R-Trees

An efficient structure for spatial data management

A.N. Yzelman

This report was written as fulfilment of the requirements for the masters degree 'Scientific Computing' at the University of Utrecht, under supervision of Gerard Sleijpen and Arno Swart. The research was largely conducted at Alten Nederland B.V. (Capelle aan den Ijssel) in the form of a practical internship, under supervision of Eric Haesen. The software described in this report has been implemented in C++. The software is intended for by Shell Exploration and Production, Rijswijk (supervision: Hans Molenaar). By authority of Eric Haesen and Hans Molenaar, Alten Nederland and Shell EP agree that this report and the implemented software can be freely distributed.

CONTENTS

Preface

This master's thesis is written in context of the *Scientific Computing* Master's programme, which I followed at Utrecht University in the Netherlands, during the academic year 2006/2007. It reports on the findings done while doing an internship at Alten Nederland, part of the larger Alten Group based in France.

Alten Nederland is a company specialised in technical consultancy, and currently has consultants working for some of the larger technologically oriented companies such as ASML, Phillips and Shell, amongst others. Having specialised expertise groups in areas ranging from *Simulation and Modelling* to *Business Critical Systems* to *Embedded Systems*, it is fair to say this company not only has a great interest in technology, but also aims for a broad range. In any case, I have found Alten to be a very dynamic, fast-growing company with skilled and enthusiastic employees.

The internship consisted of researching a data management structure, primarily for use with oil reservoir simulation at Royal Dutch Shell. Apart from this, Alten Nederland itself was also interested in the research results and may use them in other application areas as well.

The internship took a full six months in total, and was essentially guided by Arno Swart (representing Alten) and Gerard Sleijpen (representing Utrecht University). Also involved from a more managing perspective were Hans Molenaar from Shell, and Eric Haesen from Alten.

I would like to express here my thanks to Arno for his guidance; I surely learnt much in these six months, certainly not in the last place a great deal about `C++` programming. Also thanks to Gerard who has always provided good criticism on the advancements of this thesis; at the very least it is safe to say this thesis would have been less well written without his comments.

Finally I would like to thank Hans and Eric for giving me this opportunity, and the colleagues at Alten for their gracious hospitality and for giving me a small taste of the business Alten is in.

Albert-Jan N. Yzelman,
August 2007,

ajy777 "at" gmail "dot" com

CHAPTER

1

ONE

Introduction

In this thesis, we will investigate a particular family of datastructures used for storing and efficient querying of *multi-dimensional data objects*, namely the so-called *R-trees*. We will theoretically and experimentally investigate a selection of R-tree variations, and attempt to find the best R-tree for use in our application area.

This area is the efficient storage of and querying of three-dimensional rectangle-like grid elements as used in an oil reservoir simulation software package called *Dynamo*, which is in use by and developed by Shell.

Globally, we measure the quality of data structures by:

- Speed of *construction* of the data structure (Section 4.6).
- Speed of answering to (random) *queries* (Section 4.7),
- *Memory* allocation and usage (Section 4.8),

In all cases our objective is, of course, to minimise.

R-trees are data structures designed to store multi-dimensional objects. An essential problem with multi-dimensional objects (or rather, any object with dimension two or larger), is that there exists no good ordering on such objects. This makes the design of searchable multi-dimensional data structures difficult.

R-trees have in common that they attempt to order data elements by using *Minimum Bounding Rectangles (MBRs)*. This can be done in many different ways, giving rise to just as many different R-Tree variations. The R-tree data structure was introduced in 1984, by A. Guttman in [**Gut84**].

1.1. Searching ordered sets

We will first demonstrate why an ordering is advantageous when constructing a data structure. Assume we wish to efficiently store and query a finite set S of integers. For simplicity, we will assume these integers are all positive, i.e., $S = \{s_1, s_2, \ldots, s_n\} \subset \mathbb{Z}$. Also, we will assume the only query we are interested in, is if a given number x is contained in S, i.e. if $x \in S$. If we simply store all values in S in a simple, unordered array, searching for any x will (worst case) require n comparisons, or rather $n/2$ comparisons on average. This is because we will have to check for each element $s \in S$ if $x = s$, until such an x is found.

Now we will make use of the natural ordering of integers. Assume we have now stored the elements of S in a *sorted* array, in ascending order. Then, we may search for a given x by checking if x falls in the range of the array, and if so, recursively check if this also holds for the appropriate

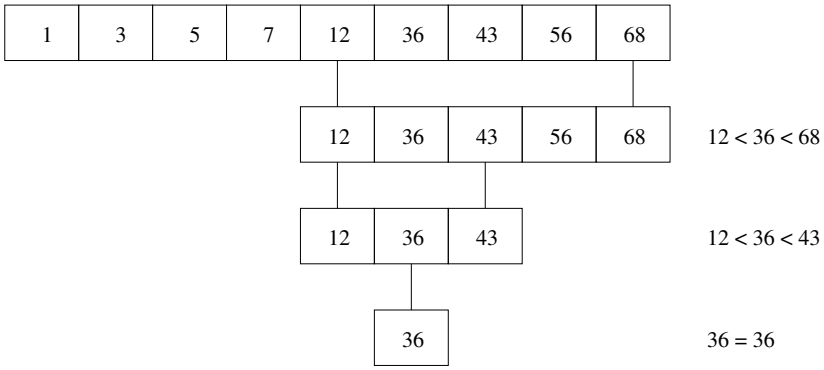


FIGURE 1.1. Illustration of a bisection search method on a sorted array. We check if 36 contained in the array.

half of that array[1]. See figure 1.1 for an illustration. Continually taking the half of the array will eventually result in considering a subarray of size 2, for which we can easily check if it contains x. If it does not, then certainly $x \notin S$.

By only considering the values at the 'halves' of the array, we have compared at most $\lceil \log_2 n \rceil$ ($\log_2 n$ rounded up) elements of S with x, at the cost of pre-sorting the elements of S in an array. The array still requires the storage of only the n elements of S, and as such the only loss incurred when compared to the unsorted search described earlier, is presorting[2], which only needs to be executed once. Hence, when a lot of queries are performed on the same dataset, querying the sorted list results in better efficiency.

1.2. MBRs

Sadly, we cannot unambiguously sort multi-dimensional data; there is no unambiguous way we can state a given cube is "less than" a given other cube, for example. However, in an application using a very finely discretised grid, having queries run with a complexity proportional to the number of grid elements or worse, is simply unacceptable. One can furthermore imagine that such linear complexity is at the very least undesirable in any other application as well. An added problem is that we would like R-trees to handle multi-dimensional data of arbitrary shape, which increases the difficulty of defining a sane ordering even more.

To tackle these problems, R-trees employ minimum bounding rectangles[3]. An MBR of a given multi-dimensional object o, with non-zero volume, is defined to be the (hyper)rectangle R containing o with the volume of R as small as possible.

On these MBRs we then define some ordering, which we will use to construct the data structure. Note that since MBRs still are multi-dimensional objects, any ordering imposed on it still is ambiguous. However, orderings based on containment, volume and/or (Hilbert) coordinates (Section 2.5.2) of the MBR centres produce reasonable datasets, as will be seen later on.

1.3. R-Trees

The way R-trees work with MBRs is as follows. Consider a simple tree as in figure 1.2. Each tree node is linked to the MBR which contains the MBRs of the children nodes, while the MBRs

[1]This method is also generally known as the *bisection method*. This method can also be generalised to higher dimensions, and is used in the Dynamo (Section 1.4) package at the time of writing.

[2]One can also opt to sort the array while initially constructing it, by means of sorted injection.

[3]Since R-trees were firstly introduced to handle two-dimensional objects, the terminology refers to rectangles here; in larger dimensions these are of course easily extended to hyperrectangles. For brevity however, we will keep referring to rectangles in this context.

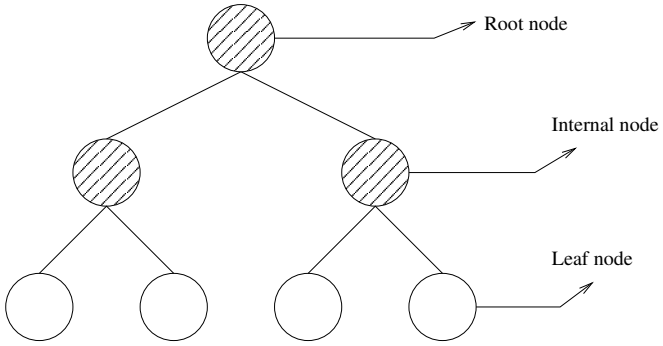


FIGURE 1.2. A schematic view of a simple tree data structure.

of the leaf nodes (the nodes which do not have children) contain the data elements stored there. Typically, the nodes in an R-tree contain between $m \leq M$ children or elements, with m and M fixed beforehand. Also, all data elements are stored in the leaf nodes and all leaf nodes are on the same tree level.

One can intuitively understand that when MBRs do not overlap, searching a given data element will cause a search algorithm to follow a single path from the tree root (the top node) to a leaf node; hence taking only logarithmic time. One also can immediately see that the MBR structure is solely determined by which data elements are stored in which leaf nodes. Hence the true difficulty with R-trees is finding a method which constructs the most favourable structure with respect to the MBR overlap and other MBR properties.

We will see that many different R-tree variations exists. We shall investigate some of them, and attempt to find an R-tree which has excellent query efficiency for problems of the type we will introduce in the next section. At this point it seems prudent to conclude with this intuitive introduction of R-trees and leave the details and more exact definitions for Chapter 2. We will first proceed with explaining more about the context in which we wish to apply these R-trees.

1.4. Context

We will now give a small introduction to the context in which Shell looks to apply the R-tree data structures.

1.4.1. Oil Reservoirs. Consider an oil field, and assume one wants to extract oil from them. This is usually done by drilling *wells*, which of course must be put at strategic locations so that extracting oil can be done as efficiently as possible. Two things are needed for this to succeed; firstly, a rather good "map" of the oil field must be available, and secondly, one has to be able to predict the effect of drilling somewhere, before actually doing so. The latter because drilling a well is not an endeavour without cost; these are multi-million euro operations. Also, drilled wells may influence the oil reservoir; possibly in such a way making it harder to efficiently extract oil.

1.4.2. Simulation. The above is one of the reasons why oil companies such as Shell are willing to invest in developing simulation software to predict what would happen to an oil reservoir given various scenarios; indeed, even if such a simulation would prevent only a single unnecessary test drill from happening, millions would be saved already.

The area of fluid dynamics provides us with a set of differential equations which enables us to model fluid flows through porous media. Obviously, in our case we are interested in only the fluid oil which flows through the many different ground layers which make up the oil reservoir. For the model to work accurately, the properties of oil and those of the different ground layers in the oil reservoir have to be determined as precisely as possible. After a model has been set up

for the oil reservoir in question, artificial changes can be made and the model can be solved accordingly; thus enabling us to predict the oil flow in both the original situation and other hypothetical situations.

At the core of simulation software, mathematical models such as the Finite Differences Method (FDM), the Finite Element Method (FEM) or the Finite Volumes Method (FVM) are being used to solve these models. The software package used at Shell makes use of the latter, the FV method. Since this is only meant as an introductory section about the simulation software, we will not discuss this method in detail; see for example [**BO**] for details on this method.

FVM is known to work well for problems which obey some law of conservation; in our case, the flow of oil entering a cube of oil reservoir ground is equal to the flow exiting from an adjacent cube. Also, FVM can be easily applied to *unstructured grids*; why this is a good advantage in our case will become apparent later on.

1.4.3. Current Software Structure. For the purpose of simulating oil reservoirs, Shell has developed a package called *Dynamo*. Dynamo is a highly modular package capable of reading in external reservoir field data and obtaining a discretisation of it, upscaling (or *coarsening*) this grid and finally performing a wide range of simulations. During each step, the user is able to modify a great amount of settings to fine-tune the resulting oil reservoir model and perform the precise simulations they are interested in. These three steps are examined in greater depth below.

- Discretisation; *Voxel Grid extraction*
- Upscaling; *Reduce++*
- Simulation; *Modular Reservoir Simulator (MoReS)*

Voxel Grid extraction. In oil reservoir simulation, we discretise a certain volume; this volume consists of porous rock, containing water, gases and oil. Generally, the reservoir area is divided into beam-like subareas to which we will refer to as *grid elements*. For each grid element, ground properties such as porosity, saturation, permeability[4], overall ground type (or *facies*) classification[5], are determined. The procedure of obtaining a suitable discretisation and getting the required data from the reservoir field data, is the first phase in preparation for the actual simulation. We will refer to this as the voxel grid extraction phase.

Consider a cross-section of an oil reservoir field as in figure 1.3(a). The different colours denote different kinds of ground layers the ground is built up from. These different ground layers have different ground properties, which directly influences the flow of liquids such as oil, water or gas. To determine a discretisation, virtual pillars are inserted at strategic positions in the virtual representation of the oil reservoir. Along the edge of these pillars, we set *ticks* at least at the places where two ground layers come together. A pillar is modelled using splines, and can either be straight (linear) or be bended (quadratic, or listric).

Assume that our cross-section has pillars inserted at regular intervals, as in figure 1.3(b). Now, according to the ticks on these pillars, we can define a two-dimensional discretisation in a rectangular-like grid based upon ground layer similarity; we create rectangular grid elements of which the area shares as much ground layer similarity as can be expected. This is illustrated in figure 1.3(c). Note that in reality, this all happens in three dimensions and thus we obtain beam-like (three-dimensional hyperrectangular) grid elements; see figure 1.1.4.

As can be seen in the figures, some pillars have less ticks on them than others, thus resulting into non-existent grid elements, also called *void blocks*. This happens frequently around places where ground layers are fractured due to planar movement; they are also called *fault lines*. Void

[4] A permeability coefficient is stored for each direction along any of the three (x-, y-, or z-) axis. Axis alignment is derived from the input data.

[5] I.e.; sand, clay, rock, et cetera.

blocks can also be manually defined on otherwise normal grid elements, when we are not interested in calculating the solution for that block.

As a consequence of this method of voxel grid extraction, the grid elements and their properties are stored in an *three-dimensional matrix* (when working on three-dimensional problems), with corresponding grid blocks that are generally *not localised* in space. Let us clarify this again using the two-dimensional example. In the discretisation figure 1.3(c), we may denote the upper-left rectangle with the coordinates $(0, 0)$. The rectangle on the right of that one will be denoted by $(1, 0)$ et cetera, such that the bottom-right rectangle is denoted by $(6, 3)$. Let us assume we can store all relevant grid element data such as their location and ground properties in a single data element E_{ij}, with i, j corresponding to the rectangle coordinates as introduced above. We thus have defined a matrix E containing all grid-element data, or in other words, *E contains the voxel grid*.

This three-dimensional matrix E is also referred to as the ijk-grid. Hence, although there is some logic in the order at which the rectangular grid elements are indexed in this ijk-grid, there is no guarantee that elements at neighbouring indexes are neighbours in the spatial sense as well; see figure 1.3(c) around the fault lines, for example. The spatial representation of an ijk-grid is also called the *Paleo domain*.

This lack of ordering initially results in an $O(n)$ complexity[6] when searching for a grid element with specific spatial properties, with n is the total number of grid elements. Therefore, it is desirable that the individual grid elements are pre-processed and stored in a memory and query efficient datastructure to speed up the many queries that must be executed during the actual simulation run.

Note that the procedure of determining a discretisation is not done by Shell itself; external companies are responsible for supplying grid data. For example, via the software application *Petrel*[7].

Reduce++. Reduce++ is software package part of Dynamo developed at Shell which reads in the fine voxel grid, described above. Those grid elements are coarsened and conditioned in preparation of being passed on to the finite volume (FV) solver.

Coarsening, put simply, is the practise of combining several small beam-like grid elements into a single beam-like element. Of course, the properties of the individual grid elements need to somehow be combined to form the properties of the resulting beam element.

Several areas of the discretisation of an oil reservoir can be coarsened when one is not interested in oil flow details there. This is for instance useful when considering to drill a well somewhere; one would be interested primarily in what happens to the oil flow around that well, and not particularly in what happens to the oil a few kilometres farther away, for example.

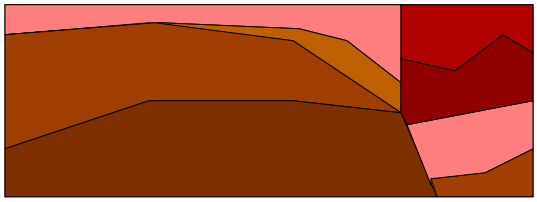
MoReS. MoReS is software, also part of Dynamo, which does the actual oil reservoir simulation. It uses the grid element output of Reduce++ on which specialised FV is applied. This FV application is repeated for each time step for which a simulation is required.

Note that the precision of the solution this solver returns depends primarily on the discretisation quality; apart from the initial grid measurement uncertainties from the first phase, the second phase may very well amplify those uncertainties to too large levels when one is not careful. The point is these few steps are sensitive and errors may easily spoil the end result.

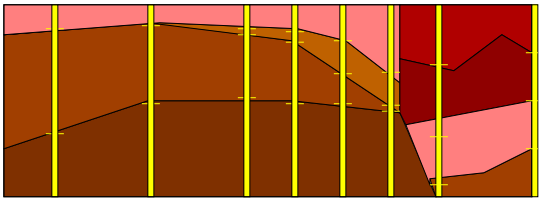
1.4.4. Queries. The extraction phase comes up with an initial grid, and both the upscaling and simulation phase need to extract information (by *querying*) from that grid, and possibly modify that grid.

[6] On complexity and the O-symbol, refer to section 3.1.

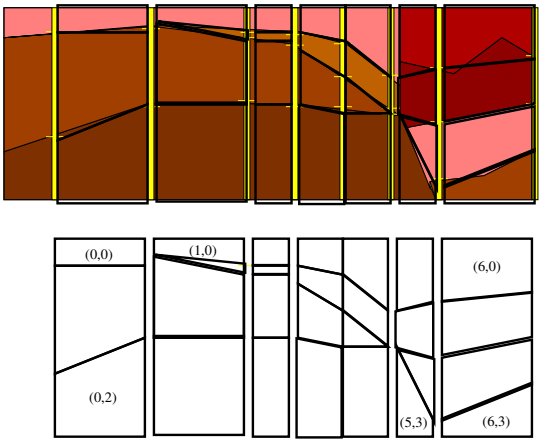
[7] See `http://www.slb.com/petrel`.



(a) An example cross-section of an oil-reservoir.



(b) An example cross-section of an oil-reservoir with measurement pillars inserted.



(c) Example of discretisation according to ground layer data.

FIGURE 1.3. Simplified illustrations regarding voxel grid extraction.

If we extract grid elements with certain properties from the data structure, we say we have performed a certain *query* on the data structure. The queries that certainly will be applied in simulation are the following.

- Point containment
- Line intersection
- Box intersection

Point query. Point queries take a single point in 3D and finds and returns all grid elements containing this point. Such a query would be useful for determining the ground properties at a given point, for example.

Line query. A line query finds and returns all grid elements which intersect a line segment given by two points in space. This query is for example useful when one is interested which grid elements a certain well would intersect. Actually, a well may be curved, so instead of doing a single line query for a well, one approximates the well path by a polyline consisting of multiple line segments and executes a query for each line segment.

Box query. A box is defined by two points which denote two opposite cornerpoints of a box. All grid elements falling within, or intersecting with this box will be returned by this query. This is for instance useful when one wants to visualise only part of the entire grid.

Some more exotic queries which are not the queries we are primarily interested in, but which will be investigated later on, are the following.

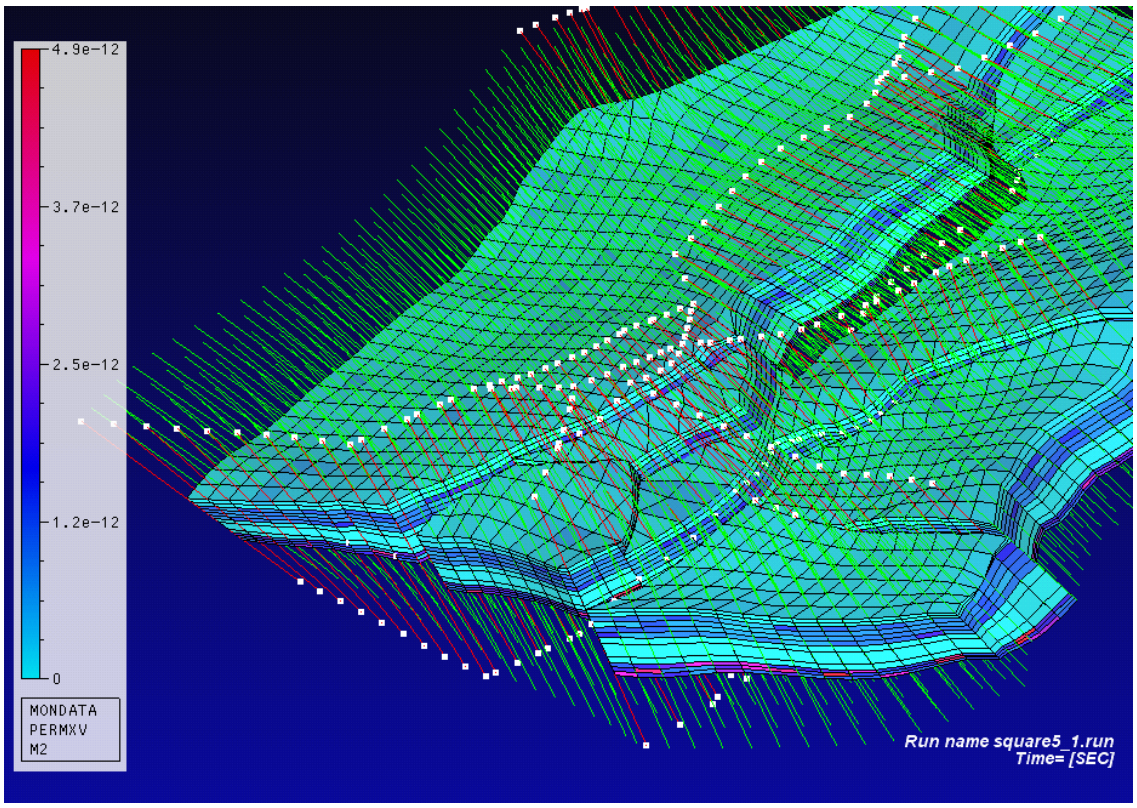


FIGURE 1.4. A visualisation of a grid used in simulation at Shell.

- Nearest neighbours
- Plane intersection

Nearest neighbours. The nearest neighbourhood query takes as input a single point, and firstly finds a grid element containing that point. Alternatively, one could also supply this query with an initial grid element. The query than proceeds with finding and returning the k nearest neighbours of this grid element, where k is variable. This query can be useful for the FVM solver; the method employed calculates solutions by calculating the flow of oil from one adjacent grid element to another. Adjacency of two elements implies they are neighbours, hence justifying a need for this type of query. Note that alternatively, one could use a MBR slightly larger than the grid element to query on to find its neighbours.

Plane intersection. As an extension to the line query, we could also implement a plane query which finds and returns all grid elements intersecting with a given plane. This type of query is useful when visualising only a slice of the entire grid, for example.

We assume there is no pre-defined systematic way of querying; i.e., the queries to-be done are more or less *random*. If there was a systematic way of querying, it would have been possible to design a highly efficient tailored data structure. Since this is not the case, we will end up with more general data strucures which can be used in many other applications. When doing experiments however (Chapter 4), we only consider Shell's context described above.

CHAPTER

2

TWO

R-tree data structures

To solve the problem of achieving a good query time on the grid elements, we will research the so-called *R-tree data structure* and compare it to the performance of a tree built using a *bisection algorithm*[1]. The latter structure is used by Shell at the time of this project initiation. Since both solutions make use of the concept of a tree, we will start our theoretical review there.

2.1. Definition of a Tree

Many data structures on which searching is an important operation, are based on trees. Before going into detail about specific (R-)trees, we will first properly define the notion of a general tree.[2] Since a tree can easily be defined by using graphs, we will define those first.

2.1.1. Graphs. We will start with defining the notion of an *(undirected) graph*. An example of such a graph is given in figure 2.1(a).

DEFINITION 2.1 (Graph). *An graph G is given by a pair (V, E), where the set V consists of vertexes and the set $E \subset V \times V$ consists of edges. A single edge $e \in E$ defines an connection between two vertexes $v, w \in V$; hence $e = (v, w) = (w, v)$.*

DEFINITION 2.2 (Path). *For some undirected graph $G = (V, E)$, a path p of length $k \in \mathbb{Z}, k \geq 1$ from a vertex $v_1 \in V$ to a vertex $v_k \in V$ is given by a sequence of vertexes as follows:*

$$p =< v_1, v_2, \ldots, v_{k+1} >, v_i \in V$$

such that for all $i \in [1, k] \subset \mathbb{Z}$ we have that $(v_i, v_{i+1}) \in E$.

Note that as such, $< v_1, v_2 >, v_1 = v_2$ actually could denote a path of length 1 as long as $(v_1, v_2) \in E$.

When for each pair of vertexes $v, w \in V$ there exists a path which starts at v and ends at w, we say the graph G is *connected*. When there exists a path $p =< v_1, v_2, \ldots, v_k, v_{k+1} >$ such that $v_1 = v_{k+1}$ and $v_2 \neq v_k$ if $k > 2$, we say that p forms a *cycle* and as such G is a *cyclic* graph. If no such path exists, we say G is *acyclic*. Figure 2.1(b) shows an undirected, connected, cyclic graph.

[1] See section 2.2.

[2] The following line of definitions is adapted from [CLRS03, Appendix B.4 and B.5].

2.1.2. Free Tree. An undirected, connected, acyclic graph is also called a *free tree*. This is called this way because such graphs consists of many edges which branch into more edges at given vertexes; just like branches in a tree. A free tree is shown in figure 2.1(c).

THEOREM 2.3 (Unique Paths in a Free Tree). *Let $F = (V, E)$ be a free tree. Then, for each pair $v, w \in V$ there exists an unique path p starting at v and ending at w.*

PROOF. Firstly note that a free tree is a connected tree; therefore, there is at least one path connecting v to w. We will assume two such paths exist and show a contradiction.

Let $p^1 =< v, v_1^1, v_2^1, \ldots, v_k^1, w >$ be the first path connecting v to w, and let $p^2 =< v, v_1^2, v_2^2, \ldots, v_k^2, w >$ be the second path. Let there exist at least one i such that $v_i^1 \neq v_i^2$ so that the paths are not equal (i.e., $p^1 \neq p^2$). As such, there exists an index j such that:

$$j = \arg\min_{1 \leq i \leq k} v_i^1 \neq v_i^2$$

Then the vertex $d = v_j$ is the vertex at which both paths diverge.

Since both p^1 and p^2 define a path between a common start and ending point, similarly, there also exists a vertex c at which the two paths converge again. Note that we thus have found two completely different paths $\hat{p}^1$ and $\hat{p}^2$ connecting vertexes c and d. If we connect those two paths, the resulting path defines a cycle, while a free tree contains no cycle; contradiction.

Hence there cannot exist two different paths in a free tree connecting the same vertexes v and w, and thus there exists precisely one such path. □

2.1.3. Rooted Trees. There is one dissimilarity between a free tree and a tree "in the real world"; a free tree has no *root* from which the rest of the tree grows. Given this, we now define a *rooted tree*.

DEFINITION 2.4 (Rooted Tree). *A rooted tree R is a free tree (V, E) from which one of the vertexes in V is distinguished from all other vertexes; i.e., $R = (r, V, E)$ for some $r \in V$. r is called the* root *of R.*

Note that as such, any free tree can be made a rooted tree. Vertexes in a rooted tree R are commonly called *nodes*; hence, the root is a special node. Also, since a rooted tree is a special case of a free tree, Theorem 2.3 applies and we may state that there exists exactly one path from the root node to each other node in V.

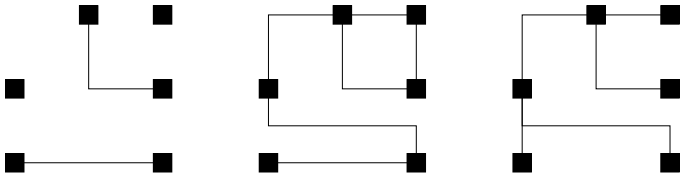
Each node on the path from an arbitrary node v in V to r is called a *ancestor* of v, and on the other hand, if some node v is an ancestor of w, then w is a *descendant* of v. The first vertex p on the path from a vertex $v \in V$ to r (for which $p \neq v$) is called the *parent* of v. Then also, we may define v to be a *child* of p. When two nodes $v, w \in V$ have the same parent p, then v and w are *siblings*.

