RESULTS ON GEOMETRIC NETWORKS AND DATA STRUCTURES

RESULTATEN OVER RUIMTELIJKE
NETWERKEN EN GEGEVENSSTRUCTUREN

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van
doctor aan de Universiteit Utrecht
op gezag van de Rector Magnificus,
Prof. dr. W. H. Gispen, ingevolge het
besluit van het College voor Promoties
in het openbaar te verdedigen op
maandag 17 mei 2004 des ochtends te 10:30 uur

door

Herman Johannes Haverkort

geboren op
4 juli 1974,
te Arnhem

promotoren:	Prof. dr. Mark H. Overmars, faculteit Wiskunde & Informatica, Universiteit Utrecht
	Prof. dr. Mark T. de Berg, faculteit Wiskunde & Informatica, Technische Universiteit Eindhoven

This work was supported by the Netherlands Organisation for Scientific Research NWO, grant number 612.015.003.

NWO

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 101.

Advanced School for Computing and Imaging

Contents

Chapter 1

Introduction

Computational geometry is the area of algorithms research that deals with computations on geometric objects. Examples of such objects are points, lines and polygons in the plane—which may represent a city plan—or balls, blocks and more complex shapes in three dimensions—which may represent the interior of a power plant. In these cases, the geometric objects represent physical objects in the real world. But this is not always the case. For example, a database storing the age and salary of a company's employees can also be thought of as a database that stores points in a two-dimensional space: each point represents an employee, with one coordinate indicating the age and the other coordinate indicating the salary of the employee. Therefore, geometric computations are found in many applications of computers: databases, computer-aided design, geographical information systems, flight simulators, other virtual reality applications, robotics, computer vision and route planning are just a few examples.

This thesis contains some of the fruits of my research as a PhD student in the computational-geometry group of Utrecht University from 1999 to 2004. In the remainder of this chapter I will introduce my main research areas of these years—*geometric data structures* and *geometric networks*—and I will discuss the results we obtained. The next chapters are six articles written in these areas: four articles about geometric data structures (Chapter 2 to 5) and two articles about geometric networks (Chapter 6 and 7).[1]

1.1 Geometric data structures

To do efficient computations on geometric objects, it is crucial that we can store, search, and sometimes update, sets of geometric objects efficiently. When the objects are one-dimensional points, this is relatively easy: we can sort them by

[1]The article in Chapter 3 is a slightly modified version from the publication in *Discrete and Computational Geometry*. In this thesis, the analysis of our algorithm has been sharpened a little, leading to better bounds on the query times.

their coordinate in that dimension, and put them in memory in that order. This makes it possible to find points fast. It is like looking up words in a dictionary: thanks to the ordering, we can find a word without turning all pages one by one.

When an object cannot be described by a single point in a one-dimensional space, it is less clear how to store a set of objects effectively. For example, I have a collection of music CD's. I would like to order them by the year in which the music was written, so that I can find all music from a particular era fast. My CD's usually contain several works of music written in a range of years. This makes it impossible to characterise a CD by a single point on a time line: a CD is rather like a group of points, or like a segment of the time line. How should I order my CD's? By the oldest work on the CD, maybe? But then even the most recent work might be put in the very first place on the shelf, if it just happens to be on the same CD as the oldest work. If I sort like this, how can I be sure of finding a work from a certain era without checking all earlier CD's too?

When geometric objects have more than one dimension, the problem becomes even more difficult. But in many applications, a lot of questions about a set of geometric objects need to be answered fast. For example, a flight simulator should not need to scan the complete hard disk to determine if the plane is going to hit a mountain in the next second. In such applications, it is essential that geometric objects are stored in such a way that relevant objects can be identified quickly, while irrelevant objects are ignored without checking them one by one. Often we can do this by sorting the objects into groups. If we do this in a clever way, we can, hopefully, discriminate quickly between groups with potentially relevant objects and groups without such objects. Therefore, finding useful groupings of objects is a key issue in many problems in computational geometry.

A set of geometric objects that is sorted, partitioned into groups and/or otherwise preprocessed, so that certain queries about the set can be answered efficiently, is called a *geometric data structure*. The goal of research into such data structures is to make them as efficient as possible with respect to storage space, the time needed to build the data structure, the time needed to insert or delete objects, and the time needed to answer queries. Examples of such queries are: which objects lie (partly) inside a given viewing window? Or: which object is closest to a given query point? Of course, we would like our data structure to facilitate fast answers, not for just one particular query point or window, but for *any* query that might be asked. One cannot usually expect to optimize for all of the objectives mentioned at the same time. In general, the faster the queries, the larger the demands on storage, preprocessing and update time.

We do not usually measure the running times of data structure algorithms by counting milliseconds. We could, of course, but with computers getting faster all the time, this would make our results outdated even before they are published. Rather we ask the question: how well will a data structure be able to take advantage of bigger and faster computers? To answer that question, we analyse in what way the number of basic operations performed by the central processor depends on the input size (the number of objects stored) and the output size (the number of objects retrieved). The first is usually denoted by n, the second by k. We will write that algorithms have a running time of, for example, $O(n)$ or $O(n^3)$. In the first case, the running time is a linear function of n. This means that if we can double the speed of our hardware, this algorithm can process twice as much data in the same time. In the second case, the running time is a cubic function of n, which means that our double-speed computer will enable us to handle only 26% more data with this algorithm. This means that even if the second algorithm would be a little faster in practice on the current hardware, the first algorithm is probably more promising in the future.

If the amount of data is so big that we cannot keep all of it in main memory while working on it, we count the number of disk accesses rather than the number of operations. In that case we analyse how the running times depend on three parameters: the input size n, the output size k, and the amount of data transferred in one disk access.

In the most basic form, geometric data structures store points and we want to be able to retrieve, for any query range, the points that are inside, or sometimes the points that are closest to that query range. This type of data structures has been well studied and structures have been developed for simplex range queries, axis-parallel (hyper-)rectangular range queries, circular or (hyper-)spherical range queries, and point queries. With $O(n)$ space, one can build a data structure for n points in d dimensions that reports all points inside a simplex in time $O(n^{1-1/d} + k)$ [Mat93], where k is the number of points reported. For queries with axis-parallel rectangles, one can use the same data structure, or the much simpler *kd-tree* with the same query time (see e.g. [Brg97KOS] for a description). With more space, one can often get faster queries. For example, a *layered range tree* answers axis-parallel rectangle queries in time $O(\log^{d-1} n + k)$, using $O(n \log^{d-1} n)$ space [Brg97KOS]. There are also other data structures whose query times depend more heavily on the output size and less on the input size. For more about data structures for points, see, for example, the survey by Agarwal and Erickson [Aga98E].

1.1.1 Data structures for categorical data

In some problems, we are not interested in each individual object, but in classes of objects. As an example, consider a geographic information system that stores n points representing industrial plants. Each such point is coloured, such that the colour indicates the type of plant (chemical, manufacturing, food, energy, etc.). Now one can ask questions like: ”What types of industry are found within a 10×10 kilometer square around a particular town?”. Such questions can be answered by standard data structures for orthogonal range searching: we first find all factories that lie inside our square query range, and then check all of those factories to see what different colours they have. Thus we get the answer in time $O(\log n + k')$, where k' is the number of plants inside our query range. However, when our square contains a lot of factories of only few different colours, checking all factories one by one seems to be a waste of time. It would be much nicer to have a data structure that can give the answer in time $O(\log n + k)$, where

k is the number of different colours to report. Such data structures have been developed indeed: in one dimension it can be achieved with $O(n)$ storage [Jan93], in two dimensions with $O(n \log^2 n)$ storage [Jan93], and in higher dimensions with $O(n^{1+\delta})$ storage [Aga02], for an arbitrarily small constant δ.

Still, this is not always satisfactory. If the query area is big enough, it can be that virtually every colour has at least one point in the query range. But in some applications, we are interested only in the colours that have a *significant* presence in the area. The meaning of *significant presence* can be determined by several criteria, depending on the application, for example:

- report only the colour that is found most frequently in the area;
- report only colours that have at least a certain number of points in the area;
- report only colours such that at least a certain percentage of the points in the area have that colour;
- report only colours such that at least a certain percentage of all points of that colour lie in the area.

So far, we do not know of any results on this type of range searching problems, except for Krizanc et al. [Krz03], who recently studied the first criterion in the one-dimensional case.

Results in this thesis

In Chapter 2, *"Significant-presence range queries in categorical data"*, we study the problem with the last criterion mentioned above. For the one-dimensional case, we present a data structure that uses $O(n)$ storage, and can answer significant-presence queries in $O(\log n + k)$ time, where k is the number of reported colors. Unfortunately, the generalization of our approach to higher dimensions leads to a data structure that uses cubic storage already in the two-dimensional case. To show this fact, we obtain the following result which is of independent interest. Let P be a set of n points in $\mathbb{R}^d$, and t a parameter with $1 \leqslant t \leqslant \frac{n}{2d}$. In the worst case, the maximum number of combinatorially distinct boxes containing exactly t points from P is $\Theta(n^d t^{d-1})$.

Because exact significant-presence queries appear to be difficult, we also study a relaxed version of the problem, where we are also allowed to report colours whose presence is somewhat below the treshold. For example, instead of only reporting all colours of which at least 50% of the points are inside the range, we may also report some extra colours that have between 40% and 50% of the points inside the range. The size of the data structure for this type of queries depends on the number of colours, the treshold and on the required precision, but, surprisingly, not on the total number of points. For details, see Chapter 2.

1.1.2 Data structures for object data: bounding-volume hierarchies

As explained before, designing efficient data structures becomes significantly more difficult if the objects stored are not points, but objects that have some shape and size, such as line segments, balls or polyhedra. Theoretically efficient solutions for such problems are often too complicated and bear too much overhead to be useful in practice. It becomes even more difficult if we want a data structure that supports multiple types of queries at the same time. One can cheat, of course, by just taking a few data structures together and store each object multiple times: once in each structure. But this increases the storage requirements and also puts on us the burden of maintaining several structures.

In practice, so-called *bounding-volume hierarchies* often provide a good solution. They are easy to implement, and although a bounding-volume hierarchy for n objects does not store more than $2n$ pointers and geometric objects, it can be used for different types of queries. A query in a bounding-volume hierarchy does not go directly for the answer to the query; rather it generates a set of candidate answers, which then need to be checked one by one. In practice, the set of candidate answers is usually small enough to make this approach efficient. For a bounding-volume hierarchy to be useful, it should allow fast generation of candidate answers, and it should select the candidates such that they are likely to be true answers.

Below, I will first explain what a bounding-volume hierarchy is and how it is used. After that, I will explain what issues have to be addressed when designing a bounding-volume hierarchy. I will then focus on a particular class of bounding-volume hierarchies, namely R-trees, and give an overview of our results on R-trees. To conclude, I will suggest a few subjects for further research in this area.

Definition and usage

A bounding-volume hierarchy is a tree structure on a set of geometric objects (the data objects). Each object is stored in a leaf of the tree. Each internal node stores for each of its children ν an additional geometric object $V(\nu)$, that encloses all data objects that are stored in descendants of ν. In other words, $V(\nu)$ is a bounding volume for the descendants of ν. For an example, see Figure 1.1.

Bounding-volume hierarchies can be used to do many types of queries on the set of data objects. For example, the algorithm in Figure 1.2 finds all objects that intersect a query range Q and are stored in descendants of node ν. To find all data input objects that intersect Q, start the algorithm with the root of the hierarchy as ν. The query will then descend into the tree, visiting exactly those nodes whose bounding volumes intersect Q. The bounding-volume hierarchy can also be used for other types of queries, such as nearest-neighbour queries (see Figure 1.3).

The algorithms can easily be adapted to hierarchies with leaves that store multiple data objects.

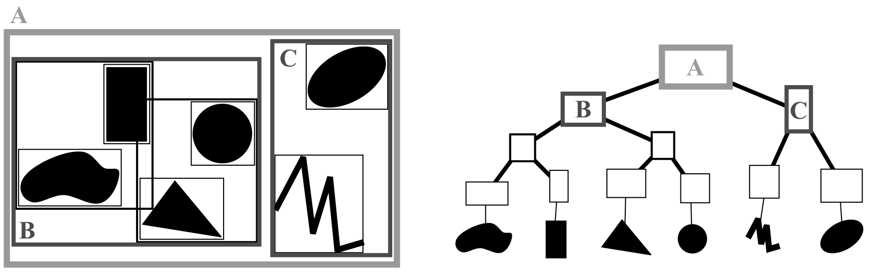


Figure 1.1: Example of a bounding-volume hierarchy, using rectangles as bounding volumes.

```
**Algorithm** *Intersected* ($Q$,$\nu$)
 1. for every child $\mu$ of $\nu$
 2.     if $V(\mu)$ intersects $Q$ then
 3.         if $\mu$ is a leaf then    {object $M$ stored in $\mu$ is a candidate answer}
 4.             if $M$ intersects $Q$ then
 5.                 report $M$
 6.         else
 7.             Intersected ($Q$,$\mu$).
```

Figure 1.2: Finding all objects that intersect Q. To find all objects that lie *completely* inside Q, replace the intersection test in line 4 by a test if M lies inside Q. To find all objects that completely *contain* Q, replace the tests in line 2 and 4 by a test if Q is completely contained in $V(\mu)$, or in M, respectively.

```
Algorithm Closest ($Q$,$\nu$)
 1. smallestDistanceFoundSoFar $\leftarrow \infty$
 2. $P \leftarrow$ an empty priority queue
 3. repeat
 4.     if $\nu$ is a leaf then      { the object $N$ stored in $\nu$ is a candidate answer }
 5.         if distance between $N$ and $Q <$ smallestDistanceFoundSoFar then
 6 .            smallestDistanceFoundSoFar $\leftarrow$ distance between $N$ and $Q$
 7 .            closestObject $\leftarrow N$
 8 .    else
 9 .        for every child $\mu$ of $\nu$
11.             insert $\mu$ in $P$ with priority (distance between $V(\mu)$ and $Q$)
12.     $\nu \leftarrow$ the node with lowest priority in $P$; let $p$ be its priority
13.     remove $\nu$ from $P$
14. until $p >$ smallestDistanceFoundSoFar or $P$ is empty
15. return closestObject.
```

Figure 1.3: Finding the object closest to Q.

Designing bounding-volume hierarchies

When designing a bounding-volume hierarchy, we have to decide what kind of bounding volumes to use, what the structure of the hierarchy should look like, and how to order the data objects in the tree.

The shape of the bounding volumes. The choice of bounding volume is determined by a trade-off between two objectives. On one hand, we would like to use bounding volumes that have a very simple shape. Thus, we need only few bytes to store them and intersection tests and distance computations are simple and fast. On the other hand, we would like to have bounding volumes that fit the corresponding data objects very tightly. Thus, we try to avoid going into subtrees that will not lead to any object that satisfies our query. On one extreme, we could use the full space as the bounding volume for everything. On the other extreme, we would use the union of the data objects as their bounding volume. Both extremes are pointless. In the first case we would traverse the complete tree for every query; in the second case, intersection tests would be just as complex as doing a complete query.

In practice, the most commonly used bounding volume is an axis-parallel (hyper-)rectangle—we will just call them boxes. The minimum (best-fit) bounding box for a given set of data objects is easy to compute, needs only few bytes of storage, and robust intersection tests are easy to implement and extremely fast. Experiments have been done with a number of other shapes though. Among them are the set-theoretic difference of two boxes [Ary00], oriented—that is: non-axis-aligned—bounding boxes [Bar96, Got96], spheres [Oos90] (with little success), the intersection of a box and a sphere [Kat97], the Minkowski sum of a box and a sphere [Lar00], a circular section of a spherical shell [Krs98], pie slices [Bar96], and discretely oriented polytopes (k-DOP's) [Jag90, Klo98], for example octagons [Sit99] or bounded aspect ratio k-DOP's [Dun99].

Circles and spheres seem to leave too little freedom to adjust the shape to fit the objects inside. But some of the more complex shapes might actually work well. It is difficult to get a clear picture from comparitive studies on this issue. Some authors who compared axis-aligned bounding boxes with discretely-oriented octagons (in two dimensions) or oriented bounding boxes (in three dimensions) reported that in the end, axis-aligned bounding boxes often seem to work better, despite the bad fit to the data; see Van den Bergen [Bgn99] and Sitzmann and Stuckey [Sit99]. Sitzmann, however, also reported positive results for octagon hierarchies on data consisting of randomly oriented line segments. The right type of bounding volume might, in fact, depend on the input: some of the non-standard bounding volumes are specifically aimed at fitting the triangles used to approximate smooth surfaces in virtual reality applications. Finding the right type of bounding volume definitively remains as a subject for futher study.

In our research, we decided to try to establish the best performance that can be achieved with axis-aligned bounding-box hierarchies, both from a theoretical and from a practical point of view.

The structure of the hierarchy. Since a bounding-volume hierarchy is a tree structure by definition, the main choice left is to decide on the degree of the nodes, that is: the number of children and/or input objects stored in a node. The optimal degree depends on the way in which the bounding-volume hierarchy is used. The cost of processing a node in the hierarchy is composed of the costs of accessing the location of the node in memory, the cost of reading the node's children pointers and their bounding volumes, and the cost of the intersection or distance computations on those bounding volumes. If the hierarchy is stored on disk, the access cost tends to be high: the disk's head must be moved to the correct physical location. Once the disk head is at the correct position, a complete block of data is read into main memory at once. Computations on those data are relatively cheap, since these are done in main memory. Therefore, high-degree nodes that fill a full block of data are preferred. On the other hand, if the hierarchy is stored in main memory, our main concern is to keep the number of intersection or distance computations down. For queries that do not yield too many answers, this is best achieved by making many low-degree nodes. For example, two nodes that are irrelevant to our query can often be skipped faster if we construct a parent node that gets these two nodes as its children. A single distance computation on the parent's bounding volume may then reveal that we can skip its two children without doing distance computations on them. Of course, this potential advantage of having many low-degree nodes only materializes if usually, the parent node will indeed be skipped if both of its children are, and usually, we do not need to go into both children after all. Whether or not this is actually the case depends on the data, the queries, and the way in which the data objects are distributed in the tree.

Another issue with regard to the structure of the hierarchy is its height. If we want to be able to go from the root to any data object fast, small height is a necessary condition, but not a sufficient one. The main problem is that the bounding volumes of a node's children may intersect. If the object lies inside their intersection, there is no way to tell which child has the object as a descendant. However, small height may still be useful to guarantee that update algorithms can run fast. Most algorithms to insert or delete an object run in time $O(h)$, where h is the height of the tree. Small height is most easily guaranteed by requiring that all leaves are at the same depth. This a sufficient, but not a necessary condition, to guarantee that the tree has height $O(\log_t n)$, where n is the number of objects stored, and t is the minimum degree of the nodes.

The distribution of the objects in the hierarchy. Finally, the way in which the objects are distributed in the hierarchy may have a huge impact on its performance. One of the major issues is that overlap between bounding volumes in the same node can make search paths branch and spread out into large parts of the hierarchy. Therefore, it is important to keep the amount of overlap small. Unfortunately, overlap cannot be avoided completely. Points can always be distributed among the different parts of the hierarchy in such a way that the bounding boxes in a node do not overlap, but with other objects this is not always possible.

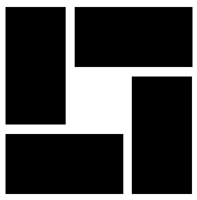


Figure 1.4: A set of rectangles for which an overlap-free hierarchy of degree two is impossible.

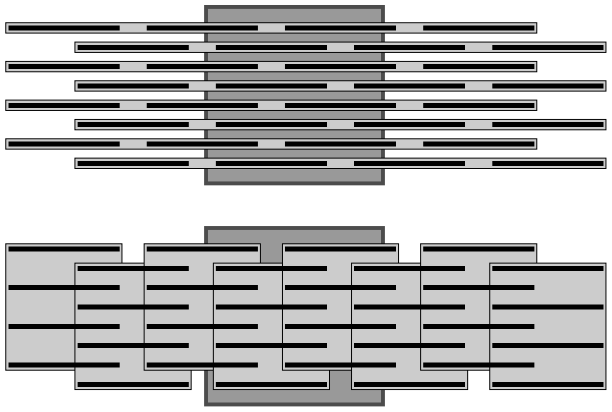


Figure 1.5: Minimizing overlap does not always lead to optimal query efficiency.

Figure 1.4 shows a set of rectangles that does not admit of an overlap-free hierarchy of bounding rectangles (or other convex bounding volumes) with nodes of degree two. The only way to avoid overlap is to cut data objects into smaller parts (clipping), but this comes at a cost: it would take more storage space, and while collecting the answers to a query, time may be wasted retrieving pointers to objects which we had found already through another part.

Moreover, minimizing the amount of overlap does not necessarily lead to optimal query efficiency, as is illustrated by the following example. In Figure 1.5, we divided the line segments into groups of four: each group corresponds to a node just above leaf level in a hierarchy with nodes of degree four. In the top figure, we did the grouping so that we minimize the overlap between the bounding boxes of the nodes. A query with the grey square will visit 8 nodes on this level. In the lower figure, the line segments are grouped in another way. A query with the grey square will now visit only 4 nodes on this level.

If minimizing overlap is not enough to guarantee optimal queries, then how should we distribute or group the objects in the hierarchy? It is this issue that is the subject of Chapters 3 to 5 of this thesis.

R-Trees

We restricted our study to hierarchies that use axis-parallel boxes as bounding volumes. Extending the study to other types of bounding volumes is an obvious subject for further research but it lies beyond the scope of our work. Bounding-volume hierarchies that use axis-parallel boxes as bounding volumes and have nodes of high degree are also known as R-trees. The R-tree was originally introduced by Guttman [Gut84]. His study has inspired two decades of research about how to distribute the data objects in an R-tree, some authors designing new distribution algorithms from scratch, others suggesting optimization heuristics to be used in conjunction with known methods.

R-trees are parametrized by the maximum degree of the nodes, denoted by t in this chapter. This parameter is set to match the characteristics of the hardware used: usually the tree is stored on disk, and t is chosen such that a node fills a full block on the disk. For in-memory applications, smaller values of t would be used. The minimum degree of the nodes is set to a fixed fraction of t; in the R-tree variants studied it ranged from 10% to 50% of t. R-trees usually store data objects in the leaves only and have all leaves on the same level in the tree, although some authors have designed variants where this is not the case (e.g. [Agg97, Kan97, Ros01]).

Essentially three types of algorithms have been designed to distribute the objects in an R-tree:

by repeated insertion: One defines an insertion algorithm that strives to optimize the tree locally; a complete tree is built by inserting the data objects one by one, e.g. [Ang97, Bkr92, Bmn90, Grc98b, Ros01]. Usually, deletion algorithms are provided as well.

by recursive partitioning: One defines an algorithm to distribute any number of data objects among up to t subtrees; the tree is built by applying the partitioning algorithm recursively top-down, e.g. [Agg97, Grc98a, Whi96]. The resulting data structure can be maintained either by using insertion and deletion heuristics as above (and, for example, rebalancing the complete tree during quiet hours), or by using the logarithmic method [Aga01APV].

by linear ordering: One defines a function that maps each data object to a one-dimensional value; the tree can then be built and maintained as a standard B-tree that uses the function values as keys [Brg00, Bhm99, Kam93, Kam94].

For an extensive survey on R-trees, see Manolopoulos et al. [Man03].

When comparing the query efficiency of R-trees built by such algorithms, one should distinguish between a static environment (the tree is built once and not changed afterwards), and a dynamic environment (the tree is continuously updated). In a dynamic environment it may be very difficult to maintain an “ideal” distribution of objects over the tree. The insertion of an object can, in principle, change the ideal distribution a lot. To allow for reasonably efficient update operations, one has to relax the ideal a bit. As a result, static trees, built with a partitioning or a linear-ordering algorithm, usually allow for more efficient queries than their dynamic counterparts or insertion-based algorithms.

Despite the huge body of research on R-trees, until recently, very little was known about the query times that can be guaranteed for worst-case data and queries. From Kanth and Singh [Kan98] and De Berg et al. [Brg00] some lower bounds for intersection queries with axis-parallel rectangles were known: query times better than $\Omega((n/t)^{1-1/d} + k/t)$ can, in general, not be guaranteed. Here n is the number of data objects, t is the degree of the nodes, k is the number of object bounding boxes intersected, and d is the number of dimensions. There were no algorithms to construct R-trees that can guarantee to do any query faster than a full traversal of the complete hierarchy, even if there are no answers to be reported. The only results in that direction were by De Berg et al. [Brg00], but they could guarantee fast queries only for relatively small query ranges (for details see the introduction of Chapter 3). Other research on R-trees was mainly experimental, or of a statistical nature, making statements about expected query times under certain assumptions on the distribution of the data and/or the queries. To our knowledge, the algorithms presented in Chapter 3 and 4 are the first algorithms that construct R-trees that guarantee worst-case query times better than $\Omega(n)$ for all axis-parallel rectangle range queries.

Note that in the bound mentioned above, as well as in all results mentioned below, k is not the number of objects intersected, but the number of data object *bounding boxes* intersected. If k would be the number of data objects intersected, it would be very difficult to prove anything about the efficiency of R-trees. Even if the objects are disjoint, their bounding boxes may in the worst case intersect in a single point, leading to a query time of $\Omega(n/t)$. For an example, see Figure 1.6. In three dimensions, there are sets of line segments such that *any* hierarchy of *convex* bounding volumes on such a set, needs $\Omega(n/t)$ time to answer a query with an axis-parallel line in the worst case [Bar96]. However, even if the objects intersected cannot be identified efficiently in the worst case, this is no reason to give up on at least identifying the object *bounding boxes* intersected efficiently. From now on, we will assume that the data objects stored in our hierarchies are in fact bounding boxes, and k will be the number of such bounding boxes intersected by the query range.

Results in this thesis

Given the fact that we use axis-parallel boxes as bounding volumes and given the maximum degree of the nodes, we set out to optimize the structure of the tree for fast intersection queries. We chose to optimize for intersection queries since such queries are an important application to start with, and they are indicative of the efficiency of some other types of queries as well. For example, queries for objects intersecting a rectangle and queries for objects contained in a rectangle visit exactly the same nodes, and nearest-neighbour queries with a point visit exactly those nodes which would be visited by a intersection query with a

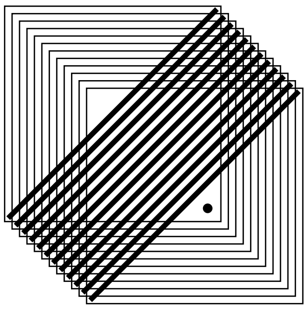


Figure 1.6: All bounding boxes of these line segments overlap in a single point. A query with that point needs to examine the complete hierarchy.

circle centered on that point and just touching the nearest neighbour. Therefore, a good performance on intersection queries is crucial and can be expected to be a good indication of the performance of several other types of queries. To avoid redundancy in the data structure, we excluded clipping variants from our studies.

Our research has led to three articles: *"Box-trees and R-trees with near-optimal query time"* (Chapter 3), *"The Priority R-Tree: a practically efficient and worst-case-optimal R-tree"* (Chapter 4), and: *"Box-trees for collision checking in industrial installations"* (Chapter 5). All of them are, in the first place, about static R-trees, that is: R-trees that are not updated anymore, once built (although Chapter 4 also discusses updates with the logarithmic method).

In Chapter 3 we prove that there are sets of rectangles, such that in *any* R-tree on such a set, there are queries that yield no answers but nevertheless visit $\Omega((n/t)^{1-1/d})$ nodes. It is not so much this bound itself which is interesting: it was already known from Kanth and Singh [Kan98] and De Berg et al. [Brg00]. What is interesting is the type of data that can bring out this worst-case behaviour. We show that such worst-case sets of rectangles and queries exist even if any one or two of the following restrictions apply:

- no point is contained in the bounding boxes of more than a constant number of data rectangles (in other words: they don't overlap much);

- the aspect ratio of the query rectangles is bounded by a constant (in other words: the query rectangles are not extremely long and thin);

- we have only two dimensions.

Only if all three of these restrictions apply, we cannot do our lower bound construction. In fact, for that case we show how to construct R-trees that can answer any axis-parallel rectangle query by visiting $O(\log^2 n + k)$ nodes.

Note that all our lower bounds, like the previous bounds by Kanth and Singh [Kan98] and De Berg *et al.*[Brg00], do not hold for replicating data structures, that is, data structures that may store each object (or a pointer to it) more than once.

In Chapter 3 we also give an algorithm for the construction of axis-aligned-bounding-box hierarchies with nodes of degree two that achieves optimal query time $\Theta(n^{1-1/d} + k)$ in the general case. In Chapter 4 we extend this method to get optimal $\Theta((n/t)^{1-1/d} + k/t)$ query time on nodes of degree t (assuming that I/O-operations dominate). Chapter 4 describes the method for nodes of degree t in detail; it is not necessary to read Chapter 3 first to understand it. Chapter 4 also presents experimental results in two dimensions. The results indicate that our algorithm creates R-trees that are efficient in practice, while being more robust than the heuristic approaches known so far.

One may wonder if it is possible to construct R-trees that combine the good properties of both constructions mentioned above: get $O((n/t)^{1-1/d}+k/t)$ query in the general case, and $O(\log^2 n + k)$ query time if the three restrictions mentioned above apply. In Chapter 3 we describe a construction, called kd-interval tree, that goes a long way towards achieving this goal. A kd-interval tree in two dimensions answers axis-parallel range queries in time $O(\sqrt{\frac{n}{t}} + k)$ and point queries in time $O(\log^2 + k)$, provided that the data rectangles don't overlap much. As overlap among the data rectangles increases, the point query performance degenerates gradually into $O(\sqrt{\frac{n}{t}} + k)$. One could use similar techniques as in Chapter 4 to get a better dependency on the degree t in the k-term.

The lower bound constructions in Chapter 3 show that it is not possible to achieve something similar in more than two dimensions: there are sets of disjoint data boxes that make any R-tree that guarantees polylogarithmic query times for point queries, spend near-linear time on some (hyper-)cube queries.

In Chapter 5, we look into the three-dimensional situation further. The data boxes in the lower-bound construction mentioned above do not look extremely strange: they can be arbitrarily close to a unit cube in shape and size. There is one peculiar thing about them though: they must be arranged in such a way that certain cubic query ranges yield no answers while there are a *lot* of data boxes nearby. It turns out that if we accept that such cases are difficult (but probably rare in practice), and if we accept that certain arrangements of extremely flat data boxes are difficult (but probably rare in practice), we can build a three-dimensional kd-interval-tree with polylogarithmic query time for the remaining cases (the cases we expect to find in practice). We prove that these query times are achieved not only for queries with boxes but also for queries with other query ranges of constant complexity. Chapter 5 describes how to build a tree with nodes of low degree; one may use the transformation algorithms described in Chapter 3 to transform the tree into a real R-tree with high-degree nodes.

To distinguish between arrangements of boxes that should be handled efficiently, and arrangements of boxes that may be considered difficult, we define the *slicing number* of a set of data objects as follows: let the slicing number λ_C with respect to a cube C be the maximum number of data object bounding boxes that intersect four parallel edges of C; then the overall slicing number λ is the maximum value of λ_C over all possible cubes C. A low slicing number means that the data boxes do not overlap much and that there are no arrangements of lots of extremely flat data boxes very close to each other.

The main results for point and axis-aligned rectangle queries can be summarized in the following tables. We use the following notation:

n: the total number of data objects in the hierarchy;

k: the number of data object bounding boxes that intersect the query range;

k_ε: (with $\varepsilon > 0$) the number of data object bounding boxes that intersect the query range, or lie within a distance of ε times the diameter of the query range;

t: the maximum degree of the nodes in the hierarchy;

α: the maximum aspect ratio (width/height or height/width) of the query range;

Results in two dimensions: asymptotic upper bounds $O(...)$			
input rectangles	*tree (chapter)*	*point queries*	*rectangle queries*
disjoint	2D kd-interval (3)	$\log^2 n$	$\sqrt{\frac{n}{t}} + k$
disjoint	2D kd-interval+lsf (3)	$\log^2 n$	$\alpha \log^2 n + k$
intersecting	PR (4)	$\sqrt{\frac{n}{t}} + \frac{k}{t}$	$\sqrt{\frac{n}{t}} + \frac{k}{t}$

Results in three dimensions: asymptotic upper bounds $O(...)$			
input boxes	*tree (chapter)*	*point queries*	*box queries*
constant slicing nr.	3D kd-int.+lsf (5)	$\log^4 n$	$\min_\varepsilon\{(\frac{1}{\epsilon})^2 \log^4 n + k_\epsilon\}$
intersecting	PR (4)	$(\frac{n}{t})^{2/3} + \frac{k}{t}$	$(\frac{n}{t})^{2/3} + \frac{k}{t}$

Subjects for further research

The 3-dimensional kd-interval-tree mentioned above has good theoretical bounds for low-degree nodes, but when turned into an R-tree (using the technique explained in Chapter 3), the dependency on the degree of the nodes is not as good as one would wish. We cannot yet say if data sets of realistic size and structure will nevertheless bring out the strength of the kd-interval-tree, and if so, for what types of data and queries this method would indeed be the method of choice.

In Chapter 4 we compare our PR-tree to two variants of the Hilbert-R-tree, which is an R-tree based on ordering objects along the Hilbert space filling curve [Kam94]. Although the Hilbert-R-tree cannot guarantee worst-case query times, and does not outperform the PR-tree, it still has advantages: it is built faster and it is much easier to implement and maintain. We tested two variants of the Hilbert-R-tree in two dimensions: one in which each data object is represented by its center point, and one in which each data object is represented by a *four*-dimensional point whose coordinates are those of the object's bounding rectangle. Naturally, the second variant is more robust when the data consists of rectangles. However, the experiments also show that the second variant is *weaker* on some sets of *point* objects. It makes one wonder if this unwanted behaviour cannot be avoided. Can we design a space-filling curve, to be used as the basis for an R-tree, which is good for both point and rectangle data?

The next big question that remains is: what is the best type of bounding volume? It might depend on the type of queries we want to perform. Are axis-aligned bounding boxes the best choice for axis-aligned rectangle queries? What would be the best bet for general range queries in two dimensions? Do the results on octagons by Sitzmann and Stuckey [Sit99] suggest that octagons, sometimes helpful, sometimes harmful, are just a little bit too much? Would the optimum be found at discretely oriented hexagons? And how would that be in three dimensions? Dodecahedra?

Our research has been primarily aimed at two- and three-dimensional settings. Our theoretical results *are* valid for multi-dimensional data as well. Unfortunately, this includes the rather disappointing lower bounds. From this we must conclude that the theoretical approach taken in this thesis, aiming for optimal worst-case query times, may not give us a data structure that is practical for high-dimensional data. In practice, one would like to have a data structure that does not only guarantee optimal query times on the worst possible data, but can also take advantage of easier data to allow for faster queries. Since in many practical situations, we do not have worst-case data, this would lead to a data structure that is much faster in practice. We do not know if our data structures take advantage of easy data or fail to do so. For two-dimensional data, it worked out well—in our experiments, the PR-tree does appear to be efficient—but this success does not necessarily carry over to higher dimensions. Handling high-dimensional data may require more study into questions of the type: what *is* easy data, and how can we design a data structure that simultaneously guarantees worst-case query times and takes advantage of easy data? Chapter 5 is an attempt to deal with the first question in three dimensions, but it is doubtful if it makes sense to generalize the approach of Chapter 5 to higher dimensions. The right questions to ask may depend on the number of dimensions. Typical applications for low-dimensional data include motion planning. There we have objects that may have a shape in, for example, four dimensions (three spatial dimensions and one time dimension). But high-dimensional data more often comes from applications where the data objects have no shape and size, but are just points whose coordinates represent the values of non-geometric properties of the objects.

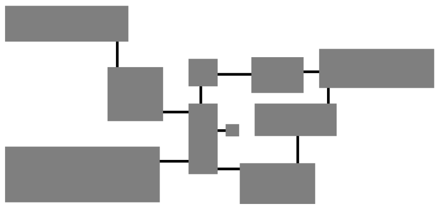


Figure 1.7: Example of a geometric network. In this case, the nodes are axis-parallel rectangles, and the connections are axis-parallel line segments.

1.2 Geometric networks

A network consists of a set of vertices or nodes, and a set of connections between the nodes. In a geometric network, the nodes and connections are geometric objects, which usually have a place and size in space but can also have non-geometric properties such as the cost of construction and the time needed to traverse a connection. Geometric networks arise frequently in our everyday life: road networks, telephone networks, and computer networks are all examples of geometric networks that we use daily. They also play a role in disciplines such as VLSI (chip) design and motion planning. Almost invariably, the purpose of the network is to provide a connection between the nodes in the network. Often it is desirable that the connection through the network between any pair of nodes is relatively short. From this viewpoint, one would ideally have a direct connection between any pair of nodes. This is usually infeasible due to the costs involved, so one has to compromise between the quality and the cost of the connections. This leads to optimization problems of the following form: find the "best" set of connections for a given set of geometric objects, subject to a given set of constraints. A well-known problem of this type is to find the minimum spanning tree on a set of points, which is a set of connections of minimum total length such that the network is connected. There are many different combinations of constraints and criteria for a good network: for example, they may concern the total cost of the network, the maximum distance between nodes, the degree of the nodes, or the number of nodes that may get disconnected if a connection fails. There is a vast literature on this type of problems, and a comprehensive overview would be beyond the scope of this thesis. For a survey, see e.g. Eppstein [Epp00], or the upcoming book by Narasimhan and Smid [Nar04]. We studied two problems in this area, as described below.

Results in this thesis

In Chapter 6, *"Facility location and the geometric minimum-diameter spanning tree"*, we study the following case. The nodes in our network are points in two dimensions, and a straight line connection between two points costs the same regardless of the distance. We are given just enough money to make a network that connects each point. So if there are n points, we can make only $n-1$ connections. In other words: we have to construct a spanning tree on the nodes. Our goal is to choose our connections such that the biggest distance between any pair of points through the network is as small as possible. So far, exact solutions take almost $O(n^3)$ time to compute. Our algorithm finds a solution where the maximum distance is up to a factor $(1+\varepsilon)$ larger than in the best solution, but it finds such a solution in time $O((\frac{1}{\varepsilon})^5 + n)$. For large sets of points and modest requirements on the accuracy, this can be much faster than the best known exact solution.

In Chapter 7, *"Optimal spanners for axis-aligned rectangles"* we study the following case. The nodes in our network are n axis-aligned rectangles in two dimensions; they are to be connected by line segments. We are given a graph on the nodes of the network. For every connection given in the graph, an axis-aligned line segment must be placed between the corresponding rectangles. What we have to figure out, is where exactly the line segment between a pair of rectangles should be placed. Imagine the rectangles are buildings, and the line segments are foot bridges. It is quite frustrating if, to walk to a room opposite one's own room in an adjacent building, one has to walk all the way to the end of a long corridor, then along the foot bridge, and then back again along the corridor in the other building. Therefore our goal is to place the line segments such that we minimize the maximum dilation, that is the worst possible ratio of the indoor distance and the shortest distance for any pair of points inside the rectangles. Our results are as follows.

- In general, the problem is NP-hard.
- If the bridge graph is a tree, then the problem can be solved by a linear program with $O(n^2)$ variables and constraints.
- If the bridge graph is a path, then the problem can be solved in $O(n^3 \log n)$ time.
- If the bridge graph is a path and the buildings are sorted vertically along this path, the problem can be solved in $O(n^2)$ time. A $(1+\varepsilon)$-approximation, where the maximum dilation is at most $(1+\varepsilon)$ times the optimum, can be computed in $O(n \log \frac{1}{\varepsilon})$ time.

This leaves the question of a strongly polynomial-time algorithm for the tree case still open: so far, we did not succeed in finding a way of generalizing the path algorithm to arbitrary trees. We hope that the insight gained from the research in Chapter 7 will eventually be helpful in finding good solutions when the bridge graph is not given and/or when we wish to connect arbitrary convex polygons rather than axis-aligned rectangles.

Chapter 2

Significant-presence range queries in categorical data

An extended abstract of this chapter was previously published as: Mark de Berg and Herman J. Haverkort: Significant-presence range queries in categorical data, in *Proc. 8th Int. Workshop on Algorithms and Data Structures (WADS)*, Ottawa, 2003 (LNCS 2748), pages 462–473. Full text submitted to *Journal of Algorithms*.

Abstract. *In traditional colored range-searching problems, one wants to store a set of n objects with m distinct colors for the following queries: report all colors such that there is at least one object of that color intersecting the query range. Such an object, however, could be an 'outlier' in its color class. Therefore we consider a variant of this problem where one has to report only those colors such that at least a fraction τ of the objects of that color intersects the query range, for some parameter τ. Our main results are on an approximate version of this problem, where we are also allowed to report those colors for which a fraction $(1-\varepsilon)\tau$ intersects the query range, for some fixed $\varepsilon > 0$. We present efficient data structures for such queries with orthogonal query ranges in sets of colored points, and for point stabbing queries in sets of colored rectangles.*

2.1 Introduction

Motivation. The range-searching problem is one of the most fundamental problems in computational geometry. In this problem we wish to construct a data structure on a set S of objects in $\mathbb{R}^d$, such that we can quickly decide for a query range which of the input objects it intersects. The range-searching problem comes in many flavors, depending on the type of objects in the input set S, on the type of allowed query ranges, and on the required output (whether one wants to report all intersected objects, to count the number of intersected objects, etc.). The range-searching problem is not only interesting because it is such a fundamental problem, but also because it arises in numerous applications in areas like databases, computer graphics, geographic information systems, and virtual reality. Hence, it is not surprising that there is an enormous literature on the subject—see for instance the surveys by Agarwal [Aga97], Agarwal and Erickson [Aga98E], and Nievergelt and Widmayer [Nie00].

In this paper, we are interested in range searching in the context of databases. Here one typically wants to be able to answer questions like: given a database of customers, report all customers whose ages are between 20 and 30, and whose income is between \$50,000 and \$75,000. In this example, the customers can be represented as points in $\mathbb{R}^2$, and the query range is an axis-parallel rectangle.[1] This is called the (planar) *orthogonal range-searching problem*, and it has been studied extensively—see the surveys [Aga97, Aga98E, Nie00] mentioned earlier.

There are situations, however, where the data points are not all of the same type but fall into different categories. Suppose, for instance, that we have a database of stocks. Each stock falls into a certain category, namely the industry sector it belongs to—energy, banking, food, chemicals, etc. Then it can be interesting for an analyst to get answers to questions like: "In which sectors companies had a 10–20% increase in their stock values over the past year?" In this simple example, the input can be seen as points in 1D (namely for each stock its increase in value), and the query is a 1-dimensional range-searching query.

Now we are no longer interested in reporting all the points in the range, but in reporting only the categories that have points in the range. This means that we would like to have a data structure whose query time is not sensitive to the total number of points in the range, but to the total number of categories in the range. This can be achieved by building a suitable data structure for each category separately, but this is inefficient if the number of categories is large. This has led researchers to study so-called *colored range-searching problems*: store a given set of colored objects—the color of an object represents its category—such that one can efficiently report those colors that have at least one object intersecting a query range [Aga02, Kre92, Gup95, Jan93].

We believe, however, that this is not always the correct abstracted version of the range-searching problem in categorical data. Consider for instance the stock example sketched earlier. The standard colored range-searching data structures would report all sectors that have *at least one* company whose increase in stock value lies in the query range. But this does not necessarily say anything about how the sector is performing: a given sector could be doing very badly in general, but contain a single 'outlier' whose performance has been good. It is much more natural to ask for all sectors for which *most* stocks, or at least a significant portion of them, had their values increase in a certain way. Therefore we propose a different version of the colored range-searching problem: given a fixed threshold parameter τ, with $0 < \tau < 1$, we wish to report all colors such that at least a

[1]From now on, whenever we use terms like "rectangle" or "box" we implicitly assume these are axis-parallel.

fraction τ of the objects of that color intersect the query range. We call this a *τ-significant-presence query*, as opposed to the standard *presence query* that has been studied before.

Problem statement and results. We study significant-presence queries in categorical data in two settings: orthogonal range searching where the data is a set of colored points in $\mathbb{R}^d$ and the query is a box, and stabbing queries where the data is a set of colored boxes in $\mathbb{R}^d$ and the query is a point. We now discuss our results on these two problems in more detail.

Let $S = S_1 \cup \cdots \cup S_m$ be a set of n points in $\mathbb{R}^d$, where m is the number of different colors and S_i is the subset of points of color class i. Let τ be a fixed parameter with $0 < \tau < 1$. We are interested in answering τ-significant-presence queries on S: given a query box Q, report all colors i such that $|Q \cap S_i| \geqslant \tau \cdot |S_i|$. For $d = 1$, we present a data structure that uses $O(n)$ storage, and that can answer significant-presence queries in $O(\log n + k)$ time, where k is the number of reported colors. Unfortunately, the generalization of our approach to higher dimensions leads to a data structure using already cubic storage in the planar case. To show this fact, we obtain the following result which is of independent interest. Let P be a set of n points in $\mathbb{R}^d$, and t a parameter with $1 \leqslant t \leqslant n/(2d)$. Then the maximum number of combinatorially distinct boxes containing exactly t points from P is $\Theta(n^d t^{d-1})$ in the worst case.

As a data structure with cubic storage is prohibitive in practice, we study an approximate version of the problem. More precisely, we study *ε-approximate significant-presence queries*: here we are required to report all colors i with $|Q \cap S_i| \geqslant \tau \cdot |S_i|$, but we are also allowed to report colors with $|Q \cap S_i| \geqslant (1-\varepsilon)\tau \cdot |S_i|$, where ε is a fixed positive constant. For such queries we develop a data structure that uses $O(M^{1+\delta})$ storage, for any $\delta > 0$, and that can answer such queries in $O(\log n + k)$ time, where $M = m/(\tau^{2d-2}\varepsilon^{2d-1})$ and k is the number of reported colors. We obtain similar results for the case where τ is not fixed, but part of the query—see Theorem 2.2.7. Note that the amount of storage does not depend on n, the total number of points, but only on m, the number of colors. This should be compared to the results for the previously considered case of presence queries on colored points sets. Here the best known results are: $O(n)$ storage with $O(\log n + k)$ query time for $d = 1$ [Jan93], $O(n \log^2 n)$ storage with $O(\log n + k)$ query time for $d = 2$ [Jan93], $O(n \log^4 n)$ storage with $O(\log^2 n + k)$ query time for $d = 3$ [Gup95], and $O(n^{1+\delta})$ storage with $O(\log n + k)$ query time for $d \geqslant 4$ [Aga02]. These bounds all depend on n, the total number of points; this is of course to be expected, since these results are all on the exact problem, whereas we allow ourselves approximate answers.

In the point-stabbing problem we are given a parameter τ and a set $B = B_1 \cup \cdots \cup B_m$ of n colored boxes in $\mathbb{R}^d$, and we wish, for a query point q, to report all colors i such that the number of boxes in B_i containing q is at least $\tau \cdot |B_i|$. We study the ε-approximate version of this problem, where we are also allowed to report colors such that the number of boxes containing q is at least

$(1-\varepsilon)\tau \cdot |B_i|$. Our data structure for this case uses $O(M^{1+\delta})$ storage, for any $\delta > 0$, and it has $O(\log n + k)$ query time, where $M = m/(\tau\varepsilon)^d$. The best results for standard colored stabbing queries, where one has to report all colors with at least one box containing the query point, are as follows. For $d = 2$, there is a structure using $O(n \log n)$ storage with $O(\log^2 n + k)$ query time [Gup95], and for $d > 2$ there is a structure using $O(n^{1+\delta})$ storage with $O(\log n + k)$ query time [Aga02].

2.2 Orthogonal range queries

Our global approach is to first reduce significant-presence queries to standard presence queries. We do this by introducing so-called *test sets*.

Test sets for orthogonal range queries

Let P be a set of n points in $\mathbb{R}^d$, and let τ be a fixed parameter with $0 < \tau < 1$. A set T of boxes—that is, axis-parallel hyperrectangles—is called a τ*-test set* for P if:

1. any box from T contains at least τn points from P, and
2. any query box Q that contains at least τn points from P fully contains at least one box from T.

We call the boxes in T *test boxes*. We can answer a significant-presence query on P by answering a presence query on T: a query box Q contains at least τn points from P if and only if it contains at least one test box. This does not yet reduce the problem to a standard presence-query problem, because T contains boxes instead of points. However, like Agarwal *et al.* [Aga02], we can map the set T of boxes in $\mathbb{R}^d$ to a set of points in $\mathbb{R}^{2d}$, and the query box Q to a box in $\mathbb{R}^{2d}$, in such a way that a box $b \in T$ is fully contained in Q if and only if its corresponding point in $\mathbb{R}^{2d}$ is contained in the transformed query box.[2] This means we can apply the results from the standard presence queries on colored point sets.

It remains to find small test sets. As it turns out, this is not possible in general: below we show that there are point sets that do not admit test sets of near-linear size. Hence, after studying the case of exact test sets, we will turn our attention to approximate test sets.

Exact test sets. Let t be a parameter with $1 \leqslant t \leqslant n$. Define a t*-box* to be a minimal box containing at least t points from P, that is, a box b containing at least t points such that there is no strictly smaller box $b' \subset b$ that contains t or more points. It is easy to see that any (τn)-box must be a test box, and that the

[2]In fact, the transformed query box is unbounded to one side along each coordinate-axis, so it is a d-dimensional 'octant'.

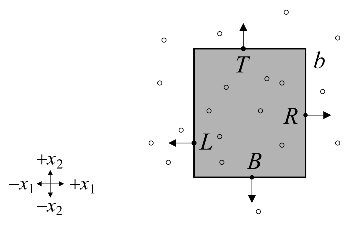


Figure 2.1: Peeling a (τn)-box b in two dimensions $(\tau n = 12)$. The black dots are the four points of $D(b)$. Initially, each point is extreme in only one direction, as indicated by the arrows. We can choose any of them, let us take T.

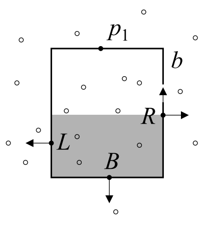


Figure 2.2: For p_2, we cannot take R, as it is extreme in two directions among the remaining points of $D(b)$. So we have to take one of the others, for example L.

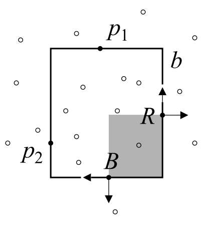


Figure 2.3: Now, all remaining points of $D(b)$ are extreme in 2 directions: we stop peeling here. R and B together form the basis $D^*(b)$ of b. We conclude that b has a peeling sequence of type $+x_2, -x_1$.

collection of all (τn)-boxes forms a τ-test set. Hence, the smallest possible test set consists exactly of these (τn)-boxes.

In the 1-dimensional case a box is a segment, and a minimal segment is uniquely defined by the point from P that is its left endpoint. This means that any set of n points on the real line has a test set that has size $(1-\tau)n+1$. Unfortunately, the size of test sets increases rapidly with the dimension, as the next lemma shows.

Lemma 2.2.1 *For any set P of n points in $\mathbb{R}^d$, there is a τ-test set that has size $O(\tau^{d-1}n^{2d-1})$. Moreover, for some sets P, any τ-test set has size $\Omega(\tau^{d-1}n^{2d-1})$.*

Proof: By the observation made before, bounding the size of a test set boils down to bounding the number of (τn)-boxes. In this proof, when we use the term direction we mean one of the $2d$ directions $+x_1, -x_1, ..., +x_d, -x_d$. Let b be a (τn)-box, and let $D(b)$ be a set of points in b such that there is at least one point of $D(b)$ on each facet of b. If there are more such sets, let $D(b)$ be a set with minimum cardinality.

The central concept in the proof is that of a peeling sequence, which is defined as follows: a *peeling sequence* for $D(b)$ is a sequence $p_1, p_2, ...$ of points from $D(b)$ with the following property: any p_i in the sequence is extreme in exactly one direction among the points in $D(b) - \{p_1, ..., p_{i-1}\}$. Ties are broken arbitrarily, i.e. if multiple points are extreme in the same direction, we appoint one of them to be the extreme point in that direction. The *type* of a peeling sequence is the sequence $\vec{d_1}, \vec{d_2}, ...$ of directions such that $\vec{d_i}$ is the unique direction in which p_i is extreme among $D(b) - \{p_1, ..., p_{i-1}\}$. Note that there are $(2d)!/(2d-\ell)! = O(1)$ different sequence types of a given length ℓ, so we have $O(1)$ different sequence types of length between 0 and d.

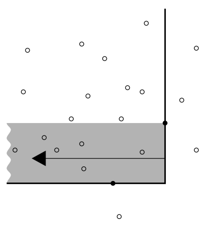


Figure 2.4: Constructing a (τn)-box with sequence type $+x_2, -x_1$ in two dimensions. First choose a basis of two points for the remaining directions (the black dots). Then follow the sequence type in reverse order. The extreme point for direction $-x_1$ must be one of the first τn points found when traversing the shaded area in the direction of the arrow.

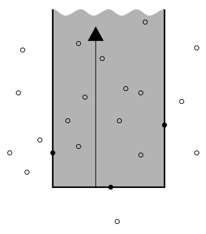


Figure 2.5: The extreme point for the first direction of the sequence, $+x_2$, must be the (τn)'th point in the shaded area.

It is easy to see that there must be a peeling sequence $\sigma(b)$ of length $q = \max(0, |D(b)| - d)$: consider an incremental construction of the sequence, peeling off points from $D(b)$ one at a time, as illustrated in Figs. 2.1–2.3. There are $2d$ directions, so as long as there are more than d points left there must be a point that is extreme in only one direction, which we can peel off.

Call $D^*(b) := D(b) - \sigma(b)$ the *basis* of b. We charge the box b to its basis $D^*(b)$, and we claim that each basis is charged $O((\tau n)^{d-1})$ times. Since there are $O(n^d)$ possible bases, this proves the theorem. To prove the claim, consider a basis D^*, and choose a sequence type. Any (τn)-box b whose basis $D(b)$ is equal to D^* and whose peeling sequence has the given type can be reconstructed incrementally as follows—see Figs. 2.4 and 2.5 for an illustration. Start with $D = D^*$. Now consider the last direction $\vec{d_q}$ of the sequence type. Since the last point p_q of the peeling sequence is extreme only in direction $\vec{d_q}$, it must be contained in the semi-infinite box which is bounded in all other directions by planes through points in D. Hence, only the first τn points in this semi-infinite box are candidates for p_q, otherwise the box would already contain too many points. A similar argument shows there are only τn choices for $p_{q-1}, ..., p_2$. The first point p_1 from the sequence (which is the last point added in the reconstruction) is then fixed, as b must contain exactly τn points—see Figure 2.5.

To prove the lower bound, consider the following configuration (shown in Fig. 2.6 for the planar case). We pair the $2d$ directions $+x_1, -x_1, ..., +x_d, -x_d$ into d pairs $(\vec{d_{11}}, \vec{d_{12}})$, $(\vec{d_{21}}, \vec{d_{22}})$,..., $(\vec{d_{d1}}, \vec{d_{d2}})$ so that no pair contains opposite directions, that is $\vec{d_{i1}} \neq -\vec{d_{i2}}$ for $1 \leqslant i \leqslant d$. Let h_i be the 2-plane spanned by the directions $\vec{d_{i1}}$ and $\vec{d_{i2}}$ and containing the origin. On each 2-plane h_i, we place n/d points $p_i(1), ..., p_i(n/d)$ such that all of them are in the positive quadrant with respect to the origin and both directions $\vec{d_{i1}}$ and $\vec{d_{i2}}$. We place these points along a staircase. More precisely, we require that for $1 < j \leqslant n/d$, the point $p_i(j)$ is closer to the origin than $p_i(j-1)$ with respect to direction $\vec{d_{i1}}$, and further from

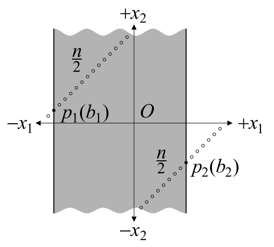


Figure 2.6: A lower bound on the number of (τn)-boxes in two dimensions. The four directions are grouped in two pairs $(-x_1, +x_2)$ and $(+x_1, -x_2)$. We place a staircase of $n/2$ points in the positive quadrant for each pair (in two dimensions, these quadrants are coplanar; in higher dimensions this is not necessarily the case). Choosing one defining point on each staircase fixes two sides of a box. We have $\Theta(n^2)$ ways to do so.

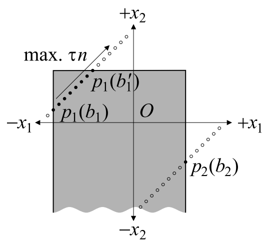


Figure 2.7: Choosing one additional point on one staircase fixes another side of the box. This additional point must be one of the first $\Theta(\tau n)$ points found when walking up the staircase from the first defining point on that staircase. On the remaining staircase, we will have no choice but to choose the point such that the box will contain exactly τn points.

the origin with respect to direction $\vec{d_{i2}}$. Any box containing at least one point from each of these sets can now be specified by choosing two points $p_i(b_i)$ and $p_i(b_i')$ in each 2-plane h_i; we define the box b to be the minimum bounding box of the points chosen. By choosing $b_i' \leqslant b_i + (\tau n - 1)/(d-1) - 1$ for $1 \leqslant i < d$, and $b_d' = b_d - 1 + \sum_{i=1}^{d-1}(b_i' - b_i + 1)$, we get a box containing exactly τn points. Having $\Theta(n)$ choices for each b_i $(1 \leqslant i \leqslant d)$ and $\Theta(\tau n)$ choices for each b_i' $(1 \leqslant i \leqslant d-1)$, we can construct $\Theta(\tau^{d-1} n^{2d-1})$ different (τn)-boxes. □

Note that already in the plane, the bound is cubic in n.

