\log_B N + \varepsilon^\gamma + k_\varepsilon/B)$ disk blocks for any ε, where B is the block size, $\gamma = 1-d$ for convex queries and $\gamma = -d$ otherwise, and k_ε is the number of points lying within the extended query range. The extended query range Q_ε is the locus of points lying at distance at most $\varepsilon \cdot \mathrm{diam}(Q)$ from Q, where $\mathrm{diam}(Q)$ is the diameter of Q. Our second index, the oBAR-B-*tree*, is able to store spatial objects of arbitrary shapes and provides similar query guarantees if the objects form a low-density scene, see Section 1.2.2. Both our indexes can be bulk-loaded efficiently as well. In addition, we believe that they are also simple enough to be of practical interest.

Approximate range searching. Given the fact that exact range searching either uses non-linear storage or incurs super-logarithmic query time, it is natural to seek for approximate solutions. In Section 1.1.2 the following approximate range searching queries were defined for a set of points P:

- **Approximate range searching.** Return a set P^* such that $\{o \in P : o \cap Q \neq \emptyset\} \subseteq P^* \subseteq \{o \in P : o \cap Q_\varepsilon \neq \emptyset\}$.

- **Approximate range aggregate.** Suppose each point $p \in P$ is associated with a weight $\omega(p) \in \mathbb{R}$, compute $\bigoplus_{p \in P^*} \omega(p)$ for some P^* such that $P \cap Q \subseteq P^* \subseteq P \cap Q_\varepsilon$. The operator $\oplus$ should be associative and commutative. It could for instance be MAX, and then the query will report the maximum-weight point in P^*. If we set $\omega(p) = 1$ for all $p \in P$ choose $\oplus$ to be $+$ (summation), then the query counts the number of points in P^*. We say that the function is *duplicate-insensitive* if $x \oplus x = x$ for any x. For example MAX is a duplicate-insensitive function while $+$ is not.

- **Approximate nearest neighbor.** For a query point q, return a point $p \in P$ such that $d(p,q) \leq (1+\varepsilon)d(p^*,q)$, where p^* is the true nearest neighbor of q and $d(p,q)$ is the (Euclidean) distance between p and q.

These problems were first considered by Arya and Mount [13], who proposed the BBD-tree. Later, a similar, but simpler structure, called the BAR-tree was proposed by Duncan et al. [38]. In this chapter, we extend the BAR-tree to external memory. Furthermore, we generalize the structure to accommodate arbitrary shapes rather than just points, so that our index can serve as an alternative to R-trees.

Our structures are based on the BAR-tree, which is described in Section 1.3.1.

The I/O-model and previous work. For the analysis of external memory data structures, the standard *I/O model* by Aggarwal and Vitter [8] is often used, see Section 1.2.1. In this model, the memory has a limited size M but any computation in memory is free. Only the number of I/O-operations is considered when analyzing the cost of an algorithm. In one I/O-operation a disk block of B items is read from or written to the external memory. The size of a data structure is measured in the number of disk blocks it occupies. Many fundamental problems have been solved in the I/O model. For example, sorting N elements takes $\text{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$ I/O-operations. Please refer to the comprehensive surveys by Vitter [88] and Arge [10] on I/O-efficient algorithms and data structures.

There has been some work on an efficient disk layout of the BAR-tree. Using standard techniques such as a breadth-first blocking scheme and I/O-efficient priority queues, it is pretty straightforward to lay out a BAR-tree on disk such that approximate range aggregate and approximate nearest neighbor can be answered with $O(\log_B N + \varepsilon^\gamma)$ and $O(\log_B N + \varepsilon^{1-d}(1 + \frac{1}{B}\log_{M/B}(1/\varepsilon)))$ I/O-operations, respectively.

In his thesis [37] Duncan gave an I/O-efficient variant of the BAR-tree, which uses a breadth-first blocking scheme. The number of I/O-operations for the construction is $O((N/B)\log N)$ and the number of I/O-operations to answer an approximate range searching query is claimed to be $O(\log_B N + \varepsilon^\gamma + k_\varepsilon/B)$. However Duncan made the implicit assumption that all blocks contain $\Theta(B)$ nodes,

	BAR-B-tree
Size	$O(N/B)$
bulk-load	$O(\text{sort}(N))$
apxRS	$O(\log_B N + \varepsilon^\gamma + k_\varepsilon/B)$
apxRA	$O(\log_B N + \varepsilon^\gamma)$
apxNN	$O(\log_B N + \varepsilon^{1-d}(1 + \frac{1}{B}\log_{M/B}(1/\varepsilon)))$
Update	$O(\log_B N + \frac{1}{B}\log_{M/B}(N/B)\log(N/B))$

Table 5.1: Summary of our results for the BAR-B-tree on a set of N points in $\mathbb{R}^d$ for approximate range searching queries (apxRS), approximate range aggregate queries (apxRA) and approximate nearest-neighbor queries (apxNN). For approximate range searching queries and approximate range aggregate queries, $\gamma = 1 - d$ if the range is convex, and $-d$ otherwise. The update bound is amortized. The dimensionality d is assumed to be a constant.

which is not necessarily the case. Some leaves may contain a small number of points and the query cost is in fact $O(\log_B N + \varepsilon^\gamma + k_\varepsilon)$ in the worst case. For an approximate range searching query that has a potentially large output, it is crucial to have an output term of $O(k_\varepsilon/B)$ rather than $O(k_\varepsilon)$. As typical values of B are on the order of hundreds to thousands, the difference between $O(k_\varepsilon/B)$ disk accesses and $O(k_\varepsilon)$ disk accesses can be significant.

Agarwal et al. [1] gave a general framework for externalizing and dynamizing *weight-balanced partitioning trees* such as the BAR-tree. They describe how a weight-balanced partition tree can be constructed in the optimal $O(\text{sort}(N))$ I/O-operations. Like Duncan [37], they use a breadth-first blocking scheme for storing the resulting tree on disk. To remove the assumption made by Duncan they group blocks together which contain too few nodes. As a result there is at most one block containing too few nodes. This improvement ensures that the resulting layout only uses $O(N/B)$ disk blocks, but the approximate range searching cost is still $O(\log_B N + \varepsilon^\gamma + k_\varepsilon)$, since there can be $\Theta(B)$ blocks each containing one (or a few) subtrees of constant size whose stored points have to be reported. In the worst case $\Theta(k_\varepsilon)$ blocks have to be loaded, each containing $O(1)$ points inside the query.

Our results. Two main results are obtained in this chapter. We first give a new blocking scheme for the BAR-tree that yields the first disk-based index structure, the BAR-B-*tree*, which answers all of the aforementioned approximate queries efficiently. In particular, the BAR-B-tree answers an approximate range searching query in the desired $O(\log_B N + \varepsilon^\gamma + k_\varepsilon/B)$ I/O-operations, i.e., achieving an $O(\log_B N)$ search term and an $O(k_\varepsilon/B)$ output term simultaneously. Such terms are optimal when disk-based indexes are concerned. Unfortunately it seems difficult to reduce the error term $O(\varepsilon^\gamma)$. The bounds for the costs of other queries and operations match the previous results on externalizing BAR-trees [37, 1], in

	oBAR-B-tree
Size	$O(\lambda N/B)$
bulk-load	$O(\text{sort}(\lambda N))$
apxRS	$O(\log_B N + \lceil \lambda/B \rceil \varepsilon^\gamma + \lambda k_\varepsilon/B)$
apxRA	$O(\log_B N + \lceil \lambda/B \rceil \varepsilon^\gamma)$
apxNN	$O(\log_B N + \lceil \lambda/B \rceil \varepsilon^{1-d}(1 + \frac{1}{B} \log_{M/B}(1/\varepsilon)))$

Table 5.2: Summary of our results for the oBAR-B-tree on a set of N objects in $\mathbb{R}^d$ for approximate range searching queries (apxRS), approximate range aggregate queries (apxRA) and approximate nearest-neighbor queries (apxNN). For approximate range searching queries and approximate range aggregate queries, $\gamma = 1 - d$ if the range is convex, and $-d$ otherwise. The cost of approximate range aggregate queries holds only for duplicate-insensitive aggregates. The dimensionality d is assumed to be a constant.

particular we can also bulk-load and update the BAR-B-tree efficiently. Please see Table 5.1 for the detailed results.

Next, we generalize the BAR-B-tree to the *oBAR-B-tree*, which indexes not just points, but arbitrary spatial objects of constant complexity. The approximate range searching queries, approximate range aggregate queries, and approximate nearest-neighbor queries are generalized to objects as follows. Let S be a set of objects in $\mathbb{R}^d$ and Q the query range. For an approximate range searching query and an approximate range aggregate query, we return a subset $S^* \subseteq S$ (or the aggregate $\bigoplus_{o \in S^*} \omega(o)$) where S^* includes all objects in S that intersect Q, does not include any object that does not intersect Q_ε, and may optionally include some objects that intersect Q_ε but not Q. For an approximate nearest-neighbor query, the definition remains the same with the distance definition between an object o and the query point q being $d(o, q) = \min_{p \in o} d(p, q)$. Our idea is based on range searching data structures for *low-density scenes* [24]. The *density* of 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 $\rho(o) \geqslant \rho(b)$ where $\rho(o)$ denotes the radius of the smallest enclosing ball of o, see Section 1.2.2. It is believed that for many realistic inputs, λ is small. For instance, if all objects of S are disjoint and *fat*, then λ is a constant. The oBAR-B-tree exhibits the same performance bounds as the BAR-B-tree if λ is a constant, and the costs grow roughly linearly with λ for most operations. Unfortunately, we do not have a bound on the update cost for the oBAR-B-tree, but we expect the actual cost to be small in practice. Please refer to Table 5.2 for the detailed costs of various operations. Another nice feature of the oBAR-B-tree is that it will automatically reduce to the BAR-B-tree if all the objects are points ($\lambda = 1$ for a set of points).

To summarize, with the BAR-B-tree and the oBAR-B-tree, we present a disk-based spatial indexes that is as versatile as the R-tree and in addition provide provable guarantees on various operations. In particular, the query costs on range

searching are, except for the $O(\varepsilon^\gamma)$ term, similar to that of the B-tree answering one-dimensional range queries on points. In addition, these two spatial indexes are not difficult to implement, and we expect them to be fairly practical as well. It would be interesting to compare them with R-trees to see how well they behave in practice.

5.1 An I/O-efficient BAR-tree on points

In this section we describe the BAR-B-tree, an efficient layout for the BAR-tree on disk that achieves all the desired bounds listed in Table 5.1. We introduce our two-stage blocking scheme in Section 5.1.1, and analyze its query cost when answering an approximate range searching query in Section 5.1.2. The query costs for an approximate range aggregate query and an approximate nearest-neighbor query follow from Agarwal et al. [1] and Duncan [37]. Finally we give an efficient bulk-loading algorithm in Section 5.1.3, and talk about updates. For the remainder of this chapter we assume that $\mathcal{T}$ has at least $B/2$ nodes, otherwise the solution is trivial.

5.1.1 The blocking scheme

For any node $\mu \in \mathcal{T}$, let $\mathcal{T}_\mu$ be the subtree rooted at μ, and we define $|\mathcal{T}_\mu|$, the size of $\mathcal{T}_\mu$, to be the number of nodes in $\mathcal{T}_\mu$ (including μ). At every node we store the size of both its subtrees. Our blocking scheme consists of two stages. In the first stage the tree is blocked such that for any $\mu \in \mathcal{T}$, $\mathcal{T}_\mu$ is stored in $O(\lceil |\mathcal{T}_\mu|/B \rceil)$ blocks. As we will see, this property will guarantee the $O(k_\epsilon/B)$ term in the bound on the query cost. However, a root-to-leaf path in $\mathcal{T}$ may be covered by $\Theta(\log N)$ such blocks. In the second stage we make sure that any root-to-leaf path can be traversed by accessing $O(\log_B N)$ blocks.

Tree-blocks. In the first stage we block the tree $\mathcal{T}$ into *tree-blocks* such that $\mathcal{T}$ is stored in $O(\lceil |\mathcal{T}|/B \rceil)$ blocks. The blocking procedure is detailed in Algorithm 1. We traverse the tree $\mathcal{T}$ in a top-down fashion, and keep in a set $\mathcal{S}$ all nodes μ for which a block will be allocated such that μ is the topmost node in the block. Initially $\mathcal{S}$ only contains the root of $\mathcal{T}$. For any node $\mu \in \mathcal{S}$, we find a connected subtree rooted at μ to fit in one block using an adapted breadth-first strategy with a queue $\mathcal{Q}$. Throughout the blocking algorithm we maintain the invariant that $|\mathcal{T}_\mu| \geq B/2$ for any μ that is ever added to $\mathcal{S}$ or $\mathcal{Q}$. The invariant is certainly true when the algorithm initializes (line 1).

For a node $\mu \in \mathcal{S}$, we fill a block with a top portion of $\mathcal{T}_\mu$ by an adapted breadth-first search (BFS) (line 4–23). The BFS starts with $\mathcal{Q} = \{\mu\}$ (line 4), which is consistent with the invariant since μ is a node from $\mathcal{S}$. For each node ν encountered in the BFS search, we distinguish among the following three cases. (a) If $|\mathcal{T}_\nu| \leq B$, then we allocate a new block to store the entire $\mathcal{T}_\nu$ (line 7–9). Note that this block contains at least $B/2$ nodes by the invariant. (b) Let ν_1, ν_2 be

---
Algorithm 1: Algorithm to construct tree-blocks

Input: a binary tree $\mathcal{T}$
Output: a set of tree-blocks stored on disk

1 initialize $\mathcal{S} := \{$root of $\mathcal{T}\}$, and a block $\mathcal{B} := \emptyset$;
2 **while** $\mathcal{S} \neq \emptyset$ **do**
3   remove any node μ from $\mathcal{S}$;
4   initialize a queue $\mathcal{Q} := \{\mu\}$;
5   **while** $\mathcal{Q} \neq \emptyset$ **do**
6   remove the first node ν from $\mathcal{Q}$, let ν_1, ν_2 be ν's children;
7   **if** $|\mathcal{T}_\nu| \leq B$ **then**
8   put $\mathcal{T}_\nu$ in a new block $\mathcal{B}'$;
9   write $\mathcal{B}'$ to disk;
10   **else if** $|T_{\nu_1}| \geq B/2$ and $|T_{\nu_2}| \geq B/2$ **then**
11   add ν to $\mathcal{B}$;
12   add ν_1, ν_2 to $\mathcal{Q}$;
13   **else**
14   suppose $|\mathcal{T}_{\nu_1}| < B/2$;
15   **if** $|\mathcal{B}| + |\mathcal{T}_{\nu_1}| + 1 \leq B$ **then**
16   add ν and $\mathcal{T}_{\nu_1}$ to $\mathcal{B}$;
17   add ν_2 to $\mathcal{Q}$;
18   **else**
19   add ν to $\mathcal{S}$;
20   **if** $|\mathcal{B}| = B$ **then**
21   write $\mathcal{B}$ to disk and reset $\mathcal{B} := \emptyset$;
22   add all nodes of $\mathcal{Q}$ to $\mathcal{S}$;
23   set $\mathcal{Q} := \emptyset$;
24   **if** $|\mathcal{B}| \neq \emptyset$ **then**
25   write $\mathcal{B}$ to disk and reset $\mathcal{B} := \emptyset$;
---

the two children of ν. If both $\mathcal{T}_{\nu_1}$ and $\mathcal{T}_{\nu_2}$ have more than $B/2$ nodes, then we add ν to the block and continue the BFS process (line 10–12). It is safe to add ν_1, ν_2 to $\mathcal{Q}$ as we have ensured the invariant. (c) Otherwise, it must be the case that one of the subtrees is smaller than $B/2$ nodes while the other one has more than $B/2$ nodes. Without loss of generality we assume $|\mathcal{T}_{\nu_1}| < B/2$, and then check if $\mathcal{T}_{\nu_1}$ plus ν itself still fit in the current block. If so we add ν and the entire $\mathcal{T}_{\nu_1}$ to the current block, add ν_2 to $\mathcal{Q}$ (notice that $|\mathcal{T}_{\nu_2}| > B/2$) and continue the BFS; else we put ν into $\mathcal{S}$. Note that when a node is extracted from $\mathcal{S}$, the current block $\mathcal{B}$ is always empty and ν can thus always be stored in it; in particular line 19 is never executed for ν. Hence each node is inserted in $\mathcal{S}$ and extracted from $\mathcal{S}$ at most once. Please refer to Figure 5.1 for an illustration of this blocking algorithm.
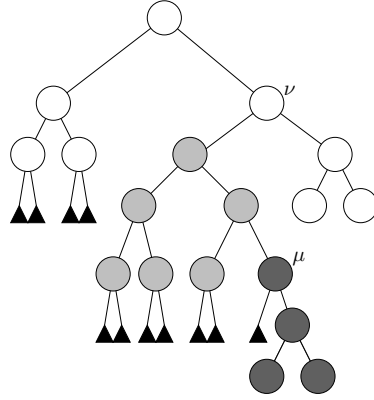
Figure 5.1: Three tree-blocks (white, light gray and dark gray) obtained using the blocking scheme for $B = 8$. The black triangles denote the existence of a subtree of size at least $B/2$. The right subtree of ν is placed completely in the white block. The node μ and its right subtree do not fit in the light gray block so a new block must be started at μ. Note that depending on the subtrees, the light and dark gray block could store some nodes of the subtrees.

Lemma 5.1 *For any $\mu \in \mathcal{T}$, the nodes in $\mathcal{T}_\mu$ are stored in $O(\lceil |\mathcal{T}_\mu|/B \rceil)$ blocks.*

Proof: First consider the case where μ is the topmost node in some block, i.e., μ has been added into $\mathcal{S}$. Suppose a tree-block $\mathcal{B}$ contains less than $B/2$ nodes of the tree $\mathcal{T}_\mu$. There is at least one block below $\mathcal{B}$ since by the invariant the subtree of $\mathcal{T}$ rooted at the topmost node in $\mathcal{B}$ has size at least $B/2$. We claim that at least one block $\mathcal{B}'$ directly below $\mathcal{B}$ contains at least $B/2$ nodes. Now suppose for a contradiction that no block directly below $\mathcal{B}$ contains more than $B/2$ nodes. Let $\mathcal{B}'$ be a block below $\mathcal{B}$ containing less than $B/2$ nodes and let ν be the highest node in $\mathcal{B}'$. The node ν was not placed in $\mathcal{B}$ either because the size of $\mathcal{T}_\nu$ was between B and $B/2$ (case (a)) or because the size of one of its subtrees, say $\mathcal{T}_{\nu_1}$, was less than $B/2$ (case (c)). The former case immediately leads to a contradiction. The latter case also leads to a contradiction since $\mathcal{T}_{\nu_1}$, together with ν, fits in $\mathcal{B}$ and would then have been placed in $\mathcal{B}$. So there must be a block $\mathcal{B}'$ below $\mathcal{B}$ whose size is at least $B/2$. We charge $\mathcal{B}$ to $\mathcal{B}'$. Each block containing at least $B/2$ nodes is charged at most once, namely by the block directly above it. The number of tree-blocks is thus $O(\lceil |\mathcal{T}_\mu|/B \rceil)$.

Next consider the case where $\mu \in \mathcal{B}$ but μ is not the topmost node in $\mathcal{B}$. Let $\mu_1, \ldots, \mu_t$ be the nodes of $\mathcal{T}_\mu$ stored immediately below $\mathcal{B}$. By the blocking algorithm's invariant, we have $|\mathcal{T}_{\mu_i}| \geq B/2$, so $|\mathcal{T}_\mu| > t \cdot B/2$, or $t = O(|\mathcal{T}_\mu|/B)$. Applying the case above, the number of blocks used to store the tree $\mathcal{T}_\mu$ is thus $1 + O(\sum_{i=1}^{t}(\lceil |\mathcal{T}_{\mu_i}|/B \rceil)) = O(\lceil |\mathcal{T}_\mu|/B \rceil + t) = O(\lceil |\mathcal{T}_\mu|/B \rceil)$. $\qquad\square$
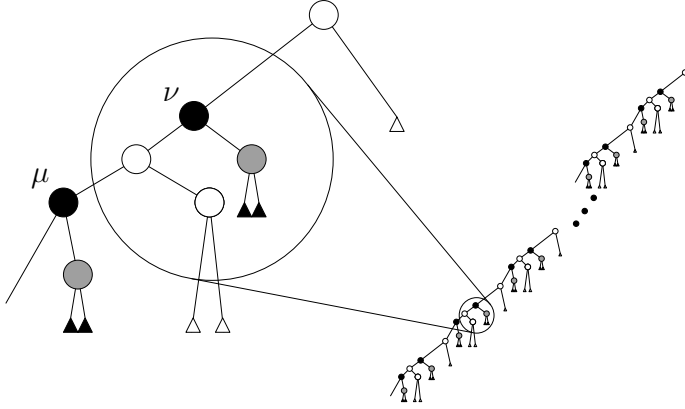
Figure 5.2: A bad example for tree-blocks. The block nodes are nodes of which one child, depicted in gray, has size $B/2 - 1$. The black triangles represent the subtrees of a gray node. All white triangles represent subtrees of sufficiently large size. The leftmost path L will be split into $\Theta(\log N)$ blocks, since ν and μ cannot be placed in one block.

Remark 5.2 It might be conceptually simpler to use a bottom-up approach to block the tree $\mathcal{T}$ such that Lemma 5.1 also holds. However, our top-down approach is in fact easier to implement, especially when used in the top-down bulk-loading algorithms for BAR-trees. Please see Section 5.1.3 for details.

The blocked BAR-tree resulting after the first stage might have depth as bad as $\Theta(\log N)$, as illustrated by the following example. Consider a root-to-leaf path L in a BAR-tree $\mathcal{T}$ of length $\Theta(\log N)$. We say a node ν is balanced if both subtrees below ν have size at most $\beta|\mathcal{T}_\nu|$ for some fixed $0 < \beta < 1$. In a BAR-tree every other node on L may be unbalanced. In particular each unbalanced node might have one subtree, say $\mathcal{T}_{\nu_1}$, of size $B/2 - 1$ (see Figure 5.2). Consider an unbalanced node ν. The algorithm will always store every unbalanced node ν and its subtree $\mathcal{T}_\nu$ of size $B/2-1$ in one block. If two such nodes ν and μ on L would be stored together in one block $\mathcal{B}$, all nodes on the path from ν to μ would be stored in $\mathcal{B}$ as well. This path contains at least one node other than ν and μ. The total number of nodes in $\mathcal{B}$ would thus be at least $2(1 + (B/2 - 1)) + 1 = B + 1$. This cannot happen, since the algorithm would never put $B + 1$ nodes in one block. So every unbalanced node on L is stored in a different block.

In the second stage we introduce *path-blocks* which ensure that $O(\log_B N)$ blocks have to be accessed in order to visit all nodes on any root-to-leaf path.

Path-blocks.   To identify the places where a path-block has to be introduced we visit $\mathcal{T}$, in a top-down fashion. The algorithm for constructing path-blocks is given in Algorithm 2. We keep in a set $\mathcal{S}$ all nodes μ for which we consider

---

Algorithm 2: Algorithm to construct path-blocks

Input: a binary tree $\mathcal{T}$ stored in tree blocks
Output: a set of tree and path-blocks stored on disk

1 initialize a stack $\mathcal{S} := \{\text{root of } \mathcal{T}\}$;
2 **while** $\mathcal{S} \neq \emptyset$ **do**
3   remove any node μ from $\mathcal{S}$;
4   initialize a queue $\mathcal{Q} := \{\mu\}$;
5       $\widehat{\mathcal{T}}_\mu = \emptyset$;
6   (* Get $\log B$ levels of $\mathcal{T}$ starting at μ *)
7   **while** *height* $\widehat{\mathcal{T}}_\mu \leq \log B$ **do**
8   remove ν from $\mathcal{Q}$, let ν_1, ν_2 be ν's children;
9   **if** *Adding ν to $\widehat{\mathcal{T}}_\mu$ does not increase its depth* **then**
10   add ν to $\widehat{\mathcal{T}}_\mu$;
11   add ν_1, ν_2 to $\mathcal{Q}$;
12   **else**
13   add ν to $\mathcal{Q}$;
14   (* Check if $\widehat{\mathcal{T}}_\mu$ should be stored in a path-block *)
15   **if** *Any path of the tree in $\widehat{\mathcal{T}}_\mu$ crosses more than c blocks* **then**
16   remove nodes in $\widehat{\mathcal{T}}_\mu$ from the tree-blocks storing them;
17   store $\widehat{\mathcal{T}}_\mu$ in a new block $\mathcal{B}$ and write $\mathcal{B}$ to disk;
18   (* Consider only subtrees of size at least B *)
19   **while** $\mathcal{Q} \neq \emptyset$ **do**
20   remove a node ν from $\mathcal{Q}$;
21   **if** $|\mathcal{T}_\nu| > B$ **then**
22   add ν to $\mathcal{S}$;

---

the tree $\widehat{\mathcal{T}}_\mu$ consisting of the first $\log B$ levels of $\mathcal{T}$ encountered by a BFS starting at μ. We check all root-to-leaf paths in $\widehat{\mathcal{T}}_\mu$. If there is at least one such path that is divided among more than c tree-blocks for some integer constant $c \geq 2$, then we introduce a path-block that stores $\widehat{\mathcal{T}}_\mu$. We also remove all nodes of $\widehat{\mathcal{T}}_\mu$ from the tree-blocks where they are stored. The queue $\mathcal{Q}$ used in the BFS is then emptied by moving a node ν in $\mathcal{Q}$ to $\mathcal{S}$ if it is the root of a subtree $\mathcal{T}_\nu$ whose size is larger than B. We continue until $\mathcal{S}$ is empty. This completes our two-stage blocking scheme. With the introduction of path-blocks, now we have the following.