A node which does not have children is called a *leaf* or *external* node, while others are *non-leaf* or simply *internal* nodes. For some $v \in V$, the number of nodes - excluding v - which lie on a path from v to r is called the *depth* of v. The depth of the deepest node v ($v = \arg\max_{w \in V} \text{depth}(w)$) is called the *height* of the tree. If all leaf nodes have the same depth, we say the tree is *height-balanced*.

Following the definition of a rooted tree, we may also define the notion of a subtree as follows.

DEFINITION 2.5 (Subtree). *Let $R = (r, V, E)$ be some rooted tree, and v be an arbitrary node contained in that tree. Let V' contain all descendants of v, including v itself. E' is a subset of E containing only the edges between nodes in V'. Then the tree $S = (v, V', E')$ is a* subtree *of R with root v.*

NOTATION 2.6 (Subtree). *When $R = (r, V, E)$ is a rooted tree and $v \in V, v \neq r$ a non-root node in R, then we denote the subtree of R with root v by R_v.*



(a) An undirected, unconnected, acyclic graph.

(b) An undirected, connected, cyclic graph.

(c) An undirected, connected, acyclic graph. Also called a free tree.

FIGURE 2.1. Various graph illustrations.

2.1.4. Searching through a tree. We can store data objects throughout a tree structure, at every node or only at the leaf nodes. If furthermore we can add a kind of indexing to each node, we may be able to search for a specific object in the tree without having to inspect all nodes. This is the main reason to implement data storage using a tree-like structure.

If we assume that the indexing mechanism can indicate exactly at which child of a given node the requested element should be, then the searching algorithm only has to construct a single path from the root to the node which contains the requested element. Searching time then is of an order equal to the depth of the node storing the element.

This already indicates an important aspect in data structures based on trees; we wish to keep the height of a tree as small as possible, to improve searching time. Now that we have this basic knowledge about trees, we will review some possible tree-based data structures which we can use within the reservoir simulation software.

2.2. Bisection Tree

Suppose we have some collection of polygons we wish to efficiently store in a tree. Efficient, in this case, means searching times should be as short as possible while memory usage remains low. Let us then first investigate how we should search a tree which stores spatial data. We first make some assumptions about such a *spatial tree*.

2.2.1. Spatial Tree.

DEFINITION 2.7 (d-dimensional Spatial Tree). *$S = (r, V, E, F)$ is a d-dimensional spatial tree if $R = (r, V, E)$ is a rooted tree (as in Definition 2.4) and when for each node $v \in V$ there exists some set v_O containing d-dimensional objects. Also, for each such node, we have a function $f_v : \mathbb{R}^d \to \{0, 1\}$, which indicates if the subtree $S_v = (v, V', E')$ may store[3] objects containing some given point $x \in \mathbb{R}^d$;*

$$f_v(x) = \begin{cases} 0, & \text{if } \forall \mu \in V' \quad \text{we surely have that } \nexists X \in \mu_O \text{ such that } x \in X \\ 1, & \text{otherwise} \end{cases}$$

Hence, for each $v \in V$ we have an $f_v \in F$.

Now, a generic search algorithm may look like Algorithm 2.1. Analysing this algorithm gives us an upper bound for the time required to execute it (see Section 3.4.1):

[3]It could be that there does not exist any object in the data structure satisfying the query. The point is we do not know for certain until the leaf nodes are reached.

Algorithm 2.1 Generic Search Algorithm on a Spatial Tree.

A Generic Search Algorithm which attempts returns all objects in the (sub)tree with a given root node v containing some given point $x \in \mathbb{R}^d$, where d is the dimension of the spatial tree.

GSA(v, x):

```
1: Let S be an empty set of objects.
2: for each object o ∈ v_O do
3:    if o contains x then
4:       Add o to S.
5:    end if
6: end for
7: for each child node c of v do
8:    if f_c(x) == 1 then
9:       Let S = S ∪ GSA(c, x).
10:   end if
11: end for
12: return S.
```

$$T(r) = O(a + b + \sum_{v \in r_C} f_v(x)T(v)), \tag{2.1}$$

where a is the maximum number of elements stored at each node, b the maximum number of children, r the root node and r_C the children of the root. Thus, we see that the best results are achieved when the number of recursions is limited. This can be achieved by any of the following:

(1) $\cap_{u \in v_C}\{x \in \mathbb{R}^d | f_u(x) = 1\} = \emptyset$, for any $v \in V$ (as mentioned earlier in Section 2.1.4).
(2) The tree height is small (also see Section 2.1.4) (thus limiting recursion depth).
(3) The maximum number of children (b) is small (thus limiting recursion breadth).

(1) is the case when there is *no point overlap* between sibling nodes; hence, when all objects containing a point x are stored within the same node. With regards to (2), the tree height h is smallest (under constant b) when $h = \frac{\log n}{\log b}$, where n is the total number of elements stored. Note that this requires that each node (possibly excluding a single node) would have exactly b children; the tree should be *fully packed*. Furthermore, a larger b may directly result in a smaller tree (in terms of height) when it is packed.

If there is point overlap between sibling nodes, item (3) quickly becomes a large issue. However, note that a small b conflicts with demand (2); having many children would indeed reduce tree height, but in case of bad overlap properties, the GSA may recurse over too many children and again spoil the searching time. In conclusion, when building a spatial tree, one has to carefully take into account these three basic properties. In practise, b is chosen so that algorithms like the GSA can perform their operations as much as possible in cache memory.

2.2.2. Bisection. The bisection algorithm is inspired by the following Ham sandwich theorem[4]. The following representation of this theorem is adapted from [CMed].

THEOREM 2.8 (Ham sandwich theorem). *Let there be given any d finite measures, $\mu_1, \mu_2, \ldots, \mu_d$ defined on the Borel subsets $\mathbb{R}^d$. If each μ_i is absolutely continuous with respect to Lebesgue measure, there exists a hyperplane which simultaneously bisects each measure.*

PROOF. For an outline, see [CMed, p. 74]. For the original proof, see [ST42]. □

[4] or also called the Stone-Tukey theorem, [ST42].

This theorem only works for continuous measures. However, [CMed] proves a following similar theorem for general measures. We adapt the following theorem from that text:

THEOREM 2.9 (Generalised ham sandwich theorem). *Given any d finite measures $\mu_1, \mu_2, \ldots, \mu_d$ on the Borel sets of $\mathbb{R}^d$. Then, there exists a hyperplane $h = \{x \in \mathbb{R}^d | x \cdot v = c\}$ with $v \in \mathbb{S}^{d-1}$, with $\mathbb{S}^{d-1}$ a $d-1$-dimensional unit sphere, such that for all $1 \leq i \leq d$:*

$$\mu_i(h^-) \leq \frac{\mu_i(\mathbb{R}^d)}{2}, \quad \mu_i(h^+) \leq \frac{\mu_i(\mathbb{R}^d)}{2} \tag{2.2}$$

with $h^- = \{x \in \mathbb{R}^d | x \cdot v < c\}$ and $h^+ = \{x \in \mathbb{R}^d | x \cdot v > c\}$.

PROOF. See [CMed]. □

Now suppose we have some finite subset $W \subset \mathbb{R}^2$, which contains n 2-dimensional points $o_i \in \mathbb{R}^2$, $1 \leq i \leq n$. Let us define $\mu(X) : X \subset W \to \mathbb{R} : \#\{X \cap W\}$. We easily see that $\mu(\emptyset) = 0$. Also, for any two disjoint subsets A and B from W, we have that $\forall o_i \in W$, if $o_i \in A$, then $o_i \notin B$ and vice versa. Thus $\mu(A \cup B) = \mu(A) + \mu(B)$; μ is countably additive for any countable sequence of disjoint subsets from W. Hence μ is a valid (non-continuous) measure on W, and we may apply Theorem 2.9 with only the single measure μ.

Therefore, we can bisect any object W containing n points into two smaller objects containing at most $\lceil n/2 \rceil$ points by using a straight line (in case of an odd number n, the hyperplane must cut precisely through a single point). Note that while Theorem 2.9 gives us proof of the existence of a splitting hyperplane which achieves this result, it does not provide us with an algorithm to actually find the desired split. Ham sandwich algorithms which split point clouds in W exist and work in $O(n)$ worst case time [LMS94].

The idea of using this bisection concept with respect to building a spatial tree is as follows. Let W correspond to the root of a spatial tree, and let the two halves of W after bisection correspond to children of that root. If the number of points was odd, we also add a third child for storing the cut-through point; this point then is excluded from the other two children. Hence, the constructed tree has two or three children corresponding to non-overlapping subsets of W; see figure 2.2. This principle can be recursively applied on the individual children nodes, with the exception of the third child, if it exists (figure 2.3). In our application of course, instead of using d-dimensional *points*, we would like to use d-dimensional *objects*.

Note however that we cannot use Theorem 2.8 to claim that we can bisect some d-dimensional subset W containing n d-dimensional smaller *objects* into two areas containing $\lceil n/2 \rceil$ objects. This is because $\mu(X) : X \subset W \to \mathbb{R} : \#\{o_i \in W | o_i \in X\}$ no longer defines a measure; an object o_i may be contained two disjoint areas of W and as such, μ no longer is additive. See figure 2.4.

To elude this problem, we can try and represent objects by a single point, like the centre coordinate of a rectangle, or the left-bottom cornerpoint. This may however lead to bad results in some cases. Consider what would happen if the objects to-be added were stretched to irregular lengths, for example; see figure 2.5. Notice how the rectangles seem to randomly intersect through the four areas resulting from the bisected method. The objects do not restrain themselves to a single area at all, which makes searching less efficient.

We can however try to define a different measure, such as $\mu(X) = \sum_{o \in X} \text{Volume}(o \cap W)$. Applying the ham-sandwich theorem with respect to this measure, we see it is possible to get two disjoint areas of W containing an equal amount of object *volume*. This allows for the bisection plane to cut through multiple objects, in effect causing the 'third child' discussed earlier to contain an amount of objects ranging from 0 to n.

In any case, the basic principle has been set and we generalise for a bisection-based tree data structure for d-dimensional objects in the pseudo-Algorithm A.2.1. Bisection based, in the sense

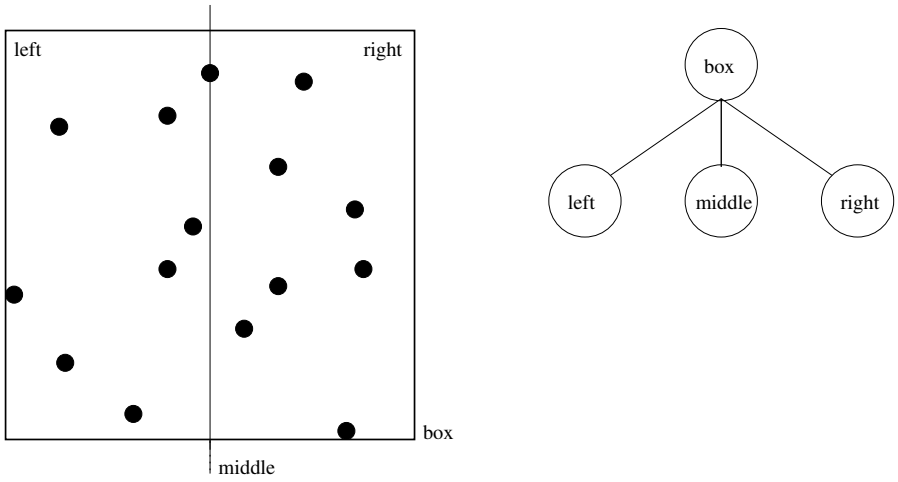


FIGURE 2.2. A bisection tree with height one on a subset containing a finite number of points.

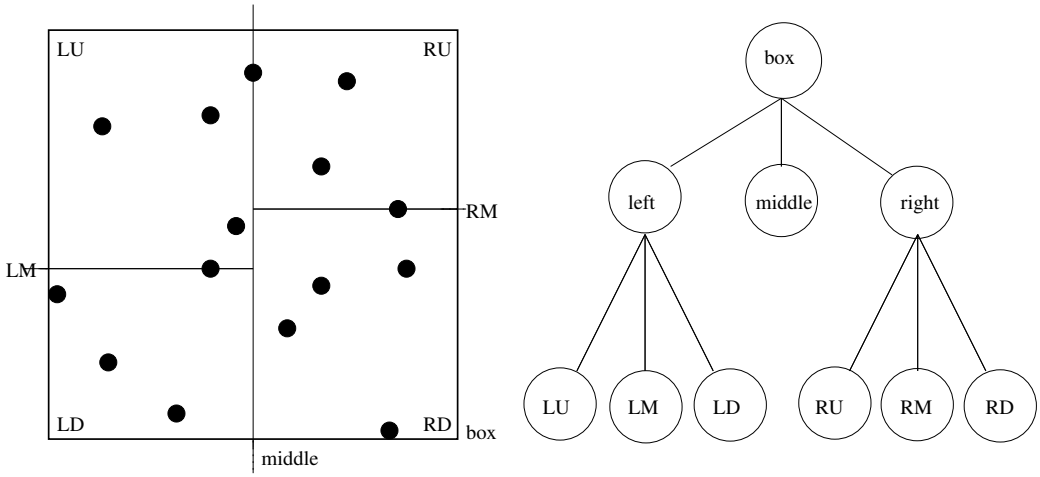


FIGURE 2.3. A bisection tree with height two on a subset containing a finite number of points.

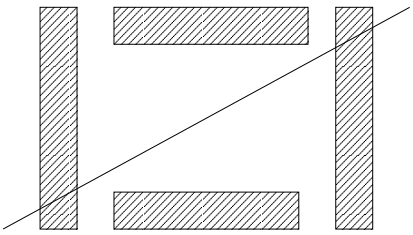


FIGURE 2.4. Taking the number of objects as measure, we see that the upper subregion of the figure counts 3 objects, and the lower one also 3. This sums to 6 while the total number of objects is 4; the measure is not additive.

that we use a hyperline of dimension $d-1$ to split the object sets. See figure 2.6 for an illustration. The Shell bisection method does not necessarily demand that space be bisected in such a way that the number of object in each subspace is halved; it simply halves the space on a single dimension.

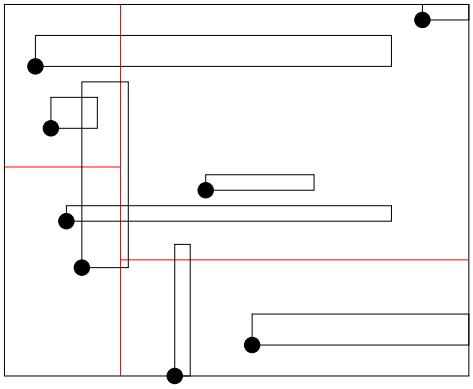


FIGURE 2.5. Bisection method applied to rectangles while only considering their bottom-left cornerpoint.

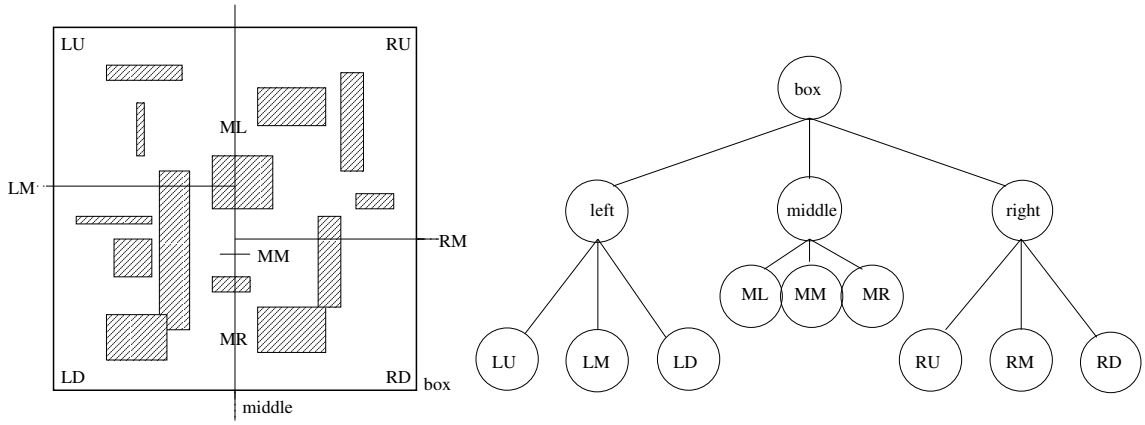


FIGURE 2.6. A bisection tree on a subset containing a finite number of two-dimensional objects.

Since the number of iterations done in the bisection algorithm A.2.1 directly controls the resulting tree height, we can explicitly choose the resulting tree height h. Typically, one chooses $h = \log n$ and we achieve a $O(n \log n)$ running time for both construction of and worst case GSA-querying of a Bisection tree (Theorem 3.5 and 3.8, section 3.4.1).

The bisection software developed at Shell constructs hyperplanes which are always perpendicular to some axis; for example, in the two dimensional case, the first hyperplane would be chosen perpendicular to the x-axis, the second along the y-axis, the third yet again along the x-axis, et cetera; it thus does not guarantee a well-structured bisection tree, since the number of objects at each side of the line may not be distributed evenly.

2.3. R-Tree

Datastructures which may give better results are the so called R-trees. In general, the construction of these trees uses the spatial information of the objects which we want to store; this in contrast to blindly partitioning the space as the bisection tree does. Also, we have a potential of using more than three children at each node, which may or may not further improve potential performance.

2.3.1. Minimum Bounding Rectangle (MBR). The idea of R-trees is to impose ordering on higher dimensional objects by using an ordering on minimum bounding rectangles. MBRs can

easily be defined on any d-dimensional object o using intervals. For readability, this text assumes the following non-standard notation of an interval.

NOTATION 2.10 (Interval). *An interval $[a, b]$, $a, b \in \mathbb{R}$ is defined as:*

$$[a, b] = \{a + t(b - a) | \forall t \in \mathbb{R}, 0 \leq t \leq 1\}. \tag{2.3}$$

DEFINITION 2.11 (Minimum Bounding (Hyper-)Rectangle). *Let b_i^- and b_i^+ denote the infimum and supremum values of $o \in \mathbb{R}^d$ on the i^{th} dimension ($i \in [1, d]$). Then the MBR of o, denoted $MBR(o)$, is given by:*

$$\begin{aligned} MBR(o) &= [b_1^-, b_1^+] \times [b_2^-, b_2^+] \times \cdots \times [b_d^-, b_d^+] \\ &= \Pi_{i=1}^d [b_i^-, b_i^+] \end{aligned} \tag{2.4}$$

2.3.1.1. *Operations on MBRs.* Several operations on minimum bounding rectangles will be used throughout this text. We will define them explicitly here.

DEFINITION 2.12 (Union of MBRs). *Let x and y be two MBRs as in Definition 2.11, and let their infimum and supremum values be given by b^{X-}, b^{Y-} and b^{X+}, b^{Y+} respectively. Then:*

$$x \cup y = \Pi_{i=1}^d [b_i^{X-}, b_i^{X+}] \cup [b^{Y-}, b^{Y+}] \tag{2.5}$$

DEFINITION 2.13 (Intersection of MBRs). *Similarly to Definition 2.12, we define the intersection:*

$$x \cap y = \Pi_{i=1}^d [b_i^{X-}, b_i^{X+}] \cap [b^{Y-}, b^{Y+}] \tag{2.6}$$

Note that if there exists an $i \in \mathbb{N}$ so that $[b_i^{X-}, b_i^{X+}] \cap [b^{Y-}, b^{Y+}] = \emptyset$, then $x \cap y = \emptyset$. We say that a bounding box x does not intersect another bounding box y when $x \cap y = \emptyset$.

2.3.2. R-tree Definition. The R-tree has the following six defining properties [Gut84, p. 48].

- Every leaf node contains between m and M objects.
- At each leaf node, for each object stored there, an MBR containing that object is stored.
- Every non-leaf node has between m and M children unless it is the root[5].
- At each non-leaf node, for each of its children, an MBR which contains all MBRs of that child is stored.
- The root node has at least two children unless it is a leaf node[6].
- All leaves have the same depth.

It is clear from these properties that the R-tree is a height-balanced tree. An R-tree of maximum height is achieved when all nodes contain their minimum amount of children m, hence

$$h \leq \lceil \log_m n \rceil - 1, \tag{2.7}$$

where n is the number of objects stored in the tree. An R-tree of minimum height is somewhat less interesting, but is attained when all nodes contain the maximum amount of children

$$h \geq \lceil \log_M n \rceil - 1. \tag{2.8}$$

[5]We still wish to be able to let the tree store less than m objects *in total*.

[6]Having only one child would indicate that child could have been the root node as well; hence the root node would be unnecessary, in all respects.

Note that all objects are stored in the leaf nodes; hence search queries are expected to take $O(h) = O(\log_M n)$ time, when the MBR at sibling nodes have sufficiently low overlap. In the line of previous definitions, the R-tree is more precisely defined as follows.

DEFINITION 2.14 (R-tree). *Let $S = (r, V, E, F)$ be a spatial tree as in Definition 2.7. For a node $v \in V$, let v_C be the set of children of v, and we denote the number of children of v by $|v_C|$. The set of d-dimensional objects stored at v is denoted by v_O, the number of objects stored is denoted $|v_O|$. S is a R-tree if:*

(1) *For each leaf node $v \in V$, unless $v = r$, the set v_O is non-empty; $m \leq |v_O| \leq M$.*
(2) *If the root r is non-leaf, $2 \leq |r_C| \leq M$.*
(3) *For each leaf node $v \in V$, there is stored $v_{MBR} \equiv MBR(v) \equiv \cup_{o \in v_O} MBR(o)$.*
(4) *For each non-leaf node $v \in V, v \neq r, m \leq |v_C| \leq M$.*
(5) *For each non-leaf node $v \in V$, there exists a $v_{MBR} \equiv MBR(v) \equiv MBR(\cup_{v \in v_C})$.*
(6) *For each non-leaf node $v \in V$, the set v_O is empty.*
(7) *For all leaf nodes $v, w \in V$, the depth d_v of v equals the depth d_w of w.*
(8) *For each node $f_v \in F$, $f_v(x) = 1$ iff x is contained in v_{MBR}, and $f_v(x) = 0$ otherwise.*

We will now describe the object insertion, deletion and query algorithms in greater detail. The algorithms are given in pseudocode in Appendix A.

2.3.3. Insertion. Let us firstly consider insertion operations. Suppose we have an object o which is to be added to a R-tree R. We first construct $o_{\text{MBR}} = \text{MBR}(o)$. We next find a leaf v in R for which its MBR v_{MBR} has the greatest overlap with o_{MBR}, in comparison to all other leaf nodes in R. This leaf v is the leaf at which o will be stored.

When the number of children $|v_C|$ at v is below $M - 1$, then simply adding o to v poses no problem. If $|v_C| = M$ however, there is no more room to add o. Common practise then is to somehow split v into two nodes, and insert o into one of them.

Note that splitting one node into two may cause the parent node to become overflowed as well; in this case, splitting is recursively applied on the parent. If the root node overflows, it is splitted as usual and a new root node containing the newly splitted nodes is created. Also note that although parent nodes may be recursively visited by the splitting algorithm, no sibling nodes may be called. Hence the splitting recursion only 'bubbles upward', and does not cause a complete reconstruction. In pseudocode, the insertion procedure is as in Algorithm A.3.1.

2.3.4. Splitting. Adding objects one-at-a-time in a given R-tree using the insertion algorithm is the common way of building dynamic R-trees. When adding objects, the insertion algorithm enforces the R-tree characteristics by splitting when necessary. Splitting has to be done with optimising expected query performance in mind; this can be done in various ways.

In the paper [**Gut84**] introducing the R-trees, three node-splitting variations were proposed, named exponential, quadratic, and linear split. The algorithms for splitting are given in detail in Algorithms A.3.4, A.3.5, and A.3.6. Each splitting strategy is explained below.

2.3.4.1. *Exponential split.* Splitting causes one node v to split in two nodes v_1 and v_2. These two nodes will have their own two MBRs $v_{1\text{MBR}}$ and $v_{2\text{MBR}}$ which will of course differ from the MBR of the original node.

Suppose $|v_C| = k$ and as such, $|v_{1C}| + |v_{2C}| = k + 1$, after insertion. The $k + 1$ objects or child nodes can be distributed over the two children in what would seem like 2^{k+1} different ways, ignoring the fact that each child should have between m and M entries. The exponential split algorithm considers each different partition and selects the partition resulting in the least amount of total MBR volume. Since there are an exponential number of splits, this splitting strategy is expected to run in exponential time. In [**GRLL98b**] however, it is shown that the running time

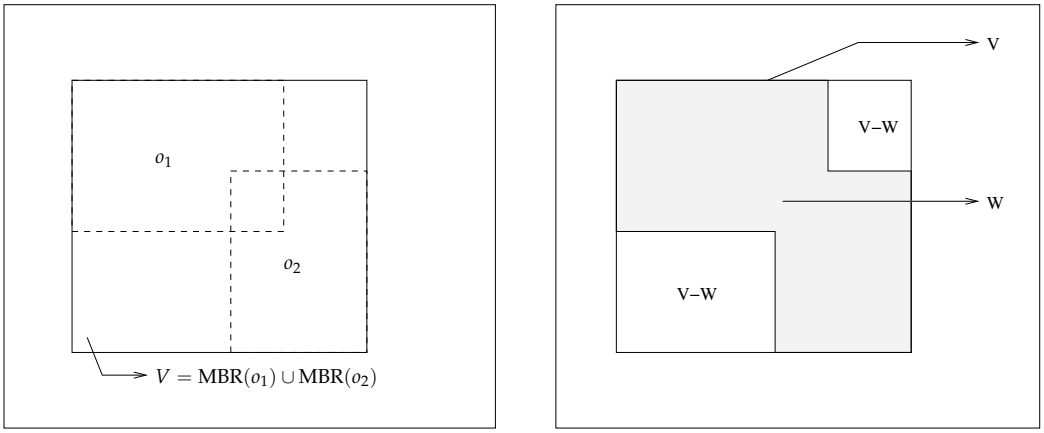


FIGURE 2.7. Illustration of dead space ($V - W$) when constructing a MBR containing two smaller objects o_1 and o_2.

can be brought back to polynomial time due to effects like symmetry. As such, they introduced a polynomial algorithm which is able to find the optimal split.

With respect to searching the R-tree, we would want the MBR to remain as small as possible, with respect to volume; this would minimise the probability that a query would intersect with the current MBR [Gut84, p. 51]. The exponential (or rather, polynomial) split variation chooses the build-up of v_1 and v_2 which results in the smallest total MBR volume of v_1 and v_2. In the current experimentation code, neither the explicit exponential nor polynomial algorithm is implemented.

Note that we considered all splits and came up with a global minimum with respect to the *cost function* $f_C(v_{1\text{MBR}}, v_{2\text{MBR}} v_{2\text{MBR}}) = \text{Volume}(v_{1\text{MBR}}) + \text{Volume}(v_{2\text{MBR}})$. The rationale here is that we minimise the probability a random query will intersect any of these bounding boxes. Other cost functions could be designed and tested, see Section 2.4.2. As an alternative to an exhaustive search as presented here, one could also opt for local search algorithms; this is, however, not the way Guttman proceeded as we will see now.

2.3.4.2. *Quadratic split.* A polynomial running time for dealing with node splitting likely is much too costly. We will now consider a quadratic cost splitting strategy. We firstly introduce a new concept.

DEFINITION 2.15 (Dead space of a MBR). *Let x be a MBR containing n objects or other MBRs $o_1, o_2, \ldots, o_n$. Let V denote the volume of x and let $W = \sum_{i=1}^{n} volume(o_i)$. The* dead space *$D$ is then defined as:*

$$D = V - W$$

Figure 2.7 shows a simple illustration of dead space, in two dimensions using two MBRs o_1 and o_2.

We see that this notion of dead space gives us a measure of how wasteful a given MBR is. The smaller the dead space of a MBR, the better the chance that an intersecting query indeed finds useful results. The quadratic split algorithm which we will describe now, aims to minimise dead space.

The algorithm starts out with finding the two objects o_1, o_2 for which $\text{MBR}(o_1) \cup \text{MBR}(o_2)$ results in the highest amount of dead space. o_1 is then added to v_1 and o_2 to v_2. For all remaining objects o_i, we will consider the effect of adding a single object to either v_1 or v_2 as follows.