Remark 2.2.2 A different way to state the result above is as follows. Let P be a set of n points in $\mathbb{R}^d$, and let t be a parameter with $1 \leqslant t \leqslant n/(2d)$. Then the maximum number of combinatorially distinct boxes containing exactly t points from P is $\Theta(n^d t^{d-1})$. In other words, we have proved a tight bound on the number of t-sets for ranges that are boxes instead of hyperplanes. Since t-sets have been studied extensively—see e.g. [Dey98] and [Sha01]—we suspected that the case of box-ranges would have been considered as well, but we have only found a result on this for $t = 2$: Alon *et al.* [Alo85] proved that the maximum number of 2-boxes is $(1 - \frac{1}{2^{2^{d-1}-1}})n^2/2 + o(n^2)$.

Remark 2.2.3 The lower-bound example in the proof of Lemma 2.2.1 is quite contrived, and one may hope that much smaller test sets are possible if the points are distributed more regularly. This is not the case, however. As an example, consider the planar case with $\tau = 1/2$, and suppose the point set P is distributed uniformly at random in the unit square. Then the number of $(n/2)$-rectangles is still $\Theta(n^3)$ with high probability. This can be seen as follows. Consider the

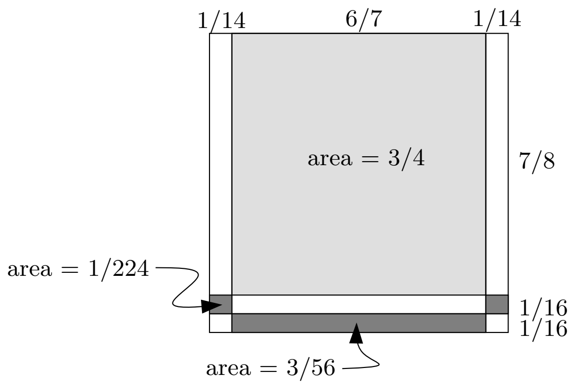


Figure 2.8: Partitioning of the unit square used in the argument in Remark 2.2.3.

partitioning of the unit square into nine regions, as in Fig. 2.8. Since the points are distributed uniformly, the expected number of points in a region of area α is αn. Moreover, the number of points in the region is at least $(2/3)\alpha n$ with probability greater than $1 - \exp(-\alpha n/18)$, which follows from standard tail estimates on the binomial distribution. Hence, the following properties hold simultaneously with high probability:

(1) each of the three darkly shaded regions in Fig. 2.8 has $\Theta(n)$ points;

(2) the lightly shaded region has at least $n/2$ points, which also implies that the six bottommost regions together have at most $n/2$ points.

It follows from (1) that there are $\Theta(n^3)$ triples of points such that each darkly shaded region contains one point from the triple, and it follows from (2) that for each such triple there is a rectangle with these points on the left, right, and bottom edge that contains exactly $n/2$ points.

Approximate test sets. The worst-case bound from Lemma 2.2.1 is quite disappointing. Therefore we now turn our attention to approximate test sets. A set T of boxes is called an *ε-approximate τ-test set* for a set P of n points if

1. any box from T contains at least $(1-\varepsilon)\tau n$ points from P;
2. any query box Q that contains at least τn points from P fully contains at least one box from T.

This means we can answer ε-approximate significant-presence queries on P by answering a presence query on T.

Lemma 2.2.4 *For any set P of n points in $\mathbb{R}^d$ ($d > 1$) and any ε with $0 < \varepsilon < 1/2$, there is an ε-approximate τ-test set of size $O(1/(\varepsilon^{2d-1}\tau^{2d-2}))$. Moreover, there are sets P for which any ε-approximate τ-test set has size $\Omega(1/(\varepsilon^{2d-1}\tau^{d}))$.*

Proof: To prove the upper bound, we proceed as follows. We will construct test sets recursively, starting with the full set P as input. If the size of the current set P is less than τn_0, where n_0 is the original number of points, there is nothing to do. Otherwise, we choose a hyperplane h orthogonal to the x_1-axis, such that at most half of the points in P lies on either side of h. Then we construct three test sets, one for queries on one side of h, one for queries on the other side, and one for queries intersecting h. The first two test sets are constructed by applying the procedure recursively. The latter set is constructed as follows.

Let n be the number of points in the current set P. We construct a collection $H_2(P)$ of $n(2d-1)/(\varepsilon\tau n_0)$ hyperplanes orthogonal to the x_2-axis, such that there are $\varepsilon\tau n_0/(2d-1)$ points of P between any pair of consecutive hyperplanes.[3] We do the same for the other axes, except the x_1-axis, obtaining sets $H_3(P), \ldots, H_d(P)$.

From these collections of hyperplanes we construct our test set as follows. Take any possible subset H^* of $2d-2$ hyperplanes from $H_2(P) \cup \cdots \cup H_d(P)$ such that $H_2(P)$ up to $H_d(P)$ each contribute exactly two hyperplanes to H^*. Let $P(H^*)$ be the set of points in P that lie on or between the hyperplanes contributed by $H_i(P)$, for all $2 \leqslant i \leqslant d$. Construct a collection $H_1(H^*)$ of hyperplanes orthogonal to the x_1-axis, such that there are $\varepsilon\tau n_0/(2d-1)$ points of $P(H^*)$ between each pair of consecutive hyperplanes. For each such hyperplane $h' \in H_1(H^*)$, construct a test box b with the following properties:

1. b is bounded by h', the hyperplanes from H^*, and one additional hyperplane parallel to h' and through a point of $P(H^*)$;
2. b is a $((1-\varepsilon)\tau n_0)$-box.

Of all the test boxes thus constructed, we discard those that do not intersect h. Hence we will only keep boxes for which h' is relatively close to h: there cannot be more than $(1-\varepsilon)\tau n_0$ points from $P(H^*)$ between h and h'.

This implies that the total number of test boxes we create in this step is bounded by $(1-\varepsilon)\tau n_0 \ / \ (\varepsilon\tau n_0/(2d-1)) \leqslant (2d-1)/\varepsilon$ for a fixed set H^*. Hence, we create at most $(n(2d-1)/(\varepsilon\tau n_0))^{2d-2} \cdot (2d-1)/\varepsilon$ boxes in total. The number $T(n)$ of boxes created in the entire recursive procedure therefore satisfies:

$$\begin{aligned} T(n) &= 0 && \text{if } n < \tau n_0 \\ T(n) &\leqslant 2T(n/2) + \left(\tfrac{2d-1}{\varepsilon\tau n_0}\right)^{2d-2} \cdot \tfrac{2d-1}{\varepsilon} \cdot n^{2d-2} && \text{otherwise.} \end{aligned}$$

This leads to $|T| = T(n_0) = O(1/(\varepsilon^{2d-1}\tau^{2d-2}))$.

We now argue that T is an ε-approximate τ-test set for P. By construction, every box in T contains at least $(1-\varepsilon)\tau n_0$ points, so it remains to show that

[3]If there are more points with the same x_2-coordinate, we choose the hyperplanes such that we have at most $\varepsilon\tau n_0/(2d-1)$ points strictly in between consecutive hyperplanes, and at least $\varepsilon\tau n_0/(2d-1)$ points in between or on consecutive hyperplanes.

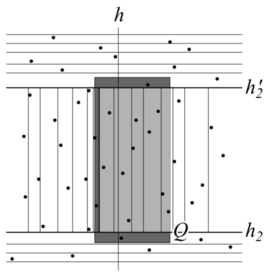


Figure 2.9: An example query range Q (shaded area) that intersects h, showing also h_2, h_2' and the grid $H_1(\{h_2, h_2'\})$. The three dark areas of Q each contain at most $\varepsilon\tau n_0/3$ points. Hence, if Q contains at least τn_0 points, the bright area of Q contains at least $(1-\varepsilon)\tau n_0$ points, and a test box like the one shown above, bounded by h_2, h_2' and a grid line from $H_1(\{h_2, h_2'\})$, must lie inside Q.

every box Q that contains at least τn_0 points from P fully contains at least one box b from T. Let h be the first hyperplane used in the recursive construction. If at least τn_0 points in Q lie to the same side of h, we can assume that there is a test box contained in Q by induction. If this is not the case, we will show that a test box b inside Q was created for queries intersecting h. To see that such a box must exist, observe that for any i with $2 \leqslant i \leqslant d$, there must be a hyperplane $h_i \in H_i(P)$ that intersects Q and has at most $\varepsilon\tau n_0/(2d-1)$ points from $Q \cap P$ below it. Similarly, there is a hyperplane $h_i' \in H_i(P)$ intersecting Q with at most $\varepsilon\tau n_0/(2d-1)$ points from $Q \cap P$ above it. Note that $h_i \neq h_i'$. Let H^* be the set $\{h_2, h_2', h_3, h_3', \ldots, h_d, h_d'\}$. Since each of these hyperplanes 'splits off' at most $\varepsilon\tau n_0/(2d-1)$ points from Q, they define, together with the facets of Q orthogonal to the x_1-axis, a box contained in Q and containing at least $(1-\varepsilon+\varepsilon/(2d-1))\tau n_0$ points. From this, it follows that our construction, when processing this particular H^*, must have produced a test box $b \subset Q$. The proof is illustrated in Fig. 2.9.

To prove the lower bound, recall the construction used in Lemma 2.2.1 for the lower bound for the exact case. There we used d staircases of n/d points each. We then picked two points from each staircase, with at most $(\tau n - 1)/(d-1)$ points between (and including) them, except for the last staircase, where we picked only one point. Each such combination of points defined a different (τn)-box, thus given $\Omega(\tau^{d-1} n^{2d-1})$ different (τn)-boxes. Now, for the approximate case, we consider a subset of $(n/d)/(\varepsilon\tau n+2)$ so-called *anchor points* along each staircase, such that two consecutive anchor points have $\varepsilon\tau n + 1$ points in between. We now pick two anchor points from each staircase, except the last staircase, where we pick one. We make sure that in between two chosen anchor points from the same staircase, there are at most $(\tau n - 1)/(d-1)$ points. We then pick a final point on the last staircase to obtain a (τn)-box. Each of these boxes must be captured by a different test box, because the intersection of two such boxes contains less than $(1-\varepsilon)\tau n$ points. The lower bound follows. $\square$

Putting it all together. To summarize, the construction of our data structure for ε-approximate significant-presence queries on $S = S_1 \cup \cdots \cup S_m$ is as follows. We construct an ε-approximate τ-test set T_i for each color class S_i. This gives us a collection of $M = O(m/(\varepsilon^{2d-1}\tau^{2d-2}))$ boxes in $\mathbb{R}^d$. We map these boxes to a set

$\hat{S}$ of colored points in $\mathbb{R}^{2d}$, and construct a data structure for the standard colored range-searching problem (that is, presence queries) on P, using the techniques of Agarwal *et al.* [Aga02]. Their structure was designed for searching on a grid, but using the standard trick of normalization—replace every coordinate by its rank, and transform the query box to a box in this new search space in $O(\log n)$ time before running the query algorithm—we can employ their results in our setting.

The same technique works for exact queries, if we use exact test sets. This gives a good result for $d = 1$, if we use the results from Gupta *et al.* [Gup95] on quadrant range searching.

Theorem 2.2.5 *Let $S = S_1 \cup \cdots \cup S_m$ be a colored point set in $\mathbb{R}^d$, and τ a fixed constant with $0 < \tau < 1$. For $d = 1$, there is a data structure that uses $O(n)$ storage such that exact τ-significant-presence queries can be answered in $O(\log n + k)$ time, where k is the number of reported colors. For $d > 1$, there is, for any ε with $0 < \varepsilon < 1/2$ and any $\delta > 0$, a data structure for S that uses $O(M^{1+\delta})$ storage such that ε-approximate τ-significant-presence queries on S can be answered in $O(\log n + k)$ time, where $M = O(m/(\varepsilon^{2d-1}\tau^{2d-2}))$.*

Remark 2.2.6 Observe that, since we only have constantly many points per color, we could also use standard range-searching techniques. But this would increase the term k in the reporting time to $O(k/(\varepsilon^{2d-1}\tau^{2d-2}))$, which is undesirable.

The case of variable τ. Now consider the case where the parameter τ is not given in advance, but is part of the query. We assume that we have a lower bound τ_0 on the value of τ in any query. Then we can still answer queries efficiently, at only a small increase in storage. To do so, we build a collection of $O(T)$ substructures, where $T = \log(1/\tau_0)/\log(1 + \varepsilon/2)$. More precisely, for integers i with $0 \leqslant i \leqslant T$, we define $\tau_i := (1 + \varepsilon/2)^i\tau_0$, and for each such i we build a data structure for $(\varepsilon/2)$-approximate τ_i-significant-presence queries on S. To answer a query with a query box Q and query parameter τ, we first find the largest τ_i smaller than or equal to τ, and we query with Q in the corresponding data structure. This leads to the following result.

Theorem 2.2.7 *Let $S = S_1 \cup \cdots \cup S_m$ be a colored point set in $\mathbb{R}^d$, and τ_0 a fixed constant with $0 < \tau_0 < 1$. For $d > 1$, any $0 < \varepsilon < 1/2$ and any $\delta > 0$, there is a data structure for S that uses $O(M^{1+\delta}/\varepsilon)$ storage such that, for any $\tau \geqslant \tau_0$, one can answer ε-approximate τ-significant-presence queries on S in $O(\log n + k)$ time, where $M = O(m/(\varepsilon^{2d-1}\tau_0^{2d-2}))$ and k is the number of reported colors.*

Proof: By Theorem 2.2.5, the size of substructure i is $O(M^{1+\delta}(\tau_0/\tau_i)^D) = O(M^{1+\delta}/(1+\varepsilon/2)^{Di})$, where $M = O(m/(\varepsilon^{2d-1}\tau_0^{2d-2}))$ and $D = (2d-2)(1+\delta)$. The total size of all substructures is therefore $O(M^{1+\delta}\sum_{i=0}^{T}(1+\varepsilon/2)^{-Di}) = O(M^{1+\delta}/\varepsilon)$.

It remains to show that queries are answered correctly. Note that $\tau_i \leqslant \tau \leqslant (1 + \varepsilon/2)\tau_i$. Now, any color j with $|Q \cap S_j| \geqslant \tau_i|S_j|$ will be reported by our algorithm, so certainly any color with $|Q \cap S_j| \geqslant \tau |S_j|$ will be reported. Second, for any reported color j we have:

$$\begin{aligned} |Q \cap S_j| &\geqslant (1-\varepsilon/2)\cdot \tau_i |S_j| \\ &\geqslant (1-\varepsilon/2)\cdot \tau/(1+\varepsilon/2)\cdot |S_j| \\ &\geqslant (1-\varepsilon)\tau \cdot |S_j|. \end{aligned}$$

This proves the correctness of the algorithm. □

2.3 Stabbing queries

Let $B = B_1 \cup \cdots \cup B_m$ be a set of n colored boxes in $\mathbb{R}^d$, where B_i denotes the subset of boxes of color i. Let τ be a constant with $0 < \tau < 1$. For a point q, we use $B_i(q)$ to denote the subset of boxes from B_i that contain q. We want to preprocess B for the following type of stabbing queries: given a query point q, report all colors i such that $|B_i(q)| \geqslant \tau \cdot |B_i|$. As was the case for range queries, we are not able to obtain near-linear storage for exact queries for $d > 1$, so we focus on the ε-approximate variant, where we are also allowed to report a color if $|B_i(q)| \geqslant (1-\varepsilon)\tau \cdot |B_i|$.

Our approach is similar to our approach for range searching. Thus we define an *ε-approximate τ-test set* for a set B_i to be a set T_i of test boxes such that

1. for any point q with $|B_i(q)| \geqslant \tau \cdot |B_i|$, there is a test box b with $q \in b$;
2. for any test box b and any point $q \in b$, we have $|B_i(q)| \geqslant (1-\varepsilon)\tau \cdot |B_i|$.

This means we can answer a query by reporting all colors i for which there is a test box $b \in T_i$ that contains q.

Lemma 2.3.1 *For any set B_i of boxes in $\mathbb{R}^d$, there is an ε-approximate τ-test set T_i consisting of $O(1/(\varepsilon\tau)^d)$ disjoint boxes. Moreover, for $\varepsilon < 1/(2d)$, there are sets of boxes in $\mathbb{R}^d$ for which any ε-approximate τ-test set has size $\Omega(((1-\tau)/(\varepsilon\tau))^d)$.*

Proof: For each of the d main axes, sort the facets of the input boxes orthogonal to that axis, and take a hyperplane through every $(\varepsilon\tau n_i/d)$-th facet, where $n_i := |B_i|$. This gives d collections of $d/(\varepsilon\tau)$ parallel planes, which together define a grid with $O(1/(\varepsilon\tau)^d)$ cells. We let T_i consist of all cells that are fully contained in at least $(1-\varepsilon)\tau \cdot |B_i|$ boxes from B_i. Clearly T_i has the required number of boxes, and has property (2). (Note: using the fact that, coming from infinity, we must cross at least $d(1-\varepsilon)/\varepsilon \geqslant (1/\varepsilon)-1$ hyperplanes before we can come to a cell from T_i, we can in fact obtain a slightly stronger bound on the size of T_i for the case where τ is large.)

It remains to show that T_i has property (1). Let q be a point for which $|B_i(q)| \geqslant \tau \cdot |B_i|$, and let C be the cell containing q. Since any cell is crossed by at most $\varepsilon\tau n_i$ facets, we must have $C \in T_i$.

The lower bound is proved as follows. For each of the main axes, take a collection of $(1-\tau)/(2d\varepsilon\tau)$ hyperplanes orthogonal to that axis. Slightly 'inflate' each hyperplane to obtain a very thin box. This way each intersection point of d hyperplanes becomes a tiny hypercube. Next, each of these thin boxes is replaced by $2\varepsilon\tau n_i$ identical copies of itself. Note that each tiny hypercube is now covered by $2d\varepsilon\tau n_i$ boxes, and that there are $((1-\tau)/(2d\varepsilon\tau))^d$ such hypercubes. Add a collection of $(1-2d\varepsilon)\tau n_i$ big boxes, each containing all the tiny hypercubes. The tiny hypercubes are now covered by exactly τn_i boxes, and the remaining space is covered by at most $(1-2\varepsilon)\tau n_i$ boxes. (Since we have used slightly less than n_i boxes in total, we need to add some more boxes, at some arbitrary location disjoint from all other boxes.) Any test set must contain each of the hypercubes, and the result follows. $\square$

To solve our problem, we construct a test set T_i for each color class B_i according to the lemma above. This gives us a collection of $M = O(m/(\varepsilon\tau)^d)$ colored boxes. Applying the results of Agarwal *et al.* [Aga02] again, we get the following result.

Theorem 2.3.2 *Let $B = B_1 \cup \cdots \cup B_m$ be a colored set of boxes in $\mathbb{R}^d$, and τ a fixed constant with $0 < \tau < 1$. For $d = 1$, there is a data structure that uses $O(n)$ storage such that exact τ-significant-presence queries can be answered in $O(\log n + k)$ time, where k is the number of reported colors. For $d > 1$, there is, for any ε with $0 < \varepsilon < 1/2$ and any $\delta > 0$, a data structure for B that uses $O(M^{1+\delta})$ storage such that ε-approximate τ-significant-presence queries on B can be answered in $O(\log n + k)$ time, where $M = O(m/(\varepsilon\tau)^d)$.*

Remark 2.3.3 Note that, since the test boxes from any given color are disjoint, we can simply report the color of each box containing the query point q. Thus we do not have to use the structure of Agarwal *et al.*, but we can apply results from standard non-colored stabbing queries [Chz88]. This way we can slightly reduce storage to $O(M \log^{d-2+\delta} M)$ at the cost of a slightly increased query time of $O(\log^{d-1} M + k)$. Also note that we can treat the case of variable τ in exactly the same way as for range queries.

2.4 Concluding remarks

Standard colored range searching problems ask to report all colors that have at least one object of that color intersecting the query range. We considered the variant where a color should only be reported if some constant pre-specified fraction of the objects intersects the range. We developed efficient data structures for an approximate version of this problem for orthogonal range searching queries and for stabbing queries. One obvious open problem is whether there exists a data structure for the exact problem with near-linear space. We have shown that this is impossible using our test-set approach, but perhaps a completely different approach is possible. Another open problem is to close the gap between our upper and lower bounds for the size of approximate test sets for orthogonal range searching. Finally, one can develop structures that can report the color that has the most points in the query range. Krizanc *et al.* [Krz03] recently studied this problem for $d = 1$, and it would be interesting to generalize their results to $d \geqslant 2$.

Acknowledgements. We thank Joachim Gudmundsson and Jan Vahrenhold for inspiring discussions about the subject of this paper. Herman Haverkort's work is supported by the Netherlands' Organization for Scientific Research (NWO).

Chapter 3

Box-trees and R-trees with near-optimal query time

Previously published as: Pankaj K. Agarwal, Mark de Berg, Joachim Gudmundsson, Michael Hammar and Herman J. Haverkort: Box-trees and R-trees with near-optimal query time, in *Discrete and Computational Geometry*, 28(3), 2002, pages 291–312.

In the following text, the bounds in Lemma 3.3.6, Lemma 3.3.7, Lemma 3.3.8, Theorem 3.3.9, Lemma 3.3.10, Theorem 3.3.11, and Corollary 3.4.4 are a little stronger than in the previously published version. This is due to slighlty more detailed calculations in the proofs.

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

3.1 Introduction

Motivation and problem statement. *Window queries* report all objects of a given set that intersect a d-dimensional axis-parallel query window, that is, a d-dimensional box. Preprocessing a set S of geometric objects in $\mathbb{R}^d$ for answering such queries is central to many applications and has been widely studied in several areas, including computational geometry, computer graphics, spatial databases, GIS, and robotics [Brg97KOS, Man99]. In order to expedite and simplify the data structure, a window query is often answered in two steps. In the first step, called the *filtering* step, each object is replaced by the smallest box containing the object and the query procedure reports the bounding boxes that intersect the query window. (Instead of boxes, other simple shapes such as spheres, ellipsoids or cylinders have also been used.) The second step, called the *refinement step*, extracts the actual objects among these bounding boxes that intersect the query window [Bri94, Ore90]. A few recent results show that under certain reasonable assumptions on the input objects, the number of bounding boxes intersecting a query window is not much larger than the number of objects intersecting the window, which makes this approach quite attractive; see the paper by Zhou and Suri [Zho99] and the references therein. There has been much work on the filtering step, and we also focus on this step. More precisely, we wish to preprocess a set S of n boxes in $\mathbb{R}^d$ so that all boxes of S intersecting a d-dimensional query box can be reported efficiently. We will refer to this query as the *box-intersection query*. A related query is the *box-containment* query in which we want to report all boxes in S that contain a query point.

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

In many applications, especially in the database applications, the set S is too large to fit in the main memory, therefore it is stored on disk. In that case, the main goal is to minimize the number of disk accesses needed to answer a window query, and the performance of an algorithm is analyzed under the standard external memory model [Agg88]. This model assumes that each disk access transmits a contiguous block of t units of data in a single *input/output operation* (or *I/O*). The efficiency of a data structure is measured in terms of the amount of disk space it uses (measured in units of disk blocks), the number of I/Os required to answer a query, and the number of I/Os needed to construct the data structure. In the context of bounding-box hierarchies, several schemes have been proposed that construct a tree as above but in which the fanout of each node depends on t. Some notable examples of external-memory bounding-box hierarchies are various variants of R-trees; see the survey paper [Gae98]. We can still define the *crossing* nodes and the *stabbing number* as earlier, and one can argue that the number of I/Os needed to answer a query is proportional to the stabbing number plus the output size.

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

Previous results. As noted above, several efficient data structures have been proposed for answering a box-intersection query. For example, Chazelle [Chz88] showed that a compressed range tree can be used to answer a d-dimensional box-intersection query in time $O(\log^{d-1} n + k)$ using $O(n \log^{d-1} n / \log \log n)$ space (where k is the number of boxes reported). This data structure is too complex to be practical even in $\mathbb{R}^2$. As for bounding volume hiearchies, we know of only one result on the query complexity of box-intersection queries (besides the results on R-trees discussed later): if one maps each d-dimensional box to a point in $\mathbb{R}^{2d}$, constructs a kd-tree on these points, and converts the kd-tree back to a box-tree, then the query time is known to be $O(n^{1-1/2d} + k)$ [Aga98E, Lau78]. A number of heuristics based on kd-trees have also been proposed to answer box-intersection queries [Aga98E, Nie97]. Several papers [Got96, Klo98] describe how to construct bounding-box hierarchies or other bounding-volume hierarchies (for example, using k-DOPs as bounding volumes), but they do not obtain bounds on the worst-case query complexity.[1]

Some of the most widely used external-memory bounding-box hierarchies are the R-tree and its variants. An R-tree, originally introduced by Guttmann [Gut84], is a B-tree, each of whose leaves is associated with an input box. All leaves of an R-tree are at the same level, the degree of all internal nodes except of the root is between t and $2t$, for a given parameter t, and the degree of the root varies between 2 and $2t$. We will refer to t as the *minimum degree* of the tree. To minimize the query complexity, several methods have been proposed [Fal92, Fal87, Gae98, Leu97] for ordering the input boxes along the leaves—varying from simple heuristics to space filling curves—but none of them guarantee the worst-case performance. In the worst case, a linear number of bounding boxes might intersect a query box even if it intersects only $O(1)$ input boxes. The only analytical results are by Theodoridis and Sellis [The96], who present a model that predicts the average performance of R-trees for range queries, and Faloutos *et al.* [Fal87], but they prove bounds on the query time only in the 1-dimensional case when the

[1]Barequet *et al.* [Bar96] gave an algorithm to construct a bounding-box hierarchy in $\mathbb{R}^2$, and they claimed that if the boxes in S are pairwise disjoint, then the resulting hierarchy has $O(\log n)$ stabbing number. But the argument presented in the paper has a technical problem.

input intervals are uniformly distributed and have at most two different lengths. Recently, de Berg *et al.* [Brg00] described an algorithm for constructing an R-tree on boxes in $\mathbb{R}^2$ so that all k boxes containing a query point can be reported in $O((\sigma + \log \rho) \log n / \log t)$ I/Os. Here ρ is the ratio of the maximum and the minimum x-lengths of the input boxes, and σ is the *point-stabbing number* of S, that is, σ is the maximum number of input boxes containing any point in the plane. For a box-intersection query, the number of I/Os is $O((\sigma + \log \rho + w + k) \log n / \log t)$, where w is the ratio of the x-length of the query box to the smallest x-length of an input box.

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

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

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

Finally, we give general methods to convert box-trees with small query complexity into R-trees with small query complexity. When we apply these results to our box-trees, we improve the result of de Berg *et al.* [Brg00]: our query complexity does not depend on the parameter w (which makes their query complexity linear in the worst case), and it is linear in $\sqrt{\sigma}$ instead of in σ. We also introduce the concept of *semi-R-trees*; these are similar to ordinary R-trees—the degree of each internal node, except for the root, is between t and $2t$ for some given parameter t—except that the leaves do not have to be at the same level. We give a general algorithm to convert a box-tree with small query complexity into a semi-R-tree with small query complexity; the bound obtained here is better than that for R-trees. This leads to semi-R-trees with (almost) optimal query complexity.

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

3.2 Lower Bounds

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

We start with a simple generalization of the 2-dimensional lower bound given by de Berg *et al.* [Brg00].

Theorem 3.2.1 *For any n and $d \geqslant 2$, there is a set of n disjoint unit hypercubes in $\mathbb{R}^d$ with the following property: for any semi-R-tree $\mathcal{T}$ of minimum degree t there is a query box not intersecting any box from S such that a query with that box visits $\Omega((n/t)^{1-1/d})$ nodes in $\mathcal{T}$.*

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

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

Let $n_1, \ldots, n_{2d}$ be the outward normals of a d-dimensional box. We can pair these normals into d pairs (n_{11}, n_{12}), (n_{21}, n_{22}), $\ldots$, (n_{d1}, n_{d2}) so that no pair contains opposite normals, that is, $n_{i1} \neq -n_{i2}$ for $1 \leq i \leq d$. Let h_i be the 2-plane spanned by the vectors n_{i1} and n_{i2} and containing the origin. Let b be a d-dimensional box containing the origin. Since $n_{i1} \neq -n_{i2}$, the facets f_{i1}, f_{i2} of b normal to n_{i1} and n_{i2}, respectively, share a $(d-2)$-face f_i, which is orthogonal to the 2-plane h_i. The intersection of f_i and h_i is a point c_i. Conversely, by specifying a point c_i on each h_i, $1 \leq i \leq d$, we can represent a unique d-dimensional box in which c_i lies on the facets normal to n_{i1} and n_{i2}. We will therefore define each box $b(j) \in S$ by a d-tuple $(c_1(j), \ldots, c_d(j))$, where the facets of $b(j)$ whose outward normals are n_{i1} and n_{i2} pass through $c_i(j)$. We next describe how to choose the points $c_i(j)$, for $1 \leq i \leq d$ and $0 \leq j < n$.

On each 2-plane h_i, we choose a line ℓ_i of slope -1; the exact equation of ℓ_i will be specified below. We will refer to h_1 as the *primary* plane, and to

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

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

Lemma 3.2.2 *Let $\mathcal{T}$ be any semi-R-tree of minimum degree t on the set S constructed above. Then either there is a primary query point contained in $\Omega(\mu)$ bounding boxes stored in $\mathcal{T}$, or one of the secondary query points is contained in $\Omega(n/(t\mu^{1/(d-1)}))$ bounding boxes stored in $\mathcal{T}$.*

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

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

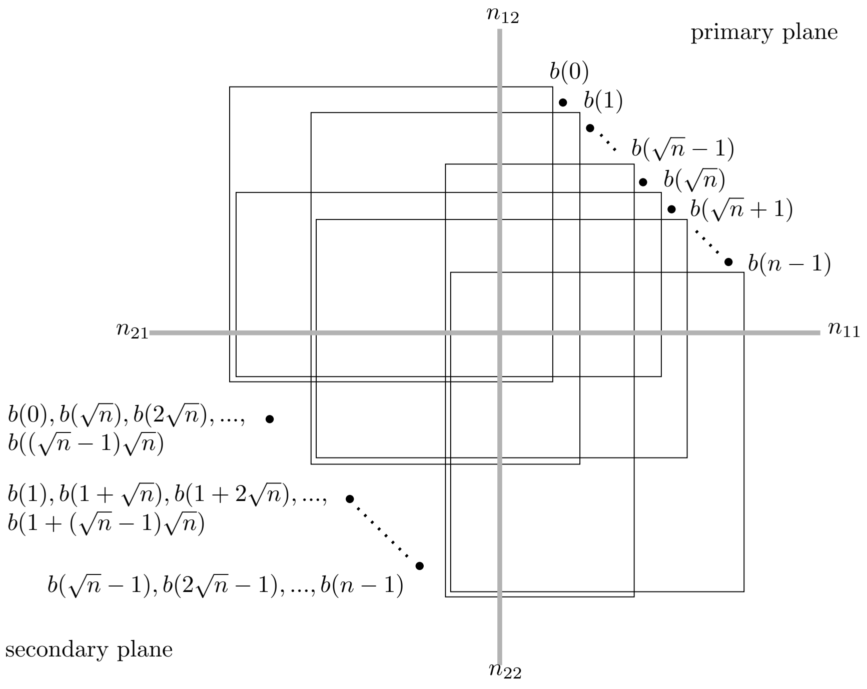

n_{12}
primary plane
$b(0)$
$b(1)$
$b(\sqrt{n}-1)$
$b(\sqrt{n})$
$b(\sqrt{n}+1)$
$b(n-1)$
n_{21}
n_{11}
$b(0), b(\sqrt{n}), b(2\sqrt{n}), ...,$
$b((\sqrt{n}-1)\sqrt{n})$
$b(1), b(1+\sqrt{n}), b(1+2\sqrt{n}), ...,$
$b(1+(\sqrt{n}-1)\sqrt{n})$
$b(\sqrt{n}-1), b(2\sqrt{n}-1), ..., b(n-1)$
secondary plane
n_{22}

Figure 3.1: The lower bound construction in two dimensions for $\mu = \hat{\mu} = \sqrt{n}$. In this case, the primary and the secondary plane coincide. Each of the lower left corners is shared by $\sqrt{n}$ boxes (shown slightly displaced for clarity). The black dots indicate the locations of the query points in Q_1 and Q_2.

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

Theorem 3.2.3 *For any n, $d \geqslant 2$, and $\varepsilon > 0$, there is a set S of n ε-hypercubes in $\mathbb{R}^d$ with the following property: for any semi-R-tree $\mathcal{T}$ of minimum degree t there is a point not contained in any box from S such that a query with that point visits $\Omega((n/t)^{1-1/d})$ nodes in $\mathcal{T}$.*

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

Theorem 3.2.4 *For any n, $d \geqslant 3$, and $\varepsilon > 0$, there is a set S of n disjoint ε-hypercubes in $\mathbb{R}^d$ with the following property: for any semi-R-tree $\mathcal{T}$ of minimum degree t there is a hypercube not intersecting any box from S such that a query with that hypercube visits $\Omega((n/t)^{1-1/d})$ nodes in $\mathcal{T}$.*

Proof: We apply a variant of the construction above with $\mu = n^{1-1/(d-1)}$ to obtain a set of $(d-1)$-dimensional boxes in the hyperplane $x_1 = 0$. The variation is that we treat all planes on which we put the corners as secondary planes. We use the remaining dimension to make the boxes into d-dimensional ε-hypercubes, and we translate each box into the x_1-direction such that they become disjoint and intersect the x_1-axis in the order $b(1), b(2), \ldots, b(n)$. In between every pair $b(j), b(j+1)$ we put a query point. These $n-1$ query points play the role of the primary query points. The secondary query points are replaced by query ranges which are hypercubes. We can do that in such a way that the intersection of such a range with a secondary plane is a square that misses S and that has one corner coinciding with the secondary query points we had previously. It is easy to see that the bound in Lemma 3.2.2 still holds. □

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

Theorem 3.2.5 *For any n, $d \geqslant 3$ and $\varepsilon > 0$, there is a set S of n disjoint ε-hypercubes in $\mathbb{R}^d$ with the following property: for any semi-R-tree $\mathcal{T}$ of minimum degree t, if the number of nodes visited by any point query is μ, then there is a hypercube not intersecting any box from S such that a query with that hypercube visits $\Omega(n/(t\mu^{1/(d-1)}))$ nodes in $\mathcal{T}$.*

3.3 From kd-trees to box-trees

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

3.3.1 The configuration-space approach

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

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

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

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

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

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

Lemma 3.3.1 *The number of cells at depth i in a d-dimensional kd-tree that intersect an axis-parallel f-flat ($0 \leqslant f \leqslant d$) is $O(2^{if/d})$.*

A kd-tree and, hence, our box-tree has the following property: the number of objects stored in the two subtrees of any given node differ by at most one. We call such trees *perfectly balanced*. The perfect balance in our box-tree will be advantageous when we will convert it to an R-tree. We can now analyze the range query complexity of a cs-box-tree.

Lemma 3.3.2 *Let S be a set of n possibly intersecting boxes in $\mathbb{R}^d$. There is a perfectly balanced box-tree for S such that the number of nodes at level i that are visited by a range query with an axis-aligned box is $O(2^{i(1-1/d)} + k)$, where k is the number of boxes in S intersecting the query range. The box-tree can be built in $O(n \log n)$ time.*

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

Assume w.l.o.g. that $x_i = x_i^-(Q)$ is one of these separating hyperplanes, and let b be the input box it separates from Q. Then we must have $x_i^+(b) \leqslant x_i^-(Q)$.

But there must also be a box b' with $x_i^+(b') > x_i^-(Q)$, otherwise the bounding box of ν would not intersect Q. We conclude that the points representing b and b' in the configuration space lie on or on opposite sides of the hyperplane $x_i^+ = x_i^-(Q)$. Consequently, the hyperplane $x_i^+ = x_i^-(Q)$ intersects the cell in configuration space of the node in the kd-tree corresponding to ν.

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

We can conclude the following. Suppose Q visits a node ν of the second type. Then in configuration space there is a pair of hyperplanes, both of the form $x_i^+ = x_i^-(Q)$ or $x_i^- = x_i^+(Q)$ and both intersecting the cell in configuration space of the kd-tree node corresponding to ν. But then the cell is also intersected by the $(2d-2)$-flat that is the intersection of these two hyperplanes. By Lemma 3.3.1 there are only $O(2^{i(2d-2)/2d}) = O(2^{i(1-1/d)})$ such nodes at level i.

For the building time, see section 3.3.4. □

This leads directly to the following theorem.

Theorem 3.3.3 *Let S be a set of n possibly intersecting boxes in $\mathbb{R}^d$. There is a perfectly balanced box-tree for S such that the number of nodes visited by a range query with an axis-aligned box is $O(n^{1-1/d} + k\log n)$, where k is the number of boxes in S intersecting the query range. The box-tree can be built in $O(n\log n)$ time.*

Proof: From Lemma 3.3.2 we get a bound for the stabbing number on each level in the tree. Since a kd-tree has height $\lceil\log n\rceil$, so does a cs-box-tree, and summation over all levels yields a total query complexity of $\sum_{i=0}^{\lceil\log n\rceil} O(2^{i(1-1/d)}+k) = O(n^{1-1/d} + k\log n)$. □

Improving the query time. We now show how to reduce the $O(k\log n)$ term in the query complexity to $O(k)$. The idea is the same as in a priority search tree [Brg97KOS]: input elements (boxes in our case) that have a high chance of being reported are pushed to high levels in the tree. In our case, the boxes that extend farthest in one of the x_i-directions are stored high in the tree. More precisely, the construction of the tree $\mathcal{T}$ for a set S of boxes in $\mathbb{R}^d$ is as follows.

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

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

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

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

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

Lemma 3.3.4 *The number of nodes of weight at least w visited by a query with a query box Q is $O((n/w)^{1-1/d} + k)$.*

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

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

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

The box-tree that we have constructed basically corresponds to a kd-tree in configuration space, as before. The priority leaves make that the tree in configuration space is strictly speaking not a kd-tree, but it is easy to see that Lemma 3.3.1 still holds. Moreover, there is still a one-to-one correspondence between nodes of the box-tree and nodes of the kd-tree in configuration space. Hence, we can use the fact that there are two distinct hyperplanes through facets of Q separating a box in the subtree of ν from Q in the same way as in the proof of Lemma 3.3.2: it implies that there is a $(2d-2)$-flat in configuration space (defined by a pair of facets of Q) intersecting the cell in the kd-tree corresponding to ν. It follows that the total number of nodes ν to which the second case applies at a given level i is $O(2^{i(1-1/d)})$.

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

For the building time, see section 3.3.4. $\square$

The following theorem follows directly.

Theorem 3.3.5 *Let S be a set of n possibly intersecting boxes in $\mathbb{R}^d$. There is a box-tree for S such that the number of nodes that are visited by a range query with an axis-aligned box is $O(n^{1-1/d} + k)$, where k is the number of boxes in S intersecting the query range.*

3.3.2 The kd-interval-tree approach

The cs-box-tree of the previous section has optimal query complexity for point queries (and range queries) if the input consists of arbitrary, intersecting boxes. Unfortunately, if the input boxes are disjoint then the query complexity for point queries does not improve. In this section we develop a different box-tree, the *kd-interval tree*, whose query complexity is much better if σ, the point-stabbing number of the input set S, is small. The query complexity for range queries increases only slightly. This approach only works in the plane; Theorem 3.2.5 states that a similar result in more than two dimensions cannot be obtained.

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

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

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

For each of the 4 inner normals of the edges of $b(\nu)$, take the box from S that extends farthest in the direction of that normal. This results in a set S^* of at most 4 boxes. Each box in S^* is put in a *priority leaf.*

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

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

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

Lemma 3.3.6 *If we query a 1-dimensional kd-interval tree storing a set S of n boxes with a line segment on the defining line ℓ, then we visit at most $O(\log \frac{n}{w} + k)$ nodes of weight at least w, where k is the number of boxes to be reported.*

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

Consider a 1D-node that is visited when we query with s. When the interval corresponding to this node is completely contained in s, then by the above observation all input boxes in the subtree intersect s. Hence, there cannot be more than $O(k)$ such nodes. When the interval is not completely contained in s, then it contains an endpoint of s, and there are only $O(\log(n/w))$ such nodes with weight at least w.

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

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

Lemma 3.3.7 *(i) If we query a 1-dimensional kd-interval tree storing a set S of n boxes with a query box Q, then we visit at most $O(\lceil \sqrt{\sigma/w}\, \rceil \log(n/\sigma) + k)$ nodes of weight at least w, where k is the number of boxes to be reported.*
(ii) If σ is $O(\log n)$, then the query time is $O(\log n + k)$.
(iii) If the projection of Q onto the line ℓ that stabs the boxes in S contains the intersections of all boxes with ℓ, then the query time reduces to $O(k)$.

Proof: (i) See Figure 3.2. If Q intersects ℓ then the query is equivalent to querying with $Q \cap \ell$, so the result follows from the previous lemma. Otherwise, assume w.l.o.g. that ℓ is vertical and that Q lies to the right of ℓ. Consider a 1D-node ν that is visited when we query with Q. When the interval corresponding to this node is completely contained in the projection of Q onto ℓ, then the input box in the subtree extending farthest to the right must be intersected. This box is stored in a priority leaf immediately below ν, to which we can charge the visit of ν. Hence, there can be at most k such nodes. When the interval is not completely contained in the projection of Q, then it contains an endpoint of the projection of Q, and there are only $O(\log(n/w))$ such nodes of weight at least w.

Now consider a 2-dimensional cs-priority-box-tree that is visited. Suppose the interval of the 1D-node that is the parent of this subtree is completely contained in the projection of Q. Then we can argue again (using the priority leaves) that we can charge all the visited nodes to input boxes intersecting Q. If the interval of the 1D-node that is the parent of this subtree is not completely contained in the projection of Q, we argue as follows. First observe that the interval must then

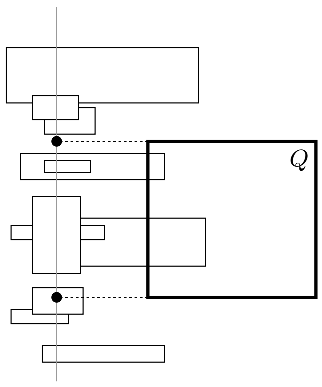


Figure 3.2: Querying a 1-dimensional kd-interval tree with a box Q.

contain an endpoint of the projection of Q, so there are only at most two such parent nodes on each level in the tree. In the 2-dimensional configuration-space box-tree below such a parent, we apply Lemma 3.3.4 to bound the number of visited nodes of weight w by $O(\sqrt{n'/w} + k')$, where n' is the number of boxes stored in the cs-priority-box-tree and k' is the number of answers reported in this subtree. Note that $n' \leqslant \sigma$, since the cs-box-trees are used only to store sets of boxes that share a single point. Furthermore, when the parent is on depth i in the tree, we have $n' \leqslant n/2^i$. Hence, the overall number of cs-nodes visited is:

$$O\left(\log\frac{n}{w} + \left(\sum_{i=0}^{\lfloor\log(n/\sigma)\rfloor}\left\lceil\sqrt{\frac{\sigma}{w}}\right\rceil\right) + \left(\sum_{i=\lfloor\log(n/\sigma)\rfloor+1}^{\log n}\left\lceil\sqrt{\frac{n}{2^i w}}\right\rceil\right) + k\right)$$

which is $O(\lceil\sqrt{\sigma/w}\,\rceil\log(n/\sigma) + k)$. This finishes the proof of part (i) of the lemma.

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

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

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

if σ is $O(\log n)$.

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

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

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

For each of the 4 inner normals of the edges of $b(\nu)$, take the box from S that extends farthest in the direction of that normal. This results in a set S^* of at most 4 boxes. Each box in S^* is put in a *priority leaf*, which is an immediate child of ν.

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

We call the nodes of the main tree, which correspond to 2-dimensional cells, *2D-nodes*. Next we analyze the performance of the kd-interval tree.

Lemma 3.3.8 *The number of nodes of weight at least w that are visited by a range query with an axis-aligned box is $O(\sqrt{n/w} + k)$, where k is the number of reported answers. The number of such nodes visited by a point query is $O(\lceil \sqrt{\sigma/w} \rceil \log^2(n/\sigma) + k)$. If σ is $O(\log n)$, a point query visits $O(\log^2 n)$ nodes.*

Proof: Consider a 2D-node that is visited when we query with an axis-aligned box Q. We distinguish four different types of such nodes (see Figure 3.3). We bound their number and the number of nodes visited in 1-dimensional kd-interval-subtrees for each type separately.

Inner nodes: These are 2D-nodes whose bounding boxes lie completely inside Q. The number of inner nodes is easy to bound, since all input boxes in the subtree of such a node intersect Q. Hence, the total number of such nodes, or nodes in their 1-dimensional associated kd-interval trees, is $O(k)$.

Side nodes: These are 2D-nodes whose bounding boxes cut exactly one edge of Q. In this case the input box that extends farthest into the direction of the inner normal of this edge must intersect Q. This box is stored in a priority leaf immediately below the node. The same reasoning applies to their 1-dimensional

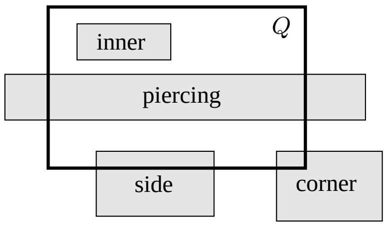


Figure 3.3: Four different types of 2D-nodes with respect to a query range Q.

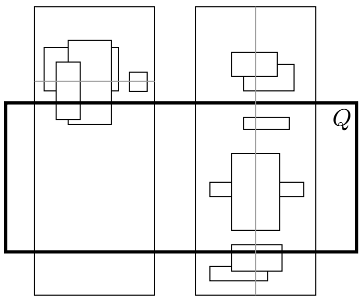


Figure 3.4: Piercing nodes with parallel splitting lines (to the left) and orthogonal splitting lines (to the right).

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

Piercing nodes: These are 2D-nodes that cut two opposing edges of Q, but do not contain any corners of Q. There are two cases—see Figure 3.4: the splitting line used at such a node ν on depth i is parallel to the intersected edges, or it is orthogonal to them. In the first case we can apply Lemma 3.3.7(iii) to get a bound of $O(k')$, where k' is the number of reported answers. In the latter case we can apply Lemma 3.3.6 to obtain a $O(\log((n/2^i)/w) + k')$ bound on the number of nodes visited in the 1-dimensional kd-interval tree associated with ν. From Lemma 3.3.1 we learn that the number of such nodes ν on depth i is $O(2^{i/2})$. Using the fact that nodes at depths greater than $\log(n/w)$ must have weight less than w, we get a grand total of:

$$O(k) + \sum_{i=0}^{\lfloor \log(n/w) \rfloor} O\left(2^{i/2} \log \frac{n}{2^i w}\right) = O\left(\sqrt{\frac{n}{w}} + k\right)$$

Corner nodes: These are 2D-nodes that contain one or more corners of Q. There are $O(1)$ such nodes on each level in the tree. To obtain the total number of visited nodes in the associated 1-dimensional kd-interval trees, we have to sum up the bounds of Lemma 3.3.7 for each of them. Using the fact that for the subset of boxes stored in a subtree on depth i, the stabbing number σ is at most $n/2^i$, we get a total of:

$$O\left(k + \sum_{i=0}^{\log(n/w)} \left\lceil \sqrt{\frac{\min(\sigma, n/2^i)}{w}} \right\rceil \left\lceil \log \frac{n}{2^i \sigma} \right\rceil \right) = O\left(\left\lceil \sqrt{\frac{\sigma}{w}} \right\rceil \log^2 \frac{n}{\sigma} + k \right)$$

If σ is $O(\log n)$, this bound is $O(\log^2 n + k)$.

There are no other types of nodes whose bounding boxes intersect Q. Adding up the number of nodes for all four cases gives the desired bound for box-queries. Note that in the case of point queries, we only have corner nodes. For the building time, see section 3.3.4. □

This leads to the following theorem.

Theorem 3.3.9 *Let S be a set of n possibly intersecting boxes in the plane, such that no single point is contained in more than σ boxes. There is a box-tree for S such that the number of nodes visited by a range query with an axis-aligned box is $O(\sqrt{n} + k)$, where k is the number of boxes in S intersecting the query range. The number of nodes visited by a point query is $O(\sqrt{\sigma} \log^2(n/\sigma) + k)$. If σ is $O(\log n)$, this reduces to $O(\log^2 n)$. The box-tree can be built in $O(n \log n)$ time.*

3.3.3 The longest-side-first approach

Recall that a kd-interval tree is basically a modified kd-tree, where each node is split by a line. The orientations of these lines depend on the level in the tree in such a way, that orientations take turns in a round-robin fashion on any path from the root down into the tree. An interesting variation of the kd-interval tree arises when we replace the round-robin splitting strategy by the longest-side splitting rule as suggested by Dickerson et al. [Dic00]. In such a longest-side-first kd-interval tree, the number of nodes whose corresponding cell is pierced by a query box is small if the query box is fat. We use this to prove the following lemma.

Lemma 3.3.10 *The number of nodes of weight at least w that are visited by a range query with an axis-aligned box is $O(\alpha \log^2 n + \lceil \sqrt{\sigma/w} \rceil \log^2(n/\sigma) + k)$, where k is the number of reported answers. The number of such nodes visited by a point query is $O(\lceil \sqrt{\sigma/w} \rceil \log^2(n/\sigma) + k)$. If σ is $O(\log n)$, the bounds reduce to $O(\alpha \log^2 n + k)$.*

Proof: In the analysis in the previous subsection, the piercing nodes were responsible for the $O(\sqrt{n/w})$ term in the query complexity. This term arose because in a normal kd-tree, there can be $O(\sqrt{n/w})$ piercing nodes.

In the longest-side-first kd-tree, however, the number of disjoint cells that cut opposing sides of a query box of aspect ratio α is $O(\alpha \log n)$ [Dic00]. As before, we have two types of piercing nodes: those with splitting lines that are orthogonal to the intersected edges of Q, and those with parallel splitting lines. For the first case, observe that such splitting lines separate two disjoint cells that cut opposing sides of the query box. This implies that there can be at most $O(\alpha \log n)$ piercing nodes with orthogonal splitting lines, each of which can have a 1-dimensional kd-interval tree in which $O(\log n + k')$ nodes are visited. For the second case, observe that the total number of piercing nodes on all levels in the tree is at most $O(\alpha \log^2 n)$, and each of them can have a 1-dimensional kd-interval tree in which $O(k')$ nodes are visited. Hence, we get a grand total of $O(\alpha \log^2 n + k)$ for both types of piercing nodes.

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

Theorem 3.3.11 *Let S be a set of n boxes in the plane with stabbing number σ. There is a box-tree for S such that the number of nodes that are visited by a range query with a query box of aspect ratio α is $O(\alpha \log^2 n + \sqrt{\sigma} \log^2(n/\sigma) + k)$, where k is the number of boxes in S intersecting the query range. The number of such nodes visited by a point query is $O(\sqrt{\sigma} \log^2(n/\sigma) + k)$. If σ is $O(\log n)$, the bounds reduce to $O(\alpha \log^2 n + k)$. The box-tree can be built in $O(n \log n)$ time.*

3.3.4 Building the box-trees

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

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

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

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

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

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

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

3.4 From box-trees to R-trees

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

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

Theorem 3.4.1 *Let $\mathcal{T}$ be a box-tree for a set of n boxes in $\mathbb{R}^d$ such that any query with a range of a given type visits at most $f(w)$ nodes of weight w or more. Then $\mathcal{T}$ can be converted in $O(n)$ time to an R-tree of minimum degree t where every query with a range of the same type visits at most $O(f(t) \log n / \log t)$ nodes.*

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

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

The construction of Theorem 3.4.1 results in losing a logarithmic factor in the query complexity. Next we show how to improve this result for perfectly balanced box-trees. Recall that a box-tree is called perfectly balanced if for any node the weight of its left and right child differ by at most one.

Theorem 3.4.2 *Let $\mathcal{T}$ be a perfectly balanced box-tree for a set of n boxes in $\mathbb{R}^d$ such that any query with a range of a given type visits at most $f(i)$ nodes at level i in $\mathcal{T}$. Then $\mathcal{T}$ can be converted in $O(n)$ time to an R-tree of minimum degree t where every query with a range of the given type visits at most $O(\sum_{i=0}^{(\log n/\log t)-1} f(i\log t))$ nodes.*

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

We can now construct an R-tree from $\mathcal{T}$ as follows. From the leaf level of the box-tree, walk up the tree until a level i is encountered where all nodes have weight at least t. Thus there must be at least one node with weight at most $t-1$ on the level just below i, and therefore, by the perfect-balance property, no node on that level has weight more than t. This implies that the weight of nodes at level i cannot exceed $2t$. Hence, each subtree rooted at a node at this level can be compressed in a single leaf (which will be a node in the R-tree). Recurse on the new tree. The recursion ends when there are less than t leaves, which are compressed to a single node which will form the root of the R-tree. It is immediately clear that this construction can be done in $O(n)$ time.
The bound on the query complexity immediately follows from the construction. □

Finally, we can show that that we can also improve Theorem 3.4.1 for the general case if we are willing to settle for semi-R-trees instead of real R-trees. Recall that the difference between a semi-R-tree and an R-tree is that in the former we do not require all leaves to be at the same depth.

Theorem 3.4.3 *Let $\mathcal{T}$ be a box-tree for a set of n boxes in $\mathbb{R}^d$ such that any query with a range of a given type visits at most $f(w)$ nodes of weight w or more. Then $\mathcal{T}$ can be converted in $O(n)$ time to a semi-R-tree of minimum degree t where every query with a range of the same type visits at most $O(f(t))$ nodes.*

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

- The total number of semi-R-trees is less than t. In this case, we are done immediately.

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

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

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

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

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

Corollary 3.4.4 *Let S be a set of n boxes in $\mathbb{R}^d$ with stabbing number σ.*

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

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

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

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

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

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

Proof: Part (i) follows from Theorem 3.4.2 and Lemma 3.3.2. Part (ii) follows from Theorem 3.4.3 and Theorem 3.3.5, and part (iii) follows from Theorem 3.4.3 and Lemma 3.3.8. Part (iv) follows from Theorem 3.4.3 and Theorem 3.3.10. To obtain part (v), we use Theorem 3.4.1 instead of Theorem 3.4.3. $\square$

3.5 Conclusions

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

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

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

Chapter 4

The Priority R-Tree: a practically efficient and worst-case-optimal R-tree

An extended abstract of this chapter will appear as: Lars Arge, Mark de Berg, Herman J. Haverkort and Ke Yi: The Priority R-Tree: a practically efficient and worst-case optimal R-tree, Proc. *ACM Special Interest Group on Management of Data (SIGMOD)*, 2004

Abstract. *We present the Priority R-tree, or PR-tree, which is the first R-tree variant that always answers a window query using $O((N/B)^{1-1/d} + T/B)$ I/Os, where N is the number of d-dimensional (hyper-) rectangles stored in the R-tree, B is the disk block size, and T is the output size. This is provably asymptotically optimal and significantly better than other R-tree variants, where a query may visit all N/B leaves in the tree even when $T = 0$. We also present an extensive experimental study of the practical performance of the PR-tree using both real-life and synthetic data. This study shows that the PR-tree performs similar to the best known R-tree variants on real-life and relatively nicely distributed data, but outperforms them significantly on more extreme data.*

4.1 Introduction

Spatial data naturally arise in numerous applications, including geographical information systems, computer-aided design, computer vision and robotics. Therefore spatial database systems designed to store, manage, and manipulate spatial data have received considerable attention over the years. Since these databases often involve massive datasets, disk based index structures for spatial data have been researched extensively—see e.g. the survey by Gaede and Günther [Gae98].

Especially the R-tree [Gut84] and its numerous variants (see e.g. the recent survey by Manolopoulos et al. [Man03]) have emerged as practically efficient indexing methods. In this paper we present the Priority R-tree, or *PR-tree*, which is the first R-tree variant that is not only practically efficient but also provably asymptotically optimal.

4.1.1 Background and previous results

Since objects stored in a spatial database can be rather complex they are often approximated by simpler objects, and spatial indexes are then built on these approximations. The most commonly used approximation is the minimal bounding box: the smallest axis-parallel (hyper-) rectangle that contains the object. The R-tree, originally proposed by Guttman [Gut84], is an index for such rectangles. It is a height-balanced multi-way tree similar to a B-tree [Bay72, Com79], where each node (except for the root) has degree $\Theta(B)$. Each leaf contains $\Theta(B)$ data rectangles (each possibly with a pointer to the original data) and all leaves are on the same level of the tree; each internal node v contains pointers to its $\Theta(B)$ children, as well as for each child a minimal bounding box covering all rectangles in the leaves of the subtree rooted in that child. Figure 4.1 shows an example. If B is the number of rectangles that fits in a disk block, an R-tree on N rectangles occupies $\Theta(N/B)$ disk blocks and has height $\Theta(\log_B N)$. Many types of queries can be answered efficiently using an R-tree, including the common query called a window query: Given a query rectangle Q, retrieve all rectangles that intersect Q. To answer such a query we simply start at the root of the R-tree and recursively visit all nodes with minimal bounding boxes intersecting Q; when encountering a leaf l we report all data rectangles in l intersecting Q.