Lemma 5.3 *Any root-to-leaf path in $\mathcal{T}$ can be traversed by accessing $O(\log_B N)$ blocks.*

Proof: Let L be a root-to-leaf path in $\mathcal{T}$. We divide L in two parts. The first part consists of all nodes which are the root of a subtree of size at least B. The second

part consists of all other nodes. All nodes in the first part of L are placed in some $\widehat{\mathcal{T}}_\mu$ during the construction of the path-blocks. Any root-to-leaf path in any $\widehat{\mathcal{T}}_\mu$, if it does not reach a leaf of $\mathcal{T}$, is at least $\log B$ long. Therefore, we can traverse the first part of L by accessing $O(\log_B N)$ blocks. For the second part of L we argue that the top-most node in this part is the root of a subtree of size at most B. By Lemma 5.1 this tree, and therefore the second part of L, is stored in $O(1)$ blocks. Therefore we need to access $O(\log_B N)$ blocks in total when traversing L. $\quad\square$

Since any path-block has at least $B/2$ nodes, it is easy to see that Lemma 5.1 still holds. In particular, we obtain the desired space bound for the BAR-B-tree.

Theorem 5.4 *A* BAR-B-*tree on N points in $\mathbb{R}^d$ takes $O(N/B)$ disk blocks.*

Since our blocking scheme has no redundancy, i.e., each node of $\mathcal{T}$ is stored in only one block, after the two-stage blocking process we can group blocks together such that all of them are at least half-full. So the space utilization of the BAR-B-tree can be at least 50%.

5.1.2 Analysis of the range searching cost

Since no node is stored in multiple blocks we can use the standard query algorithm for BSPs, that is, we start from the root, and visit all nodes μ of $\mathcal{T}$ where the region associated with μ intersects with the query range Q. The traversal can be performed either breadth-first or depth-first, with the use of an I/O-efficient stack or queue such that the extra overhead is $O(1)$ I/O-operations per B nodes. So we only need to bound the number of blocks that store all the visited nodes.

Theorem 5.5 *An approximate range searching query Q in a* BAR-B-*tree can be answered by accessing $O(\log_B N + \varepsilon^\gamma + k_\varepsilon/B)$ blocks.*

Proof: Note that the region of any visited node must intersect Q. Next, we make a distinction between (a) nodes whose regions also intersect the boundary of Q_ε and (b) those whose regions are completely contained in Q_ε.

We start by proving a bound on the number of blocks containing the type-(a) nodes. Duncan [37] proved that the number of such nodes is $O(\log N + \varepsilon^\gamma)$. The $O(\log N)$ term comes from a constant number of root-to-leaf paths in $\mathcal{T}$. By Lemma 5.3 these nodes are covered by $O(\log_B N)$ blocks. So in total we need to access $O(\log_B N + \varepsilon^\gamma)$ blocks for nodes of type (a).

Next we give a bound on the number of blocks that cover all the type-(b) nodes. These nodes are organized in t disjoint subtrees $\mathcal{T}_{\mu_1}, \ldots, \mathcal{T}_{\mu_t}$, such that $\mathrm{R}(\mu_i) \subseteq Q_\varepsilon$ and $\mathrm{R}(p(\mu_i)) \not\subseteq Q_\varepsilon$, where $p(\mu_i)$ denotes the parent of μ_i and $\mathrm{R}(\nu)$ denotes the region associated with the node ν. We will in fact bound the number of blocks covering all these t subtrees. Note that since $\mathrm{R}(\mu_i)$ is contained in Q_ε, $\mathrm{R}(p(\mu_i))$ must intersect the boundary of Q_ε, i.e., a type-(a) node. Each parent $p(\mu_i)$ has only one child whose region is inside Q_ε, since otherwise $\mathrm{R}(p(\mu_i))$ would be completely inside Q_ε. From Duncan et al. [37, 38] we know that

there are in total $O(\varepsilon^\gamma)$ type-(a) nodes who have a child associated with a region completely inside Q_ε, hence $t = O(\varepsilon^\gamma)$.

Note that the subtree $\mathcal{T}_{\mu_i}$ stores at least $|\mathcal{T}_{\mu_i}|/2$ points of P inside Q_ε. By Lemma 5.1 $\mathcal{T}_{\mu_i}$ is stored in $O(\lceil|\mathcal{T}_{\mu_i}|/B\rceil)$ blocks. Thus the total number of blocks covering all the t subtrees is

$$
\begin{aligned}
O\left(\sum_{i=1}^{t}\lceil|\mathcal{T}_{\mu_i}|/B\rceil\right) &= O\left(t + \sum_{i=1}^{t}|\mathcal{T}_{\mu_i}|/B\right)\\
&= O(\varepsilon^\gamma + k_\varepsilon/B).
\end{aligned}
$$

$\square$

5.1.3 An efficient construction algorithm

We are left with giving an I/O-efficient bulk-loading algorithm to build a BAR-B-tree with a set of N points in $\mathbb{R}^d$. We use the "grid" method introduced by Agarwal et al. [1], which we briefly review here. The grid method can be used to construct a class of space partitioning structures I/O-efficiently, including the BAR-tree. The idea is to first build a grid G of size $\Theta((M/B)^c)$ in memory for some constant $0 < c \le 1/2$, as in the c-grid BSP of Section 2.2. This grid G is then used to build the top $\Theta(\log(M/B))$ levels of the BSP. We know for each cell of G how many points are contained within them. This knowledge is used in the placement of the splitting hyperplane h of some node in the BSP. For the recursive construction we only need to scan that part of the data set which is contained in the cells of G which were intersected by h.

Next the point set is partitioned into subsets that correspond to the subtrees below the $\Theta(\log(M/B))$ levels. This process can be completed with a constant number of scans of the data set. Finally we recurse to build the subtrees. The recursion stops when we have less than M points to deal with, for which we just build the entire subtree in memory. The overall cost is then $O(N/B \cdot \log(N/B)/\log(M/B)) = O(\text{sort}(N))$ I/O-operations.

Observing that each recursive call to the grid method must still have at least B points to deal with, since $M/(M/B)^c \ge B$, our top-down blocking scheme for the tree-blocks (Algorithm 1) is very easy to couple with the also top-down grid method. Consider one round in the construction of $\mathcal{T}$ using the grid method. Let $\widehat{\mathcal{T}}$ be the $\Theta(\log(M/B))$ levels of $\mathcal{T}$ constructed in that round. The grid method gives us all the sizes of the subtrees which are going to be built recursively, see [1], so we can run Algorithm 1 while constructing $\widehat{\mathcal{T}}$. However, we have to be careful at the bottom of $\widehat{\mathcal{T}}$. If we need to create a new block $\mathcal{B}$ for some node ν and there are not enough nodes to fill $\mathcal{B}$ within the subtree $\mathcal{T}_\nu$ rooted at ν within $\widehat{\mathcal{T}}$, we do not add ν to $\widehat{\mathcal{T}}$. At a later stage we start a recursive call of the grid method with ν. Note that with this adaptation of the grid method $\widehat{\mathcal{T}}$ still has $\Theta(\log(M/B))$ levels of $\mathcal{T}$ since we construct $O(\log B)$ levels less by the early termination of

the construction of $\widehat{\mathcal{T}}$. This modification to the grid method has no influence on the asymptotic bound of the construction cost.

After we have constructed all the tree-blocks, we build the path-blocks as described above in Section 5.1.1. We first note that the queue $\mathcal{Q}$ in Algorithm 2 stores a reference to a node ν and not ν itself, because ν could be stored in a block which is not in memory. To obtain the size of the subtree rooted at ν without using any I/O-operations in lines 16–19 of Algorithm 2 we also store the size of the subtree in $\mathcal{Q}$. During the construction of the tree-blocks we can store at every node the sizes of its subtrees without any additional I/O-operations.

To build one path-block, a top subtree of some $\mathcal{T}_\mu$ with $O(\log B)$ levels needs to be read, and this subtree may spread across several tree-blocks. To bound the number of read-operations on all tree-blocks we argue as follows. Let V be the set of nodes in $\mathcal{T}$ which are the roots of the $\mathcal{T}_\mu$'s considered, i.e. the nodes which were at one point placed in the stack $\mathcal{S}$ of Algorithm 1. Any tree-block $\mathcal{B}$ is read at most $1 + t_\mathcal{B}$ times where $t_\mathcal{B}$ is the number of nodes of V contained in $\mathcal{B}$, i.e., once for the tree $\widehat{\mathcal{T}}$ containing the top-most node of $\mathcal{B}$ and once for each of the $t_\mathcal{B}$ nodes. The total number of read-operations is thus $\sum(1 + t_\mathcal{B}) = O(N/B + |V|)$ where the sum is over all blocks. By Algorithm 1 the nodes stored in $\mathcal{S}$ are the root of a subtree of size at least $B/2$, so $|V| = O(N/B)$. Since there are $O(N/B)$ path-blocks and every tree-block is written at most as many times as it was read, it takes $O(N/B)$ I/O-operations to build all the path-blocks. This completes the analysis of our bulk-loading algorithm.

Theorem 5.6 *It takes $O(\mathrm{sort}(N))$ I/O-operations to bulk-load a* BAR-B-*tree on a set of N points in $\mathbb{R}^d$.*

5.1.4 Updating the BAR-B-tree.

Since we have an efficient bulk-loading algorithm, we can use the *partial rebuilding* technique [69, 1]. In the next paragraph we describe how to handle insertions. Deletions can be handled similarly. A node ν in a BAR-tree is *out of balance* when the size of a subtree rooted at a grandchild of ν is more than β times the size of the subtree rooted at ν for some constant $0.5 \le \beta < 1$.

BAR-B-tree update algorithm.   To insert a point into the BAR-B-tree $\mathcal{T}$, we first follow a root-to-leaf path to find the leaf block where the point should be located. According to Lemma 5.3 this takes $O(\log_B N)$ I/O-operations. After inserting the point we check the nodes on this path to see if any of them is out of balance. Among all the nodes which are out of balance, we rebuild the whole subtree $\mathcal{T}_\nu$ rooted at the highest such node ν in block $\mathcal{B}$. Let $\mathcal{T}_\nu'$ be the BAR-B-tree built on the points in $\mathcal{T}_\nu$. Algorithm 1 has to be adapted slightly for the construction of the tree-blocks of $\mathcal{T}_\nu'$. The construction differs from Algorithm 1 in that in line 1 a new block is not initialized, but $\mathcal{B}$, from which ν and its descendants are removed, is used instead. In the second step, the construction of the path-blocks, we only consider tree-blocks which store nodes of $\mathcal{T}_\nu'$. Using

standard analysis, it can be shown that the amortized cost of these two steps is $O(\frac{1}{B}\log_{M/B}(N/B)\log(N/B))$ I/O-operations. Unless $\mathcal{T}'_\nu$ fits completely in $\mathcal{B}$, we introduce a new block $\mathcal{B}'$, which we call a *semi-path-block*. We start at the highest node in $\mathcal{B}$ and visit, as in a BFS, the first $\log B$ levels of the subtree consisting of the nodes in $\mathcal{B}$ and $\mathcal{T}'_\nu$. We move the visited nodes to $\mathcal{B}'$. The construction of a semi-path-block takes $O(1)$ additional I/O-operations. In total the amortized cost of an update is $O(\log_B N + \frac{1}{B}\log_{M/B}(N/B)\log(N/B))$.

Next we will show that the bounds on the number of blocks which stores the tree and the number of blocks which need to be accessed for the traversal of any root-to-leaf path still hold. For this discussion we assume that $\mathcal{T}'_\nu$ does not fit in $\mathcal{B}$, since otherwise there is no change in the number of blocks.

Number of blocks in which the BAR-B-tree is stored. The blocking scheme ensures that $\mathcal{T}'_\nu$ is stored in $O(|\mathcal{T}'_\nu|/B)$ blocks. The block containing ν might have less than $B/2$ nodes after the partial rebuild while it had more than $B/2$ before the rebuild, but we can charge $\mathcal{B}$ to one of the blocks below $\mathcal{B}$ storing nodes of $\mathcal{T}_\nu$. This block $\mathcal{B}_s$ contains at least $B/2$ nodes using a similar argument as in Lemma 5.1. The semi-path-block which was introduced next might increase the total number of blocks by one; we charge this extra block to $\mathcal{B}_s$. Any block containing at least $B/2$ nodes of $\mathcal{T}'_\nu$ is charged at most twice. The updated BAR-B-tree is thus stored in $O(N/B)$ blocks.

Number of blocks which need to be accessed for the traversal of any root-to-leaf path. To support updates we weaken the restriction that for any root-to-leaf path L in a BAR-B-tree one has to access at most c blocks for every $\log B$ consecutive nodes of L. We sometimes allow $c+1$ blocks to be accessed, but only if the following invariant holds.

Invariant 5.7 Let L be any root-to-leaf path in a BAR-B-tree $\mathcal{T}$ and let $\mathcal{B}_1 \dots \mathcal{B}_p$ be the blocks which need to be accessed for the traversal of L. If there is an $i > 1$ such that $\mathcal{B}_i \dots \mathcal{B}_{i+c}$ store less than $\log B$ nodes of L then $\mathcal{B}_{i-1}$ is a semi-path-block.

The initial BAR-B-tree satisfies Invariant 5.7 since at most c blocks have to be accessed for traversing $\log B$ nodes of any root-to-leaf path. Let $\mathcal{B}$ be the block storing the highest node ν which is out of balance after an update. Let $\mathcal{B}'$ be the semi-path-block constructed during the update and let L' be any path within $\mathcal{B}$. Note that $\mathcal{B}'$ contains the same nodes of L' if L' has length at most $\log B$ and does not contain ν. Also note that if L' is larger than $\log B$ and does not contain ν then L' is stored in two blocks.

Suppose that after an update Invariant 5.7 does not hold and let L be a root-to-leaf path which invalidates the invariant. The block $\mathcal{B}'$ can never be part of that part of L which invalidates the invariant, because $\mathcal{B}'$ either contains as many nodes of L as $\mathcal{B}$ did or at least $\log B$ nodes. In both cases the invariant holds.

The only place were the invariant can be invalidated is directly below $\mathcal{B}'$. Since $\mathcal{B}'$ is a semi-path-block the invariant can only be invalidated because one has to access $c + 2$ blocks in order to traverse $\log B$ nodes below $\mathcal{B}'$. This however is impossible; if L contains nodes from $\mathcal{T}_\nu'$ then one has to access at most $c + 1$ blocks by construction and otherwise $\mathcal{B}'$ contains the same nodes as $\mathcal{B}$ because all nodes in $\mathcal{B}$ of L are moved to $\mathcal{B}'$. So at most $c + 1$ blocks need to be accessed. After an update the invariant thus must hold. From the invariant it follows that any root-to-leaf path can be traversed by accessing $O(\log_B N)$ blocks.

5.2 Storing objects: the oBAR-B-tree

In this section we show how to externalize the object BAR-tree (oBAR-tree), see Chapter 4, for a set S of objects of constant complexity with density λ. We first review the oBAR-tree briefly below.

The oBAR-tree is based on the idea of *guarding sets* [20, 66]. For a subset $X \subseteq S$, we call a set of points G_X a *guarding set* of X if the region associated with any leaf in the BAR-tree constructed on G_X intersects at most $O(\lambda)$ objects of X. To build the oBAR-tree on S, we first compute for each object $o \in S$ a constant number of points, called the *guards* of o, with the property that the guards of any subset X of S form a guarding set for X. Let G be the set of all guards. We build the oBAR-tree by first constructing a BAR-tree $\mathcal{T}$ on G, with the adaptation that whenever we are going to build a subtree for a region R with a subset $G' \subset G$, we delete all guards from G' whose objects do not intersect R. Then we store at each leaf of $\mathcal{T}$ all objects of S that intersect the region associated with the leaf. We showed in Chapter 4 that each leaf stores $O(\lambda)$ objects, and an approximate range searching query can be answered in time $O(\log N + \lambda \cdot \{\varepsilon^\gamma + k_\varepsilon\})$.

5.2.1 Building the oBAR-B-tree

We first compute all the guards with one scan over S as explained in Section 4.1. Next we build the BAR-B-tree on the set of all guards G using our bulk-loading algorithm of Section 5.1.3. The adaptation of removing guards during the construction as described above can be easily accommodated in the algorithm, and we can build and lay out the tree $\mathcal{T}$ on disk in $O(\text{sort}(\lambda N))$ I/O-operations. During the process we can also compute for each leaf ν of $\mathcal{T}$, the set of at most $O(\lambda)$ objects that intersect the region $\mathrm{R}(\nu)$. We omit the technical details.

Finally, for each leaf block $\mathcal{B}$ of $\mathcal{T}$, we store all the intersecting objects consecutively on disk. More precisely, consider a block $\mathcal{B}$ and let L be the set of leaves stored in $\mathcal{B}$. The objects intersecting the regions of the nodes in L are stored together in one list as follows. Let $\nu_1, \cdots, \nu_{|L|}$ be the leaves in L ordered according to an in-order traversal of $\mathcal{T}$. We first store the objects intersecting $\mathrm{R}(\nu_1)$, then the objects intersecting $\mathrm{R}(\nu_2)$, and so on. Note that an object might be stored more than once in the list. At every leaf ν_i we store a pointer to the first and last object in the list intersecting $\mathrm{R}(\nu_i)$. Since each leaf has $O(\lambda)$ intersecting objects,

each such list occupies $O(\lambda B/B) = O(\lambda)$ disk blocks, so the overall space usage of these lists is $O(\lambda N/B)$ blocks. This completes the description of the oBAR-B-tree. Note that the oBAR-B-tree automatically reduces to the BAR-B-tree when all the objects are points.

Theorem 5.8 *Let S be a set of N objects in $\mathbb{R}^d$ with density λ. An oBAR-B-tree on S takes $O(\lambda N/B)$ blocks and any root-to-leaf path can be traversed by accessing $O(\log_B N)$ blocks. It can be constructed using $O(\operatorname{sort}(\lambda N))$ I/O-operations.*

5.2.2   Analysis of the range searching cost

In this section we prove the bounds stated in Table 5.2. The cost of approximate range aggregate queries and approximate nearest neighbor queries can be analyzed in the same way as for the BAR-B-tree, leading to the same number of visited leaves; the only difference is that for every visited leaf ν, we now have to check the $\lceil \lambda/B \rceil$ blocks containing the objects intersecting $R(\nu)$. However, note that since in an oBAR-B-tree an object might be stored at several leaves, we can only handle duplicate-insensitive aggregates.

For approximate range searching queries we have to be careful not to report an object more than once. We adapt the query algorithm as follows. For every object in a leaf ν intersecting the approximate query range Q_ε we do the following. We compute a unique reference point p in the intersection of the object o and Q_ε, for instance in $\mathbb{R}^2$ the upper left most point of o contained in Q_ε. We report o only if p is contained in $R(\nu)$. We now bound the cost for an approximate range searching query.

Theorem 5.9 *Let S be a set of N objects of constant complexity in $\mathbb{R}^d$ with density λ. An oBAR-B-tree for S answers an approximate range searching query Q using $O(\log_B N + \lceil \lambda/B \rceil \varepsilon^\gamma + \lambda k_\varepsilon/B)$ I/O-operations, where k_ε is the number of objects intersecting the extended query range Q_ε.*

Proof: The query cost of answering a range searching query Q consists of two parts: the cost to visit the nodes of $\mathcal{T}$ and the cost to read the object lists. Since $\mathcal{T}$ is a BAR-B-tree with possible removal of guards during construction, which only reduces the number of nodes, the cost of visiting $\mathcal{T}$ can still be bounded by Theorem 5.5. So we only concentrate on the cost of reading the object lists of the visited leaves.

Note that the region of any visited leaf must intersect Q. Next, we make a distinction between leaves whose regions also intersect the boundary of Q_ε and those whose regions are completely contained in Q_ε. There are at most ε^γ leaves of the former type [37]. For these leaves we can check all objects intersecting their regions using $O(\lceil \lambda/B \rceil)$ I/O-operations each.

The latter type of leaves can be covered in t disjoint subtrees $\mathcal{T}_{\mu_1}, \ldots, \mathcal{T}_{\mu_t}$, such that $R(\mu_i) \subseteq Q_\varepsilon$ and $R(p(\mu_i)) \not\subseteq Q_\varepsilon$, where $p(\mu_i)$ denotes the parent of μ_i. Note that there are $O(\varepsilon^\gamma)$ such subtrees following the same reasoning in the proof

of Theorem 5.5. For any μ_i, let $k(\mu_i)$ denote the number of objects that intersect $R(\mu_i)$ and have at least one guard in $R(\mu_i)$. Since $\mathcal{T}_{\mu_i}$ is a BAR-tree (with pruning) built on the $O(k(\mu_i))$ guards of these objects, we have $|\mathcal{T}_{\mu_i}| = O(k(\mu_i))$. Furthermore, since each object intersecting Q_ε has guards in at most a constant number of these subtrees, we have $\sum_{i=1}^t k(\mu_i) = O(k_\varepsilon)$.

By Lemma 5.1 we know that the leaves of any $\mathcal{T}_{\mu_i}$ are stored in $O(\lceil|\mathcal{T}_{\mu_i}|/B\rceil)$ blocks. Let $\nu_1, \nu_2, \ldots$ be the leaves of $\mathcal{T}_{\mu_i}$ ordered according to an in-order traversal of $\mathcal{T}$. From our blocking algorithm we know that these leaves are partitioned into $O(\lceil|\mathcal{T}_{\mu_i}|/B\rceil)$ subsets, each stored in a block. Since in a block, the objects intersecting consecutive leaves are also stored consecutively in the object list, the total number of I/O-operations to read these objects is $O(\lceil|\mathcal{T}_{\mu_i}|/B\rceil + \lambda|\mathcal{T}_{\mu_i}|/B) = O(\lceil\lambda|\mathcal{T}_{\mu_i}|/B\rceil)$. Thus, the overall cost for reading the object lists for all the leaves whose regions are completely inside Q_ε is

$$
\begin{aligned}
O\left(\sum_{i=1}^t \left\lceil \frac{\lambda|\mathcal{T}_{\mu_i}|}{B} \right\rceil\right) &= O\left(t + \sum_{i=1}^t \frac{\lambda|\mathcal{T}_{\mu_i}|}{B}\right) \\
&= O\left(t + \sum_{i=1}^t \frac{\lambda k(\mu_i)}{B}\right) \\
&= O(\varepsilon^\gamma + \lambda k_\varepsilon/B)
\end{aligned}
$$

I/O-operations.

Computing the unique reference point in the intersection of an object and the query range and checking whether this point is contained in the cell can all be done in main memory, so the bound on the query cost follows.   □

Updating the oBAR-B-tree.   The oBAR-B-tree can be updated by first updating the BAR-B-tree $\mathcal{T}$, followed by updating the object lists. Since each object only has a constant number of guards, the cost of the former can be effectively bounded as in Table 5.1. Unfortunately, we do not have a worst-case or amortized bound for the latter, as one object may intersect many regions of the leaves of $\mathcal{T}$. Nevertheless, since on average an object only intersects $O(\lambda)$ such regions, we expect the actual update cost to be small in practice.

Part II

Bounding Volume Hierarchies
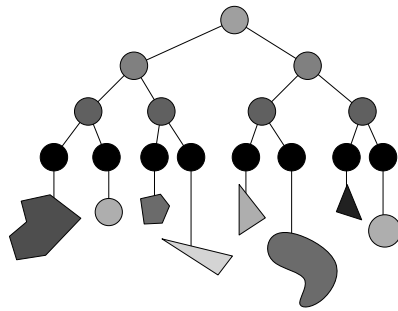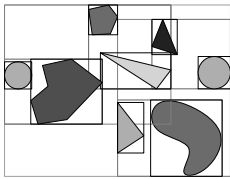
Chapter 6

Bounding-Volume Hierarchies on c-DOPs

A bounding-volume hierarchy (BVH), see Section 1.3.2, on a set S of n 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 constant-complexity bounding volume of the set of objects corresponding to the leaves in the subtree of ν. In this chapter we use a *c-discretely oriented polytope*, or c-DOP as a bounding volume. As in Chapter 2 let $\mathcal{C}$ be a fixed set of c non-parallel hyperplanes. A c-DOP is a convex polytope and each of its facets is parallel to some hyperplane in $\mathcal{C}$. Hence, a c-DOP has at most $2c$ facets. When we speak of a DOP, we always mean a c-DOP. The bounding DOP stored at ν is denoted by $bdop(\nu)$. The better $bdop(\nu)$ approximates the set of objects stored below ν the less unnecessary nodes in the hierarchy are visited.

In this chapter we first describe how to construct a BVH for arbitrary c-DOPs in arbitrary dimensions with optimal worst-case query time. Secondly, we describe a BVH for c-DOPs in the plane that yields better query times when the input c-DOPs do not overlap each other too much. The second structure uses the first structure as a plug-in to handle configurations of overlapping c-DOPs locally.

6.1 A worst-case optimal DOP-tree

In this section a generalization of the cs-boxtree by Agarwal et al. [2] is described to obtain a DOP-tree with optimal query time. More exactly we show the following.

Theorem 6.1 *For any set of (possibly intersecting) c-DOPs in $\mathbb{R}^d$ ($c \geq d \geq 2$), there is a constant-degree DOP-tree such that DOP-queries can be answered in time $O(n^{1-1/c} + k)$, where k is the number of reported answers. Moreover, the bound on the query time is optimal when convex bounding volumes are used. This*
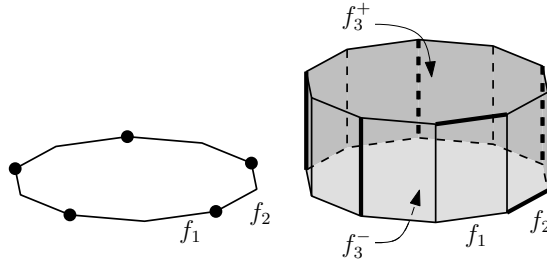
63

Figure 6.1: An example on how to construct the c-DOP for $\mathbb{R}^d$ which is used to generate a worst-case input set for $c = 6$ and $d = 3$. On the left a regular 10-gon is shown. (Note that $2c + 4 - 2d = 10$.) The vertices between the matched edges are highlighted. On the right the resulting 6-DOP after taking the Cartesian product of the 10-gon with the interval $[0, 1]$ for the third dimension. The edges between two matched facets are highlighted. Note that the facets f_1 and f_2 which were matched on the left are no longer matched on the right. The facet f_1 is now matched with the top-facet f_3^+ and f_2 with the bottom-facet f_3^-.

is true even for point queries: for any n, there is a set S of n c-DOPs such that for any constant-degree DOP-tree $\mathcal{T}$ on S there is a query point p not contained in any DOP from S such that a query with p visits $\Omega(n^{1-1/c})$ nodes in $\mathcal{T}$.