Let w_1 denote the area increase of $\text{MBR}(v_1)$ if a o_i was added to v_1, and let w_2 denote the area increase when it was added to v_2 instead. The difference between them is denoted $w = |w_1 - w_2|$. When w is large, adding o_i to the 'wrong' node would have bad consequences.

Hence we will add the o_i for which w had the largest value. That object will of course be added to the node for which its area increase was smallest. This entire process is repeated until all objects are assigned to either v_1 or v_2.

Since this splitting strategy requires to compare all possible parings of two objects to each other in order to determine the combination which results in the largest dead space, this algorithm will take quadratic ($O(n^2)$, see Section 3.1) time, asymptotically. Also, after finding the pair with the largest dead space, we actively search for the object not yet assigned to v_1 or v_2 which could cause the largest dead space; this also takes $O(n^2)$ time.

2.3.4.3. *Linear split.* To reduce the $O(n^2)$ complexity of the Quadratic cost algorithm, we will now modify the selection procedure to run in $O(n)$ time. With respect to finding an initial pair by searching for the pair resulting in the largest dead space, we instead find a pair o_1, o_2 for which their MBRs are as far away from each other as possible. With 'far away' we mean the relative distance of the MBR extreme coordinates in a single dimension. This only requires recording extreme values on each dimension for each object (which are readily available since this is how MBRs are defined), and thus runs in linear asymptotic time for both the number of objects to be split and the number of dimensions. The algorithm then proceeds by randomly selecting a not-added object and adds it to the set which would require the least overall MBR enlargement. Because we only check for a randomly selected object its effect on two MBRs, this part also runs in linear time.

Guttman concluded [**Gut84**, Section 5] that while the linear split algorithm resulted in slightly slower query times, the reduction in efficiency was acceptable with respect to the gain in construction time over the other algorithms. He also noted that the exponential split method was too slow for large datasets. Also worth noting is that the linear split method resulted in the R-tree taking up more memory when compared to the quadratic split method [**Gut84**, Figure 4.10].

2.3.5. Deletion. A method for object deletion is given in Algorithm A.3.3. When an object with a given ID is to be deleted, we assume its MBR is also available. The deletion algorithm then executes a simple search using that MBR information to locate in which leaf node the object is stored. When found, the object is simply deleted from the set of objects stored at the thereby belonging leaf node.

Such a deletion is not trivial however; *underflows* may arise. When the number of children or objects stored drops below the minimum value, we must define some way of underflow handling. In the case of basic R-trees, underflows are handled by simply re-inserting the objects stored at the underflowed leaf. This is done instead of, for example, merging two sibling nodes, because we cannot determine if the objects stored at two siblings should be grouped together; merging blindly may lead to a poor R-tree structure, and thus would result in bad average query times.

Note that this is another manifestation of the non-existent ordering of spatial data; merging is implementable for one-dimensional data [**MNPT06**, p. 12]. This has been done for, for example, B-trees [**CLRS03**] from which R-trees are derived.

2.3.6. Searching. An algorithm for each of the queries presented in Section 1.4.4 is given; for the point-search query, see algorithm A.1.1. The box-containment query can, like the point-containment query, be straightforwardly implemented as in Algorithm A.1.2. An algorithm for the line query is less straightforward. This algorithm requires that we are able to efficiently check whether a line crosses a given MBR; this may be easily checked by use of projection.

PROPOSITION 2.16 ((Finite) Line-Rectangle intersection). *A finite line, or rather, line segment l in d dimensions defined by its starting point $s \in \mathbb{R}^d$ and ending point $e \in \mathbb{R}^d$ does* not *intersect with an MBR as in Definition 2.11, if $\exists i \in [1, 2, \ldots, d]$ so that:*

$$[s_i, e_i] \cap [b_i^-, b_i^+] = \emptyset \tag{2.9}$$

with b_i^- and b_i^+ as in Definition 2.11.

PROOF. By contradiction. Suppose equation 2.9 holds for a given $j \in \mathbb{Z}([1,d])$, while the line l *does* intersect the MBR. Remember that the MBR is defined to hold all points $x \in \mathbb{R}^d$ such that $b_i^- \leq x_i \leq b_i^+, \forall i \in \mathbb{Z}([1,d])$, and thus also for $i = j$. Assuming the MBR is non-empty, there must be at least one such point x_i. This is in direct contradiction with equation 2.9.

Hence if a line intersects the MBR, equation 2.9 cannot hold, for any i. □

Note that while this theorem gives us a way to quickly check when a line does *not* intersect a MBR, it cannot conclusively state when a line *does* intersect. Consider for example Figure 8(a). We therefore also present the following theorem:

THEOREM 2.17 ((Infinite) Line-Rectangle Intersection). *Given a line l (of infinite length) defined as $l : \{x \in \mathbb{R}^d | x = s + t \cdot d, t \in \mathbb{R}\}$, where $d \in \mathbb{R}^d$ is the line* direction *and $s \in \mathbb{R}^d$ is the line* origin. *Let $j \in [1, 2, \ldots, d]$. We define a function $g^j : \mathbb{R} \to \mathbb{R}^d$ as follows:*

$$g^j(x) = s + \frac{x - s_j}{d_j} d \tag{2.10}$$

Then, there exists a j such that either $y_1 = g^j(b_j^-)$ or $y_2 = g^j(b_j^+)$ is on a surface of the MBR, iff l intersects a MBR given by b^- and b^+.

PROOF. Suppose there exists a j such that y_1 or y_2 is on an edge of the MBR. Note that this implies that the line defined by $g^j(x)$ intersect the MBR. Since the starting point of this line is the same as that of l, and since the slopes are identical (d in both cases), g^j and l denote the same line; l intersects the MBR.

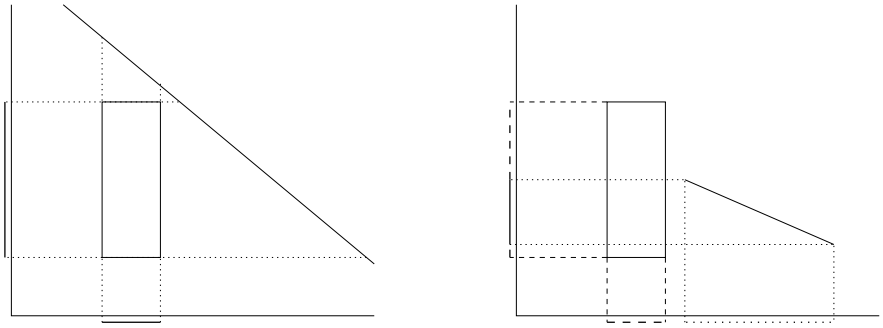
Now suppose we have a line l intersecting the MBR. Then there exists an 'entry point' $\tilde{y}^1$ of the line into the MBR, and an 'exit point' $\tilde{y}^2$. These points certainly lie on two different edges e_1, e_2 of the MBR. Each edge of the MBR is a $d-1$ dimensional surface, fixed at a coordinate b_i^- or b_i^+ on the ith dimension, for some $i \in [1, 2, \ldots, d]$; hence there exist j, k such that $\tilde{y}_j^1 = b_j^-$ or b_j^+ and $\tilde{y}_k^2 = b_k^-$ or b_k^+. Now let us reparametrise the line $l(t) = s + t \cdot d$ to $g^{j,k}(\phi(t)) = s + \frac{t - s_{j,k}}{d_{j,k}} \cdot d$. Then $g^{j,k}(b_j^-) = \tilde{y}^{1,2}$ or $g^{j,k}(b_j^+) = \tilde{y}^{1,2}$. □

Note that this theorem indeed does not work for line segments; consider for example Figure 8(b). However, Theorem 2.17 and Proposition 2.16 together do give a correct algorithm for full line - hyperrectangle intersection detection as follows. Remember that we query using line pieces of finite length. Therefore, we first use proposition 2.16 for quick contradiction detection. If all intervals are covered by the line piece, we consider that line to be of infinite length and check if such a line intersects the MBR using Theorem 2.17. If it does, then the intersection must have happened within the bounds of the finite length piece (since that covered the complete MBR intervals, by the first theorem). Given this, a valid line-MBR detection algorithm can be constructed as in Algorithm A.1.3.

Now suppose we wish to find all objects in d-dimensional space which intersect a given $d-1$-dimensional hyperplane H. Let this hyperplane be given by the following expression:

$$H : \{x \in \mathbb{R}^d | n \cdot x = b\}, \text{for given } n \in \mathbb{R}^d, b \in \mathbb{R}. \tag{2.11}$$

We will present here a naive algorithm for hyperplane-MBR intersection. It rests on the observation that any intersecting hyperplane should also intersect at least one edge of the MBR.



(a) A line which does not intersect but does have interval overlap on each dimension.

(b) A line piece which would intersect the MBR if it were but a tiny bit longer.

FIGURE 2.8. Some figures illustrating line-box intersection.

Therefore, it is sufficient to simply check for all edges if they intersect with the hyperplane. If they do not, then there is certainly no intersection between the hyperplane and MBR. On the other hand, if any edge does intersect, then trivially the hyperplane also does intersect with the MBR. A single edge e of the MBR may be given by:

$$e_j = \{(v_1, \ldots, v_{j-1}, x, v_{j+1}, \ldots, v_d) \in \mathbb{R}^d | \text{ where } v_i \in \{b_i^-, b_i^+\}, \text{and } x \in \mathbb{R}, b_j^- \leq x \leq b_j^+\} \tag{2.12}$$

and hence e intersects H iff for $x = \frac{b - \sum_{i=1}^{j-1} n_i v_i - \sum_{i=j+1}^{d} n_i v_i}{n_j} = \frac{b^- - n \cdot v}{n_j} + v_j$, then $x \in [b_j^-, b_j^+]$. Algorithm A.1.6 is based on this result. Note that since this approach requires us to check each edge (worst case), this algorithm has exponential running time with respect to the number of dimensions d.

This leaves us with only the k-nearest-neighbours query. This query tries to find all objects with its MBR closest to a given point $p \in \mathbb{R}^d$, so obviously this query has a lot in common to the simpler point-query algorithm. The difference lies therein that when we found an object from a point query on p, we have to extend our search to find $k-1$ other closest objects; or if no object is found, we still somehow have to find the k objects closest to that point.

Note that from any given point we can derive the distance to an MBR; that is, the minimum distance to all objects contained in that MBR. A simple algorithm which achieves this is given in Algorithm A.1.11. Now, a valid $k$nn query algorithm would be to first follow Algorithm A.3.2 for the point p. The node returned by that algorithm is the node v whose MBR either contains p or is closest to p. We will now find the k neighbours, starting at the current node by following these two phases:

(1) *(Fill)* Let $S = \{MBR(o) | o \in v_O\}$. If $|S| > k$, sort S according to the distances of the MBRs to p. Then drop the $|S| - k$ MBRs with the greatest distance. If $|S| < k$, sort the unvisited sibling nodes w_i of v according to distance of MBR(w_i) from p. Then recursively visit the sibling nodes in that order, and keep adding the object MBRs with the shortest distance to p to S. Stop if $|S| = k$ and remember the last node v visited during filling.

(2) *(Extend)* Find the maximum distance r of an object MBR in S. From node v, start to visit all other unvisited nodes w_i in the tree for which the distance MBR(w_i) to p is less than r. If any object with its MBR distance to p lower than r is encountered, let this MBR replace the object corresponding to the distance r and update r accordingly and continue this

loop. Stop when there are no other unvisited nodes with their MBR distance to p lower than r.

This algorithm is detailed in Algorithm A.1.8. Note that this algorithm surely finds the k closest objects to a point p, with regards to their MBR. Since this query was not one of the three 'basic queries' introduced in the first chapter, we do not provide theoretical analysis of this particular query due to time limitations. The algorithm however was implemented in the experimentation software, and as such, the $k$nn query is included in the experiments section of this thesis.

2.3.7. Expectation. The basic R-tree as introduced in [Gut84] with linear, quadratic or exponential (or rather polynomial) split methods may be regarded sub-optimal with respect to the variations on R-trees proposed later. For example, R*-trees (1990) have been reported to perform almost always better [KSS90, p. 331] in terms of query performance, at the cost of a relatively small increase in construction time.

In current literature, many R-tree variants have been proposed[7]. These include:

- R$^+$-tree ([SRF87], 1987)
- R*-tree ([KSS90], 1990)
- Hilbert R-tree ([KF94], 1994)
- Priority R-tree ([AdBHY04], 2004).

Also, various methods for generating static R-trees have been proposed. These methods differ from the usual dynamic approach in that they process an entire set of input objects, instead of adding objects one-by-one into a dynamic R-tree. This new method is called *bulk-loading*. After a static R-tree is constructed, one may still apply tree update methods as used in dynamic R-trees; hence bulk-loading can be viewed as a pre-processing step. Bulk-loading algorithms include:

- Packed R-tree ([RK85], 1985)
- Hilbert Packed R-tree ([MNPT06, p. 36], 1993)
- Sort-Tile Recursive R-tree ([LEL97], 1997)
- Top-down Greedy Split ([GRLL98a], 1998).

Bulk-update algorithms (methods for inserting sets of objects into an existing R-tree) have also been proposed, as well as methods for merging (grafting) two somewhat equally sized already existing R-trees [MNPT06].

The R-tree variants mentioned here are but a small subset of all variants proposed. We will only concentrate on an even smaller subset of the variants mentioned above; this is because not all R-tree variants are expected to be useful in our current context, and due to time limitations. The variants chosen for further research and experimentation were selected by considering the aforementioned papers with our current context in mind.

2.4. Bulk-Loading: Top-down Greedy Split (TGS)

The basic R-tree adds objects one-at-a-time to the R-tree. In cases when all objects to be added are known in advance, which is true for us, we may want to think about using the extra information this gives us. This is the main idea of the *Packed R-tree* variant.

2.4.1. Packed R-tree. This R-tree variation was introduced in 1985, one year after the first appearance of R-trees in 1984. Inventors Roussopoulos and Leifker [RK85], proposed the following method to build a so-called *packed R-tree*.

DEFINITION 2.18 (Packed R-tree). *Let $S = \{o_1, o_2, \ldots, o_n\}$ be a set of objects to be stored in an R-tree structure. Also assume that we can impose some kind of ordering to the MBRs $MBR(o_i)$, so that we can cut the object set S into m smaller subsets S_i, each containing neighbouring objects with respect to*

[7] for a small overview, see [MNPT06]

the imposed order. These subsets are stored at the leaf nodes of the R-tree to be built. If we now proceed with calculating the MBRs of the leaf nodes and then recursively execute the procedure described just now to build the internal nodes higher up, we have constructed a packed R-tree.

In [RK85], the imposed order is based on the coordinate centre of a MBR; each MBR is ordered by the ith dimension of the centre coordinate. Which coordinate dimension is used, is selected cyclically at each recursive step. See figure 2.4.9(a) for an illustration as to how packed bulk loading proceeds. Note that this is quite similar to the bisection tree described earlier. Instead of dividing the object space in two by means of a hyperplane, we now directly split the set of objects according to their position on a single dimension. Another difference is that the bisection tree partitions the input set in three subsets, while the packed R-tree splits it in M_E subsets. In retrospect, the packed R-tree mechanism may be viewed as another generalisation of the bisection method.

Note that a result of this construction method, is that each child contains precisely m children or objects where possible; it may contain less if there were not enough data elements to store in the tree. A tree which has the maximum number of children at each node is called a *packed tree*, hence this variation name. As a results of this, when the number of input elements n is known, the memory requirements for a packed R-tree are also known.

The imposed order used in this definition obviously does not always yield good results. Consider for example the situation in figure 2.4.9(a); drawing the MBRs of the internal nodes of the resulting tree at depth three yields a great amount of overlap and dead space. See Figure 9(b). Also, any case in which data elements are closely packed in a few dimensions, and greatly separated from each other in other dimensions would result in similar badly constructed trees when using an imposed ordering on based on a single dimension.

The reason why things go wrong in these cases, is because the imposed ordering had bad localisation properties; object which are close in such orderings, generally do not lie close in space, even more so when the number of dimensions gets larger than two. Other orderings have been proposed to work with this packing strategy, one of which uses the Hilbert curve; see Section 2.6.

A packed R-tree is a *bulk-loading* algorithm in the sense that it attempts to load lots of data into the R-tree, in one strike. As we have seen, this packed strategy builds a tree bottom-up. However, when we query a R-tree, we work top-down; therefore, is it not better to try to make query times as efficient as possible by bulk-loading top-down as well? This is what the *Top-down Greedy Split (TGS)* algorithm is based on.

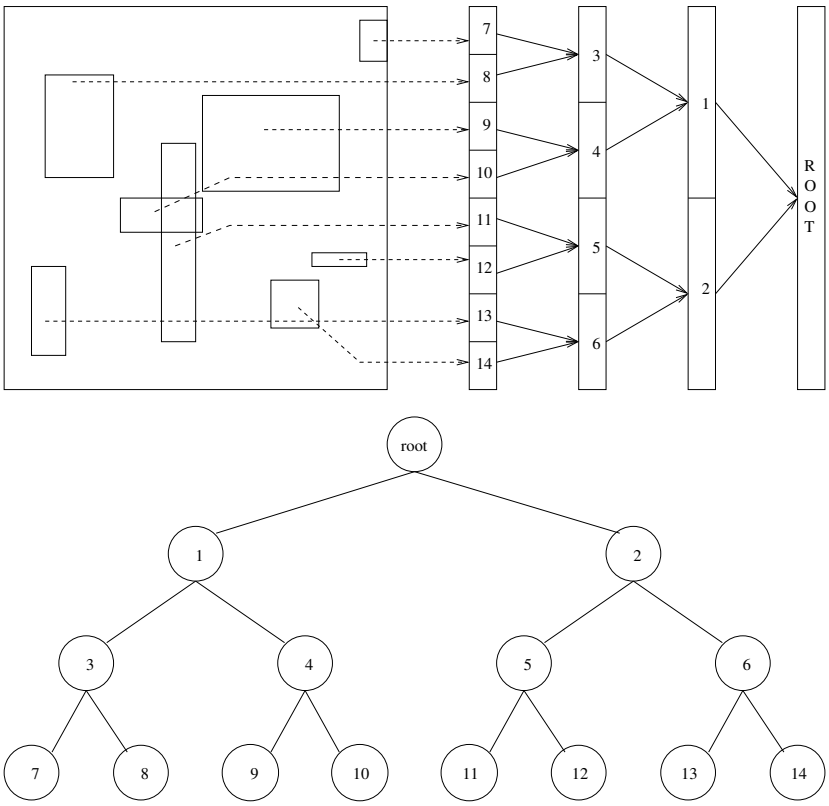
2.4.2. TGS. In the original TGS paper [GRLL98a], García, López and Leutenegger summarise the underlying principles as the following two basic ideas.

(1) *"Minimise first the top levels since the potential for cost reduction is higher."*
(2) *"Consider all partitions induced by guillotine cuts such that resulting sub-trees are fully packed."*

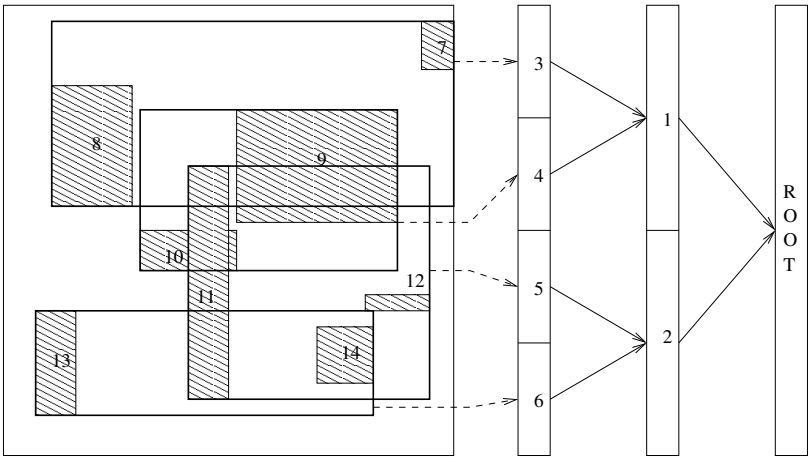
The first item refers to the fact that if we build the R-tree top-down, we can better control MBR properties like overlap and dead space; these are no longer defined by the leaf nodes constructed at first as is the case with packed R-trees. Remember also that as with packed R-trees, each node in the R-tree should have the maximum of M children or elements stored.

The second item refers to exactly *how* we are going to achieve this, namely by applying axis-orthogonal cuts which partitions any set S of n objects into M subsets $S_1, S_2, \ldots, S_M$ which are fully packed; i.e., contain enough elements to completely fill an R-tree. Almost surely, there exist multiple sets $S_1, S_2, \ldots, S_M$ satisfying this. Therefore, a greedy partitioning method is proposed.

This method considers all possible binary splits $\tilde{S}_1, \tilde{S}_2$ resulting from axis-orthogonal cuts and selects the best pair with respect to some function $f_C(\tilde{S}_1, \tilde{S}_2)$. This process then is applied to firstly, each of the best subsets, and secondly recursively on the resulting subsets. This is done



(a) Overview of where the rectangles 7 to 14 are stored in the packed R-tree.



(b) Overview at one tree level higher. The thicker rectangles denote the MBRs of the internal nodes at tree height 2. Notice that the Packed R-tree algorithm does not always result in a good R-tree; a large amount of overlap can be observed here.

FIGURE 2.9. Illustrations of the construction of a packed R-tree. The ordering used is the maximum y-value of each rectangular object. The packed R-tree has a maximum number of children or elements of 2.

until a total of M subsets are created. This partitioning method is detailed in Algorithm A.4.3 and A.4.4.

This partitioning is first done on the total amount of n input objects. The M subsets returned then denote the objects which are to be stored at each child of the resulting R-tree root. This process of splitting is then repeated for each of these children, until the number of elements to be stored at a single child becomes less than M. The final algorithms are given in Algorithm A.4.1 and A.4.2.

In the experiments done in [GRLL98a], the cost function has been chosen to be the volume (or area, in the two-dimensional case) of $\text{MBR}(\cup_{s\in\tilde{S}_1})$ and $\text{MBR}(\cup_{s\in\tilde{S}_2})$; in effect, we let TGS minimise the MBR volumes. Minimising the MBR volumes results in a smaller probability that a random query is contained in resulting MBR. Other cost functions are possible and may work better in our application; one alternative researched is minimising the intersection volume of the MBRs. The rationale of this cost function is that we wish to recurse in as few children of a node as possible when doing a query. Minimising the area that multiple MBRs have in common should help achieve this.

Algorithm A.4.1 deviates from the one in [GRLL98a] since the algorithm description given there was not detailed enough for our purposes. Instead, our algorithm is derived from the one presented in [AS07, p. 8-10].

2.4.3. Random TGS. As explained above, the TGS algorithm recursively calls the BestBinarySplit algorithm to obtain $p \leq M$ partitions of some input set I at level l. These partitions each contain no more than M^l elements, where $l \leq h$ is the current tree level under construction. The BestBinarySplit considers $q = \lceil |I|/M^l \rceil - 1$ different splits for each available ordering in S and tries to find the one resulting in the lowest cost, as explained above. Hence, we may say the algorithm considers the following matrix, where the c_{ij} denote the outcome of the cost function for each combination of q splits and ordering in S.

$$\begin{pmatrix} c_{11} & c_{21} & \cdots & c_{|S|1} \\ c_{12} & c_{22} & \cdots & c_{|S|2} \\ \vdots & \vdots & \ddots & \vdots \\ c_{1q} & c_{2q} & \cdots & c_{|S|q} \end{pmatrix}. \tag{2.13}$$

At each matrix entry, the cost function as evaluated by use of f_C for the specific split number and the specific ordering is stored. Only one of those entries is the optimal minimum split, but what if we would settle for some sort of local minimum as well? Local, in the sense that we only consider a subset of entries of the matrix in (2.13) and determine the minimum thereof.

Let $f \in \mathbb{R}, 0 \leq f < 1$. When the BestBinarySplit algorithm is about to start processing some entry c_{ij}, we generate some random number $g \in [0,1)$. If $g < f$, we indeed calculate c_{ij}. If it is not, we skip calculating c_{ij}. This results in a matrix in which $f \cdot d|S|$ elements are evaluated, on average. Taking the minimum of only those evaluated entries results instead of the true optimal minimum value, results in the new *'Random' TGS* method. We will denote the factor f as the *sampling rate*.

2.4.4. Expectation. Due to the many evaluations of the cost functions required for executing the TGS bulk-loading algorithm, we expect a construction time significantly slower than that of the basic R-tree. The data in [AdBHY04, Figure 12] supports this expectation; the TGS bulk-loading algorithm is by far the slowest variation experimented with, in terms of construction time.

This construction time however still seems to scale linearly, like the other variants. In the same experiment, on real-life application datasets, the TGS method obtains the best query times

[AdBHY04, p. 23-24]; but this advantage seems to deteriorate on more extreme (synthetic) datasets [AdBHY04, Figure 16].

The Random TGS method (or simply RTGS) is expected to have the same complexity with regards to construction time, but depending on the sampling rate, it should be a factor faster than the standard TGS method. With respect to query efficiency, it is expected that RTGS performs worse than TGS. How much worse however, should be determined experimentally.

2.5. HilbertTGS

The drawback of the normal TGS method is that it needs to consider multiple orderings resulting in, typically, d quicksort calls at each recursive call of the bulk-loading algorithm. The number of calls directly influences the running time, and as such, in the most optimal case, we would like to sort only one time. The HilbertTGS bulk-loading scheme tries to achieve this by considering only a single ordering, namely the so-called *Hilbert-ordering*, which we will explain now.

2.5.1. The Hilbert curve. The Hilbert curve is one of the most *fair space-filling* curves, at least when the number of dimensions is not too large [MAK02, p. 19]. In line with this article, "fair" denotes that the curve "behaves similarly towards all dimensions". This attributes to the *localisation* property of the Hilbert curve; points close to each other on the Hilbert curve also lie close to each other in the Euclidean sense. This will become more clear later on.

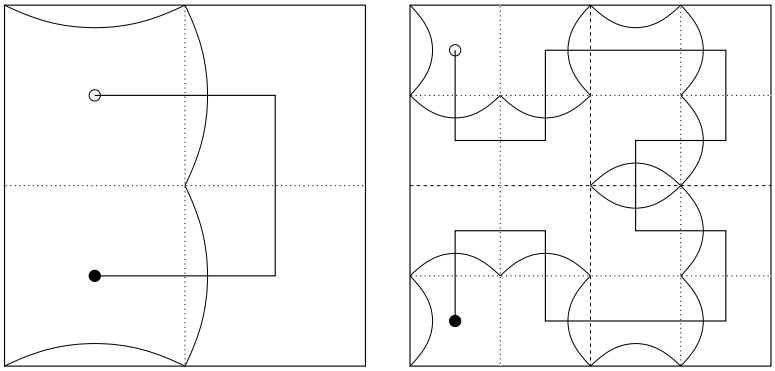
DEFINITION 2.19. *First order Hilbert curve*
The first order Hilbert-curve is defined in $\mathbb{R}^d$ as follows. Let U denote the $[0,1] \times [0,1]$ unit cube. Let us split U in four subsquares $U_1^1, \ldots, U_4^1$ of equal size (in terms of area). Let $p_1^1, \ldots, p_4^1$ denote the centre coordinates of $U_1^1, \ldots, U_4^1$. Then, a first order Hilbert curve is a Hamiltonian walk $H^1 = \{h_1^1, \ldots, h_4^1\}$, where points are visited in the same order as the set H^1, with $H^1 = P^1 = \{p_1^1, \ldots, p_4^1\}$ (thus the set H^1 is the same as P^1 but the order of their elements may differ).

Note that the order of H^1 may differ and as such, multiple possible first order Hilbert curves exist. We will denote $U_1^1, \ldots, U_4^1$ as the *subcells* of the first order Hilbert curve. Using this, we may define the nth order Hilbert curve as follows.