Guttman gave several algorithms for updating an R-tree in $O(\log_B N)$ I/Os using B-tree-like algorithms [Gut84]. Since there is no unique R-tree for a given dataset, and because the window query performance intuitively depends on the amount of overlap between minimal bounding boxes in the nodes of the tree, it is natural to try to minimize bounding box overlap during updates. This has led to the development of many heuristic update algorithms; see for example [Bmn90, Kam94, Sel87] or refer to the surveys in [Gae98, Man03]. Several specialized algorithms for bulk-loading an R-tree have also been developed [Btd98, Dwt94, Grc98a, Kam93, Leu97, Rou85]. Most of these algorithms use $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os (the number of I/Os needed to sort N elements), where M is the number of rectangles that fits in main memory, which is much less than the $O(N\log_B N)$ I/Os needed to build the index by repeated insertion. Furthermore, they typically produce R-trees with better space utilization and query performance than R-trees built using repeated insertion. For example, while experimental results have shown that the average space utilization of dynamically maintained R-trees is between 50% and 70% [Bmn90], most bulk-loading algorithms are capable of obtaining over 95% space utilization. After bulk-loading an R-tree it can of course be updated using the standard R-tree updating algorithms. However, in that case its query efficiency and space utilization may degenerate over time.

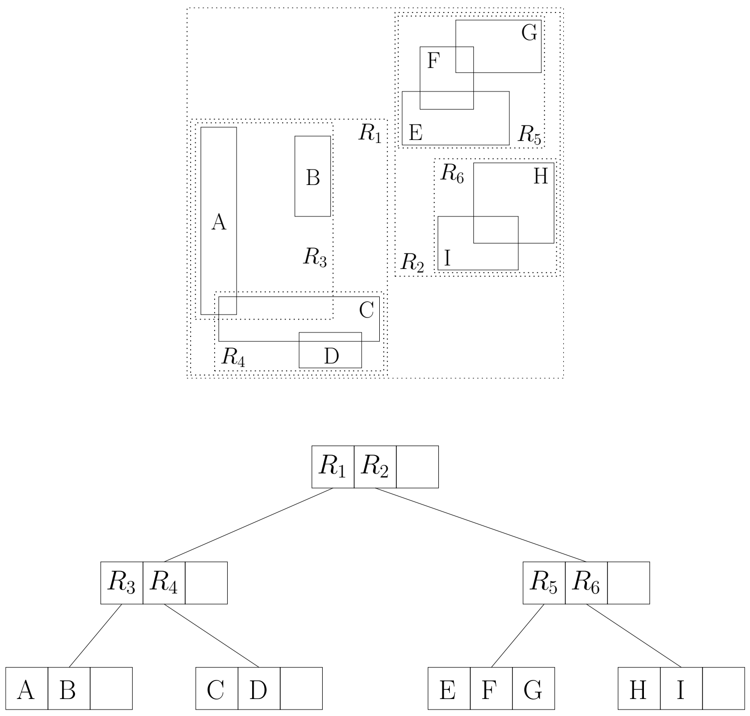


Figure 4.1: R-tree constructed on rectangles A, B, C, . . . , I (block size = 3).

One common class of R-tree bulk-loading algorithms work by sorting the rectangles according to some global one-dimensional criterion, placing them in the leaves in that order, and then building the rest of the index *bottom-up* level-by-level [Dwt94, Kam93, Leu97]. In two dimensions, the so-called packed Hilbert R-tree of Kamel and Faloutsos [Kam93], which sorts the rectangles according to the Hilbert values of their centers, has been shown to be especially query-efficient in practice. The Hilbert value of a point p is the length of the fractal Hilbert space-filling curve from the origin to p. The Hilbert curve is very good at clustering spatially close rectangles together, leading to a good index. A variant of the packed Hilbert R-tree, which also takes the extent of the rectangles into account (rather than just the center), is the four-dimensional Hilbert R-tree [Kam93]; in this structure each rectangle $((x_{\min}, y_{min}), (x_{\max}, y_{\max}))$ is first mapped to the four-dimensional point $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$ and then the rectangles are sorted by the positions of these points on the four-dimensional Hilbert curve. Experimentally the four-dimensional Hilbert R-tree has been shown to behave slightly worse than the packed Hilbert R-tree for nicely distributed realistic data [Kam93]. However, intuitively, it is less vulnerable to more extreme datasets because it also takes the extent of the input rectangles into account.

Algorithms that bulk-load R-trees in a *top-down* manner have also been developed. These algorithms work by recursively trying to find a good partition of the data [Btd98, Grc98a]. The so-called Top-down Greedy Split (TGS) algorithm of García, López and Leutenegger [Grc98a] has been shown to result in especially query-efficient R-trees (TGS R-trees). To build the root of (a subtree of) an R-tree on a given set of rectangles, this algorithm repeatedly partitions the rectangles into two sets, until they are divided into B subsets of (approximately) equal size. Each subset's bounding box is stored in the root, and subtrees are constructed recursively on each of the subsets. Each of the binary partitions takes a set of rectangles and splits it into two subsets based on one of several one-dimensional orderings; in two dimensions, the orderings considered are those by $x_{\min}, y_{\min}, x_{\max}$ and $y_{\max}$. For each such ordering, the algorithm calculates, for each of $O(B)$ possible partitioning possibilities, the sum of the areas of the bounding boxes of the two subsets that would result from the partition. Then it applies the binary partition that minimizes that sum.[1]

While the TGS R-tree has been shown to have slightly better query performance than other R-tree variants, the construction algorithm uses many more I/Os since it needs to scan all the rectangles in order to make a binary partition. In fact, in the worst case the algorithm may take $O(N \log_B N)$ I/Os. However, in practice, the fact that each partition decision is binary effectively means that the algorithm uses $O(\frac{N}{B} \log_2 N)$ I/Os.

While much work has been done on evaluating the practical query performance of the R-tree variants mentioned above, very little is known about their theoretical worst-case performance. Most theoretical work on R-trees is concerned with estimating the expected cost of queries under assumptions such as uniform distribution of the input and/or the queries, or assuming that the input are points rather than rectangles. See the recent survey by Manolopoulos et al. [Man03]. The first bulk-loading algorithm with a non-trivial guarantee on the resulting worst-case query performance was given only recently by Agarwal et al. [Aga01BGHH]. In d dimensions their algorithm constructs an R-tree that answers a window query in $O((N/B)^{1-1/d} + T \log_B N)$ I/Os, where T is the number of reported rectangles. However, this still leaves a gap to the $\Omega((N/B)^{1-1/d} + T/B)$ lower bound on the number of I/Os needed to answer a window query [Aga01BGHH, Kan98]. If the input consists of points rather than rectangles, then worst-case optimal query performance can be achieved with e.g. a kdB-tree [Rob81] or an O-tree [Kan98]. Unfortunately, it seems hard to modify these structures to work for rectangles. Finally, Agarwal et al. [Aga01BGHH], as well as Haverkort et al. [Hav02], also developed a number of R-trees that have good worst-case query performance under certain conditions on the input.

[1]García et al. describe several variants of the top-down greedy method. They found the one described here to be the most efficient in practice [Grc98a]. In order to achieve close to 100% space utilization, the size of the subsets that are created is actually rounded up to the nearest power of B (except for one remainder set). As a result, one node on each level, including the root, may have less than B children.

4.1.2 Our results

In Section 4.2 we present a new R-tree variant, which we call a *Priority R-tree* or *PR-tree* for short. We call our structure the Priority R-tree because our bulk-loading algorithm utilizes so-called priority rectangles in a way similar to the recent structure by Agarwal et al. [Aga01BGHH]. Window queries can be answered in $O((N/B)^{1-1/d} + T/B)$ I/Os on a PR-tree, and the index is thus the first R-tree variant that answers queries with an asymptotically optimal number of I/Os in the worst case. To contrast this to previous R-tree bulk-loading algorithms, we also construct a set of rectangles and a query with zero output, such that all $\Theta(N/B)$ leaves of a packed Hilbert R-tree, a four-dimensional Hilbert R-tree, or a TGS R-tree need to be visited to answer the query. We also show how to bulk-load the PR-tree efficiently, using only $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os. After bulk-loading, a PR-tree can be updated in $O(\log_B N)$ I/Os using the standard R-tree updating algorithms, but without maintaining its query efficiency. Alternatively, the external logarithmic method [Arg03, Pro03] can be used to develop a structure that supports insertions and deletions in $O(\log_B \frac{N}{M} + \frac{1}{B}(\log_{M/B}\frac{N}{B})(\log_2 \frac{N}{M}))$ and $O(\log_B \frac{N}{M})$ I/Os amortized, respectively, while maintaining the optimal query performance.

In Section 4.3 we present an extensive experimental study of the practical performance of the PR-tree using both real-life and synthetic data. We compare the performance of our index on two-dimensional rectangles to the packed Hilbert R-tree, the four-dimensional Hilbert R-tree, and the TGS R-tree. Overall, our experiments show that all these R-trees answer queries in more or less the same number of I/Os on relatively square and uniformly distributed rectangles. However, on more extreme data—large rectangles, rectangles with high aspect ratios, or non-uniformly distributed rectangles—the PR-tree (and sometimes also the four-dimensional Hilbert R-tree) outperforms the others significantly. On a special worst-case dataset the PR-tree outperforms all of them by well over an order of magnitude.

4.2 The Priority R-tree

In this section we describe the PR-tree. For simplicity, we first describe a two-dimensional pseudo-PR-tree in Section 4.2.1. The pseudo-PR-tree answers window queries efficiently but is not a real R-tree, since it does not have all leaves on the same level. In Section 4.2.2 we show how to obtain a real two-dimensional PR-tree from the pseudo-PR-tree, and in Section 4.2.3 we discuss how to extend the PR-tree to d dimensions. In Section 4.2.4 we explain how a pseudo-PR-tree can serve as the basis of a structure that supports efficient insertions and deletions while maintaining optimal query efficiency. Finally, in Section 4.2.5 we show that a query on the packed Hilbert R-tree, the four-dimensional Hilbert R-tree, as well as the TGS R-tree can be forced to visit all leaves even if $T = 0$.

4.2.1 Two-dimensional pseudo-PR-trees

In this section we describe the two-dimensional pseudo-PR-tree. Like an R-tree, a pseudo-PR-tree has the input rectangles in the leaves and each internal node ν contains a minimal bounding box for each of its children ν_c. However, unlike an R-tree, not all the leaves are on the same level of the tree and internal nodes only have degree six (rather than $\Theta(B)$).

The basic idea of a pseudo-PR-tree is (similar to the four-dimensional Hilbert R-tree) to view an input rectangle $((x_{\min}, y_{\min}), (x_{\max}, y_{\max}))$ as a four-dimensional point $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$. The pseudo-PR-tree is then basically just a kd-tree on the N points corresponding to the N input rectangles, except that four extra leaves are added below each internal node. Intuitively, these so-called *priority leaves* contain the extreme B points (rectangles) in each of the four dimensions. Note that the four-dimensional kd-tree can easily be mapped back to an R-tree-like structure, simply by replacing the split value in each kd-tree node ν with the minimal bounding box of the input rectangles stored in the subtree rooted in ν. The idea of using priority leaves was introduced in a recent structure by Agarwal et al. [Aga01BGHH], they used priority leaves of size one rather than B.

In section 4.2.1 below we give a precise definition of the pseudo-PR-tree, and in section Section 4.2.1 we show that it can be used to answer a window query in $O(\sqrt{N/B} + T/B)$ I/Os. In Section 4.2.1 we describe how to construct the structure I/O-efficiently.

The Structure

Let $S = \{R_1, \ldots, R_N\}$ be a set of N rectangles in the plane and assume for simplicity that no two of the coordinates defining the rectangles are equal. We define $R_i^* = (x_{\min}(R_i), y_{\min}(R_i), x_{\max}(R_i), y_{\max}(R_i))$ to be the mapping of $R_i = ((x_{\min}(R_i), y_{\min}(R_i)), (x_{\max}(R_i), y_{\max}(R_i)))$ to a point in four dimensions, and define S^* to be the N points corresponding to S.

A pseudo-PR-tree $\mathcal{T}_S$ on S is defined recursively: if S contains at most B rectangles, $\mathcal{T}_S$ consists of a single leaf; otherwise, $\mathcal{T}_S$ consists of a node ν with six children, namely four priority leaves and two recursive pseudo-PR-trees. For each child ν_c, we let ν store the minimal bounding box of all input rectangles stored in the subtree rooted in ν_c. The node ν and the priority leaves below it are constructed as follows: The first priority leaf $\nu_p^{x_{\min}}$ contains the B rectangles in S with minimal $x_{\min}$-coordinates, the second $\nu_p^{y_{\min}}$ the B rectangles among the remaining rectangles with minimal $y_{\min}$-coordinates, the third $\nu_p^{x_{\max}}$ the B rectangles among the remaining rectangles with maximal $x_{\max}$-coordinates, and finally the fourth $\nu_p^{y_{\max}}$ the B rectangles among the remaining rectangles with maximal $y_{\max}$-coordinates. Thus the priority leaves contain the "extreme" rectangles in S, namely the ones with leftmost left edges, bottommost bottom edges, rightmost right edges, and topmost top edges.[2] After constructing the priority leaves, we

[2]S may not contain enough rectangles to put B rectangles in each of the four priority leaves. In

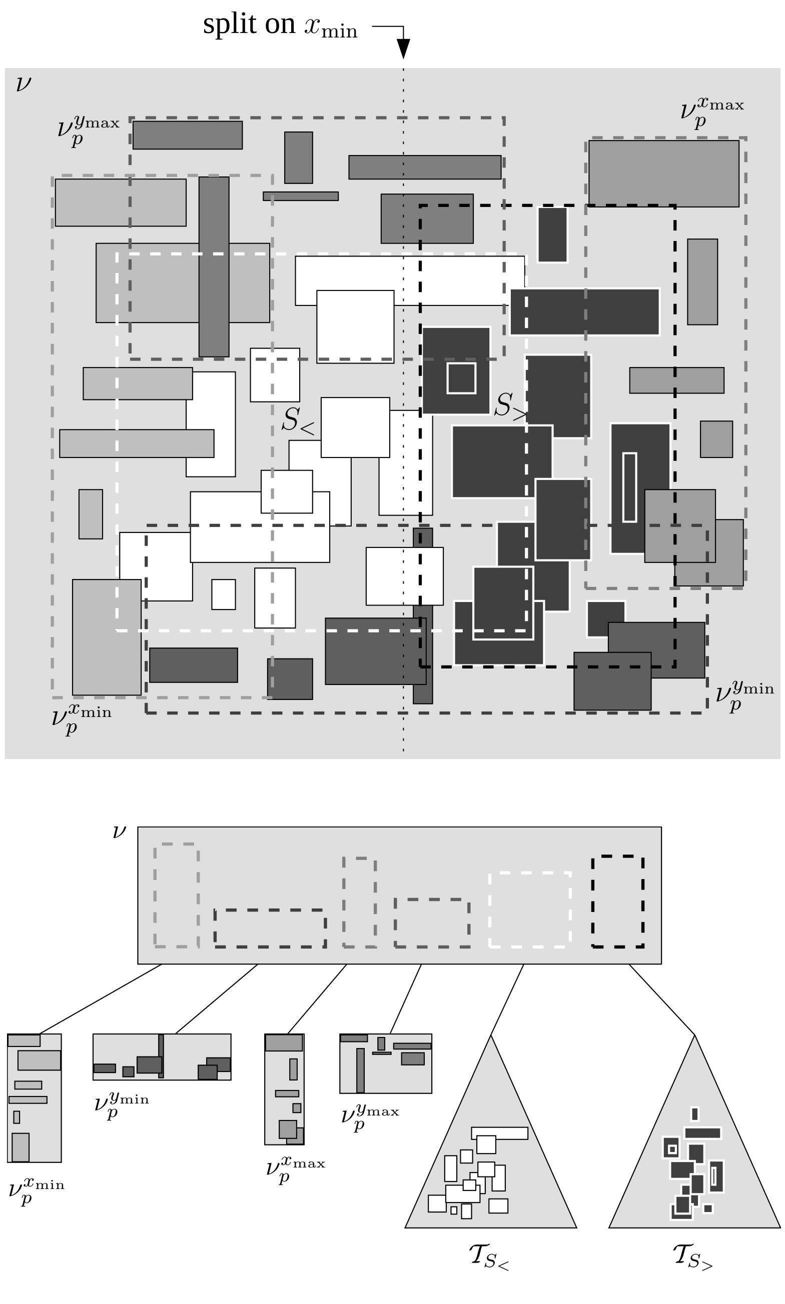


Figure 4.2: The construction of an internal node in a pseudo-PR-tree.

divide the set S_r of remaining rectangles (if any) into two subsets, $S_<$ and $S_>$, of approximately the same size and recursively construct pseudo-PR-trees $\mathcal{T}_{S_<}$ and $\mathcal{T}_{S_>}$. The division is performed using the $x_{\min}$, $y_{\min}$, $x_{\max}$, or $y_{\max}$-coordinate in a round-robin fashion, as if we were building a four-dimensional kd-tree on S_r^*, that is, when constructing the root of $\mathcal{T}_S$ we divide based on the $x_{\min}$-values, the next level of recursion based on the $y_{\min}$-values, then based on the $x_{\max}$-values, on the $y_{\max}$-values, on the $x_{\min}$-values, and so on. Refer to Figure 4.2 for an example. Note that dividing according to, say, $x_{\min}$ corresponds to dividing based on a vertical line ℓ such that half of the rectangles in S_r have their left edge to the left of ℓ and half of them have their left edge to the right of ℓ.

We store each node or leaf of $\mathcal{T}_S$ in $O(1)$ disk blocks, and since at least four out of every six leaves contain $\Theta(B)$ rectangles we obtain the following (in Section 4.2.1 we discuss how to guarantee that almost every leaf is full).

Lemma 4.2.1 *A pseudo-PR-tree on a set of N rectangles in the plane occupies $O(N/B)$ disk blocks.*

Query complexity

We answer a window query Q on a pseudo-PR-tree exactly as on an R-tree by recursively visiting all nodes with minimal bounding boxes intersecting Q. However, unlike for known R-tree variants, for the pseudo-PR-tree we can prove a non-trivial (in fact, optimal) bound on the number of I/Os performed by this procedure.

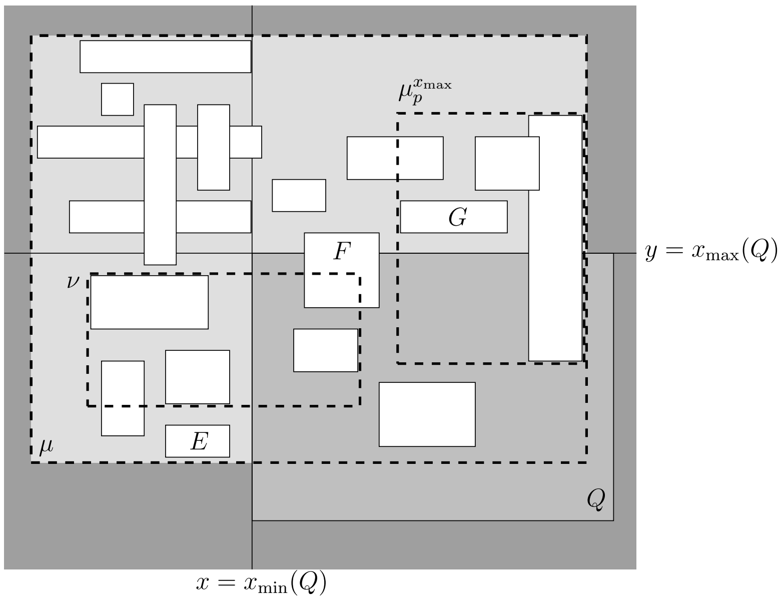
Lemma 4.2.2 *A window query on a pseudo-PR-tree on N rectangles in the plane uses $O(\sqrt{N/B} + T/B)$ I/Os in the worst case.*

Proof: Let $\mathcal{T}_S$ be a pseudo-PR-tree on a set S of N rectangles in the plane. To prove the query bound, we bound the number of nodes in $\mathcal{T}_S$ that are "kd-nodes", i.e. not priority leaves, and are visited in order to answer a query with a rectangular range Q; the total number of leaves visited is at most a factor of four larger.

We first note that $O(T/B)$ is a bound on the number of nodes ν visited where all rectangles in at least one of the priority leaves below ν's parent are reported. Thus we just need to bound the number of visited kd-nodes where this is not the case.

Let μ be the parent of a node ν such that none of the priority leaves of μ are reported completely, that is, each priority leaf μ_p of μ contains at least one rectangle not intersecting Q. Each such rectangle E can be separated from Q by a line containing one of the sides of Q—refer to Figure 4.3. Assume without loss of generality that this is the vertical line $x = x_{\min}(Q)$ through the left edge of Q, that is, E's right edge lies to the left of Q's left edge, so that $x_{\max}(E) \leqslant x_{\min}(Q)$.

that case, we may assume that we can still put at least $B/4$ in each of them, since otherwise we could just construct a single leaf.



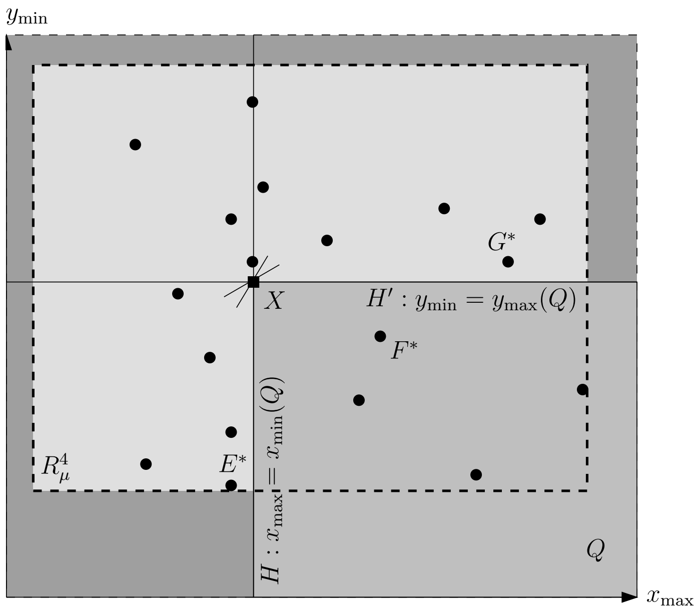


Figure 4.3: The proof of Lemma 4.2.2, with μ in the plane (upper figure), and μ in four-dimensional space (lower figure—the $x_{\min}$ and $y_{\max}$ dimensions are not shown). Note that $X = H \cap H'$ is a two-dimensional hyperplane in four-dimensional space. It contains a two-dimensional facet of the transformation of the query range into four dimensions.

This means that the point E^* in four-dimensional space corresponding to E lies to the left of the axis-parallel hyperplane H that intersects the $x_{\max}$-axis at $x_{\min}(Q)$. Now recall that $\mathcal{T}_S$ is basically a four-dimensional kd-tree on S^* (with priority leaves added), and thus that a four-dimensional region R^4_μ can be associated with μ. Since the query Q visits μ, there must also be at least one rectangle F in the subtree rooted at μ that has $x_{\max}(F) > x_{\min}(Q)$, so that F^* lies to the right of H. It follows that R^4_μ contains points on both sides of H and therefore H must intersect R^4_μ.

Now observe that the rectangles in the priority leaf $\mu_p^{x_{\max}}$ cannot be separated from Q by the line $x = x_{\min}(Q)$ through the left edge of Q: Rectangles in $\mu_p^{x_{\max}}$ are extreme in the positive x-direction, so if one of them lies completely to the left of Q, then all rectangles in μ's children—including ν—would lie to the left of Q; in that case ν would not be visited. Since (by definition of ν) not all rectangles in $\mu_p^{x_{\max}}$ intersect Q, there must be a line through one of Q's other sides, say the horizontal line $y = y_{\max}(Q)$, that separates Q from a rectangle G in $\mu_p^{x_{\max}}$. Hence, the hyperplane H' that cuts the $y_{\min}$-axis at $y_{\max}(Q)$ also intersects R^4_μ.

By the above arguments, at least two of the three-dimensional hyperplanes defined by $x_{\min}(Q)$, $x_{\max}(Q)$, $y_{\min}(Q)$ and $y_{\max}(Q)$ intersect the region R^4_μ associated with μ when viewing $\mathcal{T}_S$ as a four-dimensional kd-tree. Hence, the intersection X of these two hyperplanes, which is a two-dimensional plane in four-dimensional space, also intersects R^4_μ. With the priority leaves removed, $\mathcal{T}_S$ becomes a four-dimensional kd-tree with $O(N/B)$ leaves; from a straightforward generalization of the standard analysis of kd-trees we know that any axis-parallel two-dimensional plane intersects at most $O(\sqrt{N/B})$ of the regions associated with the nodes in such a tree [Aga01BGHH]. All that remains is to observe that Q defines $O(1)$ such planes, namely one for each pair of sides. Thus $O(\sqrt{N/B})$ is a bound on the number of nodes ν that are not priority leaves and are visited by the query procedure, where not all rectangles in any of the priority leaves below ν's parent are reported. $\square$

Efficient construction algorithm

Note that it is easy to bulk-load a pseudo-PR-tree $\mathcal{T}_S$ on a set S of N rectangles in $O(\frac{N}{B}\log N)$ I/Os by simply constructing one node at a time following the definition in Section 4.2.1. We will now describe how, under the reasonable assumption that the amount M of available main memory is $\Omega(B^{4/3})$, we can bulk-load $\mathcal{T}_S$ using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os.

Our algorithm is a modified version of the kd-tree construction algorithm described in [Aga01APV, Pro03]; it is easiest described as constructing a four-dimensional kd-tree $\mathcal{T}_S$ on the points S^*. In the construction algorithm we first construct, in a preprocessing step, four sorted lists $L_{x_{\min}}$, $L_{y_{\min}}$, $L_{x_{\max}}$, $L_{y_{\max}}$ containing the points in S^* sorted by their $x_{\min}$-, $y_{\min}$-, $x_{\max}$-, and $y_{\max}$-coordinate, respectively. Then we construct $\Theta(\log M)$ levels of the tree, and recursively construct the rest of the tree.

To construct $\Theta(\log M)$ levels of $\mathcal{T}_S$ efficiently we proceed as follows. We first choose a parameter z (which will be explained below) and use the four sorted lists to find the (kN/z)-th coordinate of the points S^* in each dimension, for all $k \in \{1, 2, ..., z-1\}$. These coordinates define a four-dimensional grid of size z^4; we then scan S^* and count the number of points in each grid cell. We choose z to be $\Theta(M^{1/4})$, so that we can keep these counts in main memory.

Next we build the $\Theta(\log M)$ levels of $\mathcal{T}_S$ without worrying about the priority leaves: To construct the root ν of $\mathcal{T}_S$, we first find the slice of z^3 grid cells with common $x_{\min}$-coordinate such that there is a hyperplane orthogonal to the $x_{\min}$-axis that passes through these cells and has at most half of the points in S^* on one side and at most half of the points on the other side. By scanning the $O(N/(Bz))$ blocks from $L_{x_{\min}}$ that contain the $O(N/z)$ points in these grid cells, we can determine the exact $x_{\min}$-value x to use in ν such that the hyperplane H, defined by $x_{\min} = x$, divides the points in S^* into two subsets with at most half of the points each. After constructing ν, we subdivide the z^3 grid cells intersected by H, that is, we divide each of the z^3 cells in two at x and compute their counts by rescanning the $O(N/(Bz))$ blocks from $L_{x_{\min}}$ that contain the $O(N/z)$ points in these grid cells. Then we construct a kd-tree on each side of the hyperplane defined by x recursively (cycling through all four possible cutting directions). Since we create $O(z^3)$ new cells every time we create a node, we can ensure that the grid still fits in main memory after constructing z nodes, that is, $\log z = \Theta(\log M)$ levels of $\mathcal{T}_S$.

After constructing the $\Theta(\log M)$ kd-tree levels, we construct the four priority leaves for each of the z nodes. To do so we reserve main memory space for the B points in each of the priority leaves; we have enough main memory to hold all priority leaves, since by the assumption that M is $\Omega(B^{4/3})$ we have $4 \cdot \Theta(B) \cdot \Theta(z) = O(M)$. Then we fill the priority leaves by scanning S^* and "filtering" each point R_i^* through the kd-tree, one by one, as follows: We start at the root of ν of $\mathcal{T}_S$, and check its priority leaves $\nu_p^{x_{\min}}$, $\nu_p^{y_{\min}}$, $\nu_p^{x_{\max}}$, and $\nu_p^{y_{\max}}$ one by one in that order. If we encounter a non-full leaf we simply place R_i^* there; if we encounter a full leaf ν_p and R_i^* is more extreme in the relevant direction than the least extreme point R_j^* in ν_p, we replace R_j^* with R_i^* and continue the filtering process with R_j^*. After checking $\nu_p^{y_{\max}}$ we continue to check the priority leaves of the child of ν in $\mathcal{T}_S$ whose region contains the point we are processing; if ν does not have such a child (because we arrived at leaf level in the kd-tree) we simply continue with the next point in S^*.

It is easy to see that the above process correctly constructs the top $\Theta(\log M)$ levels of the pseudo-PR-tree $\mathcal{T}_S$ on S, except that the kd-tree divisions are slightly different than the ones defined in Section 4.2.1, since the points in the priority leaves are not removed before the divisions are computed. However, the bound of Lemma 4.2.2 still holds: The $O(T/B)$ term does not depend on the choice of the divisions, and the kd-tree analysis that brought the $O(\sqrt{N/B})$ term only depends on the fact that each child gets at most half of the points of its parent.

After constructing the $\Theta(\log M)$ levels and their priority leaves, we scan through the four sorted lists $L_{x_{\min}}$, $L_{y_{\min}}$, $L_{x_{\max}}$, $L_{y_{\max}}$ and divide them into four sorted lists for each of the $\Theta(z)$ leaves of the constructed kd-tree, while omitting the points already stored in priority leaves. These lists contain $O(N/z)$ points each; after writing the constructed kd-tree and priority leaves to disk we use them to construct the rest of $\mathcal{T}_S$ recursively.

Note that once the number of points in a recursive call gets smaller than M, we can simply construct the rest of the tree in internal memory one node at a time. This way we can make slightly unbalanced divisions, so that we have a multiple of B points on one side of each dividing hyperplane. Thus we can guarantee that we get at most one non-full leaf per subtree of size $\Theta(M)$, and obtain almost 100% space utilization. To avoid having an underfull leaf that may violate assumptions made by update algorithms, we may make the priority leaves under its parent slightly smaller so that all leaves contain $\Theta(B)$ rectangles. This also implies that the bound of Lemma 4.2.1 still holds.

Lemma 4.2.3 *A pseudo-PR-tree can be bulk-loaded with N rectangles in the plane in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os.*

Proof: The initial construction of the sorted lists takes $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os. To construct $\Theta(\log M)$ levels of $\mathcal{T}_S$ we use $O(N/B)$ I/Os to construct the initial grid, as well as $O(N/(Bz))$ to construct each of the z nodes for a total of $O(N/B)$ I/Os. Constructing the priority leaves by filtering also takes $O(N/B)$ I/Os, and so does the distribution of the remaining points in S^* to the recursive calls. Thus each recursive step takes $O(N/B)$ I/Os in total. The lemma follows since there are $O(\log\frac{N}{B}/\log M) = O(\log_M\frac{N}{B})$ levels of recursion. $\Box$

4.2.2 Two-dimensional PR-tree

In this section we describe how to obtain a PR-tree (with degree $\Theta(B)$ and all leaves on the same level) from a pseudo-PR-tree (with degree six and leaves on all levels), while maintaining the $O(\sqrt{N/B} + T/B)$ I/O window query bound.

The PR-tree is built in stages bottom-up: In stage 0 we construct the leaves V_0 of the tree from the set $S_0 = S$ of N input rectangles; in stage $i \geqslant 1$ we construct the nodes V_i on level i of the tree from a set S_i of $O(N/B^i)$ rectangles, consisting of the minimal bounding boxes of all nodes in V_{i-1} (on level $i-1$). Stage i consists of constructing a pseudo-PR-tree $\mathcal{T}_{S_i}$ on S_i; V_i then simply consists of the (priority as well as normal) leaves of $\mathcal{T}_{S_i}$; the internal nodes are discarded.[3] The bottom-up construction ends when the set S_i is small enough so that the rectangles in S_i and the pointers to the corresponding subtrees fit into one block, which is then the root of the PR-tree.

[3]There is a subtle difference between the pseudo-PR-tree algorithm used in stage 0 and the algorithm used in stages $i > 0$. In stage 0, we construct leaves with input rectangles. In stages $i > 0$, we construct nodes with pointers to children and bounding boxes of their subtrees. The number of children that fits in a node might differ by a constant factor from the number B of rectangles that fits in a leaf, so the number of children might be $\Theta(B)$ rather than B. For our analysis the difference does not matter and is therefore ignored for simplicity.

Theorem 4.2.4 *A PR-tree on a set S of N rectangles in the plane can be bulk-loaded in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os, such that a window query can be answered in $O(\sqrt{N/B}+T/B)$ I/Os.*

Proof: By Lemma 4.2.3, stage i of the PR-tree bulk-loading algorithm uses $O((|S_i|/B)\log_{M/B}(|S_i|/B)) = O((N/B^{i+1})\log_{M/B}\frac{N}{B})$ I/Os. Thus the complete PR-tree is constructed in

$$\sum_{i=0}^{O(\log_B N)} O\left(\frac{N}{B^{i+1}}\log_{M/B}\frac{N}{B}\right) = O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) \text{ I/Os.}$$

To analyze the number of I/Os used to answer a window query Q, we will analyze the number of nodes visited on each level of the tree. Let T_i ($i \leqslant 0$) be the number of nodes visited on level i. Since the nodes on level 0 (the leaves) correspond to the leaves of a pseudo-PR-tree on the N input rectangles S, it follows from Lemma 4.2.2 that $T_0 = O(\sqrt{N/B}+T/B)$; in particular, there are constants N' and c such that for $N/B \geqslant N'$, $B \geqslant 4c^2$, and $T \geqslant 0$, we have $T_0 \leqslant c\sqrt{N/B}+c(T/B)$. There must be T_{i-1} rectangles in nodes of level $i \geqslant 1$ of the PR-tree that intersect Q, since these nodes contain the bounding boxes of nodes on level $i-1$. Since nodes on level i correspond to the leaves of a pseudo-PR-tree on the N/B^i rectangles in S_i, it follows from Lemma 4.2.2 that for $N/B^{i+1} \geqslant N'$ and $B \geqslant 4c^2$ we have $T_i \leqslant (c/\sqrt{B^i})\sqrt{N/B}+c(T_{i-1}/B)$. We can now write out the recurrence for $N/B^{i+1} \geqslant N'$, that is, for $i \leqslant (\log_B \frac{N}{N'})-1$:

$$\begin{aligned} T_i &\leqslant \frac{c}{\sqrt{B^i}}\sqrt{\frac{N}{B}} + c\frac{T_{i-1}}{B} \\ &\leqslant \frac{c}{\sqrt{B^i}}\left(1+\frac{c}{\sqrt{B}}\right)\sqrt{\frac{N}{B}} + c\frac{c}{B}\frac{T_{i-2}}{B} \\ &\leqslant \ldots \\ &\leqslant \frac{c}{\sqrt{B^i}}\left(\sum_{j=0}^{i}\left(\frac{c}{\sqrt{B}}\right)^j\right)\sqrt{\frac{N}{B}} + c\left(\frac{c}{B}\right)^i\frac{T}{B} \end{aligned}$$

With $B \geqslant 4c^2$, it follows that:

$$T_i \leqslant \frac{2c}{(2c)^i}\sqrt{\frac{N}{B}} + \frac{c}{(4c)^i}\frac{T}{B}$$

Summing over all levels $i \leqslant (\log_B \frac{N}{N'})-1$, we find that the total number of nodes visited on those levels is at most:

$$\sum_{i=0}^{\lfloor \log_B \frac{N}{N'}\rfloor -1} \frac{2c}{(2c)^i}\sqrt{\frac{N}{B}} + \frac{c}{(4c)^i}\frac{T}{B} = O\left(\sqrt{\frac{N}{B}}+\frac{T}{B}\right)$$

The higher levels, with less than N' nodes, just add an additive constant, so we conclude that $O(\sqrt{N/B} + T/B)$ nodes are visited in total. □

4.2.3 Multi-dimensional PR-tree

In this section we briefly sketch how our PR-tree generalizes to dimensions greater than two. We focus on how to generalize pseudo-PR-trees, since a d-dimensional PR-tree can be obtained using d-dimensional pseudo-PR-trees in exactly the same way as in the two-dimensional case; that the d-dimensional PR-tree has the same asymptotic performance as the d-dimensional pseudo-PR-tree is also proved exactly as in the two-dimensional case.

Recall that a two-dimensional pseudo-PR-tree is basically a four-dimensional kd-tree, where four priority leaves containing extreme rectangles in each of the four directions have been added below each internal node. Similarly, a d-dimensional pseudo-PR-tree is basically a $2d$-dimensional kd-tree, where each node has $2d$ priority leaves with extreme rectangles in each of the $2d$ standard directions. For constant d, the structure can be constructed in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os using the same grid method as in the two-dimensional case (Section 4.2.1); the only difference is that in order to fit the $2d$-dimensional grid in main memory we have to decrease z (the number of nodes produced in one recursive stage) to $\Theta(M^{1/2d})$.

To analyze the number of I/Os used to answer a window query on a d-dimensional pseudo-PR-tree, we analyze the number of visited internal nodes as in the two-dimensional case (Section 4.2.1); the total number of visited nodes is at most a factor $2d$ higher, since at most $2d$ priority leaves can be visited per internal node visited. As in the two-dimensional case, $O(T/B)$ is a bound on the number of nodes ν visited where all rectangles in at least one of the priority leaves below ν's parent are reported. The number of nodes ν visited such that each priority leaf of ν's parent contains at least one rectangle not intersecting the query can then be bounded using an argument similar to the one used in two dimensions; it is equal to the number of regions associated with the nodes in a $2d$-dimensional kd-tree with $O(N/B)$ leaves that intersect the $(2d-2)$-dimensional intersection of two orthogonal hyperplanes. It follows from a straightforward generalization of the standard kd-tree analysis that this is $O((N/B)^{1-1/d})$ [Aga01BGHH].

Theorem 4.2.5 *A PR-tree on a set of N d-dimensional hyper-rectangles can be bulk-loaded in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os, such that a window query can be answered in $O((N/B)^{1-1/d} + T/B)$ I/Os.*

4.2.4 LPR-tree: doing insertions and deletions

In this section we describe and analyze the logarithmic pseudo-PR-tree, or LPR-tree for short. This tree enables us to maintain an R-tree-like structure efficiently without losing the worst-case optimal query time. The structure of an LPR-tree differs from a normal R-tree in two ways. First, the leaves are on different levels.

Second, the internal nodes store some additional information, which is explained below. Still, exactly the same query algorithms as for a real R-tree can be used on an LPR-tree. We describe the LPR-tree in two dimensions—generalization to higher dimensions can be done in the same way as with PR-trees.

Structure

An LPR-tree consists of a root with a number of subtrees. Each subtree is a normal pseudo-PR-tree, except that the internal nodes (kd-nodes) store some additional information, and the kd-nodes are grouped to share blocks on disk. Both adaptations serve to make efficient deletions possible. We will refer to these subtrees as APR-trees (annotated pseudo-PR-trees).

In each internal node ν of an APR-tree, the following information is stored:

- pointers to all of ν's children, and a bounding box for each child;
- the split value which was used to cut ν in the four-dimensional kd-tree;
- for each priority leaf of ν, the least extreme value of the relevant coordinate of any rectangle stored in that leaf, that is:
  - the largest $x_{\min}$-coordinate in $\nu_p^{x_{\min}}$;
  - the largest $y_{\min}$-coordinate in $\nu_p^{y_{\min}}$;
  - the smallest $x_{\max}$-coordinate in $\nu_p^{x_{\max}}$;
  - the smallest $y_{\max}$-coordinate in $\nu_p^{y_{\max}}$.

Recall that the internal nodes of pseudo-PR-trees have degree only six. In an APR-tree, we group these nodes into blocks as follows. With each internal node at depth i in a tree such that $i = 0 \pmod{\lfloor \log B \rfloor}$, we store, in the same block, all its descendant internal nodes down to level $i + \lfloor \log B \rfloor - 1$. In the following sections, we will keep writing about nodes that share a block as separate nodes, but in the analysis, we will use the fact that we can follow any path of length l down into the tree with only $O(l/\log B)$ I/Os.

An LPR-tree is the structure that results from applying the logarithmic method [Arg03, Pro03] to APR-trees. An LPR-tree has up to $\lceil \log(N/B) \rceil + 3$ subtrees $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_{\lceil \log(N/B) \rceil + 2}$. Subtree $\mathcal{T}_0$ stores at most B rectangles, and $\mathcal{T}_i$ ($i > 0$) is an APR-tree that stores at most $2^{i-1}B$ rectangles. So $\mathcal{T}_0$ and $\mathcal{T}_1$ will never contain more than one leaf; the other subtrees may have more nodes. The smaller subtrees, that is $\mathcal{T}_0$ up to tree $\mathcal{T}_m$ for some $m = \log \frac{M}{B} - O(1)$, have a total size of at most $B + \sum_{i=0}^{\log(M/B)-O(1)} 2^{i-1}B = O(M)$; we keep these subtrees completely in main memory. From the larger subtrees, that is tree $\mathcal{T}_{\lceil \log(N/B) \rceil + 2}$ down to tree $\mathcal{T}_l$, for some $l = \log \frac{N}{M} + O(1)$, we keep the top $i - l$ levels in main memory; these have a total size of $\sum_{i=l}^{\lceil \log(N/B) \rceil + 2} O(2^{i-l}B) = O(\sum_{i=0}^{\log(N/B)-\log(N/M)} 2^i B) = O(M)$. The lower levels of the larger subtrees are stored on disk. If $l > m + 1$, subtrees $\mathcal{T}_{m+1}, ..., \mathcal{T}_{l-1}$ are stored on disk completely.

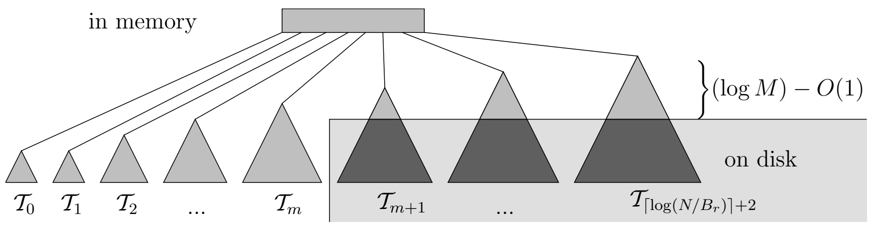


Figure 4.4: An LPR-tree. The bright part is kept in main memory; the dark part is stored on disk.

Algorithms

To *bulk-load* an LPR-tree with a set of N rectangles, we build an APR-tree on the rectangles and store it as $\mathcal{T}_{\lceil\log N/B\rceil+1}$. All other subtrees are left empty.

To *insert* a rectangle R in an LPR-tree, we proceed as follows. Check $\mathcal{T}_0$. If $\mathcal{T}_0$ is full, we find the subtree $\mathcal{T}_j$ with smallest j such that $\mathcal{T}_j$ is empty. We take all rectangles from $\mathcal{T}_0$ to $\mathcal{T}_j$ together and build a new APR-tree $\mathcal{T}_j$ on them. The old APR-trees $\mathcal{T}_i$, for $0 \leqslant i < j$, are discarded. Having made sure that there is space in $\mathcal{T}_0$, we add R to $\mathcal{T}_0$.

To *delete* a rectangle R from an LPR-tree, we proceed as follows. We search for R in each subtree $\mathcal{T}_i$. We start at the root of each subtree; in each internal node ν, we compare R to the information stored about the priority leaves and the split value of ν to decide in which child to continue the search. When we find the leaf that contains R, we delete R from it. If this leaf is a priority leaf ν_p and its parent ν has one or two kd-nodes (λ and possibly μ) as children, we check if ν_p still contains more than $B/2$ rectangles. If this is the case, we are done. Otherwise, we check out the leaves that follow ν_p in the sequence $\nu_p^{x_{\min}}$, $\nu_p^{y_{\min}}$, $\nu_p^{x_{\max}}$, $\nu_p^{y_{\max}}$, the priority leaves of λ (or λ itself, if it is a leaf), and the priority leaves of μ (or μ itself), to find the $B/2$ rectangles in those leaves that are the most extreme in the relevant coordinate. We move those rectangles into ν_p. As a result, one or more of the leaves drawn from may have become underfull, that is: containing $B/2$ rectangles or less; we will replenish them in a similar manner. Leaves that are kd-nodes, and priority leaves of kd-nodes that do not have kd-children, cannot be replenished. We will just leave them underfull, and delete them when they become completely empty.

Every now and then we do a *clean-up*, where we rebuild the entire LPR-tree from scratch (using the bulk-loading algorithm). More precisely, we maintain a counter N_0, which is the number of rectangles present at the last clean-up[4], a counter I, which is the number of insertions since then, and a counter D, which is the number of deletions since then. As soon as $D \geqslant N_0/2$ or $I \geqslant N_0$, we do a clean-up.

[4]When the LPR-tree is initialized by bulk-loading, we set N_0 to the number of rectangles present at bulk-loading, and consider the bulk-loading to be the first clean-up.

Query complexity

Let us first verify that we can query an LPR-tree efficiently.

Lemma 4.2.6 *A window query in an LPR-tree on N rectangles in the plane needs $O(\sqrt{N/B} + T/B)$ I/Os in the worst-case.*

Proof: An APR-tree has a structure very similar to a pseudo-PR-tree, but as a result of deletions, an APR-tree can be somewhat unbalanced. Nevertheless, even after deletions, the kd-nodes in an APR-tree $\mathcal{T}_i$ $(i > 0)$ will always form a subset of the kd-nodes of an APR-tree on at most $2^{i-1}B$ rectangles. Furthermore, the deletion algorithm ensures that the priority leaves in an APR-tree always contain $\Theta(B)$ rectangles, except possibly the priority leaves of kd-nodes that have no kd-children. It is easy to see now that the analysis for pseudo-PR-trees (proof of Lemma 4.2.2) still goes through, if we just write 2^{i-1} rather than $O(N/B)$ for the number of leaves. Thus we find that the number of nodes visited in an APR-tree $\mathcal{T}_i$ is $O(\sqrt{2^{i-1}} + T_i/B)$, where T_i is the number of answers found in $\mathcal{T}_i$.

Taking the sum of the number of I/Os needed for all trees $\mathcal{T}_i$, we find:

$$\sum_{i=m+1}^{\lceil \log \frac{N}{B} \rceil + 2} O\left(\sqrt{2^{i-1}} + \frac{T_i}{B}\right) = O\left(\sqrt{\frac{N}{B}} + \frac{T}{B}\right)$$

□

Bulk-loading complexity

Lemma 4.2.7 *An LPR-tree can be bulk-loaded with $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os.*

Proof: For bulk-loading the APR-tree $\mathcal{T}_{\lceil \log N/B \rceil + 1}$, we use to same algorithm as for pseudo-PR-trees, now storing the additional information and grouping the internal nodes into blocks at no additional I/O-cost. The algorithm uses $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os (Lemma 4.2.3). □

Insertion complexity

Lemma 4.2.8 *Inserting a rectangle in an LPR-tree takes $O(\frac{1}{B}(\log_{M/B} \frac{N}{B})(\log_2 \frac{N}{M}))$ I/Os amortized.*

Proof: We will bound the I/Os spent on the insertions done in between any two clean-up operations, including the I/Os needed for the second clean-up if that was caused by an insertion.

Just after clean-up, N_0 rectangles are present, and they are all stored in $\mathcal{T}_{k+1}$, with $k = \lceil \log N_0/B \rceil$. Recall that an insertion, if $\mathcal{T}_0$ is full, finds the first empty tree $\mathcal{T}_j$, and then constructs $\mathcal{T}_j$ from $\mathcal{T}_1, \ldots, \mathcal{T}_{j-1}$. Tree $\mathcal{T}_{k+1}$ would only become involved in this when a new rectangle is to be inserted after $2^k B$ insertions have filled up all trees $\mathcal{T}_i$ with $0 \leqslant i \leqslant k$. However, this cannot happen before the second clean-up, since a clean-up is done as soon as $N_0 < 2^k B$ rectangles have been inserted—or even earlier, if it is triggered by deletions. If the clean-up is caused by an insertion, we charge the cost, $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os (Lemma 4.2.7), to the $N_0 = \Theta(N)$ insertions that caused it, which is $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os amortized per insertion. (Clean-ups caused by deletions will be charged to the deletions—see Section 4.2.4.)

It remains to account for the construction of trees $\mathcal{T}_j$ with $m \leqslant j \leqslant k$ in between clean-ups (trees $\mathcal{T}_j$ with $0 \leqslant m$ are constructed in main memory). Note that only rectangles inserted since the last clean-up are involved in this. By Lemma 4.2.7, the cost of constructing $\mathcal{T}_j$ is $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os amortized per rectangle in the tree. When $\mathcal{T}_j$ is constructed, all rectangles that are put in come from a tree $\mathcal{T}_i$ with $i < j$. It follows that a rectangle can be included at most $k - m + 1 = O(\log_2 \frac{N}{M})$ times in a new tree that is (partly) built in external memory. This leads to an amortized cost of moving rectangles between clean-ups of $O(\frac{1}{B}(\log_{M/B} \frac{N}{B})(\log_2 \frac{N}{M}))$ I/Os.

Adding the bounds, we see that moving rectangles between clean-ups dominates, and gives the bound claimed. □

Deletion complexity

Lemma 4.2.9 *Deleting a rectangle from an LPR-tree takes* $O((\log_B \frac{N}{M})(\log_2 \frac{N}{M}))$ *I/Os amortized.*

Proof: We first need to find the rectangle. In the worst case, we have to check all APR-trees, $O(\log_2 \frac{N}{M})$ of which are (partially) stored on disk. Since the higher levels of these trees are stored in main memory, and the remaining internal nodes are blocked in groups of height $\Theta(\log_B)$, walking down a path in such an APR-tree takes at most $O(\log_B \frac{N}{M})$ I/Os. In total, $O((\log_B \frac{N}{M})(\log_2 \frac{N}{M}))$ I/Os may be needed to locate the rectangle.

The replenishing of the priority leaves is accounted for as follows. Let the *external height* h of a priority leaf ν_p be the largest number of kd-nodes that are found on any path from ν down into the APR-tree and are stored on disk, or have their priority leaves stored on disk. Since the higher levels of the larger APR-trees are stored in main memory, $h = O(\log_2 \frac{N}{M})$. Let the *rank* of a priority leaf be four times its external height, minus its rank among its siblings, i.e. -1 for $\nu_p^{x_{\min}}$; -2 for $\nu_p^{y_{\min}}$; -3 for $\nu_p^{x_{\max}}$, and -4 for $\nu_p^{y_{\max}}$. When we remove a rectangle from a priority leaf, we put a charge of $2r/B$ in its place, where r is the rank of the priority leaf. We only replenish a priority leaf if it gets half-empty, which implies that it contains a total charge of r. By moving rectangles from lower-ranked priority leaves in, we create gaps in those priority leaves, but since all of these have lower rank, we need to put a total charge of at most $r - 1$ in their place. Hence, the replenishing of a priority leaf ν_p frees a charge of at least 1, which pays for the $O(1)$ I/Os that are needed to replenish ν_p. Replenishing priority leaves thus takes $O(\frac{\max(r)}{B}) = O(\frac{1}{B} \log_2 \frac{N}{M})$ I/Os amortized per deletion.

When D becomes $N_0/2$, which is more than $N/4$, the LPR-tree is rebuilt at a cost of $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os (Lemma 4.2.7). This is $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ per deletion, amortized.

Adding the amortized cost of locating and deleting the rectangle, replenishing the priority leaves and rebuilding the LPR-tree, we find that a deletion takes $O((\log_B \frac{N}{M})(\log_2 \frac{N}{M}))$ I/Os amortized. □

Speeding up deletions

When the insertion of a rectangle leads to building a subtree $\mathcal{T}_i$, we put in all rectangles that were inserted by then and are not stored in a previously built subtree $\mathcal{T}_j$ with $j > i$. This makes it possible to find a rectangle in an LPR-tree without searching all $\log_2 \frac{N}{M}$ subtrees T_i that are stored on disk: we just need to keep track of the time of insertion of each rectangle. When we want to find a particular rectangle in the LPR-tree, we only need to search the subtree T_i that was constructed earliest after the rectangle's insertion in the LPR-tree.

To make this work, we need to keep three additional structures with the LPR-tree:

- a single number *time*, initially set to zero, which we increase with every update so that we can use it to put unique time stamps on update operations;
- a *time index*, implemented as a B-tree of which the top $\Theta(\log_B M)$ levels are kept in main memory; the lower levels are stored on disk. The time index stores for every rectangle in the forest the time stamp of its insertion. It can do so using any type of key that uniquely identifies rectangles.
- in main memory: for each non-empty subtree $\mathcal{T}_i$, the time at which it was built.

The algorithms are modified as follows. When an LPR-tree is cleaned up, we set *time* to zero, and rebuild the time index to store the zero time stamp for all rectangles. When a subtree $\mathcal{T}_i$ is built or modified because of an insertion, we increment *time* and record it as the time of construction of $\mathcal{T}_i$. When a rectangle is inserted, we also increment *time* and insert the rectangle's key with this time stamp in the time index. When a rectangle R is deleted, we query the time index to get the rectangle's time of insertion, find the subtree $\mathcal{T}_i$ that was constructed earliest after the rectangle was inserted, and search only $\mathcal{T}_i$ to find R.

Lemma 4.2.10 *An LPR-tree with time stamps can be bulk-loaded with* $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ *I/Os.*

Proof: The LPR-tree itself is built with $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os (Lemma 4.2.7). The time index can be built in the same time bound: sort the keys of the rectangles in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os and build a B-tree on them. □

Lemma 4.2.11 *Inserting a rectangle in an LPR-tree with time stamps takes* $O((\log_B \frac{N}{M}) + \frac{1}{B}(\log_{M/B} \frac{N}{B})(\log_2 \frac{N}{M}))$ *I/Os amortized.*

Proof: Inserting the rectangle in the LPR-tree takes$O(\frac{1}{B}(\log_{M/B} \frac{N}{B})(\log_2 \frac{N}{M}))$ I/Os amortized (Lemma 4.2.8). Inserting the rectangle in the time index takes $O(\log_B \frac{N}{M})$ I/Os. Add these bounds to get the bound claimed. □

Lemma 4.2.12 *Deleting a rectangle from an LPR-tree with time stamps takes* $O(\log_B \frac{N}{M})$ *I/Os amortized.*

Proof: We first find the rectangle in the time index; this takes $O(\log_B \frac{N}{M})$ I/Os. With the result we determine which subtree $\mathcal{T}_i$ to search; walking down that subtree takes at most $O(\log_B \frac{N}{M})$ I/Os. Replenishing the priority leaves takes $O(\frac{1}{B} \log_2 \frac{N}{M})$ I/Os amortized (see the proof of Lemma 4.2.9), and rebuilding the LPR-tree as soon as $D \geqslant N_0/2$ takes $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os (Lemma 4.2.9), that is $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os amortized per deletion. Adding it all up, we find that a deletion takes $O(\log_B \frac{N}{M})$ I/Os amortized. □

LPR-tree: the bounds

The lemmas above lead to the following theorem.

Theorem 4.2.13 *An LPR-tree with time stamps on a set of N rectangles in the plane can be bulk-loaded in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os, such that a window query can be answered in $O(\sqrt{N/B} + T/B)$ I/Os in the worst case, a rectangle can be inserted in $O((\log_B \frac{N}{M}) + \frac{1}{B}(\log_{M/B} \frac{N}{B})(\log_2 \frac{N}{M}))$ I/Os amortized, and a rectangle can be deleted in $O(\log_B \frac{N}{M})$ I/Os amortized.*

4.2.5 Lower bound for heuristic R-trees

The PR-tree is the first R-tree variant that always answers a window query worst-case optimally. In fact, most other R-tree variants can be forced to visit $\Theta(N/B)$ nodes to answer a query even when no rectangles are reported ($T = 0$). In this section we show how this is the case for the packed Hilbert R-tree, the four-dimensional Hilbert R-tree, and the TGS R-tree.

Theorem 4.2.14 *There exist a set of rectangles S and a window query Q that does not intersect any rectangles in S, such that all $\Theta(N/B)$ nodes are visited when Q is answered using a packed Hilbert R-tree, a four-dimensional Hilbert R-tree, or a TGS R-tree on S.*

Proof: We will construct a set of *points* S such that all leaves in a packed Hilbert R-tree, a four-dimensional Hilbert R-tree, and a TGS R-tree on S are visited when answering a *line* query that does not touch any point. The theorem follows since points and lines are all special rectangles.