Below we prove the two statements in the theorem separately. The lower bound is proved in Lemma 6.2 and the upper bound in Lemma 6.4.

Lemma 6.2 *For any n, c and d with $c \geq d \geq 2$, there is a set S of n c-DOPs in $\mathbb{R}^d$, such that for any constant-degree BVH $\mathcal{T}$ using convex bounding volumes on S there is a point, not contained in any DOP from S, such that a query with this point visits $\Omega(n^{1-1/c})$ nodes in $\mathcal{T}$.*

Proof: We first explain how to construct a set $S = \{D_1, \ldots, D_n\}$ as in the Lemma by making n modified copies of a d-dimensional polyhedron D defined as follows.

In the plane D is a regular $2c$-gon. In higher dimensions D is constructed as follows, see Figure 6.1 for an example in $\mathbb{R}^3$. We start with a regular $(2c + 4 - 2d)$-gon on a 2-dimensional plane, and we extend this polygon in the remaining $d - 2$ dimensions by taking the Cartesian product of the polygon with the interval $[0, 1]$ for every dimension. Note that an interval for dimension $2 < i \leq d$ used in the Cartesian product defines two hyperplanes, H_i^- and H_i^+, which bound D in this dimension.

The DOPs $D_1, \ldots, D_n$ are constructed by moving the facets of D slightly in certain ways. For this we need a perfect matching of the $(d - 1)$-dimensional facets of D that matches every facet to an adjacent one. We find such a matching by first determining a perfect matching of the edges of the $(2c + 4 - 2d)$-gon

(which is trivial). Then, for each $2 < i \leq d$ in order, we add the hyperplanes H_i^- and H_i^+, which bound D in the i-th dimension. Let f_i^- and f_i^+ be the facets defined by these hyperplanes. We cannot add the pair (f_i^-, f_i^+) to the current matching, since f_i^- and f_i^+ are not adjacent. But both facets are adjacent to all facets in the current matching, so we can choose any pair (f_1, f_2) from the current matching, and match up f_1 with f_i^+ and f_2 with f_i^-.

The set S is constructed by making copies of D in which the defining half-spaces have been offset slightly from their original positions—choosing the offsets small enough to ensure that all DOPs have $2c$ non-degenerate $(d-1)$-dimensional facets, and keeping all facets parallel to the original facets. When we fix the exact position of the affine hull of any $(d-2)$-dimensional facet f of such a variation D', this fixes the exact position of the two halfspaces that define the $(d-1)$-dimensional facets adjacent to f. We can therefore uniquely specify a DOP D' by fixing, for each of the c pairs (f_1, f_2) in the matching, the intersection point of $f_1 \cap f_2$ and a plane A orthogonal to it. For the DOPs in S, we restrict those intersection points to a discrete set of $n^{1/c}$ points on a line whose normal is the sum of the normals of the intersections of f_1 and f_2 with A. For an example in $\mathbb{R}^2$ see Figure 6.2. By selecting the points such that they lie close enough to D, we guarantee that any DOP D' that adheres to these restrictions has $2c$ non-degenerate $(d-1)$-dimensional facets.

The set S consists of n different DOPs that are defined by taking all possible combinations of defining points from c sets $V_1, V_2, ..., V_c$ on c lines $\ell_1, \ell_2, ..., \ell_c$ of $n^{1/c}$ points each.

We now prove that for any constant-degree BVH on S there is a point query without answers that visits $\Omega(n^{1-1/c})$ nodes. Each set of defining points V_i divides the line ℓ_i in $n^{1/c} - 1$ bounded segments and two partially unbounded segments. Let P be a set of $c(n^{1/c} - 1)$ points with one point in the interior of each bounded segment of each line—see Figure 6.2. Note that the points in P are not contained in any DOP in S. Now any constant-degree BVH on S contains $\Theta(n)$ bounding volumes $\mathcal{B}$ that each contain at least two DOPs from S. Any two DOPs from S differ in the defining point from at least one set V_i and therefore their convex bounding volume contains at least one point of P on ℓ_i. Hence any convex bounding volume in $\mathcal{B}$ contains at least one point of P. There are $\Theta(n)$ such bounding volumes and only $O(n^{1/c})$ points in P. It follows that there exists a point $p \in P$ which is contained in at least $\Theta(n)/O(n^{1/c}) = \Omega(n^{1-1/c})$ bounding volumes. If we query with this point p we thus have to visit at least $\Omega(n^{1-1/c})$ nodes. $\qquad\square$

This lower bound shows that in general we cannot hope for a better point-query time than $O(n^{1-1/c})$. Next we describe a DOP-tree—we call this DOP-tree a CSP-DOP tree—achieving this bound even for range queries with c-oriented ranges. The CSP-DOP tree is simply a c-dimensional CS-priority-box tree [2, 49] since any DOP D is defined by $\bigcap_{i=1}^{c} Slab_i(x_i^-, x_i^+)$ where $Slab_i(x_i^-, x_i^+)$ is the smallest slab enclosing D, see Section 2.1.2. The construction of a CSP-DOP tree is therefore similar to the construction of a CS-priority-box tree. For complete-
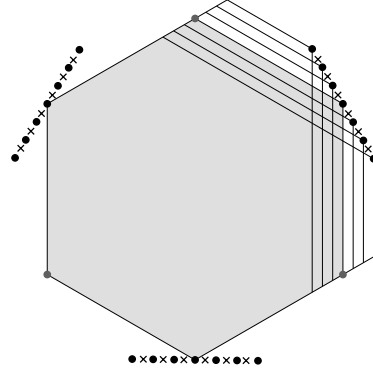
Figure 6.2: Some of the input DOPs for the lower bound example in the plane for point-containment queries in bounding DOP hierarchies where $c = 3$. The input DOPs shown in the figure only differ at one defining point in V. The defining points of the input DOPs are depicted as points and the possible point-containment queries as crosses.

ness, we briefly describe the construction.

Let S be the set of n DOPs. We map every DOP in S to a point p in $\mathbb{R}^{2c}$, which we call the *configuration space*, with coordinates $(x_1^-, \ldots, x_c^-, x_1^+, \ldots, x_c^+)$. We then build a BSP on these points. This intermediate BSP $\mathcal{T}$ is built recursively, starting at the root: at every node ν we store the $2c$ points that have the largest value of x_i^+ or the smallest value of x_i^- of the current set of points for some $i \in \{1, ..., c\}$ and we store these extreme points at ν; we then split the remaining points with an axis-parallel hyperplane such that approximately half of the points lie on either side, like in a kd-tree, and distribute these points among the children of ν. The orientations of the splitting planes are chosen as in a kd-tree, cycling through all orientations on any path down the tree.

Next we replace every point in the intermediate BSP $\mathcal{T}$ by its corresponding DOP. At every internal node ν we store a bounding DOP containing all DOPs below ν. The DOPs corresponding to the $2c$ extreme points are stored in a priority leaf directly below ν. Note that a DOP corresponding to such a point is extreme in the direction of one of the normals of the c orientations.

The intermediate BSP is technically not a real $2c$ dimensional kd-tree (because at every step we take out some extreme points) but it is easy to see that our tree has the same size ($O(n)$), depth ($O(\log n)$) and construction time ($O(n \log n)$) as an ordinary $2c$ dimensional kd-tree. For the analysis of the query time in the CSP-DOP tree we use the following fact about kd-trees, which also holds for the intermediate BSP. This fact is given here without proof.

Lemma 6.3 *The number of cells in a c-dimensional kd-tree that intersect an axis-parallel f-flat ($0 \leq f \leq c$) is $O(n^{f/c})$.*

Lemma 6.4 *The number of visited nodes by a* DOP-*query Q in a* CSP-DOP *tree is $O(n^{1-1/c} + k)$.*

Proof: Let $Q = \bigcap_{i=1}^{c} Slab_i(x_i^-(Q), x_i^+(Q))$ where $Slab_i(x_i^-(Q), x_i^+(Q))$ is the smallest slab enclosing Q. We define the weight of a node ν to be the number of DOPs stored in the subtree rooted at ν. In the remainder of this proof we only consider internal nodes of weight at least $2c$ as the total number of visited nodes is at most a constant constant factor off. Let ν be a visited node of weight at least $2c$. We distinguish two cases: either Q intersects at least one DOP stored in a priority leaf of ν, or Q does not intersect any DOP stored in a priority priority leaf of ν.

There are $O(k)$ nodes of the first case. For the second case we argue as follows. The DOPs stored in the priority leaves of ν are separated from Q by a hyperplane h through a facet f of Q. Not all DOPs can be separated by the same h otherwise h would also separate the DOP that extends furthest in the direction of the inner normal of f. This would contradict the fact that the query with Q visits ν. We therefore have at least two distinct hyperplanes through facets of Q separating a DOP in the subtree of ν from Q.

Assume w.l.o.g. that $x_i = x_i^-(Q)$ is one of these separating hyperplanes, and let D be the input DOP it separates from Q. Then we have $x_i^+(D) \leq x_i^-(Q)$, but there is also a DOP D' with $x_i^+(D') > x_i^-(Q)$, otherwise $bdop(\nu)$ would not intersect Q. The points in configuration space on which D and D' are mapped thus 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. The same argument is applicable to the second hyperplane $x_j = x_j^-(Q)$ or $x_j = x_j^+(Q)$. Therefore there is a hyperplane in configuration space with points on or on opposite sides of $x_j^+ = x_j^-(Q)$ or $x_j^- = x_j^+(Q)$, respectively.

We can conclude the following. If Q visits a node ν of the second case 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 node in the intermediate BSP corresponding to ν. This cell is then also intersected by the intersection of these two hyperplanes, which is a $(2c-2)$-flat. By Lemma 6.3 there are only $O(n^{1-1/c})$ of such nodes. □

6.2   A framework for DOP-trees on input with low stabbing number

In this section we consider the setting that is probably most relevant in practice: the DOPs may intersect, but not too much. To quantify this we use the so-called *stabbing number* of the input set S, see Section 1.2.2. This is the smallest number σ such that no point in the plane is contained in more than σ DOPs from S. For example, if the DOPs in S are disjoint, then $\sigma = 1$. In practice, especially when
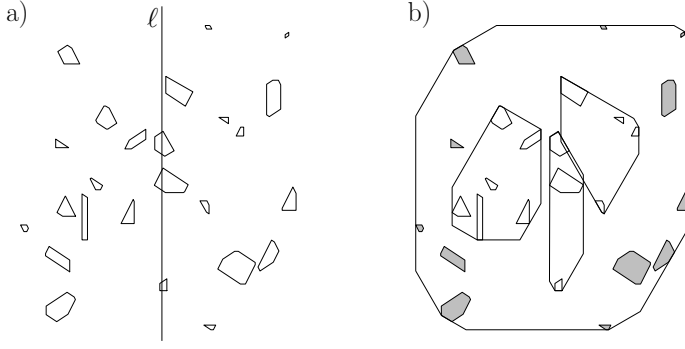
Figure 6.3: Transforming a BSP-node with a splitting line ℓ into a 6-DOP-node with 12 priority leaves containing the gray DOPs.

the DOPs from S are bounding DOPs of an underlying set of disjoint objects, one may expect that σ is some small constant.

6.2.1   From BSP-trees to DOP-trees

In this section we describe and analyze a general method for creating a DOP-tree on a set S of c-DOPs in the plane from a c-oriented BSP on a set of representative points. Our method creates a DOP-tree of branching degree at most $2c + 3$, which can be turned into a binary DOP-tree as a postprocessing step. The method works as follows.

1. Pick an arbitrary representative point in each DOP in S. Let P be the resulting set of representative points.

2. Construct a c-oriented BSP $\mathcal{T}_P$ on the set P.

3. Next, transform the BSP $\mathcal{T}_P$ into a BVH $\mathcal{T}_S$, see Figure 6.3, by inserting the DOPs from S into $\mathcal{T}_P$ in a top-down manner, starting at the root of $\mathcal{T}_P$ with the complete input set S. A recursive call proceeds as follows. We get as input a set of DOPs $S_\nu \subseteq S$ which is to be inserted into the subtree rooted at a node ν. Instead of the splitting line ℓ_ν stored at ν, we store at ν the bounding DOP $bdop(\nu) := bdop(S_\nu)$ of all DOPs in S_ν, thus converting a BSP-node into a BVH-node. We then split S_ν into three sets. The set S_ν^- contains all DOPs in S_ν that lie completely to the left of ℓ_ν, the set S_ν^+ contains all DOPs in S_ν to the right of ℓ_ν, and $S_\nu^\times$ contains the DOPs in S_ν that are intersected by ℓ_ν.

   (i) The set S_ν^- (if non-empty) is inserted recursively into the left child ν^- of ν, and the set S_ν^+ (if non-empty) is inserted recursively into the right child ν^+ of ν.

(ii) Recall that $\mathcal{C}$ is the set of lines defining the c orientations of the splitting lines used by the BSP $\mathcal{T}_P$ (and the DOPs). Remove the line parallel to ℓ_ν from $\mathcal{C}$, to obtain a set $\mathcal{C}^*$. Construct a DOP-tree for the set $S_\nu^\times$ (if non-empty) by calling a subroutine $CreateSubTree(S_\nu^\times, \mathcal{C}^*)$ and make this tree a subtree of ν.

Note that the algorithm above can create nodes of degree one when only one of the subsets for which a subtree is created is non-empty. We contract these nodes with their only child in a postprocessing step. Next we describe the $CreateSubTree$ subroutine.

$CreateSubTree(S^*, \mathcal{C}^*)$ is a recursive subroutine, which works as follows.

1. If $\mathcal{C}^* = \emptyset$, we return a CSP-DOP tree for S^* as described in Section 6.1. Otherwise, proceed with steps 2–4.

2. Create a root node μ for the tree to be constructed. Define $S_\mu := S^*$ and $\mathcal{C}_\mu := \mathcal{C}^*$. Store the bounding DOP $bdop(S_\mu)$ at μ.

3. Let $\{\vec{n_1}, \ldots, \vec{n_{2c}}\}$ be the normals to the lines in $\mathcal{C}$ (not only the lines in $\mathcal{C}^*$). Note that for every line we have normals in both directions. For each normal $\vec{n_i}$ in turn, we remove from S_μ the DOP D_i extending furthest in the direction $\vec{n_i}$, and store it in a leaf μ_i directly below μ. We call such leaves *priority leaves*. Note that a DOP stored in a priority leaf because it is extreme in some direction $\vec{n_i}$ is not considered when we look for extreme DOPs in subsequent directions $\vec{n_j}$ with $j > i$.

4. Let $S' := S_\mu \setminus \{D_1, \ldots, D_{2c}\}$ be the remaining DOPs in S_μ. If S' is not empty, we find a splitting line ℓ_μ parallel to the first line in $\mathcal{C}_\mu$, such that ℓ_μ splits the set of representative points of the DOPs in S' in two sets of roughly equal size. Now we create three sets of DOPs S_μ^-, S_μ^+, and $S_\mu^\times$ that contain the DOPs in S' that lie completely to the left, completely to the right, or across ℓ_μ, respectively.

   (i) Construct two subtrees of μ by calling $CreateSubTree(S_\mu^-, \mathcal{C}_\mu)$ and $CreateSubTree(S_\mu^+, \mathcal{C}_\mu)$, respectively. (If S_μ^- or S_μ^+ is empty the corresponding call is skipped.)

   (ii) Let $\mathcal{C}_\mu^*$ be the set $\mathcal{C}_\mu$ with its first line, which is parallel to ℓ_μ, removed. Create a subtree of μ for $S_\mu^\times$ (if this set is non-empty) by calling $CreateSubTree(S_\mu^\times, \mathcal{C}_\mu^*)$.

This finishes the description of the construction algorithm. In the next two sections we prove bounds on the properties and the performance of the DOP-tree created with the algorithm above.

6.2.2 Properties of the DOP-tree

We first need to introduce some definitions. Any node which was already present in the original BSP $\mathcal{T}_P$ is called a c-node. For $0 \leqslant m < c$, an m-node ν is any

node constructed in step 1 or 2 of a call to *CreateSubTree*$(S^*, \mathcal{C}^*)$ with $|\mathcal{C}^*| = m$. An m-tree is a subtree rooted at an m-node-node; thus an m-tree only contains m'-nodes for $m' \leqslant m$. The k-parent of an m-node ν, for some $m < k \leqslant c$, is its lowest ancestor that is a k-node.

We start with two easy lemmas on the size and the depth of $\mathcal{T}_S$.

Lemma 6.5 $\mathcal{T}_S$ *uses* $O(n)$ *storage.*

Proof: The complete structure is a bounding-volume hierarchy that stores n DOPs in $O(n)$ leaves. All nodes have degree at least two. Hence the total number of nodes in the hierarchy is $O(n)$, and each of them uses only $O(1)$ storage.   □

Lemma 6.6 *The depth of* $\mathcal{T}_S$ *is* $O(depth(\mathcal{T}_P))$.

Proof: Consider any c-node ν which had height h in $\mathcal{T}_P$. The number of DOPs stored below ν is therefore at most 2^h. When traversing any path from the 'middle child' $\nu^\times$ of ν down to a leaf below $\nu^\times$, the number of nodes stored below the current node is reduced by a factor at least two with each step, except, possibly, when we go from an m-node to an $(m-1)$-node (for $0 < m < c$). The total height of the subtree rooted at $\nu^\times$ is therefore at most $h + c - 1$. It follows that by the transformation from $\mathcal{T}_P$ to $\mathcal{T}_S$, the height of ν increases by at most an additive term c. The depth of $\mathcal{T}_S$ must therefore be at most $\text{depth}(\mathcal{T}_P) + c = O(\text{depth}(\mathcal{T}_P))$.   □

Lemma 6.7 *Given* BSP $\mathcal{T}_P$, *the* BVH $\mathcal{T}_S$ *can be constructed in* $O(n \cdot depth(\mathcal{T}_P))$ *time.*

Proof: We first analyse how much time it costs to distribute the DOPs S_ν that have to be stored under an m-node ν $(0 < m < c)$ among the children of ν. Suppose the DOPs in S_ν are given sorted by the coordinates of their representative points, when projected on a line orthogonal to the first line in $\mathcal{C}_\nu$—that is the line which is parallel to the splitting line ℓ_ν that is chosen for ν. In $O(|S_\nu|)$ time, we can find the bounding DOP $bdop(\nu)$ of ν, we can find the DOPs that have to be placed in the priority leaves, choose a splitting line ℓ_ν that divides the representative points of the remaining DOPs into subsets of roughly equal size, and distribute the DOPs among the three subtrees of ν, while keeping them sorted. A c-node can obviously be constructed in the same time bound, since a c-node requires less work (c-nodes have no priority leaves, and the splitting line is already known).

Note that any DOP is routed through at most one node on every level of $\mathcal{T}_S$. Since the depth of the tree is $O(\text{depth}(\mathcal{T}_P))$ and there are n DOPs, the total time needed for distributing the DOPs is $O(n \cdot \text{depth}(\mathcal{T}_P))$, plus the time needed for sorting. Every m-node whose parent is an m-node as well, gets its set of DOPs from its parent in the correct order. However, when an m-node gets its DOPs from its $(m+1)$-parent, they have to be sorted. Thus, every DOP undergoes sorting $c = O(1)$ times, so the total time spent sorting is $O(n \log n)$.

This accounts for the cost for distributing the DOPs over the priority leaves and 0-trees. To finish off, the 0-trees are built in $O(n \log n)$ total time as described in Section 6.1.

Adding these bounds proves the bound on the total construction time. $\square$

6.2.3   Analysis of the query time

In this section we prove the bounds on the number of nodes in $\mathcal{T}_S$ visited by a point query and a DOP-query. To this end, we associate a region of the plane with every node ν in $\mathcal{T}_S$. Consider all the ancestors of ν. At each such ancestor μ, we used a splitting line ℓ_μ to guide the construction; the node ν can either lie in the subtree corresponding to the subset lying completely in one of the half-planes defined by ℓ_μ, or not. (In the latter case ν is a priority leaf directly below μ, or ν lies in the subtree created for $S_\mu^\times$.) The region of ν, denoted R(ν), is defined as the intersection of all half-planes corresponding to ancestors of the former type.

Observation 6.8 For any node ν in $\mathcal{T}_S$ the bounding DOP $bdop(\nu)$ is a subset of R(ν).

Proof: Suppose this is not the case, then at least one object in S_ν is (partially) outside R(ν). This object thus intersects one of the splitting lines of an ancestor μ of ν, or it lies completely to the 'wrong' side of such a splitting line. But such an object would not be an object in S_ν, by construction. $\square$

Corollary 6.9 *The number of visited c-nodes in $\mathcal{T}_S$ is at most the number of nodes that would be visited by the same query in $\mathcal{T}_P$.*

We now analyse the number of nodes visited by a point query. Here we make use of the fact that any 0-tree stores $O(\sigma)$ DOPs in the worst case—we prove this in Section 6.2.5 (Theorem 6.26)—, where σ is the stabbing number of the input set.

Lemma 6.10 *The number of nodes visited by a point query in $\mathcal{T}_S$ is $O(\sigma^{1-1/c} \cdot depth(\mathcal{T}_P) \cdot \log^{c-1} n + k)$.*

Proof: In $\mathcal{T}_P$ a point query q visits $O(depth(\mathcal{T}_P))$ nodes, so by Corollary 6.9 q is contained in the bounding DOPs of at most $O(depth(\mathcal{T}_P))$ c-nodes in $\mathcal{T}_S$. For each such node, its $(c-1)$-subtree may also contain bounding DOPs that contain q. A point query in an m-tree $(0 < m < c)$ is contained in the bounding DOPs of $O(\log n)$ m-nodes, since the depth of an m-tree is $O(\log n)$. At each m-node, its $(m-1)$-subtree may also contain bounding DOPs that contain q. Thus, an m-tree on n DOPs contains at most $T(n,m) = O(\log n)(1 + T(n, m-1))$ bounding DOPs that contain q in total, where $T(n,1) = O(\log n)$. This solves to $T(n,m) = O(\log^m n)$, and the total number of nodes visited in $\mathcal{T}_S$, excluding 0-trees, is therefore $O(depth(\mathcal{T}_P) \cdot \log^{c-1} n)$. For each visited 1-node ν, we may
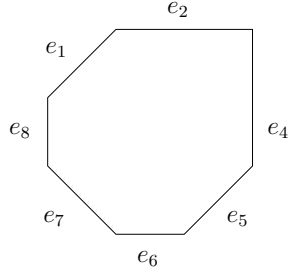
Figure 6.4: A 4-DOP whose orientations are axis-parallel and the two diagonals. The edges e_2 and e_4 are not adjacent since there is a diagonal orientation between the horizontal and the vertical orientation in the cyclic ordering. All other edges sharing a corner are adjacent.

need to visit its 0-subtree $\mathcal{T}_\nu$. By Theorem 6.26 such a subtree $\mathcal{T}_\nu$ stores only $O(\sigma)$ DOPs, and by Theorem 6.1 it can be queried by visiting $O(\sigma^{1-1/c} + k_\nu)$ nodes, where k_ν is the number of answers found in $\mathcal{T}_\nu$. Thus we visit $O(\sigma^{1-1/c} \cdot \text{depth}(\mathcal{T}_P) \cdot \log^{c-1} n + k)$ nodes in total. □

Next we analyse the number of internal nodes visited by a DOP-query in $\mathcal{T}_S$; the number of leaf-nodes is at most a constant factor more. To this end we introduce the notion of *defining segments*. The set of defining segments of an m-node ν, denoted $\text{DefSeg}(\nu)$, is the intersection of $bdop(\nu)$ with the splitting lines ℓ_μ of all k-parents μ of ν (for every $k \in \{m+1, ..., c\}$). Note that any DOP in the subtree rooted at ν intersects all defining segments of that node.

In the following we say that two edges of a query range Q (or some other c-DOP) are *adjacent* if their outward normals are adjacent in the cyclic ordering of all $2c$ possible outward normals, see Figure 6.4. Hence, if Q has $2c$ edges then this corresponds to the usual definition. If, however, some of the possible outward normals are not used by Q, then two edges may be non-adjacent in the above sense of the word, even when they are incident to the same DOP-vertex.

We distinguish the following types of visited nodes:

inner nodes are nodes ν such that $bdop(\nu)$ is completely contained in Q;

side nodes are nodes ν such that $bdop(\nu)$ intersects only one edge of Q;

stabbing nodes are nodes ν such that $bdop(\nu)$ intersects at least two edges of Q (but no vertex), and have a defining segment that intersects at most one edge of Q;

embracing nodes are nodes ν such that all defining segments $\text{DefSeg}(\nu)$ and $bdop(\nu)$ intersect the interiors of at least two non-adjacent edges of Q;

corner nodes are nodes ν such that $bdop(\nu)$ contains at least one vertex of Q;

Observe that a visited node that is not an inner node, must intersect at least one edge of Q. If it intersects only one edge, it is a side node. If it intersects two adjacent edges, then (by our definition of adjacency) it must also intersect their common vertex, so it is a corner node. Otherwise it intersects at least two non-adjacent edges and is therefore a stabbing node or an embracing node, depending on the defining segments. Thus, together these types cover all nodes whose bounding DOPs intersect Q. Note that some embracing nodes might be corner nodes as well. Also note that, in the definition of embracing nodes, the condition that $bdop(\nu)$ intersects the interiors of at least two non-adjacent edges of Q is implied by the condition that the defining segments in $\text{DefSeg}(\nu)$ intersect the interiors of at least two non-adjacent edges of Q when $\text{DefSeg}(\nu)$ is not empty; so the condition on $bdop(\nu)$ is only needed for the case where ν is a c-node for which $\text{DefSeg}(\nu)$ is empty.