DEFINITION 2.20. *nth order Hilbert curve*
Let $U_1^{n-1}, U_2^{n-1}, \ldots, U_{4^{n-1}}^{n-1}$ be the subcells of the $(n-1)$th order Hilbert curve $H^{n-1} = \{h_1^{n-1}, \ldots, h_{4^{n-1}}^{n-1}\}$. Let us split each U_i^{n-1} again in four equally sized subsquares to obtain a series U_j^n, $1 \leq j \leq 4^n$. Also let U_j^n be a subsquare of $U_{\lfloor j/4 \rfloor}^{n-1}$. Denote with $p_1^n, \ldots, p_{4^n}^n$ the centre coordinates of the squares U_j^n. The nth order Hilbert curve is a Hamiltonian walk H^n on the points in $P^n = \{p_1^n, \ldots, p_{4^n}^n\}$, again not necessarily in that order. The order of the elements in H^n is restrained as follows:

1. *For each group $G_i = \{H_j^n | \lfloor j/4 \rfloor = i\}, 1 \leq i \leq 4^{n-1}$, there is a $k \in [1, 4^{n-1}]$ such that $G_i = \{p_j^n | \lfloor j/4 \rfloor = k\}$.*
2. *Let $i \in [1, 4^{n-1}]$. Let U_k^{n-1} be the subcell of the $(n-1)$th order Hilbert curve for which $h_i^{n-1} \in U_k^{n-1}$. Then, for each group $G_i = \{H_j^n | \lfloor j/4 \rfloor = i\}$, each point $x \in G_i$ also must be contained in U_k^{n-1}.*
3. *For each successive pair of points $v, w \in H^n$, $v - w$ is nonzero at precisely one dimension only.*

Restraint (1) makes sure the Hilbert curve always sequentially visits the points contained in a single subcell of the $(n-1)$th order Hilbert curve. Restraint (2) furthermore demands that the order in which the subcells of the lower order Hilbert curve are visited, correspond to the lower order Hilbert curve itself. The third restraint enforces that the Hilbert curve always consists of straight lines between centre points of subcells. Example first, second and third order Hilbert



(a) First-order. (b) Second-order.

FIGURE 2.10. Two basic Hilbert curves.

curves are drawn in figures 2.4.10(a), 2.4.10(b) and 2.11; the bold lines denote the classical Hilbert curve defined here.

Note that in essence, we simply cut an unit square in 4^n subsquares and define a curve which visits all the centres of these subsquares. Following this notion, a Hilbert curve can also be defined by a spline-like curve visiting only the cornerpoints of subsquares, with the requirement that each subsquare, excluding their cornerpoints, contains exactly one curve. A single curve can then be refined recursively by dividing the containing square in four, and drawing four splines along those new subsquares such that the start and ending cornerpoints remain the same. These spline-like representation of the Hilbert curve is also drawn in the figures 2.10(a), 2.10(b) and 2.11.

Considering the fact that this spline-like representation visits the cornerpoints of subsquares, one can easily see that for each point $x \in U$, there is a cornerpoint arbitrarily close to x as the order n of the Hilbert curve tends to infinity. We hence say the Hilbert curve is *space filling*. Since the Hilbert curve also is defined as a series of uninterrupted connections between points, it is also a *continuous* curve. The Hilbert curve is easily extended to multiple dimensions; we then use (hyper)cubes to bisect the unit hypercube.

2.5.2. Hilbert coordinates. The Hilbert curve can be used to obtain a mapping from $\mathbb{R}^d \to \mathbb{R}$ as follows. Consider some d-dimensional hypercubic domain in which we have an nth order Hilbert curve. Let us decide on a start p_S and ending point p_E of the Hilbert curve and using this convention, we may define for each point $x \in \mathbb{R}^d$ *on the Hilbert curve* the relative distance $d(x)$ to p_S, where $d(p_E) = 1$ and $d(p_S) = 0$. This is the general idea of using the Hilbert coordinate to map multi-dimensional coordinates to a number in $[0, 1] \subset \mathbb{R}$.

Mathematically we define the Hilbert coordinate as follows. Let P_n be the collection of points a d-dimensional nth order Hilbert curve visits, and let $x \in \mathbb{R}^d$ be the spatial coordinate we wish to transform to a Hilbert coordinate. Let $p \in P$ be the point for which $p = \arg\min_{y \in P_n} |x - y|$. Now suppose the point p is the ith point the Hilbert curve visits. Then the nth order Hilbert coordinate c is given by:

$$c_n = i/|P| = \frac{i}{2^{dn}}. \tag{2.14}$$

Following this, we can define the Hilbert coordinate as follows.

DEFINITION 2.21. *Hilbert coordinate*

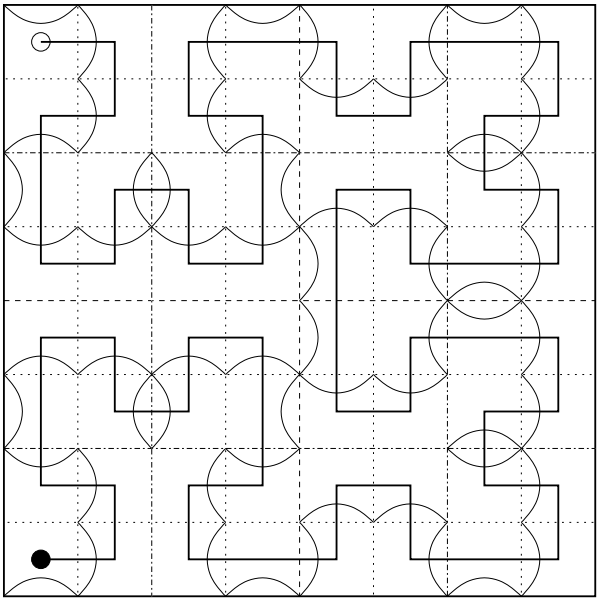


FIGURE 2.11. A third-order Hilbert cure.

Let c_n be as in equation 2.14. Then the Hilbert coordinate c is given by:

$$c = \lim_{n \to \infty} c_n. \tag{2.15}$$

Note that calculating the exact Hilbert coordinate as in equation 2.15 numerically is inefficient and often not even possible; we therefore opt to approximate the Hilbert coordinate by calculating c_n for a prefixed n.

Straightforward application of formula 2.14 requires we know exactly how the Hilbert curve is drawn. In implementation, this means we either need to store the Hilbert curve in memory, or that we need to recalculate it every time a Hilbert coordinate transform is performed; for large n, this might prove to be too costly. Luckily [**BG01**], proposed a method which does not need information on the whole Hilbert curve to calculate the Hilbert transformation. We will prepare some terminology needed to explain this method in more detail.

Consider figure 2.13. We now have systematically numbered each cell. Note that each cell contains the same Hilbert curve segment length and also that each centre of a cell always is on the Hilbert curve. Without loss of generalisation, assume the cube in the figure is the unit cube $U = [0,1] \times [0,1]$. For each coordinate $x \in U$, we can determine to which cell it belongs to. We will map such a point to the Hilbert coordinate belonging to the centre point of that cell.

We can easily determine the Hilbert coordinate by using the cell numberings introduced in figure 2.13. We can interpret the grid numbers as being in base 4, and as such, the cell number we wish to know the Hilbert coordinate of divided by the maximum number of cells minus one, returns a Hilbert coordinate in $[0,1]$; we are actually mapping cells to Hilbert coordinates. So, $d(c) = c \cdot \frac{1}{3}$ exactly describes the Hilbert coordinate value in cell c for the first order Hilbert curve. For the nth order we have[8]:

$$d(c) = \text{toDec}(c) \cdot \frac{1}{4^n - 1}, \tag{2.16}$$

[8] for three dimensions only

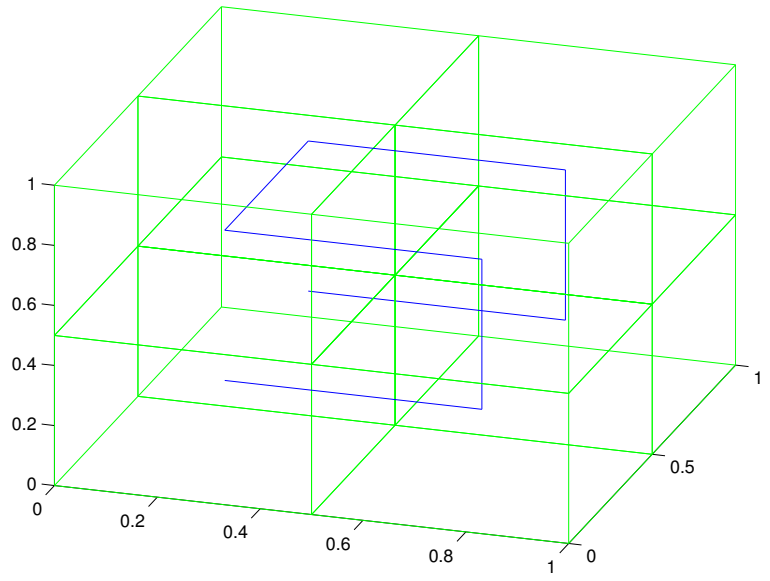


FIGURE 2.12. A three dimensional first order Hilbert curve drawn in a unit box subdivided in 8 cells.

where toDec converts the base-4 number c to a decimal number. We can however calculate the Hilbert coordinates more efficiently by first subdividing c by 4^n in base 4. We then have cell numbers ranging from 0 to $0.33\ldots3$, the number of digits being dependent on the order of the Hilbert curve used. Hence all cells now map to numbers in $[0, \text{toDec}(0.33\ldots3)]$. We could multiply the number by a constant to obtain values exactly in $[0, 1]$, but this does not matter for the resulting ordering so we skip such a step. Conversion to decimal numbers is however necessary, because in implementation, comparison functions work natively in base 10 and not 4.[9] In any case, we propose to use the following formula for cell-to-Hilbert coordinate transformation using a nth order Hilbert curve:

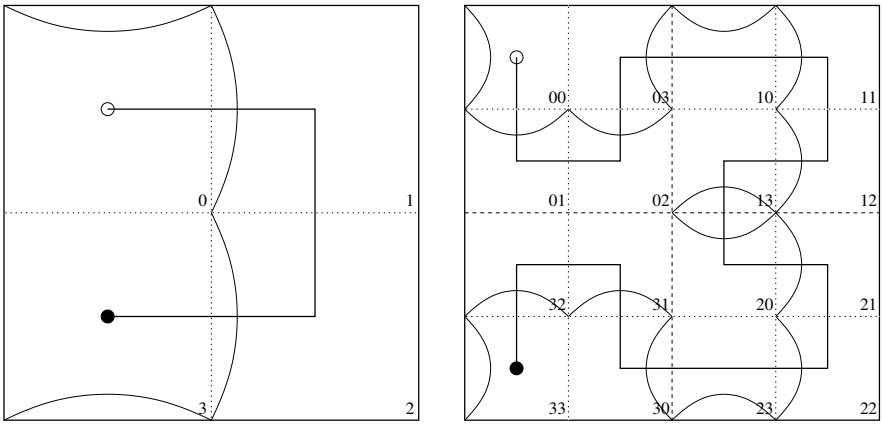
$$d(c) = \text{toDec}(c/4^n), \quad \text{where the division is carried out in base 4.} \tag{2.17}$$

In three dimensions, we have that the first-order Hilbert curve is as in figure 2.12. The domain of this Hilbert curve is now subdivided into 8 cells instead of 4. This leads to a cell-Hilbert coordinate formula similar to equation 2.17, but with the division carried out in base 8.

The idea of using the cell codes for fast Hilbert coordinate mapping comes from [**BG01**], as already mentioned. In this article, a recursive method is described for determining the cell number corresponding to a given coordinate. We will now briefly describe this procedure. For more details, we refer to the article.

For brevity, we stick with the two-dimensional case. Suppose we have the unit rectangle U and some point $x \in U$ and we want to determine in which cell x is contained. Assume we are using an nth-order Hilbert curve with n fixed. Consider for example figure 2.14. We see that x is closest to the point d which corresponds to the fourth cell. We thus remember the Hilbert code 0.3 and *'zoom in'* on that cell; see figure 2.15.

[9] Simply interpreting a base 4 number as a base 10 number is not advisable. There would be a 'hole' between 0.3 and 1 which, for example, may cause distance-based operations to behave unexpectedly.



(a) (b)

FIGURE 2.13. The first- and second-order Hilbert curves with a systematic grid indexing.

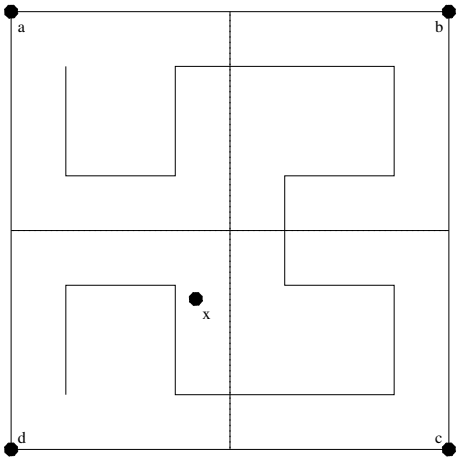


FIGURE 2.14. Searching for a Hilbert coordinate belonging to x, using a second-order Hilbert curve.

We first note that the new a, b, c, and d coordinates in figure 2.15 can be derived from those in figure 2.14 as follows:

$$a_{\text{new}} = \frac{c+d}{2}, \quad b_{\text{new}} = \frac{a+c}{2},$$
$$c_{\text{new}} = \frac{a+d}{2}, \quad d_{\text{new}} = d. \tag{2.18}$$

We now see that point b is closest to x, which corresponds to the second cell. Our Hilbert coordinate then becomes 0.31 in base 4. We of course do not need to stop at only 2 recursions; we can apply the described method until we have reached the required precision. We do need different tables for each cell $1 \ldots 4$ we may recurse in; see [BG01, p.7-8]. Also note that this method does not require the explicit evaluation of the Hilbert curve whatsoever; it only recursively applies the tables corresponding to the closest cell to x to obtain the Hilbert coordinates in base 4. Pseudocode for this process is given in Algorithm A.6.1.

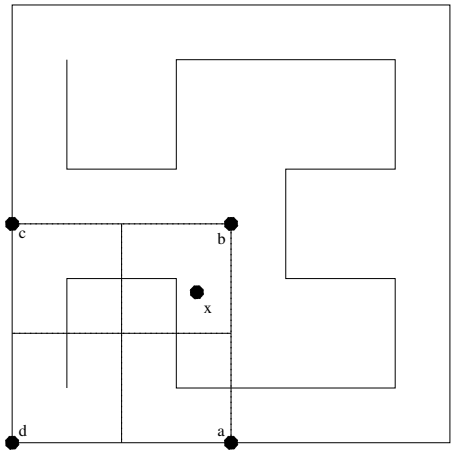


FIGURE 2.15. Searching for a Hilbert coordinate belonging to x, using a second-order Hilbert curve. Zoomed in on the fourth cell.

2.5.3. TGS using Hilbert coordinate ordering. We can sort any set of bounding boxes by sorting on the Hilbert coordinates of their centre coordinate points. We may use solely this ordering instead of the orderings based on the lowest coordinate on each dimension as in the basic TGS algorithm. In this case, we then only have to sort over a single ordering instead of three. Thus we can pre-sort the input set according their Hilbert coordinates instead of calling a sort-method at each recursive call of Algorithm A.4.4. This results in an adaptation of the original TGS algorithms outlined in A.4.1 and A.4.4 to those in A.5.1 and A.5.2.

In [KF93], a packed R-tree variant using Hilbert coordinates was introduced. While generally faster with respect to datastructure construction compared to other R-tree variants, our initial experiments showed the query times were slower than those of the bisection method. Therefore, this variants was not researched in detail. The paper introducing this variant, also did research to different ways to obtain Hilbert coordinates. On basis of this, we decided to use the bounding box centre as the coordinate to transform to a Hilbert coordinate.

Another possibility for using the Hilbert ordering is a Hilbert mapping from $\mathbb{R}^{2d}$ to $[0,1]$ as follows. In the case of hypercubic bounding boxes defined by a minimum and maximum coordinates $b^-, b^+ \in \mathbb{R}^d$, we now map the coordinate $(b_1^-, b_2^-, \ldots, b_d^-, b_1^+, \ldots, b_d^+) \in \mathbb{R}^{2d}$ to its Hilbert coordinate. A third researched alternative uses the centre coordinate $c \in \mathbb{R}^d$ of (any type of) bounding box, and a diameter bounding box $d \in \mathbb{R}^d$ denoting the range of the bounding box on each dimension[10]. The coordinate transformed is then also $2d$-dimensional: $(c_1, c_2, \ldots, c_d, d_1, \ldots, d_d) \in \mathbb{R}^{2d}$.

The experiments done in [KF93] show that the best results were achieved when taking the Hilbert value of the centre coordinate of the bounding box. Note that the experiments have been performed solely for $d = 2$ and for rectangular bounding boxes only; we have assumed the result would hold for $d > 2$, or at least for $d = 3$. Of course, more research on this particular aspect may be warranted.

2.5.4. Expectation. The HilbertTGS bulk-loading mechanism differs from the original TGS algorithm only in the fact that it uses a single Hilbert-coordinate ordering instead of d low-coordinate orderings. This should result in faster construction times, but may result in a less efficient R-tree with respect to query times. No previous works presenting this variation has been found, so we cannot give more substantial expectations beforehand. Note however that this

[10] For cubic bounding boxes, $d = b^+ - b^-$.

particular bulk-loading strategy has been implemented and used in our own experiments, see Chapter 4.

2.6. Hilbert R-Tree

The previously defined HilbertTGS method builds a static R-tree using the Hilbert coordinate. In [KF94], a dynamic Hilbert R-tree is proposed. Here, the one-dimensional ordering inferred by the Hilbert transformation is used to apply merging and deferred splitting on under- and over-flows, respectively. This results in a datastructure essentially similar to B^+-trees [MNPT06, p.20].

Each leaf node of an Hilbert R-tree stores the Hilbert coordinate of the MBR of each object stored. Also, each node[11] stores the maximum Hilbert value of each of its children or objects locally stored. This latter maximum Hilbert value is commonly denoted the LHV-value.

2.6.1. Hilbert R-tree object insertion. Suppose we want to insert an object o into the tree. Let h denote the Hilbert coordinate of the centre of MBR(o). Then we select the leaf node v for insertion to be the one with the smallest LHV-value greater than h. When v has less than the maximum number of objects stored, we insert o in v and are done. When the leaf node is full, we proceed with overflow handling. The pseudocode for this algorithm is given in Algorithm A.7.1.

Note that this way of insertion does not guarantee the MBRs at each nodes are still correct; no MBR updates whatsoever are performed. Therefore, we need to call a specialised update algorithm which is called after each insert operation. See Algorithm A.7.3.

2.6.2. Overflows – deferred splitting. The Hilbert R-tree uses so-called s-to-$s+1$ splitting. Assume the set of siblings of v is denoted by S. The subset $\tilde{S} \subset S$ contains at most s elements which have its LHV-value closest to that of v.[12] We denote with the set C all children or objects stored at all nodes $w \in \tilde{S}$.

If all nodes in $\tilde{S}$ are full, we create a new sibling node $\tilde{v}$ and redistribute $C \cup o$ among the nodes $\tilde{S} \cup \tilde{v}$. If the nodes in $\tilde{S}$ were not full, the objects $C \cup o$ were redistributed among $\tilde{S}$ alone.

Redistribution of a set P among a set of nodes Q in this case means that we simply split P into $|Q|$ subsets according to the LHV- or Hilbert values of the items in P. Hence each node Q will contain at most $\lceil |P|/|Q| \rceil$ nodes, which is always less than the maximum number of nodes allowed, since certainly $\lceil \frac{|P|-1}{|Q|} \rceil < M$, because not all nodes in Q were full.

The pseudocode for handling overflows is given in Algorithm A.7.4.

2.6.3. Hilbert R-tree object deletion. Suppose we wish to delete an object o. We can perform a standard MBR search to find the leaf node v which stores o. We proceed by simply deleting o from v, after which we check for underflows. The pseudocode is in Algorithm A.7.5.

2.6.4. Underflows – merging. We again construct the sets $\tilde{S}$ and C as with overflow handling. If all nodes in $\tilde{S}$ are underflowing, we simply delete one node from $\tilde{S}$ (while retaining the objects or children it stored in C) and redistribute C over the remaining nodes. If not all nodes were underflowing, we simply redistribute C over the complete set $\tilde{S}$. This algorithm is given in depth in Algorithm A.7.6.

[11] Note that this also includes leaf nodes.

[12] Note that implicitly, $\tilde{S}$ thus always contains v.

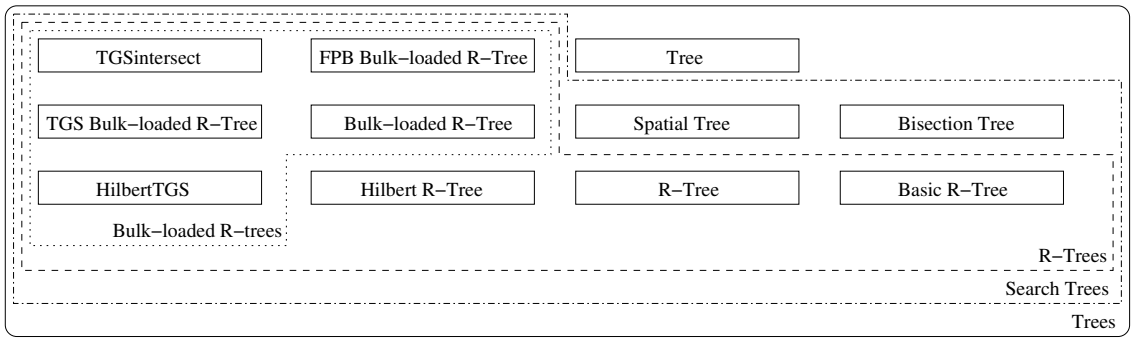


FIGURE 2.16. Visualisation of the classification of all trees introduced so far.

2.6.5. Expectation. The Hilbert R-tree explained here is expected to be superior to more conventional dynamic R-tree variants, such as the R^*-tree [KF94, Section 4.1]. The best value for s is expected to be 2 [KF94, Section 5]. On the other hand, this variant is reported to be (negatively) sensitive to large objects [MNPT06, p. 22] (which is logical, since the size is not preserved in the Hilbert coordinate mapping). Also, higher dimensionality causes the Hilbert curve to be less localised, reducing the Hilbert R-tree performance [MNPT06, p. 22].

2.7. Summary

In this chapter, several tree datastructures for efficient spatial storage and querying were presented. We will now give a brief concluding summary of all variants presented. We start with an overall classification of all tree concepts introduced so far.

We have introduced a general tree, from which all other datastructures were derived. Searchable trees were defined using the spatial tree; the R-trees and the bisection tree both are spatial trees. The introduced R-tree variants can be subdivided into normal (dynamic) R-trees on one hand, and the bulk-loaded R-trees on the other hand. This classification is visualised in figure 2.16. For the exact generalisation structure, we refer to the class diagram in figure B.1. Note that these diagrams closely resemble each other.

CHAPTER

3

THREE

Theoretical Performance

3.1. Algorithm Analysis

The R-trees we propose here to solve the introduced problem can be built in different ways, which lead to different R-tree variants. Of course, different R-tree variants take more time to construct than others; therefore, we would like determine how fast given construction algorithms are. Not only do we want to compare the algorithms of various R-tree variants, we also wish to compare construction times, memory usage and query times to the bisection method currently in use at Shell.

To be able to analyse memory usage or algorithms, be it construction or query algorithms, we will introduce some basic tools and notations here.

DEFINITION 3.1. *"big-Oh"-notation*
Suppose we have two functions $f, g : V \subset \mathbb{R} \to W$, for some W for which for any $x \in W$, we can determine the absolute value $|x| \in \mathbb{R}([0, \infty))$. f is $O(g)$ if and only if:

$$\exists x_0, \exists c > 0 \quad \textit{such that}$$

$$|f(x)| \leq c|g(x)|, \quad \forall x \in V, x \geq x_0$$

In this thesis, we will commonly write that $f = O(g)$ or $f(x) = O(g(x))$ or *f is of asymptotic order $O(g)$* when f is $O(g)$ as in the above definition. Intuitively, f being of asymptotic order $O(g)$ means that the function $g(x)$ grows faster than or equally fast as the function $f(x)$ as x tends to infinity, in terms of *orders* of growth. For example, linear functions such as $f_a(x) = ax$, $a \in \mathbb{R}$, $a > 0$ are all of the same order; the grow linearly with respect to x. Hence $f_a = O(f_1(x)) = O(x)$, for all a.

In computing sciences, we generally consider the orders of magnitude as in table 3.1. Next to the big-Oh notation, we also have big-Theta and big-Omega which denote if two functions grow equally fast, or if a function certainly grows faster than another, respectively.

DEFINITION 3.2. *"big-Theta"-notation*
Let the functions f, g be as in definition 3.1. f is $\Theta(g)$ if and only if:

Big-Oh notation	Order
$O(1)$	constant
$O(\log x)$	logarithmic
$O(\log^c x), c > 1$	poly-logarithmic
$O(x^c), 0 < c < 1$	fractional power
$O(x)$	linear
$O(x \log x)$	linearithmic
$O(x^2)$	quadratic
$O(x^c), c > 2$	polynomial
$O(c^x), c > 1$	exponential
$O(x!)$	factorial

TABLE 3.1. Common orders of growth in big-Oh notation, in ascending order. Adapted from [Wik07].

$$\exists x_0, \exists c_1 > 0, \exists c_2 > 0 \quad \textit{such that}$$

$$c_1|g(x) \leq |f(x)| \leq c_2|g(x)|, \quad \forall x \in V, x \geq x_0$$

DEFINITION 3.3. *"big-Omega"-notation*
Let the functions f, g be as in definition 3.1. f is $\Omega(g)$ if and only if:

$$\exists x_0, \exists c > 0 \quad \textit{such that}$$

$$|f(x)| \geq c|g(x)|, \quad \forall x \in V, x \geq x_0$$

For analysis of general divide-and-conquer algorithms, the *Master Theorem* is a very useful tool. Whilst algorithms on tree data structures at first may seem to have very little to do with conventional divide-and-conquer algorithms such as quicksort, we will show the opposite is true in this chapter. The master theorem, as given in [CLRS03, p. 73], may be formulated as follows:

THEOREM 3.4. *Master Theorem*
Let $a, b \in \mathbb{R}, f : \mathbb{Z} \to \mathbb{R}$ and let $T : \mathbb{Z} \to \mathbb{R}$ denote the running time of an algorithm applied on a problem of size n:

$$T(n) = aT(\lceil \frac{n}{b} \rceil) + f(n) \tag{3.1}$$

Then, the asymptotic behaviour of $T(n)$ may be bounded differently for each of the following cases. Write

$$t = \frac{\log a}{\log b}.$$

(1) *If*

$$f(n) = O(n^{t-\epsilon}), \quad \textit{for some } \epsilon > 0,$$

then

$$T(n) = \Theta(n^t)$$

(2) *If*

$$f(n) = \Theta(n^t),$$

then

$$T(n) = \Theta(n^t \log n) = \Theta(f(n) \log(n))$$

(3) *If*

$$f(n) = \Omega(n^{t+\epsilon}), \quad \textit{for some } \epsilon > 0$$

and

$$af(\lceil \frac{n}{b} \rceil) \leq cf(n), \quad \textit{for all } n \geq n_0 \in \mathbb{Z}$$

with $c < 1$ *constant, then*

$$T(n) = \Theta(f(n))$$

PROOF. See [CLRS03, Chapter 4.4]. □

Note that we can rewrite the conditions $f(n) = O(n^{t-\epsilon})$ and $f(n) = \Omega(n^{t+\epsilon})$ to $f(n) = O(n^s)$ and $f(n) = \Omega(n^s)$ for $s < t$ or $s > t$, respectively.

3.2. Construction

When construction times may vary due to the specifics of input elements or other aspects, we will only review the worst-case running times. We will derive the construction times for the bisection method and the basic R-tree method quite detailed, while for other R-tree variants we derive only asymptotic bounds.

3.2.1. Bisection tree. Here we consider Algorithm A.2.1 and subsequently use the notations as introduced there. Let us assume a subset $W \subset \mathbb{R}^d$ is always rectangular and stored by means of two (minimum and maximum) coordinates. Then, in implementation, selecting a bisection hyperplane can be done in 3 flops when we simply bisect on the middle coordinate of a cyclically selected dimension[1]; let $w^-, w^+ \in \mathbb{R}^d$ represent W and let i be the selected dimension. Then the bisection plane is given by $p = \{x \in W | e_i \cdot x = \frac{w_i^- + w_i^+}{2}\}$.