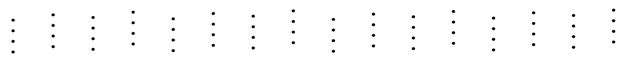


Figure 4.5: Worst-case example

For convenience we assume that $B \geq 4$, $N = 2^k B$ and $N/B = B^m$, for some positive integers k and m, so that each leaf of the R-tree contains B rectangles, and each internal node has fanout B. We construct S as a grid of N/B columns and B rows, where each column is shifted up a little, depending on its horizontal position (each row is in fact a Halton-Hammersley point set; see e.g. [Chz01]). More precisely, S has a point $p_{ij} = (x_{ij}, y_{ij})$, for all $i \in \{0, ..., N/B - 1\}$ and $j \in \{0, ..., B - 1\}$, such that $x_{ij} = i + 1/2$, and $y_{ij} = j/B + h(i)/N$. Here $h(i)$ is the number obtained by reversing, i.e. reading backwards, the k-bit binary representation of i. An example with $N = 64, B = 4$ is shown in Figure 4.5.

Now, let us examine the structure of each of the three R-tree variants on this dataset.

Packed Hilbert R-tree: The packed Hilbert R-tree sorts the points by the Hilbert values. To compare the Hilbert values of two points, we must first check if they lie in the same quadrant of a sufficiently large square whose sides are powers of two: in our case, a square of size 2^k suffices. If they lie in the same quadrant, we zoom in on that quadrant, and see if they lie in the same subquadrant of that quadrant. We keep zooming in until we arrive at the level where the points lie in different quadrants. Then, we decide which quadrant comes first on the Hilbert curve.

Now consider two points p_{ij} and $p_{i'j'}$. Note that both y_{ij} and $y_{i'j'}$ are smaller than 1, so all bits before the “decimal point” of the y-coordinates of these points are the same, namely zero. In addition, if $i = i'$, then $x_{ij} = x_{i'j'}$, and therefore, starting from a square of size 2^k, we have to zoom in more than k levels deep to distinguish between the positions of p_{ij} and $p_{i'j'}$ on the Hilbert curve. On the other hand, if $i \neq i'$, then $|x_{ij} - x_{i'j'}| \geqslant 1$, so the x-coordinates of p_{ij} and $p_{i'j'}$ differ in at least one of the bits before the ”decimal” point. Hence, we do not have to zoom in more than k levels and can compare them on the basis of i and i' only, ignoring j and j'. As a result, the Hilbert curve visits the columns in our grid of points one by one, and when it visits a column, it visits all points in that column before proceeding to another column. Therefore, the packed Hilbert R-tree makes a leaf for every column, and a horizontal line can be chosen to intersect all these columns while not touching any point.

Four-dimensional Hilbert R-tree: The analysis is similar to the one for the packed Hilbert R-trees.

TGS R-tree: The TGS algorithm will partition S into B subsets of equal size and partition each subset recursively. The partitioning is implemented by choosing a partitioning line that separates the set into two subsets (whose sizes are multiples of N/B), and then applying binary partitions to the subsets recursively until we have partitioned the set into N subsets of size N/B. Observe that on all

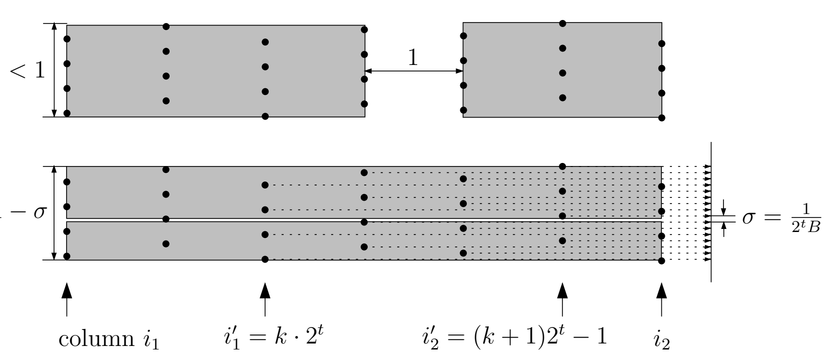


Figure 4.6: TGS partitioning the worst-case example. A vertical division creates two bounding boxes with a total area of less than $i_2 - i_1 - 1$. A horizontal division creates two bounding boxes with a total area of more than $(i_2 - i_1)(1 - 2\sigma) > i_2 - i_1 - 1$.

levels in this recursion, the partitioning line will leave at least a fraction $1/B$ of the input on each side of the line. Below we prove that TGS will always partition by vertical lines; it follows that TGS will eventually put each column in a leaf. Then a line query can intersect all leaves but report nothing.

Suppose TGS is about to partition the subset $S(i_1, i_2)$ of S that consists of columns i_1 to i_2 inclusive, with $i_2 > i_1$, i.e. $S(i_1, i_2) = \{p_{ij} | i \in \{i_1, ..., i_2\}, j \in \{0, ..., B-1\}\}$. When the greedy split algorithm gets to divide such a set into two, it can look for a vertical partitioning line or for a horizontal partitioning line. Intuitively, TGS favors partitioning lines that create a big gap between the bounding boxes of the points on each side of the line. As we will show below, we have constructed S such that the area of the gap created by a horizontal partitioning line is always roughly the same, as is the area of the gap created by a vertical line, with the latter always being bigger.

Partitioning with a vertical line would always leave a gap of roughly a square that fits between two columns—see Figure 4.6. More precisely, it would partition the set $S(i_1, i_2)$ into two sets $S(i_1, c-1)$ and $S(c, i_2)$, for some $c \in \{i_1 + 1, ..., i_2\}$. The bounding boxes of these two sets would each have height less than 1, and their total width would be $(c - 1 - i_1) + (i_2 - c)$, so their total area A_v would be less than $i_2 - i_1 - 1$.

The width of a gap around a horizontal partitioning line depends on the number of columns in $S(i_1, i_2)$. However, the more columns are involved the bigger the density of the points in those columns when projected on the y-axis, and the lower the gap that can be created—see Figure 4.6 for an illustration. As a result, partitioning with a horizontal line can lead to gaps that are wide and low, or relatively high but not so wide; in any case, the area of the gap will be roughly the same. More precisely, we can estimate the total area of the bounding boxes resulting from partitioning with a horizontal line as follows. The partitioning line must leave at least a fraction $1/B$ of the points on each side, so there must be at least one full row of $S(i_1, i_2)$ on each side of the line. Hence, the width of both bounding boxes resulting from the partition step must be $i_2 - i_1$. Observe that the set $\{i_1, ..., i_2 + 1\}$ contains at least two different multiples of 2^s if s is such that $i_2+2-i_1 \geqslant 2^{s+1}$; let t be the largest such value of s, i.e. $t = \lfloor \log(i_2+2-i_1)-1 \rfloor$, let i_1' be the smallest multiple of 2^t that is at least i_1, and let i_2' be $i_1' + 2^t - 1$. Note that if we let i go through all values $\{i_1', ..., i_2'\}$, then the first $k - t$ bits of the k-bit representation of i remain constant, while the last t bits assume all possible values. Consequently, the last $k - t$ bits of $h(i)$ will remain constant, while the first t bits will assume all possible values. Hence, if we project all points in $S(i_1, i_2)$ on the y-axis, the distance between each pair of consecutive points is at most $\sigma = 2^{k-t}/N = 1/(2^t B)$, and the distance between the topmost and the bottommost point is at least $1 - \sigma$. When we partition this set by a horizontal line, the total height of the resulting bounding boxes must be at least $1 - 2\sigma$, and their total area A_h must be at least $(i_2 - i_1)(1 - 2\sigma) = i_2 - i_1 - 2(i_2 - i_1)/(2^t B)$. With $t \geqslant \log(i_2 + 2 - i_1) - 1$, we find that A_h is more than $i_2 - i_1 - 4/B$.

Recall that A_v is less than $i_2 - i_1 - 1$. Since $B \geqslant 4$, we can conclude that $A_h > A_v$, and that partitioning with a vertical line will always result in a smaller total area of bounding boxes than with a horizontal line. As a result, TGS will always cut vertically between the columns. □

4.3 Experiments

In this section we describe the results of our experimental study of the performance of the PR-tree. We compared the PR-tree to several other bulk-loading methods known to generate query-efficient R-trees: The packed Hilbert R-tree (denoted H in the rest of this section), the four-dimensional Hilbert R-tree (denoted H4), and the TGS R-tree (denoted TGS). Among these, TGS has been reported to have the best query performance, but it also takes many I/Os to bulk-load. In contrast, H is simple to bulk-load, but it has worse query performance because it does not take the extent of the input rectangles into account. H4 has been reported to be inferior to H [Kam93], but since it takes the extent into account (like TGS) it should intuitively be less vulnerable to extreme datasets.

4.3.1 Experimental setup

We implemented the four bulk-loading algorithms in C++ using TPIE [Arg02]. TPIE is a library that provides support for implementing I/O-efficient algorithms and data structures. In our implementation we used 36 bytes to represent each input rectangle; 8 bytes for each coordinate and 4 bytes to be able to hold a pointer to the original object. Each bounding box in the internal nodes also used 36 bytes; 8 bytes for each coordinate and 4 bytes for a pointer to the disk block storing the root of the corresponding subtree. The disk block size was chosen to be 4KB, resulting in a maximum fanout of 113. This is similar to earlier experimental studies, which typically use block sizes ranging from 1KB to 4KB or fix the fanout to a number close to 100.

As experimental platform we used a dedicated Dell PowerEdge 2400 workstation with one Pentium III/500MHz processor running FreeBSD 4.3. A local 36GB SCSI disk (IBM Ultrastar 36LZX) was used to store all necessary files: the input data, the R-trees, as well as temporary files. We restricted the main memory to 128MB and further restricted the amount of memory available to TPIE to 64MB; the rest was reserved to operating system daemons.

4.3.2 Datasets

We used both real-life and synthetic data in our experiments.

Real-life data

As the real-life data we used the TIGER/Line data [Tiger] of geographical features in the United States. This data is the standard benchmark data used in spatial databases. It is distributed on six CD-ROMs and we chose to experiment with the road line segments from two of the CD-ROMs: disk one containing data for sixteen eastern US states and disk six containing data from five western US states; we use Eastern and Western to refer to these two datasets, respectively. To obtain datasets of varying sizes we divided the Eastern dataset into five regions of roughly equal size, and then put an increasing number of regions together to obtain datasets of increasing sizes. The largest set is just the whole Eastern dataset. For each dataset we used the bounding boxes of the line segments as our input rectangles. As a result, the Eastern dataset had 16.7 million rectangles, for a total size of 574MB, and the Western data set had 12 million rectangles, for a total size of 411MB. Refer to Table 4.1 for the sizes of each of the smaller Eastern datasets. Note that the biggest dataset is much larger than those used in previous works (which only used up to 100,000 rectangles) [Kam93, Grc98a]. Note also that our TIGER data is relatively nicely distributed; it consist of relatively small rectangles (long roads are divided into short segments) that are somewhat (but not too badly) clustered around urban areas.

dataset:	1	2	3	4	5
million rectangles:	2.08	5.67	9.16	12.66	16.72
size (MB):	72	194	315	435	574

Table 4.1: The sizes of the Eastern datasets

Synthetic data

To investigate how the different R-trees perform on more extreme datasets than the TIGER data, we generated a number of synthetic datasets. Each of these synthetic datasets consisted of 10 million rectangles (or 360MB) in the unit square.

- SIZE(max_side): We designed the first class of synthetic datasets to investigate how well the R-trees handle rectangles of different sizes. In the SIZE(max_side) dataset the rectangle centers were uniformly distributed and the lengths of their sides uniformly and independently distributed between 0 and max_side. When generating the datasets, we discarded rectangles that were not completely inside the unit square (but made sure each dataset had 10 million rectangles). A portion of the dataset SIZE(0.001) is shown in Figure 4.7.

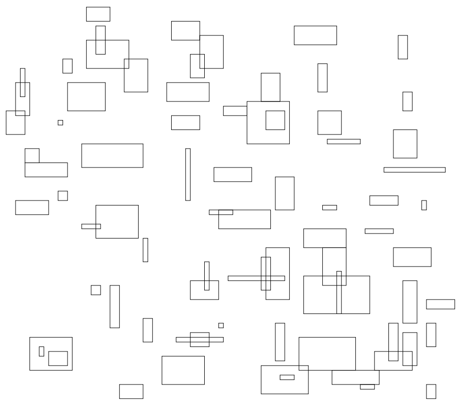


Figure 4.7: Synthetic dataset SIZE(0.001)

- ASPECT(a): The second class of synthetic datasets was designed to investigate how the R-trees handle rectangles with different aspect ratios. The areas of the rectangles in all the datasets were fixed to 10^{-6}, a reasonably small size. In the ASPECT(a) dataset the rectangle centers were uniformly distributed but their aspect ratios were fixed to a and the longest sides chosen to be vertical or horizontal with equal probability. We also made sure that all rectangles fell completely inside the unit square. A portion of the dataset ASPECT(10) is shown in Figure 4.8. Note that if the input rectangles are bounding boxes of line segments that are almost horizontal or vertical, one will indeed get rectangles with very high aspect ratio—even infinite in the case of horizontal or vertical segments.

- SKEWED(c): In many real-life multidimensional datasets different dimensions often have different distributions. Some of these distributions may be highly skewed compared to the others. We designed the third class of datasets to investigate how this affects R-tree performance. SKEWED(c) consists of uniformly distributed points that have been "squeezed" in the y-dimension, that is, each point (x, y) is replaced with (x, y^c). An example of SKEWED(5) is shown in Figure 4.9.

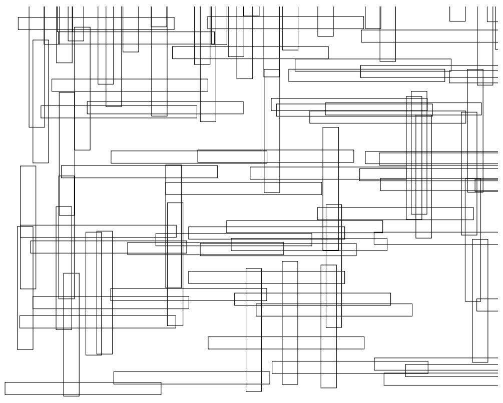


Figure 4.8: Synthetic dataset ASPECT(10)

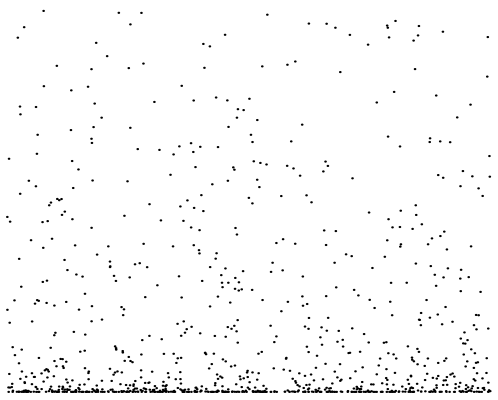


Figure 4.9: Synthetic dataset SKEWED(5)

- CLUSTER: Our final dataset was designed to illustrate the worst-case behavior of the H, H4 and TGS R-trees. It is similar to the worst-case example discussed in Section 4.2. It consists of 10 000 clusters with centers equally spaced on a horizontal line. Each cluster consists of 1000 points uniformly distributed in a $0.000\,01 \times 0.000\,01$ square surrounding its center. Figure 4.10 shows a part of the CLUSTER dataset.

Figure 4.10: Synthetic dataset CLUSTER

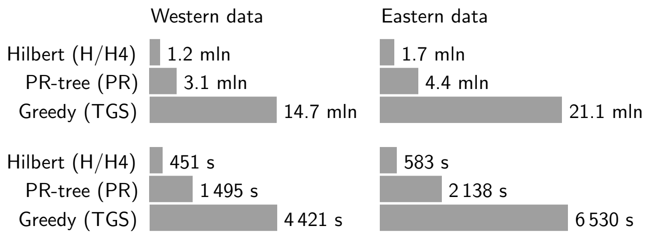


Figure 4.11: Bulk-loading performance on TIGER data: I/O (upper figure) and time (lower figure).

4.3.3 Experimental results

Below we discuss the results of our bulk-loading and query experiments with the four R-tree variants.

Bulk-loading performance

We bulk-loaded each of the R-trees with each of the real-life TIGER datasets, as well as with the synthetic datasets for various parameter values. In all experiments and for all R-trees we achieved a space utilization above 99%.[5] We measured the time spent and counted the number of 4KB blocks read or written when bulk-loading the trees. Note that all algorithms we tested read and write blocks almost exclusively by sequential I/O of large parts of the data; as a result, I/O is much faster than if blocks were read and written in random order.

Figure 4.11 shows the results of our experiments using the Eastern and Western datasets. Both experiments yield the same result: The H and H4 algorithms use the same number of I/Os, and roughly 2.5 times fewer I/Os than PR. This is not surprising since even though the three algorithms have the same $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/O bounds, the PR algorithm is much more complicated than the H and H4 algorithms. The TGS algorithm uses roughly 4.5 times more I/Os than PR, which is also not surprising given that the algorithm makes binary partitions so that the number of levels of recursion is effectively $O(\log_2 N)$. In terms of time, the H and H4 algorithms are still more than 3 times faster than the PR algorithm, but the TGS algorithm is only roughly 3 times slower than PR. This shows that H, H4 and PR are all more CPU-intensive than TGS.

Figure 4.12 shows the results of our experiments with the five Eastern datasets. These experiments show that the H, H4 and PR algorithms scale relatively linearly with dataset size; this is a result of the $\lceil\log_{M/B}\frac{N}{B}\rceil$ factor in the bulk-loading bound being the same for all datasets. For H and H4 this means that the core step

[5]When R-trees are bulk-loaded to subsequently be updated dynamically, near 100% space utilization is often not desirable [Dwt94]. However, since we are mainly interested in the query performance of the R-tree constructed with the different bulk-loading methods, and since the methods could be modified in the same way to produce non-full leaves, we only considered the near 100% utilization case.

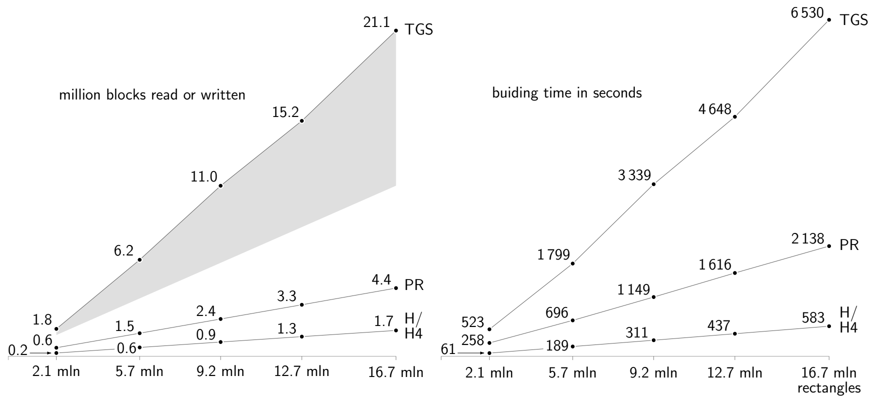


Figure 4.12: Bulk-loading performances on Eastern datasets: I/O (left) and time (right)

of the algorithm, which is sorting the rectangles by their position on the Hilbert curve, runs in the same number of passes—namely two—in all experiments. For the PR algorithm it means that only once, we have to build a grid and divide among its cells a set of rectangles that is too big too fit in memory. All recursive steps can be done in main memory. The cost of the TGS algorithm seems to grow in an only slightly superlinear way with the size of the data set. This is a result of the $\lceil \log_B N \rceil$ factor in the bulk-loading bound being almost the same for all data sets. It means that in all data sets, the subtrees just below root level have roughly the same size ($B^{\lceil \log_B N \rceil - 1}$ rectangles), and since each of them is too big to fit in main memory, we need a significant number of I/Os to build them. This, together with preprocessing, accounts for a big portion of the number of I/Os that scales linearly with the size of the data set. The slightly superlinear trend comes from the cost of building the root, which varies from roughly 20% of the total number of I/Os on set 1, to roughly 45% on set 5 (shaded in Figure 4.12).

In our experiments with the synthetic data we found that the performance of the H, H4 and PR bulk-loading algorithms was practically the same for all the datasets, that is, unaffected by the data distribution. This is not surprising, since the performance should only depend on the dataset size (and all the synthetic datasets have the same size). The PR algorithm performance varied slightly, which can be explained by the small effect the data distribution can have on the grid method used in the bulk-loading algorithm (subtrees may have slightly different sizes due to the removal of priority boxes). On average, the H and H4 algorithms spent 381 seconds and 1.0 million I/Os on each of the synthetic datasets, while the PR algorithm spent 1289 seconds and 2.6 million I/Os. On the other hand, as expected, the performance of the TGS algorithm varied significantly over the synthetic datasets we tried; the binary partitions made by the algorithm depend heavily on the input data distribution. The TGS algorithm was between 4.6 and 16.4 times slower than the PR algorithm in terms of I/O, and between 2.8 and 10.9

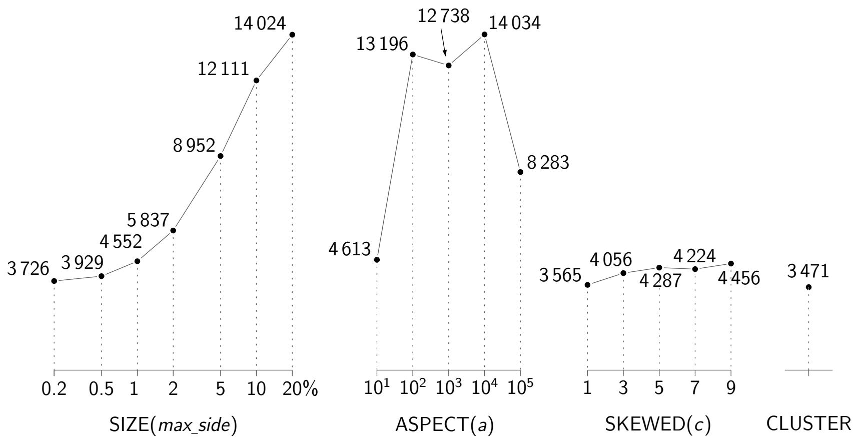


Figure 4.13: Bulk-loading time in seconds of Top-down Greedy Split on synthetic data sets of 10 million rectangles (SIZE and ASPECT) or points (SKEWED and CLUSTER) each.

times slower in terms of time. The performance of TGS on the SIZE(max_side), ASPECT(a), SKEWED(c) and CLUSTER datasets is shown in Figure 4.13.

Query performance

After bulk-loading the four R-tree variants we experimented with their query performance; in each of our experiments we performed 100 randomly generated queries and computed their average performance (a more exact description of the queries is given below). Following previous experimental studies, we utilized a cache (or "buffer") to store internal R-tree nodes during queries. In fact, in all our experiments we cached all internal nodes since they never occupied more than 6MB. This means that when reporting the number of I/Os needed to answer a query, we are in effect reporting the number of leaves visited in order to answer the query.[6] For several reasons, and following previous experimental studies [Bmn90, Grc98a, Kam93, Kam94], we did not collect timing data. Two main reasons for this are (1) that I/O is a much more robust measure of performance, since the query time is easily affected by operating system caching and by disk block layout; and (2) that we are interested in heavy load scenarios where not much cache memory is available or where caches are ineffective, that is, where I/O dominates the query time.

TIGER data: We first performed query experiments using the Eastern and Western datasets. The results are summarized in Figure 4.14 and 4.15. In Figure 4.14 we show the results of experiments with square window queries with

[6]Experiments with the cache disabled showed that in our experiments the cache actually had relatively little effect on the window query performance.

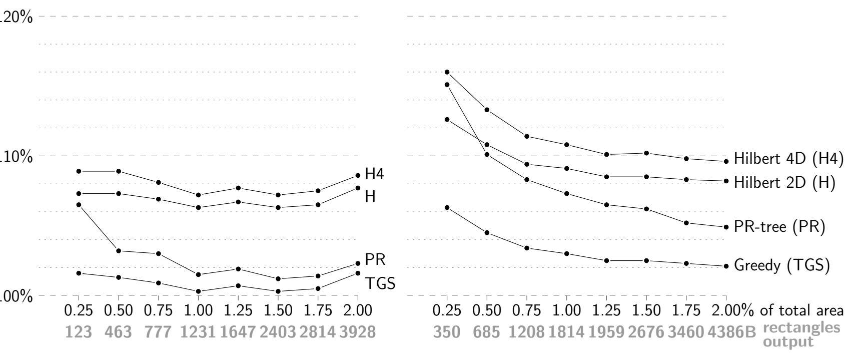


Figure 4.14: Query performance for queries with squares of varying size on the Western TIGER data (left) and the Eastern TIGER data (right). The performance is given as the number of blocks read divided by the output size T/B.

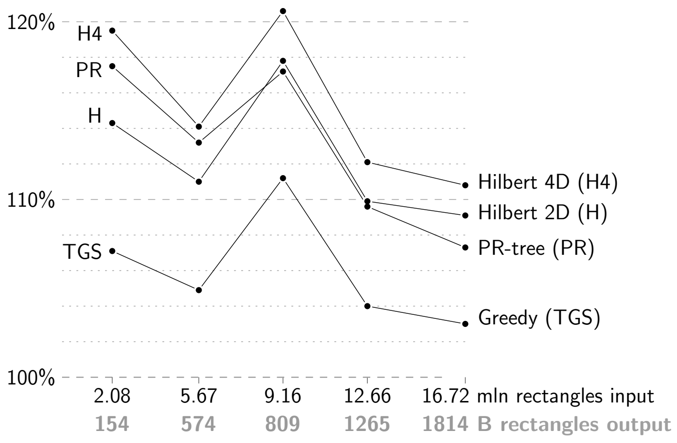


Figure 4.15: Query performance for queries with squares of area 0.01 on Eastern TIGER data sets of varying size. The performance is given as the number of blocks read divided by the output size T/B.

areas that range from 0.25% to 2% of the area of the bounding box of all input rectangles. We used smaller queries than previous experimental studies (for example, the maximum query in [Kam93] occupies 25% of the area) because our datasets are much larger than the datasets used in previous experiments—without reducing the query size the output would be unrealistically large and the reporting cost would thus dominate the overall query performance. In Figure 4.15 we show the results of experiments on the five Eastern datasets of various sizes with a fixed query size of 1%. The results show that all four R-tree variants perform remark-

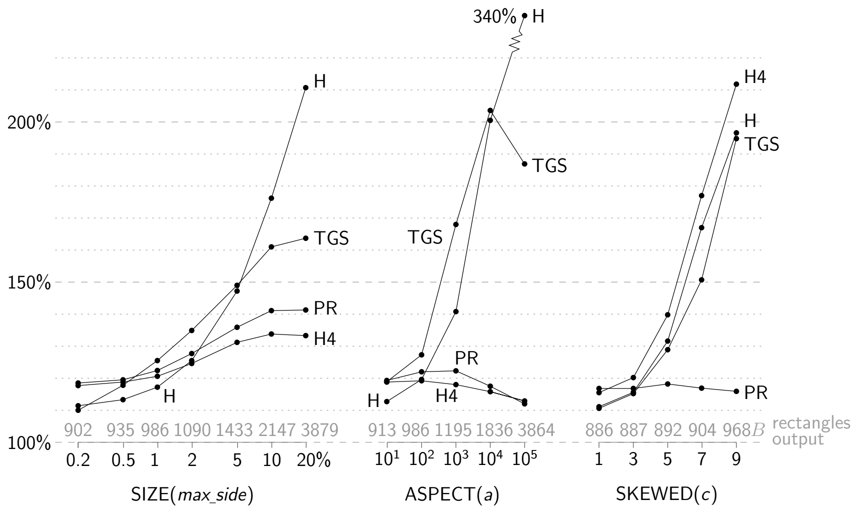


Figure 4.16: Query performance for queries with squares of area 0.01 on synthetic data sets. The performance is given as the number of blocks read divided by the output size T/B.

ably well on the TIGER data; their performance is within 10% of each other and they all answer queries in close to T/B, the minimum number of necessary I/Os. Their relative performance generally agrees with earlier results [Kam93, Grc98a], that is, TGS performs better than H, which in turn is better than H4. PR consistently performs slightly better than both H and H4 but slightly worse than TGS.

Synthetic data. Next we performed experiments with our synthetic datasets, designed to investigate how the different R-trees perform on more extreme data than the TIGER data. For each of the datasets SIZE, ASPECT and SKEWED we performed experiments where we varied the parameter to obtain data ranging from fairly normal to rather extreme. Below we summarize our results.

The left side of Figure 4.16 shows the results of our experiments with the dataset SIZE(max_side) when varying max_side from 0.002 to 0.2, that is, from relatively small to relatively large rectangles. As queries we used squares with area 0.01. Our results show that for relatively small input rectangles, like the TIGER data, all the R-tree variants perform very close to the minimum number of necessary I/Os. However, as the input rectangles get larger, PR and H4 clearly outperform H and TGS. H performs the worst, which is not surprising since it does not take the extent of the input rectangles into account. TGS performs significantly better than H but still worse than PR and H4. Intuitively, PR and H4 can handle large rectangles better, because they rigorously divide rectangles into groups of rectangles that are similar in all four coordinates. This may enable these algorithms to group likely answers, namely large rectangles, together so that they can be retrieved with few I/Os. It also enables these algorithms to group small rectangles nicely, while TGS, which strives to minimize the total area of bounding boxes, may be indifferent to the distribution of the small rectangles in the presence of large rectangles.

The middle of Figure 4.16 shows the results of our experiments with the dataset ASPECT(a), when we vary a from 10 to 10^5, that is, when we go from rectangles (of constant area) with small to large aspect ratio. As query we again used squares with area 0.01. The results are very similar to the results of the SIZE dataset experiments, except that as the aspect ratio increases, PR and H4 become significantly better than TGS and especially H. Unlike with the SIZE dataset, PR performs as well as H4 and they both perform close to the minimum number of necessary I/Os to answer a query. Thus this set of experiments re-emphasizes that both the PR-tree and H4-tree are able to adopt to varying extent very well.

The right side of Figure 4.16 shows the result of our experiments with the dataset SKEWED(c), when we vary c from 1 to 9, that is, when we go from a uniformly distributed point set to a very skewed point set. As query we used squares with area 0.01 that are skewed in the same way as the dataset (that is, where the corner (x, y) is transformed to (x, y^c)) so that the output size remains roughly the same. As expected, the PR performance is unaffected by the transformations, since our bulk-loading algorithm is based only on the relative order of coordinates: x-coordinates are only compared to x-coordinates, and y-coordinates are only compared to y-coordinates; there is no interaction between them. On the other hand, the query performance of the three other R-trees degenerates quickly as the point set gets more skewed.

As a final experiment, we queried the CLUSTER dataset with long skinny horizontal queries (of area 1×10^{-7}) through the 10 000 clusters; the y-coordinate of the leftmost bottom corner was chosen randomly such that the query passed through all clusters. The results are shown in Table 4.2. As anticipated, the query performance of H, H4 and TGS is very bad; the CLUSTER dataset was constructed to illustrate the worst-case behavior of the structures. Even though a query only returns around 0.3% of the input points on average, the query algorithm visits 37%, 94% and 25% of the leaves in H, H4 and TGS, respectively. In comparison, only 1.2% of the leaves are visited in PR. Thus the PR-tree outperforms the other indexes by well over an order of magnitude.

tree:	H	H4	PR	TGS
# I/Os:	32 920	83 389	1 060	22 158
% of the R-tree visited:	37%	94%	1.2%	25%

Table 4.2: Query performances on synthetic dataset CLUSTER.

4.3.4 Conclusions of the experiments

The main conclusion of our experimental study is that the PR-tree is not only theoretically efficient but also practically efficient. Our bulk-loading algorithm is slower than the packed Hilbert and four-dimensional Hilbert bulk-loading algorithms but much faster than the TGS R-tree bulk-loading algorithm. Furthermore, unlike for the TGS R-tree, the performance of our bulk-loading algorithm does not depend on the data distribution. The query performance of all four R-trees is excellent on nicely distributed data, including the real-life TIGER data. On extreme data however, the PR-tree is much more robust than the other R-trees (even though the four-dimensional Hilbert R-tree is also relatively robust).

4.4 Concluding remarks

In this paper we presented the PR-tree, which is the first R-tree variant that can answer any window query in the optimal $O(\sqrt{N/B} + T/B)$ I/Os. We also performed an extensive experimental study, which showed that the PR-tree is not only optimal in theory, but that it also performs excellent in practice: for normal data, it is quite competitive to the best known heuristics for bulk-loading R-trees, namely the packed Hilbert-R-tree [Kam93] and the TGS R-tree [Grc98a], while for data with extreme shapes or distributions, it outperforms them significantly.

The PR-tree can be updated using any known update heuristic for R-trees, but then its performance cannot be guaranteed theoretically anymore and its practical performance might suffer as well. Alternatively, we can use the dynamic version of the PR-tree using the logarithmic method, which has the same theoretical worst-case query performance and can be updated efficiently. In the future we wish to experiment to see what happens to the performance when we apply heuristic update algorithms and when we use the theoretically superior logarithmic method.

Chapter 5

Box-trees for collision checking in industrial installations

An extended abstract of this chapter appeared as: Herman J. Haverkort, Mark de Berg and Joachim Gudmundsson: Box-trees for collision checking in industrial installations, in *Proc. 18th ACM Symposium on Computational Geometry*, Barcelona, 2002, pages 53–62. Full text to appear in *Computational Geometry – Theory and Applications*.

Abstract. *A box-tree is a bounding-volume hierarchy that uses axis-aligned boxes as bounding volumes. We describe a new algorithm to construct a box-tree for objects in a 3D scene, and we analyze its worst-case query time for approximate range queries. If the input scene has certain characteristics that we derived from our application—collision detection in industrial installations—then the query times are polylogarithmic, not only for searching with boxes but also for range searching with other constant-complexity ranges.*

5.1 Introduction

Motivation. Collision checking is an important operation in all applications where objects move around in a 3D scene—virtual reality, computer animation, and robotics are obvious examples. A popular way of doing collision checking is the following two-phase approach. In the first phase, the *filtering phase*, one finds all primitive objects in the scene whose bounding box intersects the query object (or its bounding box). In the second phase, the *refinement phase*, one tests for each of these primitives (if any) whether it actually intersects the object. To speed up the filtering phase, the set S of bounding boxes of the primitives in the

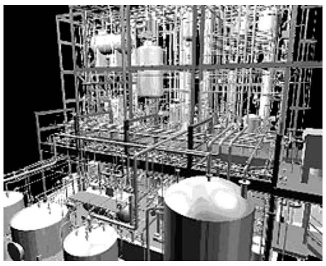


Figure 5.1: CAD model of a carbon black unit. Designed by OLAJTERV Process and Energy, Hungary.

scene is often stored in a bounding-volume hierarchy. This is a binary tree whose leaves store the boxes in S, and where each internal node ν stores the bounding box $b(\nu)$ of all boxes stored in the subtree rooted at ν. We call such a tree a *box-tree*; sometimes it is more precisely called an *axis-aligned-bounding-box tree*, or *AABB-tree* for short. A query with a query range Q is performed by traversing the tree in a top-down manner, only visiting nodes ν such that $b(\nu)$ intersects Q. This way we end up exactly in the leaves storing boxes that intersect Q.

The query time in a box-tree is determined by the number of nodes visited, and the goal is therefore to organize the tree in such a way that this number is kept as small as possible. Agarwal *et al.* [Aga01BGHH] recently showed that a box-tree exists that has $O(n^{2/3} + k)$ query time for ranges that are axis-parallel boxes, where n is the total number of boxes in S and k is the number of boxes intersecting the query range. This bound is rather disappointing: if the query time would really be that bad, box-trees would not be used so much in practice. Unfortunately, the bound is optimal. Agarwal *et al.* prove that there are sets of input boxes for which the worst-case query time of any box-tree is $\Omega(n^{2/3} + k)$.[1] This is the starting point for our work: we want to understand what makes box-trees perform well in practical applications even though in theory they may perform badly.

The application we have in mind comes from the MOLOG project [Molog]. The goal of this project is to add motion support to CAD systems used to design large industrial installations, such as depicted in Fig. 5.1.

Adding motion support will help the designer of an industrial installation to decide whether it will be possible to move certain parts out of the installation, for maintenance or replacement. The approach taken in the MOLOG project is based on the *probabilistic path planner* [Ama96, Kav95, Sve97], a technique for motion planning that has proved very successful in many applications. A basic test performed many times by the probabilistic path planner is collision checking: given a query object—the object for which we are planning a motion, at a certain position and orientation—does it collide with the CAD model? We can now state

[1]In general, the worst-case query time of a box-tree in d-dimensional space is $\Theta(n^{1-1/d} + k)$. In this paper we focus on 3-dimensional box-trees, because this is most natural in our application.

the goal of this paper as follows: we want to design a provably efficient box-tree for storing scenes that are CAD models of large industrial installations.

Further background. The lower bounds of Agarwal *et al.* mentioned earlier imply that, to be able to design provably efficient box-trees for CAD models of large industrial installations, we have to make use of the properties of the bounding boxes of the primitives in such CAD models. The *realistic input models* [Brg97KSV] suggested in the literature do not seem applicable in our setting: the industrial installation of Fig. 5.1, for instance, contains many long and thin pipes that are relatively close together. But if we forget about the pipes, the scene seems to be well-behaved. Hence, the assumption we make is that the boxes in S can be partitioned into two subsets, one containing only long and thin (almost) disjoint pipes, and one forming a low-density scene [Brg97KSV]. Here a pipe is defined to be an axis-aligned box whose shortest dimension is at most a constant β times shorter than its middle dimension—see Section 5.2.3 for formal definitions of these concepts. It is important to note that our algorithm to construct the box-tree does not need this assumption; we only use it in the analysis.

Unfortunately, with the assumption just stated one still cannot prove good bounds: the $\Omega(n^{2/3} + k)$ lower bound for range queries with a box even holds if the input consists of disjoint unit squares arranged in a grid-like fashion. Therefore we analyze approximate range queries. More precisely, instead of the parameter k in the time bound, we use k_ϵ, which is the number of boxes intersecting the extended range Q_ϵ. For a given $\epsilon > 0$, the extended range Q_ϵ is the set of points lying at L_∞-distance at most ϵw from Q, where w is the length of the longest edge of Q. The expectation is that in practice k_ϵ will not be much larger than k for moderately small ϵ, at least when the query range is rather fat. Note that in our application, the query range is (the bounding box of) an object for which we are planning a motion. If the object is a forklift truck or some other car-like device, its bounding box is likely to be fat. The concept of approximate range searching was also used by Arya and Mount [Ary00], who considered approximate range queries on a set of points. The parameter ϵ is not used by our query algorithm—the algorithm still visits only nodes whose bounding boxes are intersected by Q—but it is only used in the analysis. (So perhaps approximate range searching is a slight misnomer.)

Our results. We describe a new, simple algorithm to construct a box-tree on a set of boxes in 3D. This algorithm generalizes the 2D kd-interval tree described by Agarwal *et al.* [Aga01BGHH] to 3D, with one additional crucial twist: We partition the input boxes into three subsets, according to the orientation of their longest edge, and construct separate box-trees for these subsets; these subtrees are then combined to form the final tree. Our main contribution is a rather involved analysis of the worst-case query time of this box-tree in the setting described above, showing it is polylogarithmic. More precisely, we prove that the number of visited nodes is $O(\frac{1}{\epsilon}(\frac{1}{\epsilon} + \lambda)\log^4 n + k_\epsilon)$, where λ is a constant depending on the scene parameters. Typically, λ will only be large if the input contains many flat 'plates' that are very close together—see section 5.2.2 for details. Note that the choice of ϵ determines a trade-off between the terms in the bound: choosing ϵ small will cause a large factor in the first term, but k_ϵ will be close to k. On the other hand, choosing ϵ big keeps the first term down, but k_ϵ might grow to $O(n)$. In any case, since ϵ is only a parameter in the analysis and not for the algorithm, the bound on the query time will be the lowest bound over all possible values of ϵ; in other words: $O(\min_{0<\epsilon\leqslant 1}\{\frac{1}{\epsilon}(\frac{1}{\epsilon}+\lambda)\log^4 n + k_\epsilon\})$.

This result should be compared with the results for approximate range searching in a set of points in 3-space. Here, the best result that uses boxes as bounding volumes is by Dickerson *et al.* [Dic00], who show that the query time in a so-called *longest-side-first kd-tree* is $O(\min_{0<\epsilon\leqslant 1}\{(\frac{1}{\epsilon})^2 \log^3 n + k_\epsilon\})$. Our result is more general than this, as we store boxes instead of points and the bounds we get are only slightly worse.

We have also designed a variant of the box-tree, where an interior node uses a different type of bounding volume: instead of a bounding box, it can use a donut-like shape, namely the difference of two boxes. This was inspired by Arya and Mount [Ary00], who show that a similar structure for points—they call it BBD-tree—outperforms kd-trees in the worst case: the time for approximate range queries in 3D in a BBD-tree is $O(\min_{0<\epsilon\leqslant 1}\{\log n + (\frac{1}{\epsilon})^2 + k_\epsilon\})$. (The same result can be obtained using BAR-trees [Dun99, Dun99GK]. BAR-trees use convex, but not necessarily axis-parallel, bounding volumes whose facets have a bounded number of different orientations.) In our case, a similar improvement is possible: our *BBD-interval tree* has a worst-case query time of $O(\min_{0<\epsilon\leqslant 1}\{\log^3 n + \frac{\lambda}{\epsilon}\log^2 n + (\frac{1}{\epsilon})^2 \log n + k_\epsilon\})$. However, despite the fact that the theoretical asymptotic bounds of the BBD-interval tree are better than those of the kd-interval tree, we will only describe the latter in this article. There are two reasons for this. First, the analysis of the kd-interval tree will already demonstrate all of the main ideas, and thus everything which might inspire future research. The BBD-interval tree has little to add: it combines the ideas described in this article with the principles of the BBD-tree, but it takes many pages of tedious analysis to describe and analyse how we can get the details of the BBD-interval tree right. Second, the details being much more complex than those of the kd-interval tree, the BBD-interval tree is probably relatively cumbersome to implement and will have significantly higher hidden constants in the asymptotic bounds. For this reason, we think that the kd-interval tree is more likely to be the structure of choice in practice. Therefore, we will only describe the latter in this article. The details of the BBD-interval tree can be found in the appendix of the technical report version of this article [Hav02a].

Finally, in this article we extend our results to constant-complexity query ranges of arbitrary shape, showing that the time for approximate queries with such ranges is $O(\min_{0<\epsilon\leqslant 1}\{(\lambda/\epsilon^2)\log^4 n + k_\epsilon\})$ in our LSF-interval tree—in a BBD-interval tree, this would be $O(\min_{0<\epsilon\leqslant 1}\{(\log^3 n + \lambda \log^2 n)/\epsilon^2 + k_\epsilon\})$. Similar extensions were given for the case of point data by Dickerson *et al.* [Dic00] and by Arya and Mount [Ary00], who achieved query times of $O((\log^3 n)/\epsilon^3 + k_\epsilon)$ and $O(\log n + (\frac{1}{\epsilon})^3 + k_\epsilon)$, respectively. Note that the dependency on ϵ in our bounds is better by a factor of $O(\frac{1}{\epsilon})$; only for convex ranges they were able to prove the dependency we get for general ranges. Our proof technique also applies to their structures, which implies an improvement of their query time by a factor of $O(\frac{1}{\epsilon})$ for non-convex ranges.

5.2 The LSF-interval tree

In this section we first describe how to construct a kd-interval tree with longest-side-first splitting, or LSF-interval tree for short, for a set of boxes in 3-space. After that we analyse its performance for approximate range queries.

5.2.1 The construction

Our three-dimensional LSF-interval tree is a generalisation of the two-dimensional kd-interval tree with longest-side-first splitting as described by Agarwal *et al.* [Aga01BGHH]. In fact, the two-dimensional substructures in our three-dimensional structure are basically their two-dimensional structures.

Our construction algorithm takes as input a set of 3-dimensional axis-parallel boxes and their joint bounding box. The algorithm then works top-down, recursively constructing subtrees on subsets of the input. In a generic step of the construction, we have as input a set S of 3-dimensional axis-parallel boxes and a *defining region* R. The construction is started with the full input set as input and the bounding box of the entire scene as defining region. In the recursive steps, the defining regions can be axis-parallel boxes, rectangles, line segments, or points. Each input box $b \in S$ will intersect R; more precisely, the defining regions will always be such that if aff(R) denotes the affine hull of R, then $b \cap \text{aff}(R) \subset R$. If the defining region R is d-dimensional, for some $d \in \{0, 1, 2, 3\}$, then we call the subtree storing S a *d-LSF-interval tree*, and we call its root a *d-node*.

We will now describe an algorithm to construct a d-LSF-interval tree for a set S of input boxes and a defining region R. The algorithm produces a tree whose nodes have degree at most nine; conversion to a binary tree can easily be done and does not affect the asymptotic bounds.

We proceed as follows:

1. We create a root node ν, storing the bounding box $b(\nu)$ of the boxes in S.

2. For each of the six directions $+x$, $-x$, $+y$, $-y$, $+z$, and $-z$ we take the box in S extending farthest in that direction. Each of these at most six boxes is stored in a separate leaf, called a *priority leaf*, immediately below the root node ν. Let S' denote the set of remaining boxes. Assume S' is non-empty; otherwise we are done.

3. If $d = 0$, we recursively build a 0-LSF-interval tree for S' using the point R as defining region, and we make the root of this tree a child of ν. (In fact, for $d = 0$, building a cs-priority-box-tree [Aga01BGHH] could make

a better choice, but in our analysis the better performance of a cs-priority-box-tree would be overshadowed by other terms. In the analysis presented in this paper, we only need the priority leaves, and the division of boxes among the children does not matter.)

Otherwise, if $d > 0$, let e be a longest edge of R, where $e = R$ if R is a line segment. Let h be a plane orthogonal to e. Define h^- to be the halfspace on one side of h, and h^+ to be the halfspace on the other side of h. Define S^- to be the subset of boxes in S' lying completely in h^-, S^+ to be the subset of boxes in S' lying completely in h^+, and $S^\times$ to be the subset of boxes intersecting h. We choose h such that $|S^-| < |S'|/2$ and $|S^+| \leqslant |S'|/2$. We then recursively construct three subtrees whose roots become children of the root node ν:

- The subset S^- is stored in a d-LSF-interval tree with $R \cap h^-$ as defining region.
- The subset S^+ is stored in a d-LSF-interval tree with $R \cap h^+$ as defining region.
- The subset $S^\times$ is stored in a $(d-1)$-LSF-interval tree with $R \cap h$ as defining region.

We could start the construction with the entire input set S and any box R completely containing S as defining region. To achieve good performance, however, we first need to apply one simple but crucial step: we divide S into three 'oriented' subsets S_x, S_y, and S_z, where S_x, S_y and S_z contain all boxes whose longest edges are parallel to the x-axis, y-axis and z-axis, respectively, with ties broken arbitrarily. We then build an LSF-interval tree for each of these three subsets separately, and combine them at the top level. For each of the subsets, we say that the *primary axis* is the axis that corresponds to the orientation of the longest edges of the boxes in the set; the other axes are called *secondary axes*.

5.2.2 Analysis for box-intersection queries

We will analyse the query time in 3-dimensional LSF-interval trees for a box-intersection query in the subtree constructed for S_x. The analysis for S_y and S_z is similar; therefore, the asymptotic bounds we obtain hold for the entire tree as well. Recall that a query with a range Q visits all nodes ν whose bounding box $b(\nu)$ intersects Q. In the analysis, however, we work with a slightly extended range Q_ϵ, and we will charge the visiting of some of the nodes to 'approximate answers', that is, to input boxes intersecting Q_ϵ.

In the analysis we will use the following notation:

Q: the query range;

$w = w(Q)$: the length of the longest edge of the query range;

$\epsilon > 0$: the factor determining the size of the extended query range; to simplify the formulae we assume that $\epsilon \leqslant 1$, although the analysis can easily be adapted to values greater than 1. Our analysis holds for any $0 < \epsilon \leqslant 1$. Since ϵ is only used in the analysis and not by the algorithm, this implies that the actual query time is bounded by the minimum over all ϵ with $0 < \epsilon \leqslant 1$.

Q_ϵ: the *extended query range*, which consists of Q and all points within a distance ϵw from Q in the L_∞-metric;

k_ϵ: the number of input boxes intersecting the extended query range Q_ϵ; by $k_\epsilon(\mathcal{T})$ we will denote the number of input boxes in a subtree $\mathcal{T}$ that intersect Q_ϵ.

We also use a parameter that describes certain properties of the distribution of the input boxes over the space.

$\lambda \geqslant 1$: the *slicing number* of S, defined as follows. Let the slicing number λ_C of S with respect to a cube C be the maximum number of input boxes that intersect four parallel edges of C; then the overall slicing number λ is the maximum value of λ_C over all possible cubes C. Note that a box also intersects an edge if it fully contains that edge. Hence, λ is also an upper bound on the *stabbing number* σ of S, which is defined as the largest number of input boxes with a non-empty common intersection.

At the end of this section, we will show that if the input consists of a set of pipes with small stabbing number, together with a set of arbitrary boxes with low density, the complete input set will have low slicing number.

We will do the analysis bottom-up, first analysing the query time in 1-dimensional subtrees, then in 2-dimensional subtrees, then in 3-dimensional subtrees. We will denote the subtree we are analyzing by $\mathcal{T}$, and its defining region by $R(\mathcal{T})$. The subtree rooted at a node ν is denoted by $\mathcal{T}_\nu$. Sometimes we will speak of the defining region $R(\nu)$ of a node ν, which is simply the defining region $R(\mathcal{T}_\nu)$ of its subtree.

Before we proceed we state a lemma that we will need at various occasions.

Lemma 5.2.1 *Let $\mathcal{T}$ be a d-dimensional LSF-interval-tree and let C be a k-dimensional cube, with $1 \leqslant k \leqslant d \leqslant 3$. Then there are only $O(\log^{k-1} n)$ d-nodes in $\mathcal{T}$ whose defining regions are disjoint and intersect opposite facets of C.*

Proof: The d-nodes in a d-dimensional LSF-interval tree basically form a d-dimensional longest-side-first kd-tree. Hence, the lemma is in fact an easy generalization of Lemma 3.2 from Duncan *et al.* [Dic00] (the hypercube stabbing lemma). For completeness we give a proof, which closely follows the proof of Duncan *et al.*

Let h be the height of $\mathcal{T}$. Suppose X is a set of defining regions of d-nodes in $\mathcal{T}$ that are disjoint and intersect at least one pair of opposite facets of C. We will prove that for any such set $|X| \leqslant 2^k(h+1)^{k-1}$. Since the height of the tree is $O(\log n)$, this means that X must have size $O(\log^{k-1} n)$.

For all d-nodes ν in $\mathcal{T}$, let $l(\nu)$ be the height of the subtree $\mathcal{T}_\nu$ rooted at ν. Let $p(\nu)$ be the number of pairs of opposite facets of C intersected by $R(\nu)$. Let $s(\nu)$ be the number of 'single' facets of C intersected, that is, facets that intersect $R(\nu)$, while their opposites do not intersect $R(\nu)$. For each node ν in $\mathcal{T}$, we define $X(\nu)$ as the set of regions $R(\mu)$ in $\mathcal{T}_\nu$ that are in X. For all $p \geqslant 0$ and $s, l \geqslant -1$, let $x(p, s, l)$ be the maximum size of $X(\nu)$ over all nodes ν in $\mathcal{T}$ with $p(\nu) = p$, $s(\nu) = s$ and $l(\nu) \leqslant l$ (if there are no such nodes, $x(p, s, l) = 0$). First note that $x(p, s, l) = 0$ for $p = 0$, regardless of s and l: if $p(\nu) = 0$, then $R(\nu)$ does not intersect opposite facets of C, and neither does any of its descendants, hence $X(\nu)$ must be empty. Furthermore, if $s = -1$ and/or $l = -1$, we have $x(p, s, l) = 0$ as well (nodes ν with $s(\nu) < 0$ or $l(\nu) < 0$ do not exist). We claim that for $p \geqslant 1$ and $s, l \geqslant 0$ we have:

$$x(p,s,l) \leqslant \max \begin{cases} 1 \\ x(p,s,l-1) \\ 2 \cdot x(p-1,s+1,l-1) \\ x(p,s,l-1) + x(p,s-1,l-1) \end{cases}$$

To see this, examine a region $R(\nu)$ with $p(\nu) = p \geqslant 1$, $s(\nu) = s$ and $l(\nu) = l$. Let c be the length of the sides of C and assume without loss of generality that $R(\nu)$ is cut into two d-dimensional subregions $R(\nu_1)$ and $R(\nu_2)$ by a plane orthogonal to the x-axis. Recall that, as a result of the longest-side-first cutting rule used in the construction of the tree, the x-axis must be the one that is parallel to the longest edges of $R(\nu)$. Since $R(\nu)$ intersects at least one pair of opposite facets of C, its longest side must have length at least c. Therefore, the size of $R(\nu)$ in the x-dimension must be at least c, otherwise this cutting plane could not have been chosen. Therefore, if the plane cuts C, $R(\nu)$ does not fit between the facets of C that are parallel to the cut, so $R(\nu)$ must intersect at least one of these facets. We can now bound $|X(\nu)|$ as follows.

- If ν has children ν_1 and ν_2, but $R(\nu)$ is in X itself, then none of its decendants can be, since the regions in X are all disjoint. Therefore, $|X(\nu_1)| = |X(\nu_2)| = 0$ and $|X(\nu)| = 1$. Also, if $R(\nu)$ is not cut, $|X(\nu)| \leqslant 1$.
- If $R(\nu)$ is cut by a plane that does not intersect C, we get $p(\nu_1) = p$, $s(\nu_1) = s$, $p(\nu_2) = 0$, and $s(\nu_2) = 0$ (or the other way around, exchanging ν_1 and ν_2), and therefore $|X(\nu)| = |X(\nu_1)| + |X(\nu_2)| \leqslant x(p, s, l-1) + x(0, 0, l-1) = x(p, s, l-1)$.
- If $R(\nu)$ is cut by a plane that intersects C, and both facets of C that are parallel to the cutting plane are intersected by $R(\nu)$, then the cut separates these facets and we get $p(\nu_1) = p(\nu_2) = p-1$, $s(\nu_1) = s(\nu_2) = s+1$ and therefore, $|X(\nu)| \leqslant 2 \cdot x(p-1, s+1, l-1)$.
- If $R(\nu)$ is cut by a plane that intersects C, and only one of the facets of C that are parallel to the cutting plane are intersected by $R(\nu)$, we get $p(\nu_1) = p(\nu_2) = p$, $s(\nu_1) = s$, and $s(\nu_2) = s-1$ (or the other way around), and therefore, $|X(\nu)| \leqslant x(p, s, l-1) + x(p, s-1, l-1)$.

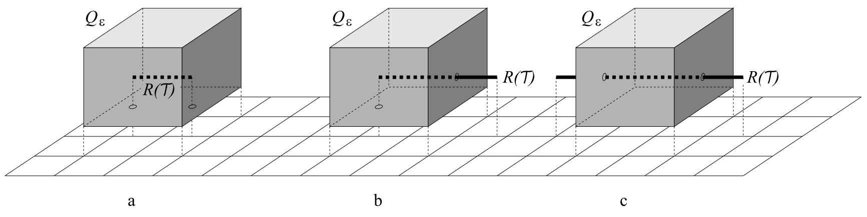


Figure 5.2: Two planes containing the line segment $R(\mathcal{T})$ intersect Q_ϵ.

Since $x(p, s, l) = \max(|X(\nu)|)$, the claim follows.

By induction it is now easy to show that $x(p, s, l) \leqslant 2^p(l + 1)^{p+s-1}$. Notice that the root ν of $\mathcal{T}$ has $p(\nu) + s(\nu) \leqslant k$, and therefore, $|X| = |X(\nu)| \leqslant x(p(\nu), s(\nu), h) \leqslant 2^k(h + 1)^{k-1}$. □

1-dimensional subtrees

In a 1-dimensional subtree $\mathcal{T}$, the defining region $R(\mathcal{T})$ is a line segment that intersects all input boxes stored in $\mathcal{T}$. The worst-case query time in $\mathcal{T}$ depends on the relation of $R(\mathcal{T})$ to the query range. In particular, we distinguish three cases, depending on how many of the two axis-parallel planes containing $R(\mathcal{T})$ intersect Q_ϵ.

Case 1: Two planes containing $R(\mathcal{T})$ intersect Q_ϵ. This case is illustrated in Fig. 5.2. Parts (a) and (b) of the figure correspond to part (i) in the lemma below, part (c) to part (ii).

Lemma 5.2.2 *Let $\mathcal{T}$ be a 1-LSF-interval tree storing n boxes. Suppose we query $\mathcal{T}$ with a box Q such that both axis-parallel planes containing $R(\mathcal{T})$ intersect Q_ϵ.*

(i) If the axis-parallel projection of Q_ϵ onto the line containing $R(\mathcal{T})$ contains at least one endpoint of $R(\mathcal{T})$, we visit $O(k_\epsilon(\mathcal{T}))$ nodes.