Lemma 6.11 *The number of inner nodes, side nodes, and stabbing nodes visited by a DOP-query Q in all m-trees ($m < c$) of $\mathcal{T}_S$ together is $O(k)$.*

Proof: The bound on the number of inner nodes is easy to see, so we concentrate on the side and stabbing nodes. Let S_ν be the set of DOPs stored below some node ν.
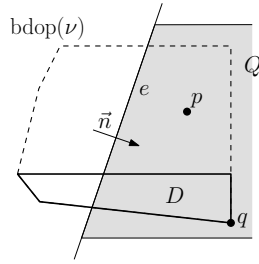
Figure 6.5: The bounding DOP $bdop(\nu)$ of a side node ν intersects only one edge e of the query Q. The DOP D is extreme in the direction of $\vec{n}$, the inward normal of e.

First we bound the number of side nodes. Let ν be a side node and let e be the edge of Q it intersects. Let H be the halfplane that contains Q and is bounded by the line that contains e. Let $\overrightarrow{n}$ be the direction of the inward normal of e—see Figure 6.5. Let p be any point inside $bdop(\nu) \cap Q$, and let q be the point that extends furthest in direction $\overrightarrow{n}$, in the DOP $D \in S_\nu$ that extends furthest in that direction. Obviously, the line segment pq lies inside H, and if q would not lie in Q, the segment $pq \in bdop(\nu)$ would intersect another edge of Q than e—contradicting the definition of a side node. Hence, D intersects Q. Since D is the DOP of S_ν which extends furthest in the direction of a normal to one of the orientations from $\mathcal{C}$, it is stored in a priority leaf at ν. We charge ν to that priority leaf. By definition, only $O(k)$ priority leaves contain DOPs that intersect Q, and
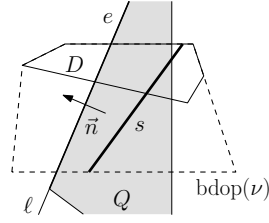
Figure 6.6: The edge e of a query Q intersects $bdop(\nu)$ of a stabbing node ν, but not s, one of the defining segments of ν. The DOP D is extreme in the direction of $\vec{n}$, the normal of e directed from s towards ℓ.

each priority leaf is charged at most once (namely by its parent). Therefore the total number of side nodes is $O(k)$.

It remains to bound the number of stabbing nodes. Let ν be a stabbing node. Let s and e be a defining segment of ν and an edge of Q, such that e intersects $bdop(\nu)$ but not s—see Figure 6.6 for an example. By the definition of stabbing nodes, such a pair exists. Let ℓ be the line that contains e. Since $bdop(\nu)$ is convex and by definition does not include any vertex of e, no line segment inside $bdop(\nu)$ can intersect ℓ without intersecting e. This implies that s must lie completely on one side of ℓ. Let D be the DOP stored below ν that is extreme in the direction of the normal $\vec{n}$ of e that is directed from s towards ℓ. Since any DOP stored below ν intersects all defining segments of ν, the DOP D must intersect s in some point p. Furthermore, since $bdop(\nu)$ intersects ℓ, the DOP D must contain a point q on the other side of ℓ. Since D is convex, it contains the line segment pq that intersects ℓ; hence D intersects e, and in the immediate neighborhood of that intersection, D intersects Q. Again, we charge ν to the priority leaf that contains D, and find that the total number of stabbing nodes is $O(k)$. $\square$

We now bound the number of corner nodes.

Lemma 6.12 *The number of corner nodes visited by a DOP-query Q in $\mathcal{T}_S$ is $O(\sigma^{1-1/c} \cdot depth(\mathcal{T}_P) \cdot \log^{c-1} n + k)$.*

Proof: The number of corner nodes visited by a DOP-query Q in $\mathcal{T}_S$ is the number of nodes ν whose bounding DOPs $bdop(\nu)$ contain at least one vertex of Q, which are exactly the nodes visited while doing a point query for each vertex of Q. The bound thus follows directly from Lemma 6.10. $\square$

We are now left with proving a bound on the number of embracing nodes. We start by characterizing the places where they may be found in the tree.

Observation 6.13 All ancestors of embracing nodes are embracing nodes.

Proof: Once a node has a defining segment that intersects less than two edges of Q, all its descendants have a defining segment that intersects less than two
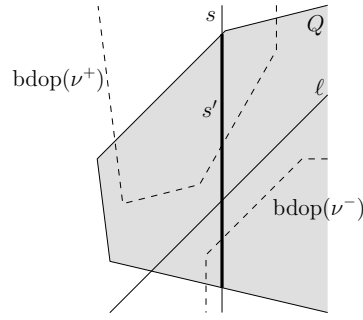
Figure 6.7: The query Q intersects $bdop(\nu)$ of an embracing node ν. The splitting line ℓ stored at ν separates the bounding DOPs of its children $bdop(\nu^+)$ and $bdop(\nu^-)$. The part s' within Q of the defining segment s cannot be completely contained in both $bdop(\nu^+)$ and $bdop(\nu^-)$.

edges of Q, and once a node's bounding DOP intersects less than two edges of Q, all its descendants intersect less than two edges of Q. So all descendants of inner nodes, side nodes, and stabbing nodes must be inner nodes, side nodes, or stabbing nodes, so no embracing node can have an inner, side or stabbing node as an ancestor. With the same reasoning as for side nodes, a corner node cannot have any embracing descendants either, unless it happens to be an embracing node itself. It follows that every ancestor of an embracing node is an embracing node. □

Lemma 6.14 *An m-tree $(1 \leqslant m < c)$ on n DOPs whose root is an embracing node contains $O(\log^m n)$ embracing nodes—excluding embracing nodes in 0-trees.*

Proof: Let ν be an embracing m-node. Since $m < c$, the node ν has at least one defining segment s, and since ν is an embracing node, s intersects two edges of Q—see Figure 6.7. Let $s' := s \cap Q$. Since the bounding DOPs $bdop(\nu^-)$ and $bdop(\nu^+)$ of the left and right child of ν are disjoint, they cannot both contain s' completely. Any child that does not contain s' completely, has a defining segment that intersects less than two edges of Q and therefore cannot be an embracing node. Hence, at most one of ν^- and ν^+ can be an embracing node. In addition, the child created for $S_\nu^\times$ may be an embracing $(m-1)$-node. The maximum number of embracing nodes in an m-tree on n DOPs is therefore

$$T(n, m) \leqslant 1 + T(n/2, m) + T(n, m - 1),$$

where $T(n, m) = 1$ for $n \leqslant 2c$, and $T(n, 0) = 0$ for any n. This recurrence solves to $T(n, m) = O(\log^m n)$. □

Lemma 6.15 *The number of embracing nodes visited by a* DOP*-query Q in the tree $\mathcal{T}_S$, excluding its 0-trees, is $O(tna(\mathcal{T}_P, Q) + tna\_disj(\mathcal{T}_P, Q) \log^{c-1} n)$, where $tna(\mathcal{T}_P, Q)$ is the number of regions in $\mathcal{T}_P$ that intersect at least two non-adjacent edges of Q, and $tna\_disj(\mathcal{T}_P, Q)$ is the maximum size of a set of such cells that are pairwise disjoint.*

Proof: By Observation 6.13 the embracing nodes together form a subgraph $\mathcal{T}'$ of $\mathcal{T}_S$ that is a tree rooted at the root of $\mathcal{T}_S$.

First consider the c-nodes of $\mathcal{T}'$. By Observation 6.8, the number of such nodes is at most the number of cells in $\mathcal{T}_P$ that intersect at least two non-adjacent edges of Q, which is $O(tna(\mathcal{T}_P, Q))$ by definition.

The other nodes of $\mathcal{T}'$ are grouped into subtrees. The roots of these trees are $(c-1)$-nodes that have a c-node in $\mathcal{T}'$ as parent. By Lemma 6.14 any such subtree has $O(\log^{c-1} n)$ nodes. It remains to bound the number of subtrees. We just argued that the number of c-nodes in $\mathcal{T}'$ is $O(tna(\mathcal{T}_P, Q))$ by definition, leading to a bound of $O(tna(\mathcal{T}_P, Q))$ on the number of subtrees. We now strengthen the bound on the number of subtrees to $O(tna\_disj(\mathcal{T}_P, Q))$. Let μ be an embracing $(c-1)$-node in $\mathcal{T}'$ and let ν be its parent. The only defining segment of μ, the splitting line used at ν clipped to $bdop(\nu)$, must intersect two non-adjacent edges of Q by definition. Then the left and right child of ν in $\mathcal{T}_P$ also intersect those two non-adjacent edges of Q. We now bound the number of embracing nodes in $\mathcal{T}_P$ having two embracing child nodes. Consider the subgraph $\mathcal{T}'_P$ of $\mathcal{T}_P$ that consists of all embracing nodes in $\mathcal{T}_P$. Since the cells at the leaves of $\mathcal{T}'_P$ are disjoint, $\mathcal{T}'_P$ has $O(tna\_disj(\mathcal{T}_P, Q))$ leaves and as many nodes of degree two. The number of subtrees of $\mathcal{T}'$ whose root is an embracing $(c-1)$-node is thus $O(tna\_disj(\mathcal{T}_P, Q))$. Together with Lemma 6.14 this implies that the total size of the subtrees we are considering is $O(tna\_disj(\mathcal{T}_P, Q) \log^{c-1} n)$. Adding up the bounds proves the lemma. $\square$

Lemma 6.16 *The total number of embracing nodes in 0-trees visited by a* DOP*-query Q is $O(\sigma^{1-1/c} \cdot depth(\mathcal{T}_P) \cdot \log^{c-1} n + k)$.*

Proof: The number of embracing nodes in 0-trees that are also corner nodes is bounded to $O(\sigma^{1-1/c} \cdot depth(\mathcal{T}_P) \cdot \log^{c-1} n + k)$ by Lemma 6.12. It remains to bound the number of embracing nodes in 0-trees that do not contain any vertex of Q. Let ν be such a node, let e be an edge of Q that intersects $bdop(\nu)$, and let s be the defining segment of ν that is parallel to e. From here we can follow the proof on the number of stabbing nodes, Lemma 6.11, and find that there are only $O(k)$ such nodes. $\square$

The following theorem, which follows from the preceding lemmas, summarizes the performance of our DOP-tree construction algorithm.

Theorem 6.17 *Let S be a set of n c-DOPs in the plane such that no point is contained in more than σ DOPs from S, and let $\mathcal{T}_P$ be a c-oriented BSP on the set of representative points of S. Then there is a BVH $\mathcal{T}_S$ on S such that:*

(i) a point query in $\mathcal{T}_S$ visits $O(\sigma^{1-1/c} \cdot depth(\mathcal{T}_P) \cdot \log^{c-1} n + k)$ nodes;

(ii) a DOP-query in $\mathcal{T}_S$ with a DOP Q visits
$O(tna(\mathcal{T}_P, Q) + (tna\_disj(\mathcal{T}_P, Q) + \sigma^{1-1/c} \cdot depth(\mathcal{T}_P)) \cdot \log^{c-1} n + k)$
nodes,

where:

- *k is the number of DOPs in S that intersect Q;*

- *$tna(\mathcal{T}_P, Q)$ is the number of cells in $\mathcal{T}_P$ that intersect at least two non-adjacent[1] edges of Q ;*

- *$tna\_disj(\mathcal{T}_P, Q)$ is the maximum size of a set of such cells that are pairwise disjoint.*

This BVH can, given $\mathcal{T}_P$, be constructed in $O(n \cdot depth(\mathcal{T}_P))$ time.

Proof: The point query time is given by Lemma 6.10. For the DOP-query time, add up the bounds for the number of visited nodes in the c-tree (Corollary 6.9), for inner nodes, side nodes, and stabbing nodes (Lemma 6.11), for embracing nodes (Lemma 6.15 and Lemma 6.16), and for corner nodes (Lemma 6.12). Finally, the bound on the construction time is proven in Lemma 6.7.   □

6.2.4   Instantiations

In Section 6.2.1 we explained how a BVH for n c-DOPs in the plane can be constructed from a BSP on a set of n points P. The bounds on the query time of the BVH depend on the chosen BSP. In this section we give bounds for BVH's based on standard kd-trees, kd-trees with longest sides cut first [36], our c-grid BSP's from Section 2.2, and BAR-trees [37].

Theorem 6.17 gives bounds on the query time of the BVH that depend on properties of the BSP $\mathcal{T}_P$, more specifically on $tna(\mathcal{T}_P, Q)$, $tna\_disj(\mathcal{T}_P, Q)$, and $depth(\mathcal{T}_P)$.

Standard kd-trees   A kd-tree on n points has $depth(\mathcal{T}_P) = O(\log n)$, $tna(\mathcal{T}_P, Q) = O(\sqrt{n})$, and, trivially, $tna\_disj(\mathcal{T}_P, Q) \leqslant tna(\mathcal{T}_P, Q) = O(\sqrt{n})$.

Corollary 6.18 *Let S be a set of n axis-parallel rectangles in the plane such that no point is contained in more than σ rectangles from S. Then there is a BVH $\mathcal{T}_S$ on S such that:*

---

[1]Recall that in our definition of adjacency the outward normal of the two edges must be adjacent in the cyclic ordering of all $2c$ possible outward normals

(i) *a point query in $\mathcal{T}_S$ visits $O(\sqrt{\sigma}\log^2 n + k)$ nodes;*

(ii) *an axis-parallel rectangle query Q in $\mathcal{T}_S$ visits $O(\sqrt{n}\log n + \sqrt{\sigma}\log^2 n + k)$ nodes,*

where k is the number of DOPs in S that intersect Q. Such a tree $\mathcal{T}_S$ can be constructed in $O(n \log n)$ time.

The BVH constructed is in fact a slightly simplified version of the kd-interval-tree of Agarwal et al. [2] (the original kd-interval-tree has more priority leaves). We obtain the same bounds as in their paper. In his thesis [49] Haverkort improved the analysis by exploiting the fact that kd-trees are well-balanced and that the lowest levels of the tree cannot contain roots of big 0-trees. This refined analysis also applies in our case. The bounds are then $O(\sqrt{\sigma}\log^2(n/\sigma) + k)$ for point queries and $O(\sqrt{n} + \sqrt{\sigma}\log^2(n/\sigma) + k) = O(\sqrt{n} + k)$ for rectangle queries.

kd-trees built with longest sides cut first   We also have depth$(\mathcal{T}_P) = O(\log n)$. Dickerson et al. [36] give an upper bound of $O(\alpha \log n)$ on the number of disjoint cells intersected by a query range with aspect ratio α. Clearly this is also an upper bound on $tna\_disj(\mathcal{T}_P, Q)$. Furthermore, we observe that

$$tna(\mathcal{T}_P, Q) \leqslant tna\_disj(\mathcal{T}_P, Q) \cdot \text{depth}(\mathcal{T}_P) = O(\alpha \log^2 n).$$

Corollary 6.19 *Let S be a set of n axis-parallel rectangles in the plane such that no point is contained in more than σ rectangles from S. Then there is a BVH $\mathcal{T}_S$ on S such that:*

(i) *a point query in $\mathcal{T}_S$ visits $O(\sqrt{\sigma}\log^2 n + k)$ nodes;*

(ii) *a rectangle query in $\mathcal{T}_S$ with an axis-parallel rectangle Q with aspect ratio α visits $O((\alpha + \sqrt{\sigma})\log^2 n + k)$ nodes,*

where k is the number of DOPs in S that intersect Q. Such a tree $\mathcal{T}_S$ can be constructed in $O(n \log n)$ time.

The BVH constructed is in fact a slightly simplified version of the lsf-interval-tree of Agarwal et al. [2] and we obtain the same bounds as in their paper. Again, a refined analysis by Haverkort [49] applies such that a point query visits $O(\sqrt{\sigma}\log^2(n/\sigma) + k)$ nodes and a range query with a box of aspect ratio α visits $O(\alpha \log^2 n + \sqrt{\sigma}\log^2(n/\sigma) + k)$ nodes.

c-grid BSPs   We have depth$(\mathcal{T}_P) = O(\log n)$, for any $\varepsilon > 0$ we have by Theorem 2.2 and its proof $tna(\mathcal{T}_P, Q) = O(n^{1/2+\varepsilon})$, and, trivially,

$$tna\_disj(\mathcal{T}_P, Q) \leqslant tna(\mathcal{T}_P, Q) = O(n^{1/2+\varepsilon}).$$

Corollary 6.20 *Let S be a set of n c-DOPs in the plane such that no point is contained in more than σ DOPs from S. Then for any $\varepsilon > 0$ there is a BVH $\mathcal{T}_S$ on S such that:*

(i) a point query in $\mathcal{T}_S$ visits $O(\sigma^{1-1/c} \log^c n + k)$ nodes;

(ii) a DOP-query in $\mathcal{T}_S$ with a c-DOP Q visits $O(n^{1/2+\varepsilon} + \sigma^{1-1/c} \log^c n + k)$ nodes,

where k is the number of DOPs in S that intersect Q. Such a tree $\mathcal{T}_S$ can be constructed in $O(n \log n)$ time.

Approximate range searching (BAR-trees). One hopes to achieve a polylogarithmic DOP-query time, but unfortunately this is impossible for exact queries. However, such results can be achieved if one is willing to settle for ε-*approximate range searching*, as introduced by Arya and Mount [13]. From Section 1.1.2 recall that 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. One BSP with a good query bound for approximate range searching is the BAR-tree [37]. A BAR-tree has $O(\log n)$ depth and can answer any convex approximate query in $\mathbb{R}^2$ in $O(\varepsilon^{-1} + k_\varepsilon + \log n)$ time where k_ε is the number of DOPs in Q_ε. The query algorithm can easily be adapted such that only objects that actually intersect Q are reported; the number of objects intersecting Q_ε is then only used for the analysis. The BAR-tree can thus answer any exact convex query in $\mathbb{R}^2$ in $O(\min_{\epsilon>0}\{\varepsilon^{-1} + k_\varepsilon\} + \log n)$ time [50].

Theorem 6.17 cannot be applied to BAR-trees directly, because a BAR-tree does not give non-trivial bounds on $tna(\mathcal{T}_P, Q)$ and $tna\_disj(\mathcal{T}_P, Q)$. However, we can apply the DOP-tree construction algorithm of Section 6.2.1 without modifications, and with a few small changes in the analysis we can still derive non-trivial bounds on the query time of the resulting DOP-tree.

The necessary changes are the following. For the analysis of the point and DOP-query time we still classify visited nodes as inner, side, stabbing, embracing or corner nodes, but inner nodes are now defined as nodes ν with $bdop(\nu)$ completely contained in Q_ε (rather than Q). Side, stabbing, embracing and corner nodes are now defined as nodes that fit the original definitions (with respect to Q) and are not completely contained in Q_ε. For instance, a side node is a node ν such that $bdop(\nu)$ intersects only one edge of Q and is not completely contained in Q_ε. It is easy to verify that the full analysis leading to Theorem 6.17 still goes through, with k replaced by k_ε, except the proof of Lemma 6.15. There we need to replace $tna(\mathcal{T}_P, Q)$ by $tna_\varepsilon(\mathcal{T}_P, Q)$: the number of cells in $\mathcal{T}_P$ that intersect at least two non-adjacent edges of any query range Q and are not completely contained in Q_ε. Furthermore, we cannot argue anymore that an embracing $(c-1)$-node with a c-node as parent must have two embracing 'siblings' in

the BSP, and so the number of such nodes can only be bounded to $tna_\varepsilon(\mathcal{T}_P, Q)$ instead of $tna\_disj(\mathcal{T}_P, Q)$.

We can thus apply Theorem 6.17 with k replaced by k_ε, and $tna(\mathcal{T}_P, Q)$ and $tna\_disj(\mathcal{T}_P, Q)$ both replaced by $tna_\varepsilon(\mathcal{T}_P, Q)$. Duncan [37] defined a *corner-cut* BAR-tree, which uses four evenly spaced cutting directions. He proved that for a corner-cut BAR-tree $\mathcal{T}_P$ we have depth$(\mathcal{T}_P) = O(\log n)$, and that for convex query ranges Q, there are only $O(1/\varepsilon + \log n)$ nodes in $\mathcal{T}_P$ whose region intersect Q but do not lie completely inside Q_ε. Clearly this is also an upper bound on $tna_\varepsilon(\mathcal{T}_P, Q)$. We obtain the following result:

Corollary 6.21 *Let S be a set of n 4-DOPs in the plane, where the set of orientations $\mathcal{C}$ is a set of 4 evenly spaced orientations, and no point is contained in more than σ DOPs from S. Then there is a BVH $\mathcal{T}_S$ on S such that:*

   (i) *a point query in $\mathcal{T}_S$ visits $O(\sigma^{3/4} \log^4 n + k)$ nodes;*

   (ii) *an exact DOP-query in $\mathcal{T}_S$ with a DOP Q visits*
        $O(\sigma^{3/4} \log^4 n + \min_{\epsilon > 0}\{\varepsilon^{-1} \log^3 n + k_\varepsilon\})$ *nodes,*

where k_ε is the number of DOPs in S with at least one point at distance at most $\epsilon \cdot diam(Q)$ from Q, where $diam(Q)$ is the diameter of Q. Such a tree $\mathcal{T}_S$ can be constructed in $O(n \log n)$ time.

Remark 6.22 The results in Corollary 6.21 hold for 4-DOPs for which the set of orientations $\mathcal{C}$ is a set of 4 evenly spaced orientations. Since boxes are 4-DOPs we can use the same argument as in Theorem 5.2.13 of Haverkort's thesis [49] to obtain the following result for constant-complexity query ranges of arbitrary shape.

Let S be a set of n 4-DOPs in the plane as in Corollary 6.21. Then there is a BVH $\mathcal{T}_S$ on S such that a constant-complexity query range Q in $\mathcal{T}_S$ visits $O(\min_{\epsilon > 0}\{\varepsilon^{-1} \sigma^{3/4} \log^4 n + k_\varepsilon\})$ nodes.

6.2.5　The number of DOPs in a 0-tree

In Lemma 6.10 we used that only $O(\sigma)$ DOPs can end up in any single 0-tree, where σ is the stabbing number of the input set. In this section we prove this. Recall that all DOPs ending up in the 0-tree rooted at some node ν intersect all defining segments of ν. Thus we can bound the number of DOPs in any single 0-tree by solving the following combinatorial-geometry problem: what is the maximum size of any set $\mathcal{D}$ of DOPs in the plane with the following properties:

(P1) There is a set L of c lines such that every edge of any DOP in $\mathcal{D}$ is parallel to some line in L;

(P2) (the interior of) each DOP in $\mathcal{D}$ intersects every line in L;

(P3) no point in the plane lies in the interior of more than σ DOPs from $\mathcal{D}$.

For a given set L, we call a DOP *admissible* if it has all edges parallel to lines in L and intersects every line in L. We start with a simple lemma.

Lemma 6.23 *The interiors of any two DOPs in a $\mathcal{D}$ intersect each other.*

Proof: Suppose two DOPs in a $\mathcal{D}$ are disjoint. Then they can be separated by a line ℓ parallel to an edge of one of the DOPs and, hence, by a line parallel to a line in L. However, this contradicts that every DOP intersects all lines in L. □

Corollary 6.24 *If the DOPs in the input set $\mathcal{D}$ are interior-disjoint, then $|\mathcal{D}| = 1$.*

We continue with a simple proof for a special, but interesting case.

Lemma 6.25 *Let $\mathcal{D}$ be a set of DOPs that are bounding DOPs of some underlying set of disjoint objects and satisfying properties (P1)–(P3). Then $|\mathcal{D}| \leqslant 4c\sigma + 1$.*

Proof: It follows from Lemma 6.23 that any two DOPs in $\mathcal{D}$ intersect. Furthermore, for any two intersecting DOPs $D, D' \in \mathcal{D}$ there must be a vertex from D inside D', or vice versa. This follows since D and D' are bounding DOPs of disjoint objects. We charge the intersection between D and D' to this vertex. By property (P3), any vertex can be charged at most σ times. Since a DOP has at most $2c$ vertices we can have at most $2c|\mathcal{D}|\sigma$ intersections, otherwise a vertex would be charged too often. On the other hand, any two DOPs in $\mathcal{D}$ intersect, so there are $\binom{|\mathcal{D}|}{2} = |\mathcal{D}|(|\mathcal{D}|-1)/2$ pairwise intersections. Hence, $|\mathcal{D}|(|\mathcal{D}|-1)/2 \leqslant 2c|\mathcal{D}|\sigma$, which implies $|\mathcal{D}| \leqslant 4c\sigma + 1$. □

Bounding the size of $\mathcal{D}$ for the general case, where the DOPs in $\mathcal{D}$ can intersect in an arbitrary manner, is a lot more difficult. We can show that in the worst case $|\mathcal{D}| = \Omega(c\sigma)$, but we have not been able to prove a matching upper bound for variable c. Nevertheless we can prove a bound that is linear in σ.

Theorem 6.26 *Let $\mathcal{D}$ be a set of DOPs satisfying properties (P1)–(P3). Then $|\mathcal{D}|$ is $\Omega(c\sigma)$ and $O(c^4\sigma)$ in the worst case.*

It follows that for constant c, the number of c-DOPs in any single 0-tree is $O(\sigma)$. In the remainder of this section we first prove the lower bound of this theorem, and then the upper bound.

Lemma 6.27 *Let $\mathcal{D}$ be a set of DOPs satisfying properties (P1)–(P3). Then $|\mathcal{D}|$ is $\Omega(c\sigma)$ in the worst case.*

Proof: We construct a set $\mathcal{D}$ of size at least $c\lfloor \sigma/2 \rfloor$ as follows. Let L be a set of c lines in general position. From every line of L, take a segment that intersects all other lines from L and inflate it to a narrow DOP that contains those intersections in its interior, see Figure 6.8. If the DOPs are narrow enough, each intersection among lines in L now lies in exactly two DOPs (those constructed from the lines intersecting in that point) and no point lies in more than two DOPs. To get a set
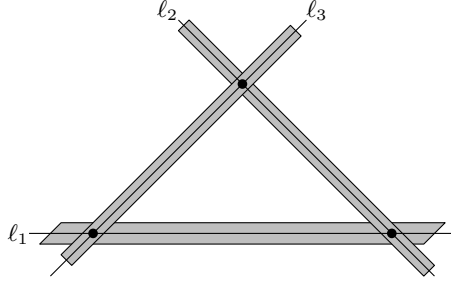
Figure 6.8: A lower bound example for $c = 3$. On every line there are $\lfloor \sigma/2 \rfloor$ DOPs. Every intersection point between two lines are thus covered by σ DOPs.