Constructing W_l and W_r thus boils down to copying $2(d-1)$ values from w^- and w^+ and setting $\frac{w_i^- + w_i^+}{2}$ on the remaining two ith coordinates;

$$W_l^- = w^-, \quad W_r^+ = w^+ \tag{3.2}$$

and

[1] This is how the Shell bisection method currently works.

$$W_l^+ = \begin{pmatrix} w_1^+ \\ w_2^+ \\ \vdots \\ w_{i-1}^+ \\ \frac{w_i^- + w_i^+}{2} \\ w_i^+ \\ \vdots \\ w_d^+ \end{pmatrix}, \quad W_r^- = \begin{pmatrix} w_1^- \\ w_2^- \\ \vdots \\ w_{i-1}^- \\ \frac{w_i^- + w_i^+}{2} \\ w_{i+1}^- \\ \vdots \\ w_d^- \end{pmatrix} \tag{3.3}$$

This is followed by sorting the elements into three groups O_m, O_l and O_r. This can be done by iterating over all objects o in O and comparing the ith dimension coordinate range of MBR(o) to $\frac{w_i^- + w_i^+}{2}$. There are three different cases;

(1) The interval on the ith dimension of MBR(o) contains only values lower than $\frac{w_i^- + w_i^+}{2}$.
(2) The interval on the ith dimension of MBR(o) contains only values larger than $\frac{w_i^- + w_i^+}{2}$.
(3) The interval on the ith dimension of MBR(o) contains $\frac{w_i^- + w_i^+}{2}$.

All three cases can be distinguished from by doing two comparisons between the extreme coordinates of MBR(o) and $\frac{w_i^- + w_i^+}{2}$, hence splitting O in O_m, O_l and O_r takes $2|O|$ flops. Calling the construction algorithm recursively requires $c = O(1)$ time.

If we also assume that set operations like those in lines $2-7$ in Algorithm A.2.1 can be done in a maximum of $O(z)$ time, where z is the number of elements in the set involved, we can present the following theorem on the construction time of the bisection tree.

THEOREM 3.5. *The construction time of the bisection tree is* $t_{bisection} = h(3 + 2d + 2n + c)$. *This is of asymptotic order* $O(hn)$.

PROOF. Note that at each recursion level $0 \leq l \leq h$, the construction algorithm has to sort the full amount of n objects, resulting at a running time of $3 + 2d + 2n + c$ per level. Since the number of recursion levels h is supplied to the algorithm, a running time of $h(3 + 2d + 2n + c) = O(hn)$ is achieved, since d is typically much smaller than n. □

3.2.2. Basic R-tree. We will assume here we have an input set O consisting of n elements to be added into a basic R-tree. This will be done by calling the Insert-algorithm A.3.1 for each element in O. The worst-case running time of the insertion algorithm for a single element is $2 + t_{\text{choose}} + t_S + t_{\text{update}}$, where t_{choose} denotes the maximum time required for the ChooseLeaf subalgorithm (A.3.2), t_S the maximum time for the appropriate split algorithm (A.3.4, A.3.5, A.3.6), and t_{update} the maximum time required for updating the modified leaf node(s) and their ancestors.

The leaf selection algorithm follows a single path from the tree root to a leaf node. For each node visited, an instance of the algorithm is run. A single run of the ChooseLeaf algorithm requires $3 + \tilde{n}(10d + 4)$ flops (See Algorithm A.3.2), where $\tilde{n}$ is the number of child nodes. Since the number of visited nodes equals the tree height, the total running time of Algorithm A.3.2 is bounded as follows.

$$t_{\text{choose}} = \sum_{i=1}^{h} (3 + \tilde{n}_i(10d + 4)) < \log_m n \, [3 + M(10d + 4)] \,, \tag{3.4}$$

We assume adding an element to a leaf node requires $O(1)$ time. After insertion, the chosen leaf node may overflow and the split algorithm may be called.

THEOREM 3.6 (Split-on-insertion worst-case running time). *The worst case running time for the splitting algorithm called when a leaf node overflows, is $t_S = c \cdot h = O(c \log_m n)$, where n is the number of data elements stored, and c is the worst case running time of the splitting algorithm applied to only a single node.*

PROOF. As mentioned in the text, a splitting algorithm may only call itself on the parent of the node it is currently splitting. Let the worst case running time of the splitting algorithm on a single node (that is, without considering a possible overflow of its parent node) be given by c.

Since the splitting algorithm can only be called on the parent of a given node, the worst situation occurs when the splitting algorithm is called on a leaf node and continues to create overflows all the way up to the root node; then the splitting algorithm is called a number of times equal to the height h of the tree.

As such the complete splitting algorithm takes $ch = O(c(\lceil \log_{m_E} n \rceil - 1)) = O(\log_m n)$ (equation 2.7). □

Updating the MBR of an internal node involves finding the extreme coordinates of the MBRs of its child nodes. This can be most efficiently done in $3d$ time. Updating a single leaf node and all of its ancestors therefore takes $3dh$ time. Hence the total time required for a single insertion is now certainly less than:

$$t_{\text{basic-insert}} < 2 + \log_m n(3 + M(10d + 4)) + 3dh + c \log_m n - 1$$

or, rather:

$$t_{\text{basic-insert}} < 1 + \log_m n(3 + 3d + M(10d + 4) + c). \tag{3.5}$$

with c unknown. Depending on which split algorithm we use, the asymptotic order of c is known to be $O(M)$, $O(M^2)$ or $O(g^M)$, for some $g \in \mathbb{R}$. We can however already give an worst-case basic R-tree construction time.

THEOREM 3.7. *The worst case construction time for a basic R-tree is as follows:*

$$t_{basic} = 2n + (3 + 3d + M(10d + 4) + c) \log_m n!, \tag{3.6}$$

This running time is of order $O(n \log n)$.

PROOF. A single insertion takes time dependent on the number of elements currently stored in the tree, according to equation 3.5, worst case. Hence the worst case tree construction time when continuously adding n elements is as follows.

$$\sum_{i=1}^{n} t_{\text{basic-insert}}(i) \tag{3.7}$$

which is equal to

$$\sum_{i=1}^{n} [2] + (3 + 3d + M(10d + 4) + c) \sum_{i=1}^{n} \log_m i.$$

This can furthermore be simplified to

$$2n + (3 + 3d + M(10d + 4) + c) \log_m n!$$

Also note that $\log_m n! \leq \log_m n^n = n \log_m n$, completing the proof. □

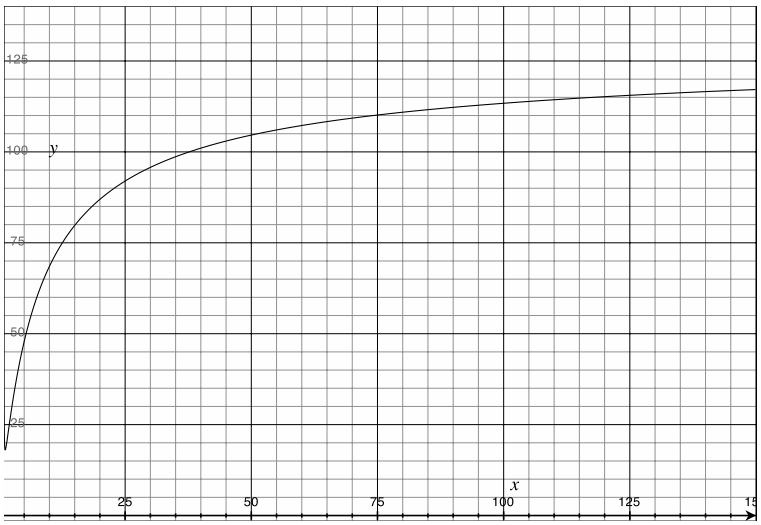


FIGURE 3.1. Illustration of the graph $y = \frac{2x+(3+3d+M(10d+4)+c)\log_m x!}{(3+2d+2x)\log_3 x}$ (equation 3.8) for $m = 2$, $M = 4$ and $c = 10m^2 = 40$. x represents the number of grid elements to be added in the trees, and y the construction time factor between the basic R-tree and the bisection tree.

Compared to the bisection tree construction method, the R-tree construction algorithm is at most $\frac{t_{\text{basic}}}{t_{\text{bisection}}}$ times slower, worst case. Let us assume that $h = log_3 n$ for the bisection tree. Then this ratio becomes:

$$f(n) = \frac{t_{\text{basic}}}{t_{\text{bisection}}} = \frac{2n + (3 + 3d + M(10d + 4) + c)\log_m n!}{(3 + 2d + 2n)\log_3 n}. \tag{3.8}$$

See figure 3.1 for the asymptotic behaviour of this factor. Note that this factor is always larger than 1; the basic R-tree construction time is always slower than that of the bisection tree. With respect to the number of elements to be stored, however, the factor by which the basic R-tree construction time is slower, is asymptotically bounded. For this, we calculate:

$$\begin{aligned}\lim_{n\to\infty} f(n) &= \frac{2n + (3 + 3d + M(10d + 4) + c)\log_m n!}{(3 + 2d + 2n)\log_3 n} \\ &= O(\frac{n \log n}{n \log n}) \\ &= O(1),\end{aligned}$$

and indeed see that $f(n)$ converges to a constant for large n.

3.2.3. TGS Bulk-Loading. We will start our analysis with Algorithm A.4.4, *TGSBestBinarySplit*. Let us use the notations defined there. This algorithm cuts the input set O into $p = \lceil n/m \rceil - 1$ equally large subsets O_i, with the size of O_i equal to m. Then, all sets $O_l^i = \cup_{k=1}^{i} O_i$ and $O_r^i = \cup_{k=i+1}^{p} O_i$ for all $i \in [1, p] \subset \mathbb{Z}$ are considered and the best split, with respect to a given cost function, is remembered. This is done for all different orderings in S, after the input set O has been sorted according to such an ordering in S.

At all times, we of course have that $O_l \cup O_r$ contains $\tilde{n} = |O|$ elements; thus we have that this algorithm at least sorts all elements in O, $|S|$ times. We will assume the cost for this is of order

$O(|S|(\tilde{n}\log\tilde{n})$, based on the expected[2] running time of a quicksort algorithm. Processing the different subsets O_l^i and O_r^i require $O(t_{f_C})$ time per pair, with t_{f_C} the time required for evaluating the cost function. Note that this is true only if we make a single pass over all elements in O while storing the bounding boxes of both O_l^i and O_r^i for all i; this requires a pre-processing step of $O(n)$ at each call of this algorithm. Failure to implement this will result in a quadratic-order construction time, instead of the construction time derived here. The total execution time for Algorithm A.4.4 is of order:

$$t_{\text{TGSBestBinarySplit}} = O(|S|(\tilde{n}\log\tilde{n} + pt_{f_C})) = O(|S|(\tilde{n}(\log\tilde{n} + \frac{t_{f_C}}{m}))). \tag{3.9}$$

We now move to Algorithm A.4.3, *TGSPartition*. This algorithm cuts an input set O into $p \leq M$ smaller subsets each containing no more than m elements. This is done by recursively calling Algorithm A.4.4. Worst-case, the binary split algorithm keeps cutting its input in half, which results in a total of $2^{\tilde{n}/m} - 1$ calls to the binary split algorithm; see figure 3.2. Note that the tree height equals $\log_2 \tilde{n}/m$ and the total number of recursion calls equals $2^{\tilde{n}/m} - 1$. Also note that the number of recursion calls at the ith recursion level equals 2^i. Combined with equation 3.9, this results in the following expression for the worst-case running time of Algorithm A.4.3:

$$t_{\text{TGSPartition}} = \sum_{i=0}^{\log_2 \tilde{n}/m-1} [\sum_{j=0}^{2^i} O(|S|(\frac{\tilde{n}}{2^i}(\log\frac{\tilde{n}}{2^i} + \frac{t_{f_C}}{m})))]$$

which can be rewritten to

$$\begin{aligned}
t_{\text{TGSPartition}} &= \sum_{i=0}^{\log_2 \tilde{n}/m-1} O(|S|\tilde{n}(\log\frac{\tilde{n}}{2^i} + \frac{t_{f_C}}{m})) \\
&= \sum_{i=0}^{\log_2 \tilde{n}/m-1} O(|S|\tilde{n}(\log\tilde{n} - \log 2^i + \frac{t_{f_C}}{m})) \\
&= O(|S|\log_2(\tilde{n}/m-1)\tilde{n}(\log\tilde{n} + \frac{t_{f_C}}{m}) - |S|\tilde{n}\sum_{i=0}^{\log_2 \tilde{n}/m-1} \log 2^i) \\
&= O(|S|\log_2(\tilde{n}/m)\tilde{n}(\log\tilde{n} + \frac{t_{f_C}}{m}) - |S|\tilde{n}\sum_{i=0}^{\log_2 \tilde{n}/m-1} i\log 2) \\
&= O(|S|\log_2(\tilde{n}/m)\tilde{n}(\log\tilde{n} + \frac{t_{f_C}}{m}) - |S|\tilde{n}\log 2\cdot\frac{(\log_2(\tilde{n}/m)-1)\log_2\tilde{n}/m}{2}) \\
&= O(|S|\log_2(\tilde{n}/m)\tilde{n}(\log\tilde{n} + \frac{t_{f_C}}{m} - \frac{(\log_2(\tilde{n}/m)-1)\log 2}{2})) \\
&= O(\tilde{n}|S|\log\tilde{n}(\log\tilde{n} + \frac{t_{f_C}}{m})) \\
&= O(|S|(\tilde{n}\log^2\tilde{n}\frac{t_{f_C}}{m})).
\end{aligned} \tag{3.10}$$

We now move up to Algorithm A.4.2, *TGSBulkLoadChunk*. This algorithm partitions an input set O into no more than M smaller subsets, each of size no larger than n/M^h, and recurses on each such subset with $h-1$ until $h = 0$. The thereby belonging call tree is shown in figure 3.3. Using equation 3.10 we obtain the following worst-case running time for Algorithm A.4.2.

[2] the quadratic worst-case running time is rarely attained in practical use of the quick-sort algorithm, although care must be taken to not attempt to sort an already ordered set; this would result in a $O(n^2)$ running time.

$$t_{\text{TGSBulkLoadChunk}} = M + \sum_{i=1}^{h} M^i (O(|S|(\frac{n}{M^i}\log^2(\frac{n}{M^i})\frac{t_{f_C}}{m})) + M)$$

which can be rewritten as

$$\begin{aligned}
t_{\text{TGSBulkLoadChunk}} &= O(M + \sum_{i=1}^{h}\left[|S|(n\log^2(\frac{n}{M^i})\frac{t_{f_C}}{\frac{n}{M^{i+1}}}) + M\right]) \\
&= O((h+1)M + |S|\sum_{i=1}^{h}\left[\log^2(\frac{n}{M^i})M^{i+1}t_{f_C}\right]) \\
&= O(|S|t_{f_C}(\sum_{i=0}^{h} M^i(\log(n) - \log M^i)^2)) \\
&= O(|S|t_{f_C}(\sum_{i=0}^{h} M^i(\log^2(n) - 2\log(n)\log(M^i) + \log^2(M^i)))) \\
&= O(|S|t_{f_C}(\log^2(n)\sum_{i=0}^{h} M^i - 2\log(n)\sum_{i=0}^{h} M^i\log(M^i) + \sum_{i=0}^{h} M^i\log^2(M^i))) \\
&= O(|S|t_{f_C}(\log^2(n)\frac{M^h-1}{M-1} - 2\log(n)\log(M)\sum_{i=0}^{h} iM^i + \log^2(M)\sum_{i=0}^{h} i^2M^i)) \\
&= O(|S|t_{f_C}(\log^2(n)\frac{M^h-1}{M-1} \\
&\quad -2\log(n)\log(M)\frac{(h+2)M^{h+2} - (h+1)M^{h+1} + 2M^2 + M}{(M-1)^2} \\
&\quad +\log^2(M)\sum_{i=0}^{h} i^2M^i)) \\
&= O(|S|t_{f_C}(\log^2(n)M^h - 2\log(n)M^h + h^2M^h)) \qquad (3.11)
\end{aligned}$$

which may be simplified to

$$t_{\text{TGSBulkLoadChunk}} = O(|S|t_{f_C}(\log^2(n)M^h + h^2M^h)) \qquad (3.12)$$

Now for the main TGSBulkLoad algorithm, A.4.1, we see that h is initially set to $\max(0, \lceil\frac{\log n}{\log M}\rceil)$. Note that as such, $h = O(\log_M n)$. Based on equation 3.10, the worst-case construction time for TGS bulk-loading is of the following asymptotic order:

$$t_{\text{TGSBulkLoad}} = O(|S|t_{f_C} n\log^2(n)). \qquad (3.13)$$

3.2.3.1. *HilbertTGS.* HilbertTGS is a special application of the basic TGS bulk-loading algorithm in the sense that we use only one ordering, the Hilbert ordering, to initially sort the array. Hence $|S| = 1$ and the worst-case construction time of the HilbertTGS algorithm seems to be of order $O(n\log^2 n)$. However, due to the modifications introduced in Algorithm A.5.1 and A.5.2, this order may not be as sharp as possible. We therefore analyse the entire HilbertTGS bulk-loading algorithm again, using results from the previous section when needed.

In order for the quicksort algorithm to perform accurately, we first have to calculate the Hilbert values of the MBR of each object we wish to insert. This takes $n \cdot t_{toHilbert}$ time, where n is the total number of elements to add in the tree, and $t_{\text{toHilbert}}$ is time required for transforming a coordinate in $\mathbb{R}^d$ to its Hilbert value. We assume here tables are known for any dimension d,

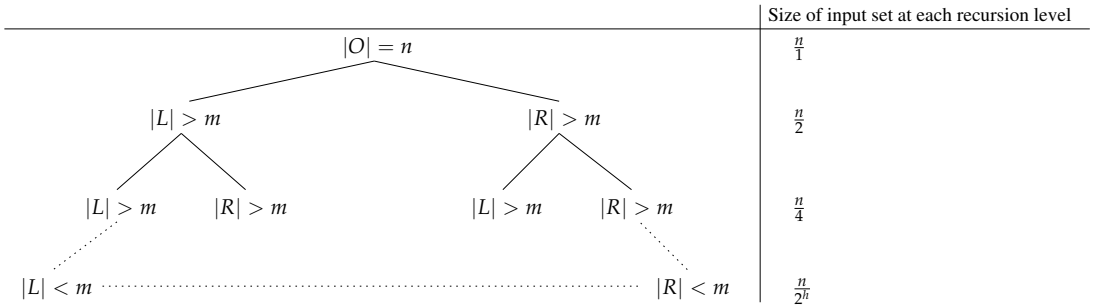


FIGURE 3.2. A call tree denoting the recursion of the TGSPartition algorithm and the progression of the number of input elements at each recursive step.

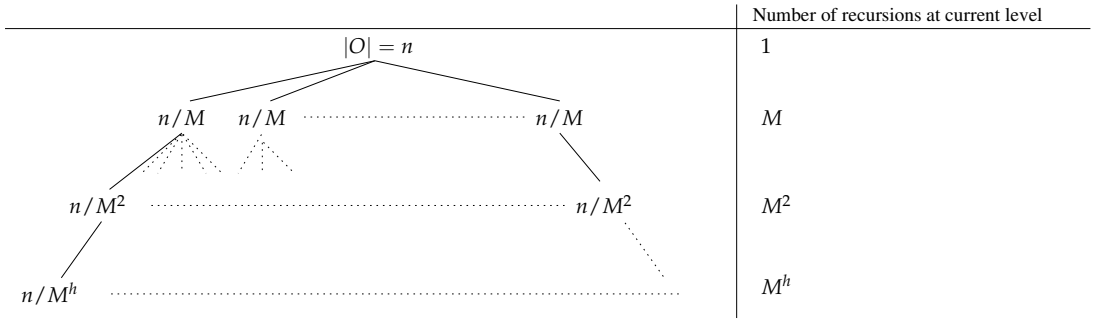


FIGURE 3.3. A call tree denoting the recursion of the TGSBulkLoadChunk algorithm, in much the same way as figure 3.2 did for the TGSPartition subalgorithm.

to obtain an dimension-independent analysis; in the experimentation program however, we only implemented methods for $d = 2$ and $d = 3$.

It is easy to see that the running time of the actions taken in line 3 and 7 of Algorithm A.6.1 take $O(2^d \cdot d)$ time ($d > 1$); this is because we loop over each cell (of which there are 2^d) and apply Algorithm A.6.2, which takes $O(d)$ time. Algorithm A.6.3 which is called on line 6 takes $O(d)$ time as well[3].

Algorithm A.6.4 converts a base 8 fractional number to base 10. This algorithm loops over the number of digits in the input number. This is equal to the order of the Hilbert curve used, which we will denote by l. Hence Algorithm A.6.4 takes $O(l)$ time, and the entire Hilbert coordinate transformation thus takes $O(d + l(2d - 1 + d) + l)$. Or, simplified:

$$t_{\text{toHilbert}} = O(d(1 + 3l)) = O(3dl) \tag{3.14}$$

Hence the initial sort in Algorithm A.5.1 takes

$$t_{\text{HilbertInitialSort}} = O(n(3dl + \log n)) \tag{3.15}$$

time[4]. Another difference with the original TGS algorithm occurs in Algorithm A.5.2, which is the replacement function of the original TGSBestBinarySplit algorithm. The only difference is that the algorithm does not sort on-the-fly any more. Thus, judging from equation 3.9, we have:

[3] Assuming the operations on line 1 and 2 of take $O(1)$ Algorithm A.6.3 time, which is the case when optimised correctly.

[4] Assuming applying the quicksort algorithm will take $O(n \log n)$ time.

$$t_{\text{HilbertBestBinarySplit}} = O(t_{f_C}^-) \tag{3.16}$$

This affects the expected running time of Algorithm A.4.3 when used for HilbertTGS as well:

$$\begin{aligned} t_{\text{HilbertTGSPartition}} &= \sum_{i=0}^{\log_2 \tilde{n}/m-1} \left[\sum_{j=0}^{2^i} O(\frac{t_{f_C}}{2^i} m) \right] \\ &= \sum_{i=0}^{\log_2 \tilde{n}/m-1} O(m \cdot t_{f_C}) \\ &= O(\log \frac{\tilde{n}}{m-1} m \cdot t_{f_C}) \\ &= O(m t_{f_C} \log \tilde{n}). \end{aligned} \tag{3.17}$$

Hence the running time for the BulkLoadChunk algorithm also differs from the original TGS running time:

$$\begin{aligned} t_{\text{HilbertTGSBulkLoadChunk}} &= M + \sum_{i=1}^{h} M^i (O(\frac{n}{M^{i+1}} t_{f_C} \log n/M^i) + M) \\ &= O((h+1)M + \frac{n}{M} t_{f_C} (h \log n - \sum_{i=1}^{h} \log M^i)) \\ &= O((h+1)M + \frac{n}{M} t_{f_C} (h \log n - \sum_{i=1}^{h} i \log M)) \\ &= O((h+1)M + \frac{n}{M} t_{f_C} (h \log n - \frac{h(1+h)}{2} \log M)) \\ &= O(n t_{f_C} h \log n) \end{aligned} \tag{3.18}$$

Now, the main HilbertTGS bulk-loading algorithm A.5.1 sets $h = \max(0, \lceil \frac{\log n}{\log M} \rceil)$ and thus:

$$\begin{aligned} t_{\text{HilbertTGS}} &= t_{\text{HilbertInitialSort}} + t_{\text{HilbertTGSBulkLoadChunk}} \\ &= O(ndl + \log n + t_{f_C} nh \log n) \\ &= O(ndl + \log n + t_{f_C} nh \log n) \\ &= O(ndl + t_{f_C} n \log^2 n), \end{aligned} \tag{3.19}$$

which is of the expected order given at the start of this section, assuming dl is far less than $\log^2 n$.

3.2.4. Hilbert R-tree. Since we now have calculated $t_{\text{toHilbert}}$, we can easily calculate the order of Hilbert R-tree construction. For convenience, we repeat equation 3.14:

$$t_{\text{toHilbert}} = O(3dl).$$

The Hilbert R-tree behaviour differs with the basic R-tree in that it handles under- and overflows using Hilbert coordinates by merging and applying s to $s+1$-splitting respectively; see section 2.6.2. Let us start with analysing the overflow algorithm A.7.4.

A costly operation there is found on line 16; it requires visiting each object stored in the current node, and this has to be done for all nodes in $\tilde{S}$. The total number of elements is sM, worst case; when each node involved in the s to $s+1$ split has the maximum number of elements. Also,

calculating a single MBR is a $O(d)$ operation, and calculating its Hilbert value takes $O(dl)$ time. Thus the loop in 14-17 takes $O(s^2Mdl)$ time. which dominates the remainder of the algorithm.

Another costly operation is sorting all elements in $\tilde{S}$. This takes $O(sM \log sM) = O(sM(\log s + \log M))$ time. It is unclear if this is of lower order than $O(s^2Mdl)$, so we take them together to obtain

$$O(sMdl(s + \log sM)).$$

Take note that the Hilbert R-tree overflow algorithm bubbles upward, if the parent node also overflows. Since the tree height is of order $O(\log n)$, the total time required for the overflow algorithm is, worst case:

$$t_{\text{HilbertOverflow}} = O(sMdl \log(n)(s + \log sM)) \tag{3.20}$$

On a side note, we have that the underflow algorithm A.7.6 has the same complexity as the overflow algorithm. This is because the need for sorting and redistributing is exactly the same as that of the overflow algorithm. We thus also have:

$$t_{\text{HilbertUnderflow}} = O(sMdl \log(n)(s + \log sM)) \tag{3.21}$$

The HilbertUpdate algorithm A.7.3 largely has the same complexity as well, save for the need to sort the elements at each node. Also, the algorithm only bubbles upward and does not visit any sibling nodes. Thus we obtain:

$$t_{\text{HilbertUpdate}} = O(Mdl \log(n)) \tag{3.22}$$

Analysing the HilbertChooseLeaf algorithm A.7.2 we see that the main difference lies in the fact that we do not select on MBRs, but on LHV-values, thus losing the running time dependency on the number of dimensions d. However, on leaf nodes we have to calculate the Hilbert coordinate values in $O(ld)$ time per leaf node. Hence:

$$t_{\text{HilbertChooseLeaf}} = O(M(\log(n) + ld)) \tag{3.23}$$

Now we are able to put it all together and obtain the following running time order for inserting a single element in an Hilbert R-tree:

$$\begin{aligned} t_{\text{HilbertInsert}}(n) &= t_{\text{HilbertChooseLeaf}} + t_{\text{HilbertUnderflow}} + t_{\text{HilbertUpdate}} \\ &= O(M(\log(n) + ld) + sMdl \log(n)(s + \log sM) + Mdl \log(n)) \\ &= O(M(\log(n) + ld(1 + \log(n)(s(s + \log sM) + 1)))) \\ &= O(sldM \log(n)(s + \log sM)) \end{aligned} \tag{3.24}$$

When inserting n elements into an Hilbert R-tree one-by-one, we thus obtain the following construction time:

$$
\begin{aligned}
t_{\text{HilbertConstruction}} &= \sum_{i=1}^{n} O(sldM \log(i)(s + \log sM)) \\
&= O(sldM(s + \log sM) \sum_{i=1}^{n} \log(i)) \\
&= O(sldM(s + \log sM) \log(n!)) \\
&= O(sldMn(s + \log sM) \log n), \qquad (3.25)
\end{aligned}
$$

which is of the same order as the basic R-tree construction time.

3.3. Summary

The following table summarises the results presented in this chapter:

Datastructure	Asymptotic construction time
Bisection tree	$n \log n$
Hilbert R-tree	$n \log n$
Basic R-tree	$n \log n$
HilbertTGS	$n \log^2 n$
TGS[a]	$n\|S\| \log^2 n$

[a] With, standardly, $|S|$ equal to the number of dimensions d.

3.4. Querying

3.4.1. GSA Analysis. We will now analyse Algorithm 2.1 to get a feel of the general performance of spatial trees. Assuming correctness of the tree with respect to the functions f_v, we can simplify analysis as follows. We assume operations such as retrieving elements or children from a given node are $O(1)$ operations. We also assume the union operation on two sets is an $O(1)$ operation[5].

At each call of the GSA on a given node v, all elements stored at v are visited (lines 2-6). Remember that the number of elements stored at v is denoted by $|v_O|$. Then the running time $T(v)$ up until line 6 is $O(1 + 3|v_O|) = O(|v_O|)$.

The algorithm proceeds by visiting all children of v on which it evaluates f_v and compares its result to 1; so $T(v)$ up until line 8 is $O(|v_O| + |v_C|)$ (with $|v_C|$ the number of children of v). At line 9, things get complicated; on basis of the result at line 8, GSA may be recursively called here.