(ii) Otherwise, we visit $O(\log n + k_\epsilon(\mathcal{T}))$ nodes.

Proof: Since both axis-parallel planes containing $R(\mathcal{T})$ intersect Q_ϵ, we know that $R(\mathcal{T})$ itself must intersect Q_ϵ. Hence, an (input or bounding) box b stored in $\mathcal{T}$ intersects Q_ϵ if and only if $b \cap R(\mathcal{T})$ intersects $Q_\epsilon \cap R(T)$. We can therefore analyse the query time in this case as if the situation were completely 1-dimensional, that is, as if $\mathcal{T}$ were a 1-tree storing segments on a line, which is queried with a segment on the same line. An analysis of this case, proving the lemma, can be found in the paper by Agarwal *et al.* [Aga01BGHH]. □

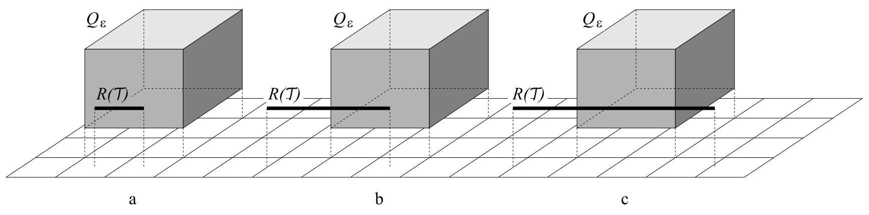


Figure 5.3: One plane containing the line segment $R(\mathcal{T})$ intersects Q_ϵ.

Case 2: One plane containing $R(\mathcal{T})$ intersects Q_ϵ. This case is illustrated in Fig. 5.3. Part (a) of the figure corresponds to part (i) in the lemma below, parts (b) and (c) to part (ii).

Lemma 5.2.3 *Let $\mathcal{T}$ be a 1-LSF-interval tree storing n boxes with stabbing number σ. Suppose we query $\mathcal{T}$ with a box Q such that one axis-parallel plane containing $R(\mathcal{T})$ intersects Q_ϵ.*

(i) If the axis-parallel projection of Q_ϵ onto the line containing $R(\mathcal{T})$ contains $R(\mathcal{T})$ completely, then we visit $O(k_\epsilon(\mathcal{T}))$ nodes.

(ii) Otherwise, we visit $O(\log n + \sigma + k_\epsilon(\mathcal{T}))$ nodes.

Proof: Let g be the axis-parallel plane containing $R(\mathcal{T})$ and intersecting Q_ϵ. For any (input or bounding) box b stored in $\mathcal{T}$, we know that b intersects Q_ϵ if and only if $b \cap g$ intersects $Q_\epsilon \cap g$. We can therefore analyse the query time in this case as if the situation were completely 2-dimensional, that is, as if $\mathcal{T}$ were a 1-tree storing rectangles in the plane, which is queried with a rectangle in the plane. An analysis of this case, proving the lemma, can be found in the paper by Agarwal *et al.* [Aga01BGHH]. □

Case 3: No plane containing $R(\mathcal{T})$ intersects Q_ϵ. In the analysis of this case we will take into account how much of the query range is 'within reach' of the tree. More precisely, consider the intersection of $R(\mathcal{T})$ with the projection of Q_ϵ on the line containing $R(\mathcal{T})$. We denote by $C_Q(\mathcal{T})$ the length of this intersection divided by the length of the longest edge of Q—see Fig. 5.4. In the next subsection we will sum the bound for several different disjoint subtrees $\mathcal{T}$, and then we will use the fact that their $C_Q(\mathcal{T})$-values sum up to at most $1 + 2\epsilon$.

Figure 5.4 illustrates the cases that arise in the next lemma, with part (a) of the figure corresponding to part (i) of the lemma, and parts (b) and (c) corresponding to part (ii).

Lemma 5.2.4 *Let $\mathcal{T}$ be a 1-LSF-interval tree storing n boxes with slicing number λ. Suppose we query $\mathcal{T}$ with a box Q such that no axis-parallel plane containing $R(\mathcal{T})$ intersects Q_ϵ.*

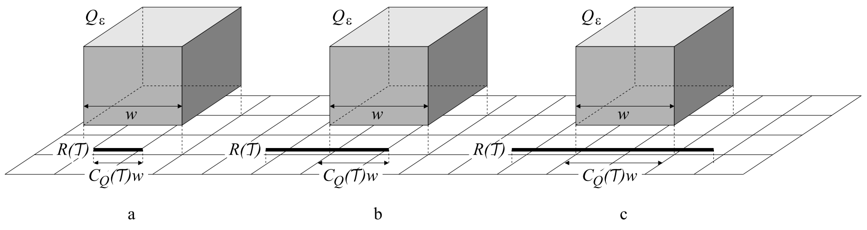


Figure 5.4: No plane containing the line segment $R(\mathcal{T})$ intersects Q_ϵ.

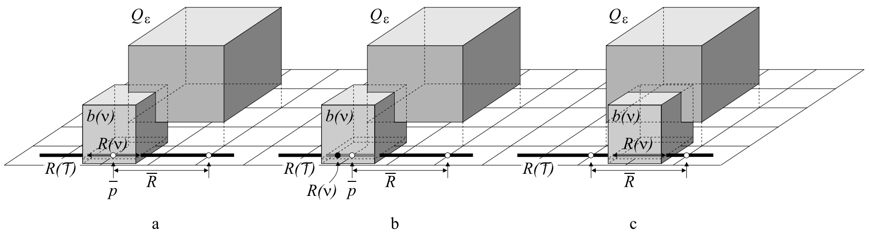


Figure 5.5: a. A node's bounding box $b(\nu)$ such that $b(\nu) \cap R(\mathcal{T}) \not\subset \overline{R}$. $R(\nu)$ is a line segment (black) that contains $\overline{p}$. — b. A node's bounding box $b(\nu)$ such that $b(\nu) \cap R(\mathcal{T}) \not\subset \overline{R}$. $R(\nu)$ is a point (black dot). — c. A node's bounding box $b(\nu)$ such that $b(\nu) \cap R(\mathcal{T}) \subset \overline{R}$.

(i) If the axis-parallel projection of Q_ϵ onto the line containing $R(\mathcal{T})$ contains $R(\mathcal{T})$ completely, then we visit $C_Q(\mathcal{T}) \cdot O(\frac{\lambda}{\epsilon}) + O(\lambda)$ nodes.

(ii) Otherwise, we visit $O(\log n + \frac{\lambda}{\epsilon})$ nodes.

Proof: Since the maximum degree of each node is nine, the number of visited leaf nodes is at most nine times the number of visited internal nodes. Hence, we can restrict our attention to bounding the latter number. Let $\overline{Q_\epsilon}$ denote the axis-parallel projection of Q_ϵ onto the line containing $R(\mathcal{T})$, and let $\overline{R} := \overline{Q_\epsilon} \cap R(\mathcal{T})$, i.e., in Fig. 5.4, $\overline{R}$ is the part of $R(\mathcal{T})$ indicated by the stick measuring $C_Q(\mathcal{T})w$. Let ν be a visited internal node of $\mathcal{T}$, and let $b(\nu)$ be its bounding box. We distinguish two cases: $b(\nu) \cap R(\mathcal{T}) \subset \overline{R}$ (Fig. 5.5c), and $b(\nu) \cap R(\mathcal{T}) \not\subset \overline{R}$ (Fig. 5.5a and 5.5b). We claim that the number of nodes to which the first case applies is $C_Q(\mathcal{T}) \cdot O(\frac{\lambda}{\epsilon}) + O(\lambda)$, and that the number of nodes to which the second case applies is $O(\sigma + \log n)$, where σ is the stabbing number of the boxes stored in the tree. Note that in part (i) of the lemma the second case cannot arise. Together with the fact that $\lambda \geqslant \sigma$ and $C_Q(\mathcal{T}) \leqslant 1 + 2\epsilon$, this means that proving the claim above will establish the lemma.

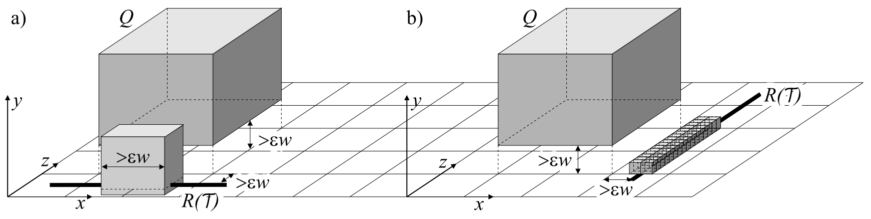


Figure 5.6: a. A shield on a defining region parallel to the primary axis. — b. Arrangement of cubes intersected by shields on a defining region parallel to a secondary axis.

We first bound the number of nodes for which $b(\nu) \cap R(\mathcal{T}) \not\subset \overline{R}$, since this is the easier case. Let ν be such a node. Since $b(\nu) \cap R(\mathcal{T})$ cannot be disjoint from $\overline{R}$—otherwise $b(\nu)$ would not intersect Q_ϵ (not to mention Q) and ν would not be visited—it follows that $b(\nu)$ must contain an endpoint $\overline{p}$ of $\overline{R}$. Now there are two possibilities.

One is that $R(\nu)$, the defining region of ν, is a line segment containing $\overline{p}$ (see Fig. 5.5a). Since the defining regions of 1-nodes at a fixed level of the tree are disjoint and the depth of the tree is $O(\log n)$, there are only $O(\log n)$ such nodes.

The other possibility is that $R(\nu)$ is a point—see Fig. 5.5b—then all boxes stored in $\mathcal{T}(\nu)$ must contain the point $R(\nu)$. But then the priority leaf immediately below ν storing the box extending farthest into the direction of $\overline{p}$ must contain $\overline{p}$. We charge the visit of ν to this leaf. Since a leaf gets charged only from its parent, and there are at most σ input boxes containing any given point, there are at most 2σ such nodes.

Thus we find a bound of $O(\log n + \sigma) = O(\log n + \lambda)$ for the case of $b(\nu) \cap R(\mathcal{T}) \not\subset \overline{R}$.

Now consider the nodes ν such that $b(\nu) \cap R(\mathcal{T}) \subset \overline{R}$. We shall charge the visit of ν to a certain priority leaf directly below it, called a *shield*. Each shield will be charged at most once, namely from its parent. Bounding the maximum number of shields will then prove this part of the claim.

We start by defining the shields. Recall that the primary axis of S_x—the axis parallel to the longest edges of the boxes in S_x—is the x-axis. Since the two remaining (secondary) axes play equivalent roles, we can assume that the y-axis is not parallel to $R(\mathcal{T})$. Let us also assume w.l.o.g. that the y-coordinate of $R(\mathcal{T})$ is smaller than the smallest y-coordinate of Q (i.e. $R(\mathcal{T})$ lies diagonally under Q, like in Fig. 5.6). A *shield* is now defined as a priority leaf whose corresponding input box b extends into the positive y-direction from $R(\mathcal{T})$ over a distance of at least ϵw. That is, if $y_{\max}(b)$ is the maximum y-coordinate of b and $y(R(\mathcal{T}))$ is the y-coordinate of $R(\mathcal{T})$, then b is a shield if $y_{\max}(b) - y(R(\mathcal{T})) \geqslant \epsilon w$.

We now argue that each visited internal node ν for which it holds that $b(\nu) \cap R(\mathcal{T}) \subset \overline{R}$, has at least one shield as a child. Indeed, since none of the two axis-parallel planes containing $R(\mathcal{T})$ intersects Q_ϵ, the y-distance of $R(\mathcal{T})$ and Q must be at least ϵw. This means that the bounding box of ν must extend over a distance at least ϵw into the y-direction from $R(\mathcal{T})$, otherwise ν would not be visited. Hence, the input box extending farthest into the y-direction, extends that far; the priority leaf directly below ν storing this box is a shield.

It remains to bound the number of shields. We consider two subcases.

The first subcase is that $R(\mathcal{T})$ is parallel to the x-axis, as in Fig. 5.6a. In this case the length of any box in S_x along $R(\mathcal{T})$ is at least its length in any other direction. In particular, a shield will cover a portion of $\overline{R}$ of length at least ϵw. Since no point is contained in more than σ input boxes, there can be at most $\sigma \cdot \text{length}(\overline{R})/(\epsilon w)$ shields in this case. Because $\text{length}(\overline{R}) = C_Q(\mathcal{T}) \cdot w$ by definition, the number of shields is bounded by $\frac{\sigma}{\epsilon} \cdot C_Q(\mathcal{T})$.

The second subcase is that $R(\mathcal{T})$ is parallel to the z-axis—see Fig. 5.6b. In this case, a shield must extend over a distance of at least ϵw upwards from $R(\mathcal{T})$ and over a distance of at least $\epsilon w/2$ into either the positive or negative x-direction from $R(\mathcal{T})$. Now imagine a line-up of $\lceil \frac{2}{\epsilon} C_Q(\mathcal{T}) \rceil$ cubes of size $\frac{\epsilon}{2} w$ whose lower right edges together cover Q's projection on $R(\mathcal{T})$. Add a copy of this line-up shifted right over a distance of $\epsilon w/2$, so that in the second line-up, the lower *left* edges together cover Q's projection—see Fig. 5.6b. Since a shield extends away from $R(\mathcal{T})$ in both orthogonal directions over a distance greater than the size of the cubes in the line-up, it must intersect the four edges parallel to $R(\mathcal{T})$ of at least one of these cubes. Since the slicing number of the input boxes is at most λ, there can be at most $2\lambda \lceil \frac{2}{\epsilon} C_Q(\mathcal{T}) \rceil \leqslant 2\lambda + 4C_Q(\mathcal{T})\frac{\lambda}{\epsilon}$ shields in this case.

Using $\lambda \geqslant \sigma$, we conclude that the bounds for both subcases are within $O(\lambda) + C_Q(\mathcal{T}) \cdot O(\frac{\lambda}{\epsilon})$, which finishes the proof of our claim. □

2-dimensional subtrees

Let $\mathcal{T}$ be a 2-dimensional subtree. As before, it will be useful to take into account how much of the query range's boundary is 'within reach' of the tree. More precisely, consider the edges of Q_ϵ's projection on the plane containing $R(\mathcal{T})$. Denote by $C_Q(\mathcal{T})$ the sum of the lengths of the intersections of these edges with $R(\mathcal{T})$, divided by w, the length of the longest edge of the query range.

We distinguish two cases, depending on whether or not the plane containing the 2-dimensional defining region $R(\mathcal{T})$ intersects Q_ϵ.

Case 1: The plane containing $R(\mathcal{T})$ intersects Q_ϵ. This case is illustrated in Fig. 5.7. Parts (a) and (b) of the figure correspond to case (i) in the lemma below, part (c) to case (ii), and part (d) to case (iii).

Lemma 5.2.5 *Let $\mathcal{T}$ be a 2-LSF-interval tree storing n boxes with stabbing number σ. Suppose we query $\mathcal{T}$ with a box Q such that the plane containing $R(\mathcal{T})$ intersects the extended query range Q_ϵ. Let $\overline{Q}_\epsilon$ denote the intersection of Q_ϵ with the plane containing $R(\mathcal{T})$.*

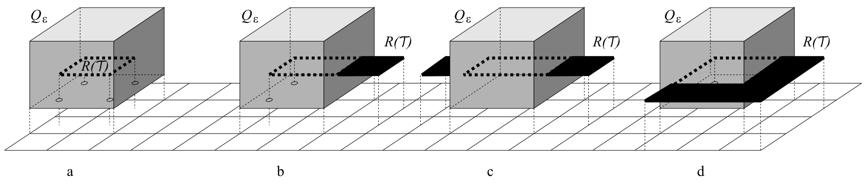


Figure 5.7: The plane containing the rectangle $R(\mathcal{T})$ intersects Q_ϵ.

(i) If at most one edge of $\overline{Q}_\epsilon$ *intersects* $R(\mathcal{T})$*, then we visit* $O(k_\epsilon(\mathcal{T}))$ *nodes.*

(ii) If two opposite edges, and no other edges, of $\overline{Q}_\epsilon$ *intersect* $R(\mathcal{T})$*, then we visit* $O(\log^2 n + k_\epsilon(\mathcal{T})) + C_Q(\mathcal{T}) \cdot O(\frac{1}{\epsilon}\log^2 n)$ *nodes.*

(iii) Otherwise we visit $O(\frac{1}{\epsilon}\log^2 n + \sigma \log n + k_\epsilon(\mathcal{T}))$ *nodes.*

Proof: First we observe that the longest edge of Q_ϵ has length $(1+2\epsilon)w$ and that its shortest edge has length at least $2\epsilon w$. Hence, the aspect ratio of Q_ϵ and the aspect ratio of $\overline{Q}_\epsilon$ are at most $1 + 1/(2\epsilon)$.

Since $R(\mathcal{T})$ intersects Q_ϵ, we know for any (input or bounding) box b stored in $\mathcal{T}$ that b intersects Q_ϵ if and only if $b \cap R(\mathcal{T})$ intersects $Q_\epsilon \cap R(T)$. We can therefore analyse the query time in this case as if the situation were completely 2-dimensional, that is, as if $\mathcal{T}$ were a 2-tree storing rectangles in the plane, which is queried with $\overline{Q}_\epsilon$. Since $\overline{Q}_\epsilon$ has aspect ratio at most $1 + 1/(2\epsilon)$, parts (i) and (iii) of the lemma now immediately follow from the results by Agarwal *et al.* [Aga01BGHH].

For part (ii), we need a bit more refined analysis. Consider the collection N of all visited 2-nodes ν in $\mathcal{T}$ whose defining region $R(\nu)$ intersects two opposite edges of $\overline{Q}_\epsilon$, and no other edges. This collection forms a subgraph $\mathcal{G}(N)$ of $\mathcal{T}$, which is a tree rooted at the root of $\mathcal{T}$. We shall first bound the number of nodes in N, and then the number of visited descendants.

To bound the number of nodes in N, we cover $\overline{Q}_\epsilon$ with at most $\lceil\alpha\rceil$ squares with side length $(1+2\epsilon)w/\alpha$, where $\alpha \leqslant 1 + 1/(2\epsilon)$ is the aspect ratio of $\overline{Q}_\epsilon$ (see Fig 5.8a). From a bound on the number of nodes intersecting these squares, we can derive a bound on the number of nodes in N as follows. At most $\alpha C_Q(\mathcal{T}) + 1$ of the squares intersect $R(\mathcal{T})$. Now consider a node $\nu \in N$. Since $R(\nu)$ intersects two opposite sides of $\overline{Q}_\epsilon$, it intersects two opposite sides of at least one of the $\alpha C_Q(\mathcal{T}) + 1$ squares used to cover $\overline{Q}_\epsilon \cap R(\mathcal{T})$. Observe that the leaves of $\mathcal{G}(N)$—that is, the nodes that have no children in N; they need not be leaves of $\mathcal{T}$—have disjoint defining regions. Lemma 5.2.1 implies that the number of such leaves is $O(\log n) + C_Q(\mathcal{T}) \cdot O(\alpha \log n)$. If we include their ancestors in the count, we obtain a bound of $O(\log^2 n) + C_Q(\mathcal{T}) \cdot O(\alpha \log^2 n)$ on the number of nodes in N.

It remains to bound the number of descendants of the nodes in N. These are organized into subtrees whose roots are children of nodes in N and are not in N

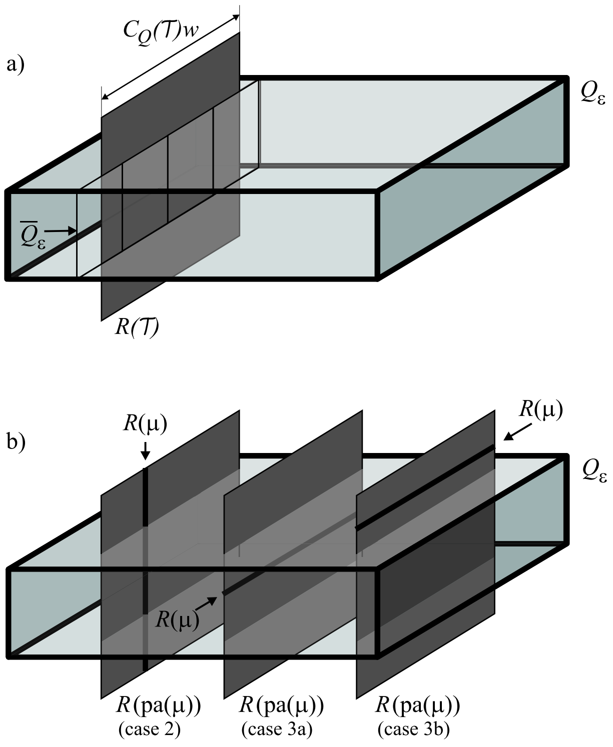

a)
$C_Q(\mathcal{T})w$
Q_ε
$\overline{Q}_\varepsilon$
$R(\mathcal{T})$
b)
$R(\mu)$
$R(\mu)$
Q_ε
$R(\mu)$
$R(\text{pa}(\mu))$ (case 2)
$R(\text{pa}(\mu))$ (case 3a)
$R(\text{pa}(\mu))$ (case 3b)

Figure 5.8: (For ease of visualization, we changed the orientation of $R(\mathcal{T})$ as compared to Fig. 5.7.) a. Covering $\overline{Q}_\epsilon$ with $\lceil\alpha\rceil$ squares. — b. μ is a 1-node whose defining region cuts a 2-node intersecting opposite edges of $\overline{Q}_\epsilon$, that is: opposite facets of Q_ϵ.

themselves. Consider such a root node μ. Let $\text{pa}(\mu) \in N$ be the parent of μ. There are three cases.

- The first case is that μ is a 2-node. In this case $R(\mu)$ intersects at most one edge of $\overline{Q}_\epsilon$, as in part (i) of the lemma; if it would intersect two opposite edges it would be in N, and the case where a vertex of $\overline{Q}_\epsilon$ lies in $R(\mu)$ cannot occur when we are handling part (ii) of the lemma. The total number of visited nodes of $\mathcal{T}_\mu$ is $O(k_\epsilon(\mathcal{T}_\mu))$ by part (i) of the lemma. Summing over all nodes μ thus gives us a total bound of $O(k_\epsilon(\mathcal{T}))$ for these subtrees.

- The second case is that the root is a 1-node μ and $R(\mu)$ cuts $R(\text{pa}(\mu))$ such that $\text{pa}(\mu)$ has two children in N—see Fig. 5.8b case 2.

  The number of nodes of degree two in $\mathcal{G}(N)$ is no more than the number of leaves in $\mathcal{G}(N)$, so there can be at most $O(\log n) + C_Q(\mathcal{T}) \cdot O(\alpha \log n)$

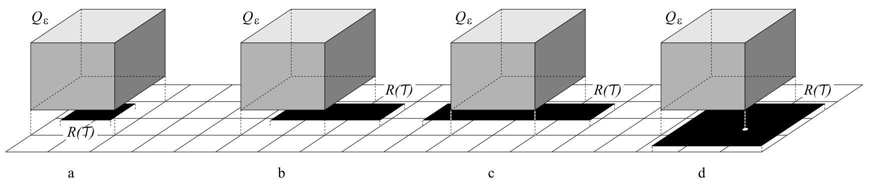


Figure 5.9: The plane containing the rectangle $R(\mathcal{T})$ is disjoint from Q_ϵ.

such nodes μ. Lemma 5.2.2(ii) states that the query time in each such tree is $O(\log n + k_\epsilon(\mathcal{T}_\mu))$, so the total query time in these trees is $O(\log^2 n + k_\epsilon(\mathcal{T})) + C_Q(\mathcal{T}) \cdot O(\alpha \log^2 n)$.

- The third case is that the root is a 1-node μ, where $R(\mu)$ cuts $R(\mathrm{pa}(\mu))$ such that $\mathrm{pa}(\mu)$ has at most one child in N—see Fig. 5.8b case 3a and case 3b.

  Now $R(\mu)$ must lie completely inside the projection of Q_ϵ onto the line containing $R(\mu)$. Lemma 5.2.2(i) (for case 3a) and Lemma 5.2.3(i) (for case 3b) state that the query time in each tree rooted at such a node is $O(k_\epsilon(\mathcal{T}_\mu))$. Since the number of such nodes is asymptotically bounded by the size of N, the total query time in these 1-trees is $O(\log^2 n + k_\epsilon(\mathcal{T})) + C_Q(\mathcal{T}) \cdot O(\alpha \log^2 n)$.

In total, we find a bound of $O(\log^2 n + k_\epsilon(\mathcal{T})) + C_Q(\mathcal{T}) \cdot O(\alpha \log^2 n)$. With $\alpha \leqslant 1 + 1/(2\epsilon)$, this proves part (ii) of the lemma. □

Case 2: The plane containing $R(\mathcal{T})$ does not intersect Q_ϵ. This case is illustrated in Fig. 5.9. Part (a) of the figure corresponds to case (i) in the lemma below, parts (b) and (c) to case (ii), and part (d) to case (iii).

Lemma 5.2.6 *Let $\mathcal{T}$ be a 2-LSF-interval tree storing n boxes with slicing number λ. Suppose we query $\mathcal{T}$ with a box Q such that the plane containing $R(\mathcal{T})$ does not intersect Q_ϵ. Let $\overline{Q}_\epsilon$ denote the axis-parallel projection of Q_ϵ onto the plane containing $R(\mathcal{T})$.*

(i) If $\overline{Q}_\epsilon$ contains $R(\mathcal{T})$ completely, then we visit $O(k_\epsilon(\mathcal{T}))$ nodes.

(ii) If $R(\mathcal{T})$ intersects at least one edge but no vertex of $\overline{Q}_\epsilon$, then we visit $O(\lambda \log^2 n + k_\epsilon(\mathcal{T})) + C_Q(\mathcal{T}) \cdot O(\frac{\lambda}{\epsilon} \log^2 n)$ nodes.

(iii) Otherwise we visit $O(\frac{\lambda}{\epsilon} \log^2 n + k_\epsilon(\mathcal{T}))$ nodes.

Proof: *(i)* Without loss of generality, suppose $R(\mathcal{T})$ is horizontal and lies below Q. Then for every node ν visited in $\mathcal{T}$, the subtree rooted at ν must contain an input box which raises high enough to intersect Q. In particular, there is a priority leaf immediately below ν that stores an input box intersecting Q. We can charge

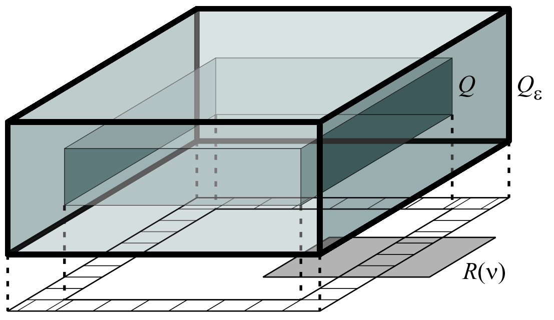


Figure 5.10: Covering $\overline{Q}_\epsilon \setminus \overline{Q}$ with squares.

the visit to ν to that priority leaf. Since there are at most $k_\epsilon(\mathcal{T})$ such priority leaves and each of them is charged at most once, the bound follows.

(ii) We can distinguish two types of visited nodes.

The first type of nodes are 2-nodes whose defining regions lie completely inside $\overline{Q}_\epsilon$ and descendants of such nodes. Here a similar argument as in the proof of part (i) applies: any such node has a priority leaf below it that intersects Q_ϵ, so there are only $O(k_\epsilon(\mathcal{T}))$ such nodes.

The second type of nodes are the remaining ones. Let N be the collection of all remaining visited 2-nodes. For any node $\nu \in N$, we know that $R(\nu)$ intersects the complement of $\overline{Q}_\epsilon$ as well as $\overline{Q}$, the projection of Q onto the plane containing $R(\mathcal{T})$.

To bound the number of nodes in N we cover $\overline{Q}_\epsilon \setminus \overline{Q}$ using at most $4(\lceil \frac{1}{\epsilon} \rceil + 1)$ squares with side length ϵw, which are contained in $\overline{Q}_\epsilon \setminus \overline{Q}$—see Fig. 5.10. For any node $\nu \in N$ we have that $R(\nu)$ intersects two opposite edges of at least one of these squares. Since $R(\nu) \subset R(\mathcal{T})$ and $R(\mathcal{T})$ does not contain a vertex of $\overline{Q}_\epsilon$, we can restrict our attention to squares that are used to cover two opposite 'sides' of $\overline{Q}_\epsilon \setminus \overline{Q}$ and that intersect $R(\mathcal{T})$. Hence, the number of squares we have to consider is at most $2\lceil C_Q(\mathcal{T})/\epsilon \rceil$. As before, we observe that the nodes of N form a subgraph $\mathcal{G}(N)$ of $\mathcal{T}$, which is a tree whose leaves have disjoint defining regions. Hence, by Lemma 5.2.1 there are $O(\log n) + C_Q(\mathcal{T}) \cdot O(\frac{1}{\epsilon} \log n)$ leaves in $\mathcal{G}(N)$. If we include their ancestors in the count, we find a bound of $O(\log^2 n) + C_Q(\mathcal{T}) \cdot O(\frac{1}{\epsilon} \log^2 n)$ on the number of nodes in N.

It remains to bound the number of descendants of nodes in N. The descendants are organized into subtrees whose roots are children of nodes in N and are not in N themselves. Consider such a root node ν. Let $\mathrm{pa}(\mu) \in N$ be the parent of μ. There are three cases.

- The first case is that μ is a 2-node. But then μ must be of the first type—its defining region must lie completely inside $\overline{Q}_\epsilon$—so we already counted these nodes and their descendants earlier.

- The second case is that μ is a 1-node and $R(\mu)$ cuts $R(\mathrm{pa}(\mu))$ in such a way that $\mathrm{pa}(\mu)$ has two children in N.

  The analysis for this case is done in the same way as in the proof of Lemma 5.2.5(ii), now referring to Lemma 5.2.3 instead of Lemma 5.2.2.

  Since the number of nodes of degree two in $\mathcal{G}(N)$ is at most its number of leaves, there can be at most $O(\log n) + C_Q(\mathcal{T}) \cdot O(\frac{1}{\epsilon} \log n)$ such nodes μ. Lemma 5.2.3(ii) states that the query time in each such tree is $O(\log n + \sigma + k_\epsilon(\mathcal{T}_\mu))$, so the total query time in these trees is

  $$O(\log^2 n + \sigma \log n + k_\epsilon(\mathcal{T})) + C_Q(\mathcal{T}) \cdot O\left(\frac{1}{\epsilon} \log^2 n + \frac{\sigma}{\epsilon} \log n\right).$$

  (Note that the k_ϵ terms always add up to $O(k_\epsilon(\mathcal{T}))$.)

- The remaining case is that μ is a 1-node and $\mathrm{pa}(\mu)$ is cut by $R(\mu)$ such that it has at most one child in N.

  Now $R(\mu)$ lies completely inside the projection of Q_ϵ onto the line containing $R(\mu)$. Lemma 5.2.4(i) and Lemma 5.2.3(i) state that the query time in such trees is $O(\lambda) + C_Q(\mathcal{T}_\mu) \cdot O(\frac{\lambda}{\epsilon})$ and $O(k_\epsilon(\mathcal{T}_\mu))$, respectively. The number of nodes to which this applies is clearly bounded by the number of nodes in N, which is $O(\log^2 n) + C_Q(\mathcal{T}) \cdot O(\frac{1}{\epsilon} \log^2 n)$. Hence, the total query time in these 1-trees is

  $$O(\lambda \log^2 n + k_\epsilon(\mathcal{T})) + C_Q(\mathcal{T}) \cdot O\left(\frac{\lambda}{\epsilon} \log^2 n\right) + \sum C_Q(\mathcal{T}_\mu) \cdot O\left(\frac{\lambda}{\epsilon}\right),$$

  where the sum is over all 1-nodes μ that are a child of a node in N and are such that $R(\mu)$ lies completely inside the projection of Q_ϵ onto the line containing $R(\mu)$. Note that each point of an edge of $\overline{Q}_\epsilon$ lies in $O(\log n)$ defining regions of 2-nodes (one node on each level), so $\sum_{\nu \in N} C_Q(\mathcal{T}_\nu) = O(\log n) C_Q(\mathcal{T})$. The same bound holds if we sum over the 1-nodes μ that are children of nodes in N. Hence, we find a total query time for this case of $O(\lambda \log^2 n + k_\epsilon(\mathcal{T})) + C_Q(\mathcal{T}) \cdot O(\frac{\lambda}{\epsilon} \log^2 n)$.

Putting the three cases together, and using $\sigma \leqslant \lambda$, we find an overall bound of $O(\lambda \log^2 n + k_\epsilon(\mathcal{T})) + C_Q(\mathcal{T}) \cdot O(\frac{\lambda}{\epsilon} \log^2 n)$.

(iii) We can distinguish three types of visited nodes: the two types that were also considered in the proof of part (ii), and a third type, namely 2-nodes containing a corner of $\overline{Q}$ and their descendant 1-nodes and 0-nodes.

The number of nodes of the first two types can be bounded as in the proof of part (ii). Using that $C_Q(\mathcal{T}) \leqslant 4(1 + 2\epsilon)$, we get a bound of $O(\frac{\lambda}{\epsilon} \log^2 n + k_\epsilon(\mathcal{T}))$ for these types. As for the third type, we note that there are $O(\log n)$ 2-nodes containing a corner of $\overline{Q}$. If μ is a 1-node that is a child of such a node, then the query time in $\mathcal{T}_\mu$ is $O(\log n + \sigma + k_\epsilon(\mathcal{T}))$ or $O(\log n + \frac{\lambda}{\epsilon})$ by Lemma 5.2.3 or Lemma 5.2.4, respectively, so we have $O(\log^2 n + \frac{\lambda}{\epsilon} \log n + k_\epsilon(\mathcal{T}))$ nodes of the third type. $\square$

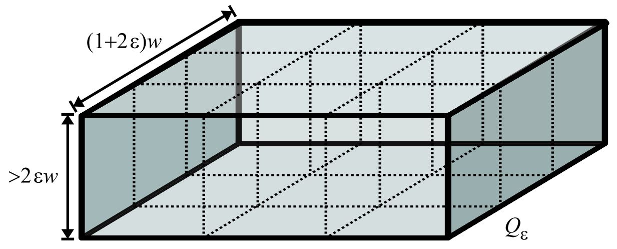


Figure 5.11: Covering Q_ϵ with $O((\frac{1}{\epsilon})^2)$ cubes.

3-dimensional trees

Finally we can prove our main result.

Theorem 5.2.7 *Let $\mathcal{T}$ be a 3-LSF-interval tree storing n boxes with slicing number λ. Then a query in $\mathcal{T}$ with a box Q will visit $O(\min_{0<\epsilon\leqslant 1}\{\frac{1}{\epsilon}(\frac{1}{\epsilon}+\lambda)\log^4 n + k_\epsilon\})$ nodes, where k_ϵ is the number of boxes intersecting the extended range Q_ϵ.*

Proof: Fix an arbitrary $0 < \epsilon \leqslant 1$. As observed before, it suffices to bound the number of visited internal nodes. These can be partitioned into four categories, namely 3-nodes ν such that $R(\nu)$ intersects:

(i) at most one facet of Q_ϵ,

(ii) more than one facet of Q_ϵ, but none of its edges,

(iii) at least one edge of Q_ϵ, but none of its vertices,

(iv) at least one vertex of Q_ϵ,

where each category also includes the descendant 2-nodes, 1-nodes and 0-nodes of the 3-nodes. We will now treat these cases one by one.

(i) 3-Nodes ν such that $R(\nu)$ intersects at most one facet of Q_ϵ, plus their descendant 2-nodes, 1-nodes, and 0-nodes. Any such node must have a priority leaf directly below it that stores a box intersecting Q_ϵ. Hence, the total number of nodes in this category is $O(k_\epsilon)$.

(ii) 3-Nodes ν such that $R(\nu)$ intersects more than one facet of Q_ϵ but none of its edges, plus their descendant 2-nodes, 1-nodes, and 0-nodes.

Let N be the collection of 3-nodes in this category, and let $\mathcal{G}(N)$ be the subgraph of $\mathcal{T}$ formed by these nodes. $\mathcal{G}(N)$ is a forest of trees.

To bound the number of nodes in N, we cover Q_ϵ by $O((\frac{1}{\epsilon})^2)$ cubes that are contained in Q_ϵ and are as big as the smallest edges of Q_ϵ — see Fig. 5.11. Any node in N must intersect opposite facets of at least one of these cubes. Because the leaves of $\mathcal{G}(N)$ have disjoint defining regions, their number is bounded by $O((\frac{1}{\epsilon})^2 \log^2 n)$ by Lemma 5.2.1. The total number of nodes in N is therefore bounded by $O((\frac{1}{\epsilon})^2 \log^3 n)$.

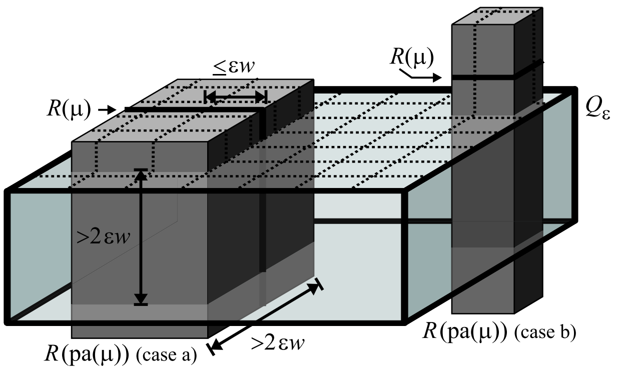


Figure 5.12: 3d-nodes that intersect more than one facet of Q_ϵ, but none of its edges.

It remains to bound the number of descendant 2-nodes, 1-nodes, and 0-nodes of the nodes in N. These are organized in subtrees whose roots are children of nodes in N. Let μ be such a root and let $\mathrm{pa}(\mu) \in N$ be its parent. There are two cases, as illustrated in Fig. 5.12.

- $R(\mu)$ cuts $R(\mathrm{pa}(\mu))$ in such a way that $\mathrm{pa}(\mu)$ has two children in N—see case (a) in Fig. 5.12.

  Since the number of nodes of degree two in $\mathcal{G}(N)$ is bounded by the number of leaves in N, there are only $O((\frac{1}{\epsilon})^2 \log^2 n)$ such roots. Lemma 5.2.5(ii) states that the query time in each subtree rooted at such a node is $O(\log^2 n + k_\epsilon(\mathcal{T}_\mu)) + C_Q(\mathcal{T}_\mu) \cdot O(\frac{1}{\epsilon} \log^2 n)$, so the total query time in these subtrees is

  $$O\left(\frac{1}{\epsilon^2}\log^4 n + k_\epsilon\right) + \sum_\mu C_Q(\mathcal{T}_\mu) \cdot O\left(\frac{1}{\epsilon}\log^2 n\right),$$

  where the sum is over all 2-nodes μ in the current category such that $R(\mathcal{T}_\mu)$ cuts opposite facets of Q_ϵ.

  We proceed to bound $\sum_\mu C_Q(\mathcal{T}_\mu)$. To simplify the discussion, let's assume that the defining regions $R(\mu)$ and $R(\mathrm{pa}(\mu))$ cut the top and bottom facet of Q_ϵ, as in Fig. 5.12, case a. Then for each node μ we have that $C_Q(\mathcal{T}_\mu)w$ is the length of $R(\mu)$ as seen from above. Note that $R(\mathrm{pa}(\mu))$ has height at least $2\epsilon w$, because the height of Q_ϵ is at least that much. Therefore, the length of the horizontal edges of $R(\mathrm{pa}(\mu))$ orthogonal to $R(\mu)$ is at least $2\epsilon w$ as well, otherwise $R(\mathrm{pa}(\mu))$ would have been cut by a horizontal plane. Cover the top facet of Q_ϵ by $O((\frac{1}{\epsilon})^2)$ squares of side length ϵw. Since $R(\mathrm{pa}(\mu))$ has horizontal edges of length at least $2\epsilon w$, it must intersect opposite sides of at least one such square s. If this happens for m 2-nodes μ,

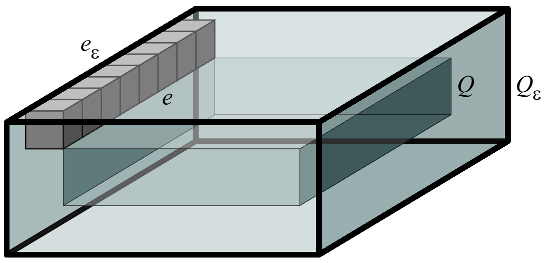


Figure 5.13: Covering an edge of Q with $O(\frac{1}{\epsilon})$ cubes.

then there are at least m disjoint defining regions of 3-nodes that intersect opposite sides of s. Lemma 5.2.1 tells us that s is cut by $O(\log n)$ disjoint defining regions. Hence, the total length within s of all regions $R(\mu)$ as seen from above is $O(\epsilon w \log n)$. Summed over all squares we find that the total length of all regions $R(\mu)$ as seen from above is $O(\frac{w}{\epsilon} \log n)$. This implies that $\sum_\mu C_Q(\mathcal{T}_\mu) = O(\frac{1}{\epsilon} \log n)$. It follows that the total number of nodes for this case is $O((\frac{1}{\epsilon})^2 \log^4 n + k_\epsilon(\mathcal{T}))$.

- $R(\mathcal{T})$ cuts $R(\mathrm{pa}(\mu))$ such that $\mathrm{pa}(\mu)$ has at most one child in N—see case (b) in Fig. 5.12.

  In this case $R(\mu)$ lies completely inside the projection of Q_ϵ onto the plane containing $R(\mu)$. Lemma's 5.2.6(i) and 5.2.5(i) state that the number of visited nodes in each such tree is $O(k_\epsilon(\mathcal{T}_\mu))$, which adds up to $O(k_\epsilon(\mathcal{T}))$.

In total, there are $O((\frac{1}{\epsilon})^2 \log^4 n + k_\epsilon)$ nodes in this category.

(iii) 3-Nodes ν such that $R(\nu)$ intersects at least one edge of Q_ϵ but does not contain one of its vertices, plus their descendant 2-nodes, 1-nodes, and 0-nodes.

In this case $R(\nu)$ must intersect an edge e_ϵ of Q_ϵ *and* the corresponding edge e of Q (the edge with both endpoints lying at an L_∞-distance of ϵw from e_ϵ), otherwise ν would not be visited. For each pair e, e_ϵ of corresponding edges, we take a set of $O(\frac{1}{\epsilon})$ cubes of size ϵw, such that each cube has an edge contained in e and the opposite edge contained in e_ϵ, and such that together they cover e completely — see Fig. 5.13. Let N be the collection of 3-nodes in the current category, and let $\mathcal{G}(N)$ be the subgraph of $\mathcal{T}$ formed by these nodes. $\mathcal{G}(N)$ is a forest of trees.

Any 3-node in N must intersect opposite edges of a facet of at least one of these cubes. Summing over the facets of all cubes and using Lemma 5.2.1 again, we find that there are only $O(\frac{1}{\epsilon} \log n)$ leaves in $\mathcal{G}(N)$ and, hence, $O(\frac{1}{\epsilon} \log^2 n)$ 3-nodes in N in total.

The descendant 2-nodes, 1-nodes, and 0-nodes are organized in subtrees that are rooted at 2-nodes μ with a node $\mathrm{pa}(\mu)$ in N as parent. We distinguish two cases, as illustrated in Fig. 5.14.

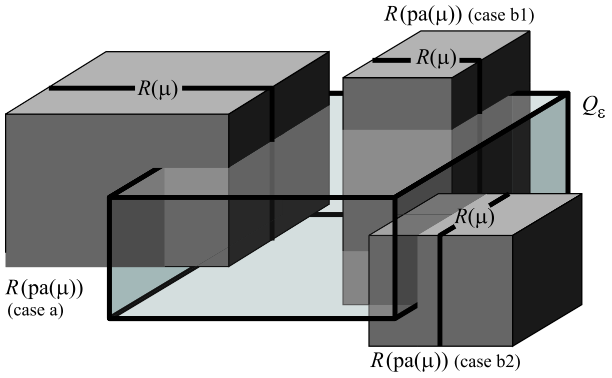


Figure 5.14: 3d-nodes that intersect an edge of Q_ϵ, but none of its vertices.

- For the subtrees rooted at node μ such that $\mathrm{pa}(\mu)$ has two children in N (case (a) in Fig. 5.14), we can apply Lemma 5.2.5(iii) and find a bound of $O(\frac{1}{\epsilon}\log^2 n + \sigma \log n + k_\epsilon(\mathcal{T}_\mu))$ for each subtree. Since the number of such nodes is bounded by the number of leaves in $\mathcal{G}(N)$, we get a total of $O((\frac{1}{\epsilon})^2 \log^3 n + \frac{\sigma}{\epsilon}\log^2 n + k_\epsilon)$ nodes.

- For the other subtrees, of which there are $O(\frac{1}{\epsilon}\log^2 n)$, we apply Lemmas 5.2.5(i) and (ii) (case (b1) in Fig. 5.14) and Lemma 5.2.6(ii) (case (b2)) to find a total bound for all such subtrees of

$$O\left(\frac{\lambda}{\epsilon}\log^4 n + k_\epsilon\right) + \sum C_Q(\mathcal{T}_\mu)\cdot O\left(\frac{\lambda}{\epsilon}\log^2 n\right).$$

  Because any point in 3-space lies in at most $O(\log n)$ defining regions of 3-nodes, we have $\sum C_Q(\mathcal{T}_\mu) = O((1+2\epsilon)\log n)$ and we get a bound of $O(\frac{\lambda}{\epsilon}\log^4 n + k_\epsilon)$.

In total, the number of nodes in this category is $O(\frac{1}{\epsilon}(\frac{1}{\epsilon}+\lambda)\log^4 n + k_\epsilon)$.

(iv) 3-Nodes ν such that $R(\nu)$ contains at least one vertex of Q_ϵ, plus their descendant 2-nodes, 1-nodes and 0-nodes.

At most $O(\log n)$ 3-nodes can contain a vertex of Q_ϵ. By Lemma 5.2.6(iii) each of them may have a 2-subtree $\mathcal{T}$ with query time $O(\frac{\lambda}{\epsilon}\log^2 n + k_\epsilon(\mathcal{T}))$, leading to a total of $O(\frac{\lambda}{\epsilon}\log^3 n + k_\epsilon)$ visited nodes in this category.

Since the number of visited nodes of each category is within the claimed bound, this proves the theorem. □

Remark 5.2.8 *If the query range has bounded aspect ratio, then it can be shown that the number of visited nodes reduces to $O(\min_{0<\epsilon\leqslant 1}\{\frac{\lambda}{\epsilon}\log^4 n + k_\epsilon\})$.*

5.2.3 Pipes and low-density scenes

Our research is motivated by the MOLOG project [Molog], where we need to perform collision checking in CAD models of industrial installations such as in Fig. 5.1. Let S be the set of bounding boxes in the given scene. For the analysis we assume that S can be partitioned into two subsets S_P and S_D, such that S_P is a set of *pipes* and S_D forms a *low-density scene* [Brg97KSV, Sta98]. These concepts are defined as follows.

Definition 5.2.9 *Let b be a 3-dimensional axis-parallel box, and consider its length in x-, y-, and z-direction. The box b is called a β-*pipe *if the shortest of these three lengths is at most β times shorter than the shortest-but-one.*

Next we define the density of a scene, specialized to sets of boxes. (The original definition by van der Stappen and Overmars [Sta98] uses balls instead of cubes, but this is equivalent up to a constant.)

Definition 5.2.10 *A set B of boxes in 3-space has* density δ *if the following holds: any cube C is intersected by at most δ boxes from B whose longest edge is longer than the edge length of C.*

Recall that the stabbing number of a set of boxes is defined as the maximum number of boxes with a non-empty intersection. Next we show that low-density sets and sets of pipes with low stabbing number also have low slicing number, which means that we can use the analysis of the previous subsection.

Lemma 5.2.11 *Let $S = S_P \cup S_D$ be a set of boxes in 3-space such that S_P is a set of β-pipes with stabbing number σ and S_D has density δ. Then the slicing number of S is at most $(\beta + 2)\sigma + \delta$.*

Proof: Let C be a cube of edge length c. Since a box that slices C has edge length at least c, the set S_D has slicing number at most δ.

It remains to bound the number of pipes slicing C. A pipe slicing C has to occupy a volume of at least $c \times c \times c/\beta = c^3/\beta$ in the cube, unless it contains one of the six sides of the cube completely. In the latter case, the pipe has to contain either the top-right-back corner or the bottom-left-front corner of C, and since each of these corners can be contained in at most σ input boxes, there can be at most 2σ such pipes. To bound the number of pipes in the former case, we observe that the total volume of the intersection of the pipes with C is at most σc^3. Therefore, the total number of boxes slicing the cube is at most $\delta + 2\sigma + \sigma c^3/(c^3/\beta) = \delta + (\beta + 2)\sigma$. □

By putting together Lemma 5.2.11 and Theorem 5.2.7, we get the following corollary.

Corollary 5.2.12 *Let $S = S_P \cup S_D$ be a set of boxes in 3-space such that S_P is a set of β-pipes with stabbing number σ and S_D has density δ. There is a box-tree for S such that the number of nodes visited by a range query with a query box Q is $O(\min_{0<\epsilon\leqslant 1}\{\frac{1}{\epsilon}(\frac{1}{\epsilon} + \lambda)\log^4 n + k_\epsilon\})$, where $\lambda = \delta + (\beta + 2)\sigma$ and k_ϵ is the number of boxes intersecting the extended range Q_ϵ.*

5.2.4 Analysis for other types of ranges

In the previous sections we assumed that the query range Q is an axis-parallel box. In this section we will generalize our results to constant-complexity ranges of arbitrary shape. A 3D query range is said to have constant complexity if its boundary consists of a constant number of algebraic surface patches of constant maximum degree, which are in turn bounded by a constant number of curves of constant maximum degree. In the analysis we only need the restriction that ∂Q, the boundary of Q, has a constant number of local extrema in any orthogonal cross-section, which is a condition fulfilled by the constant-complexity requirement.

We first prove a general theorem, that states that an LSF-interval tree with good query complexity for approximate range queries with boxes also has good query complexity for approximate range queries with other shapes. To this end we define a node ν to be *chargeable* with respect to a given range if all input boxes stored in $\mathcal{T}_\nu$ intersect that range, or if ν has a child with this property. Nodes for which this is not the case are *unchargeable*.

Theorem 5.2.13 *Let $\mathcal{T}$ be a d-dimensional box-tree on a set of n boxes, with $d \in \{2,3\}$. Suppose that, for any $0 < \epsilon \leqslant 1$, a query with a box B visits $O(f(n,\epsilon))$ nodes that are unchargeable with respect to the extended query box B_ϵ. Then a query with a constant-complexity range Q visits $O(\min_{0<\epsilon\leqslant 1}\{(\frac{1}{\epsilon})^{d-1}f(n,1) + k_\epsilon\})$ nodes of $\mathcal{T}$, where k_ϵ is the number of objects intersecting the ϵ-extended query range Q_ϵ.*

Proof: We first prove the theorem for $d = 2$.

Fix any $0 < \epsilon \leqslant 1$. We claim that we can cover ∂Q by $O(\frac{1}{\epsilon})$ squares of edge length $\epsilon w/3$, where w is the diameter of Q (as was also shown for convex ranges by Arya and Mount [Ary00]). To see this, consider a regular grid whose cells have size $\epsilon w/3$. Then ∂Q will intersect only $O(\frac{1}{\epsilon})$ grid cells, because for any two adjacent cells intersected by a connected portion of ∂Q the following holds: either they contain a local extremum of ∂Q, or the length of the portion of ∂Q within the cells is at least $\epsilon w/3$. Since the total length of ∂Q is $O(w)$, only $O(\frac{1}{\epsilon})$ grid cells can contain a portion of ∂Q of size $O(\epsilon w)$.

Now consider a query with a range Q. The number of visited nodes that are chargeable with respect to Q_ϵ is clearly $O(k_\epsilon)$. Any visited unchargeable node must have a bounding box that intersects at least one of the squares in the covering of ∂Q. To bound the number of such nodes, consider a square s in the covering. Define its extended square $s_{\epsilon'}$ as the set of points within L_∞-distance $\epsilon'\epsilon w/3$ from s. The boundary of the extended square has edge length $(1+2\epsilon')\epsilon w/3$ and intersects ∂Q, so even for ϵ' as large as 1, it is fully contained in Q_ϵ. Hence, any node that is unchargeable with respect to Q_ϵ is unchargeable with respect to $s_{\epsilon'}$ for $\epsilon' = 1$. The number of nodes ν such that $b(\nu)$ intersects s and that are unchargeable with respect to $s_{\epsilon'}$ is $O(f(n,\epsilon'))$. Summing over all squares s and plugging in $\epsilon' = 1$, we get a bound of $O(\frac{1}{\epsilon}f(n,1))$ on the number of unchargeable nodes.

Hence, the total number of visited nodes is bounded by $O(\frac{1}{\epsilon}f(n,1)+k_\epsilon)$, as claimed.

The proof for $d = 3$ is similar. We start by covering ∂Q by cubes of edge length $\epsilon w/3$, where w is the diameter of Q. We claim that ∂Q intersects $O((\frac{1}{\epsilon})^2)$ cells of a regular grid with cells of the required size. Indeed, any intersected cell must have an intersected facet, so we can bound the number of intersected cells by summing the number of intersected facets over all $O(\frac{1}{\epsilon})$ grid planes intersecting Q. Since ∂Q consists of a constant number of algebraic surface patches of constant maximum degree, which are in turn bounded by a constant number of curves of constant maximum degree, the same must hold for the intersection of ∂Q with a grid plane. Therefore, at most $O(\frac{1}{\epsilon})$ facets can be intersected in each grid plane, and it follows that Q can be covered using $O((\frac{1}{\epsilon})^2)$ cubes of the required size. From here we can follow the proof for the case $d = 2$. $\square$

The analysis of the previous section shows that in all bounds derived there, the $O(k_\epsilon)$ term on the number of visited internal nodes is caused solely by nodes with a priority leaf as a child that stores a box intersecting the extended query range. Such nodes are chargeable, so Theorem 5.2.13 and Corollary 5.2.12 together imply the following result.

Corollary 5.2.14 *Let $S = S_P \cup S_D$ be a set of boxes in 3-space such that S_P is a set of β-pipes with stabbing number σ and S_D has density δ. There is a box-tree for S such that the number of nodes visited by a range query with a constant-complexity range Q is $O(\min_{0<\epsilon\leqslant 1}\{(\lambda/\epsilon^2)\log^4 n + k_\epsilon\})$, where $\lambda = \delta + (\beta+2)\sigma$ and k_ϵ is the number of boxes intersecting the extended range Q_ϵ.*

Remark 5.2.15 *The dependency on ϵ that we get is better by a factor of $O(\frac{1}{\epsilon})$ than what Dickerson* et al. *[Dic00] and Arya and Mount [Ary00] get for queries with non-convex query ranges in point sets. Applying Theorem 5.2.13 to their structure, however, improves the dependency on ϵ by a factor of $O(\frac{1}{\epsilon})$, leading to the same dependency as we get.*

5.3 The BBD-interval tree

The bounding-volume hierarchy of the previous section is based on the longest-side-first kd-tree. It turns out that we can improve the results if we base the bounding-volume hierarchy on the so-called BBD-tree by Arya *et al.* [Ary00]. The resulting hierarchy is somewhat unorthodox, however, as it uses non-convex bounding volumes.

Define a *donut* to be the set-theoretic difference of two boxes, one being contained in the other. That is, a donut is defined as $R^+ \setminus R^-$, where R^+ and R^- are boxes and $R^- \subset R^+$. The inner box R^- may be empty, in which case a donut is simply a box. The inner box may also touch the boundary of the outer box, in which case a degenerate type of donut results. It is not allowed to split the outer box, that is, $R^+ \setminus R^-$ should be connected. A *bounding donut* of a set of objects is a donut $R^+ \setminus R^-$ that contains all objects and whose outer box R^+ is the bounding box of the set. A *donut tree* for a set of objects is a bounding-volume hierarchy that uses bounding donuts.

Like a kd-tree, the BBD-tree by Arya *et al.* is a tree representing a recursive decomposition of space. Unlike in a kd-tree, however, the regions corresponding to the nodes of a BBD-tree are not boxes — they are donuts. It is possible to construct a donut tree on a set of boxes using a BBD-tree in a similar way as one can construct a box-tree from a kd-tree. The main advantage is that BBD-trees have a stronger 'packing property' than kd-trees: whereas in a longest-side-first kd-tree there can be $O(\log^{d-1} n)$ nodes whose regions are disjoint and intersect opposite facets of a cube, there can be only $O(1)$ such nodes in a BBD-tree [Ary00]. This is the main reason that we can show the following result.