$\mathcal{D}$ of size $c\lfloor \sigma/2 \rfloor$, we put $\lfloor \sigma/2 \rfloor$ copies of each of these c DOPs in $\mathcal{D}$. It is easily verified that this set satisfies properties (P1)–(P3).   □

We now prove the upper bound. We need the following observation.

Observation 6.28 Suppose all lines in $\mathcal{C}$ intersect in a single point p. Any DOP D that does not contain p, must be separated from it by a line ℓ' parallel to a line ℓ_i in $\mathcal{C}$. Then ℓ' also separates D from ℓ_i, and therefore D is inadmissible. Hence, if all lines in $\mathcal{C}$ intersect a single point p—which is always the case if $c = 2$—every admissible DOP must contain p, and there can be only σ such DOPs in S.

For the case that the lines in L do not have a common intersection, we prove Theorem 6.26 as follows. We first prove that for each DOP $D \in \mathcal{D}$ there is a cell in the arrangement $\mathcal{A}_L$ induced by the lines in L such that D intersects at least three edges of that cell. It follows that it intersects three edges of a cell in the arrangement $\mathcal{A}'$ of the three lines from L that contain those edges of $\mathcal{A}_L$. We then prove that for any such arrangement $\mathcal{A}'$, there is a set of at most $4c+1$ points—we call them *guards*—such that if a DOP intersects all edges of any three-edge cell in $\mathcal{A}'$, it must contain at least one guard. It follows that the total number of DOPs in $\mathcal{D}$ cannot exceed $\binom{c}{3}(4c+1)\sigma = O(c^4\sigma)$, thus proving the theorem.

Lemma 6.29 *If there is no single point where all lines in L intersect then for any $D \in \mathcal{D}$ there is a, possibly unbounded, cell in the arrangement $\mathcal{A}_L$ such that D intersects at least three edges of that cell.*

Proof: Suppose for a contradiction that D intersects at most two edges of any cell in $\mathcal{A}_L$. We distinguish two cases.

Case (i): D contains an intersection point p of two or more lines in L. Now D intersects two edges of every cell in $\mathcal{A}_L$ incident to p. If D does not intersect a third edge of at least one of those cells then, since not all lines intersect in p, some line in L does not intersect D, contradicting (P2).
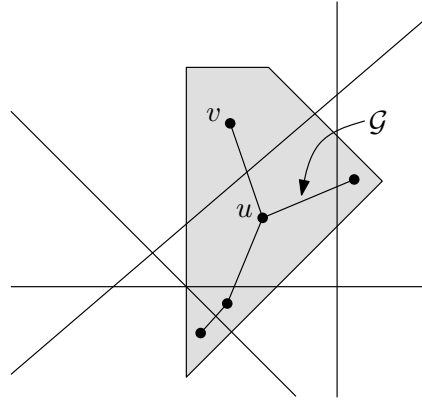
Figure 6.9: The nodes in the graph $\mathcal{G}$ represent the cells intersected by D. There is an edge between two nodes if the edge between the corresponding cells is intersected by D.

Case (ii): D does not contain an intersection point p of two or more lines in L. Consider the graph $\mathcal{G}$ whose nodes represent the cells in $\mathcal{A}_L$ intersected by D, and with an arc (u, v) between two nodes u and v if and only if D intersects the edge in $\mathcal{A}_L$ between the cells C_u and C_v represented by u and v, see Figure 6.9.

The degree of any node u in $\mathcal{G}$ is the number of edges of C_u in $\mathcal{A}_L$ that are intersected by D. If D intersects at most two edges of any cell in $\mathcal{A}_L$, the graph $\mathcal{G}$ must be a path. (Since D is convex and does not contain a vertex of $\mathcal{A}_L$, it cannot be a cycle.) If this path would contain less than c arcs, D would intersect less than c lines from L, contradicting (P2). So we may assume that the path contains c arcs and $c + 1$ nodes.

Let p and q be points of D in the cells at the ends of this path. Without loss of generality, assume that the line segment pq is vertical and oriented upwards. Let $C_0, ..., C_c$ be the cells in $\mathcal{A}_L$ represented by the nodes on this path, so that p lies in C_0 and q lies in C_c, and let $\ell_{\pi(i)}$ be the line containing the edge between C_{i-1} and C_i, where π is a permutation of $1 \ldots c$ which defines the order of the orientations of the c intersected lines.

Let $H_{-1}, ..., H_{-c}, H_1, ..., H_c$ be the halfplanes such that $H_{-1} \cap ... \cap H_{-c} \cap H_1 \cap ... \cap H_c = D$ and each H_i is bounded by a line h_i parallel to $\ell_{\pi(|i|)}$ that touches D. Let e_i be the edge of D that is defined by H_i, that is, the intersection of h_i with the boundary of D (some of these 'edges' may in fact be vertices of D). Without loss of generality, assume that the edges appear in the order $e_1, ..., e_c, e_{-1}, ..., e_{-c}$ on a clockwise walk along the boundary of D. Denote this set of $2c$ edges by E.

Note that for any $i \in \{0, ..., c\}$ and $j \in \{1, ..., c\}$, the edges e_{-j} and e_j cannot both intersect C_i (or its boundary) since then $\ell_{\pi(|j|)}$, which lies between e_{-j} and
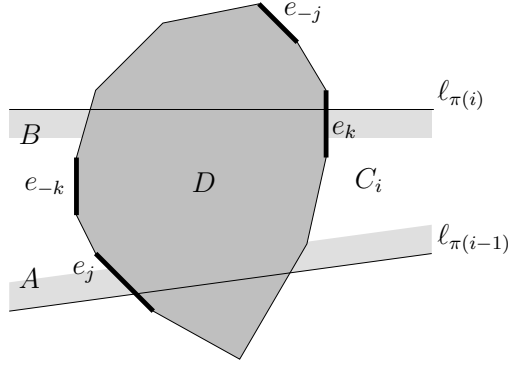
Figure 6.10: A cell C_i such that the halfplane A, bounded from below by $\ell_{\pi(i-1)}$, and the halfplane B, bounded from above by $\ell_{\pi(i)}$, are each intersected by more than half of the edges of D. This leads to a contradiction, as $\ell_{\pi(k)}$ passes between e_{-k} and e_k and therefore it must intersect C_i.

e_j, would pass through C_i, contradicting the fact that C_i is a single cell of $\mathcal{A}_L$. It follows that at most half of the edges in E intersect the closed halfplane bounded from above by $\ell_{\pi(1)}$ (and thus, C_0). Symmetrically, at most half of the edges in E intersect the closed halfplane bounded from below by $\ell_{\pi(c)}$ (and thus, C_c). As a consequence there is an $i \in \{1, ..., c-1\}$ such that more than half of the edges in E intersect the closed halfplane A bounded from below by $\ell_{\pi(i-1)}$ and more than half of the edges in E intersect the closed halfplane B bounded from above by $\ell_{\pi(i)}$—see Figure 6.10.

Consider the edge e_j in E that intersects A such that its predecessor in the clockwise ordering of edges does not intersect A. Because more than half of the edges in E intersect A, edge e_{-j} intersects A. But since e_j and e_{-j} cannot both intersect C_i, edge e_{-j} lies above $\ell_{\pi(i)}$. Next consider the edge e_k in E that intersects B such that its predecessor in the clockwise ordering of edges does not intersect B. Observe that e_k lies on the path clockwise along the boundary of D from e_{-j} to e_j. Now e_{-k} must intersect B (because more than half of the edges intersect B) and A (because it lies clockwise between e_j and e_{-j}). Hence it intersects C_i, as does e_k, which contradicts the observation that e_k and e_{-k} cannot both intersect C_i.

In both cases there cannot be an admissible DOP D which intersects at most two edges of any cell in $\mathcal{A}_L$. So there has to be a cell in $\mathcal{A}_L$ of which at least three edges are intersected by D. □

We now prove that a bounded cell in an arrangement of three lines can be guarded by $c + 1$ guard points such that when a DOP intersects all edges of this cell it must contain a guard. After that we show how to reduce the case of an unbounded three-edge cell to the case of a bounded cell.

Lemma 6.30 *For any arrangement $\mathcal{A}'$ of three lines from L, there is a set of at most $c + 1$ guard points, such that if a DOP intersects all edges of the bounded three-edge cell in $\mathcal{A}'$, it must contain at least one guard.*

Proof: Let $D = H_{-1} \cap ... \cap H_{-c} \cap H_1 \cap ... \cap H_c$ be a DOP, where the boundary lines of the halfplanes h_{-i} and h_i are parallel to the line $\ell_i \in L$. Assume D intersects all three edges of the triangle $\triangle$ that is the bounded face of $\mathcal{A}'$.

Without loss of generality, let ℓ_1, ℓ_2 and ℓ_3 be the lines that define the bounded cell $\triangle$, such that s_1, s_2 and s_3 are the sides of $\triangle$ that lie on ℓ_1, ℓ_2 and ℓ_3, respectively, in counterclockwise order around $\triangle$. For $i \in \{1, 2, 3\}$, let v_i be the vertex of $\triangle$ which is not on ℓ_i. Without loss of generality, let ℓ_1 be horizontal and bound $\triangle$ from below—see Figure 6.11 a).

We define for every line $\ell_i \in L$ a set of vertices V_i. For $i \in \{1, 2, 3\}$, we define $V_i = \{v_1, v_2, v_3\} \setminus \{v_i\}$, and for $i \in \{4, ..., c\}$, the set V_i consists of the unique vertex of $\triangle$ such that a line through that vertex and parallel to ℓ_i intersects the interior of $\triangle$. The set of guards is now defined as follows. For every line $\ell_i \in L$ we define ℓ_i' to be a line through the vertices in V_i, and parallel to ℓ_i. We place a guard g_i at the midpoint of the segment defined by $\ell_i' \cap \triangle$. We also place a guard g_0 at the center point of D.

Note that g_1, g_2 and g_3 form a triangle $\triangle'$ which is isomorphic to $\triangle$ and on whose edges the guards g_i for $i \in \{4, ..., c\}$ are placed. For $i \in \{1, 2, 3\}$, let s_i' be the edge of $\triangle'$ which is parallel to ℓ_i. Observe that D must intersect at least two edges of $\triangle'$.

We prove that D contains a guard by deriving a contradiction from the assumption that it does not. Let e_i be the edge of D that is defined by H_i.

If $\triangle'$ would have an edge s_i' that lies completely inside D or is intersected by only one edge of D, it would follow that one of the endpoints of s_i' lies in D. Since both endpoints are guards, this would contradict our assumptions. So any edge s_i' from $\triangle'$ is intersected either by no edge of D, or by two edges of D. Without loss of generality, assume that s_1' is one of the edges of $\triangle'$ that intersects two edges of D. Let e_j be the left edge (that is: the one whose intersection with s_1' is closest to g_3), and let e_k be the right edge. For readability we write V_j and g_j instead of $V_{|j|}$ and $g_{|j|}$.

We distinguish the following six cases (some of them overlap).

1. $v_1 \in V_j$ and $v_1 \in V_k$ (see Figure 6.11 b)).

   Note that g_j and g_k both lie on s_1'. The guard g_j must lie to the right of the intersection of s_1' with e_j since otherwise s_3 would be separated from D by the line that contains e_j. By the assumption that D does not contain a guard, e_k must intersect s_1' to the left of g_j. Hence, H_k, and thus D, is contained in the halfplane H_k' that lies left of the line through g_j and parallel to ℓ_k. By a similar argument, we find that D is contained in the halfplane H_j' that lies right of the line through g_k and parallel to ℓ_j. Let p be the bottommost point of the intersection $H_j' \cap H_k'$ of those halfplanes. Observe that the triangle $g_k p g_j$ is congruent with $g_j v_1 g_k$—in particular, both triangles are exactly
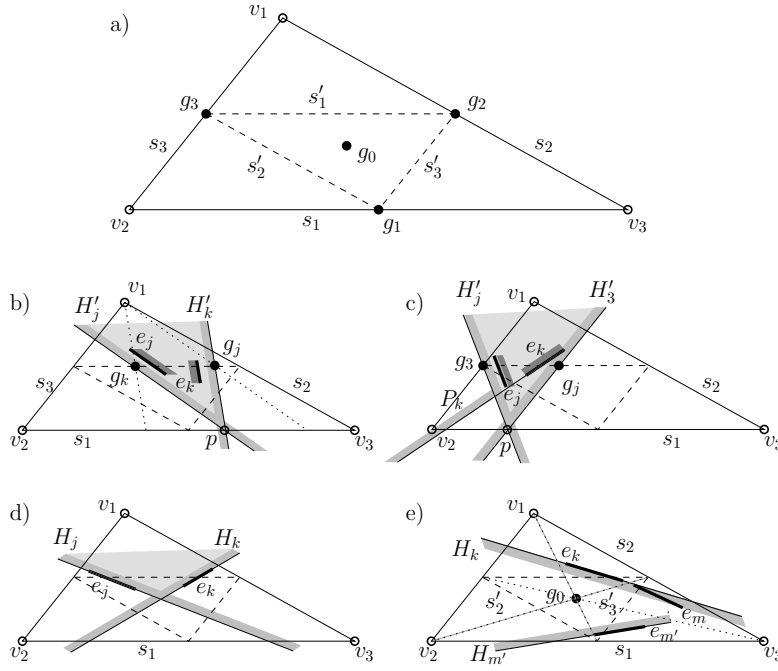
Figure 6.11: Illustrating the case analysis for Lemma 6.30

half as high as $\triangle$. The point p must therefore lie on s_1, so $H'_j \cap H'_k \supset D$ lies completely above ℓ_1, which contradicts our assumptions.

2. $v_1 \in V_j$ and $v_2 \in V_k$ (see Figure 6.11 c)).

   The proof for this case is very similar to that of the previous case. Again, H_k, and thus D must be contained in the halfplane H'_k that lies left of the line through g_j and parallel to ℓ_k. Since $V_k = \{v_2\}$, the part of H'_k that intersects ℓ_1 is in fact contained in the halfplane H'_3 that lies left of the line through g_j and parallel to ℓ_3. From here we can follow the same argument as for the previous case, with k replaced by 3.

3. $v_3 \in V_j$ and $v_1 \in V_k$.

   This case is symmetric to the previous case.

4. $v_3 \in V_j$ and $v_2 \in V_k$ (see Figure 6.11 d)).

   Clearly, the bottommost point of the intersection of H_j and H_k must lie in $\triangle'$, so that D must lie completely above ℓ_1, which contradicts our assumptions.

5. $v_3 \in V_k$ (see Figure 6.11 e)).

   Since H_k separates D from the upper half of s_2, the DOP D must intersect the lower half of s_2, and therefore also s_3'. Let e_m be the first edge of D that intersects s_3' when we walk along the boundary of D in clockwise order from e_k. Note that V_m cannot contain v_1, since H_m would then separate D from the lower half of s_2. Furthermore, V_m cannot contain v_2, since that would imply that D has a reflex vertex on the clockwise walk from e_k to e_m. So $V_m = \{v_3\}$.

   Now s_3' must be intersected by a second edge $e_{m'}$ of D. If $V_{m'}$ contains v_3 or v_1, we can treat this case as case 1 or 2, respectively, on s_3' instead of s_1'. Otherwise, that is, if $V_{m'} = \{v_2\}$, we repeat the above argument on s_3' instead of s_1', and find that D intersects not only s_1' and s_3', but also s_2'.

   Since by assumption, D does not contain the center g_0 of $\triangle$ (and $\triangle'$), there must be a line through g_0 such that D lies completely on one side of that line. Clearly, any such line separates D from one of the six edges of $\triangle$ and $\triangle'$, contradicting our assumptions.

6. $v_2 \in V_j$.

   This case is symmetric to the previous case.

Since the above covers all cases and each of them leads to a contradiction, we must conclude that D contains a guard. $\square$

   Finally we show how to place guards in an unbounded three-edge cell such that when a DOP intersects all three edges it must contain a guard.

Lemma 6.31 *Let $\mathcal{A}'$ be an arrangement of three lines from L. An unbounded three-edge cell C in $\mathcal{A}'$ can be guarded by $c + 1$ points such that if any DOP D intersects all edges of C, then D contains a guard.*

Proof: Let $\triangle$ be such an unbounded cell, and let v and w be its vertices. Draw a line ℓ_v through v with one of the given c orientations such that ℓ intersects the unbounded edge of $\triangle$ incident to w at a point x that is as far away from w as possible—see Figure 6.12. From the choice of ℓ_v it follows that D must intersect the segment wx. If x coincides with w, it follows that D contains $x = w$ and one
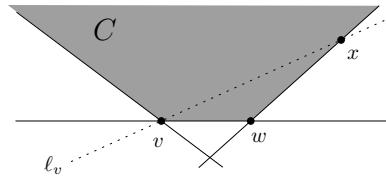
Figure 6.12: Reducing the case of an unbounded cell to that of a bounded cell.

guard at $x = w$ suffices. Otherwise, note that D must also intersect vx since it
intersects the unbounded edge of $\triangle$ incident to v. Hence, D intersects all three
sides of the triangle vwx. This triangle can be guarded with $c + 1$ points, as we
have seen before.   □

Note that for every unbounded three-edge cell in $\mathcal{A}'$ one of the guards, the one
in the middle of vw, was already added for the bounded cell of $\mathcal{A}'$, so we need c
extra guards for each of the three unbounded three-edge cells. The total number
of guards for $\mathcal{A}'$ is thus $4c + 1$.

Combining Observation 6.28 and the Lemma's 6.29, 6.30 and 6.31, we find
that the total number of admissible DOPs cannot exceed $\binom{c}{3}(4c+1)\sigma$, thus proving
the upper bound of Theorem 6.26.

Although this amounts to bounds of 13σ for $c = 3$ and 68σ for $c = 4$, a
careful look at these simple cases would reveal that many guards coincide or are
redundant. For $c = 3$ we would find that 3 guards suffice, and for $c = 4$ we can
do with 12 guards, see Figure 6.13.
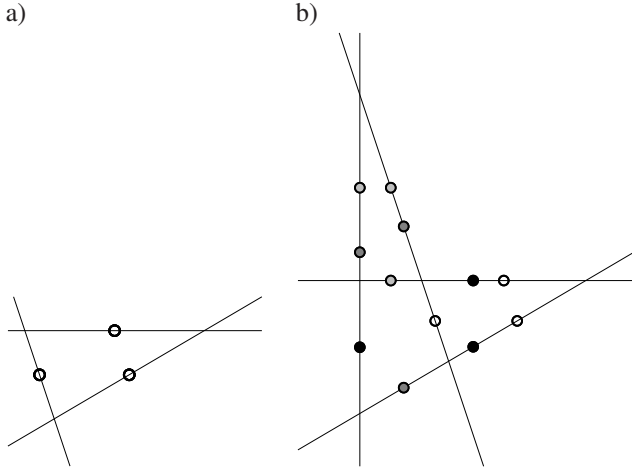
Figure 6.13: a) Guards for $c = 3$ b) Guards for $c = 4$

Chapter 7

Experimental Results on External-Memory DOP-trees

In Chapter 6 we introduced a framework for constructing a bounding-volume hierarchy based on a c-oriented BSP[1] for a set of objects in $\mathbb{R}^2$ that fits in internal memory. We called the resulting BVH a DOP-tree. It uses a *c-discretely oriented polygon*, or c-DOP as a bounding volume. We gave a theoretical bound on the query time using the query time in the BSP and the stabbing number of the bounding DOPs of the input set. Recall from Section 1.2.2 that the stabbing number of a scene is the maximum number of objects intersecting in any point in the plane. It is believed that many realistic scenes have a low stabbing number.

In this chapter we experiment with queries in DOP-trees on very large input sets. The DOP-trees do not fit in internal memory and are therefore stored on disk.

In the first section we describe an external-memory variant of the DOP-trees. In Section 7.2 we address the experimental setup for the three sets of experiments. The results of these experiments are given in Section 7.3. In the first experiment we consider the effect on the query cost of the number of items that can be stored in a single block. The experiment shows that when the block size is increased by some factor f the query cost is decreased by a factor less than f. In the second set of experiments we investigate the effect of increasing the number of orientations used for describing the bounding DOP on the query cost. In some experiments using more orientations decreases the cost of a range query in a set of objects, in other experiments it does not help to use more orientations. Furthermore we show that the leaf blocks have the largest influence on the total cost of a query. In the third and final experiment we compare the DOP-trees to the PR-tree by Arge et al. [11].

The O-tree [80] is an external-memory BVH which uses an axis-aligned bounding box and a bounding box whose sides are parallel to the diagonals. The O-tree

---

[1]A BSP is c-oriented when all splitting lines stored at the nodes of $\mathcal{T}$ have an orientation in a fixed set of c orientations.

thus basically uses 4-DOPs as bounding volume. As far as we know the O-tree is the only external-memory BVH which uses a specific DOP as bounding volume. We do not compare our DOP-trees to the O-tree, because the O-tree is built using repeated insertions while the DOP-trees are bulk-loaded.

7.1   An external-memory DOP-tree

In this section we describe how we have constructed the external-memory DOP-trees used in our experiments. We make a distinction between blocks which only store the input objects, *content blocks*, and blocks which store only internal nodes with bounding volumes of the objects below, *navigation blocks*. A block is either a content block or a navigation block, we do not allow any other type of blocks. A content block stores at most B_c input objects and a navigation block at most B_n DOPs. During the construction the objects in the input set are approximated by a bounding c-DOP. In the content blocks, however, they are stored without the bounding volume.

In Chapter 6 an internal-memory DOP-tree is constructed by first computing a representative point for each object of the input set. On the resulting set a c-oriented BSP $\mathcal{T}_P$ is constructed. Finally the DOPs in the input set are inserted into $\mathcal{T}_P$ and $\mathcal{T}_P$ is converted into a BVH as follows. We start at the root ν of $\mathcal{T}_P$ and store the bounding DOP of the input set and for every orientation the most extreme DOP at ν. Next the input set is split in three sets, one set of DOPs which intersect the splitting line ℓ stored at ν and two sets containing DOPs completely on either side of ℓ. The latter two sets are inserted recursively. On the first set an auxiliary tree, a $(c-1)$-*tree*, is built which becomes the third child of ν.

The external-memory DOP-tree construction in this chapter differs from the construction described in Chapter 6 in the use of priority leaves. In the external-memory version of the DOP-tree we do not store the most extreme DOP for each orientation in $\mathcal{C}$ at the node ν, but we store the B most extreme DOPs for each orientation in a separate content block. We call these content blocks *priority blocks*. A further adaption to the DOP-tree construction algorithm is that we add priority leaves not only to the $(c-1)$-trees, but also to the main tree. These priority blocks were added because a preliminary experiment showed that the query cost decreases when priority blocks are also used for the main tree.

The DOP-trees are constructed I/O-efficiently using a bulk-loading algorithm based on the "grid" technique introduced by Agarwal et al. [1]. Let $\mathcal{T}$ be the c-oriented BSP we want to construct. In the grid technique $\Theta(\log(M/B))$ levels of $\mathcal{T}$ are constructed using a c-grid BSP of size $\Theta((M/B)^{c_1})$ residing in memory for some constant $c_1 \leq 1/c$. Let $\widehat{\mathcal{T}}$ be the $\Theta(\log(M/B))$ levels. Next the input set is partitioned into subsets that correspond to the subtrees below each leaf of $\widehat{\mathcal{T}}$. While partitioning the input set we also transform $\widehat{\mathcal{T}}$ into a DOP-tree for which we keep track of the B most extreme DOPs in each of the c orientations and the DOPs intersecting the splitting line at every node in $\widehat{\mathcal{T}}$. This process can be completed with a constant number of scans of the input set. Finally we recurse to build

the subtrees. The recursion stops when we have less than M DOPs to deal with, for which we just build the entire subtree in memory. The overall cost is then $O(N/B \cdot \log(N/B)/\log(M/B))$ I/Os.

During the construction a navigation block $\mathcal{B}$ has an out-degree of at most $3+2c$, that is, below $\mathcal{B}$ there are at most $2c$ priority blocks and 3 navigation blocks for the three subtrees. If there is at least one navigation block below $\mathcal{B}$ then the $2c$ priority blocks contain in total $2cB_c$ input objects. The total number of navigation blocks is thus bounded by $O(N/B_c)$. In a post-processing phase we group the navigation blocks bottom-up so that the number of navigation blocks becomes $O(N/(B_cB_n))$. The grouping is done as follows. We check every navigation block $\mathcal{B}'$ directly below some block $\mathcal{B}$. If $\mathcal{B}$ can accommodate all nodes in $\mathcal{B}'$ then we move all nodes from $\mathcal{B}'$ to $\mathcal{B}$. We continue until there is no block below $\mathcal{B}$ whose nodes can all be accommodated for by $\mathcal{B}$.

As a second postprocessing step we merge content blocks containing less than B_c input objects below one navigation block. We group the content blocks together as follows. Consider a navigation block $\mathcal{B}$. At $\mathcal{B}$ we count the number of objects in each content block below $\mathcal{B}$. We sort the content blocks by the number of objects which still fit in the block. For each non-full block we try to find a sibling content block such that after merging the two blocks the number of free places is minimized. After the second postprocessing step the navigation block can have more than one reference to a content block, but a content block is always referred to by one navigation block.

7.2 Experimental setup

In this section we describe the experimental setup of our experiments. For the DOP-tree framework, introduced in Chapter 6, we have to choose which c-oriented BSPs we would like to use and which orientations we will use. Next we describe the input sets on which the DOP-trees are built and how they are constructed. We then address how the queries were chosen to determine the cost of querying the DOP-trees. We conclude with a description of the software and hardware used in the experiments and the settings of the software.