If we assume that the time needed for the GSA algorithm on the ith child of v, denoted by v_i, to complete is equal to $T(v_i)$, then $T(v) = O(|v_O| + |v_C| + \sum_{i=1}^{|v_C|} f_{v_i}(x)T(v_i))$. Let $a = \max_{v \in V} |v_O|$ and $b = \max_{v \in V} |v_C|$, then $T(v)$ has an upper bound of order

$$
O(a + b + \sum_{w \in v_C} f_w(x)T(w)). \qquad (3.26)
$$

Applying this result to the bisection tree structure, we obtain the following theorem.

THEOREM 3.8 (Bisection Tree GSA performance). *The Generic Search Algorithm 2.1 applied on a Bisection Tree runs in $O(n + h)$ time, worst case.*

PROOF. We have a generic running time of $T(r) = O(a + b + \sum_{v \in r_C} f_v(x)T(v))$, with r the tree root. As per construction of the bisection tree, there is absolutely *no* overlap between siblings. Also, b is fixed at 3, while a may vary. Elements are only stored at the leaf nodes, thus for any

[5] This can for example be realised by using a linked list.

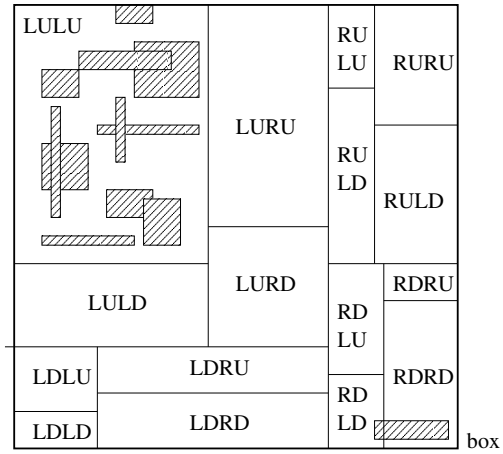


FIGURE 3.4. Illustration in two dimensions of a badly constructed bisection tree with $h = 4$. Each rectangle corresponds to a leaf node; leaf node LULU (left-up-left-up) contains all elements in the box, except for one which is contained in RDM (right-down-mid, the node containing the elements on the bisection line between RDL and RDR).

non-leaf we have $a = 0$. The tree depth is fixed at h. If we assume, without loss of generality, that $h \geq 2$, we can obtain:

$$\begin{aligned}
T(r) &= O(a + b + \sum_{v \in r_C} f_v(x)T(v)) \\
&= O(b + \sum_{v \in r_C} f_v(x)T(v)), \quad (a = 0 \text{ for internal nodes.}) \\
&= O(b + T(v)), \quad \text{for some } v \in r_C \\
&= O(b + (b + T(\tilde{v}))), \quad \text{for some } \tilde{v} \in v_C, v \in r_C \\
&= O([\sum_{i=0}^{h-1} b] + a) \\
&= O((h-1)b + a)
\end{aligned}$$

Then, worst case we have that $a = n$, resulting in a running time of $O(h + n)$. □

The worst case scenario happens when all elements are mapped onto a single leaf element. This only happens when the bisecting hyperplanes were chosen poorly enough so that (nearly) all elements are continuously found at a single side of any such plane. Figure 3.4 illustrates such a bad case. We see that querying the bottom-right elements would take time proportional to the tree height (4), while querying for any element in the upper-left rectangle (LULU) would take time proportional to both the tree height, and nearly all elements in the complete dataset (11 out of 12).

Note that this worst-case scenario is worse than when we directly would check all elements for intersection; this would be independent of trees and thus would run in $O(n)$ time, losing the dependency on the tree height. However, if we assume that the number of elements stored at each leaf node is much smaller than n, as would be the case when the bisecting hyperplanes are chosen well enough, the asymptotic expected running time for querying the bisection tree is $O(h)$.

3.4.2. Point Query. We will now determine the worst case, best case and average running times of a single random point query on a general R-tree. Let us be given a valid R-tree $R = (r, V, E)$ with a minimum number of elements or children m and a maximum M. Also, let the height of R be given by h, and the number of elements stored in R be given by n. As mentioned before, the point query is a special case of the earlier analysed GSA; however, we will now analyse the point query using the Master Theorem (3.4) to obtain a more thorough analysis.

To easily use this theorem, we will utilise an indexing scheme on the nodes of R as given in figure 3.5. This indexing gives us the ability to distinguish between nodes on different levels on the R-tree by their superscript indexes. To make analysis easier, the leaf nodes v_i^h in this representation denote the *objects* stored at the leaf nodes of the original tree R; the leaf nodes now each correspond to a single stored element, instead of the tree node storing a number of objects. The variables n_i, for $0 \leq i \leq h$ denote the number of nodes at the given tree level. It is easily seen that n_i is bounded as follows:

$$m^i \leq n_i \leq M^i, \quad 0 \leq i \leq h-1, \tag{3.27}$$

and $n_h = n$.

In the case of an R-tree, the tree height is also bounded by the variables m and M, as follows:

$$\lceil \log_M n \rceil \leq h \leq \lceil \log_m n \rceil \tag{3.28}$$

We will make no further assumptions on the R-tree from here on out; we will try to achieve an analysis as general as possible. Assume the point-query algorithm called on a node $v \in V$ has a running time of $\tilde{T}(v)$. If the node v is a leaf node, which for the sake of this analysis now corresponds to a single stored object, returning that element takes $O(1)$ time; $\tilde{T}(v_j^h) = O(1)$ for all j. Also, we see that an recursion is given by $\tilde{T}(v_j^i) = O(M + \sum_{v_k^{i+1} \in v_{j_C}^i} f(v_k^{i+1})\tilde{T}(v_k^{i+1}))$. These relations are still not completely in the form of equation 3.1, but this can be overcome by shifting from a running time function $\tilde{T}$ based on nodes, to a function T based on the number of objects stored in the subtree with those nodes as root. Let us further examine this.

The root node v_0^0 of course stores exactly n elements. Since we are not to make any assumptions on tree specifics, we can only proceed with bounds for the problem sizes on lower levels: the subtrees of the nodes at depth 1 each store between $\lceil n/M \rceil$ and $\lceil n/m \rceil$ objects. Generally, the nodes at depth i store in a number of elements $e_i \in [\lceil \frac{n}{M^i} \rceil, \lceil \frac{n}{m^i} \rceil] \subset \mathbb{Z}$. With this information we can derive two bounds for the running time function T:

$$\begin{aligned} T_1(1) &= O(1) \\ T_1(n) &= O(\lceil p_{\text{point}} \cdot M \rceil \cdot T(\lceil n/M \rceil) + M) \end{aligned} \tag{3.29}$$

and

$$\begin{aligned} T_2(1) &= O(1) \\ T_2(n) &= O(\lceil p_{\text{point}} \cdot m \rceil \cdot T(\lceil n/m \rceil) + m) \end{aligned} \tag{3.30}$$

with $p_{\text{point}} \in [0, 1] \subset \mathbb{R}$; p_{point} denotes the average probability that the point-query algorithm will recurse in one of the children of an internal node. This representation is somewhat flawed, since p_{point} in reality varies for each tree node. The variable p_{point} describes in essence the R-tree quality; for large values, the algorithm recurses into many children and for the smaller values of p_{point} the algorithm recurses into relatively few children on each tree level. Given that each

node has in between m and M children, it seems to be reasonable to assume that indeed T_1 and T_2 bound T:

$$\forall n \in \mathbb{Z} : T(n) \in [T_1(n), T_2(n)] \tag{3.31}$$

In terms of Theorem 3.4, we find the factors $t_1 = \frac{\log \lceil p \cdot M \rceil}{\log M}$ and $t_2 = \frac{\log \lceil p \cdot m \rceil}{\log m}$ and the functions $f_1(n) = O(M)$ and $f_2(n) = O(m)$.

THEOREM 3.9. *Worst-case point query time. For single random point-queries, the running time $T(n)$ is of order $\Theta(n)$, worst case.*

PROOF. Worst case, we recurse over all nodes; i.e., $p_{\text{point}} = 1$. As such we have:

$$t_1 = 1, \quad \text{and } t_2 = 1$$

We start with the first bound. Since $f_1(n) = O(M) = \Theta(1) = O(1)$, letting $\epsilon = 1$ results in $f_1(n) = O(n^{t-\epsilon}) = O(1)$. As such, case 1 of Theorem 3.4 applies and we obtain $T_1(n) = \Theta(n)$. Analogously, a similar bound is achieved for $T_2(n)$ and as such, under assumption of equation 3.31, $T(n) = \Theta(n)$, worst case.

□

THEOREM 3.10. *Best-case point query time. For single random point-queries, the running time $T(n)$ is in $\Theta(\log(n))$, best case.*

PROOF. Best case, we recurse over only one node; $\lceil p_{\text{point}} m \rceil = \lceil p_{\text{point}} M \rceil = 1$. Thus,

$$t_1 = 0, \quad \text{and } t_2 = 0$$

$f(n) = \Theta(1) = \Theta(n^{t_1})$ so case 2 of the Master theorem applies, resulting in $T_1(n)$ being in $\Theta(\log(n))$. This also applies for $T_2(n)$ and as such, $T(n) = \Theta(\log(n))$.

□

Since there exist many variations of R-trees, all resulting in different tree build-ups, trying to derive an average running time seems an hopeless task. However, the way in which different R-tree variations query is different solely in which nodes, and in how many nodes, the point-query algorithm has to recurse. This is reflected in the variables t_1 and t_2 by the parameter p_{point}. Obviously, when $\lceil p \cdot M \rceil > 1$ (not best-case), a situation similar to the worst case bound arises. Since the f functions are still asymptotically constant, case 1 always applies, and as such:

$$\begin{aligned} T_1(n) &= \Theta(n^{\frac{\log \lceil p_{\text{point}} \cdot M \rceil}{\log M}}) \\ T_2(n) &= \Theta(n^{\frac{\log \lceil p_{\text{point}} \cdot m \rceil}{\log m}}) \end{aligned}$$

Hence, if for some R-tree variation the factor p_{point} is known, we can estimate the query performance using the above bounds. Otherwise, we have to make due only with the knowledge the asymptotic running time is somewhere in between the best and worse case presented earlier. Note that p_{point} at a given node v is directly dependent on the volumes of the bounding boxes BB_i; $p_{\text{point}} = \frac{\text{Volume}(\cup_i BB_i)}{\text{Volume}(v_{\text{MBR}})}$.

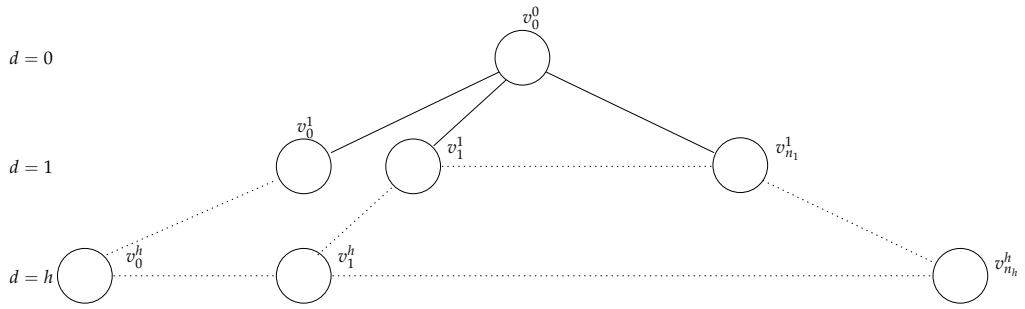


FIGURE 3.5. An indexing scheme for the R-tree nodes. The variable d denotes the tree depth.

3.4.3. Line query. For line queries, we can calculate bounds as was done for point queries by using a parameter p_{line} instead of p_{point}. Note that it is a lot more likely for a line query to intersect with multiple bounding boxes at a given internal node than when doing a point query; the chance of intersection of an arbitrary bounding box with a random line segment is larger than intersection with a single random point. When the line segment is quite small with respect to the bounding boxes, we still obtain the same best case query time as presented for point queries; we then still recurse into only one child at each tree level. Also, it is clear that worst-case we still obtain the same bound as in Theorem 3.9; we still cannot do worse than recurse into all children at each tree level.

Hence the only interesting question here is how much more greater is p_{line} than p_{point}, when the underlying tree structure is unchanged? This question is equivalent to asking what the ratio $\frac{P(l \cap B \neq \emptyset)}{P(x \in B)}$ is, with B an arbitrary bounding box contained in the unit bounding box U, x a random point in U, and l a random line segment in U. As mentioned earlier, $P(x \in B)$ equals $\frac{\text{Volume}(B)}{\text{Volume}(U)} = \text{Volume}(B)$. We will now proceed with calculating $P(l \cap B \neq \emptyset)$.

Let us for this purpose define a random line segment in a more precise manner.[6]

DEFINITION 3.11. *Let l_1, l_2 be random in $\mathbb{R}^d$. Then, a line segment l consists of all points x contained in $\{l_1 + t(l_2 - l_1)\}$, $t \in [0,1]$, $l_1, l_2 \in \mathbb{R}^d$.*

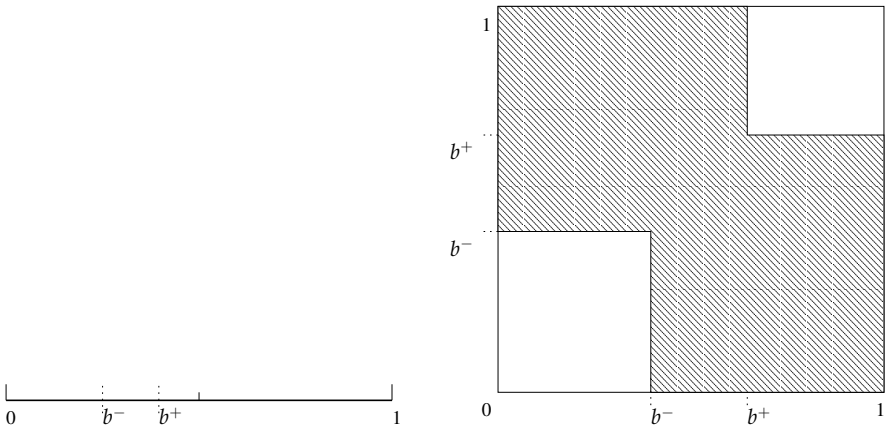
Without loss of generality, assume we have some bounding box B with minimum coordinate $b^- \in [0,1]^d$ and maximum coordinate $b^+ \in [0,1]^d$ with $b_i^- \leq b_i^+$ for $i = 1, \ldots, d$. B is contained in the unit bounding box U.[7]

Let us consider first the case of $d = 1$. The resulting problem is visualised in 3.figure 6(a). Consider a plot with the position of l_1 on the horizontal axis and l_2 on the vertical axis, where the area resulting in intersection is marked. Such a plot is easily constructed for the one-dimensional case and is shown in 3.figure 6(b). From this figure, we derive a function $f_{P(l \cap B \neq \emptyset)}(b^-, b^+)$ denoting the probability of intersection, given a bounding box $B \cap U$:

$$\begin{aligned} f_{P(l \cap B \neq \emptyset)}(b^-, b^+) &= 1 - (b^-)^2 - (1 - b^+)^2 \\ &= 2b^+ - (b^-)^2 - (b^+)^2, \end{aligned} \tag{3.32}$$

[6] Note that actually, one can define random line segments in multiple ways. One can for example randomly determine a line centre point, randomly generate a line length and an angle α (or multiple angles when in more than two dimensions) which the line makes with respect to the x-axis; this also results in a perfectly randomly generated line. However, these two examples do *not* result in the same distribution of random lines. The average line length, for example, differs by almost a factor two [**Ros04**].

[7] i.e., U has minimum coordinate $(0, \ldots, 0) \in \mathbb{R}^d$ and maximum coordinate $(1, \ldots, 1) \in \mathbb{R}^d$.



(a) The problem of a line intersecting an interval, or rather, a one-dimensional box.

(b) The probability of intersection visualised in a two-dimensional graph. The x-axis denotes the value of l_1 and the y-axis the value of l_2. The filled area represents when the line segment between l_1 and l_2 intersects the interval between b^- and b^+.

FIGURE 3.6. Some figures illustrating line-box intersection in the one-dimensional case.

assuming, of course, that $b^- \leq b^+$. This case can be easily extended into more dimensions. To this end, consider for example the two-dimensional case in figure 7(a). Now imagine we choose some position y along the vertical axis, and an angle $\phi \in [0, \pi]$. We then construct a line $l(y, \phi)$ as in figure 7(b). Let the line length $l(y, \phi)$ from y to the edge of U be denoted by s. The line may intersect B on the way to the edge; hence we can define q as the distance on $l(y, \phi)$ from y to the first edge of B hit, and r the distance from the last edge of B hit until the edge of U. See figure 7(c).

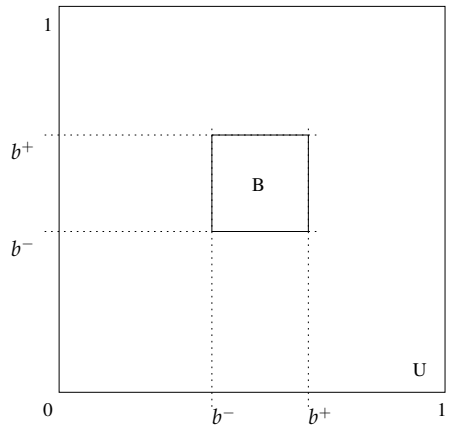
Now suppose the line segment constructed by the points l_1 and l_2 is a subset of $l(y, \phi)$ for some y and ϕ. Then, the probability $f_{P(l \cap B | l \subset l(y,\phi))}(l, \phi)$ that this line segment intersects B is, according to equation 3.32, equal to $\frac{2r - q^2 - r^2}{s}$. Thus the probability any line segment hits B equals:

$$\int_0^\pi \int_0^1 f_{P(l \cap B | l \subset l(y,\phi))}(l, \phi) dy d\phi = \int_0^\pi \int_0^1 \frac{2r(y,\phi) - q^2(y,\phi) - r^2(y,\phi)}{s(y,\phi)} dy d\phi \tag{3.33}$$

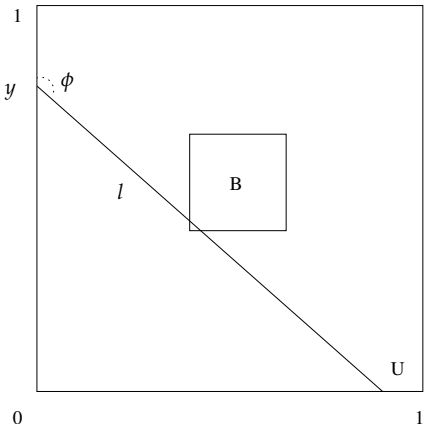
Evaluating this integral is not straightforward. It requires distinguishing multiple cases when integrating over ϕ, as can be seen in figure 7(d). These cases depend on the value y. As such, calculating the two-dimensional case would be a tedious task, and calculating it for three dimensions or more is even harder. This, combined with the fact that we rather have a result applicable for all dimensions, is the reason why we will work with an upper bound on the probability we wish to calculate. This upper bound is derived as follows.

Consider figure 3.8. Note that l cannot intersect B if both l_1 and l_2 are both in the first or third column or row. This notion generalises quite simply to higher dimensions; on each dimension we have three different intervals, and if l_1 and l_2 fall in the same interval on any dimension, intersection is not possible. Hence the probability for no intersection is bounded as follows:

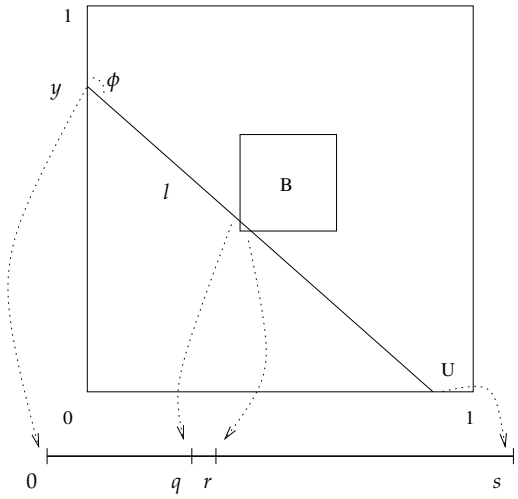
$$P(l \cap B = \emptyset) \geq 1 - \Pi_{i=0}^{d-1} (b_i^+)^2 (1 - 2b_i^- + (b_i^-)^2) \tag{3.34}$$



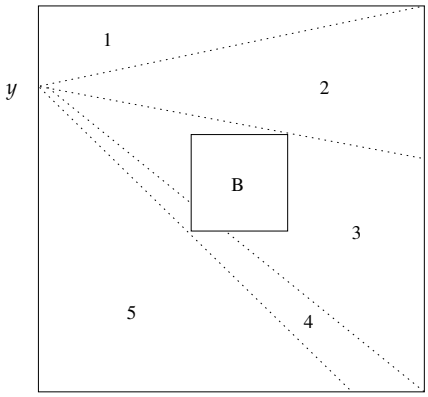
(a) The problem of a line intersecting a rectangle.



(b) Figure 7(a) with the line $l(y, \phi)$.



(c) Figure 7(b) with the variables q, r, s, as in the text.



(d) The five different cases (depending on ϕ) required to evaluate equation 3.33, for a single value y fixed.

FIGURE 3.7. Some figures illustrating line-box intersection in the two-dimensional case.

and hence the probability that l does intersect B is at most:

$$P(l \cap B \neq \emptyset) \leq \Pi_{i=0}^{d-1}(b_i^+)^2(1 - 2b_i^- + (b_i^-)^2) \tag{3.35}$$

We may also write Volume(B) in terms of b^- and b^+:

$$\text{Volume}(B) = \Pi_{i=0}^{d-1} b_i^+ - b_i^-, \quad \text{and thus,}$$

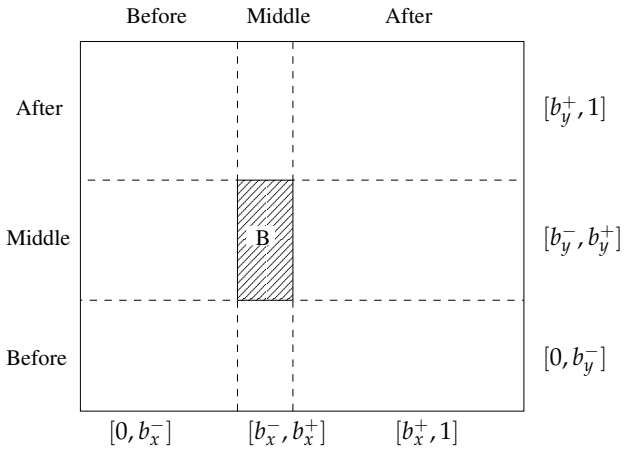


FIGURE 3.8. Two dimensional case of a three-way interval split on each dimension, along the boundaries of bounding box B

$$P(x \in B) = \frac{\text{Volume}(B)}{\text{Volume}(U)} = \Pi_{i=0}^{d-1} b_i^+ - b_i^- \tag{3.36}$$

Given 3.35, an upper bound for the increased difficulty of line queries with respect to point queries is as follows:

$$\frac{P(l \cap B \neq \emptyset)}{P(x \in B)} \leq \frac{\Pi_{i=0}^{d-1}(b_i^+)^2(b_i^- - 1)^2}{\Pi_{i=0}^{d-1} b_i^+ - b_i^-} \tag{3.37}$$

3.4.4. Box query. As in the previous subsection, the best and worst case scenarios as for point queries still hold, although only for small enough query boxes in case of the best case query time. We will again proceed with determining how much more complicated box queries are with respect to point queries. We will therefore calculate the ratio $\frac{P(Q \cap B)}{P(x \in B)}$, with x an random point, B any bounding box contained in U and Q the random query box, also contained in U. We firstly precisely define an exact notion of a random bounding box.

DEFINITION 3.12. *Let* $(x_1^1, x_2^1, \ldots, x_d^1, x_1^2, x_2^2, \ldots, x_d^2)$ *be independently uniformly distributed in* $[0,1]^{2d}$. *Let* $q^- = (q_1^-, q_2^-, \ldots, q_d^-)$ *with* $q_i^- = \min(x_i^1, x_i^2)$ *and* $q^+ = (q_1^+, q_2^+, \ldots, q_d^+)$ *with* $q_i^+ = \max(x_i^1, x_i^2)$. q^- *and* q^+ *define the minimum and maximum bound respectively for a* random bounding box Q.

In contrast to a line query, we are able to quickly determine the exact probability $P(Q \cap B)$ as follows. On each dimension, we define three intervals as inspired by figure 3.8; the interval *'before' B* ($[0, b_i^-]$), the *middle* interval on B ($[b_i^-, b_i^+]$) and the interval *'after' B* ($[b_i^+, 1]$), with b^- and b^+ as in Definition 3.12. We see that Q does intersect B if and only if for each dimension i, we have that q_i^- is not in the same interval (the before, middle, after intervals from figure 3.8) as q_i^+, unless it is in the middle interval.

So, the probability of intersection of a random bounding box Q with any bounding box B is derived as follows. First we begin by deriving the chance that for any dimension i, q_i^- and q_i^+ are in opposite intervals, given that they are not in the middle interval. Let $x, y \sim \mathcal{U}([0, 1 - b_i^+ + b_i^-])$, that is, x and y are uniformly random in $[0, 1 - b_i^+ + b_i^-]$. Write $I_b = [0, b_i^-]$, $I_m = [b_i^-, b_i^+]$ and $I_a = [b_i^+, 1]$. Then:

$$
\begin{aligned}
p_{\text{opp}} &= P(q_i^-, q_i^+ \text{are in opposite intervals} | q_i^-, q_i^+ \notin I_m) \\
&= P((q_i^- \in I_b \wedge q_i^+ \in I_a) \vee (q_i^- \in I_a \wedge q_i^+ \in I_b) | q_i^-, q_i^+ \notin I_m) \\
&= P(q_i^- \in I_b \wedge q_i^+ \in I_a | q_i^-, q_i^+ \notin I_m) \\
&= P((x \in [0, b_i^-] \wedge y \in [b_i^-, 1 - b_i^+]) \vee (x \in [b_i^-, 1 - b_i^+] \wedge y \in [0, b_i^-])) \\
&= 1 - (1 - P(x \in [0, b_i^-]) P(y \in [b_i^-, 1 - b_i^+])) (1 - P(x \in [b_i^-, 1 - b_i^+]) P(y \in [0, b_i^-])) \\
&= 1 - (1 - P(x \in [0, b_i^-]) P(y \in [b_i^-, 1 - b_i^+]))^2 \\
&= 1 - (1 - \frac{b_i^- \cdot (1 - b_i^+)}{(1 - b_i^+ + b_i^-)^2})^2 \qquad (3.38)
\end{aligned}
$$

The probability that neither q_i^- or q_i^+ are in I_m can be calculated also:

$$
\begin{aligned}
p_{\notin I_m} &= P(q_i^- \notin I_m \wedge q_i^+ \notin I_m) \\
&= P(q_i^- \in [0, b_i^-] \cup [b_i^+, 1]) P(q_i^+ \in [0, b_i^-] \cup [b_i^+, 1]) \\
&= (1 - \frac{b_i^+ - b_i^-}{1 - b_i^-})^2 \qquad (3.39) \\
&= \left(\frac{1 - b_i^+}{1 - b_i^-} \right)^2 . \qquad (3.40)
\end{aligned}
$$

Combining equations 3.38 and 3.40 we obtain the final result:

$$
\begin{aligned}
P(Q \cap B) &= \Pi_{i=0}^{d-1} P(q_i^- \in I_m \vee q_i^+ \in I_m) + p_{\notin I_m} \cdot p_{\text{opp}} \\
&= \Pi_{i=0}^{d-1} (1 - p_{\notin I_m}) + p_{\notin I_m} \cdot p_{\text{opp}} \\
&= \Pi_{i=0}^{d-1} (1 - (1 - \frac{b_i^+ - b_i^-}{1 - b_i^-})^2) + (1 - \frac{b_i^+ - b_i^-}{1 - b_i^-})^2 \cdot (1 - (1 - \frac{b_i^- \cdot (1 - b_i^+)}{(1 - b_i^+ + b_i^-)^2})^2) \\
&= \Pi_{i=0}^{d-1} (1 - (\frac{1 - b_i^+}{1 - b_i^-})^2) + (\frac{1 - b_i^+}{1 - b_i^-})^2 \cdot (1 - (1 - \frac{b_i^- \cdot (1 - b_i^+)}{(1 - b_i^+ + b_i^-)^2})^2) \qquad (3.41)
\end{aligned}
$$

3.4.5. Summary. The best and worst case point-query times have been given in Theorem 3.10 and 3.9, respectively. It has been argued that for small enough line segments and small enough boxes, this logarithmic best-case running time can also be achieved for line- and box-queries. Naturally, by the simple fact that an R-tree cannot do worse than recursing into all nodes as in the worst-case point-query case, the linear worst-case point-query running time also applies for line- and box-queries.