Theorem 5.3.1 *Let S be a set of boxes in 3-space with slicing number λ. There is a donut-tree for S such that a query with a box Q visits $O(\min_{0<\epsilon\leqslant 1}\{\log^3 n + (\lambda/\epsilon)\log^2 n + (\lambda/\epsilon^2)\log n + k_\epsilon\})$ nodes, where k_ϵ is the number of boxes intersecting the extended range Q_ϵ.*

This theorem can also be combined with Theorem 5.2.13 to get the following result:

Corollary 5.3.2 *Let S be a set of boxes in 3-space with slicing number λ. There exists a donut-tree for S such that a query with a constant-complexity range Q visits $O(\min_{0<\epsilon\leqslant 1}\{(1/\epsilon^2)\log^3 n + (\lambda/\epsilon^2)\log^2 n + k_\epsilon\})$ nodes, where k_ϵ is the number of boxes intersecting the extended range Q_ϵ.*

As mentioned in the introduction, the details of the construction of the donut-tree and the analysis of its performance are similar to those of the LSF-interval tree, but still rather technical. Therefore we omit the details here. The interested reader can find them in the technical report [Hav02a] on which this article is based.

5.4 Concluding remarks

We have developed a new algorithm to construct box-trees, and analyzed its performance for approximate range queries when the input is a low-density scene combined with (almost) disjoint pipes. We proved that in such a setting—which was motivated by the need to perform collision checking in CAD models of industrial installations—one can achieve polylogarithmic query times. This is in sharp contrast with the $\Omega(n^{2/3} + k)$ lower bound for the query time in box-trees for arbitrary input proved by Agarwal *et al.* [Aga01BGHH]. Our bounds almost match the best known bounds for range queries using box-trees in the much simpler case of point data.

The assumptions we use in the analysis cannot be relaxed much further. In particular, we can give a lower bound construction showing that it is not possible to achieve polylogarithmic performance for box-trees when the input is uncluttered [Brg97KSV] instead of having low-density, even for approximate queries.

Our results can be used to perform ϵ-approximate nearest-neighbor searching, using the techniques described for instance in Duncan's thesis [Dun99]. Thus, for input scenes satisfying the requirements above, approximate nearest-neighbor queries take time $O((\lambda/\epsilon^2)(\log^4 n)(\log\lambda + \log\frac{1}{\epsilon} + \log\log n))$ with our LSF-interval-tree, or $O(((1/\epsilon^2)\log^3 n + (\lambda/\epsilon^2)\log^2 n)(\log\lambda + \log\frac{1}{\epsilon} + \log\log n))$ in our BBD-interval-tree. (Note that for nearest-neighbor searching, ϵ is given as part of the query.)

In our future work we plan to investigate the performance of box-trees experimentally. We want to fine-tune our algorithm for constructing box-trees—in particular, we want to investigate whether the use of priority leaves, which are so convenient in the theoretical analysis, pays off in practice—and we want to compare it to existing heuristics.

Chapter 6

Facility location and the geometric minimum-diameter spanning tree

Previously published as: Joachim Gudmundsson, Herman J. Haverkort, Sang-Min Park, Chan-Su Shin and Alexander Wolff: Facility location and the geometric minimum-diameter spanning tree, in *Computational Geometry—Theory and Appl.*, 27(1), 2004, p87–106

Abstract. *Let P be a set of n points in the plane. A dipolar spanning tree $\mathcal{T}_{pq}$ of P is a tree that spans P and has exactly two nodes of degree greater than one, namely p and q. We can think of p and q as "facilities", while the other nodes are "clients".*

The geometric minimum-diameter spanning tree *(MDST) of P is a tree that spans P and minimizes the Euclidian length of the longest path. It is known that there is always a mono- or a dipolar MDST, i.e. the MDST is a tree with only one node greater than one, or it is a dipolar tree $\mathcal{T}_{pq}$ that minimizes the distance between any two clients. So far, a dipolar MDST can only be found in slightly subcubic time.*

The discrete two-center-problem *(2CP) is to find a tree $\mathcal{T}_{pq}$ of P that minimizes the distance of any client to the closest facility. For this problem, an $O(n^{4/3}\log^5 n)$-time algorithm is known.*

In this paper, we define an intermediate problem: find the tree $\mathcal{T}_{pq}$ that minimizes the distance of any client to the other *facility, that is, the sum of the distance to the closest facility and the distance between the two facilities. We call such a tree a* minimum-sum dipolar spanning tree *(MSST). We show that the MSST of any set P can be found in $O(n^2 \log n)$ time. A variant of the MSST-algorithm yields a factor-$\frac{4}{3}$ approximation of the MDST.*

Furthermore, we give two fast approximation schemes for the MDST, i.e. factor-($1+\varepsilon$) approximation algorithms. One algorithm uses a grid and takes time $O^*((\frac{1}{\varepsilon})^{5+2/3}+n)$, where the O^*-notation hides terms of type $O(\log^{O(1)}\frac{1}{\varepsilon})$. The other uses the well-separated pair decomposition and takes $O((\frac{1}{\varepsilon})^3 n+\frac{1}{\varepsilon}n\log n)$ time. A combination of the two approaches runs in $O^*((\frac{1}{\varepsilon})^5+n)$ time. Both schemes can also be applied to 2CP and MSST.

6.1 Introduction

The MDST can be seen as a network without cycles that minimizes the maximum travel time between any two sites connected by the network. This is of importance, e.g. in communication systems where the maximum delay in delivering a message is to be minimized. Ho et al. showed there is always a mono- or a dipolar MDST [Ho91]. For a different proof, see [Has95]. Ho et al. also gave an $O(n\log n)$-time algorithm for the monopolar and an $O(n^3)$-time algorithm for the dipolar case [Ho91]. In addition, they showed that the problem becomes considerably easier when allowing Steiner points, i.e. to find a spanning tree with minimum diameter over all point sets P' that contain the input point set P. The reason is that there always is a minimum-diameter Steiner tree that is monopolar and whose pole is the center of the smallest enclosing circle of P. Thus the minimum-diameter Steiner tree can be determined in linear time [Ho91].

The cubic time bound for the dipolar case was recently improved by Chan [Chn02] to $\tilde{O}(n^{3-c_d})$, where $c_d = 1/((d+1)(\lceil d/2\rceil+1))$ is a constant that depends on the dimension d of the point set and the $\tilde{O}$-notation hides factors that are $o(n^\varepsilon)$ for any fixed $\varepsilon > 0$. In the planar case $c_d = 1/6$. Chan speeds up the exhaustive-search algorithm of Ho et al. by using new semi-dynamic data structures. Note however that c_d tends to 0 with increasing d, while the asymptotic running time of the algorithm of Ho et al. does not depend on the dimension.

Note that in the dipolar case the objective is to find the two poles $x, y \in P$ of the tree such that the function $r_x + |xy| + r_y$ is minimized, where $|xy|$ is the Euclidean distance of x and y, and r_x and r_y are the radii of two disks centered at x and y whose union covers P. On the other hand the *discrete k-center problem* is to determine k points in P such that the union of k congruent disks centered at the k points covers P and the radius of the disks is minimized. This is a typical facility location problem: there are n supermarkets and in k of them a regional director must be placed such that the maximum director-supermarket distance is minimized. This problem is NP-hard provided that k is part of the input [Gry79]. Thus, the main research on this problem has focused on small k, especially on $k = 1, 2$. For $k = 1$, the problem can be solved in $O(n\log n)$ time using the farthest-point Voronoi diagram of P. For $k = 2$, the problem becomes considerably harder. Using the notation from above, the discrete two-center problem consists of finding two centers $x, y \in P$ such that the function $\max\{r_x, r_y\}$ is minimized. Agarwal et al. [Aga98SW] gave the first subquadratic-time algorithm for this problem. It runs in $O(n^{4/3}\log^5 n)$ time.

In this paper we are interested in (a) a new facility location problem that mediates between the minimum-diameter dipolar spanning tree (MDdST) and the two-center problem and (b) fast approximations of the computationally expensive MDdST. As for our first aim we observe the following. Whereas the MDdST minimizes $|xy| + (r_x + r_y)$, the discrete two-center problem is to minimize $\max\{r_x, r_y\}$, which means that the distance between the two centers is not considered at all. If, however, the two centers need to communicate with each other for cooperation, then their distance should be considered as well—not only the radius of the two disks. Therefore our aim is to find two centers x and y that minimize $|xy| + \max\{r_x, r_y\}$, which is a compromise between the two previous objective functions. We will refer to this problem as the *discrete minimum-sum two-center problem* and call the resulting graph the *minimum-sum dipolar spanning tree* (MSST). As it turns out, our algorithm for the MSST also constitutes a compromise, namely in terms of runtime between the subcubic-time MDdST-algorithm and the superlinear-time 2CP-algorithm. More specifically, in Section 6.2 we will describe an algorithm that solves the discrete minimum-sum two-center problem in the plane in $O(n^2 \log n)$ time using $O(n^2)$ space. For dimension $d < 5$ a variant of our algorithm is faster than the more general $\tilde{O}(n^{3-c_d})$-time MDST-algorithm of Chan [Chn02] that can easily be modified to compute the MSST instead.

In Section 6.3 we turn to our second aim, approximations for the MDST. We combine a slight modification of the MSST with the minimum-diameter monopolar spanning tree (MDmST). We identify two parameters that depend on the MDdST and help to express a very tight estimation of how well the two trees approximate it. It turns out that at least one of them is a factor-4/3 approximation of the MDST.

Finally, in Section 6.4 we show that there are even strong linear-time approximation schemes (LTAS) for the MDST, i.e. algorithms that given a set P of n points and some $\varepsilon > 0$ compute in $O^*(E^c + n)$ time a spanning tree whose diameter is at most $(1 + \varepsilon)$ times as long as the diameter of a MDST. In the runtime expression $E = 1/\varepsilon$, c is a constant and the O^*-notation hides terms of type $O(\log^{O(1)} E)$. The existence of a strong LTAS for the MDST has independently been proven by Spriggs at al. [Spr03]. Their LTAS is of order $c = 3$, i.e. it takes $O^*(E^3 + n)$ time.

Our results are as follows. Our first LTAS uses a grid of $O(E) \times O(E)$ square cells and runs Chan's exact algorithm [Chn02] on one representative point per cell. The same idea has been used before [Bar99, Chn00] to approximate the diameter of a point set, i.e. the longest distance between any pair of the given points. Our first LTAS is of order $5\frac{2}{3}$.

Our second approximation scheme is based on the well-separated pair decomposition [Cal95] of P and takes $O(E^3 n + En \log n)$ time. The well-separated pair decomposition will help us to limit our search for the two poles of an approximate MDdST to a linear number of point pairs. If we run our second scheme on the $O(E^2)$ representative points in the grid mentioned above, we get a LTAS of order five. Both schemes can be adjusted to approximate the MSST and the 2CP within the same time bounds.

We will refer to the diameter d_P of the MDST of P as the *tree diameter* of P. We assume that P contains at least four points.

6.2 The minimum-sum dipolar spanning tree

It is simple to give an $O(n^3)$-time algorithm for computing the MSST. Just go through all $O(n^2)$ pairs $\{p, q\}$ of input points and compute in linear time the point m_{pq} whose distance to the current pair is maximum. In order to give a faster algorithm for computing the MSST, we need a few definitions. Let h_{pq} be the closed halfplane that contains p and is delimited by the perpendicular bisector b_{pq} of p and q. Note that $h_{pq} \cap h_{qp} = b_{pq} = b_{qp}$. Let $\mathcal{T}_{pq}$ be the tree with dipole $\{p, q\}$ where all other points are connected to the closer pole. (Points on b_{pq} can be connected to either p or q.) Clearly the tree $\mathcal{T}_{pq}$ that minimizes $|pq| + \min\{|pm_{pq}|, |qm_{pq}|\}$ is an MSST. The following two observations will speed up the MSST computation.

We first observe that we can split the problem of computing all points of type m_{pq} into two halves. Instead of computing the point m_{pq} farthest from the pair $\{p, q\}$, we compute for each ordered pair (p, q) a point $f_{pq} \in P \cap h_{pq}$ that is farthest from p. See Figure 6.1 for an example. Now we want to find the tree $\mathcal{T}_{pq}$ that minimizes $|pq| + \max\{|pf_{pq}|, |qf_{qp}|\}$. We will see that other than the points of type m_{pq} we can compute those of type f_{pq} in batch.

Our algorithm consists of two phases, see Algorithm 1. In phase I we go through all points p in P. The central (and time-critical) part of our algorithm is the procedure COMPUTEALLFARTHEST that computes f_{pq} for all $q \in P \setminus \{p\}$. For a trivial $O(n^2)$-time implementation of this procedure, see Algorithm 2. In phase II we then use the above form of our target function to determine the MSST. The second important observation that helped us to speed up COMPUTE-ALLFARTHEST is the following. Let p be fixed. Instead of computing f_{pq} for each $q \in P \setminus \{p\}$ individually, we characterize in Lemma 6.2.2 all q that have the

Algorithm 1 $\mathrm{MSST}(P)$

```
Phase I: compute all $f_{pq}$
  for each $p \in P$ do
    COMPUTEALLFARTHEST$(P, p)$
  end for $p$
Phase II: search for MSST
  for each $\{p, q\} \in \binom{P}{2}$ do
    $d_{pq} \leftarrow |pq| + \max\{|pf_{pq}|, |qf_{qp}|\}$
  end for $\{p, q\}$
return $\mathcal{T}_{pq}$ with $d_{pq}$ minimum.
```

Algorithm 2 ComputeAllFarthest(P, p) {first version}

```
for each $q \in P \setminus \{p\}$ do
  $f_{pq} \leftarrow p$
  for each $r \in P \setminus \{p, q\}$ do
    if $r \in h_{pq}$ and $|pr| > |pf_{pq}|$ then
      $f_{pq} \leftarrow r$
    end if
  end for $q$
end for $r$
return $f_{pq}$ for each $q \in P \setminus \{p\}$
```

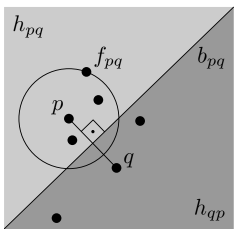


Figure 6.1: f_{pq} denotes the point farthest from p in $P \cap h_{pq}$.

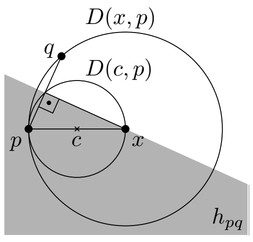


Figure 6.2: $x \in h_{pq}$ if and only if $q \notin D(x,p)$.

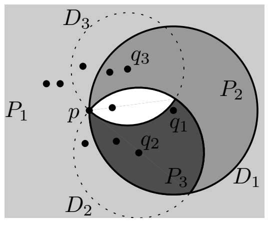


Figure 6.3: Computing the points of type f_{pq} in batch.

same f_{pq}. Our characterization uses the following direct consequence of Thales' Theorem. See Figure 6.2 as illustration.

Fact 6.2.1 Let $D(x,p)$ be the open disk that is centered at x and whose boundary contains p ($D(x,p) = \emptyset$ if $x = p$). Then $x \in h_{pq}$ if and only if $q \notin D(x,p)$.

Lemma 6.2.2 *x is farthest from p in $P \cap h_{pq}$ if and only if $q \notin D(x,p)$ and, for all $x' \in P$ with $|px'| > |px|$, $q \in D(x',p)$.*

Proof: "If" part: Due to $q \notin D(x,p)$ and Fact 6.2.1 we know that x lies in h_{pq}. Fact 6.2.1 also yields that all $x' \in P$ with $|px'| > |px|$ do *not* lie in h_{pq} since $q \in D(x',p)$ for all such x'. Thus x is farthest from p among all points in h_{pq}.

"Only if" part: suppose $q \in D(x,p)$ or suppose there is an $x' \in P$ with $|px'| > |px|$ and $q \notin D(x',p)$. In the former case we would have $x \notin h_{pq}$, in the latter $|px'| > |px|$ and $x' \in h_{pq}$. Both would contradict x being farthest from p among the points in h_{pq}. □

Lemma 6.2.2 immediately yields a way to set the variables f_{pq} in batch: go through the points $q_i \in P \setminus \{p\}$ in order of non-increasing distance from p, find all points in $P_i = P \setminus D(q_i,p)$, set f_{pq} to q_i for all $q \in P_i$, remove the points in

Algorithm 3 COMPUTEALLFARTHEST(P, p) {second version}

```
1: sort $P = q_1, \dots, q_n$ such that $|pq_1| \geq |pq_2| \geq \cdots \geq |pq_n|$
2: $P \leftarrow P \setminus \{p\}$
3: for $i \leftarrow 1$ to $n$ do
4:     $P_i \leftarrow P \setminus D(q_i, p)$
5:     for each $q \in P_i$ do
6:         $f_{pq} \leftarrow q_i$
7:     end for $q$
8:     $P \leftarrow P \setminus P_i$
9: end for $i$
10: return $f_{pq}$ for each $q \in P \setminus \{p\}$
```

P_i from P, and continue—see the second version of COMPUTEALLFARTHEST in Algorithm 3.

Figure 6.3 visualizes what happens in the first three runs through the outer for-loop of Algorithm 3: the areas shaded light, medium, and dark contain all points q with $f_{pq} = q_1$, $f_{pq} = q_2$, and $f_{pq} = q_3$, respectively. We used D_i as shorthand for $D(q_i, p)$.

Lemma 6.2.3 *For each $q \in P \setminus \{p\}$ Algorithm 3 sets f_{pq} to the point farthest from p in $P \cap h_{pq}$.*

Proof: Note that $P \setminus \{p\} = \bigcup_{i=1}^{n} P_i$. This is due to the fact that $D(q_n, p) = D(p, p) = \emptyset$. Thus the variables f_{pq} are in fact set for all $q \in P \setminus \{p\}$ in line 6 of Algorithm 3.

The values that are assigned to the f_{pq}'s are correct due to the order, in which the outer for-loop runs through the points in P: f_{pq} is set to q_i if i is the smallest index such that $q \in P_i$. This is the case if i is the smallest index such that $q \notin D(q_i, p)$ and $q \in D(q_j, p)$ for $j < i$. Since the q_j with $j < i$ are exactly the points in P farther from p than q_i, Lemma 6.2.2 yields that q_i is the point farthest from p in $P \cap h_{pq}$. $\square$

The remainder of this section deals with efficiently finding points in P_i. We give two methods. The first, which is slightly slower in the plane, also works for higher dimensions.

Method I. We use dynamic circular range searching, which is a special case of halfspace range searching in $\mathbb{R}^3$ via orthogonal projection to the paraboloid $\{(x, y, z) \mid z = x^2 + y^2\}$ [Aga95]. The necessary data structure can be build in $O(n^{1+\varepsilon})$ time and space for an arbitrarily small $\varepsilon > 0$. After each query with the halfspace H_i corresponding to the complement of the disk $D(q_i, p)$ all points in H_i must be deleted (according to step 8 of Algorithm 3). The total time for querying and deleting is $O(n^{1+\varepsilon})$. This yields an $O(n^{2+\varepsilon})$-time algorithm for finding the MSST. We will give a faster algorithm for the planar case. However, it is not clear how that algorithm can be generalized to higher dimensions. For dimensions $d \in \{3, 4\}$ computing the MSST with range searching takes $O(n^{2.5+\varepsilon})$ time [Aga95]. This is faster than Chan's MDST-algorithm [Chn02] that can easily be modified to compute the MSST instead. His algorithm runs in $\tilde{O}(n^{3-c_d})$ time, where $c_d = 1/((d+1)(\lceil d/2 \rceil + 1)) \le 1/12$ for $d \ge 3$.

Method II. We compute a partition of the plane into regions $R_1, \dots, R_n$ such that R_i contains the set P_i of all points q with $f_{pq} = q_i$. Then we do a plane sweep to determine for each $q \in P \setminus \{p\}$ the region R_i that contains it. Method II takes $O(n \log n)$ time and thus yields an $O(n^2 \log n)$-time algorithm for finding the MSST in the plane. We will use the following simple fact.

Fact 6.2.4 Given a set $\mathcal{D}$ of disks in the plane that all touch a point p, each disk contributes at most one piece to the boundary of $\bigcap \mathcal{D}$.

This helps us to bound the complexity of our planar partition.

Lemma 6.2.5 *Let $\mathcal{D} = \{D_1, \dots, D_{n-1}\}$ be a set of disks in the plane, let $D_0 = \mathbb{R}^2$, $D_n = \emptyset$, and for $i = 1, \dots, n$ let $I_i = D_0 \cap \dots \cap D_{i-1}$ and $R_i = I_i \setminus D_i$. Then $\mathcal{R}(\mathcal{D}) = \{R_1, \dots, R_n\}$ is a partition of the plane whose complexity—the total number of arcs on the boundaries of $R_1, \dots, R_n$—is $O(n)$.*

Proof: The region R_i consists of all points that lie in $D_0, \dots, D_{i-1}$ but not in D_i. Since $D_0 = \mathbb{R}^2$ and $D_n = \emptyset$ it is clear that $\mathcal{R}(\mathcal{D})$ is in fact a partition of the plane. For an example refer to Figure 6.4, where the regions R_1, R_2, R_3, and R_4 are shaded from light to dark gray.

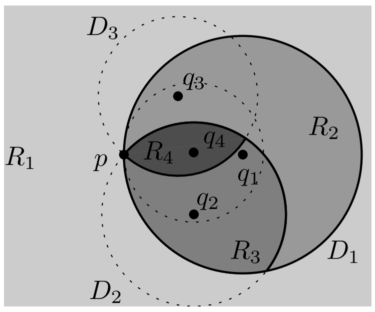


Figure 6.4: Regions of the partition $\mathcal{R}(\mathcal{D})$.

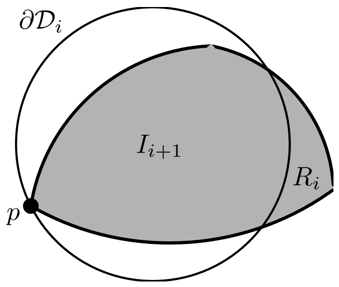


Figure 6.5: A step in the incremental construction of $\mathcal{R}(\mathcal{D})$.

If $\mathcal{R}(\mathcal{D})$ is constructed incrementally, each new disk D_i splits I_i into I_{i+1} and R_i. For illustration, see Figure 6.5, where I_i is the shaded region. Due to Fact 6.2.4, D_i contributes at most one circular arc A to the boundary of I_{i+1}. The start- and endpoint of A can split two arcs on the boundary of I_i into two pieces each. Two of these at most four pieces will belong to I_{i+1} and two to R_i. Thus the number of arcs in $\mathcal{R}(\mathcal{D})$ increases at most by three when adding a new disk to the current partition. □

Now we can give a faster implementation of COMPUTEALLFARTHEST for the planar case. More specifically we will show the following.

Lemma 6.2.6 *Let $D_0 = \mathbb{R}^2$ and $D_n = \emptyset$. Given a set Q of m points and a set $\mathcal{D} = \{D_1, \dots, D_{n-1}\}$ of disks in the plane that all touch a point p, there is an algorithm that computes in $O((m+n)\log n)$ total time for each point $q \in Q$ the region $R \in \mathcal{R}(\mathcal{D})$ that contains q. The algorithm needs $O(n+m)$ space.*

Proof: We first construct the partition $\mathcal{R}(\mathcal{D})$ and then do a plane sweep to locate the points in Q in the cells of $\mathcal{R}(\mathcal{D})$.

We use the incremental construction of $\mathcal{R}(\mathcal{D})$ as in Lemma 6.2.5. In order to find the two points where the boundary ∂D_i of a new disk D_i intersects the boundary ∂I_i of I_i we need a data structure $\mathcal{T}$ that stores the circular arcs on ∂I_i. The data structure must allow us to do search, remove, insert, and successor operations in logarithmic time. This is standard, e.g. for red-black trees [Cor90]. In $\mathcal{T}$ we store the circular arcs on ∂I_i in clockwise order starting from p. We assume that p is the leftmost point of ∂I_i.

By Fact 6.2.4, ∂D_i intersects ∂I_i at zero, one, or two points other than p. Let A and B be the first respectively last arc on ∂I_i (both incident to p), and let A' and B' be infinitesimally small pieces of A respectively B incident to p. There are three cases, which can be distinguished from each other in constant time. For illustration see Figures 6.6 to 6.8, where we have sketched ∂I_i as a polygon for simplicity.

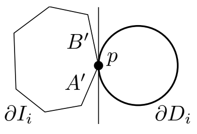


Figure 6.6:
Case (1)

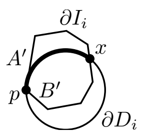


Figure 6.7:
Case (2)

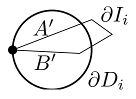


Figure 6.8:
Case (3)

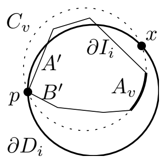


Figure 6.9:
Querying $\mathcal{T}$ at v

1. $D_i \cap I_i = \emptyset$.
   This can be verified by checking whether the tangent of D_i in p separates D_i from A and B. If this is the case, we are done since then $R_i = I_i$ and $R_j = I_j = \emptyset$ for all $j > i$.

2. $D_i \cap I_i \neq \emptyset$, and at least one of A' or B' lies outside I_i.
   We follow the part of ∂I_i from p that lies outside D_i until we reach an intersection point x (possibly again p) of ∂D_i and ∂I_i. On our way we replace all nodes in $\mathcal{T}$ that correspond to arcs outside D_i by a new node that corresponds to the arc $\partial D_i \cap I_i$ (bold in Figure 6.7). The region $R_i = I_i \setminus D_i$ is delimited by the new arc and all arcs on ∂I_i from p up to x.

3. $D_i \cap I_i \neq \emptyset$, and A' and B' lie inside I_i.
   We query $\mathcal{T}$ to find out whether and where ∂D_i and ∂I_i intersect other than in p. We follow a path from the root of $\mathcal{T}$ either to a node whose arc intersects ∂D_i, or to a leaf if ∂I_i and ∂D_i do not intersect. Let v be the current inner node of $\mathcal{T}$, A_v the corresponding arc and C_v the circle that contains A_v (and touches p). We consider the following two subcases:

   (a) $\partial D_i \cap C_v = \{p\}$.
   This occurs in the degenerate case that p and the centers of D_i and C_v are collinear. If C_v is contained in D_i, then I_i is also contained in D_i. Thus $R_i = \emptyset$, $I_{i+1} = I_i$, and we can continue with D_{i+1}.

   Otherwise A_v lies outside D_i—recall that $C_v, D_i \in \mathcal{D}$ and that all disks in $\mathcal{D}$ are pairwise different. Since A' and B' lie inside D_i, ∂D_i intersects two arcs on ∂I_i; one between A' and A_v, and one between A_v and B'. Thus we can continue to search for an intersection in any of the two subtrees of v.

   (b) $\partial D_i \cap C_v = \{p, x\}$ and $x \neq p$.
   If $x \in A_v$, we stop. Otherwise we continue our search in the left or right subtree of v depending on whether p, x, and A_v lie on C_v in clockwise or counterclockwise order, respectively. Note that in the clockwise case the part of ∂I_i from A_v to B' (in clockwise order) is completely contained in D_i, see Figure 6.9. Thus no arc in the right subtree of v intersects ∂D_i. The counterclockwise case is symmetric.

   If—in case (3b)—we reach a leaf of $\mathcal{T}$ without finding any intersection, this means that $\partial I_i \cap \partial D_i = \{p\}$, since in each step of the query we have only discarded arcs in $\mathcal{T}$ that lie on the portion of ∂I_i that cannot intersect ∂D_i. The fact that A' and B' lie inside D_i now yields $I_i \subseteq D_i$. Thus $R_i = \emptyset$, and we can continue with D_{i+1}.

   Otherwise we have found some $x \neq p$ that lies on $\partial D_i \cap \partial I_i$. If it turns out that ∂D_i and ∂I_i just touch in x, then we again have $I_i \subseteq D_i$, which means that $R_i = \emptyset$ and we can proceed with D_{i+1}. If, however, ∂D_i and ∂I_i intersect properly, then we continue as in case (2), following the part of ∂I_i outside D_i from x until we hit a second intersection point x'. Due to Fact 6.2.4 there cannot be any further intersection points.

Whenever we modify $\mathcal{T}$ we also do the necessary steps in the incremental construction of $\mathcal{R}(\mathcal{D})$: we create circular pointers around each R_i and pointers from each arc to the two regions it borders.

The plane sweep is practically the same as for locating points in a vertical decomposition of line segments. Our (multi-) set E of event points consists of the set V of vertices of $\mathcal{R}(\mathcal{D})$, the points in Q, and the set X of the left- and rightmost points of arcs that are not arc endpoints. Note that there is a linear order among the arcs in the vertical strip between any two consecutive event points in $V \cup X$. We store the points in E in an array and sort them according to non-decreasing

x-coordinate. The sweep-line status consists of the arcs in $\mathcal{R}(\mathcal{D})$ that are currently intersected by the vertical sweep line ℓ in the order in which they are intersected by ℓ. The sweep-line status $\S$ can be implemented by any balanced binary search tree (like a red-black tree) that allows insertion, deletion, and search in logarithmic time.

Each time ℓ hits an event point in X or V, we either add an arc to $\S$ or remove an arc from $\S$. (This assumes non-degeneracy of $\mathcal{D}$, i.e. no three disk boundaries intersect in a point other than p. The assumption can be overcome by using several event points for vertices in $\mathcal{R}(\mathcal{D})$ of degree greater than 3.) Each time ℓ hits an event point $q \in Q$ we determine the first arc in $\S$ above or below q and return the index i of the corresponding region R_i of $\mathcal{R}(\mathcal{D})$.

The data structure for $\mathcal{R}(\mathcal{D})$ can be set up in $O(n \log n)$ time due to Lemma 6.2.5: each step of the incremental construction takes $O(\log n)$ time for querying $\mathcal{T}$ and $O(|R_i| \log n)$ time for updating $\mathcal{T}$ and $\mathcal{R}(\mathcal{D})$. Lemma 6.2.5 also ensures that E consists of at most $O(n+m)$ points, and that $\S$ contains at most $O(n)$ arcs at any time during the sweep. Since each event point is processed in $O(\log n)$ time, the whole sweep takes $O((m+n) \log n)$ time. □

Now we can conclude:

Theorem 6.2.7 *There is an algorithm that computes an MSST in $O(n^2 \log n)$ time using quadratic space.*

Proof: We can implement COMPUTEALLFARTHEST by applying the algorithm of Lemma 6.2.6 to $\mathcal{D} = \{D(q_1, p), \ldots, D(q_{n-1}, p)\}$ and $Q = P \setminus \{p\}$. This yields a running time of $O(n \log n)$ for COMPUTEALLFARTHEST. With this subroutine, Algorithm 1 computes an MSST in $O(n^2 \log n)$ time. □

6.3 Approximating the minimum-diameter spanning tree

We first make the trivial observation that the diameter of *any* monopolar tree on P is at most twice as long as the tree diameter d_P of P. We use the following notation. Let $\mathcal{T}_{\mathrm{di}}$ be a fixed MDdST and $\mathcal{T}_{\mathrm{mono}}$ a fixed MDmST of P. The tree $\mathcal{T}_{\mathrm{di}}$ has minimum diameter among those trees with vertex set P in which all but two nodes—the poles—have degree 1. The tree $\mathcal{T}_{\mathrm{mono}}$ is a minimum-diameter star with vertex set P. Let x and y be the poles of $\mathcal{T}_{\mathrm{di}}$, and let $\delta = |xy|$ be their distance. Finally let r_x (r_y) be the length of the longest edge in $\mathcal{T}_{\mathrm{di}}$ incident to x (y, respectively) without taking into account the edge xy. Thus disks of radius r_x and r_y centered at x and y, respectively, cover P. Without loss of generality we assume $r_x \geq r_y$.

Ho et al. showed that in the dipolar case (i.e. if there is no monopolar MDST), the disk centered at y cannot be contained by the one centered at x. We will need this *stability lemma* below.

Lemma 6.3.1 *[Stability lemma [Ho91]]* $r_x < \delta + r_y$.

In order to get a good approximation of the MDST, we slightly modify the algorithm for the MSST described in Section 6.2. After computing the $O(n^2)$ points of type f_{pq}, we go through all pairs $\{p, q\}$ and consider the tree $\mathcal{T}_{pq}$ with dipole $\{p, q\}$ in which each point is connected to its closer dipole. In Section 6.2 we were searching for a tree of type $\mathcal{T}_{pq}$ that minimizes $|pq| + \max\{|f_{pq}p|, |qf_{qp}|\}$. Now we go through all trees $\mathcal{T}_{pq}$ to find the tree $\mathcal{T}_{\text{bisect}}$ with minimum *diameter*, i.e. the tree that minimizes $|pq| + |f_{pq}p| + |qf_{qp}|$. Note that the only edge in $\mathcal{T}_{pq}$ that crosses the perpendicular bisector of pq is the edge pq itself. This is of course not necessarily true for the MDdST $\mathcal{T}_{\text{di}}$. We will show the following:

Lemma 6.3.2 *Given a set P of n points in the plane there is a tree with the following two properties: it can be computed in $O(n^2 \log n)$ time using $O(n^2)$ storage, and its diameter is at most $4/3 \cdot d_P$.*

Proof: Due to Theorem 6.2.7 it suffices to show the approximation factor. We will first compute upper bounds for the approximation factors of $\mathcal{T}_{\text{bisect}}$ and $\mathcal{T}_{\text{mono}}$ and then analyze where the minimum of the two takes its maximum.

For the analysis of $\mathcal{T}_{\text{bisect}}$ consider the tree $\mathcal{T}_{xy}$ whose poles are those of $\mathcal{T}_{\text{di}}$. The diameter of $\mathcal{T}_{xy}$ is an upper bound for that of $\mathcal{T}_{\text{bisect}}$. Let r'_x (r'_y) be the length of the longest edge of $\mathcal{T}_{xy}$ incident to x (y, respectively) without taking into account the edge xy. Note that $r'_x = |xf_{xy}|$ and $r'_y = |yf_{yx}|$.

Now we compare the diameter of $\mathcal{T}_{xy}$ to the diameter of $\mathcal{T}_{\text{di}}$. Observe that $\max\{r'_x, r'_y\} \leq r_x$. This is due to our assumption $r_x \geq r_y$ and to the fact that f_{xy} and f_{yx} have at most distance r_x from both x and y. This observation yields $\text{dia}\, \mathcal{T}_{xy} = r'_x + \delta + r'_y \leqslant 2\max\{r'_x, r'_y\} + \delta \leqslant 2r_x + \delta$. Now we define two constants α and β that only depend on $\mathcal{T}_{\text{di}}$. Let

$$\alpha = \frac{\delta}{r_x + r_y} \quad \text{and} \quad \beta = \frac{r_x}{r_y}.$$

Note that $\alpha > 0$ and $\beta \geq 1$. Introducing α and β yields

$$\frac{\text{dia}\, \mathcal{T}_{\text{bisect}}}{\text{dia}\, \mathcal{T}_{\text{di}}} \leqslant \frac{\text{dia}\, \mathcal{T}_{xy}}{\text{dia}\, \mathcal{T}_{\text{di}}} \leqslant \frac{2r_x + \delta}{r_x + \delta + r_y} = \frac{\alpha(1+\beta) + 2\beta}{(1+\alpha)(1+\beta)} =: f_{\text{bisect}}(\alpha, \beta),$$

since $2r_x = 2\beta(r_x + r_y)/(1+\beta)$ and $\delta = \alpha(r_x + r_y)$. The function $f_{\text{bisect}}(\alpha, \beta)$ is an upper bound for the approximation factor that $\mathcal{T}_{\text{bisect}}$ achieves.

Now we apply our α-β-analysis to $\mathcal{T}_{\text{mono}}$. The stability lemma $r_x < \delta + r_y$ [Ho91] implies that all points in P are contained in the disk $D_{x,\delta+r_y}$ of radius $\delta + r_y$ centered at x, see Figure 6.10. Due to that, the diameter of a monopolar tree $\mathcal{T}$ that spans P and is rooted at x is at most twice the radius of the disk. We know that $\text{dia}\, \mathcal{T}_{\text{mono}} \leq \text{dia}\, \mathcal{T}$ since $\mathcal{T}_{\text{mono}}$ is the MDmST of P. Thus

$$\text{dia}\, \mathcal{T}_{\text{mono}} \leqslant 2(\delta + r_y) = 2\alpha(r_x + r_y) + \frac{2}{1+\beta}(r_x + r_y),$$

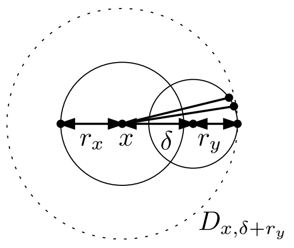


Figure 6.10: Approximating $\mathcal{T}_{\mathrm{di}}$ with $\mathcal{T}_{\mathrm{mono}}$.

since $\delta = \alpha(r_x + r_y)$ and $1+\beta = (r_x+r_y)/r_y$. Using $\operatorname{dia}\mathcal{T}_{\mathrm{di}} = (1+\alpha)(r_x+r_y)$ yields

$$\frac{\operatorname{dia}\mathcal{T}_{\mathrm{mono}}}{\operatorname{dia}\mathcal{T}_{\mathrm{di}}} \leqslant \frac{2\alpha(1+\beta)+2}{(1+\alpha)(1+\beta)} \;=:\; f_{\mathrm{mono}}(\alpha,\beta),$$

and the function $f_{\mathrm{mono}}(\alpha,\beta)$ is an upper bound of $\mathcal{T}_{\mathrm{mono}}$'s approximation factor.

In order to compute the maximum of the minimum of the two bounds we first analyze where $f_{\mathrm{bisect}} \le f_{\mathrm{mono}}$. This is always the case if $\alpha \ge 2$ but also if $\alpha < 2$ and $\beta \le g_{\mathrm{equal}}(\alpha) := \frac{\alpha+2}{2-\alpha}$. See Figure 6.11 for the corresponding regions. Since neither f_{bisect} nor f_{mono} have any local or global maxima in the interior of the (α,β)-range we are interested in, we must consider their boundary values.

1. For $\beta \equiv 1$ the tree $\mathcal{T}_{\mathrm{bisect}}$ is optimal since $f_{\mathrm{bisect}}(\alpha,1) \equiv 1$.

2. Note that the stability lemma $r_x \le \delta + r_y$ is equivalent to $\beta \le g_{\mathrm{stab}}(\alpha) := \frac{\alpha+1}{1-\alpha}$, see Figure 6.11. Along the graph of g_{stab} the tree $\mathcal{T}_{\mathrm{mono}}$ is optimal since $f_{\mathrm{mono}}(\alpha, g_{\mathrm{stab}}(\alpha)) \equiv 1$.

3. Along g_{equal} both functions equal $(3\alpha+2)/(2\alpha+2)$. This expression increases monotonically from 1 towards $4/3$ when α goes from 0 towards 2.

Standard analysis of the partial derivatives shows that f_{mono} increases while f_{bisect} decreases monotonically when α goes to infinity. So the maximum of $\min(f_{\mathrm{mono}}, f_{\mathrm{bisect}})$ is indeed attained at g_{equal}. □

6.4 Approximation schemes for the MDST

In this section we give some fast approximation schemes for the MDST, i.e. factor-$(1+\varepsilon)$ approximation algorithms. The first approximation scheme uses a grid, the second and third use the well-separated pair decomposition, and the fourth is a combination of the first and the third method. The reason for this multitude of approaches is that we want to take into account the way the running time

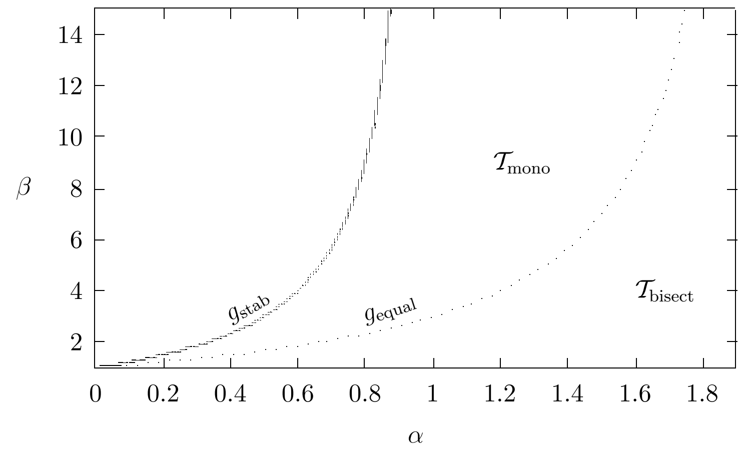


Figure 6.11: Our upper bound for the approximation factor of $\mathcal{T}_{\text{mono}}$ ($\mathcal{T}_{\text{bisect}}$) is smaller to the left (right, respectively) of g_{equal}. To the left of g_{stab} the tree $\mathcal{T}_{\text{mono}}$ is optimal.

depends not only on n, the size of the point set, but also on ε, the approximation factor.

Chan [Chn00] uses the following notation. Let $E = 1/\varepsilon$ and let the O^*-notation be a variant of the O-notation that hides terms of type $O(\log^{O(1)} E)$. (Such terms come into play e.g. when the use of the floor function is replaced by binary search with precision ε.) Then a *linear-time approximation scheme (LTAS) of order c* is a scheme with a running time of the form $O^*(E^c n)$ for some constant c. A *strong LTAS of order c* has a running time of $O^*(E^c+n)$. Our asymptotically fastest scheme for approximating the MDST is a strong LTAS of order 5.

6.4.1 A grid-based approximation scheme

The idea of our first scheme is based on a grid which has been used before e.g. to approximate the diameter of a point set [Bar99, Chn00], i.e. the longest distance between any pair of the given points. We lay a grid of $O(E) \times O(E)$ cells over P, choose an arbitrary representative point for each cell and use the exact algorithm of Ho et al. [Ho91] to compute the MDST $\mathcal{T}_R$ of the set R of all representative points. By connecting the remaining points in $P \setminus R$ to the pole adjacent to their representatives, we get a dipolar tree $\mathcal{T}_\varepsilon$ whose diameter is at most $(1 + \varepsilon)$ times the tree diameter d_P of P.

The details are as follows. Let $M = \max_{p,q \in P}\{|x(p)x(q)|, |y(p)y(q)|\}$ be the edge length of the smallest enclosing square of P and let $l = \varepsilon M/(10\sqrt{2})$ be the edge length of the square grid cells. Clearly $M \leq d_P$. Since each path in T_ε is at most by two edges of length $l\sqrt{2}$ longer than the corresponding path in T_R we have $\operatorname{dia} T_\varepsilon \;\leq\; \operatorname{dia} T_R + 2l\sqrt{2} \;\leq\; \operatorname{dia} T_R + \varepsilon d_P/5$. To see that $\operatorname{dia} T_\varepsilon \leq (1+\varepsilon)\, d_P$ it remains to prove:

Lemma 6.4.1 $\operatorname{dia} \mathcal{T}_R \;\leq\; (1 + 4\varepsilon/5)\, d_P$.

Proof: Let $\mathcal{T}_P$ be a MDST of P that is either mono- or dipolar. Such a tree always exists according to [Ho91].

Case I: $\mathcal{T}_P$ is monopolar. Let $x \in P$ be the pole of $\mathcal{T}_P$ and let $\rho_p \in R$ be the representative point of $p \in P$. Due to the definition of T_R we have

$$\operatorname{dia} \mathcal{T}_R \leqslant \min_{x' \in R} \max_{s \neq t \in R} |sx'| + |x't| \leqslant \max_{s \neq t \in R} |s\rho_x| + |\rho_x t|.$$

(The first two terms are equal if there is a monopolar MDST of R, the last two terms are equal if there is a MDmST of R with pole ρ_x.) By triangle inequality

$$\operatorname{dia} \mathcal{T}_R \leqslant \max_{s \neq t \in R} |sx| + |x\rho_x| + |\rho_x x| + |xt|,$$

i.e. we maximize the length of the polygonal chain (s, x, ρ_x, x, t) over all $s \neq t \in R$. By appending edges to points a and $b \in P$ in the grid cells of s and t, respectively, the length of the longest chain does not decrease, even if we now maximize over all $a, b \in P$ with $a \neq b$.

$$\operatorname{dia} \mathcal{T}_R \leqslant \max_{a \neq b \in P} |a\rho_a| + |\rho_a x| + 2|x\rho_x| + |x\rho_b| + |\rho_b b|.$$

Using $|a\rho_a|, |x\rho_x|, |\rho_b b| \leq l\sqrt{2}$ and the triangle inequalities $|\rho_a x| \leq |\rho_a a| + |ax|$ and $|x\rho_b| \;\leq\; |xb| + |b\rho_b|$ yields $\operatorname{dia} \mathcal{T}_R \;\leq\; 6l\sqrt{2} + \max_{a \neq b \in P} |ax| + |xb| = (1 + 3\varepsilon/5)d_P$.

Case II: $\mathcal{T}_P$ is dipolar. The analysis is very similar to case I, except the chains consist of more pieces. This yields $\operatorname{dia} \mathcal{T}_R \leq 8l\sqrt{2} + \operatorname{dia} \mathcal{T}_P \;=\; (1+4\varepsilon/5)\, d_P$. $\Box$

Theorem 6.4.2 *A spanning tree* $\mathcal{T}_P$ *of* P *with* $\operatorname{dia} \mathcal{T}_P \leq (1 + 1/E) \cdot d_P$ *can be computed in* $O^*(E^{6-1/3} + n)$ *time using* $O^*(E^2 + n)$ *space.*

Proof: In order to determine the grid cell of each point in P without the floor function, we do binary search—once on an x- and once on a y-interval of size M until we have reached a precision of l, i.e. we need $O(\log E)$ steps for each point. Using Chan's algorithm [Chn02] to compute T_R takes $\tilde{O}(|R|^{3-1/6})$ time and $\tilde{O}(|R|)$ space, where $|R| = O(E^2)$. $\Box$

6.4.2 The well-separated pair decomposition

Our second scheme uses the well-separated pair decomposition of Callahan and Kosaraju [Cal95]. We briefly review this decomposition below.

Definition 6.4.3 *Let $\tau > 0$ be a real number, and let A and B be two finite sets of points in $\mathbb{R}^d$. We say that A and B are* well-separated w.r.t. τ, *if there are two disjoint d-dimensional balls C_A and C_B both of radius r such that $A \subset C_A$, $B \subset C_B$, and the distance between C_A and C_B is at least equal to τr.*

The parameter τ will be referred to as the *separation constant*. The following lemma follows easily from Definition 6.4.3.

Lemma 6.4.4 *Let A and B be two finite sets of points that are well-separated w.r.t. τ, let x and p be points of A, and let y and q be points of B. Then (i) $|xy| \leq (1 + 2/\tau) \cdot |xq|$, (ii) $|xy| \leq (1 + 4/\tau) \cdot |pq|$, (iii) $|px| \leq (2/\tau) \cdot |pq|$, and (iv) the angle between the line segments pq and py is at most $\arcsin(2/\tau)$.*

Definition 6.4.5 *Let P be a set of n points in $\mathbb{R}^d$, and $\tau > 0$ a real number. A* well-separated pair decomposition *(WSPD) for P (w.r.t. τ) is a sequence of pairs of non-empty subsets of P, $(A_1, B_1), (A_2, B_2), \dots, (A_\ell, B_\ell)$, such that*

1. *A_i and B_i are well-separated w.r.t. τ for $i = 1, 2, \dots, \ell$, and*
2. *for any two distinct points p and q of P, there is exactly one pair (A_i, B_i) in the sequence such that (i) $p \in A_i$ and $q \in B_i$, or (ii) $q \in A_i$ and $p \in B_i$,*

The integer ℓ is called the *size* of a WSPD. Callahan and Kosaraju show that a WSPD of size $\ell = O(\tau^2 n)$ can be computed using $O(n \log n + \tau^2 n)$ time and space. The WSPD will help us to limit our search for the two poles of an approximate MDdST to a linear number of point pairs.

6.4.3 A straight-forward approximation scheme

The approximation algorithm consists of two subalgorithms: the first algorithm computes a MDmST and the second computes an approximation of the MDdST. We always output the one with smaller diameter. According to [Ho91] there exists a MDST that is either a monopolar or a dipolar tree. The MDmST can be computed in time $O(n \log n)$, hence we will focus on the problem of computing a MDdST. Let $d_{\min}$ be the diameter of a MDdST and let $\S_{pq}$ denote a spanning tree with dipole $\{p, q\}$ whose diameter is minimum among all such trees. For any dipolar spanning tree $\mathcal{T}$ with dipole $\{u, v\}$ let $r_u(\mathcal{T})$ and $r_v(\mathcal{T})$ be the length of the longest edge of $\mathcal{T}$ incident to u and v, respectively, without taking into account the edge uv. When it is clear which tree $\mathcal{T}$ we refer to, we will use r_u and r_v.

Lemma 6.4.6 *Let $(A_1, B_1), \dots, (A_\ell, B_\ell)$ be a WSPD of P w.r.t. τ, and let p and q be any two points in P. Then there is a pair (A_i, B_i) such that for every point $u \in A_i$ and every point $v \in B_i$ the inequality $\mathrm{dia}\, \S_{uv} \leq (1 + 8/\tau) \cdot \mathrm{dia}\, \S_{pq}$ holds.*

Proof: According to Definition 6.4.5 there is a pair (A_i, B_i) in the WSPD such that $p \in A_i$ and $q \in B_i$. If u is any point in A_i and v is any point in B_i, then let $\mathcal{T}$

be the tree with poles u and v where u is connected to v, p and each neighbor of p in $\S_{pq}$ except q is connected to u, and q and each neighbor of q in $\S_{pq}$ except p is connected to v. By Lemma 6.4.4(ii) $|uv| \leq (1+4/\tau)|pq|$ and by Lemma 6.4.4(iii) $r_u \leq |up| + r_p \leq 2|pq|/\tau + r_p$. Since $\operatorname{dia} \mathcal{T} = r_u + |uv| + r_v$ we have

$$\operatorname{dia} \mathcal{T} \leq \left(r_p + 2\frac{|pq|}{\tau}\right) + \left(|pq| + 4\frac{|pq|}{\tau}\right) + \left(r_q + 2\frac{|pq|}{\tau}\right) < \left(1 + \frac{8}{\tau}\right) \operatorname{dia} \S_{pq}.$$

The lemma follows due to the minimality of $\S_{uv}$. □

A first algorithm is now obvious. For each of the $O(\tau^2 n)$ pairs (A_i, B_i) in a WSPD of P w.r.t. $\tau = 8E$ pick *any* point $p \in A_i$ and *any* point $q \in B_i$, sort P according to distance from p, and compute $\S_{pq}$ in linear time by checking every possible radius of a disk centered at p as in [Ho91].

Lemma 6.4.7 *A dipolar tree $\mathcal{T}$ with $\operatorname{dia} \mathcal{T} \leqslant (1 + 1/E)\cdot d_{\min}$ can be computed in $O(E^2 n^2 \log n)$ time using $O(E^2 n + n \log n)$ space.*

6.4.4 A fast approximation scheme

Now we describe a more involved algorithm. It is asymptotically faster than the previous algorithm if $n = \Omega(E)$ (more precisely if $E = o(n \log n)$). We will prove its correctness in Section 6.4.5.

Theorem 6.4.8 *A dipolar tree $\mathcal{T}$ with $\operatorname{dia} \mathcal{T} \leq (1+1/E) \cdot d_{\min}$ can be computed in $O(E^3 n + En \log n)$ time using $O(E^2 n + n \log n)$ space.*

The idea of the algorithm is again to check only a linear number of pairs of points, using the WSPD, in order to speed up the computation of the disks around the two poles. Note that we need to find a close approximation of the diameters of the disks to be able to guarantee a $(1+\varepsilon)$-approximation of the MDdST. Obviously we cannot afford to try all possible disks for all possible pairs of poles. Instead of checking the disks we will show in the analysis that it suffices to check a constant number of ways to partition the input point set into two subsets, each corresponding to a pole. The partitions we consider are induced by a constant number of lines that are approximately orthogonal to the line through the poles. We cannot afford to do this for each possible pair. Instead we select a constant number of orientations and use a constant number of orthogonal cuts for each orientation. For each cut we calculate for each point in P the approximate distance to the farthest point on each side of the cut. Below we give a more detailed description of the algorithm. For its pseudocode refer to Algorithm 4.

Phase 1: Initializing. Choose an auxiliary positive constant $\kappa < \min\{0.9\varepsilon, 1/2\}$. As will be clear later, this parameter can be used to fine-tune which part of the algorithm contributes how much to the uncertainty and to the running time. In phase 3 the choice of the separation constant τ will depend on the value of κ and ε.

Algorithm 4 Approx-MDdST(P, ε)

```
Phase 1: initializing
 1: choose $\kappa \in (0, \min\{0.9\varepsilon, 1/2\})$; set $\gamma \leftarrow \lceil 4/\kappa \rceil$
 2: for $i \leftarrow 1$ to $\gamma$ do
 3:   $l_i \leftarrow$ line with angle $i\pi/\gamma$ to the horizontal
 4:   $F_i \leftarrow l_i$-ordering of $P$
 5: end for $i$
 6: for $i \leftarrow$ to $\gamma$ do
 7:   rotate $P$ and $l_i$ such that $l_i$ is horizontal
 8:   let $p_1, \dots, p_n$ be the points in $F_i$ from left to right
 9:   $d_i \leftarrow |p_1.x - p_n.x|$
10:   for $j \leftarrow 1$ to $\gamma$ do
11:     $b_{ij} \leftarrow$ marker on $l_i$ at distance $jd_i/(\gamma+1)$ to the right of $p_1$
12:     for $k \leftarrow 1$ to $\gamma$ do
13:       $L'_{ijk} \leftarrow l_k$-ordered subset of $F_k$ to the left of $b_{ij}$
14:       $R'_{ijk} \leftarrow l_k$-ordered subset of $F_k$ to the right of $b_{ij}$
15:     end for $k$
16:   end for $j$
17: end for $i$
Phase 2: computing approximate farthest neighbors
18: for $i \leftarrow 1$ to $\gamma$ do
19:   for $j \leftarrow 1$ to $\gamma$ do
20:     for $k \leftarrow 1$ to $n$ do
21:       $N(p_k, i, j, L) \leftarrow p_k$ {dummy}
22:       for $l \leftarrow 1$ to $\gamma$ do
23:         $p_{\min} \leftarrow$ first point in $L'_{ijl}$; $p_{\max} \leftarrow$ last point in $L'_{ijl}$
24:         $N(p_k, i, j, L) \leftarrow$ point in $\{p_{\min}, p_{\max}, N(p_k, i, j, L)\}$ furthest from $p_k$
25:       end for $l$
26:     end for $k$
27:     repeat lines 20–26 with $R$ instead of $L$
28:   end for $j$
29: end for $i$
Phase 3: testing pole candidates
30: $\tau = 8(\frac{1+\varepsilon}{(1+\varepsilon-(1+\kappa)(1+\kappa/24)} - 1)$
31: build WSPD for $P$ with separation constant $\tau$
32: $d \leftarrow \infty$ {smallest diameter so far}
33: for each pair $(A, B)$ in WSPD do
34:   choose any two points $u \in A$ and $v \in B$
35:   find $l_i$ with the smallest angle to the line through $u$ and $v$
36:   $D \leftarrow \infty$ {approximate diameter of tree with poles $u$ and $v$, ignoring $|uv|$}
37:   for $j \leftarrow 1$ to $\gamma$ do
38:     $D \leftarrow$
39:       $\min\{D, |N(u,i,j,L)u| + |vN(v,i,j,R)|, |N(u,i,j,R)u| + |vN(v,i,j,L)|\}$
40:   end for $j$
41:   if $D + |uv| < d$ then $u' \leftarrow u$; $v' \leftarrow v$; $d \leftarrow D + |uv|$ end if
42: end for $(A, B)$
43: compute $\mathcal{T} \leftarrow \S_{u'v'}$
44: return $\mathcal{T}$
```

Definition 6.4.9 *A set of points P is said to be l-ordered if the points are ordered with respect to their orthogonal projection onto the line l.*

Let l_i be the line with angle $i\pi/\gamma$ to the horizontal line, where $\gamma = \lceil 4/\kappa \rceil$. This implies that for an arbitrary line l there exists a line l_i such that $\angle l_i l \leq \pi/(2\gamma)$. For $i = 1, \ldots, \gamma$, let F_i be a list of the input points sorted according to the l_i-ordering. The time to construct these lists is $O(\gamma n \log n)$.

For each l_i, rotate P and l_i such that l_i is horizontal. For simplicity we denote the points in P from left to right on l_i by $p_1, \ldots, p_n$. Let d_i denote the horizontal distance between p_1 and p_n. Let b_{ij}, $1 \leq j \leq \gamma$, be a marker on l_i at distance $jd_i/(\gamma+1)$ to the right of p_1. Let L_{ij} and R_{ij} be the set of points in P to the left and to the right of the vertical β_{ij} through b_{ij}, respectively.

For each marker b_{ij} on l_i we construct γ pairs of lists, denoted L'_{ijk} and R'_{ijk}, where $1 \leq k \leq \gamma$. The list L'_{ijk} (R'_{ijk}) contains the points in L_{ij} (R_{ij}, respectively) sorted according to the l_k-ordering. Such a list can be constructed in $O(n)$ time since the ordering is given by F_k: we just have to filter out the points in F_k that are on the "wrong" side of β_{ij}. (Actually it is not necessary to store the whole lists L'_{ijk} and R'_{ijk}: we only need to store the first and the last point in each list.) Hence the total time complexity needed to construct the lists is $O(\gamma^3 n + \gamma n \log n)$, see lines 1–17 in Algorithm 4. These lists will help us to compute an approximate farthest neighbor in L_{ij} and R_{ij} for each point $p \in P$ in time $O(\gamma)$, as we describe below.