Binary Space Partitions.   For our experiments we used two BSPs in the DOP-tree framework. The first BSP is the c-grid BSP as described in Section 2.2. The second BSP is an adaptation of the well-known LSF-kd-tree, see Section 1.3.1. The construction of the adapted LSF-kd-tree is as follows. We start with a region R which is the bounding DOP of the objects in the input set. For every orientation $c_i \in \mathcal{C}$ we calculate the minimum distance δ_i between two halfplanes that enclose R and have orientation c_i. The longest side of R is defined as an orientation c_j for which $\delta_j = \max_i \delta_i$ with ties broken arbitrarily.

The region R is then cut by a splitting line ℓ whose orientation makes the largest angle with c_j of all $c_i \in \mathcal{C}$ such that ℓ splits the point set in half. An adapted LSF-kd-tree is constructed recursively on the resulting two sets until there are B_c

points left in a region. Note that for $c = 2$ we have an ordinary LSF-kd-tree and when we use it in the framework we obtain the longest-side-first KD-interval-tree by Agarwal et al. [2, 49].

Orientations of splitting lines.   The set of orientations $\mathcal{C}$ used in the experiments are those of a regular $2c$-gon where the first orientation is always horizontal. For $c = 2$ we thus use splitting lines that are axis-aligned and for $c = 4$ we use orientations that are axis-aligned or diagonal. For the experiments we used $c \in \{2, 3, 4, 5, 6\}$.

Input objects.   Each input set contains 12,000,000 points, segments or squares all contained within the unit square. We used three distributions for the objects in the experiments, UNIFORM, DIAGONAL, CIRCLE, see Figure 7.1.

These distributions of the input are chosen in such a way that we can investigate the behavior of the DOP-trees. The UNIFORM distribution has been chosen for the case that the input is without structure. The DIAGONAL distribution represents the case that the input is distributed along a line. The line is oriented diagonally to be able to investigate the influence of the input being distributed along a line resembling one of the splitting orientations for $c = 4$. The CIRCLE distribution eliminates this possible dependency.

For the results we will use the following color-coding for the distributions:
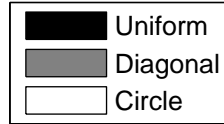
Construction of the input sets.   The points in the UNIFORM distribution are picked randomly. The DIAGONAL distribution of points consists of points picked randomly such that the distance to the diagonal is at most 0.01. The CIRCLE set contains points in a band of width 0.05 around a circle of radius 0.3. A sample of the point data sets is given in Figure 7.1 a)– c).

For the experiments with line segments we picked a point as in the construction of the point sets and then picked a segment with an arbitrary orientation going through this point. See Figure 7.1 d)– f) for an impression of the input sets. The set consists of segments of variable length. The length l of a segment is chosen arbitrarily using the following formula: $l = 2^{12*\text{rnd}-16}$ where rnd is a random real number between 0 and 1. All segments in the set are fully contained in the unit square.

The squares are constructed as follows. We start by picking the side length l arbitrarily using the following formula: $l = 2^{12*\text{rnd}-16}$ where rnd is a random real number between 0 and 1. We then picked a point p as described in the construction of the point sets. The square is then positioned such that p is the center
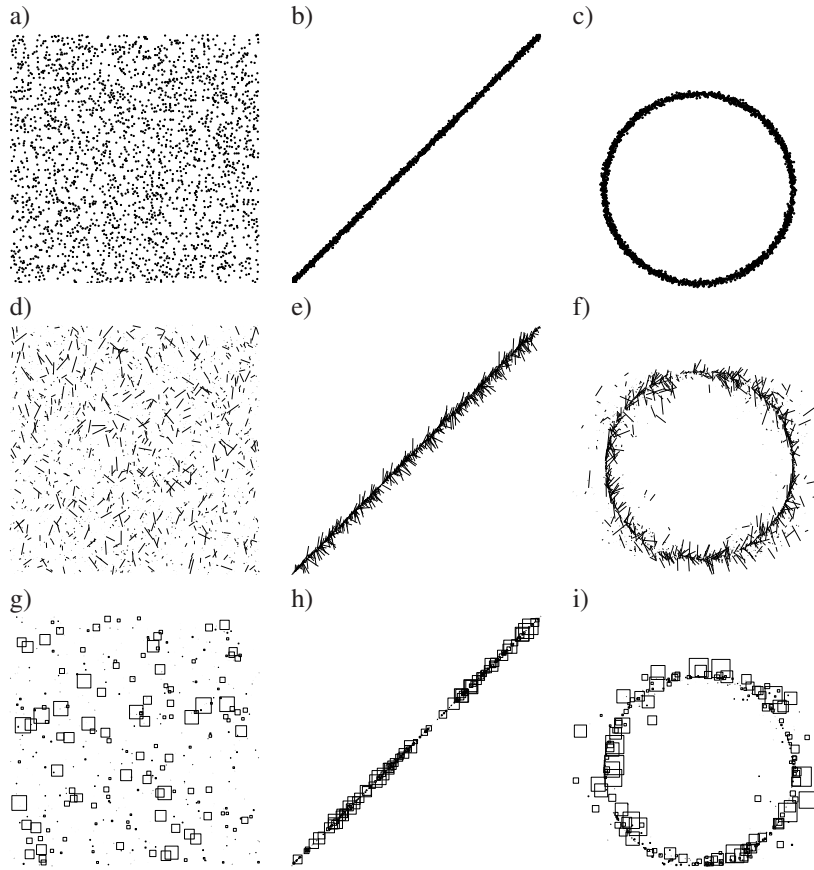
Figure 7.1: The input sets used in the experiments. a) A uniformly distributed point set. b) A point set distributed along a diagonal line. c) A point set distributed along the boundary of a circle. d) A uniformly distributed line segment set. e) and f) are sets of line segments distributed along a diagonal line respectively the boundary of a circle. g)-i) A set of boxes from left to right: uniformly distributed, distributed along a diagonal line and distributed along the boundary of a circle.

of the square. In Figure 7.1 g)– i) an impression of the resulting distribution is given. All squares are fully contained in the unit square.

Queries.   The data structures were queried with rectangle queries (or more precise square queries), since rectangle queries of some reasonable aspect ratio are the standard type of query. The queries are chosen such that they follow the underlying distribution of the objects. More precisely we choose a random object o from the input and construct a query Q such that the lower left-most point of o is the center of Q. For each distribution we created two query sets containing 100 squares. The squares in the first set have side length 0.01 and in the second set 0.1. The results in the next chapter are an average over the 100 queries in a set.

The queries were themselves not approximated by a c-DOP since the queries are simpler than the DOP that would have been obtained otherwise.

Software and hardware.   The DOP-trees are implemented in C++ using the external memory library TPIE [12]. As experimental platform we used a Dell Optiplex GX280 with one Pentium IV/3.0GHz processor running WindowsXP. A local 120 GB IDE disk was used to store all necessary files: input, DOP-trees and all temporary files.

A point is stored as two floats for its position and one unsigned integer as its identifier. Both a line segment and a box are stored using two points. A DOP also uses an unsigned integer as its identifier and $2c$ floats for the position of the halfplanes which define the DOP.

Block size.   In TPIE [12] one has to set the number of items per block manually. In our case an item is a bounding DOP or an input object. This enabled us to store less items in a block than would have fit otherwise. This is convenient for us since in practice not only the tree is stored, but also satellite data and we can now easily investigate the effect of storing satellite data in a block by considering large blocks (which can contain many items) and small blocks (which can contain fewer items because of the satellite data). Because we can use the same input set for DOP-trees with and without satellite data, we can make a fair comparison between the number of blocks accessed by a query in a DOP-tree without satellite data and with satellite data, see Section 7.3.1.

We use a PR-tree for a comparison with the DOP-trees in Section 7.3.3. The implementation of the PR-tree can handle only one size for the blocks. We set the size of the large content blocks such that our DOP-trees and the PR-tree store the same number of objects in a content block. Since the number of DOPs which can be stored in a navigation block is different for every bounding DOP, we simply set the number of DOPs in a block to the maximum possible for each c. This maximum for B_n is given by

$$\left\lfloor \frac{16384 \cdot \text{size of(word)}}{2c \cdot \text{size of(float)} + \text{size of(unsigned integer)} + \text{size of(pointer to block)}} \right\rfloor,$$

	navigation					content
	2	3	4	5	6	
Large	2340	1820	1489	1260	1092	1638
Small	585	455	372	315	273	400

Table 7.1: The number of DOPs which fit in a navigation storing c-DOPs and the number of input objects which can be stored in a content block. The size of the large content block is dictated by the number of objects which can be stored in the PR-tree while the size of the navigation block is the maximum number of DOPs which can be stored in a navigation block.

where the size of a float, a word and an unsigned integer are the same and the size of a pointer to a block is twice as large. The size of the small blocks are set to be (about) one forth of the size of a large block. The number of items which can be stored in a large (without satellite data) and a small (with satellite data) block is given in Table 7.1.

7.3 Experiments

In the first set of experiments we investigate the effect on the query time when a large part of the block is used to store satellite data. Next we investigate the effect of more tightly fitting bounding volumes for two types of DOP-trees. Finally we compare the two DOP-trees and the PR-tree.

7.3.1 Effect of block size

In experiments it is often assumed that a navigation block only stores the bounding DOP of the blocks below and references to other blocks and a content block only the objects. In practice however often additional, or *satellite*, data is stored with the bounding DOP or object, for instance the cost of an object and the maximum or total cost of all objects stored in the block below. In this experiment we investigate the effect of storing satellite data in a block. In Table 7.1 the number of bounding DOPs or objects is given which is stored in a block when satellite data is not present and when it is present. The difference in the block size has been exaggerated for this experiment.

In our experiments the number of objects and DOPs stored in a block with satellite data is roughly one forth of the number of items stored in a block without satellite data. One would expect that the average number of blocks visited by a query would be a factor of four larger when the blocks contain satellite data, because a query has to visit more blocks to report all objects in the query range.

In the experiment we used two sets of GRID-DOP-trees. The first set of GRID-DOP-trees are stored using large blocks; we denote this set of DOP-trees by S_{large}.
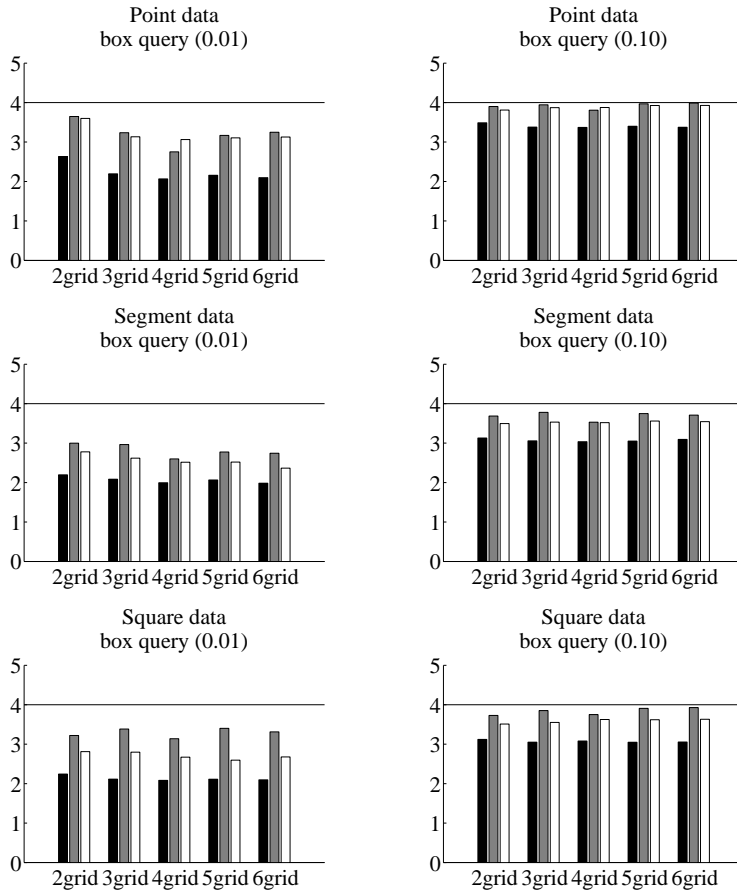
Figure 7.2: Average number of blocks accessed by a query for blocks with satellite data divided by the average number of blocks accessed without satellite data. In both cases the GRID-DOP-tree is used.

The second set of GRID-DOP-trees, S_{small}, is stored using small blocks to mimic the existence of satellite data. Both sets of GRID-DOP-trees contain a GRID-DOP-tree for each of the nine input sets described in Section 7.2. For each input set we constructed two query sets based on the input set as described in Section 7.2.

We queried the GRID-DOP-tree on an input set with the corresponding query sets. We then divide the average number of blocks accessed by a query in a tree $\mathcal{T} \in S_{small}$ by the average number of block accessed by a query in a tree $\mathcal{T}' \in S_{large}$ on the same input set. The results for the GRID-DOP-tree are shown in Figure 7.2. We did the same for LSF-DOP-trees, but the results were very similar and are therefore not shown here.

When the tree is stored in a small block the average number of accessed blocks is increased by a factor less than four. This shows that storing satellite data has less negative influence than one would expect, but also that increasing the number of items, which can be transferred between disk and memory, does not incur a decrease in the number of I/O-operations by the same fraction. One explanation is that content blocks intersecting the boundary of the query can store more objects, which do not intersect the query. This makes accessing this block relatively expensive.

For large queries the increase is closer to the expected factor than for small queries. This can be explained by the fact that more content blocks are completely contained in a large query than in a small query. For every content block in a tree $\mathcal{T} \in S_{\text{large}}$ which is completely contained in the query Q there are roughly four content blocks contained in Q in the $\mathcal{T}' \in S_{\text{small}}$ on the same input set as $\mathcal{T}$.

We observe that the factor of increase is not the same for each value of c for a specific setting. As a result the optimal number of orientations for a setting might depend on the number of items which can be stored in a block.

7.3.2 Effect of a tighter bounding DOP

In this section we investigate the effect of more orientations for the bounding DOP of the underlying objects. For this end we constructed DOP-trees based on the c-grid BSP and the adapted LSF-kd-tree. We then query these DOP-trees with 100 squares of size 0.01. In the previous experiment we made the observation that the optimal number of orientations used in the construction of the DOP-tree might be dependent on the number of items which can be stored in a block. We therefore investigate the effect of using a tighter bounding DOP for DOP-trees stored in both small and large blocks.

The number of accessed navigation and content blocks and the total of the two depends on the distribution of the input set. In order to investigate the effect of a tighter bounding volume we remove this dependency by normalizing the results. We set the outcome for the 2-GRID-DOP-tree and the 2-LSF-DOP-tree to 1. Next we discuss the results presented in Figures 7.3 – 7.8.

Accessed navigation blocks. In most cases the average number of accessed navigation blocks increases when c is increased for both classes of DOP-trees. This can be explained be the fact that a tighter bounding DOP takes more space in a block and therefore less DOPs can be retrieved using a single I/O-operation. In some cases however the number of accessed navigation blocks decreases first and then increases; for small blocks see the UNIFORM and CIRCLE distribution in Figure 7.4 and for large blocks see the CIRCLE distribution in Figure 7.8. In these cases a tighter bounding volume is able to successfully exclude some part of the DOP-tree for a query despite of the fact that less bounding DOPs can be stored in a navigation block, as one would hope.

Figure 7.3: The effect of a tighter bounding volume for GRID-DOP-trees on points. Left for small blocks, right for large blocks. The top row shows the effect on the relative number of accessed navigation blocks, the middle row the relative number of content blocks and the bottom row the relative number of accessed navigation and content blocks.
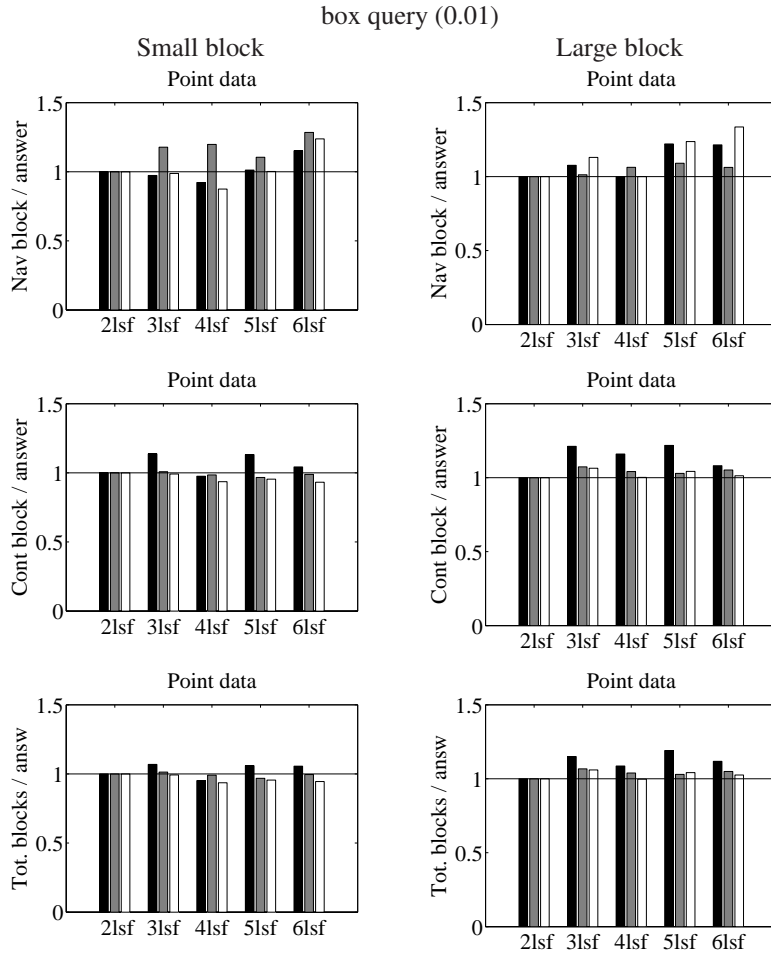
Figure 7.4: The effect of a tighter bounding volume for LSF-DOP-trees on points. Left for small blocks, right for large blocks. The top row shows the effect on the relative number of accessed navigation blocks, the middle row the relative number of content blocks and the bottom row the relative number of accessed navigation and content blocks.

box query (0.01)

Small block
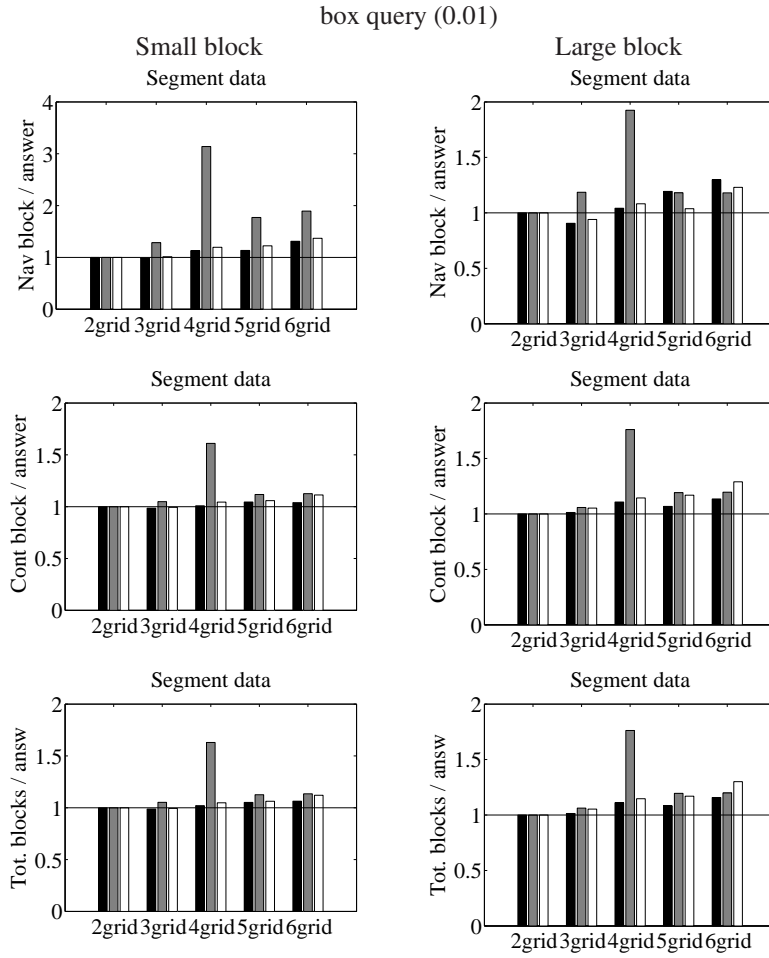
Large block

Figure 7.5: The effect of a tighter bounding volume for GRID-DOP-trees on segments. Left for small blocks, right for large blocks. The top row shows the effect on the relative number of accessed navigation blocks, the middle row the relative number of content blocks and the bottom row the relative number of accessed navigation and content blocks.
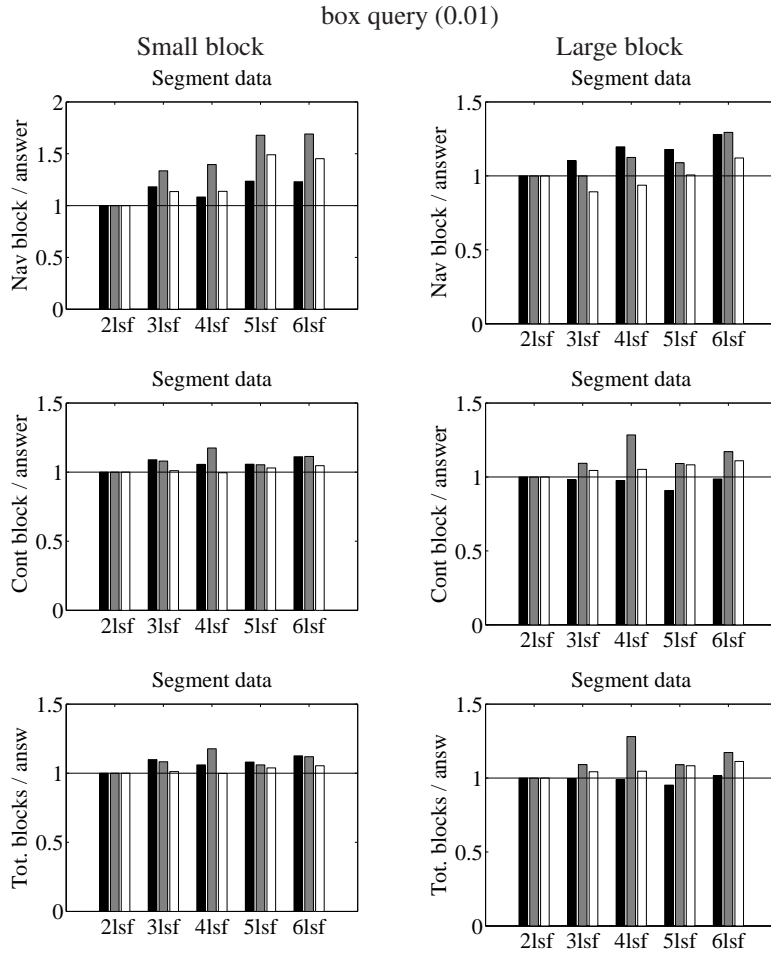
Figure 7.6: The effect of a tighter bounding volume for LSF-DOP-trees on segments. Left for small blocks, right for large blocks. The top row shows the effect on the relative number of accessed navigation blocks, the middle row the relative number of content blocks and the bottom row the relative number of accessed navigation and content blocks.

box query (0.01)