When for point-queries we assume we are recursing in a constant number of children at each R-tree node, we arrived in the described average case and achieve a running time of $\Theta(n^t)$, $0 < t \leq 1$, where $t = 1$ obviously is the worst-case linear time. The parameter t is directly dependent on the number of children the point-query search algorithm recurses in.

In the above analysis, we introduced a parameter $p \in [0, 1]$ for determining the average number of children an R-tree would recurse in. In the case of point-queries, p varies per internal node the point-query algorithm is called on and thus is in direct correlation to the quality of the R-tree; we have seen that at node-level, p is dependent on the overlap between MBRs stored at that node.

To determine p for point-queries, we made use of the probability p_{point} that a random point is contained in a given bounding box. For line- and box queries, p is of course also dependent

on node-level MBR overlap, but also on the probabilities p_{line} and p_{box}, which are expected to be significantly larger than p_{point}. Hence by assuming we use the same tree structure for all query types, and thus the amount of node-level overlap remains the same, we have obtained a method for predicting in how much more children line- and box-query algorithms will recurse in with respect to point-queries; p_{point}, p_{line}, and p_{box} indicate relative query difficulty.

A lower bound for p_{line} was found, so that $\frac{p_{\text{line}}}{p_{\text{point}}}$ gives a lower bound on how much more difficult a line query is with respect to the point-query. p_{point} and p_{box} were explicitly calculated. The lower bound for p_{line} and the values for p_{point} and p_{box} may be determined for any number of dimensions.

3.5. Storage

Since an R-tree stores in between m and M objects at each of its nodes, we can easily give bounds for the total memory usage. For the bisection tree, we know that it always has 3 children, one of which is a leaf node (the 'middle' node) and two of which are internal nodes; the order of memory usage thus can be given explicitly. For the special case of packed R-trees, we know that each R-tree node has exactly M children, save for one node if the number of input elements cannot be divided by M exactly. Hence for packed R-trees, the exact order of memory usage is equal to the order of the lower bound of the general R-tree memory usage.

3.5.1. Bisection tree. In effect, a bisection tree is a binary tree with an additional set of elements stored at each node. This additional set then corresponds to the 'middle' child of each node and contains the objects which were intersected by the bisection plane used at that node. The number of nodes in this binary representation is easily determined when we note that for the n_i from figure 3.5 in this case we have that:

$$n_i = 2^i. \tag{3.42}$$

Given this, the number of nodes in a bisection tree equals $\sum_{i=0}^{h} n_i$, where h is the tree height. Remember that h is equal to the number of recursions of the bisection tree construction algorithm, and let us assume that this number is $\lceil \log_2 n \rceil$. Then the total number of nodes equals:

$$\tilde{n} = \sum_{i=0}^{h} n_i = 2^{h+1} - 1. \tag{3.43}$$

Note that this can be easily proven by noting that $\sum_{i=0}^{h} n_i$ is, in binary, the number of h repeated ones: $\tilde{n}_{\text{binary}} = 11\ldots1$. Now we easily see that $\tilde{n}_{\text{binary}} + 1 = 100\ldots0$ which equals 2^{h+1} in the decimal system, which yields the result in equation 3.43.[8]

Let us now determine the minimum storage requirements for a single node. Each node should at least store:

(1) An MBR consisting of two d-dimensional double-precision floating numbers, totalling $2 \cdot d \cdot 64 = 128 \cdot d$ bytes[9][10].
(2) Two pointers to child nodes (in the case of internal nodes), totalling $2 \cdot 32$ bits[11].
(3) Two pointers to arrays containing objects (in the case of leaf nodes), totalling $2 \cdot 32$ bits.

[8] One could also recognise $\sum_{i=0}^{h} 2^i$ as a geometric series.

[9] Assuming a double precision float takes 64 bits of space.

[10] One can argue if a bisection tree node should store an MBR; this is not an integral part of the bisection method. However, using an MBR will speed up queries; it can prevent recursing into subtrees which do not contain any valid objects. Also, the bisection tree implementation by Shell uses MBRs.

[11] Assuming a 32-bit architecture.

(4) One pointer to an array (or part of an array) storing the objects which intersected the bisection plane, totalling 32 bits.

Thus a node in the bisection tree takes about $128 \cdot d + 5 \cdot 32 = 128 \cdot d + 160$ bits of memory, which is linear in the number of dimensions $O(d)$ and constant $O(1)$ otherwise. Given equation 3.43, a bisection tree containing n elements and has height $h = \lceil \log_2 n \rceil$ takes about $(128 \cdot d + 160) \cdot (2^{h+1} - 1) + 32 \cdot n$ bits of memory[12]. Asymptotically, this is of order $O(2^{\lceil \log_2 n \rceil})$.

3.5.2. R-tree. In its most basic state, an R-tree node has the following storage requirements:

(1) A single MBR (also see previous subsection), totalling $128 \cdot d$ bits.
(2) A pointer to an array of children nodes, totalling 32 bits.
(3) An array storing pointers to children nodes, totalling 0 bits (if leaf) or somewhere in between $32 \cdot m$ bits and $32 \cdot M$ bits.
(4) A pointer to an array of locally stored objects, totalling 32 bits.
(5) An array storing object numbers, totalling 0 bits (if non-leaf) or somewhere in between $32 \cdot m$ bits and $32 \cdot M$ bits.
(6) A pointer to the parent node, totalling 32 bits[13].

This totals to in between $128 \cdot d + 32 \cdot (3 + m)$ and $128 \cdot d + 32 \cdot (3 + M)$ bits per node. The greatest number of nodes is achieved when all nodes store exactly m children or objects. Then the n_i from figure 3.5 equal m^i and the total number of nodes can be determined by using the following proposition.

PROPOSITION 3.13. *Let $g, n \in \mathbb{Z}$. Then:*

$$\sum_{i=0}^{n} g^i = \frac{g^{n+1} - 1}{g - 1} \tag{3.44}$$

PROOF. The proof will follow the initial approach for $g = 2$ given in the previous subsection. Let $\tilde{n} = \sum_{i=0}^{n} g^i$ and $\tilde{n}_{\text{in base } g} = 11 \ldots 1$. Note that when $g > 2$, $\tilde{n}_{\text{in base } g} + 1$ no longer equals $100 \ldots 0$; it equals $11 \ldots 2$. To still achieve our desired effect, we therefore pre-multiply $\tilde{n}$ by $\bar{g} = g - 1$; $\bar{g}\tilde{n}_{\text{in base } g} = \bar{g}\bar{g} \ldots \bar{g}$ and so $\bar{g}\tilde{n}_{\text{in base } g} + 1 = 100 \ldots 0$. This means that $\bar{g}\tilde{n} + 1 = g^{n+1}$. Rewriting gives $\tilde{n} = \frac{g^{n+1}-1}{\bar{g}} = \frac{g^{n+1}-1}{g-1}$ and our proof is concluded.[14] □

Now note that the height h of an arbitrary R-tree is fixed dependent on m, M and n due to the height balanced property and the fixed range of the number of children or elements per node. A bound for h was already given earlier in equation 2.7 and equation 2.8. Following these equations we obtain bounds for the total number of nodes in an R-tree:

$$\frac{n^{\lceil \log_M n \rceil - 1} - 1}{M - 1} \leq \tilde{n} \leq \frac{n^{\lceil \log_m n \rceil - 1} - 1}{m - 1}. \tag{3.45}$$

Hence, the memory usage lower bound becomes

$$(128 \cdot d + 32 \cdot (3 + M)) \cdot \frac{M^{\lceil \log_M n \rceil - 1} - 1}{M - 1} \tag{3.46}$$

and the upper bound becomes

[12] Assuming each object is represented by a single integer (identification) number.

[13] While not required for the bisection tree, a pointer to the parent node is a necessity for R-trees due to dynamic updating algorithms (overflow handling).

[14] Alternatively, one could have simply multiplied both sides of the equation by $(g - 1)$ and obtained equality; which is how the sum rule for geometric series is usually proven.

$$(128 \cdot d + 32 \cdot (3 + m)) \cdot \frac{m^{\lceil \log_m n \rceil - 1} - 1}{m - 1}. \tag{3.47}$$

We conclude with presenting the asymptotic orders of these bounds. The lower bound is of order

$$O(M^{\lceil \log_M n \rceil - 1} - 1) \tag{3.48}$$

while the upper bound is

$$O(m^{\lceil \log_m n \rceil - 1} - 1) \tag{3.49}$$

Note that we can further bring back both bounds to $O(n)$, however, the extra information in equations 3.48 and 3.49 will become useful in explaining some of the experimental results we will present later on. Also remember that equation 3.48 is the exact order of a packed R-tree; hence packed R-trees should result in the most memory-efficient solutions.

CHAPTER

4

FOUR

Experiments

We will start this chapter with a brief overview which R-tree variants we chose for experimentation. Since these are but a small subset of the total amount of R-tree variants available, we will also motivate why we chose for these particular variants.

After this motivation, we shall proceed with explaining the testing methodology used for testing the Bisection tree and the R-tree variants. We conclude with three sections for each specific area we wanted to test; construction time, query efficiency and memory usage.

The experimentation software has been written in `C++`, and is kept as generic as possible. The actual data structures have been implemented fully according to the ANSI standard and have been tested to compile under GNU, Intel and Microsoft Visual C++ compilers. Some of the source files used for the actual experimentation however also use some POSIX standard functions for timing, or GNU specific functions for memory allocation statistics.

4.1. Experimentation machines

Experiments have been performed on the following machines:

Rivium. Rivium is a dual core small server-grade machine with 1 gigabyte of main memory. The processor is an *Intel Xeon 5120*, with each core running at 1.86 GHz, at 1066 FSB. The L2 cache size is 4096 kilobyte.

Qire. Qire is a quad core personal computer with 4 gigabytes of main memory. The processor is an *Intel Q6600*, with each core running at 2.4 GHz, also at 1066 FSB. The cores are split in two groups, each sharing an L2 cache of 4096 kilobytes.

G4. The G4 is an iBook G4 laptop, used only for debugging and development. The processor speed is 1.33 GHz, with 512 megabytes of memory and 512 kilobytes of cache memory.

4.2. Selection

The R-trees implemented for experimentation are the following:

- Basic R-tree as per [Gut84], with linear and quadratic split.
- Hilbert R-tree, as per [KF94].
- Top-down greedy split (TGS), as per [GRLL98a]. With two different cost functions, namely:

(1) The sum of the bounding box volumes,
(2) The volume of the intersection of the bounding boxes.

- TGS using the Hilbert ordering.

Also, the original bisection code used by Shell has been imported so that we can fairly compare the results of the bisection method and the R-tree method. We have chosen to implement the Basic R-tree mostly for obtaining a base reading of what R-trees are capable of, a point from which to build and improve R-tree performance by implementing variations.

Choosing which variations to implement and test was not trivial. The Hilbert R-tree was chosen since it was claimed this variation outperforms the R^* tree and other established R-trees in most cases, at the time of its introduction (1994) [KF94]. While not the most up-to-date variation available, the Hilbert R-tree still seemed to be a good variation [MNPT06], [GRLL98a]. In the latter article, the TGS bulk-loading was introduced; claiming it to be able to beat both the STR-tree and the Hilbert R-tree in terms of query efficiency.

While implementing the above variations the idea was formed to use Hilbert coordinate ordering in combination with TGS to reduce building times. Although seemingly a logical variation of TGS, we however did not found any articles about this variant or its performance. Taking into account the expected low implementation cost since both TGS and Hilbert coordinate transformation were already implemented, we chose to investigate this variant as well.

4.3. Measuring performance

As mentioned in Chapter 1, we are mainly interested the construction times, query performances and memory usage of the different methods. To obtain reliable construction time estimates, we have used the `sys/times.h` standard POSIX library to measure only the process time, which should be uninfluenced by other processes which may be running on the experiment machine. The experiment software has been set up so that the construction time includes only the parsing time of the input file, and the time required to build up the requested data structure.

To obtain reliable average query time averages, we perform n random queries for each query type implemented (point, k-nearest-neighbourhood, line, box) and average the total time required for the n queries for each single query type. We thus obtain a average query time for each different query type. n will vary during experiments from 1000 to 10^5.

Memory allocation is measured via the GNU function `malloc_stats()` contained in `malloc.h`. Since this is only available under GNU compilers, memory allocation measurement has been implemented in a separately from the construction and query time experiments, so that the other experiment code stays as portable as possible. `malloc_stats()` not only returns the allocated amount of memory, but also how much of the memory allocated is actually being used after construction of the data structure is complete. This allows us also to measure the memory efficiency of the data structures.

Datasets. We have used the following datasets for experimentation:

Dataset name	Number of non-void blocks	ijk-grid size
CRNRPNTS_1	384	$8 \times 8 \times 6$
datacomplete	8484	$43 \times 44 \times 9$
CRNRPNTS_2	11352	$43 \times 44 \times 6$
geom2	486484	$42 \times 40 \times 377$
geom1	3106719	$272 \times 411 \times 45$

4.4. Varying parameters

In the first few experiments, we vary parameters like the minimum and maximum number of children per node to see what effects these have on the datastructure performances. We also try

to find a sets of parameters which result in good R-tree performances on our selected datasets. In the following sections, we will perform more detailed experiments on those sets of parameters.

4.4.1. Varying m and M.

Parameter	Value
Experiment machine	Rivium
n	1000
Optimisation level	Debug[a]
Hilbert curve order	4
Basic R-tree split method	Quadratic

[a] C++ code compiled in debug mode will result in slower code, but allows for greater run-time control and information gathering. This is especially useful in combination with debug tools such as valgrind or gdb.

The results for varying the minimum and maximum number of children per node for the basic R-tree with quadratic split and the Hilbert R-tree are as in figure 4.1 and figure 4.2. Note that some graphs seem to be missing the first few data points. This commonly happens when those data points were 0 while the data is plotted in log-log scale; $\log 0$ is not defined properly and as such is not plotted. More detailed experiment results may be found in Appendix C.1.

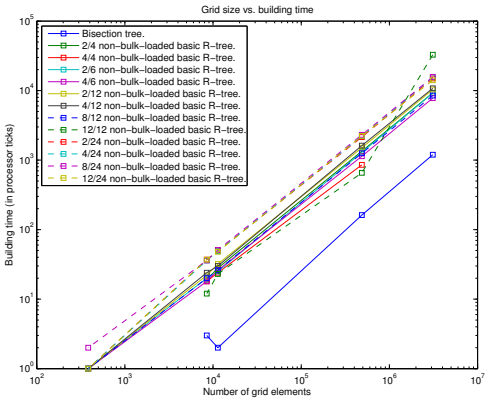
For the bulk-loading algorithms, the minimum number of children does not affect the algorithm; they will result in a fully packed tree. Hence we may obtain the figure 4.3 which only vary M, for different type of bulk-loading schemes. Note that we here supply pictures counting the number of nodes touched on average, during each query. This is another measure of counting the tree efficiency. Save of cache effects, the number of touched nodes should be directly proportional to the execution time; this is researched in section 4.5. These graphs were used instead of the wall-clock time graphs since a lot of query times were averaging to zero, resulting in uninformative graphs.

4.4.2. Varying the Hilbert transformation recursion.

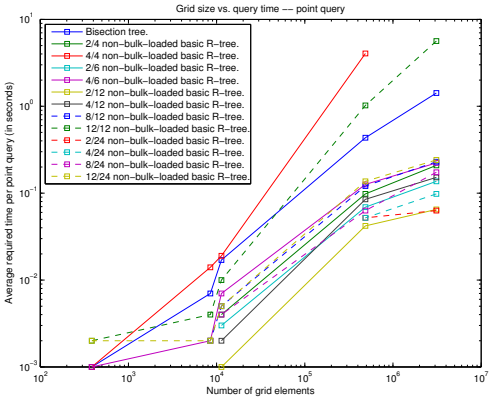
Parameter	Value
Experiment machine	Qire
n	1000
Optimisation level	O3
Minimum number of children	2
Maximum number of children	4
TGS cost function	Sum of volumes

Here we research the effect of changing the Hilbert coordinate transform precision. This experiment of course only applies to the Hilbert R-tree and the HilbertTGS bulk-loading method. The effects on building times are shown in figure 4.4. The effects on query times can be seen in figures 4.5 and 4.6.

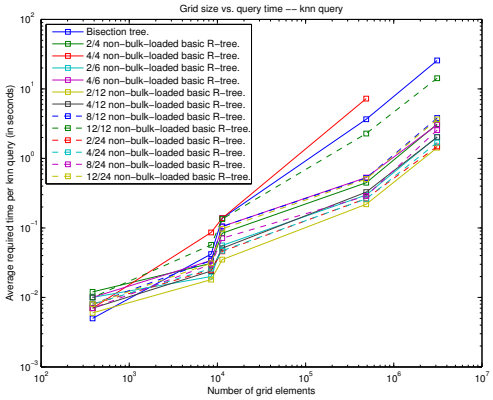
4.4.3. Varying the sampling rate in the Random TGS algorithm.



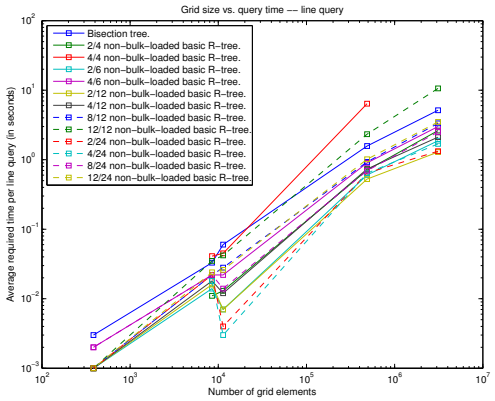
(a) Building time



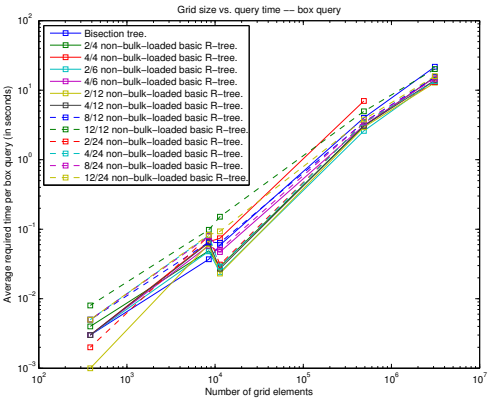
(b) Average point query time



(c) Average k-nn query time

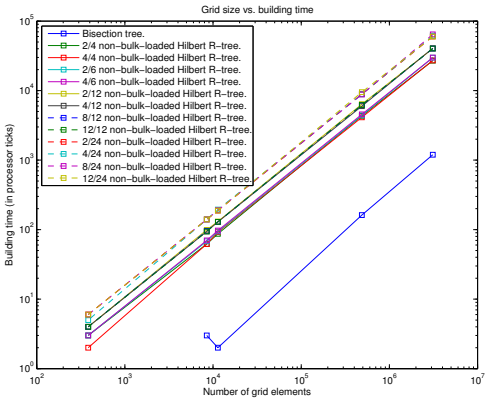


(d) Average line query time

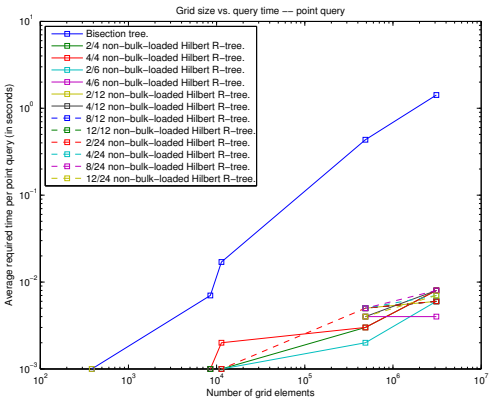


(e) Average box query time

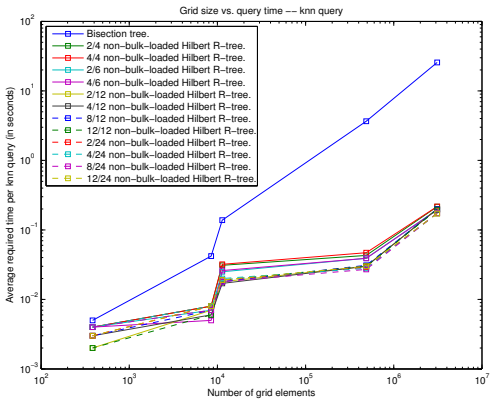
FIGURE 4.1. Some figures illustrating results of the basic R-tree with quadratic split, using different combinations of m and M.



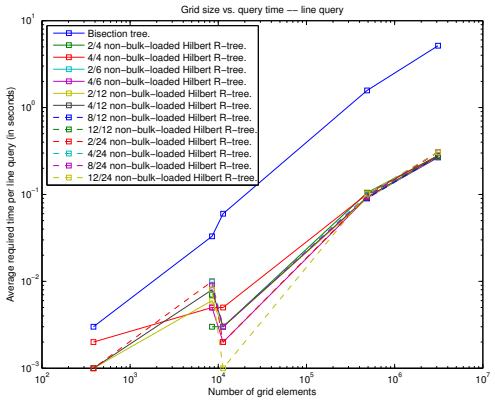
(a) Building time



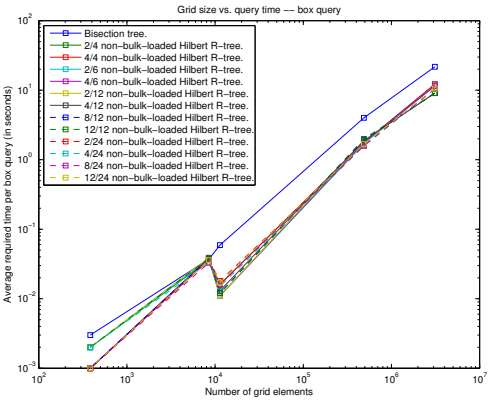
(b) Average point query time



(c) Average k-nn query time

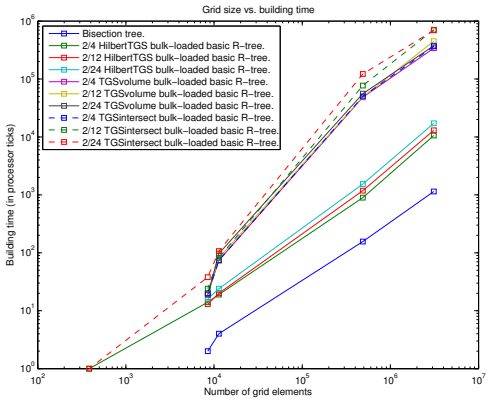


(d) Average line query time

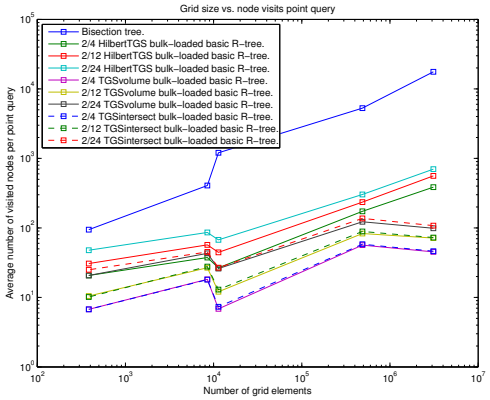


(e) Average box query time

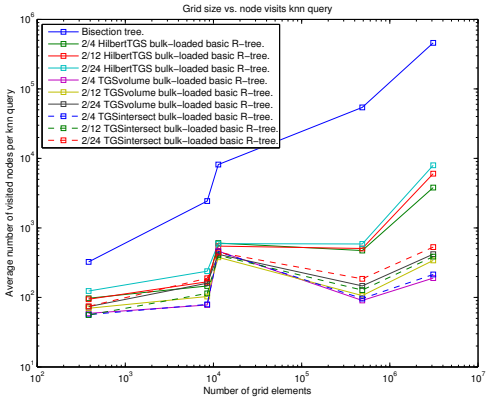
FIGURE 4.2. Some figures illustrating results of the Hilbert R-tree, using different combinations of m and M.



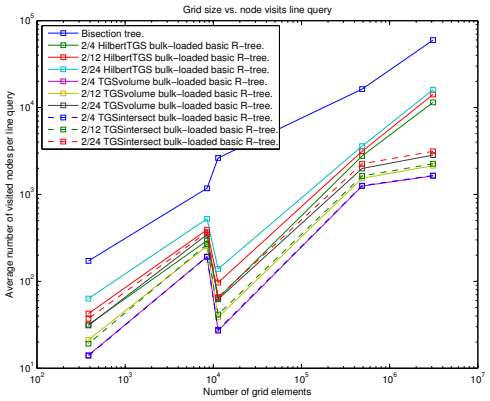
(a) Building time



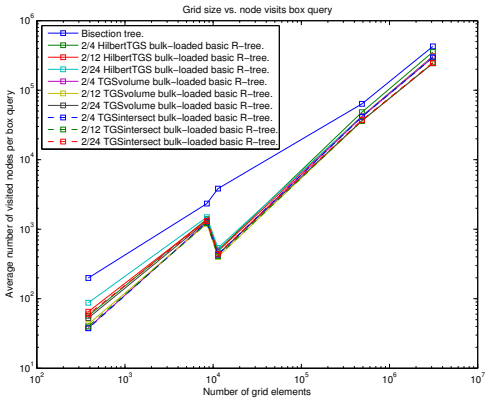
(b) Average number of nodes per point query



(c) Average number of nodes per k-nn query

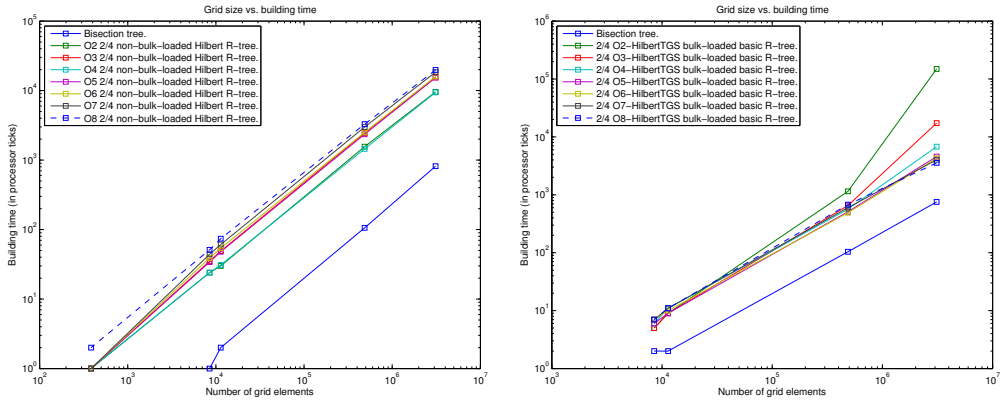


(d) Average number of nodes per line query



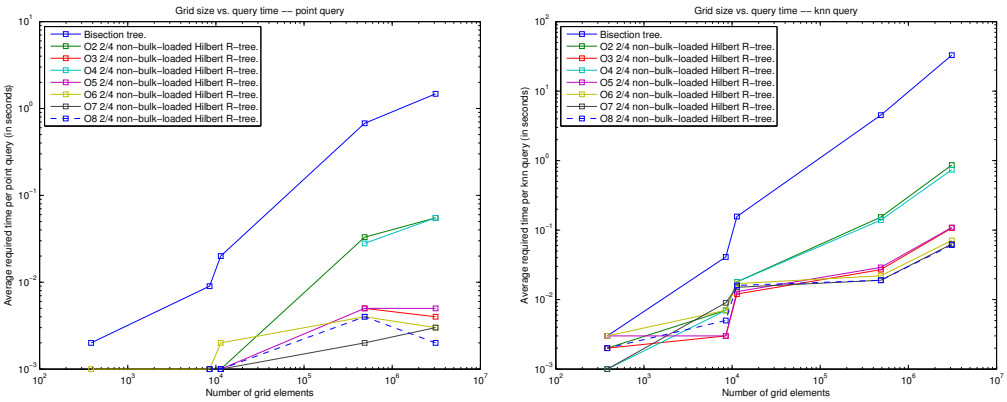
(e) Average number of nodes per box query

FIGURE 4.3. Some figures illustrating results of various bulk-loaded R-trees, using different numbers of maximum children per node.