Phase 2: Computing approximate farthest neighbors. Let the approximate distance of a point q from p be the maximum distance among all projections of q onto the lines l_k. Now let the approximate farthest neighbor $N(p,i,j,L)$ of p be the point $q \in L_{ij}$ with maximum approximate distance from p. Each $N(p,i,j,L)$ can be computed in time $O(\gamma)$ by taking the farthest point from p over all first and last elements of L'_{ijk} with $k = 1, \ldots, \gamma$. Define and compute $N(p,i,j,R)$ analogously. Hence the total time complexity of phase 2 is $O(\gamma^3 n)$, as there are $O(\gamma^2 n)$ triples of type (p,i,j). The error we make by using approximate farthest neighbors is small:

Lemma 6.4.10 *If p is any point in P, p_L the point in L_{ij} farthest from p and p_R the point in R_{ij} farthest from p, then*

(a) $|pp_L| \leqslant (1+\kappa/24) \cdot |pN(p,i,j,L)|$ *and*

(b) $|pp_R| \leqslant (1+\kappa/24) \cdot |pN(p,i,j,R)|$.

Proof: Due to symmetry it suffices to check (a). If the algorithm did not select p_L as farthest neighbor it holds that for each of the l_k-orderings there is a point farther from p than p_L. Hence p_L must lie within a symmetric 2γ-gon whose edges are at distance $|pN(p,i,j,L)|$ from p. This implies that $|pp_L| \leq |pN(p,i,j,L)|/\cos(\pi/(2\gamma)) \leq |pN(p,i,j,L)|/\cos(\pi\kappa/8)$ using $\gamma = \lceil 4/\kappa \rceil$. Thus it remains to show that $1/\cos(\pi\kappa/8) \leq 1 + \kappa/24$. Since $\cos x \geq 1 - x^2/2$ for any x, the claim is true if $1 - \pi^2\kappa^2/128 \geq 1/(1+\kappa/24)$. This inequality holds for all $0 < \kappa \leq 1/2$. □

Phase 3: Testing pole candidates. Compute the WSPD for P with separation constant τ. To be able to guarantee a $(1+\varepsilon)$-approximation algorithm the value of τ will depend on ε and κ as follows:

$$\tau = 8\left(\frac{1+\varepsilon}{1+\varepsilon-(1+\kappa)(1+\kappa/24)} - 1\right).$$

Note that the above formula implies that there is a trade-off between the values τ and κ, which can be used to fine-tune which part of the algorithm contributes how much to the uncertainty and to the running time. Setting for instance κ to 0.9ε yields, for ε small, $16/\varepsilon + 15 < \tau/8 < 32/\varepsilon + 31$, i.e. $\tau = \Theta(1/\varepsilon)$. For each pair (A, B) in the decomposition we select two arbitrary points $u \in A$ and $v \in B$. Let $l_{(u,v)}$ be the line through u and v. Find the line l_i that minimizes the angle between l_i and $l_{(u,v)}$. That is, the line l_i is a close approximation of the direction of the line through u and v. From above we have that l_i is divided into $\gamma + 1$ intervals of length $d_i/(\gamma + 1)$. For each j, $1 \leq j \leq \gamma$, compute $\min(|N(u, i, j, L)u| + |vN(v, i, j, R)|, |N(u, i, j, R)u| + |vN(v, i, j, L)|)$. The smallest of these $O(\gamma)$ values is saved, and is a close approximation of $\mathrm{dia}\ \S_{uv} - |uv|$, which will be shown below.

The number of pairs in the WSPD is $O(\tau^2 n)$, which implies that the total running time of the central loop of this phase (lines 33–41 in Algorithm 4) is $O(\gamma \cdot \tau^2 n)$. Building the WSPD and computing $\S_{u'v'}$ takes an extra $O(\tau^2 n + n \log n)$ time. Thus the whole algorithm runs in $O(\gamma^3 n + \gamma\tau^2 n + \gamma n \log n)$ time and uses $O(n \log n + \gamma^2 n + \tau^2 n)$ space. Setting $\kappa = 0.9\varepsilon$ yields $\gamma = O(E)$ and $\tau = O(E)$ and thus the time and space complexities we claimed.

6.4.5 The proof of correctness for Theorem 6.4.8

It remains to prove that the diameter of the dipolar tree that we compute is indeed at most $(1+\varepsilon)\ d_{\min}$.

From Lemma 6.4.6 we know that we will test a pair of poles u and v for which $\mathrm{dia}\ \S_{uv} \leq (1 + 8/\tau)\ d_{\min} = \frac{1+\varepsilon}{(1+\kappa)(1+\kappa/24)}\ d_{\min}$. The equality actually explains our choice of τ. In this section we will prove that our algorithm always computes a dipolar tree whose diameter is at most $(1+\kappa)(1+\kappa/24)\ \mathrm{dia}\ \S_{uv}$ and thus at most $(1+\varepsilon)\ d_{\min}$.

Consider the tree $\S_{uv}$. For simplicity we rotate P such that the line l through u and v is horizontal and u lies to the left of v, as illustrated in Figure 6.12a. Let $\delta = |uv|$. Our aim is to prove that there exists an orthogonal cut that splits the point set P into two sets such that the tree obtained by connecting u to all points to the left of the cut and connecting v to all points to the right of the cut will give a tree whose diameter is a $(1+\kappa)$-approximation of $\mathrm{dia}\ \S_{uv}$. Since the error introduced by approximating the farthest neighbor distances is not more than a factor of $(1 + \kappa/24)$ according to Lemma 6.4.10, this will prove the claim in the previous paragraph.

Denote by C_u and C_κ the circles with center at u and with radius r_u and $r_\kappa = r_u + \kappa z$ respectively, where $z = \mathrm{dia}\ \S_{uv} = \delta + r_u + r_v$. Denote by C_v the

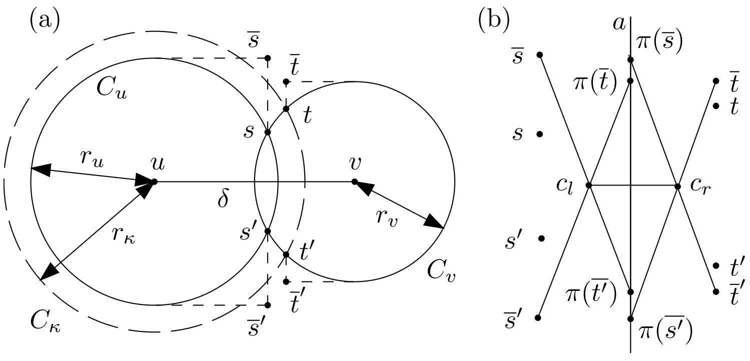


Figure 6.12: A valid cut.

circle with center at v and with radius r_v. Let s and s' (t and t') be two points on C_u (C_v) such that if C_u (C_κ) and C_v intersect, then s and s' (t and t') are the two intersection points, where s (t) lies above s' (t', respectively). Otherwise, if C_u (C_κ) and C_v do not intersect, then $s = s'$ ($t = t'$) is the intersection of the line segment (u, v) and C_u (C_v, respectively), see Figure 6.12a.

We say that a cut with a line l_κ is *valid* iff all points in P to the left of l_κ are contained in C_κ and all points of P to the right of l_κ are contained in C_v. A valid cut guarantees a dipolar tree whose diameter is at most $\delta + r_\kappa + r_v = (1+\kappa)\cdot \text{dia}\,\S_{uv}$.

We will prove that the algorithm above always considers a valid cut. For simplicity we assume that $r_u(\S_{uv}) \geq r_v(\S_{uv})$. We will show that there always exists a marker b_{ij} on l_i such that cutting l_i orthogonally through b_{ij} is valid. Actually it is enough to show that the two requirements below are valid for any $\S_{uv}$. For a point p, denote the x-coordinate and the y-coordinate of p by $p.x$ and $p.y$, respectively. For simplicity we set $u = (0, 0)$. We have

$$\text{(i)} \qquad \frac{z}{\gamma+1}\cdot\frac{1}{\cos\frac{\pi}{2\gamma}} \leq \frac{1}{2}(t.x - s.x), \quad \text{and}$$

$$\text{(ii)} \qquad \tan\frac{\pi}{2\gamma} \leq \frac{t.x - s.x}{2(r_u(\S_{uv}) + r_v(\S_{uv}))}.$$

The reason for this will now be explained. First we need to define some additional points. The reader is encouraged to study Figure 6.12 for a visual description. Let $\overline{s} = (s.x, r_u)$, $\overline{s}' = (s'.x, -r_u)$, $\overline{t} = (t.x, r_v)$ and $\overline{t}' = (t'.x, -r_v)$. Let a be the perpendicular bisector of the projections of s and t on the x-axis and let π be the orthogonal projection of the plane on a. Now we can define c_l to be the intersection point of the lines $(\overline{s}, \pi(\overline{t}'))$ and $(\overline{s}', \pi(\overline{t}))$, and c_r to be the intersection point of the lines $(\overline{t}, \pi(\overline{s}'))$ and $(\overline{t}', \pi(\overline{s}))$.

It now follows that any bisector l' that intersects the three line segments $(\overline{s}, \overline{t})$, (c_l, c_r) and $(\overline{s}', \overline{t}')$, will be a valid cut. This follows since all points to the left of

l' will be connected to u and all points to the right of l' will be connected to v, and the diameter of that tree will, obviously, be bounded by $\delta + (r_u(\S_{uv}) + \kappa z) + r_v(\S_{uv})$ which is a $(1+\kappa)$-approximation of $\mathrm{dia}\,\S_{uv}$.

From the algorithm we know that (a) there is a line l_i such that $\angle(l_i, l_{(u,v)}) \leq \pi/(2\gamma)$, and that (b) there are γ orthogonal cuts of l_i that define equally many partitions of P. The distance between two adjacent orthogonal cuts of l_i is at most $z/(\gamma+1)$. This implies that the length of the largest interval on $l_{(u,v)}$ that is not intersected by any of these orthogonal cuts is at most

$$\frac{1}{\cos\frac{\pi}{2\gamma}} \cdot \frac{z}{\gamma+1}.$$

Hence requirement (i) ensures that for every $\S_{uv}$ the distance $|c_l c_r| = (t.x - s.x)/2$ must be large enough to guarantee that there is an orthogonal cut of l_i that intersects it.

An orthogonal cut of l_i has an angle of at least $\pi/2 - \pi/(2\gamma)$ to $l_{(u,v)}$. To ensure that an orthogonal cut of l_i that intersects the line segment $\overline{c_l c_r}$ also passes between $\overline{s}$ and $\overline{t}$ and between $\overline{s}'$ and $\overline{t}'$ it suffices to add requirement (ii).

It remains to prove the following lemma which implies that for every $\S_{uv}$ there is a valid orthogonal cut.

Lemma 6.4.11 *For any $u, v \in P$ ($u \neq v$) the tree $\S_{uv}$ fulfills requirements (i) and (ii).*

Proof: The tree $\S_{uv}$ can be characterized by the relationship of the two ratios

$$\alpha := \frac{\delta}{r_u + r_v} \quad \text{and} \quad F := \frac{1+\kappa/2}{1-\kappa/2}.$$

We distinguish three cases: (1) $\alpha < 1$, (2) $1 \leq \alpha \leq F$, and (3) $\alpha > F$. For each of these three cases we will show that $\S_{uv}$ fulfills the two requirements.

Case 1: Using the following two straight-forward equalities, $s.x^2 + s.y^2 = r_u^2$ and $(\delta - s.x)^2 + s.y^2 = r_v^2$, we obtain that $s.x = (\delta^2 + r_u^2 - r_v^2)/(2\delta)$. A similar calculation for $t.x$ yields $t.x = (\delta^2 + r_\kappa^2 - r_v^2)/(2\delta)$. Inserting these values gives $t.x - s.x = (\kappa^2 z^2 + 2\kappa z r_u)/(2\delta)$. The fact that $\alpha \leq F$ allows us to further simplify the expression for $t.x - s.x$ by using the following two expressions:

$$\frac{z}{\delta} = \frac{\delta + r_u + r_v}{\delta} = 1 + \frac{r_u + r_v}{\delta} \geqslant \frac{2}{1+\kappa/2}, \quad \text{and} \quad \frac{r_u}{\delta} \geq \frac{1-\kappa/2}{2(1+\kappa/2)}.$$

From this we obtain that

$$t.x - s.x = \frac{\kappa z}{2}\left(\frac{\kappa z}{\delta} + \frac{2r_u}{\delta}\right) > \frac{\kappa z}{2}.$$

This fulfills requirement (i) since

$$\frac{z}{\gamma+1} \cdot \frac{1}{\cos\frac{\pi}{2\gamma}} \leqslant \frac{\kappa z}{4} \leqslant \frac{1}{2}(t.x - s.x). \tag{6.1}$$

For requirement (ii) note that $\tan \pi/(2\gamma) \leq 2\kappa \tan \pi/16 < 2\kappa/5$. Since $\kappa \leq 1/2$ we get that $z/\delta \geqslant 2/(1+\kappa/2) \geq 8/5$. Combining this inequality, Equality 6.1, and our assumption that $r_u \geqslant r_v$ shows that requirement (ii) is also fulfilled:

$$\frac{t.x - s.x}{2(r_u + r_v)} \geqslant \frac{\kappa z}{4\delta} \left(\frac{2r_u + \kappa z}{r_u + r_v} \right) \geqslant \frac{\kappa z}{4\delta} \geqslant \frac{2\kappa}{5}.$$

Case 2: In this case we argue in the same manner as in the previous case. Using the fact that $s.x = r_u$ and $t.x = (\delta^2 + r_\kappa^2 - r_v^2)/(2\delta)$ yields

$$t.x - s.x \geqslant \frac{\kappa z}{2} \left(\frac{\kappa z}{\delta} + \frac{2r_u}{\delta} \right) > \frac{\kappa z}{2}.$$

The rest of the proof is exactly as in case 1.

Case 3: The first requirement is already shown to be fulfilled since $t.x - s.x \geqslant \delta - r_u - r_v \geqslant \kappa z/2$, hence it remains to show requirement (ii). We have

$$\frac{t.x - s.x}{2(r_u + r_v)} \geqslant \frac{\delta - (r_u + r_v)}{2(r_u + r_v)}$$

plugging in the values gives $\kappa/(2-\kappa)$, which is at least $2\kappa/5$. The lemma follows. □

The lemma says that for every dipole $\{u, v\}$ there exists a line a such that the dipolar tree obtained by connecting all the points on one side of a to u and all the points on the opposite side to v, is a $(1+\kappa)$-approximation of $\S_{uv}$.

6.4.6 Putting things together

Combining grid- and WSPD-based approach yields a strong LTAS of order 5:

Theorem 6.4.12 *A spanning tree $\mathcal{T}$ of P with* $\operatorname{dia} \mathcal{T} \leq (1 + 1/E)\ d_P$ *can be computed in $O^*(E^5 + n)$ time using $O(E^4 + n)$ space.*

Proof: Applying Algorithm 4 to the set $R \subseteq P$ of the $O(E^2)$ representative points takes $O(E^3|R| + E|R|\log|R|)$ time using $O(E^2|R| + |R|\log|R|)$ space according to Theorem 6.4.8. Connecting the points in $P \setminus R$ to the poles adjacent to their representative points yields a $(1+\varepsilon)$-approximation of the MDdST of P within the claimed time and space bounds as in Section 6.4.1. The difference is that now the grid cells must be slightly smaller in order to compensate for the fact that we now approximate the MDdST of R rather than compute it exactly. A $(1+\varepsilon)$-approximation of the MDmST of P can be computed via the grid and an exact algorithm of Ho et al. [Ho91] in $O^*(E^2 + n)$ time using $O(E^2 + n)$ space. Of the two trees the one with smaller diameter is a $(1+\varepsilon)$-approximation of the MDST of P. □

6.5 Conclusions

On the one hand we have presented a new planar facility location problem, the discrete minimum-sum two-center problem that mediates between the discrete two-center problem and the minimum-diameter dipolar spanning tree. We have shown that there is an algorithm that computes the corresponding MSST in $O(n^2 \log n)$ time and that a variant of this tree is a factor-4/3 approximation of the MDST. It would be interesting to know whether there is a near quadratic-time algorithm for the MSST that uses $o(n^2)$ space.

On the other hand we have given four approximation schemes for the MDST. The asymptotically fastest is a combination of a grid-based approach with an algorithm that uses the well-separated pair decomposition. It computes in $O^*(\varepsilon^{-5}+n)$ time a tree whose diameter is at most $(1+\varepsilon)$ times that of a MDST. Such an algorithm is called a strong linear-time approximation scheme of order 5. Spriggs et al. [Spr03] recently improved our result by giving a strong LTAS of order 3 whose space consumption is linear in n and does not depend on ε. Is order 3 optimal? Is there an exact algorithm that is faster than Chan's [Chn02]? Is there a non-trivial lower bound on the computation time needed for the exact MDST?

Our scheme also works for higher-dimensional point sets, but the running time increases exponentially with the dimension. Linear-time approximation schemes for the discrete two-center problem and the MSST can be constructed similarly.

Acknowledgments

We thank an anonymous referee of an earlier version of this paper for suggesting Theorem 6.4.2. We also thank Pankaj K. Agarwal for pointing us to [Chn02] and Timothy Chan for sending us an updated version of [Chn02]. We thank another anonymous referee for pointing us to an error in the proof of Lemma 6.2.6. We finally thank Pat Morin for contributing the tree data structure in the proof of Lemma 6.2.6 that we needed to correct our error.

Chapter 7

Optimal spanners for axis-aligned rectangles

A brief abstract of this chapter appeared as: Tetsuo Asano, Mark de Berg, Otfried Cheong, Hazel Everett, Herman Haverkort, Naoki Katoh and Alexander Wolff: Optimal spanners for axis-aligned rectangles, in *Proc. 20th European Workshop on Computational Geometry (EWCG)*, Sevilla, 2004, pages 97–100. Full text submitted to *Computational Geometry – Theory and Applications*.

Abstract. *The dilation of a geometric graph is the maximum, over all pairs of points in the graph, of the ratio of the Euclidean length of the shortest path between them in the graph and their Euclidean distance. We consider a generalized version of this notion, where the nodes of the graph are not points but axis-parallel rectangles in the plane. The arcs in the graph are horizontal or vertical segments connecting a pair of rectangles, and the distance measure we use is the L_1-distance. The dilation of a pair of points is then defined as the length of the shortest rectilinear path between them that stays within the union of the rectangles and the connecting segments, divided by their L_1-distance. The dilation of the graph is the maximum dilation over all pairs of points in the union of the rectangles.*

We study the following problem: given n non-intersecting rectangles and a graph describing which pairs of rectangles are to be connected, we wish to place the connecting segments such that the dilation is minimized. We obtain four results on this problem: (i) for arbitrary graphs, the problem is NP-hard; (ii) for trees, we can solve the problem by linear programming on $O(n^2)$ variables and constraints; (iii) for paths, we can solve the problem in time $O(n^3 \log n)$; (iv) for rectangles sorted vertically along a path, the problem can be solved in $O(n^2)$ time, and a $(1+\varepsilon)$-approximation can be computed in linear time.

7.1 Introduction

Geometric networks arise frequently in our everyday life: road networks, telephone networks, and computer networks are all examples of geometric networks that we use daily. They also play a role in disciplines such as VLSI design and motion planning. Almost invariably, the purpose of the network is to provide a connection between the nodes in the network. Often it is desirable that the connection through the network between any pair of nodes be relatively short. From this viewpoint, one would ideally have a direct connection between any pair of nodes. This is usually infeasible due to the costs involved, so one has to compromise between the quality and the cost of the connections.

For two given nodes in a graph, the ratio of their distance in the graph and their 'direct' distance is called the *dilation* or *stretch factor* for that pair of nodes, and the dilation of a graph is the maximum dilation over all pairs of nodes. For geometric networks, this is more precisely defined as follows. Let S be a set of n points (in the plane, say), and let $\mathcal{G}$ be a graph with node set S. Now the dilation for a pair of points p, q is defined as the ratio of the length of the shortest path in $\mathcal{G}$ between p and q, and the length of the segment pq. (The length of a path is the sum of the lengths of its edges.) Again, the dilation of $\mathcal{G}$ is the maximum dilation over all pairs of points in S. A graph with dilation t is called a t-*spanner*. Ideal networks are t-spanners for small t with small cost.

Spanners were introduced by Peleg and Schäffer [Pel89] in the context of distributed computing, and by Chew [Che89] in the context of computational geometry. They have attracted much attention since—see for instance the survey by Eppstein [Epp00]. The cost of spanners can be measured according to various criteria. For example, it is sometimes defined as the number of edges (here the goal is to find a spanner with $O(n)$ edges), or as the total weight of the edges (here the goal is to find a spanner whose total weight is a constant times the weight of a minimum spanning tree). Additional properties, such as bounding the maximum degree or the diameter, have been considered as well.

We generalize the notion of spanners to geometric networks whose nodes are rectangles rather than points. Let S be a set of n non-intersecting, axis-parallel rectangles and let E be a set of axis-parallel segments connecting pairs of rectangles. For any two points p, q in the union of the rectangles, the dilation is now the ratio of the length of the shortest rectilinear path in the network between p and q and their L_1-distance. Here a path in the network is a path that stays within the union of the rectangles and the connecting segments. The dilation of the network is the maximum dilation over all pairs p, q. Again, our aim is to construct a network whose dilation is small. To illustrate the concept, imagine one is given a number of rectangular buildings, which have to be connected by footbridges. It is quite frustrating if, to walk to a room opposite ones own room in an adjacent building, one has to walk all the way to the end of a long corridor, then along the footbridge, and then back again along the corridor in the other building. Hence, one would usually place the footbridge in the middle between buildings. Following this analogy, we will call the rectangles in the input *buildings* from now on, and the connecting segments *bridges*. We call the underlying graph of the network the *bridge graph*.

The generalization we study introduces one important additional difficulty in the construction of a spanner: for points one only has to decide *which* edges to choose in the spanner, but for buildings, one also has to decide *where* to place the bridge between a given pair of buildings. It is the latter problem we focus on in this paper: we assume the topology of the network (the bridge graph) is given, and our only task is to place the bridges so as to minimize the dilation.

Formally, our problem can be stated as follows: we are given a set S of axis-parallel disjoint rectangles (buildings) in the plane, a graph $\mathcal{G}$ with node set S, and for each arc e of $\mathcal{G}$ a *bridge region* Λ_e, an axis-aligned rectangle connecting the two buildings. Buildings may degenerate to segments or points. The bridge graph $\mathcal{G}$ must only have arcs between buildings that can be connected by a horizontal or vertical segment, and may not have multiple edges or loops. The bridge regions must be disjoint from each other and the buildings. Our goal is to find a set of horizontal or vertical bridges lying in the bridge regions that has minimum dilation.

Figure 7.1 shows a bridge graph (the bridge regions are shaded) and a set of possible bridges. Note that the bridge regions Λ_2 and Λ_3 simply allow any bridge between the two buildings, but bridge region Λ_1 has been chosen so as to avoid intersecting s_3 or the bridge between s_2 and s_3.

Our results are as follows.

- In general, the problem is NP-hard.
- If the bridge graph is a tree, then the problem can be solved by a linear program with $O(n^2)$ variables and constraints.
- If the bridge graph is a path, then the problem can be solved in $O(n^3 \log n)$ time.
- If the bridge graph is a path and the buildings are sorted vertically along this path, the problem can be solved in time $O(n^2)$. A $(1 + \varepsilon)$-approximation can be computed in linear time.

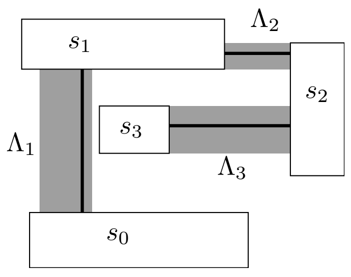


Figure 7.1: A bridge graph and a bridge configuration

7.2 The bridge graph is arbitrary

In this section we show that the bridge-placement problem is NP-hard if the bridge graph is allowed to be arbitrary. We prove this by a reduction from PARTITION. The input to PARTITION is a set B of n positive integers, and the task is to decide whether B can be partitioned into two subsets of equal sum. PARTITION is NP-hard [Gry79, Problem SP12].

Theorem 7.2.1 *It is NP-hard to decide whether the bridges in a given bridge graph on n rectangular buildings can be placed such that the dilation is at most 2.*

Let $B := \{\beta_0, \dots, \beta_{n-1}\}$ be an instance of PARTITION. For $0 \leqslant i < n$, we define $\alpha_i := \beta_i / (2 \sum_{0 \leqslant j < n} \beta_j)$. Note that $\sum_{0 \leqslant j < n} \alpha_j = 1/2$, and that B can be partitioned equally if and only if $\{\alpha_0, \dots, \alpha_{n-1}\}$ can be partitioned into two subsets of sum $1/4$. We create a bridge graph $\mathcal{G}(B)$ with $8n + 2$ buildings, as follows:

- for each $0 \leqslant i < n$, we have two point-shaped buildings, namely $P_i := (4i, 0)$ and $Q_i := (4i + 2 - 2\alpha_i, 0)$;
- for each $0 \leqslant i < n$, we have four segment-shaped buildings, namely $R_i := \{4i\} \times [1 - \alpha_i, 1]$ and $S_i := \{4i + 2 - 2\alpha_i\} \times [1 - \alpha_i, 1]$, and their mirrored images $R'_i := \{4i\} \times [-1, \alpha_i - 1]$ and $S'_i := \{4i + 2 - 2\alpha_i\} \times [-1, \alpha_i - 1]$;
- for each $0 < i < n$, we have two point-shaped buildings, namely $T_i := (4i - 1, 1)$ and $T'_i := (4i - 1, -1)$;
- we have two more point-shaped buildings $S_{-1} := (0, 2n + 3/4)$ and $S'_{-1} := (0, -2n - 3/4)$, and two more segment buildings $R_n := \{4n\} \times [1, 2n + 3/4]$ and $R'_n := \{4n\} \times [-2n - 3/4, -1]$.

The arcs in $\mathcal{G}(B)$ are as follows:

- for each $0 \leqslant i < n$, we have arcs (P_i, R_i), (P_i, R'_i), (Q_i, S_i), (Q_i, S'_i), (R_i, S_i), and (R'_i, S'_i);
- for each $0 < i < n$, we have arcs (S_{i-1}, T_i), (T_i, R_i), (S'_{i-1}, T'_i), (T'_i, R'_i), and (T_i, T'_i);
- we have arcs (S_{-1}, R_0), (S'_{-1}, R'_0), (S_{n-1}, R_n), (S'_{n-1}, R'_n).

Observe that (R_i, S_i) and (R'_i, S'_i) are the only bridges that can still be moved; all other bridges are fixed by the geometry. The construction is illustrated in Figure 7.2; the bridges to be placed are indicated as gray segments or rectangles. For the sake of clarity, we chose different scales on the x- and y-axis.

The reduction can clearly be done in polynomial time. The following lemma now implies the theorem.

Lemma 7.2.2 *The set B can be partitioned into two subsets of equal sum if and only if the bridges in $\mathcal{G}(B)$ can be placed such that the dilation is at most 2.*

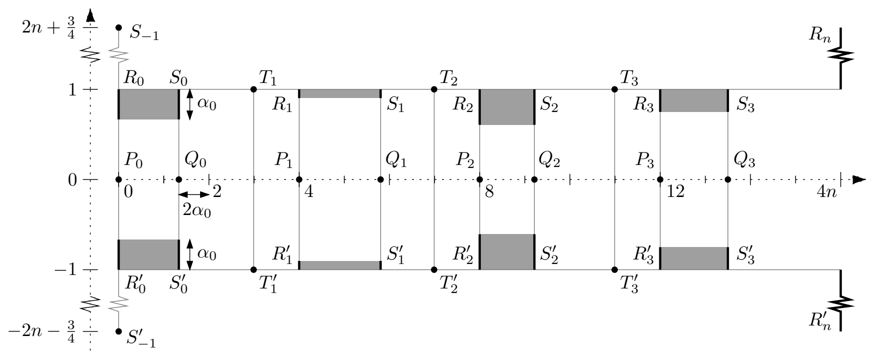

$2n+\frac{3}{4}$
1
0
-1
$-2n-\frac{3}{4}$
S_{-1}
S'_{-1}
R_0 S_0 T_1 R_1 S_1 T_2 R_2 S_2 T_3 R_3 S_3 R_n
α_0
P_0 Q_0 P_1 Q_1 P_2 Q_2 P_3 Q_3
0 2 4 8 12 $4n$
$2\alpha_0$
R'_0 S'_0 T'_1 R'_1 S'_1 T'_2 R'_2 S'_2 T'_3 R'_3 S'_3 R'_n

Figure 7.2: An instance of the bridge decision problem.

Proof: "If:" Suppose we can place the bridges in $\mathcal{G}(B)$ such that the dilation is at most 2. Then the dilation must be at most 2 for any pair (P_i, Q_i), which implies that either the bridge (R_i, S_i) must be placed in its bottommost position or (R'_i, S'_i) must be placed in its topmost position. Let I denote the set of indices for which the former holds, and I' the set of indices for which the latter holds.

Now consider S_{-1} and the top vertex of R_n. The L_1-distance between them is $4n$. The shortest path between them in $\mathcal{G}(B)$ cannot visit any P_i or Q_i, because the length of such a path would be at least $4n+2(2n+3/4)$ so its dilation would be larger than 2. Hence, the shortest path must visit $R_0, S_0, T_1, \dots, R_{n-1}, S_{n-1}$ in order from left to right. Any $i \in I$ induces an extra vertical distance $2\alpha_i$. Adding the vertical distance between S_{-1} and R_0 and along R_n, and the horizontal distance traversed, we get a total length of at least $\sum_{i\in I}(2\alpha_i) + 2(2n-1/4) + 4n$. Hence, $\sum_{i\in I}\alpha_i \leqslant 1/4$. A similar argument for S'_{-1} and the bottom vertex of R'_n shows that $\sum_{i\in I'}\alpha_i \leqslant 1/4$. It follows that I and I' induce an equal partition of B.

"Only if:" Suppose there is an equal partition of B. Then there are disjoint sets of indices I and I' with $I \cup I' = \{0, \dots, n-1\}$ such that $\sum_{i\in I}\alpha_i = \sum_{i\in I'}\alpha_i = 1/4$. For $i \in I$ place the bridges (R_i, S_i) and (R'_i, S'_i) in their bottommost position, and for $i \in I'$ place the bridges (R_i, S_i) and (R'_i, S'_i) in their topmost position.

Consider two points p, q, each lying on a building, with $p_x \leqslant q_x$. If $p_x = q_x$, then q can be reached without any detour. Otherwise, we distinguish two cases.

- The first case is that p or q (or both) have non-zero y-coordinate. Assume without loss of generality that $p_y > 0$ or that $p_y = 0$ and $q_y > 0$. Consider the path that goes up or down from p until reaching $y = 1$, then goes to the right while staying above the x-axis until the x-coordinate of q is reached, and then goes straight down or up to q.

If $p = S_{-1}$ and $q \in R_n$, then the length of the path is bounded by

$$4n + \sum_{i \in I} (2\alpha_i) + 2(2n - 1/4) = 8n.$$

Since $|p_x - q_x| = 4n$, the dilation is at most 2.

If $p \neq S_{-1}$ or $q \notin R_n$, the length of the path is bounded by

$$|p_x - q_x| + 2\sum_{i \in I} \alpha_i + |1 - p_y| + |1 - q_y| = |p_x - q_x| + 1/2 + |1 - p_y| + |1 - q_y|.$$

If p_y and q_y are not both $\leqslant 1$, then $|1 - p_y| + |1 - q_y| = |p_y - q_y|$, otherwise, $|1 - p_y| + |1 - q_y| = |p_y - q_y| + 2|1 - \max(p_y, q_y)| \leqslant |p_y - q_y| + 1/2$. In both cases the length of the path is at most $|p_x - q_x| + |p_y - q_y| + 1$, and from $|p_x - q_x| \geqslant 1$ it follows that the dilation is at most 2.

- The second case is that $p_y = q_y = 0$. Now the vertical distance traversed by the shortest path is at most $2 + \sum_{i \in I} (2\alpha_i) = 5/2$. Hence, if $|p_x - q_x| \geqslant 5/2$, the dilation is at most 2. But $|p_x - q_x| < 5/2$ implies that $p = P_i$ and $q = Q_i$ for some $0 \leqslant i < n$ or that $p = Q_i$ and $q = P_{i+1}$ for some $0 \leqslant i < n$. In the former case the dilation is 2 because either (R_i, S_i) is bottommost or (R'_i, S'_i) is topmost. In the latter case the dilation is less than 2 because the vertical distance traversed is exactly 2 and $|p_x - q_x| > 2$.

□

7.3 The bridge graph is a tree

In this section we will show that the bridge-placement problem can be solved by a linear program if the bridge graph is a tree. We start by introducing some terminology and notation, and by proving some basic lemmas.

As before, we denote the bridge graph by $\mathcal{G}$. Any set of bridges realizing $\mathcal{G}$ will be called a *configuration*.

Given a configuration B and two points p and q in the union of all buildings, we use $\pi(p, q, B)$ to denote the family of rectilinear shortest paths from p to q within the configuration (that is, paths whose links lie inside buildings or on bridges). The paths of this family are essentially the same, they differ only in how they connect two points inside the same building, and so we will simply speak about *the unique path* $\pi(p, q, B)$. The *dilation* of the path $\pi = \pi(p, q, B)$ is $\mathrm{dil}(\pi) := |\pi| / \|pq\|$, where $|\pi|$ is the total length of π and $\|pq\|$ is the L_1-distance of p and q. Figure 7.3 shows a configuration and an example path.

The *dilation* $\mathrm{dil}(B)$ *of a configuration* B is defined as the maximum dilation of any path with respect to B. Our aim is to find a configuration of minimum dilation. We first characterize pairs of points that are responsible for the dilation of a given configuration.

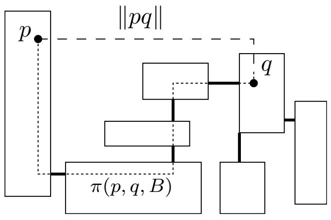


Figure 7.3

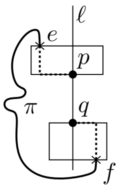


Figure 7.4

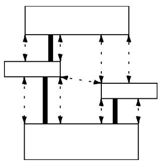


Figure 7.5

Lemma 7.3.1 *Let σ be the dilation of a configuration B whose underlying graph is a tree. Then there are points p and q with $\operatorname{dil}(\pi(p,q,B)) = \sigma$ such that the closed bounding box of p and q does not contain any point of a building other than p and q, and at least one of the points p and q is a building corner.*

Proof: Among all pairs of points (p,q) that have maximum dilation with respect to B, consider the subset of pairs where $\|pq\|$ is minimum. Choose a pair (p,q) from this subset where p is lexicographically smallest. Let β be the closed bounding box of p and q, and assume there is a point $r \in \beta$ distinct from p and q that belongs to a building. By our choice of (p,q), we have $|\pi(p,r,B)| < \sigma\|pr\|$ and $|\pi(r,q,B)| < \sigma\|rq\|$. Since $r \in \beta$ we have $\|pq\| = \|pr\| + \|rq\|$. Combining with the triangle inequality we obtain

$$|\pi(p,q,B)| \leqslant |\pi(p,r,B)| + |\pi(r,q,B)| < \\ \sigma\|pr\| + \sigma\|rq\| = \sigma\|pq\| = |\pi(p,q,B)|,$$

a contradiction, so no such point $r \in \beta$ exists.

It immediately follows that p and q are on the boundary of their buildings. It remains to prove that at least one of them is a building corner. Assume to the contrary that both are on the interior of a building edge. Then either p and q have the same x-coordinate and lie on the top and bottom edge of their buildings, or they have the same y-coordinate and lie on the left and right edge of their buildings. We discuss the first case, the second case is analogous. Clearly, moving both p and q the same distance to the left or right does not change $\|pq\|$. But what about $|\pi(p,q,B)|$? Let ℓ be the vertical line through p and q, and let e and f be the points where $\pi(p,q,B)$ leaves the building containing p and q, respectively. If e and f lie on opposite sides of ℓ as in Figure 7.4, we can move p and q slightly to the left without changing $\operatorname{dil}(\pi(p,q,B))$, a contradiction to the assumption that p is lexicographically smallest. It follows that e and f lie on the same side of ℓ (including ℓ itself), and so $|\pi(p,q,B)|$ increases if we move p and q into the opposite direction, a contradiction to the assumption that $\operatorname{dil}(\pi(p,q,B))$ is maximal. □

A point pair (p,q) as in the lemma—its bounding box contains no other point of any building and at least one of p and q is a building corner—will be called a *visible pair*—see Figure 7.5 for examples. We denote the set of all visible pairs

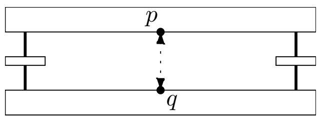


Figure 7.6

by $\mathcal{V}$. Note that the second statement of the lemma does not hold if there are cycles in the bridge graph—the maximum dilation may occur between two points in the interior of building edges, as in Figure 7.6.

Lemma 7.3.2 *For any set of n buildings, there are at most $O(n^2)$ visible pairs and they involve at most $12n$ points. These points can be computed in $O(n \log n)$ time.*

Proof: Clearly there are at most $O(n^2)$ visible pairs where both points are building corners. These pairs involve only the at most $4n$ building corners. Consider a visible pair (p, q) where only p is a building corner. Then q can be found by shooting a vertical or horizontal ray from p until it hits another building. It follows that for each building corner p there are at most two choices for q, so there are at most $8n$ such visible pairs, and at most $8n$ candidates for non-corner points that can be involved in a visible pair. They can be found in $O(n \log n)$ time by computing a vertical and a horizontal decomposition of the set of buildings [Brg97KOS]. $\square$

Lemmas 7.3.1 and 7.3.2 allow us to compute the dilation of a given configuration efficiently. The quadratic bound is tight: even if the bridge graph is a path, there can be $\Omega(n^2)$ visible pairs.

Given a bridge graph $\mathcal{G}$, our goal is to minimize

$$\max_{(p,q)\in\mathcal{V}} \operatorname{dil}(\pi(p, q, B))$$

over all configurations B realizing $\mathcal{G}$. We will now reformulate this problem as a linear program.

Theorem 7.3.3 *If the bridge graph $\mathcal{G}$ is a tree, then a placement of the bridges that minimizes the dilation can be computed by solving a linear program with $O(n^2)$ variables and constraints, where n is the number of bridges in the bridge graph.*

Proof: For each edge e of $\mathcal{G}$, we introduce a variable X_e specifying the position of the corresponding bridge; X_e is the x-coordinate of a vertical bridge or the y-coordinate of a horizontal bridge. We also introduce a variable Z. Our linear program will be such that a variable assignment is feasible if and only if the bridge assignment prescribed by the X_e is a configuration realizing $\mathcal{G}$ with dilation $\leqslant Z$. Minimizing Z will then solve the bridge-placement problem.

We will need a number of extra variables. We first define a set of points U by taking all points involved in a visible pair, as well as all bridge endpoints.

By Lemma 7.3.2, the size of U is $O(n)$. Some of the points in U are of the form $(\mathit{const}, \mathit{const})$ (namely the points in a visible pair), some are of the form (const, X_e) (the endpoints of a horizontal bridge), and some are of the form (X_e, const) (the endpoints of a vertical bridge). For each pair of points (u, v) from U that lie in the same building, we introduce an extra variable D_{uv}.

We can now describe the linear program. For each X_e, we need two simple constraints of the form $X_e \geqslant \mathit{const}$ and $X_e \leqslant \mathit{const}$, ensuring that the bridge indeed lies in the bridge region. For each D_{uv}, we add constraints enforcing $D_{uv} \geqslant \|uv\|$, as follows. Let $u = (x_u, y_u)$, $v = (x_v, y_v)$ (recall that each coordinate is either a constant, or one of the variables X_e, for some edge e). Then we add the constraints:

$$\begin{aligned} D_{uv} &\geqslant x_u - x_v + y_u - y_v \\ D_{uv} &\geqslant x_v - x_u + y_u - y_v \\ D_{uv} &\geqslant x_u - x_v + y_v - y_u \\ D_{uv} &\geqslant x_v - x_u + y_v - y_u \end{aligned}$$

Clearly, these four constraints together guarantee that $D_{uv} \geqslant \|uv\|$.

Finally, we introduce one constraint for each visible pair $(p, q) \in \mathcal{V}$. Let $bl(p, q)$ be the total length of all bridges traversed by $\pi(p, q, B)$. Since $\mathcal{G}$ is a tree, the buildings and bridges traversed by $\pi(p, q, B)$ are independent of the configuration, and so $bl(p, q)$ is a constant. We can now write

$$|\pi(p, q, B)| = bl(p, q) + \sum_{uv} \|uv\|,$$

where the sum is over the entry and exit points u and v of $\pi(p, q, B)$ for each building traversed. Note that $u, v \in U$, and u and v lie in the same building. We introduce the constraint

$$bl(p, q) + \sum_{uv} D_{uv} \leqslant Z \cdot \|pq\|.$$

We now argue that if a variable assignment is feasible in this linear program, then the bridge assignment prescribed by the X_e is a configuration realizing $\mathcal{G}$ with dilation $\leqslant Z$. Indeed, consider a visible pair (p, q). We have

$$|\pi(p, q, B)| = bl(p, q) + \sum_{uv} \|uv\| \leqslant bl(p, q) + \sum_{uv} D_{uv} \leqslant Z \cdot \|pq\|,$$

and so $\operatorname{dil}(\pi(p, q, B)) \leqslant Z$.

On the other hand, assume there is a configuration B realizing $\mathcal{G}$. Let X_e be the placement of the bridge e in B, let $D_{uv} = \|uv\|$, and let Z be the dilation of B. It is now easy to see that this variable assignment is feasible.

It follows that the bridge-placement problem can be solved by minimizing Z with respect to the linear programme described. The number of variables and constraints is $O(n^2)$. $\square$

7.4 The bridge graph is a path

In the previous section we have given a linear program for the bridge-placement problem for the case where the bridge graph is a tree. Linear programs can be solved in practice, and for integer coefficients, interior-point methods can solve them in time polynomial in the bit-complexity of the input [Kar84]. It is not known, however, if they can be solved in polynomial time on the real RAM, the standard model of computational geometry. In this section, we give polynomial time algorithms for the case where the bridge graph is a path.

Since the bridge graph $\mathcal{G}$ is a path, we can number the buildings and bridges so that bridge b_i connects buildings s_{i-1} and s_i, for $1 \leqslant i \leqslant n$ (so there are $n+1$ buildings and n bridges). Before we continue, we need to introduce some more terminology. We consider a path $\pi = \pi(p, q, B)$ to be oriented from p to q. After traversing a bridge b, the path can continue straight on to traverse the next bridge b' if b and b' are collinear. In all other cases, it has to turn.

Given a path π, a *link* ℓ of π is a maximal straight segment of the path. A link can contain more than one bridge if they are collinear. For example, in Figure 7.7 there is a link containing b_1 and b_2, and another link containing b_8, b_9, and b_{10}.

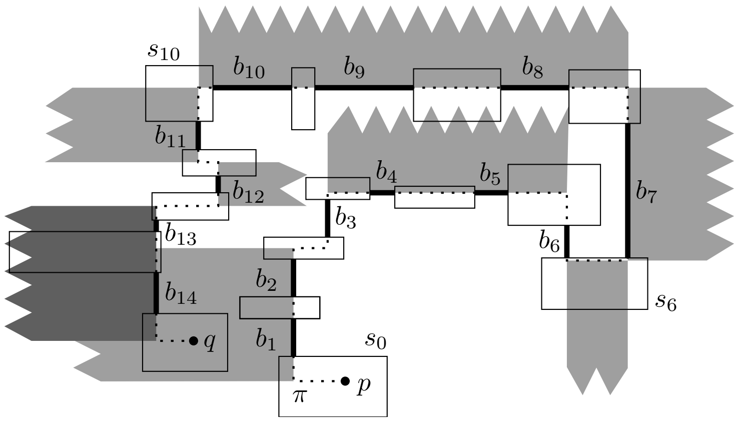


Figure 7.7: U-turns and their outer sides

The path π turns at both ends of a link (except for the first and last link). The link is a *right U-turn* if π turns right before and after the link. A *left U-turn* is defined symmetrically. In Figure 7.7, the links containing bridges (b_1, b_2), (b_4, b_5), and b_{12} are right U-turns, while the links containing b_7, (b_8, b_9, b_{10}), b_{11}, and (b_{13}, b_{14}) are left U-turns. Note that there can be U-turns that do not contain any bridges, as the link of π inside building s_6 in Figure 7.7.

The *inner side* and *outer side* of a U-turn are rectangular regions infinite on one side, and bounded by the line supporting the link and the two lines orthogonal to it through the first and last points of the link. The outer side lies locally to the left of a right U-turn, or to the right of a left U-turn, the inner side lies locally to the right of a right U-turn or to the left of a left U-turn. In Figure 7.7, the outer sides of all U-turns are shaded.

U-turns are the links of a path that determine its dilation, as the following lemma shows.

Lemma 7.4.1 *Let B and B' be configurations, (p, q) a visible pair, and $\pi := \pi(p, q, B)$ and $\pi' := \pi(p, q, B')$ the paths between p and q with respect to the two configurations. If $\operatorname{dil}(\pi') < \operatorname{dil}(\pi)$ then there exists a U-turn ℓ containing $b_i \dots b_j$ of π such that the corresponding bridges $b'_i, \dots, b'_j$ of B' lie strictly on the inner side of ℓ.*

Proof: For each U-turn ℓ of π, shade the outer side of ℓ, as in Figure 7.7. It is easy to see that π is a shortest rectilinear path from p to q that visits all the shaded regions in order. If the claim were not true, then π' would also visit all these regions in order, and so $|\pi'| \geqslant |\pi|$, a contradiction. $\square$

7.4.1 The decision problem

We will give an algorithm that takes as input the set of buildings $s_0, \dots, s_n$ and a real number $\sigma > 1$, and computes a configuration B with $\operatorname{dil}(B) \leqslant \sigma$, or determines that no such configuration exists.

The algorithm computes n sets $I_1, I_2, \dots, I_n$, where I_i is a set of possible bridges between s_{i-1} and s_i. The sets are defined recursively as follows. Assume that $I_1, \dots, I_{i-1}$ have already been defined. For each visible pair (p, q) with $p \in \bigcup_{j=0}^{i-1} s_j$ and $q \in s_i$ we define $I(p, q)$ as the set of bridges b_i connecting s_{i-1} and s_i such that the following holds: there is a set of bridges $b_1 \in I_1$, $b_2 \in I_2$, $\dots, b_{i-1} \in I_{i-1}$ such that $\operatorname{dil}(\pi(p, q, (b_1, \dots, b_i))) \leqslant \sigma$. Finally, I_i is the intersection of all $I(p, q)$.

Note that for each visible pair (p, q) we can choose the bridges in $I_1, \dots, I_{i-1}$ independently. This makes it possible to compute I_i efficiently, as we will see below. On the other hand, it implies that not every sequence of bridges chosen from the sets will be a configuration with dilation at most σ—our main lemma will be to show that such a sequence does indeed exist.

The opposite direction is nearly trivial: if a configuration with dilation at most σ exists, it can be found in the sets we constructed, as we show now.

Lemma 7.4.2 *Let $B = (b_1, b_2, \dots, b_n)$ be a configuration such that $b_i \notin I_i$ for some i. Then $\operatorname{dil}(B) > \sigma$.*

Proof: Let i be the smallest index with $b_i \notin I_i$. Since $b_i \notin I_i$, there exists a visible pair (p, q) with $p \in s_j$, $j < i$, and $q \in s_i$ such that for any set of bridges chosen from $I_1, \dots, I_{i-1}$ the path between p and q has dilation larger than σ. Since by our choice of i we have $b_k \in I_k$ for $k < i$, we have indeed $\operatorname{dil}(\pi(p, q, B)) > \sigma$. $\square$

We first argue that the sets I_i can be represented and managed easily.

Lemma 7.4.3 *Let $I_1, I_2, \ldots, I_n$ be defined as above. Then the x-coordinates (y-coordinates) of the bridges in each set form an interval.*

Proof: It is sufficient to show that the sets $I(p,q)$ are intervals. Consider a visible pair (p,q) with $p \in s_j$ and $q \in s_i$. Without loss of generality, assume the bridges in $I(p,q)$ to be vertical. Take three bridges a, b, c with x-coordinates $a_x < b_x < c_x$ and $a, c \in I(p,q)$. We will show that $b \in I(p,q)$.

Due to symmetry, we can assume $q_x \geqslant c_x$. Since $a \in I(p,q)$, a path $\pi = \pi(p,q,(b_1,\ldots,b_{i-1},a))$ exists (fat gray in Figure 7.8) with $\mathrm{dil}(\pi) \leq \sigma$ that uses bridges $b_1 \in I_1, \ldots, b_{i-1} \in I_{i-1}$. Now we can exchange the part of π from where π enters a to where π reaches q by a piece that uses b instead of a (dashed black in Figure 7.8). This new path is at most as long as π, which shows that $b \in I(p,q)$. □

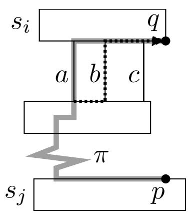


Figure 7.8: Proof of Lemma 7.4.3

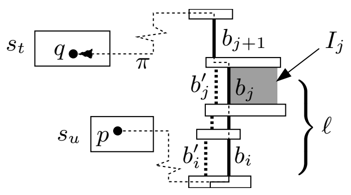


Figure 7.9: Proof of Lemma 7.4.4.

Once we know $I_1, \ldots, I_n$, we can recursively compute a configuration with dilation at most σ. Choose an arbitrary bridge $b_n \in I_n$. If bridges $b_{n-1}, b_{n-2}, \ldots, b_{i+1}$ have been computed, choose a bridge $b_i \in I_i$ whose distance from b_{i+1} is minimal. Since I_i is an “interval of bridges”, this implies that either b_i and b_{i+1} are collinear, or b_i is one of the extreme bridges in I_i. We now prove that this approach is correct.

Lemma 7.4.4 *Let $I_1, \ldots, I_n$ be given as defined above. A configuration B with dilation $\mathrm{dil}(B) \leqslant \sigma$ exists if and only if $I_n \neq \emptyset$. If it exists, it can be computed in $O(n)$ time from the intervals.*

Proof: The “only if” part follows from Lemma 7.4.2. We show the “if” part by proving that the configuration $B = (b_1, \ldots, b_n)$ defined above has dilation $\leqslant \sigma$. Since this configuration can clearly be computed in linear time from the intervals, the last statement of the lemma will follow at the same time.

Assume that $\mathrm{dil}(B) > \sigma$. Then there is a visible pair (p,q), such that the dilation $\mathrm{dil}(\pi(p,q,B)) > \sigma$. Let $\pi = \pi(p,q,B)$, and let s_u, s_t be the buildings containing p and q. Without loss of generality we can assume $u < t$. Since $b_t \in I_t$, there is a sequence of bridges $b'_1, \ldots, b'_{t-1}$ with $b'_k \in I_k$, such that the path $\pi' = \pi(p,q,(b'_1,\ldots,b'_{t-1},b_t))$ has dilation at most σ.

We have $\mathrm{dil}(\pi') \leqslant \sigma < \mathrm{dil}(\pi)$. By Lemma 7.4.1 there is a U-turn $\ell = (b_i, \ldots, b_j)$ of π (without loss of generality assumed to be a left U-turn) such that all the bridges $b'_i, \ldots, b'_j$ lie strictly to the left of ℓ, see Figure 7.9.

The last bridge of both π and π' is b_t, so $j < t$. It follows that π passes through b_{j+1}. Since ℓ is a left U-turn, the bridge b_{j+1} is strictly to the left of b_j. By definition of b_j, however, this implies that b_j is the left endpoint of I_j, and $b'_j \notin I_j$, a contradiction. $\square$

Given a point p in a building s_u, we can define a configuration B^p that is, in a sense, optimal for p by choosing bridges $b^p_1, \ldots, b^p_n$ as follows. For $k \leqslant u$, choose an arbitrary bridge $b^p_k \in I_k$. Choose bridge b^p_{u+1} as close as possible to p. The remaining bridges are chosen recursively, by choosing $b^p_k \in I_k$ to be as close to b^p_{k-1} as possible. Let m^p_i denote the endpoint of b^p_i on the building s_i. The following lemma shows that B^p is indeed optimal for p.

Lemma 7.4.5 *Let intervals $I_1, \ldots, I_n$ be as defined above, let $p \in s_u$ and $q \in s_t$, with $u < t$. Furthermore, let $B = (b_1, \ldots, b_n)$ be a configuration with $b_i \in I_i$ for $i < t$, and let B' be the configuration $(b_1, \ldots, b_u, b^p_{u+1}, \ldots, b^p_{t-1}, b_t, \ldots, b_n)$. Then $\mathrm{dil}(p, q, B') \leqslant \mathrm{dil}(p, q, B)$.*

Proof: Let $\pi = \pi(p, q, B)$, and $\pi' = \pi(p, q, B')$. Assume that $\mathrm{dil}(\pi') > \mathrm{dil}(\pi)$. By Lemma 7.4.1 there is then a U-turn $\ell = (b^p_i, \ldots, b^p_j)$ of π' (without loss of generality assumed to be a left U-turn) such that the corresponding bridges of π lie strictly to the left of ℓ. Since ℓ is a left U-turn, the bridge b^p_{i-1} (or the point p, if $i - 1 = u$) lies to the left of b^p_i. The definition of b^p_i implies that b^p_i is then the leftmost bridge in I_i, a contradiction with $b_i \in I_i$. $\square$

The following lemma shows that optimal paths are helpful in computing the intervals I_i.

Lemma 7.4.6 *Let $p \in s_u$, $q \in s_i$, with $u < i - 1$. The interval $I(p, q)$ can be computed in constant time if b^p_{i-1} and $|\pi(p, m^p_{i-1}, B^p)|$ are known.*

Proof: Recall that $I(p, q)$ is defined as the set of all bridges b_i connecting s_{i-1} and s_i, such that there is a set of bridges $b_1 \in I_1$, $b_2 \in I_2, \ldots, b_{i-1} \in I_{i-1}$ with $\mathrm{dil}(\pi(p, q, (b_1, \ldots, b_i))) \leqslant \sigma$. By Lemma 7.4.5 this is equivalent to $\mathrm{dil}(\pi(p, q, (b^p_1, b^p_2, \ldots, b^p_{i-1}, b_i))) \leqslant \sigma$. This path coincides with $\pi(p, q, B^p)$ up to and including bridge b^p_{i-1}, which is the path $\pi(p, m^p_{i-1}, B^p)$. Since the length of this path is known, we can compute $I(p, q)$ in constant time. $\square$

Lemma 7.4.7 *The intervals $I_1, \ldots, I_n$ defined above can be computed in $O(n^2)$ time and $O(n)$ space.*

Proof: Let P denote the set of all building corners and all points p such that there is a visible pair (p, q) with $p \in s_u$, $q \in s_t$, and $u < t$. By Lemma 7.3.2, P contains at most $12n$ points and it can be computed in $O(n \log n)$ time.

For each building s_t, we create a list of visible pairs (p, q) with $q \in s_t$ and $p \in \bigcup_{u=1}^{t-1} s_u$ such that not both p and q are building corners. This can be done during the same computation.

The computation then proceeds in n stages, with stage i computing interval I_i. Throughout, we maintain for each point $p \in P$ the bridge b_i^p, as well as the length of the path $\pi(p, m_i^p, B^p)$.

Consider stage i. We compute the intervals $I(p, q)$, for all pairs (p, q) with $p \in \bigcup_{u=0}^{i-1} s_u$ and $q \in s_i$ that are either visible pairs or where both p and q are building corners. (This avoids the need to precompute and store $O(n^2)$ visible pairs.) Note that all the points p appearing in such pairs are in P, and so there are at most $12n$ such pairs.

By Lemma 7.4.6, it takes constant time to compute $I(p, q)$ using the information from the previous stage. We can determine b_i^p and update the stored length for $\pi(p, m_i^p, B^p)$ in constant time as well.

It takes $O(n)$ time to compute the intersection interval I_i, so the total time spent per stage is $O(n)$. □

Lemmas 7.4.7 and 7.4.4 imply the following theorem.

Theorem 7.4.8 *Given a bridge graph $\mathcal{G}$ on a set of $n+1$ buildings that is a path and a real number $\sigma > 1$, we can in time $O(n^2)$ compute a configuration B realizing $\mathcal{G}$ with $\mathrm{dil}(B) \leqslant \sigma$ or determine that no such configuration exists.*

It seems hard to improve this result when there are $\Theta(n^2)$ visible pairs that could determine the dilation. In fact, we do not even know how to decide in $o(n^2)$ time whether a *given* configuration has dilation $\leqslant \sigma$.

If the number of visible pairs of the given set of buildings is $o(n^2/\log n)$, it is possible to do better. The difficulty is that the size of the set P is still linear, and we cannot maintain b_i^p for all points $p \in P$ explicitly. Instead, we store b_i^p and $|\pi(p, m_i^p, B^p)|$ in data structures that allow us to update them efficiently. We will need the simple data structure described in the following lemma.