Small block
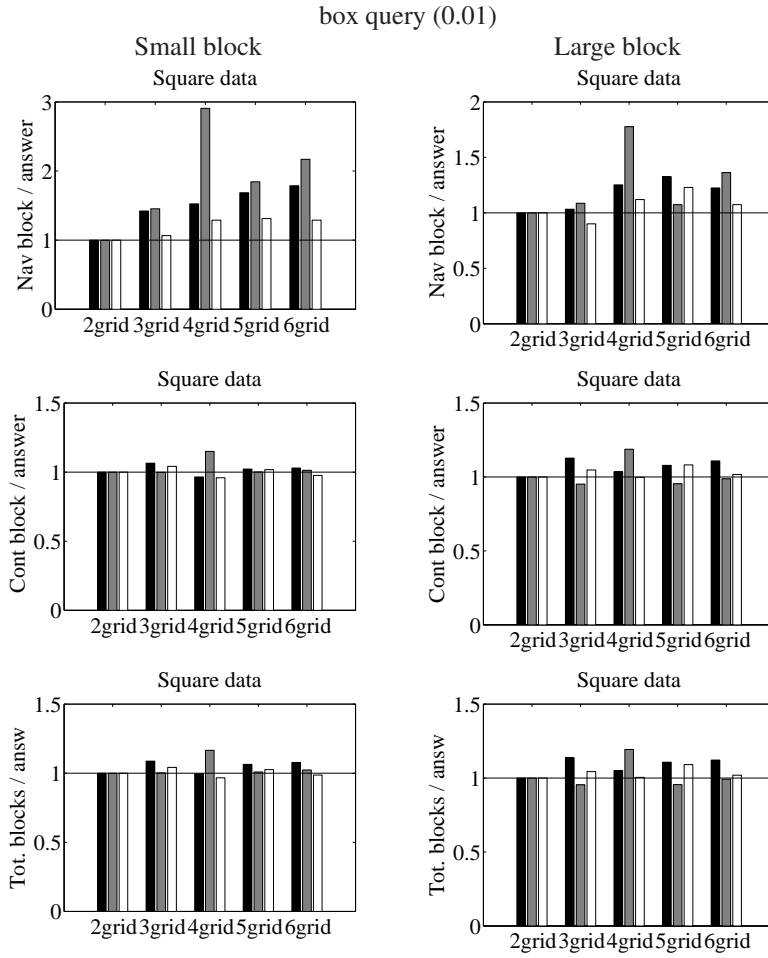
Large block

Figure 7.7: The effect of a tighter bounding volume for GRID-DOP-trees on squares. Left for small blocks, right for large blocks. The top row shows the effect on the relative number of accessed navigation blocks, the middle row the relative number of content blocks and the bottom row the relative number of accessed navigation and content blocks.
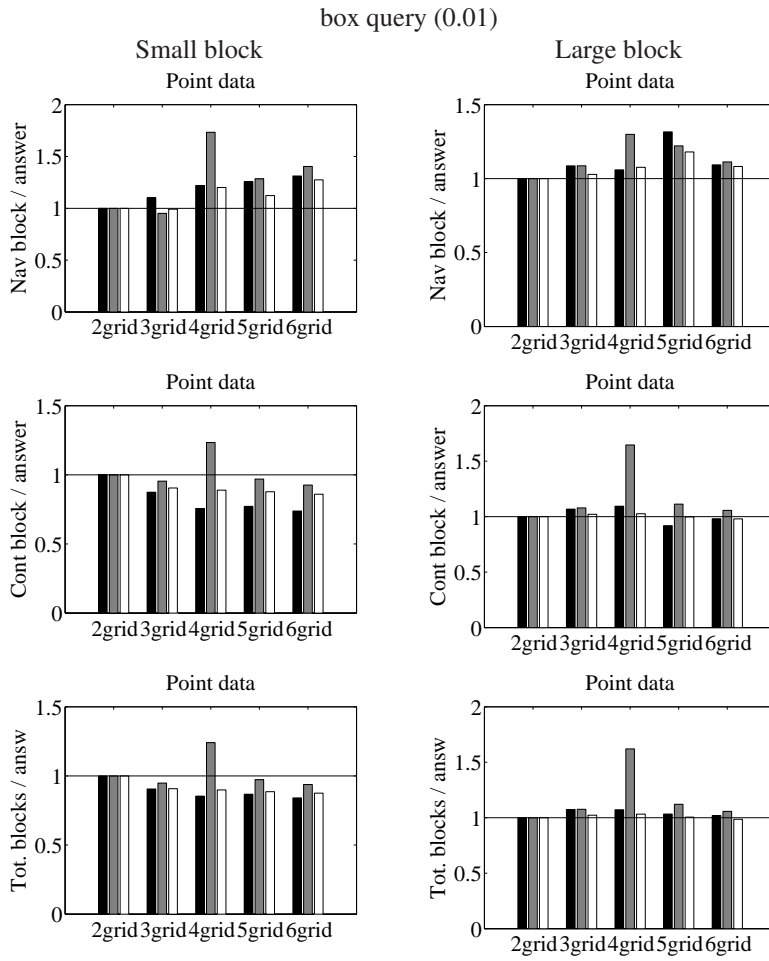
box query (0.01)
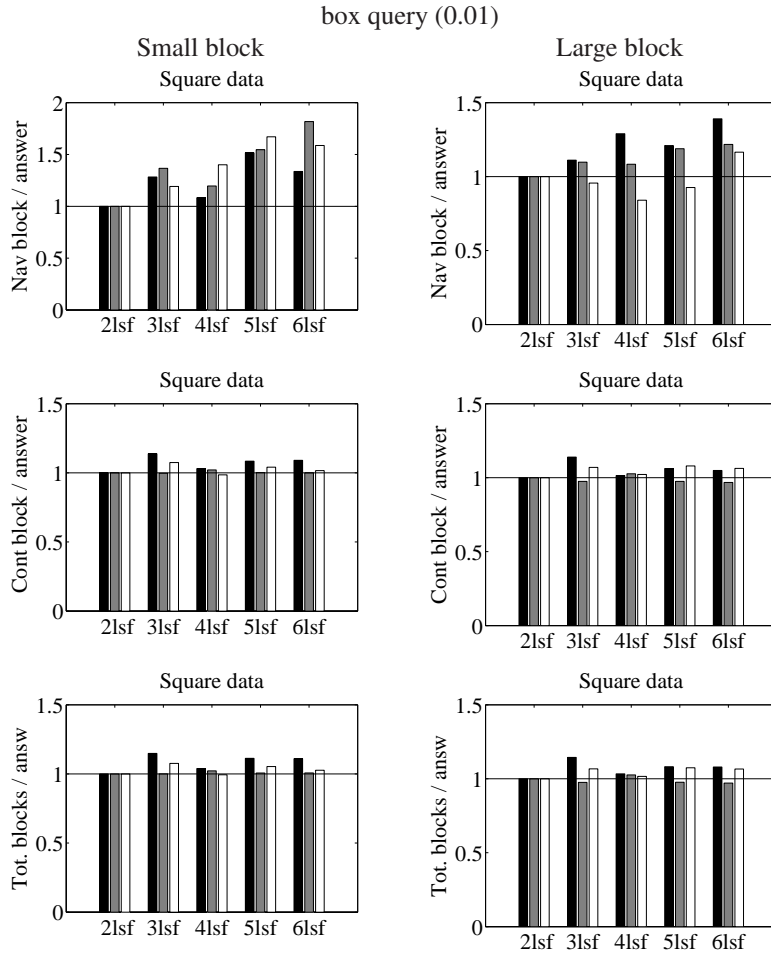
Figure 7.8: The effect of a tighter bounding volume for LSF-DOP-trees on squares. Left for small blocks, right for large blocks. The top row shows the effect on the relative number of accessed navigation blocks, the middle row the relative number of content blocks and the bottom row the relative number of accessed navigation and content blocks.
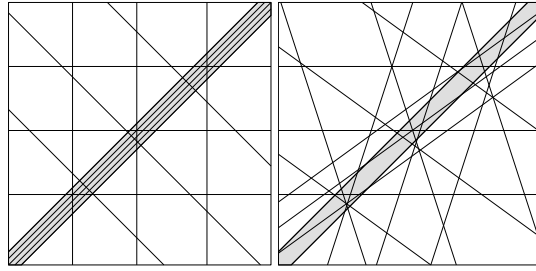
Figure 7.9: The DIAGONAL distribution is depicted as the gray area. In the figure a part of the partitioning induced by a 4-grid BSP (left) and a 5-grid BSP (right) is shown. The cells in the partitioning on the left are skinnier than the cells in the partitioning on the right.

Accessed content blocks versus total number of accessed blocks.   There is hardly a difference in the effect when we compare the effect on using a tighter bounding DOP for the content blocks to the effect for the total number of accessed blocks. The effect of a tighter bounding DOP on the total number of accessed blocks is thus largely determined by the number of accessed content blocks. This is as expected since the number of accessed navigation blocks is only a fraction of the number of accessed content blocks. It is thus important, as one would expect, to keep the number of content blocks accessed by a query as low as possible.

Total number of accessed blocks.   In most cases the DOP-trees with $c = 2$ have the lowest cost. In some settings however using a tighter bounding volume does have an advantage, see Figure 7.3 for a nice example when the DOP-tree is stored in small blocks and the UNIFORM distribution in Figure 7.6 for large blocks.

Peak for 4-GRID-DOP-trees for DIAGONAL distribution.   For the GRID-DOP-tree there is a peak at $c = 4$ for the diagonal line distribution, especially in the case for the point and segment input set. One possible explanation is that the bounding volumes of the priority blocks of a 4-GRID-DOP-tree are very skinny and very close to each other, see Figure 7.9 for a possible partitioning induced by one level in a 4-grid BSP. A query then intersects many priority blocks, while only a few objects are intersected by the query. A similar peak would then also be expected for 4-LSF-DOP-tree and 6-GRID-DOP-tree, because both DOP-trees also store the most extreme objects in priority blocks with the same orientation as the distribution. The absence of large peaks for these two DOP-trees contradicts this explanation.

Another, more plausible, explanation is that during the construction of a 4-grid BSP many skinny regions are constructed. A query can intersect many skinny regions while no or only a few objects intersect the query. The resulting DOP-

tree therefore also will have many bounding volumes intersecting the query. The 4-LSF-kd-tree tries to keep the regions in the BSP fat, which results in less bounding DOPs being intersected by the query in the 4-LSF-DOP-tree. This would explain the absence of a peak at $c = 4$ in Figure 7.4 and Figure 7.8.

7.3.3 Comparison between DOP-trees

In this section we compare the cost of querying with a square in DOP-trees on points, segments and squares. We also compare the DOP-trees with the PR-tree. Since the PR-tree can only handle large blocks we use in this section only large blocks. The DOP-trees on segments are not compared with a PR-tree because the implementation of the PR-tree can not handle segments. The cost for querying in a 2-LSF-DOP-tree are set to 1. The results are shown in Figures 7.10 – 7.11.

Small queries.   For small queries in a point set all LSF-DOP-trees perform equally well. The GRID-DOP-trees are slightly worse than the LSF-DOP-trees, but are better than the PR-tree, especially for the UNIFORM distribution. The 2-LSF-DOP-tree is the best DOP-tree for segments and squares, except for the DIAGONAL distribution in squares. In this case the PR-tree and the c-LSF-DOP-tree for $c \in \{3, 5, 6\}$ are slightly better.

Large queries.   The difference in the query cost in the DOP-trees and in the PR-tree are smaller for large queries than for smaller queries, because many content blocks are completely contained in a large query, while for small queries this is not the case. The average number of accessed blocks per answer decreases when more content blocks are completely contained in the query. The 6-LSF-DOP-tree has the lowest average cost of the trees on points. The 2-LSF-DOP-tree and the 6-LSF-DOP-tree on segments perform equally well for large queries. For squares the PR-tree is slightly better than the 4-LSF-DOP-tree and the 6-LSF-DOP-tree. Overall the 6-LSF-DOP-tree is the tree with the lowest average query cost for large queries.

LSF-DOP-tree versus GRID-DOP-tree.   When we compare a c_1-LSF-DOP-tree with a c_1-GRID-DOP-tree for some $2 \leq c_1 \leq 6$ the LSF-DOP-tree usually has the lower cost. This could be explained by the fatness of the region of the BSPs. The LSF-kd-tree tries to keep the region fat by splitting along the longest side. For the GRID-DOP-tree the regions are not considered at all. This results in very skinny regions especially in the DIAGONAL distribution when the diagonal orientations are used for the bounding DOPs.

PR-tree.   For large queries the PR-tree performs equally well as the best DOP-tree for points and squares. For small queries the PR-tree has a remarkable large difference between the relative cost for the UNIFORM distribution and the other two distributions, especially for points. This could be explained by the density of
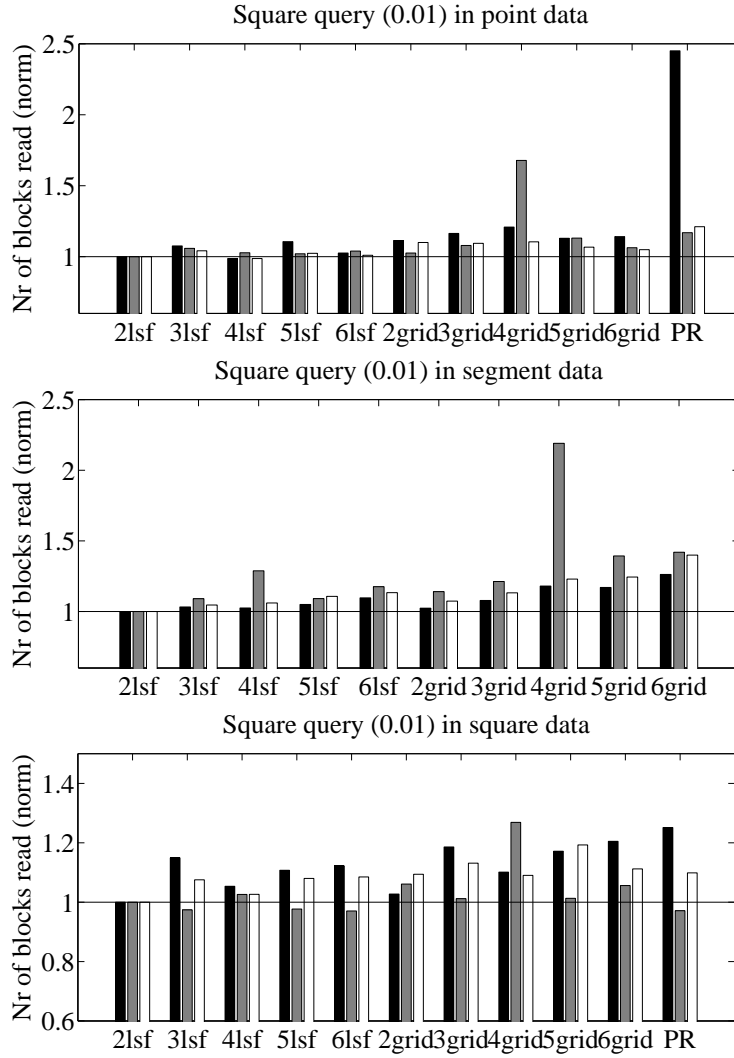
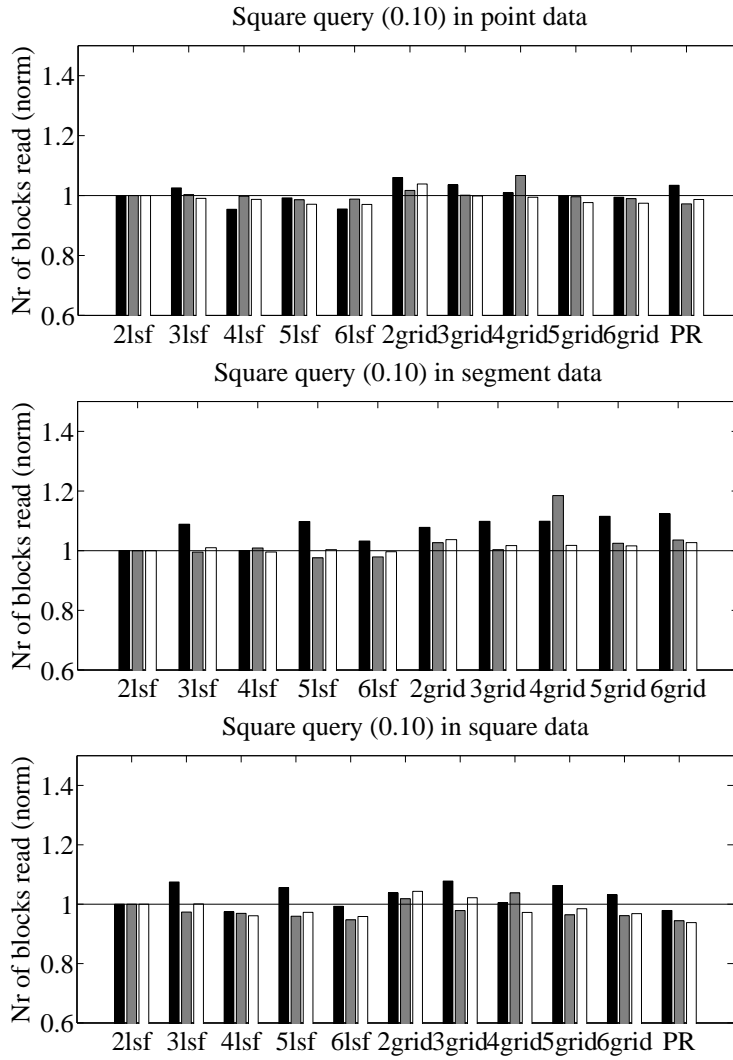Figure 7.10: Relative cost of querying DOP-trees and PR-tree with a small square.

Figure 7.11: Relative cost of querying DOP-trees and PR-tree with a large square.

the objects in the input sets. In the UNIFORM distribution the objects are equally distributed over the entire unit square, for the other two distributions however the same amount of objects is distributed over a small area of the unit square. So in the UNIFORM distribution relatively few objects intersect a small query and therefore only a few blocks might be contained in the query. For the other two distributions more objects intersect the query and therefore it is more likely that a content block contains many objects intersecting the query, which reduces the average number of blocks accessed per answer.

Line segment query. In Figure 7.12 the DOP-trees are compared using 100 queries with line segments of length 0.05. The segment queries are constructed by taking a random point from the input stream. This point is used as one of the endpoints of the segment. We use an arbitrary orientation for the line segment. As with the square queries the GRID-DOP-trees perform worse than the LSF-DOP-trees. For line segment queries in segment and square data a tighter bounding volume is usually better, but how many orientations should be used for the best result is unclear. For point data the 5-LSF-DOP-tree is the best choice when answering line segment queries.

7.4   Conclusions

In this chapter we have investigated the practical use of more tightly bounding volumes in a bounding-volume hierarchy. In our first set of experiments we investigated the effect of the block size on the average number of accessed blocks by a query. We showed that increasing the block size by a factor of four does not decrease the number of accessed blocks by the same factor. We also showed that the decrease in the number of accessed blocks is not the same for every type of bounding volume. The main conclusion of this experiment is that the block size can influence the optimal bounding volume.

In a second experiment we investigated the effect of using a tighter bounding DOP. We showed that a more tightly bounding DOP can reduce the cost of querying with a box. Unfortunately the optimal bounding DOP cannot be predicted in advance, since the optimal bounding DOP is different for every type of input, distribution, size of the query and size of the blocks. This experiment also shows that the number of accessed blocks in a DOP-tree is largely dominated by the number of accessed content blocks.

In the last experiment we compared the DOP-trees and where possible the PR-tree. For small queries the 2-LSF-DOP-tree performs best, except for squares which are distributed along the diagonal. In this case the PR-tree and the $3, 5, 6$-LSF-DOP-tree are slightly better. For large queries the 6-LSF-DOP-tree perform best, except for square data. For square data and large queries the PR-tree is slightly better than the 6-LSF-DOP-tree. Overall the LSF-DOP-trees seem to perform better than the PR-tree, in some cases even much better. However further experimentation is needed to verify this claim.
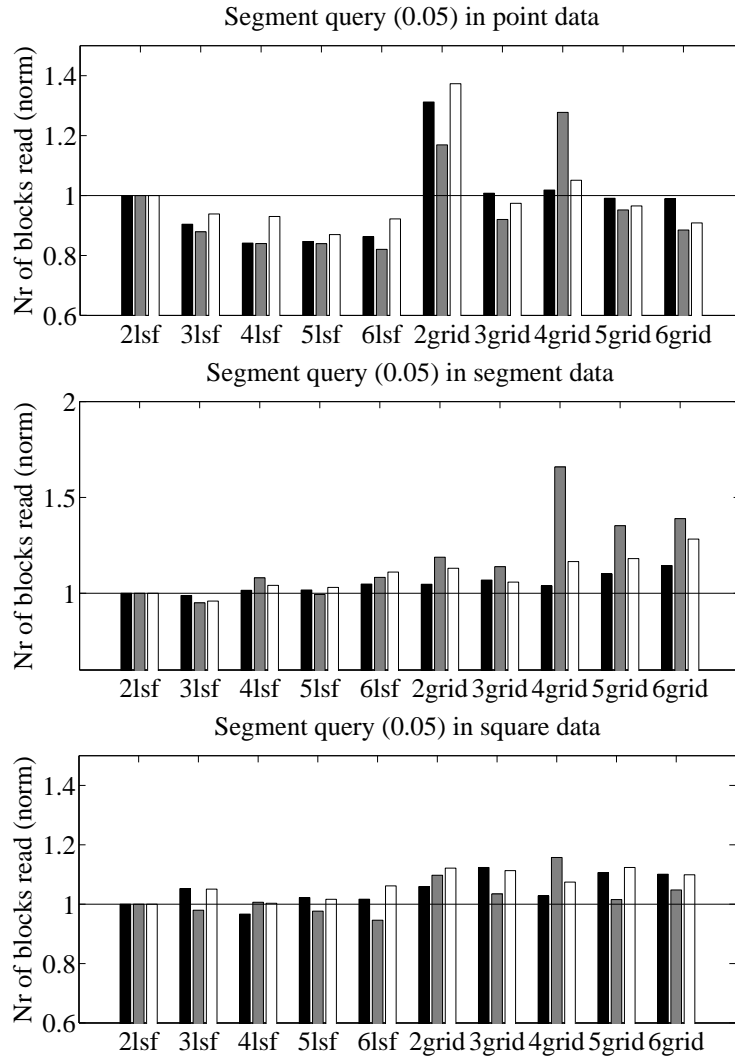
Figure 7.12: Relative cost of querying DOP-trees with a small line segment.

Acknowledgements

Chapter 8

Conclusions and Open Problems

In this thesis multifunctional geometric data structures are investigated. In this chapter some of the conclusions of this thesis are given. Existing multifunctional geometric data structures can be roughly categorized into *bounding-volume hierarchies* and *space-partitioning structures*. The first part of this thesis focused on binary space partitions (BSPs), which are a special type of space partitioning structures. In the second part of this thesis we focused on the other type of important multifunctional geometric data structure, the bounding-volume hierarchy (BVH).

We started in Chapter 2 with an investigation of c-oriented range searching in a BSP on a set of points. We introduced the first partitioning in $\mathbb{R}^d$ such that any c-oriented hyperplane intersects $O(r^{1-1/d})$ cells of the partitioning and any point only one cell. We used this partitioning to obtain a BSP, the c-grid BSP, which has the best known bound on the query time for a query with a c-DOP. A DOP-query in a c-grid BSP is answered in $O(n^{1-1/d+\varepsilon})$ time, which is optimal up to a factor of n^ε. An open question is whether there exists a partitioning into disjoint cells such that *any* hyperplane intersects $O(r^{1-1/d})$ of its cells. A related open question is if there exists a BSP which can answer a range query with an arbitrary triangle in $O(n^{1-1/d})$ time.

In Chapter 3 a framework for converting a BSP for points to a BSP for segments or curves in the plane was introduced. This framework gives us a family of BSPs on segments or curves, whose query time is about as good as the query time for the BSP for points. An extension of this framework to higher dimensions remains open.

We gave for two applications an instantiation of the framework. Let n be the number of segments or curves in the input. In the first application the c-grid BSP was used in the framework to obtain a BSP on segments for c-oriented range searching. The resulting BSP can answer range searching queries in $O(n^{1/2+\varepsilon} + k \log n)$ time where k is the number of segments or curves intersecting the query.

For the second application we applied the BAR-tree to the framework, and obtained a BSP which can answer approximate range queries in $O((1/\varepsilon)\log n + k_\varepsilon \log n)$ time, where k_ε is the number of segments or curves intersecting the extended query range. This is the first result on approximate range searching for disjoint segments in the plane.

In the following chapter we focused on sets of general objects in low-density scenes in $\mathbb{R}^d$. Let n be the number of objects in the scene. We introduced a BSP data structure, the oBAR-tree, which has size $O(n)$ and supports approximate range searching in $O(\log n + \varepsilon^{1-d} + k_\varepsilon)$ time and point location in $O(\log n)$ time where k_ε is the number of objects intersecting the extended query range. This result is more general than the result of Haverkort et al. [50], 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. It remains open how efficient the oBAR-tree is in practice.

The oBAR-tree is constructed using a BAR-tree on a set of guarding points. These points were chosen such that not too many objects intersect the leaves of the BAR-tree. This set of points is large for $d > 2$. One open question is to give a smaller set of guards for objects in $\mathbb{R}^d$ for $d > 2$.

In the last chapter on binary space partitions we described how the BAR-tree and the oBAR-tree can be extended to the external-memory setting. We first showed that existing schemes for dividing the BSP into disk blocks do not work for BAR-trees and we then introduced a new scheme. We called the resulting I/O-efficient data structures the BAR-B-tree and oBAR-B-tree. We proved that to answer an approximate range query in a BAR-B-tree on N points we need to access $O(\log_B N + \varepsilon^\gamma + k_\varepsilon/B)$ disk blocks where k_ε is the number of points intersecting the extended query range, B is the number of nodes which are stored in a disk block and $\gamma = 1 - d$ if the range is convex, and $-d$ otherwise. For the oBAR-tree we need to access $O(\log_B N + \lceil\lambda/B\rceil\varepsilon^\gamma + \lambda k_\varepsilon/B)$ blocks where λ is the density of the input set. It would be interesting to experimentally compare the various schemes for dividing BSPs into disk blocks. Another interesting question is how well the oBAR-B-tree would do in practice compared to the various existing external-memory data structures.

Chapter 6 is the first chapter on bounding-volume hierarchies. The first result in this chapter is a BVH on n c-DOPs which can answer a query with a c-DOP in $O(n^{1-1/c} + k)$ time where k is the number of DOPs intersecting the query. We showed that this BVH is optimal in the worst case. As a second result we gave a framework for constructing a BVH for a set of n objects in the plane with low stabbing number using a BSP on points. We gave for several applications an instantiation of this framework. One of the applications is c-oriented range searching. We used the c-grid BSP in the framework to obtain a BVH which can answer a query with a c-DOP in $O(n^{1/2+\varepsilon}+\sigma^{1-1/c}\log^c n+k)$ time where k is the number of c-DOPs intersecting the query and σ is the stabbing number of the set. Another application is approximate range searching. In this case the BAR-tree was used in the framework. We obtained a BVH which can answer a 4-DOP-query in

$\mathcal{T}_S$ with a DOP Q visits $O(\sigma^{3/4} \log^4 n + \varepsilon^{-1} \log^3 n + k_\varepsilon)$ where k_ε is the number of 4-DOPs intersecting the extended query range.

To prove the query bound we needed to solve the following combinatorial-geometry problem: what is the maximum size of any set $\mathcal{D}$ of DOPs in the plane with the following properties:

(P1) There is a set L of c lines such that every edge of any DOP in $\mathcal{D}$ is parallel to some line in L;

(P2) (the interior of) each DOP in $\mathcal{D}$ intersects every line in L;

(P3) no point in the plane lies in the interior of more than σ DOPs from $\mathcal{D}$.

We were able to prove that $|\mathcal{D}|$ is $\Omega(c\sigma)$ and $O(c^4\sigma)$ in the worst case. A nice open problem is to close the gap between the lower and the upper bound. We suspect that $|\mathcal{D}| = \Theta(c\sigma)$.

In Chapter 7 we implemented an external-memory variant of the bounding-volume hierarchy of Chapter 6. We investigated the effect of the number of orientations used for the bounding DOP on the query cost. In a first experiment we investigated the effect of the block size, the number of items which can be stored in a block, on the query cost. This experiment showed that when the block size is increased by some factor f the query cost is decreased by a factor less than f and that this factor is not the same for every type of bounding volume.

We therefore investigated the effect on query cost for two different block sizes. We used two BSPs in the experiment. The first is the c-grid BSP, introduced in Section 2.2, and the second is an LSF-kd-tree which was adapted to handle more orientations than the two axis-aligned orientations. We showed that there is no general rule of thumb to predict how many orientations to use to obtain the DOP-tree with the lowest query cost. Another outcome of this experiment is that the cost of a query in the DOP-tree largely depends on the number of accessed blocks containing the objects of the input set.

In a third experiment we compared the two types of DOP-trees. In most cases the 2-LSF-DOP-tree has the least cost for small queries and the 6-LSF-DOP-tree for large queries. Furthermore the experiment showed that the GRID-DOP-trees are in general worse than the LSF-DOP-trees. One reason that the LSF-DOP-trees are perform better than the GRID-DOP-trees could be that the LSF-DOP-trees try to keep the regions of the nodes in the BSP as fat as possible, where this is not the case for the GRID-DOP-trees. It would be nice to verify this explanation by using a BAR-tree as the BSP in the DOP-tree framework, since the BAR-tree ensures that all regions are fat.

Dickerson et al. [36] proved for the LSF-kd-tree that they can answer approximate range queries in polylogarithmic time. An open question is whether this is also true for our adapted LSF-kd-tree.

Bibliography

[1] P.K. Agarwal, L. Arge, O. Procopiuc, and J.S. Vitter. A Framework for Index Bulk Loading and Dynamization. In *Proc. of the 28th Annual International Colloquium on Automata, Languages, and Programming (ICALP '01)*, LNCS 2076, 115–127, 2001.

[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*, 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, M.J. Katz, and M. Sharir. Computing depth orders for fat objects and related problems. *Comput. Geom. Theory Appl.*, 5:187–206, 1995.

[6] 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.*, 382–384, 1997.

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

[8] A. Aggarwal, J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116-1127, September 1988.

[9] A.V. Aho, J.E. Hopcroft, J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974

[10] L. Arge. External memory data structures. In *J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, Handbook of Massive Data Sets*, 313–358. Kluwer Academic Publishers, 2002.

[11] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In *Proc. SIGMOD International Conference on Management of Data*, 347–358, 2004.

[12] L. Arge, O. Procopiuc, and J. Vitter. Implementing I/O-Efficient Data Structures Using TPIE, *Proc. of the 10th European Symposium on Algorithms (ESA)*, LNCS 2461, 88–100, 2002.

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

[14] A. Atramentov, S.M. LaValle Efficient Nearest Neighbor Searching for Motion Planning, In *Proc. IEEE Int'l Conf. on Robotics and Automation*, 632–637, 2002.

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

[16] G. Barequet, B. Chazelle, L. J. Guibas, J. S. B. Mitchell, and A. Tal. BOX-TREE: A hierarchical representation for surfaces in 3D. *Computer Graphics Forum* 15(3): 387–396, 1996.

[17] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In: *Proc. Management of Data (SIGMOD)*, 322–331, 1990.

[18] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18 : 509–517, 1975.

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

[20] 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.

[21] 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.*, 61–72. Springer-Verlag, 1994.

[22] M. de Berg, J. Gudmundsson, M. Hammar, M. Overmars. On R-trees with low query complexity, *Comput. Geom. Theory Appl.*, 24: 179–195, 2003.

[23] M. de Berg, H. Haverkort, and M. Streppel. Efficient c-oriented range searching with DOP-trees. *Proc. 13th Annual European Symposium on Algorithms (ESA)*, LNCS 3669, 508–519, 2005.

[24] 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.*, 294–303, 1997.

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

[26] M. de Berg, and M. Streppel. Approximate Range Searching Using Binary Space Partitions. *Comp. Geom. Theory Appl.*, 33 (3): 139–151, 2006.

[27] M. de Berg, and M. Streppel. Approximate Range Searching Using Binary Space Partitions. *Proc. 24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 3328, pages 110–121, 2004.

[28] 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*, 110–148, Springer, 2003.

[29] B. Chazelle . Filtering search: a new approach to query answering. *SIAM J. Comput.* 15(3): 703–724, 1986.

[30] B. Chazelle. Lower bounds for orthogonal range searching: I. The reporting case, *J. ACM*, 37(2): 200–212, 1990.

[31] B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9(2): 145–158, 1993.

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

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

[34] K.L. Clarkson. New Applications of Random Sampling in Computational Geometry. *Discrete and Computational Geometry* 2(2): 195–222, 1987.

[35] B.V. Dasarathy, editor *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques*, ISBN 0-8186-8930-7, (1991) .

[36] 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, 179-190, 2000.

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

[38] 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*, 300–309, 1999.

[39] P. van Emde Boas. Machine models and simulations, In: *Handbook of the-oretical computer science (vol. A): algorithms and complexity*, 1–66, 1990, ISBN 0-444-88071-2, MIT Press, Cambridge, MA, USA.

[40] S.A. Ehmann, and M.C. Lin. Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition, *Comput. Graph. Forum* 20(3):500–510, 2001.

[41] J.H. Freidman, J.L. Bentley, and R.A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time, *ACM Trans. Math. Softw.* 3(3):209–226, 1977

[42] 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. SIG-GRAPH '80.

[43] C. Fünfzig, and D.W. Fellner. Easy Realignment of k-DOP Bounding Vol-umes. In: *Graphics Interface*, 257–264, 2003.

[44] V. Gaede, and O. Günther. Multidimensional access methods. *ACM Com-puting Surveys*, 30(2), 170–231, 1998.

[45] Y.J. García, M.A. López, and S.T. Leutenegger. A greedy algorithm for bulk loading R-trees. In: *Proc. Advances in GIS*, 163–164, 1998

[46] S. Gottschalk, M.C. Lin, and D. Manocha. OBB-Tree: a hierarchical struc-ture for rapid interference detection, In *Proc. Computer Graphics (SIG-GRAPH)*, 171-180, 1996.

[47] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD International Conference on Management of Data*, 47–57, 1984.

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

[49] H.J. Haverkort. Results on Geometric Networks and Data Structures. Ph.D. Thesis, Utrecht University, 2004.

[50] H.J. Haverkort, M. de Berg, and J. Gudmundsson. Box-trees for collision checking in industrial installations. In *Proc. Symposium on Computational Geometry*, 53–62, 2002.

[51] P.M. Hubbard. Approximating polyhedra with spheres for time-critical col-lision detection, *ACM Trans. Graph.* 15(3), 179–210, 1996

[52] H.V. Jagadish. Spatial Search with Polyhedra. *In Proc. Int. Conf. Data Engineering (ICDE)*, 311–319, 1990

[53] I. Kamel, and C. Faloutsos. On Packing R-trees. In: *Proc. Conf. Information and Knowledge Management (CIKM)*, 490–499, 1993.

[54] I. Kamel, and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In: *Proc. Very Large Databases (VLDB)*, 500–509, 1994.

[55] K.V. Ravi Kanth, and A.K. Singh. Optimal Dynamic Range Searching in Non-replicating Index Structures. In *Int. Conf. Database Theory (ICDT)*, 1999, LNCS 1540:257-276.

[56] 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 Trans. on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[57] M. van Kreveld. On fat partitioning, fat covering, and the union size of polygons. *Comput. Geom. Theory Appl.*, 9:197–210, 1998.

[58] E. Larsen, S. Gottschalk, M.C. Lin, and D. Manocha. Fast Distance Queries with Rectangular Swept Sphere Volumes In *IEEE Intl. Conf. on Robotics and Automation*, 2000.

[59] S.T. Leutenegger, J.M. Edgington, and M.A. López. STR: A simple and efficient algorithm for R-tree packing. In: *Proc. 13th Int. Conf. on Data Eng.*, 497–506, 1997.

[60] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-trees: Theory and Applications*. Springer, 2005.

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

[62] J. Matoušek. Reporting points in halfspaces. In: *Proc. of 32nd Found. Comput. Sci.*, 207–215, 1991.

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

[64] J.S.B. Mitchell, D.M. Mount, and S. Suri. Query-sensitive ray shooting. *Internat. J. Comput. Geom. Appl.*, 7:317–347, 1997.

[65] 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.

[66] J. Nievergelt and P. Widmayer. Guard files: Stabbing and intersection queries on fat spatial objects. *Comput. J.*, 36:107–116, 1993.

[67] J. Nievergelt and P. Widmayer. Spatial data structures: concepts and design choices. In: M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer (eds.), *Algorithmic Foundations of Geographic Information Systems*, LNCS 1340, Springer-Verlag, 153–197, 1997

[68] P. van Oosterom. Reactive Data Structures for Geographic Information Systems. Ph.D. thesis, Leiden University, 1990.

[69] M.H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, LNCS 156, 1983.

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

[71] 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.

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

[73] Ph. Pignon. *Structuration de l' espace pour une planification hiérarchisée des trajectoires de robots mobiles*. Ph.D. thesis, LAASCNRS and Universit'e Paul Sabatier de Toulouse, 1993. Rapport LAAS No 93395 (in French).

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

[75] J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD International Conference on Management of Data*, 10–18, 1981.

[76] H. Samet. Applications of spatial data structures: computer graphics, image processing, and GIS. Addison-Wesley, Reading, MA, 1990.

[77] H. Samet. The Design and Analysis of Spatial Data Structures. Addison-Wesley, Reading, MA, 1990.

[78] H. Samet. Foundations of Multidimensional and Metric Data Structures. Morgan-Kaufmann, San Francisco, 2006

[79] T.K. Sellis, and N. Roussopoulos, and C. Faloutsos. The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects. In: *Proc. Very Large Databases (VLDB)*, 507–518, 1987.

[80] I. Sitzmann and P.J. Stuckey. The O-Tree – A Constraint-Based Index Structure, technical report, University of Melbourne, 1999.

[81] A.F. van der Stappen. *Motion Planning amidst Fat Obstacles*. Ph.D. thesis, Dept. Comput. Sci., Utrecht Univ., Utrecht, the Netherlands, October 1994.

[82] A.F. van der Stappen, M.H. Overmars, M. de Berg, J. Vleugels. Motion Planning in Environments with Low Obstacle Density, *Discrete Comput. Geom.* 20(4): 561–587, 1998

[83] M. Streppel and K. Yi. Efficient Spatial Indexes for Approximate Range Searching. In: *Abstracts 23th European Workshop on Computational Geometry*, 227–230, 2007.

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

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

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

[87] 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.

[88] J.S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[89] J. Vleugels. On Fatness and Fitness—Realistic Input Models for Geometric Algorithms Ph.D. thesis, Dept. Comput. Sci., Univ. Utrecht, Utrecht, the Netherlands, 1997.

[90] G. Zachmann. Rapid Collision Detection by Dynamically Aligned DOP-Trees. In *Proc. of IEEE Virtual Reality Annual International Symposium*, 90–97, 1998.

Acknowledgements

The last four years I was a PhD student with the Algorithms group at the Technische Universiteit Eindhoven. I still remember the day I came to Eindhoven to apply for the PhD-position. I did not have any experience in computational geometry, but this did not prevent Mark de Berg from offering me the position in his newly founded group. I would like to thank him for his confidence in me.

I started my PhD under the supervision of Mark de Berg. After Herman Haverkort joined the Algorithms group he became my daily supervisor. I would like to thank both of them for coming up with problems to work on and listening to my ideas for solving these problems. I also value the collaboration with them on the papers [27, 23, 26] which are the basis for most chapters in this thesis. I would also like to thank Ke Yi not only for our collaboration on a paper [83] which served as the source for Chapter 5, but also for the source code of the PR-tree which was used in some experiments in Chapter 7.

The group moved around the sixth floor several times before settling at the seventh floor. At first I shared an office with Joachim Gudmunsson and after a year I moved to Mohammad Abam's office. For about one year we shared our office with master student Xavier Clairbois and in the last year of my PhD Shripad Thite and Sheung-Hung Poon joined us. I would like to thank my roommates and all other people of the Algorithms group for their conversations and discussions.

I thank Marc van Kreveld, Joe Mitchell, and Gerhard Woeginger for taking part in the reading committee of my thesis. Jos Baeten I thank for completing my doctorate committee.

Life does not consist of work alone and I would like to thank my friends and family for their being there for me. Especially I would like to thank my mother Francien and her partner Bert.

Mark Schroders, Hugo Jonker, and Shripad Thite I would like thank for the pleasant coffee breaks, and their company while swimming or running. Mark introduced me to the people of Eximion. I always like visiting the Eximion house and see the progress on their games and game engine. I especially would like to thank Doki Tops for creating the cover of this thesis. For the nice evenings and weekends of board games and conversations I would like to thank Bart van Leeuwen, Martijn Sonnenberg and Roy Smits.

Bas Maas, Helga Bosch, Jolanda van Wijk, Maud de Corti, Peter Ebben, and Rob Arntz I thank for taking me into the country side to go hiking. These walks

were very relaxing and renewed my energy to work on my PhD. In these past four years I still haven't been able to finish the Pieterpad. Now I have finished writing my thesis I hope to finish it.

Last, but certainly not the least, I thank my girlfriend Suzanne Geelen for her support and her company.

Summary

Multifunctional Geometric Data Structures

Many computational problems can be stated in terms of geometric queries on a set of objects, such as: 'Which objects in the set are contained within this region?'. This query is an example of a *range searching query*. If a set of objects is often queried then it is beneficial to build a data structure on these objects such that the queries can be answered faster. This can be a very specific data structure, that only answers one specific type of geometric query. However, it can also be a general data structure, that answers many different queries and stores many types of objects. The latter type of data structures we call *multifunctional geometric data structures* and these are the topic of this thesis. Two widely used multifunctional geometric data structures are the *binary space partition* (BSP) and the *bounding-volume hierarchy* (BVH).

In this thesis the main focus is on c-oriented range searching and approximate range searching in BSPs and BVHs. In c-oriented range searching the sides of the query range have orientations from a predefined set of c orientations. The well-studied axis-aligned range searching is a, rather limited, example of c-oriented range searching. In approximate range searching one considers an extended query range in order to obtain a better theoretical bound. However, the bound is then based on the number of objects in the extended query range instead of in the original query range. One generally assumes that there are not too many objects intersected by the larger query range, which are not intersected by the original query range.

In the first part of this thesis BSPs are investigated. In Chapter 2 c-oriented range searching in a set of points is considered for which a new BSP, the c-grid BSP, is introduced. The c-grid BSP is the first BSP on a set of points, which answers a c-oriented range-searching query in the same time, up to a factor $O(n^\varepsilon)$, as the query time for axis-aligned range queries in the well-known kd-tree.

In the next chapter the focus is shifted to BSPs on segments. A framework is introduced for converting a BSP on the endpoints of the segments to a BSP on the segments. The framework is used to obtain the first BSP on line segments for c-oriented range searching and the first BSP on line segments for approximate range searching. By initializing this framework with the c-grid BSP of the pre-

vious chapter the BSP on line segments is obtained for c-oriented range searching. The BSP for approximate range searching is obtained by initializing the framework with another known BSP, the BAR-tree. Both instantiations answer a range query in a set of segments in approximately the same time as the same query in a set of points.

Chapter 4 introduces a BSP of linear size for a set of general objects that forms a so-called *low-density scene*. It is believed that in practice many sets of objects form low-density scenes. This BSP is based on the BAR-tree and is called the OBAR-tree. It answers approximate range-searching queries in a set of objects in the same time as the query time of a BAR-tree on a set of points. The OBAR-tree is the first BSP for range searching on general objects.

When handling very large sets the size of the internal memory becomes a bottleneck. The objects and the data structure no longer fit in memory and to perform the necessary computations for answering the query parts of the data structure have to be transferred from external storage, for instance a hard disk, to internal memory. These transfers dominate the total query time. In Chapter 5 a new storage method is introduced for the BAR-tree and the OBAR-tree. This method is the first whose bound on the number of transfers between external storage and internal memory for answering approximate range queries in the BAR-tree and the OBAR-tree is theoretically optimal.

The second part of this thesis deals with c-oriented range searching in BVHs. In Chapter 6 a BVH is introduced which can answer c-oriented range queries in a set of c-oriented polygons. The query time in this BVH is optimal in the worst-case. For the worst-case example the input has to intersect a lot. In practice, however, this type of input is hardly encountered. Therefore in the remainder of Chapter 6 only sets of objects, which do not intersect too much, are considered. For these sets a framework is introduced for constructing BVHs on objects using a BSP on points. The BVH obtained after instantiating the framework are called DOP-trees. We instantiate this framework with the c-grid BSP and obtain the GRID-DOP-tree, that answers c-oriented range-searching queries in almost the same time as the query time of the same query range in the c-grid BSP on a set of points. In the last chapter experiments with external-memory variants of two types of DOP-trees show that more orientations sometimes lower the query cost. These experiments are the first experiments with an external-memory BVH which uses c-oriented bounding volumes. The experiments also show that DOP-trees need less transfers between external storage and internal memory than PR-trees, one of the theoretically optimal BVHs for external memory.

Samenvatting

Multifunctionele Geometrische Gegevensstructuren

Veel computationele problemen kunnen geformuleerd worden in termen van geometrische vragen over een verzameling van objecten, zoals: 'Welke objecten in de verzameling bevinden zich in dit deel van de ruimte?'. Deze vraag is een voorbeeld van een *gebiedsdoorzoekingsvraag*. Als er veel vragen gesteld worden over een verzameling, dan is het vaak voordeliger een gegevensstructuur te bouwen op die verzameling, zodat vragen sneller beantwoord kunnen worden. Deze gegevensstructuren kunnen erg specialistisch zijn, waardoor ze slechts één type vraag snel kunnen beantwoorden. Er bestaan echter ook gegevensstructuren die meerdere typen vragen kunnen beantwoorden en verzamelingen met verschillende typen objecten kunnen opslaan. Deze laatste soort gegevensstructuren noemen we *multifunctionele geometrische gegevensstructuren* en deze zijn het onderwerp van dit proefschrift. Twee vaak voorkomende soorten multifunctionele geometrische gegevensstructuren zijn de *binary space partition* (BSP) en de *bounding volume hierarchy* (BVH). Beide gegevensstructuren hebben een boomstructuur. Bij BSP's wordt de verzameling objecten in twee delen gesplitst door een vlak (in twee dimensies door een lijn). Dit vlak wordt opgeslagen in een knoop ν van de boom. De twee resulterende verzamelingen worden recursief verder gesplitst totdat er zich één of een constant aantal objecten overblijven. Bij de knoop ν worden verwijzingen opgeslagen naar de bomen die ontstaan door de recursieve opsplitsing. Bij BVH's worden objecten gegroepeerd. Deze groepen worden weer recursief gegroepeerd totdat uiteindelijk alle objecten in één groep zitten. Bij elke knoop in de boom wordt een *omvattend volume* (Engels: bounding volume) opgeslagen en een verwijzing naar de groepen die zijn samengevoegd.

De resultaten in dit proefschrift hebben voor het grootste deel betrekking op c-geörienteerde gebiedsdoorzoeking en benaderde gebiedsdoorzoeking in zowel BSP's als BVH's. In c-geörienteerde gebiedsdoorzoeking komen de richtingen van de zijden van het gebied dat doorzocht moet worden, uit een van te voren gedefinieerde verzameling van c richtingen. Een veel bestudeerde vraag is het vinden van alle objecten in een gebied waarvan alle randen parallel zijn aan een van de assen. Deze vraag is een (vrij beperkt) voorbeeld van een c-geörienteerde gebiedsdoorzoekingsvraag en wordt een *as-parallelle gebiedsdoorzoekingsvraag* genoemd.

Bij benaderde gebiedsdoorzoeking wordt voor de analyse van de antwoordtijd een gebied gebruikt dat groter is dan het gevraagde gebied om een betere theoretische begrenzing op de antwoordtijd te geven. De begrenzing is dan wel afhankelijk van het aantal objecten dat in dit grotere gebied ligt. Over het algemeen wordt aangenomen dat er niet veel meer objecten in het grotere gebied liggen dan in het gevraagde gebied.

In het eerste deel van dit proefschrift richt de aandacht zich op BSP's. In hoofdstuk 2 wordt c-geörienteerde gebiedsdoorzoeking in een verzameling punten bekeken. Hiervoor wordt een nieuwe BSP geïntroduceerd, de c-grid BSP. De c-grid BSP is de eerste BSP, die voor een verzameling van n punten een c-geörienteerde gebiedsdoorzoekingsvraag kan beantwoorden in dezelfde tijd, op een factor $O(n^{\varepsilon})$ na, als de tijd die nodig is voor het beantwoorden van een as-parallelle gebiedsdoorzoekingsvraag in de bekende kd-tree .

In het derde hoofdstuk wordt er gekeken naar BSP's op lijnstukken. Er wordt een raamwerk geïntroduceerd voor het omzetten van een BSP op eindpunten van lijnstukken naar een BSP op lijnstukken zelf. Met dit raamwerk wordt de eerste BSP op lijnstukken voor c-geörienteerde gebiedsdoorzoeking verkregen en de eerste BSP op lijnstukken voor benaderde gebiedsdoorzoeking. Als in het raamwerk de c-grid BSP van het vorige hoofdstuk wordt gebruikt, dan wordt een BSP voor c-geörienteerde gebiedsdoorzoeking verkregen. Een BSP voor benaderde gebiedsdoorzoeking wordt verkregen door het raamwerk te initialiseren met een andere, al bekende, BSP: de BAR-tree. De tijd die nodig is voor het beantwoorden van een gebiedsdoorzoekingsvraag is voor beide instanties van het raamwerk vrijwel gelijk aan de tijd die nodig is voor het beantwoorden van dezelfde vraag in een verzameling punten.

In hoofdstuk 4 wordt een BSP van lineaire grootte voor een verzameling van objecten, die tezamen een *scène* met een *lage dichtheid* vormen, geïntroduceerd. Over het algemeen wordt aangenomen dat scènes die vaak in de praktijk voorkomen een lage dichtheid hebben. Deze BSP is gebaseerd op de BAR-tree en wordt oBAR-tree genoemd. De oBAR-tree kan benaderde gebiedsdoorzoekingsvragen beantwoorden in de tijd die nodig is voor het beantwoorden van dezelfde vraag over een verzameling punten in een BAR-tree. De oBAR-tree is de eerste BSP voor gebiedsdoorzoeking in een verzameling van objecten.

Bij zeer grote verzamelingen kunnen de verzamelingen niet meer in het werkgeheugen worden opgeslagen. Deze verzamelingen moeten opgeslagen worden op een extern opslagmedium, bijvoorbeeld een harde schijf of CD. Om de geometrische vragen toch te kunnen beantwoorden moeten delen van de gegevensstructuur en de objecten zelf van het extern opslagmedium verplaatst worden naar het werkgeheugen. Deze verplaatsingen domineren de tijd die nodig is voor het beantwoorden van de vragen. De *kosten* voor het beantwoorden van een vraag worden daarom uitgedrukt in het aantal benodigde verplaatsingen. In hoofdstuk 5 wordt een methode geïntroduceerd om de BAR-tree en de oBAR-tree op te slaan in het extern opslagmedium zodanig dat de kosten voor het beantwoorden van een gebiedsdoorzoekingsvraag minder zijn dan bij bestaande methoden.

De tweede helft van het proefschrift richt zich op c-geörienteerde gebieds-

doorzoeking in de tweede soort multifunctionele geometrische gegevensstructuren, de BVH. Als eerste wordt er een BVH gedefinieerd die c-geörienteerde gebiedsdoorzoekingsvragen over een verzameling van c-geörienteerde veelhoeken kan beantwoorden in een tijd, die optimaal is voor het slechtste geval. In het slechtste geval overlappen de veelhoeken veelvuldig. In de praktijk zal dit echter niet vaak gebeuren. Daarom beschouwen we in het restant van hoofdstuk 6 verzamelingen van c-geörienteerde veelhoeken die niet te veel overlappen. Voor deze verzamelingen wordt er een raamwerk gegeven voor de constructie van BVH, waarbij gebruik gemaakt wordt van een BSP op punten. De resulterende BVH wordt een DOP-tree genoemd. Het raamwerk wordt geïnitialiseerd met de c-grid BSP, waardoor de GRID-DOP-tree ontstaat. De GRID-DOP-tree beantwoordt een c-geörienteerde gebiedsdoorzoekingsvraag in ongeveer dezelfde tijd die nodig is voor het beantwoorden van dezelfde vraag in de c-grid BSP voor een verzameling punten.

In het laatste hoofdstuk wordt er geëxperimenteerd met een variant van de DOP-trees die wordt opgeslagen op een extern opslagmedium. In de experimenten worden twee verschillende BSP's gebruikt voor de initialisatie van het raamwerk. De experimenten tonen aan dat meer richtingen in de beschrijving van het omvattend volume, waardoor de groepen eronder beter benaderd worden door het omvattend volume, de kosten voor het beantwoorden van een gebiedsdoorzoekingsvraag verminderen. Deze experimenten zijn de eersten waarbij het omvattend volume een c-geörienteerde veelhoek is en waarbij de BVH op een extern opslagmedium wordt opgeslagen. De kosten voor het beantwoorden van een gebiedsdoorzoekingsvraag van de DOP-trees worden ook vergeleken met die van de PR-tree. De PR-tree is een van de theoretisch optimale BVH's voor externe opslagmedia. De experimenten tonen aan dat het aantal verplaatsingen tussen het opslagmedium en het werkgeheugen voor het beantwoorden van gebiedsdoorzoekingsvragen bij de DOP-trees minder is dan bij de PR-tree.

Curriculum Vitae

Michaël (Micha) Streppel was born on 29th of April 1977 in Nijmegen, the Netherlands. For his pre-university education he went to the Merletcollege in Cuijk. In 1995 he started to study chemistry at the University of Nijmegen, nowadays called Radboud University Nijmegen. He started studying computer science at the University of Nijmegen in 1999. Two years later, in 2001, he received his Master's degree for computer science and seven months later his Master's degree in chemistry. As of 2001 he worked for a company called Maassen Consulting until he started his PhD in april 2003 in the Algorithms group at the Technische Universiteit Eindhoven.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences,

Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms*

and Proofs. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java - Theory and Tool Support- *. Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical

Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support*

and Compositionality. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous*

Distributed Systems. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07