Lemma 7.4.9 *There is a data structure that stores m real numbers $a_1, \ldots, a_m$, can be built in time $O(m)$, and supports the following operations in $O(\log m)$ time:*

- *given an index $j \in \{1, \ldots, m\}$, return a_j;*
- *given two indices $j', j'' \in \{1, \ldots, m\}$ and a real number b, replace the value of a_j by $a_j + b$ for all $j' \leqslant j \leqslant j''$.*

Proof: The data structure is basically a segment tree [Brg97KOS]. It is a balanced binary tree, whose leaves correspond to the indices $1, \ldots, m$ in order. Each node v of the tree contains a real number b_v, and the value of a_j for a leaf j is the sum of b_v over the nodes on the path from the root to j. Clearly it can be returned in time $O(\log m)$. For the last operation, we find all the nodes v of the tree whose descendents' indices are in the interval $[j', j'']$, but where this statement is not true for the parent, and add b to b_v. □

Let again Λ_i be the bridge region connecting s_{i-1} and s_i. Let b and b' be two bridges in Λ_i, and consider them directed from s_{i-1} to s_i. We let $b \prec b'$ if and only if b lies left of b'. Now let P be the set of points defined in Lemma 7.4.7, and let $P_i := P \cap \bigcup_{j=0}^{i} s_j$. Consider the union of all rectangles and all bridge regions. This is a single rectilinear polygon. We order the points of P along the boundary of this polygon, in counter-clockwise order starting and ending on s_n (note that there are no points of P in s_n) and denote this order again by $\prec$.

Lemma 7.4.10 *Let $p, p' \in P_{i-1}$. If $b_i^p \prec b_i^{p'}$ then $p \prec p'$.*

Proof: If $p' \prec p$ while $b_i^p \prec b_i^{p'}$, then the paths $\pi(p, m_i^p, B^p)$ and $\pi(p', m_i^{p'}, B^{p'})$ have to cross, which is impossible. □

Theorem 7.4.11 *Given a bridge graph $\mathcal{G}$ on a set of $n+1$ buildings that is a path, and a real number $\sigma > 1$, we can in time $O(k \log n)$ compute a configuration B realizing $\mathcal{G}$ with $\operatorname{dil}(B) \leqslant \sigma$ or determine that no such configuration exists, where k is the number of visible pairs.*

Proof: It is sufficient to show how to compute the intervals I_i. We start by computing all visible pairs. This can be done in time $O(k \log n)$ (note that $k \geqslant n$), by computing both vertical and horizontal decompositions [Brg97KOS], and a modified version of the algorithm for reporting all direct visibility pairs by de Berg et al. [Brg92]. For each building s_t we build a list of visible pairs (p, q) with $q \in s_t$ and $p \in P_{t-1}$.

The algorithm proceeds again in n stages, computing I_i in stage i. We maintain two data structures, $\mathcal{P}$ (paths) and $\mathcal{B}$ (bridges). $\mathcal{P}$ is the data structure of Lemma 7.4.9. It stores for each $p \in P$ a value a_p, with the points sorted by $\prec$. If $p \in s_u$, then $a_p = 0$ up to stage $u+1$, and $a_p = |\pi(p, m_{i-1}^p, B^p)|$ when stage $i \geqslant u+2$ is about to start. $\mathcal{B}$ is a dictionary. At the beginning of stage i, it stores all the bridges b_{i-1}^p, for $p \in P_{i-2}$, in the order $\prec$. A bridge shared by several points is only stored once. For each bridge b, we store the x- or y-coordinate, and two points $p', p'' \in P_{i-2}$ such that $b_{i-1}^p = b$ if and only if $p' \preceq p \preceq p''$. This is possible by Lemma 7.4.10.

In stage i, we retrieve the list of visible pairs (p, q) with $q \in s_i$. For each pair, we compute $I(p, q)$. If $p \in s_{i-1}$, this is done directly, in constant time. Otherwise $p \in P_{i-2}$, and we compute $I(p, q)$ from b_{i-1}^p and $|\pi(p, m_{i-1}^p, B^p)|$ in constant time by Lemma 7.4.6. We can find the bridge b_{i-1}^p in $O(\log n)$ time in $\mathcal{B}$—by Lemma 7.4.10 $\mathcal{B}$ is sorted by points as well as by bridges. The value $|\pi(p, m_{i-1}^p, B^p)|$ is stored in $\mathcal{P}$. It follows that the total time, over all stages, for this computation is $O(k \log n)$.

It remains to discuss the updating of $\mathcal{P}$ and $\mathcal{B}$ to prepare them for the next stage. Let's first discuss $\mathcal{B}$. Consider the interval I_{i-1}. The part of I_{i-1} that continues straight on into I_i doesn't need to be touched. The bridges b_{i-1}^p on the left or right of I_{i-1} that cannot continue straight on (all bridges, if the orientation of I_{i-1} and I_i is different) are removed, and replaced by bridges on the edge of I_i. In addition, we insert new bridges for all $p \in P \cap s_i$. This can be done in time $O(d \log n)$, where d is the number of bridges being removed and created. We can charge the cost of removing a bridge to its creation. Since the number of bridges created during the course of the algorithm is $|P| + 2n = O(n)$, the total time for this is $O(n \log n)$.

Finally, we discuss the updating of $\mathcal{P}$. For all the bridges of I_{i-1} that go straight on to I_i, we need to increase the path length by the same value. By Lemma 7.4.10, they correspond to a single interval of points of P, and so this can be done in time $O(\log n)$. For each bridge that has been removed, we increase the path length for its interval of points, in time $O(\log n)$ per bridge removed. Finally, for each point $p \in P \cap s_i$ inserted in this stage, we set its path length to the correct value. The total cost of updating is $O(n \log n)$ according to Lemma 7.4.9. □

7.4.2 The optimization problem

We can now solve the original optimization problem using Megiddo's parametric search [Meg83].

Theorem 7.4.12 *Given a bridge graph on a set of $n + 1$ buildings that is a path, we can compute a configuration with the optimal dilation in time $O(n^3 \log n)$, or in time $O(nk \log^2 n)$, where k is the number of visible pairs.*

Proof: We run the algorithm of Lemma 7.4.7 with input σ^*, where σ^* is the optimal dilation. Since σ^* is not known, we parameterize all coordinates used by the decision algorithm in the form $a\sigma + b$. One can verify that all calculations performed by the algorithm are linear functions on the coordinates, and any linear combination of expressions of the form $a\sigma + b$ is again of this form.

Whenever the algorithm needs to compare two "numbers" $a\sigma + b$ and $a'\sigma + b'$, we compute the value σ_0 where $a\sigma_0 + b = a'\sigma_0 + b'$. We then run the decision algorithm of Theorem 7.4.8 using σ_0, which tells us whether $\sigma^* \leqslant \sigma_0$. The answer implies which of the two "numbers" is larger, and the parametrized algorithm can proceed. Note that if $\sigma^* = \sigma_0$, the outcome of the comparison is arbitrary—inspection of the algorithm shows that this is not a problem.[1]

When the parametrized algorithm finishes, it has computed a set of non-empty intervals $I_1, \dots, I_n$, since a configuration with dilation $\leqslant \sigma^*$ exists. Since the outcome of the parametrized algorithm changes for $\sigma = \sigma^*$, the algorithm must have made a comparison against σ^*. It follows that σ^* is the smallest σ_0 tested during the algorithm that resulted in a positive answer of the decision algorithm.

During the algorithm we maintain an interval of dilation values in which the optimal value is known to lie. Whenever a comparison requires answering $\sigma^* \leqslant \sigma_0$ for a σ_0 outside this interval, we can immediately return the correct answer without running the decision algorithm. At the end of the parametrized algorithm, we can report the upper end of the interval as σ^*.

[1]The reader may wonder why we do not simply augment the algorithm of Theorem 7.4.8 to report whether a configuration with dilation strictly less than σ exists. This is indeed possible, for instance by allowing open and half-open intervals I_i, but seems to be more complex than the observation that tests for equality are not actually needed.

Following Megiddo [Meg83], we organize the parametric algorithm as a "parallel" algorithm, using batches of independent computations. Recall that the algorithm of Lemma 7.4.7 proceeds in n stages, with stage i computing $I(p, q)$ for $O(n)$ pairs (p, q) with $q \in s_i$. The computations for each pair are independent, and take time $O(1)$. It follows that we can implement them in total time $O(n \log n)$ plus $O(\log n)$ calls to the decision algorithm [Meg83].

Forming the intersection I_i is equivalent to the computation of a maximum and a minimum of n "numbers" of the form $a\sigma + b$. Consider the "number" $a\sigma + b$ as the line $y = ax + b$. We compute the upper and lower envelope of all n lines, in time $O(n \log n)$ [Brg97KOS]. We can now perform binary search on the vertices of the envelopes, using $O(\log n)$ calls to the decision algorithm, to determine between which two vertices σ^* falls. This allows us to return the largest and smallest "number."

Each stage takes time $O(n \log n)$ plus $O(\log n)$ calls to the decision algorithm, so the total running time is $O(n^3 \log n)$. We can also use Theorem 7.4.11 to obtain total running time $O(nk \log^2 n)$. $\square$

7.4.3 The case of vertically sorted buildings

There is one interesting case where we can prove that there are only $O(n)$ visible pairs, namely when the buildings are sorted vertically along the path, that is, all bridges are directed vertically upwards.

Lemma 7.4.13 *If the bridge graph is a path, and the* $n + 1$ *buildings are sorted vertically along the path, then there are at most* $O(n)$ *visible pairs.*

Proof: A visible pair appears in the vertical decomposition of the set of buildings. $\square$

Theorem 7.4.11 now leads to an $O(n \log n)$-time decision algorithm for this case. It is possible to do even better, as we will show in this section.

The improvement is based on a bracket structure formed by the visible pairs. Consider a visible pair (p, q). The segment pq is vertical. Without loss of generality, let p be its bottom end. The path $\pi(p, q, B)$ is y-monotone, and since it cannot intersect pq, it lies either completely to the left or to the right of pq. We call a visible pair (p, q) where the path lies completely to the right of pq a *left-hand* visible pair, otherwise a *right-hand* visible pair.

Lemma 7.4.14 *Given a set of* $n + 1$ *vertically sorted buildings as defined above, and two left-hand visible pairs* (p, q) *and* (p', q')*, with* $p \in s_u$, $q \in s_t$, $p' \in s_{u'}$, $q' \in s_{t'}$. *Assume that* $u \leqslant u'$. *Then either the pairs are independent and* $t \leqslant u'$, *or* (p, q) *is bracketed around* (p', q')*, that is,* $p_x < p'_x$ *and* $u \leqslant u' < t' \leqslant t$.

Proof: If $u' < t$, then the building $s_{u'}$ lies completely to the right of the segment pq, and so we have $p_x < p'_x$. The path $\pi(p', q', B)$ lies completely to the right of the segment $p'q'$, and so it cannot reach q before reaching $s_{t'}$. This implies $u \leqslant u' < t' \leqslant t$. $\square$

In a left-hand visible pair (p, q), either p is the top-left corner of a building and q is on a bottom edge of a building, or q is a bottom-left corner, and p is on the top edge of a building. Lemma 7.4.14 leads to a simple algorithm to compute all left-hand visible pairs in linear time. (The same procedure, with opposite orientation, can be used to find all right-hand visible pairs.) All we need is a stack. In stage i, we repeatedly check whether $p_x \geqslant q_x$, where p is the top element of the stack and q is the bottom-left corner of s_i. While that is true, we report $(p, (p_x, q_y))$ as a visible pair and pop p from the stack. Finally, either the stack is empty, or $p_x < q_x$. In the latter case, we report $((q_x, p_y), q)$ as a visible pair. Finally, we push the top-left corner of s_i onto the stack, and proceed to the next stage.

Theorem 7.4.15 *Given a set of $n + 1$ vertically sorted buildings as defined above and a real number $\sigma > 1$. We can compute in $O(n)$ time a configuration B with dilation $\operatorname{dil}(B) \leqslant \sigma$, or determine that none exists.*

Proof: Again, we compute the intervals $I_1, \ldots, I_n$ in n stages. The visible pairs are computed during the process, using a "left-side stack" for the top-left corners and a "right-side stack" for the top-right corners. During the course of computation, we again maintain two data structures $\mathcal{P}$ and $\mathcal{B}$ to store path lengths and optimal bridges. Define the index of the top-left corner of building s_u to be $-(u + 1)$, and the index of the top-right corner of s_u to be $u + 1$.

$\mathcal{P}$ is implemented as a doubly-linked list. In this list, we store the path lengths $|\pi(p, m_i^p, B^p)|$ for all points p currently in the two stacks. The points are ordered by increasing index as defined above (which is the same as ordering them by the relation $\prec$ as defined before). The points on top of the stacks are thus found at the ends of the list. We store the path lengths by storing the *difference* between two adjacent values on the edges of the list. Only for the first and the last point in the list, we store $|\pi(p, m_i^p, B^p)|$ explicitly. Note that we do not explicitly store path lengths for points p that are not the corner of a building. However, these path lengths can be derived in constant time from path lengths that are stored in $\mathcal{P}$: if p on a building with top-left corner l is part of a left-hand visible pair, then $|\pi(p, m_i^p, B^p)|$ is simply $|\pi(l, m_i^l, B^l)| - |pl|$; similarly, if p is part of a right-hand visible pair, we can derive the path length from that of a top-right corner. Note that we can easily increase the path lengths for an interval of points in $\mathcal{P}$ in constant time by adjusting two difference or end values, provided we have pointers to the first and the last point of the interval.

$\mathcal{B}$ stores the optimal bridges b_i^p for all points p currently in the two stacks, and is implemented as a doubly-linked list as well. As before, a bridge shared by several points is stored only once. With each bridge, we store the index of the first and last point using it. For each point index, we store a pointer to the node of $\mathcal{P}$ that represents it.

A stage is now implemented as follows:

1. Using the two stacks, compute left-hand and right-hand visible pairs. Accessing the leftmost and rightmost nodes in $\mathcal{B}$ and $\mathcal{P}$, we can obtain path length values and bridge positions for these points. With these values, we compute the new interval I_i.

2. Remove from the ends of $\mathcal{P}$ all nodes for points popped from the two stacks. Remove from the ends of $\mathcal{B}$ all bridges that are not used by any point anymore (these bridges can be identified by comparing the index of the point on top of the stack with the indices of the points using the bridge). Adjust the interval of points used by the leftmost and rightmost bridge to end at the points on the top of the stacks.

3. For each bridge b in I_{i-1} that cannot go straight into I_i, update the path lengths for the corresponding interval of points in $\mathcal{P}$ (using the indices of the points for b and the pointers for these indices into $\mathcal{P}$).

4. Finally, remove all these bridges, update in $\mathcal{P}$ the interval of all points that use the remaining bridges (the bridges that do continue straight into I_i), add the top-left and top-right corner of s_{i-1} to $\mathcal{P}$ and add new bridges at the left and right margin of I_i, set the point interval of these bridges to the union of what was just deleted and the new corner points, and push the top-left and top-right corner of s_i on the two stacks.

Observe that all queries and updates of $\mathcal{B}$ and $\mathcal{P}$ are done at the ends of the lists and can be done in constant time each. Only updating path lengths in $\mathcal{P}$ requires access to an edge in the interior of the list, but this edge is found in constant time through the indices stored with the corresponding bridge at the end of $\mathcal{B}$. As before, the removal of bridges is charged to their creation. We thus spend constant time per stage, plus constant time per visible pair. $\square$

Parametric search now leads directly to the following theorem. Unlike in Theorem 7.4.12, we make no attempt to parallelize the parametric algorithm.

Theorem 7.4.16 *Given a bridge graph on a set of $n+1$ buildings that is a path, we can compute a configuration with the optimal dilation in time $O(n^2)$.*

Finally, we can compute a $(1+\varepsilon)$-approximation in linear time. We first show a quality bound for an arbitrary placement of the bridges. For completeness, we cover the general case as well.

Lemma 7.4.17 *Given a bridge graph $\mathcal{G}$ on a set of $n+1$ buildings that is a path, and any configuration B realizing $\mathcal{G}$. Then $\operatorname{dil}(B) \leqslant (\sigma^*)^2$, where σ^* is the optimal dilation. If the buildings are sorted vertically along the path, then we have $\operatorname{dil}(B) \leqslant 2\sigma^*$.*

Proof: Let $B^* = (b_1^*, b_2^*, \ldots, b_n^*)$ be an optimal configuration, that is $\operatorname{dil}(B^*) = \sigma^*$. Consider the interval of possible bridges between s_{i-1} and s_i, see Figure 7.10. Let d_i be the distance of b_i^* to the farther endpoint of the interval, and let h_i be the length of b_i^*. The pair of points (p', q') indicated in the figure has dilation $(2d_i + h_i)/h_i \leqslant \sigma^*$, which implies $2d_i \leqslant (\sigma^* - 1)h_i$.

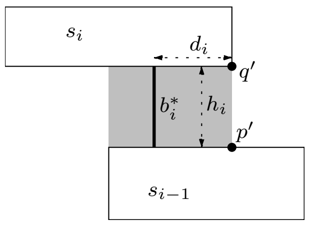


Figure 7.10: Proof of Lemma 7.4.17

Now consider any visible pair (p, q). If $\pi(p, q, B)$ uses bridges $b_u, \ldots, b_t$, we have

$$\begin{aligned} |\pi(p,q,B)| &\leqslant |\pi(p,q,B^*)| + \sum_{i=u}^{t} 2d_i \leqslant |\pi(p,q,B^*)| + (\sigma^* - 1)\sum_{i=u}^{t} h_i \\ &\leqslant |\pi(p,q,B^*)| + (\sigma^* - 1)|\pi(p,q,B^*)| \leqslant \sigma^*|\pi(p,q,B^*)| \\ &\leqslant (\sigma^*)^2\|pq\|. \end{aligned}$$

If the buildings are sorted vertically along the path, we can observe that $\|pq\| \geqslant \sum_{i=u}^{t} h_i$, and so we have

$$|\pi(p,q,B)| \leqslant |\pi(p,q,B^*)| + \sum_{i=u}^{t} 2d_i \leqslant \sigma^*\|pq\| + (\sigma^* - 1)\sum_{i=u}^{t} h_i \leqslant 2\sigma^*\|pq\|.$$

□

The lemma leads directly to a PTAS for the vertically ordered case: start with an arbitrary configuration, compute its dilation σ, and approximate σ^* by a binary search in the interval $\langle \sigma/2, \sigma]$. This gives us a $(1 + \varepsilon)$-approximation of σ^* after $O(\log(1/\varepsilon))$ calls to the decision algorithm, leading to the following result.

Theorem 7.4.18 *Given a set of $n + 1$ buildings sorted vertically along a path. We can compute a configuration with dilation at most $(1 + \varepsilon)$ times the minimum dilation in time $O(n \log(1/\varepsilon))$.*

7.5 Concluding remarks

We posed the following question: given n non-intersecting rectangles and a graph describing which pairs of rectangles are to be connected, can we find the connecting segments such that the dilation is minimized in polynomial time? We found that if the graph may contain cycles, this is not generally possible (unless P=NP), but if the graph is a path, it is possible. For the case of trees, the question is still open: so far, we can solve the problem by linear programming on $O(n^2)$ variables and constraints, but we have no strongly polynomial-time algorithm, that is, we have no polynomial-time algorithm for the real RAM model.

Having gained some insight in the bridge placement problem when the bridge graph is prescribed, it may now be interesting to study the problem with the bridge graph not given. For example: given a set of non-intersecting rectangles, find a set of connecting segments of given total length such that the dilation is minimized. Or: given a set of non-intersecting rectangles: find a set of connecting segments of minimum total length such that a given dilation is achieved. We might have to settle for approximation algorithms in this case.

When starting this research, we originally asked about how to connect convex polygonal objects by line segments unrestricted in orientation. It will be interesting to see to what extent the techniques for the axis-aligned case carry over to (approximation) algorithms for the unaligned case.

Bibliography

[Aga95] P. K. Agarwal and J. Matoušek: Dynamic half-space range reporting and its applications. *Algorithmica* 13:325–345 (1995).

[Aga97] P. K. Agarwal: Range Searching. In J. E. Goodman and J. O'Rourke (eds.), *CRC Handbook of Computational Geometry*, pp 575–598, CRC Press, 1997.

[Aga98E] P. K. Agarwal and J. Erickson: Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack (eds.), *Advances in Discrete and Computational Geometry*, Contemporary Mathematics 223: 1–56, American Mathematical Society, 1998.

[Aga98SW] P. K. Agarwal, M. Sharir, and E. Welzl: The discrete 2-center problem. *Discrete & Computational Geometry* 20:287–305 (1998).

[Aga01APV] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter: A Framework for Index Bulk Loading and Dynamization. In *Proc. 28th Int. Coll. Automata, Languages and Programming (ICALP)*, 2001, pp 115–127.

[Aga01BGHH] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort: Box-trees and R-trees with near-optimal query time. *Discrete & Computational Geometry* 28:291–312 (2002) (Chapter 3 of this thesis).

[Aga02] P. K. Agarwal, S. Govindarajan, and S. Muthukrishnan: Range searching in categorical data: colored range searching on a grid. In *Proc. 10th European Symp. Algorithms* (ESA 2002), pp 17–28.

[Agg88] A. Aggarwal and J. S. Vitter: The Input/Output complexity of sorting and related problems. *Communications ACM* 31(9):1116–1127 (1988).

[Agg97] C. Aggarwal, J. Wolf, P. Yu, and M. Epelman: The S-Tree: An Efficient Index for Multidimensional Objects. In *Proc. 5th Symp. Spatial Databases (SSD)*, 1997, LNCS 1262:350–373.

[Alo85] N. Alon, Z. Füredi, and M. Katchalski: Separating pairs of points by standard boxes. *European J. Combinatorics* 6:205–210 (1985).

[Ama96] N. Amato and Y. Wu: A randomized roadmap method for path and manipulation planning. In *Proc. Int. Conf. Robotics and Automation (ICRA)*, 1996, pp 113–120.

[Ang97] C. H. Ang and T. C. Tan: New Linear Node Splitting Algorithm for R-trees. In *Proc. 5th Symp. Spatial Databases (SSD)*, 1997, LNCS 1262: 339–349.

[Arg02] L. Arge, O. Procopiuc, and J. S. Vitter: Implementing I/O-efficient data structures using TPIE. In *Proc. 10th European Symp. Algorithms (ESA)*, 2002, pp 88–100.

[Arg03] L. Arge and J. Vahrenhold: I/O-efficient dynamic planar point location. *Computational Geometry: Theory & Applications* (to appear).

[Ary00] A. Arya and D. Mount: Approximate range searching. *Computational Geometry: Theory & Applications* 17(3-4):135–152 (2000).

[Bar96] G. Barequet, B. Chazelle, L. J. Guibas, J. S. B. Mitchell, and A. Tal: BOXTREE: A hierarchical representation for surfaces in 3D. *Computer Graphics Forum* 15(3):387–396 (1996).

[Bar99] G. Barequet and S. Har-Peled: Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. In *Proc. 10th Symp. Discrete Algorithms (SODA)*, 1999, pp 82–91.

[Bay72] R. Bayer and E. McCreight: Organization and maintenance of large ordered indexes. *Acta Informatica* 1:173–189 (1972).

[Bgn99] G. J. A. van den Bergen: *Collision Detection in Interactive 3D Computer Animation*. Ph.D. thesis, Eindhoven Technical University, 1999.

[Bhm99] C. Böhm, G. Klump, and H.-P. Kriegel: XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension. In *Proc. 6th Symp. Spatial Databases (SSD)*, 1999, LNCS 1651:75–90.

[Bkr92] B. Becker, P. G. Franciosa, S. Gschwind, T. Ohler, G. Thiemt, and P. Widmayer: Enclosing many boxes by an optimal pair of boxes. In *Proc. 17th Symp. Theoretical Aspects of Computer Science (STACS)*, 1992, LNCS 577:475–486.

[Bmn90] 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)*, 1990, pp 323–331.

[Brg92] M. de Berg, S. Carlsson, and M. Overmars: A general approach to dominance in the plane. *J. Algorithms* 13:274–296 (1992).

[Brg00] M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars: On R-trees with low stabbing number. In *Proc. 8th European Symp. Algorithms (ESA)*, 2000, LNCS 1879:167–178.

[Brg97KOS] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf: *Computational Geometry: Algorithms and Applications*, Springer, 1997.

[Brg97KSV] M. de Berg, M. J. Katz, A. F. van der Stappen, and J. Vleugels: Realistic input models for geometric algorithms. *Algorithmica* 34:81–97 (2002).

[Bri94] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger: Multi-step processing of spatial joins. In *Proc. Management of Data (SIGMOD)*, 1994, pp 197–208.

[Btd98] S. Berchtold, C. Böhm, and H.-P. Kriegel: Improving the query performance of high-dimensional index structures by bulk load operations. In *Proc. Extending Database Technology (EDBT)*, 1998, LNCS 1377:216–230.

[Cal95] P. B. Callahan and S. R. Kosaraju: A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields, *J. ACM* 42(1):67–90 (1995).

[Che89] L. P. Chew: There are planar graphs almost as good as the complete graph, *J. Computer and System Sciences* 39:205–219 (1989).

[Chn00] T. M. Chan: Approximating the diameter, width, smallest enclosing cylinder, and minimum-width annulus, In *Proc. 16th Symp. Computational Geometry (SoCG)*, 2000, pp 300–309.

[Chn02] T. M. Chan: Semi-online maintenance of geometric optima and measures. In *Proc. 13th Symp. Discrete Algorithms (SODA)*, 2002, pp 474–483.

[Chz88] B. Chazelle: A functional approach to data structures and its use in multidimensional searching, *SIAM J. Computing* 17:427–462 (1988).

[Chz01] B. Chazelle: *The Discrepancy Method: Randomness and Complexity*, Cambridge University Press, 2001.

[Com79] D. Comer: The ubiquitous B-tree. *ACM Computing Surveys* 11(2):121–137 (1979).

[Cor90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest: *Introduction to Algorithms*, MIT Press, 1990.

[Dey98] T. K. Dey: Improved bounds for planar k-sets and related problems. *Discrete & Computational Geometry* 19(30):373–382 (1998).

[Dwt94] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu: Client-server paradise. In *Proc. Very Large Databases (VLDB)*, 1994, pp 558–569.

[Dic00] M. Dickerson, C. A. Duncan, and M. T. Goodrich: K-D trees are better when cut on the longest side. In *Proc. 8th European Symp. Algorithms (ESA)*, 2000, LNCS 1879:179–190.

[Dun99] C. A. Duncan: *Balanced Aspect Ratio Trees*. Ph.D. thesis, John Hopkins University, 1999.

[Dun99GK] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov: Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *Proc. 10th Symp. Discrete Algorithms (SODA)*, 1999, pp 300–309.

[Epp00] D. Eppstein: Spanning trees and spanners, In J.-R. Sack and J. Urrutia (eds.), *Handbook of Computational Geometry*, pp 425–461, Elsevier, 2000.

[Fal87] C. Faloutos, T. Sellis, and N. Roussopoulos: Analysis of object oriented spatial access methods. In *Proc. Management of Data (SIGMOD)*, 1987, pp 426–439.

[Fal92] C. Faloutos and I. Kamel: *Packed R-trees using fractals*, technical report CS-TR-3009, University of Maryland, 1992.

[Gae98] V. Gaede and O. Günther: Multidimensional access methods, *ACM Computing Surveys* 30:170–205 (1998).

[Got96] S. Gottschalk, M. C. Lin, and D. Manocha: OBB-Tree: a hierarchical structure for rapid interference detection, In *Proc. Computer Graphics (SIGGRAPH)*, 1996, pp 171–180.

[Grc98a] Y. J. García, M. A. López, and S. T. Leutenegger: A Greedy Algorithm for Bulk Loading R-trees, In *Proc. Advances in GIS*, 1998, pp 163-164, and technical report 97-02, University of Denver.

[Grc98b] Y. J. García, M. A. López, and S. T. Leutenegger: On Optimal Node Splitting for R-trees, In *Proc. Very Large Databases (VLDB)*, 1998, pp 334–344.

[Gry79] M. R. Garey and D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[Gup95] J. Gupta, R. Janardan, and M. Smid: Further results on generalized intersection searching problems: counting, reporting, and dynamization. *J. Algorithms* 19: 282–317 (1995).

[Gut84] A. Guttmann: R-trees: a dynamic indexing structure for spatial searching. In *Proc. Management of Data (SIGMOD)*, 1984, pp 47–57.

[Has95] R. Hassin and A. Tamir: On the minimum diameter spanning tree problem, *Information Processing Letters* 53(2):109–111 (1995).

[Hav02] H. J. Haverkort, M. de Berg, and J. Gudmundsson: Box-trees for collision checking in industrial installations. In *Proc. 18th Symp. Computational Geometry (SoCG)*, 2002, pp 53–62.

[Hav02a] H. J. Haverkort, M. de Berg, and J. Gudmundsson: *Box-Trees for Collision Checking in Industrial Installations*, technical report UU-CS-2002-027, Utrecht University, 2002.

[Ho91] J.-M. Ho, D. T. Lee, C.-H. Chang, and C. K. Wong: Minimum diameter spanning trees and related problems. *SIAM J. Computing* 20(5):987–997 (1991).

[Jag90] H. V. Jagadish: Spatial Search with Polyhedra. In *Proc. Int. Conf. Data Engineering (ICDE)*, 1990, pp 311-319

[Jan93] R. Janardan and M. A. Lopez: Generalized intersection searching problems. *Int. J. Computational Geometry and Applications* 3:39–70 (1993).

[Kam93] I. Kamel and C. Faloutsos: On Packing R-trees In *Proc. Conf. Information and Knowledge Management (CIKM)*, 1993, pp 490–499.

[Kam94] I. Kamel and C. Faloutsos: Hilbert R-tree: An improved R-tree using fractals, In *Proc. Very Large Databases (VLDB)*, 1994, pp 500–509.

[Kan97] K. V. Ravi Kanth, D. Agrawal, A. El Abbadi, and A. K. Singh: Indexing Non-uniform Spatial Data. In *Proc. Int. Database Engineering and Applications Symp. (IDEAS)*, 1997, pp 289–298.

[Kan98] 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.

[Kar84] N. Karmarkar: A new polynomial-time algorithm for linear programming, *Combinatorica* 4:373–395 (1984).

[Kat97] N. Katayama and S. Satoh: The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proc. Management of Data (SIGMOD)*, 1997, pp 369–380.

[Kav95] L. Kavraki: *Random networks in configuration space for fast path planning*. Ph.D. thesis, Stanford University, 1995.

[Klo98] 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. Visualization and Computer Graphics* 4(1):21–36 (1998).

[Kre92] M. van Kreveld: *New Results on Data Structures in Computational Geometry*. Ph.D. thesis, Utrecht University, 1992.

[Krs98] S. Krishnan, A. Pattekar, M. C. Lin and D. Manocha: Spherical shells: a higher order bounding volume for fast proximity queries, *Proc. Workshop Algorithmic Foundations of Robotics (WAFR)*, 1998, pp 177–190.

[Krz03] D. Krizanc, P. Morin, and M. Smid: Range mode and range median queries on lists and trees, In *Proc. 14th Int. Symp. Algorithms and Computation (ISAAC)*, 2003, pp 517–526.

[Lar00] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha: Fast Distance Queries with Rectangular Swept Sphere Volumes. In *Proc. Int. Conf. Robotics and Automation (ICRA)*, 2000, pp 3719–3726.

[Lau78] U. Lauther: 4-dimensional binary search trees as a means to speed up associative searches in design rule verification of integrated circuits. *J. Design Automation and Fault-Tolerant Computing* 2(3):241–247 (1978).

[Leu97] S. T. Leutenegger, M. A. Lopez, and J. Edington: STR: A simple and efficient algorithm for R-tree packing, In *Proc. 13th Int. Conf. Data Engineering (ICDE)*, 1997, pp 497–506.

[Man99] Y. Manolopoulos, Y. Theodoridis, and V. Tsotras: *Advanced Database Indexing*, Kluwer Academic Publishers, 1999.

[Man03] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis: *R-Trees Have Grown Everywhere*, technical report available at http://www.rtreeportal.org/, 2003.

[Mat93] J. Matoušek: Range Searching with Efficient Hierarchical Cuttings. *Discrete & Computational Geometry* 10:157–182 (1993).

[Meg83] N. Megiddo: Applying parallel computation algorithms in the design of serial algorithms. *J. ACM* 30(4):852–865 (1983).

[Molog] *MOLOG: Motion for Logistics*, Esprit LTR Project 28226, http://www.laas.fr/molog/.

[Nar04] G. Narasimhan and M. Smid: *Geometric Spanner Networks*, Cambridge University Press, to appear.

[Nie97] 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 1340:153–198, 1997.

[Nie00] J. Nievergelt and P. Widmayer: Spatial data structures: concepts and design choices, In J.-R. Sack and J. Urrutia (eds.), *Handbook of Computational Geometry*, pp 725–764, Elsevier, 2000.

[Oos90] P. van Oosterom: *Reactive Data Structures for Geographic Information Systems*. Ph.D. thesis, Leiden University, 1990.

[Ore90] J. Orenstein: A comparison of spatial query processing techniques for native and parameter spaces. In *Proc. Management of Data (SIGMOD)*, 1990, pp 343–352.

[Pel89] D. Peleg and A. Schäffer: Graph spanners. *J. Graph Theory* 13:99–116 (1989).

[Pro03] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter: Bkd-tree: A dynamic scalable kd-tree, In *Proc. Symp. Spatial and Temporal Databases (SSTD)*, 2003, pp 46–65.

[Rob81] J. Robinson: The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. Management of Data (SIGMOD)*, 1981, pp 10–81.

[Ros01] K. A. Ross, I. Sitzmann, and P. J. Stuckey: Cost-based Unbalanced R-Trees. In *Proc. Statistical and Scientific Database Management (SSDBM)*, 2001, pp 203–212.

[Rou85] N. Roussopoulos and D. Leifker: Direct spatial search on pictorial databases using packed R-trees. In *Proc. Management of Data (SIGMOD)*, 1985, pp 17–31.

[Sha01] M. Sharir, S. Smorodinsky, and G. Tardos: An Improved Bound for k-Sets in Three Dimensions. *Discrete & Computational Geometry* 26(2): 195–204 (2001).

[Sel87] T. Sellis, N. Roussopoulos, and C. Faloutsos: The R$^+$-tree: A dynamic index for multi-dimensional objects, In *Proc. Very Large Databases (VLDB)*, 1987, pp 507–518.

[Sit99] I. Sitzmann and P. J. Stuckey: *The O-Tree—A Constraint-Based Index Structure*, technical report, University of Melbourne, 1999.

[Spr03] M. J. Spriggs, J. M. Keil, S. Bespamyatnikh, M. Segal, and J. Snoeyink: Computing a $(1+\epsilon)$-approximate geometric minimum-diameter spanning tree. *Algorithmica*, 38(4):577–589 (2004).

[Sta94] A. F. van der Stappen: *Motion Planning amidst Fat Obstacles*. Ph.D. thesis, Utrecht University, 1994.

[Sta98] A. F. van der Stappen, M. Overmars, M. de Berg, and J. Vleugels: Motion planning in environments with low obstacle density. *Discrete & Computational Geometry* 20:561–587 (1998).

[Sve97] P. Švestka: *Robot motion planning using probabilistic roadmaps*. Ph.D. thesis, Utrecht University, 1997.

[The96] Y. Theodoridis and T. Sellis: A model for the prediction of R-tree performance. In *Proc. Principles Database Systems (PODS)*, 1986, pp 161–171.

[Tiger] *TIGER/Line*$^{\text{TM}}$ *files, 1997 technical documentation*, http://www.census.gov/geo/tiger/TIGER97D.pdf, 1998.

[Whi96] D. A. White and R. Jain: Similarity Indexing: Algorithms and Performance. In *Storage and Retrieval for Image and Video Databases (SPIE)* 1996, pp 62–73.

[Zho99] Y. Zhou and S. Suri: Analysis of a bounding box heuristic for object intersection, In *Proc. 10th Symp. Discrete Algorithms (SODA)*, 1999, pp 830–839.

Acknowledgements

The results presented in this thesis are the fruits of joint work with Pankaj Agarwal, Lars Arge, Tetsuo Asano, Mark de Berg, Otfried Cheong, Hazel Everett, Joachim Gudmundsson, Mikael Hammar, Naoki Katoh, Sang-Min Park, Chan-Su Shin, Alexander Wolff, and Ke Yi. It was great to work with all of them, as each of these researchers contributed different perspectives on the problems we studied. Thank you all! Special thanks go to Mark de Berg, my adviser, especially for facilitating and encouraging all of these collaborations. Special thanks also to Otfried Cheong, whose advice and insights about the life of a computational geometer I found very helpful, and to Ke Yi, who did not only contribute to the development of the algorithm which may well be the main result of this thesis, but he also implemented it.

Jaesook Cheong, Otfried Cheong, Joachim Gudmundsson (yes, it's them again), Marc van Kreveld, Frank van der Stappen, Mirela Tănase and Jan Vahrenhold: thank you for working with me on several papers—or papers that may sometimes be—that are not included in this thesis.

The work was facilitated primarily by Mark Overmars (Utrecht University), who hired me as a PhD student, and the Netherlands' Organisation for Scientific Research NWO, which funded the position. Many thanks also to Andrzej Lingas (Lunds Universitet), Prosenjit Bose (Carleton University), Jan Vahrenhold (Universität Münster) and Pankaj Agarwal and Lars Arge (Duke University), who invited me to visit them in their universities and also covered much of the costs of these visits. These research visits were of great value to the work presented in this thesis, either by being perfect breeding grounds for getting good results, or by widening my perspective, allowing me to see my work in a broader context.

I thank Lars Arge, Harry Buhrman, Jan van Leeuwen, and Peter Widmayer, for their willingness to serve on the reading committee of my thesis.

Running a small board game company with Jeroen Doumen, Tamara Jannink and Joris Wiersinga made for many pleasant days together and for great opportunities to develop or use talents I could not (yet) use or develop in my work as a researcher. The experience gained from producing games with you often proved and often still proves to be of great value for my life as a computer scientist. Thank you very much for working with me.

I am very grateful to my parents, grandparents, friends and colleagues for their company and support during the years I studied in Utrecht. Jaesook, thank you especially for the last year—I could not have wished for nicer office mates than you and Junha. Mirela, thank you for all the music!

Samenvatting in het Nederlands

“Computationeel-geometrici” doen onderzoek naar het rekenen met meetkundige voorwerpen. Voorbeelden van zulke voorwerpen zijn punten, lijnen en veelhoeken in het vlak—die bijvoorbeeld een stadsplattegrond voorstellen—of bollen, blokken en ingewikkelder drie-dimensionale voorwerpen—die bijvoorbeeld de inrichting van een electriciteitscentrale voorstellen. In deze gevallen stellen de meetkundige voorwerpen de vorm en afmetingen van tastbare zaken in de werkelijkheid voor. Maar dat is niet altijd zo. Een bestand met daarin de leeftijd en het loon van de werknemers van een bedrijf, kan ook als een bestand met punten in het vlak worden gezien: elk punt stelt een werknemer voor, waarbij één coördinaat zijn leeftijd aanduidt en de andere coördinaat het loon. Berekeningen met meetkundige voorwerpen komen voor in veel toepassingen van computers: gegevensbeheer, computergesteund ontwerpen, geografische informatiesystemen, vluchtsimulatoren, andere toepassingen van schijnwerkelijkheid, robotica en routeplanning zijn maar een paar voorbeelden.

Dit proefschrift bevat resultaten in twee onderzoeksgebieden waarmee computationeel-geometrici zich bezighouden, ten eerste *ruimtelijke gegevensstructuren*, en ten tweede *ruimtelijke netwerken*.

Ruimtelijke gegevensstructuren

Om doeltreffend te kunnen rekenen met meetkundige voorwerpen, is het van overwegend belang dat we verzamelingen van zulke voorwerpen doeltreffend kunnen opslaan, doorzoeken, en soms ook wijzigen. Voor voorwerpen die meerdere dimensies hebben, is dat niet eenvoudig. De sleutel tot een goede oplossing ligt vaak in het op een zinvolle manier groeperen van de voorwerpen, zodat we bij het rekenen groepen met voorwerpen die terzake doen snel kunnen onderscheiden van groepen zonder zulke voorwerpen.

Een ruimtelijke gegevensstructuur is een verzameling meetkundige voorwerpen die is geordend, in groepen is opgedeeld of op een andere wijze is behandeld, zodat bepaalde vragen over de voorwerpen snel kunnen worden beantwoord.

Een belangrijk voorbeeld van zo'n vraag is: welke voorwerpen liggen binnen een bepaald zoekgebied? De doeltreffendheid van een gegevensstructuur wordt bepaald door de hoeveelheid benodigd geheugen, de tijd die het kost om de structuur te bouwen, de tijd die het kost om er voorwerpen aan toe te voegen of uit te verwijderen, en de tijd die het kost om vragen te beantwoorden. Hierbij meten we de tijd doorgaans niet in milliseconden. In plaats daarvan gaan we na hoe het aantal benodigde bewerkingen door de centrale verwerkingseenheid van de computer, of het aantal keren dat de vaste schijf moet worden gelezen of beschreven, afhangt van het aantal opgeslagen voorwerpen. We schrijven dan bijvoorbeeld dat de benodigde rekentijd $O(\sqrt{n})$ of $\Omega(\sqrt{n})$ is. Dat betekent dan, dat als n het aantal voorwerpen is, het aantal benodigde bewerkingen hoogstens, respectievelijk minstens, recht evenredig is met de wortel uit het aantal voorwerpen.

Het geval dat de voorwerpen punten zijn, is in het verleden al grondig bestudeerd. In dit proefschrift houden we ons met twee ingewikkelder gevallen bezig: gegevensstructuren voor *groepen* punten, en structuren voor *grotere voorwerpen*.

Gegevensstructuren voor groepen punten. Bij sommige vraagstukken gaat het niet om afzonderlijke punten, maar om groepen punten. Denk bijvoorbeeld aan een geografisch informatiesysteem, waarin n punten zijn opgeslagen die fabrieken voorstellen. Elk punt heeft een kleur die aangeeft wat voor een soort producten de betreffende fabriek maakt (chemicaliën, apparaten, voeding, energie, enz.). Nu kun je vragen stellen zoals: "Wat voor een soorten fabrieken staan er binnen een vierkant van tien bij tien kilometer rondom een bepaalde stad?" Zulke vragen kunnen worden beantwoord met behulp van gewone gegevensstructuren voor het zoeken van punten in rechthoekige gebieden: we vragen eerst alle punten op die binnen het vierkant liggen, en lopen die dan een voor een na om te kijken welke verschillende kleuren ze hebben. Het nadeel van die aanpak is dat het onnodig tijdrovend is als er in het zoekgebied heel veel punten liggen, maar slechts weinig verschillende kleuren. Gegevensstructuren waarmee we gelijk de verschillende kleuren kunnen vinden, zonder de tussenstap langs de afzonderlijke punten te maken, verdienen dan de voorkeur. Nog mooier is het als we het zoekresultaat kunnen beperken tot de kleuren die *betrekkelijk vaak* voorkomen in het zoekgebied, zonder dat we ook alle uitzonderingen te zien krijgen. Anders zal bij een groot gebied al gauw bijna elke kleur worden gerapporteerd, al is het maar vanwege één fabriek. Wat met "*betrekkelijk vaak*" wordt bedoeld, zal afhangen van de toepassing.

In hoofdstuk 2, *"Significant-Presence Range Queries in Categorical Data"*, bekijken we dit zoekprobleem voor het geval dat "*betrekkelijk vaak*" betekent dat tenminste een bepaald percentage van alle punten van de betreffende kleur in het zoekgebied ligt. Voor het geval van n punten in één dimensie, beschrijven we een gegevensstructuur die $O(n)$ geheugen gebruikt en vragen naar vaak voorkomende kleuren in een gebied beantwoordt in $O(\log n + k)$ tijd (hierbij is k het aantal gevonden kleuren). Helaas leidt de veralgemenisering van onze aanpak naar meer dimensies tot een gegevensstructuur die in twee dimensies al $\Omega(n^3)$ geheugen nodig heeft. We tonen dit aan met behulp van de volgende stelling. Stel: P is een verzameling van n punten in een d-dimensionale ruimte, en t is een getal tussen 1 en $\frac{n}{2d}$. Het aantal verschillend samengestelde rechthoeken die precies t punten van P bevatten is dan in het slechtste geval $\Omega(n^d t^{d-1})$ en $O(n^d t^{d-1})$.

Omdat vragen naar vaak voorkomende kleuren kennelijk moeilijk nauwkeurig te beantwoorden zijn, kijken we ook naar een minder strikte vorm van dit vraagstuk. Hierbij mogen we ook kleuren melden die eigenlijk net te weinig punten in het zoekgebied hebben. Bijvoorbeeld: in plaats van alleen alle kleuren waarvan minstens 50% van de punten in het zoekgebied ligt, mogen we ook kleuren melden die tussen 40% en 50% van de punten in het zoekgebied hebben liggen. Het benodigde geheugen voor onze gegevensstructuur voor dergelijke vragen hangt af van het aantal kleuren, de drempelwaarde en de vereiste nauwkeurigheid, maar verrassend genoeg niet van het aantal punten.

Gegevensstructuren voor grotere voorwerpen: hiërarchieën van omhullenden. Het is niet eenvoudig om doeltreffende gegevensstructuren te ontwerpen voor voorwerpen die meer zijn dan een punt, bijvoorbeeld lijnstukken, bollen of veelvlakken. Oplossingen die in theorie heel doeltreffend zijn, zijn vaak zo ingewikkeld dat ze in de praktijk nauwelijks bruikbaar zijn. Bovendien zijn ze meestal slechts geschikt voor één soort zoekopdracht. Een goede praktische oplossing wordt vaak geboden door hiërarchiën van omhullenden: ze zijn makkelijk te verwezenlijken, en hebben weinig geheugen nodig.

Een hiërarchie van omhullenden laat zich beschrijven als een boomstructuur. In de bladeren van de boom vinden we de meetkundige voorwerpen die we willen opslaan. Bij elke vertakking slaan we voor elke tak een rechthoek op, die net groot genoeg is om de voorwerpen in de bladeren aan die tak geheel te bevatten, en waarvan de zijden evenwijdig aan de assen van het assenstelsel lopen. Een zoekopdracht met een willekeurig zoekgebied kan nu als volgt worden uitgevoerd. We lopen de boom na, te beginnen bij de wortel. Telkens als we bij een vertakking komen, kijken we voor elke tak of de bijbehorende omhullende rechthoek geheel of gedeeltelijk in ons zoekgebied valt. Zo ja, dan onderzoeken we die tak verder. Zo nee, dan kunnen we die tak overslaan. Uiteindelijk vinden we op deze manier precies de bladeren met de voorwerpen waarvan de omhullende rechthoek in het zoekgebied ligt. We moeten dan alleen nog nagaan of de voorwerpen zelf ook in het zoekgebied vallen. Op een vergelijkbare manier kunnen we bijvoorbeeld ook zoeken naar het voorwerp dat het dichtst bij een gegeven zoekpunt ligt.

Bij het ontwerp van een hiërarchie van omhullenden kunnen we een aantal dingen kiezen.

Om te beginnen de vorm van de omhullenden. Hierboven heb ik de structuur beschreven aan de hand van omhullende rechthoeken, maar we zouden ook andere vormen kunnen gebruiken. In de praktijk blijken rechthoeken echter vaak goed te werken: ze vergen weinig opslagruimte, en de berekening of een rechthoek al dan niet in het zoekgebied valt kan veel sneller worden gedaan dan dezelfde berekening voor andere vormen.

Hierboven heb ik in het midden gelaten hoe de vertakkingen in de boom eruit zien: moeten die hoge graad hebben—dat wil zeggen dat er bij elke vertakking een groot aantal takken tegelijk worden afgesplitst—of lage graad? Wat het handigst is, blijkt onder meer af te hangen van de verhouding tussen de snelheid van de vaste schijf en de snelheid van de centrale verwerkingseenheid. Daarom worden hiërarchieën van omhullenden vaak zo ontworpen dat we de graad kunnen instellen, afhankelijk van de apparatuur.

De belangrijkste keuze die we verder moeten maken, is hoe de meetkundige voorwerpen in de bladeren van de boom worden gerangschikt. Een van de grootste zorgen hierbij is dat we willen vermijden dat omhullende rechthoeken van verschillende takken elkaar overlappen. Dan bestaat immers de kans dat we beide takken moeten doorzoeken, terwijl we bij een andere rangschikking van de voorwerpen misschien met het doorzoeken van één tak hadden kunnen volstaan. Met voorwerpen die groter dan punten zijn, is overlap echter niet volledig te vermijden. Bovendien blijkt het niet altijd voordelig te zijn om de hoeveelheid overlap zo klein mogelijk te houden: in sommige gevallen werkt het averechts. Ongeveer de helft van dit proefschrift is gewijd aan de vraag hoe we een hiërarchie van omhullende rechthoeken moeten inrichten om te kunnen garanderen dat zoekopdrachten werkelijk snel kunnen worden uitgevoerd.

Een hiërarchie van omhullende rechthoeken met instelbare graad wordt een R-boom genoemd. De R-boom werd in 1984 voorgesteld door Guttman. Sindsdien is er veel onderzoek gedaan naar de beste manier om de voorwerpen in de boom te rangschikken. Toch was het bij alle tot nu toe voorgestelde manieren nog zo dat voor een zoekopdracht in het slechtste geval alle bladeren van de boom moeten worden nagelopen, zelfs als de zoekopdracht niets oplevert. Wij hebben onderzoek gedaan naar manieren om een R-boom zo in te richten, dat we kunnen garanderen dat de bladeren waarvan de omhullende rechthoeken in het zoekgebied liggen snel kunnen worden gevonden.

Het was al bekend dat er verzamelingen van rechthoeken zijn, zodat elke R-boom voor zo'n verzameling een zoektijd van in het slechtste geval $\Omega(n^{1-1/d})$ heeft (hierbij is n het aantal rechthoeken en d het aantal dimensies). In dit proefschrift tonen we aan dat dit zelfs het geval is als de rechthoeken elkaar nauwelijks overlappen, of als we zoeken met bijna vierkante gebieden. Alleen als we én weinig overlap, én bijna vierkante zoekgebieden, én maar twee dimensies hebben, dan is het denkbaar dat we voor het slechtste geval een betere zoektijd kunnen garanderen. We beschrijven in dit proefschrift ook daadwerkelijk een manier om R-bomen te bouwen die onder deze voorwaarden een betere zoektijd garandeert, namelijk $O(\log^2 n + k)$. Hierbij is k het aantal bladeren waarvan de omhullende rechthoeken geheel of gedeeltelijk in het zoekgebied liggen.

Verder beschrijven we een manier om R-bomen te bouwen die voor de overige gevallen de best denkbare zoektijd, namelijk $O(n^{1-1/d} + k)$, garandeert. We hebben onze aanpak ook getest. De uitkomsten geven aan dat onze methode moeilijke verzamelingen rechthoeken beter kan verwerken dan de tot nu toe bekende methoden. Ook in makkelijker, meer gebruikelijke omstandigheden kan onze aanpak zich met de tot nu toe bekende methoden meten.

We beschrijven ook een manier om onze beide methoden te combineren, waardoor in eenvoudige gevallen zoals hierboven beschreven een zoektijd van $O(\log^2 n + k)$ wordt gegarandeerd, terwijl naar mate de invoer ingewikkelder wordt, de garantie geleidelijk verslechtert tot $O(n^{1-1/d} + k)$.

In drie dimensies is de aanname dat er weinig overlap is en dat zoekgebieden bijna vierkant—dat wil zeggen: kubusvormig—zijn, helaas niet genoeg om een goede zoektijd te kunnen garanderen. We kunnen dat, op zijn minst in theorie, echter wel als we aanvaarden dat er nog twee omstandigheden zijn waarin een zoekopdracht moeilijk mag zijn. Ten eerste de omstandigheid dat er heel veel drie-dimensionale rechthoeken—blokken dus—zijn die het zoekgebied niet snijden, maar wel heel dichtbij liggen. Ten tweede de omstandigheid dat er heel veel heel platte blokken op een bepaalde manier vlak bij elkaar liggen. We definiëren een maat voor een verzameling blokken, het plakjesgetal, die aangeeft in hoeverre de tweede omstandigheid zich voor kan doen. Onze manier om R-bomen te bouwen garandeert korte zoektijden, $O(\log^4 n + k)$, mits het aantal blokken *vlakbij* het zoekgebied niet al te veel groter is dan het aantal blokken *in* het zoekgebied, en het plakjesgetal niet al te hoog is. Dezelfde zoektijden worden bovendien ook gehaald bij het zoeken met niet-blokvormige zoekgebieden.

Ruimtelijke netwerken

Een netwerk bestaat uit knopen en verbindingen. In een ruimtelijk netwerk zijn de knopen en verbindingen meetkundige voorwerpen. Gewoonlijk hebben die een plaats en afmetingen in de ruimte, maar ze kunnen ook niet-meetkundige eigenschappen hebben, zoals de aanlegkosten en de snelheid van de verbindingen. Ruimtelijke netwerken vind je overal in het dagelijks leven: wegennetwerken, telefoonnetwerken, computernetwerken enz. Het doel van het netwerk is meestal om een verbinding tussen de knopen te verschaffen, het liefst een korte, of snelle, verbinding. Wat dat betreft zou het volmaakte netwerk er een zijn met een rechtstreekse verbinding tussen alle knopen. In de praktijk is dat meestal te duur, en moet er een tussenweg worden gevonden tussen een goed en een goedkoop netwerk. Dat leidt tot optimaliseringsproblemen van de vorm: vind voor een gegeven verzameling knopen de “beste” verzameling verbindingen die aan bepaalde voorwaarden voldoet. We hebben twee van zulke vraagstukken onderzocht.

In hoofdstuk 6, *“Facility location and the geometric minimum-diameter spanning tree”*, gaat het om het volgende geval. De knopen zijn punten in het vlak, en een verbinding tussen twee knopen kost altijd hetzelfde, ongeacht de afstand. Er zijn n knopen, en we mogen $n - 1$ verbindingen aanleggen, zodat we net elke knoop met het netwerk kunnen verbinden. Ons doel is de verbindingen zo te kiezen dat de grootste afstand tussen twee punten in het netwerk zo klein mogelijk is. Nauwkeurige oplossingen vergen tot op heden helaas bijna $O(n^3)$ rekentijd. Onze aanpak vindt een oplossing waarbij de grootste afstand in het netwerk tot $(1 + \varepsilon)$ keer zo groot is als in de beste oplossing, maar het vindt die oplossing in $O((\frac{1}{\varepsilon})^5 + n)$ tijd. Hierbij kan de onnauwkeurigheid ε willekeurig dicht bij nul worden gekozen. Voor grote verzamelingen knopen en bescheiden nauwkeurigheidseisen garandeert onze manier van uitrekenen een veel kortere rekentijd dan de snelste bekende precieze berekeningswijze.

In hoofdstuk 7, *“Optimal spanners for axis-aligned rectangles”*, onderzoeken we het volgende geval. De knopen in ons netwerk zijn n rechthoeken die evenwijdig aan de assen van een twee-dimensionaal assenstelsel zijn geplaatst. Ze moeten met lijnstukken worden verbonden. Daarbij is gegeven tussen welke paren rechthoeken een lijnstuk moet worden geplaatst; de bedoeling is nu om voor elk zo’n paar te bepalen waar precies het lijnstuk tussen de twee rechthoeken moet worden geplaatst. Denk hierbij bijvoorbeeld aan een aantal gebouwen die we met loopbruggen met elkaar willen verbinden. Daarbij zou het erg vervelend zijn als de bruggen zo worden geplaatst, dat je een grote omweg moet maken om een kamer in een ander gebouw te bereiken die je vanuit je eigen kamer zo kunt zien liggen. De bedoeling is daarom dat we de lijnstukken zo plaatsen dat de grootst mogelijke verhoudingsgewijze omweg—dat is de verhouding tussen de afstand over de bruggen en de korste rechtstreekse weg—zo klein mogelijk is. In het algemeen blijkt dit vraagstuk te behoren tot de zogenaamde *NP-moeilijke* problemen—wat wil zeggen dat het onwaarschijnlijk is dat een nauwkeurige oplossing snel kan worden berekend. In bijzondere gevallen kan de beste plaatsing van de lijnstukken wel gegarandeerd snel worden berekend: als het netwerk een boomstructuur heeft, dan kan dat door middel van lineair programmeren met $O(n^2)$ veranderlijken en voorwaarden; als het netwerk een pad is—dat wil zeggen dat we de rechthoeken kunnen nummeren van 1 tot n zodat elk lijnstuk opeenvolgende rechthoeken in de nummering met elkaar verbindt—dan kunnen we de beste plaatsing berekenen in $O(n^3 \log n)$ tijd.

Curriculum vitae

Herman Haverkort was born in Arnhem, the Netherlands in 1974. He went to the city gymnasium in 1986, got his first computer in 1987, and started writing computer games. In 1992 he graduated from the gymnasium, and he studied computer science ever since. In 1999, Utrecht University granted him his Master's degree. Since then he worked as a PhD student in the university's Institute of Information and Computing Sciences, while putting a part of his time into law courses and in running the Splotter board games company.