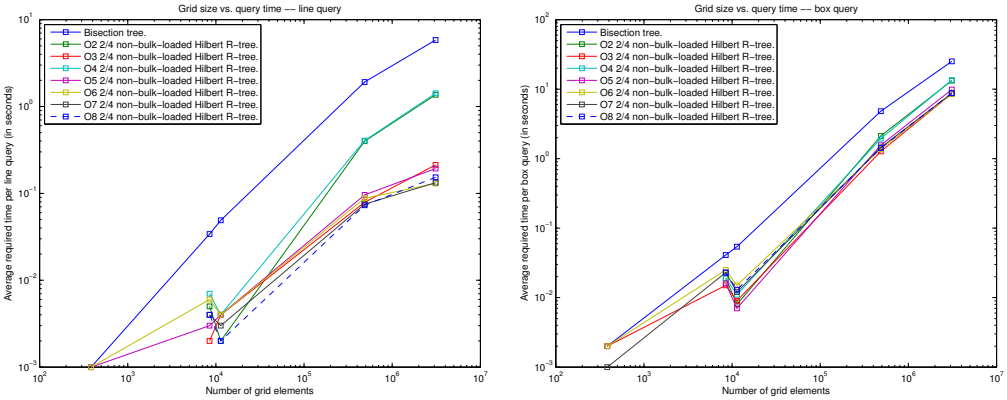


(a) Hilbert R-tree (b) HilbertTGS

FIGURE 4.4. Build times for various Hilbert curve orders.

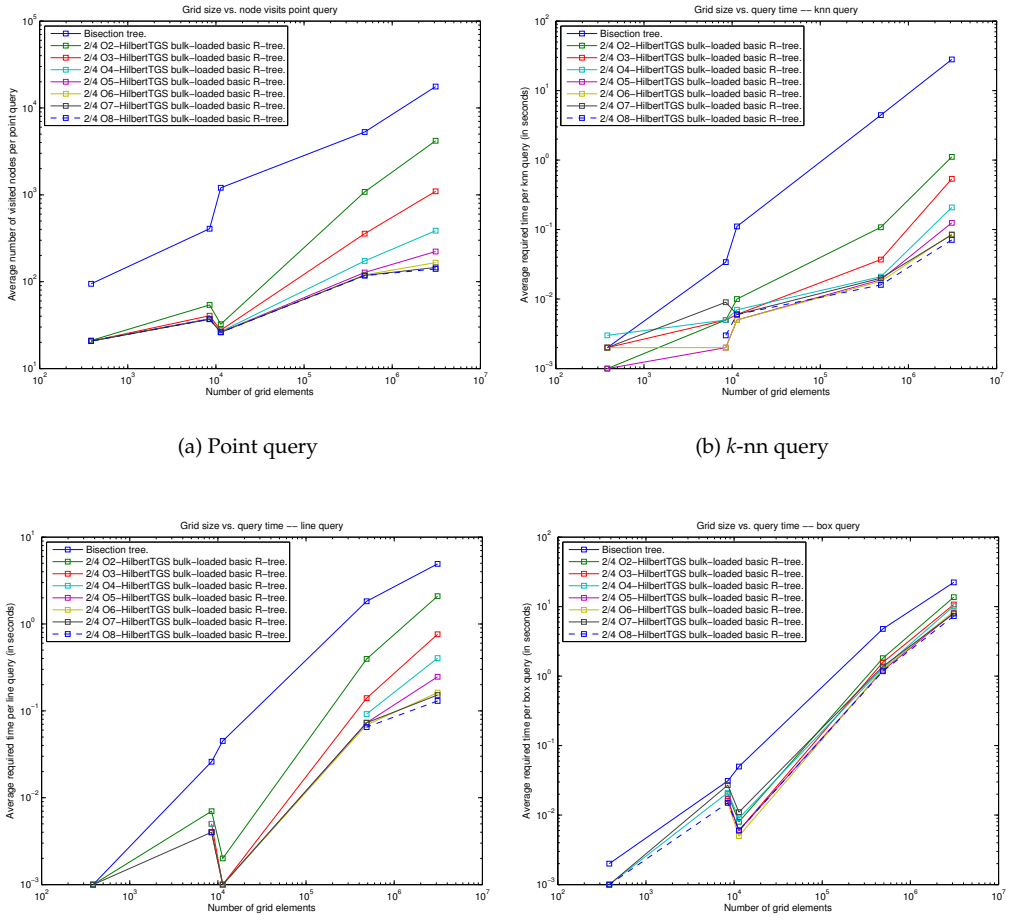


(a) Point query (b) *k*-nn query



(c) Line query (d) Box query

FIGURE 4.5. Average query times or node visits for the Hilbert R-tree with various Hilbert curve orders.



(a) Point query

(b) k-nn query

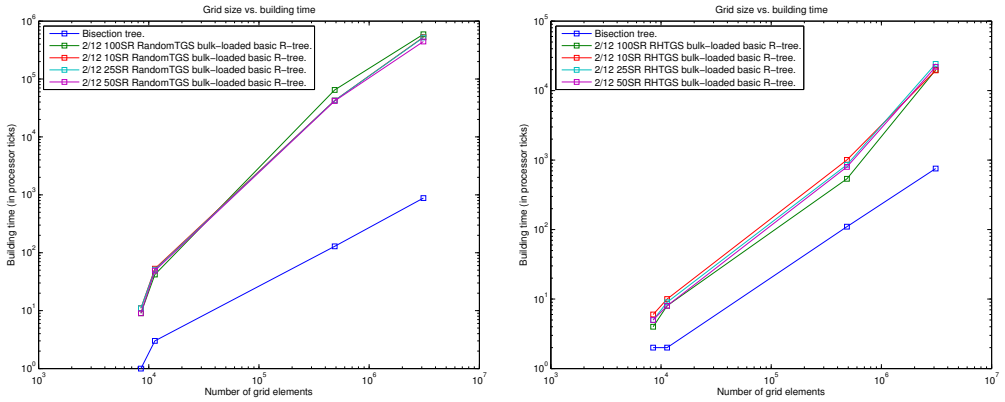
(c) Line query

(d) Box query

FIGURE 4.6. Average query times or node visits for Hilbert TGS with various Hilbert curve orders.

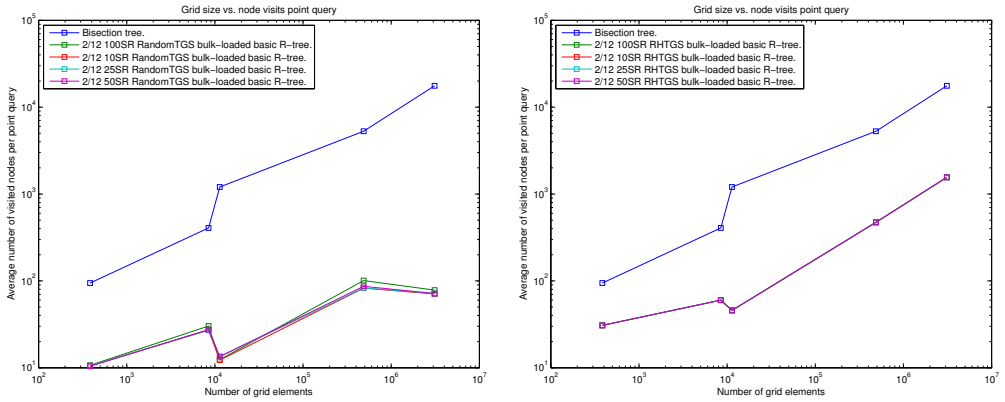
Parameter	Value
Experiment machine	Qire
n	1000
Optimisation level	O3
Hilbert curve order	3
Minimum number of children	2
Maximum number of children	12
TGS cost function	Sum of volumes

Sampling rates of 10, 25, 50 and 100 percent have been investigated for both the TGS method and the HilbertTGS method. The effects on building times are as in figure 4.7. Results of the average node visits per point query are shown in figure 4.8. The results with respect to which SR-variant is best are similar for each other query. Note that in figure 4.8(b) the number of node visits for different SR are equal.



(a) Random TGS (b) Random HilbertTGS

FIGURE 4.7. Building times of bulk-loading mechanisms using a sampling rate heuristic.



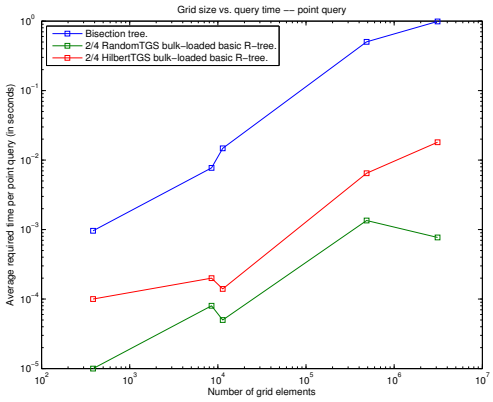
(a) Random TGS (b) Random HilbertTGS

FIGURE 4.8. Average number of visited nodes per query of bulk-loaded R-trees using sampling rate heuristics.

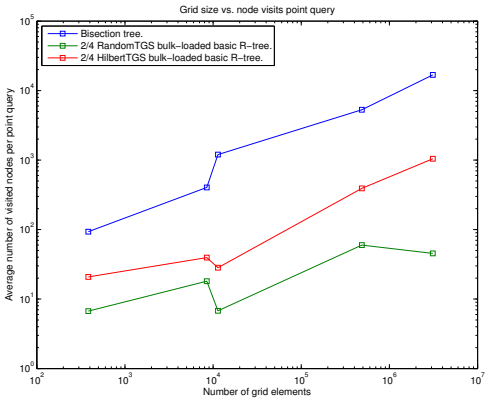
4.5. Average touched nodes versus wall-clock query time

In this section we present both node and wall-clock time graphs for comparison; see figure 4.9 and 4.10. As before, we present the experiment settings:

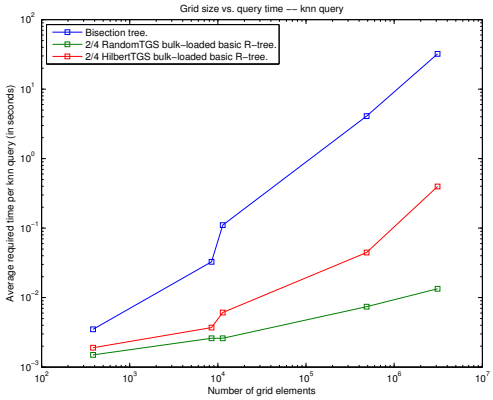
Parameter	Value
Experiment machine	Qire
n	1000 (unless otherwise noted)
Optimisation level	O3
Hilbert curve order	3
Minimum number of children	2
Maximum number of children	4
TGS cost function	Sum of volumes



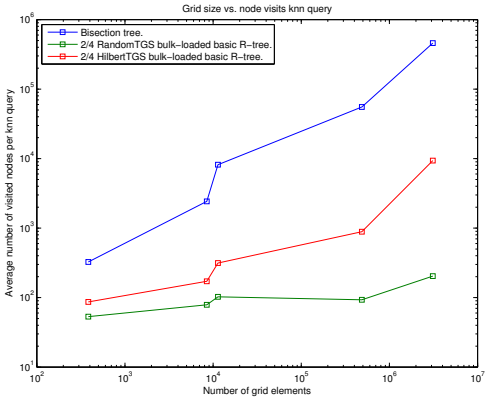
(a) Time graph for point queries, the number of queries done is $n = 10^5$



(b) Node graph for point queries, the number of queries done is $n = 1000$

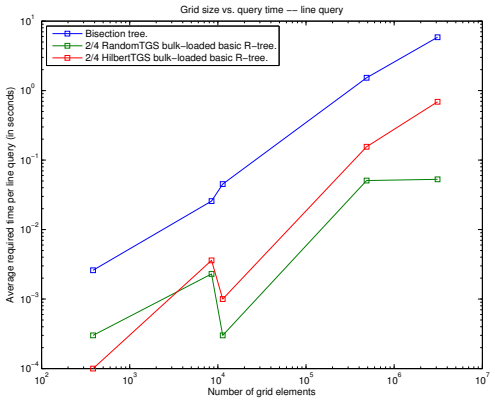


(c) Time graph for *k*nn queries

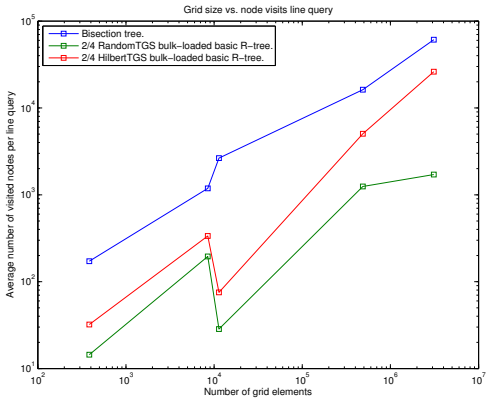


(d) Node graph for *k*nn queries

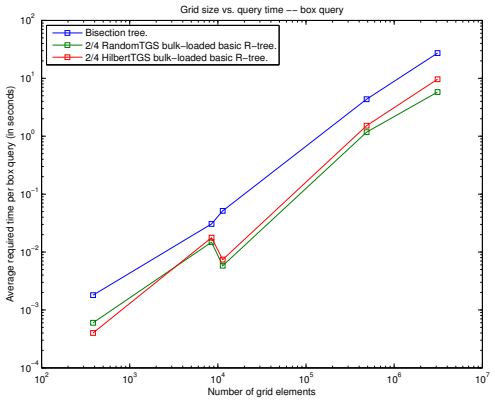
FIGURE 4.9. Comparison between time and node graphs for one-dimensional queries.



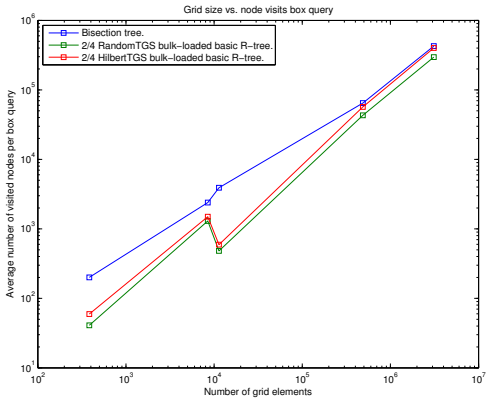
(a) Time graph for line queries



(b) Node graph for line queries



(c) Time graph for box queries



(d) Node graph for box queries

FIGURE 4.10. Comparison between time and node graphs for two-dimensional queries.

4.6. Construction time

On basis of the results presented in the previous subsection, we now proceed with more detailed experiments on the following R-tree variants:

(1) 4-HilbertTGS,
(2) 4-RandomTGS.

Experiments for $M = 12$, including the dynamic Hilbert R-tree variant have also been performed, but the results were not analysed in detail. The experiment results are included in appendix C.2 for the interested reader.

The experiments setting now are as follows:

Parameter	Value
Experiment machine	Qire
n	100000
Optimisation level	O3
Hilbert curve order	3
Minimum number of children	2
Maximum number of children	4
TGS cost function	Sum of volumes

The construction times are as in figure 4.11. In tabular format, the data is as in Table 4.1.

	384	8484	11352	486484	3106719
Bisection tree.	0	1	1	105	752
2/4 HilbertTGS bulk-loaded basic R-tree.	0	5	9	550	16595
2/4 RandomTGS bulk-loaded basic R-tree.	0	9	43	35555	261153

TABLE 4.1. Upper row data denote grid sizes, cell numbers contain the number of processor ticks required for construction.

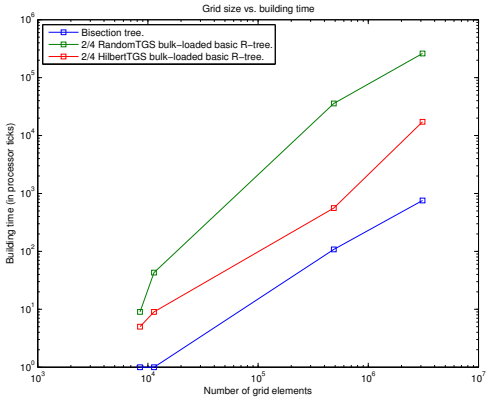


FIGURE 4.11. Construction time of bulk-loading methods and the bisection method

4.7. Query efficiency

We proceed with obtaining data on query efficiencies of the different variants also experimented with in the previous subsection. The experiments settings remain the same. The number of queries (n) done is high enough to obtain gapless wall-clock time graphs, hence we do not show here the average touched nodes graphs.

Parameter	Value
Experiment machine	Qire
n	100000
Optimisation level	O3
Hilbert curve order	3
Minimum number of children	2
Maximum number of children	4
TGS cost function	Sum of volumes

The query times are as in figure 4.12. In tabular format, the data is as in tables 4.2, 4.3, 4.4, and 4.5.

4.8. Memory usage

On memory usage, we have obtained there results as in figure 13(a) and 13(b). Allocated memory denotes the amount of memory the system has reserved for the given process, while the in-use memory denotes the number of bytes actually requested the experiment software. Global experiment settings follow.

	384	8484	11352	486484	3106719
Bisection tree.	0.00096	0.00772	0.01474	0.5031	0.98837
2/4 HilbertTGS bulk-loaded basic R-tree.	0.0001	0.0002	0.00014	0.00647	0.01805
2/4 RandomTGS bulk-loaded basic R-tree.	0.00001	0.00008	0.00005	0.00135	0.00077

TABLE 4.2. Upper row data denote grid sizes, cell numbers contain the number of processor ticks required for point queries.

	384	8484	11352	486484	3106719
Bisection tree.	0.0034	0.03588	0.11113	4.76374	25.5806
2/4 HilbertTGS bulk-loaded basic R-tree.	0.00194	0.00409	0.00605	0.03664	0.40226
2/4 RandomTGS bulk-loaded basic R-tree.	0.00124	0.00209	0.00272	0.00711	0.01278

TABLE 4.3. Upper row data denote grid sizes, cell numbers contain the number of processor ticks required for $k$nn queries.

	384	8484	11352	486484	3106719
Bisection tree.	0.00236	0.02606	0.04475	1.77424	4.30865
2/4 HilbertTGS bulk-loaded basic R-tree.	0.00017	0.00397	0.00073	0.12706	0.68414
2/4 RandomTGS bulk-loaded basic R-tree.	0.00019	0.00243	0.00035	0.04147	0.04673

TABLE 4.4. Upper row data denote grid sizes, cell numbers contain the number of processor ticks required for line queries.

	384	8484	11352	486484	3106719
Bisection tree.	0.00181	0.03019	0.05012	4.75794	21.3555
2/4 HilbertTGS bulk-loaded basic R-tree.	0.00069	0.01716	0.00682	1.27654	10.0069
2/4 RandomTGS bulk-loaded basic R-tree.	0.00054	0.01467	0.00542	0.8876	5.73007

TABLE 4.5. Upper row data denote grid sizes, cell numbers contain the number of processor ticks required for box queries.

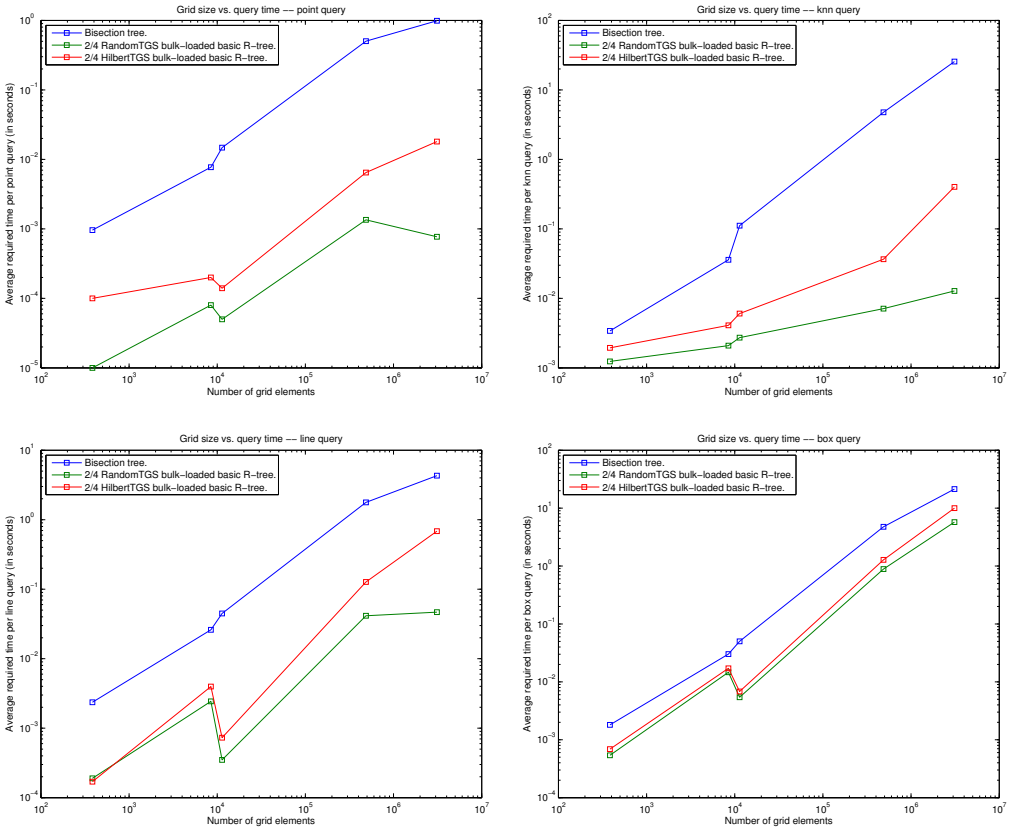
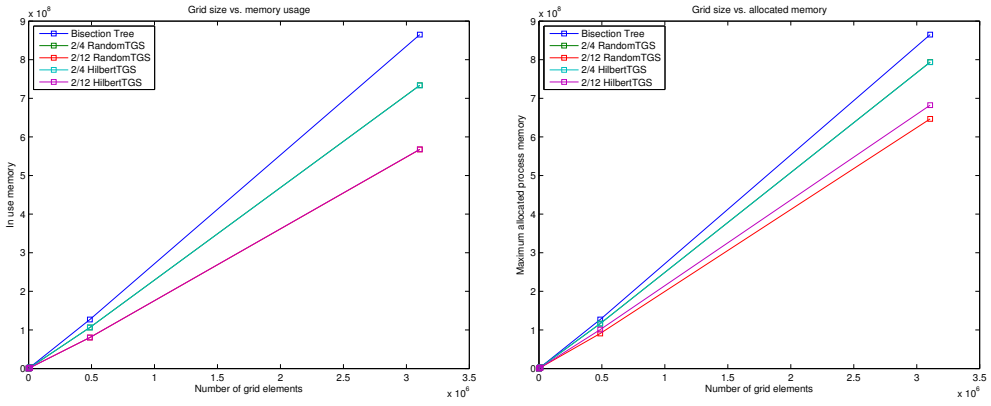


FIGURE 4.12. Query performance in terms of wall-clock query time for various variants.

Parameter	Value
Experiment machine	Qire
n	100000
Optimisation level	O3
Hilbert curve order	3
Minimum number of children	2
Maximum number of children	4 or 12
TGS cost function	Sum of volumes



(a) In-use bytes

(b) Allocated bytes

FIGURE 4.13. Memory usage of various datastructures.

CHAPTER

5

FIVE

Conclusions

In Chapter 1 we introduced a problem regarding data storage and access in the Dynamo software package by Shell. Chapter 2 introduced some data structures which may accommodate the requirements in this context. We analysed their properties theoretically in Chapter 3 and furthermore implemented the datastructures in C++ and went on with experimentation on actual, real-life datasets supplied by Shell; the experiment results were given in Chapter 4.

Now we will reflect back if the experiment results conform to the theoretical analysis, and we will decide if the proposed datastructures indeed are effective enough to be used instead of the bisection method currently employed by Shell. Furthermore, we will also suggest avenues of further research.

5.1. Experiment Analysis

Here we will analyse the experiments done in Chapter 4, starting with those dealing with finding good parameters. This is followed by analysing R-trees with the most efficient parameters we have found. Then, an in-depth comparison to the bisection method will follow.

5.1.1. Number of children. For dynamic R-trees, namely the basic R-tree with quadratic split and the Hilbert R-tree, we generally see that the higher the maximum amount of children or elements per node M, the longer the construction time; see figure 4.1 and 4.2. This is conform to equation 3.6. On the other hand, varying the minimum amount of children or elements per node does not seem to have an impact as large as varying M, which is also conform the aforementioned equation; there, m only occurs in the scaling factor $\log m$.

For static R-trees (Figure 4.3), or rather the bulk-loaded R-trees, we also see that M has a direct scaling influence on the construction times. For all tested values of M on each tested bulk-loading algorithm, we also see that the order of the construction time stays similar; the HilbertTGS construction times are of the same order as that of the bisection tree, while the other bulk-loading algorithms all seem to stay in a higher polylogarithmic order (see section 5.1.4).

This tendency is continued when we consider the query times as well. Less children per node result in faster query times for point, k-nn and line queries. The results for box queries are less decisive; the query times are more or less packed on each other, with only the bisection tree being noticeably slower.

Thus it seems that in our case, overlapping MBRs are not so much of an issue; otherwise a search would recurse into many children quickly building up the query time (See section 3.4.1). Instead, the searches seem to branch of into multiple children only in lower levels, when absolutely necessary.

Keep in mind however that a lower amount of children will result in higher trees and thus in larger memory requirements. Also note that m and M here are chosen without considering any cache effects; this is contrary to most other R-tree applications, where m and M are chosen exactly so that a single node fits into the cache exactly. Hence, instead of looking to find optimal cache effects, we look to find the most effective tree structure instead.

As a result of the experiments done here, we also have seen that the TGSintersect method generally does not results in more efficient trees with respect to query time than when using the TGSvolume method; the construction time however does increase significantly. Hence, from here on out we do not experiment with the intersection-minimising cost function but solely with the standard total volume minimising cost function when using TGS.

5.1.2. Hilbert curve order. From figure 4.4 we can see that interestingly, a higher Hilbert order always results in a longer building time for the Hilbert R-tree while the building time required for HilbertTGS may actually lessen. According to equations 3.23 and 3.24, we have that the insertion time is linearly dependent on the Hilbert curve order. The total construction time is mainly dominated by how many overflows take place during insertion. Given the aforementioned figures, we may conclude that the number of overflows remain similar on large enough datasets, regardless of the Hilbert curve order. Therefore, there is a linear correspondence between the curve order and the construction time.

This is not the case for the HilbertTGS bulkloading method. Having equation 3.19 in mind, we see that there is again a linear correspondence between the number of elements and the construction time (the factor ndl). However, for large enough datasets, this is not reflected in the figure at all; there, a low-order Hilbert curve will cause the HilbertTGS algorithm to execute much slower than when using higher order curves. This is due to the low precision low order curves yield; the generated binary splits are of such bad quality that calculating the MBRs of the splits [1] requires far more time than when using a higher order curve.

On query times (see figure 4.5), we see that on average, the higher the order, the better the query times. An exception to this seems to be the third order Hilbert curve; this particular variant seems to perform on par with the fifth order Hilbert curve in terms of query performance. This is odd, since a third order Hilbert curve can only distinguish between $8^3 = (2^3)^3 = 2^9 = 512$ different cells, while the largest dataset contains more than 3 million blocks. There seems to be no apparent explanation for this phenomenon, altough we suspect it is due to the order in which grid blocks were intially stored in the datasets supplied by Shell; they are globally ordered ascendingly, first through the x-axis, then the y- and finally the z-axis. Apperantly, dividing the grid blocks in 512 subareas higher up the R-tree and using the inferred ordering lower down the tree, leads to a favorable structure[2].

We used a third order Hilbert curve in subsequent experiments because it performed just as well as a fifth order curve, but needs less recursive calls. However, using a Hilbert curve order o for which $8^o \geq n$ (thus, $o = 8$ for the largest dataset), would result in a more optimal Hilbert-based R-tree. The question remains if the added construction time due to the larger number of recursions is worth the gain.

[1] lines 6,7, Algorithm A.5.2

[2] This structure may actually correspond to a hybrid method between a Hilbert R-tree and a so called *kd-tree*.

5.1.3. Sampling rate. Here we discuss the results of the so-called RandomTGS and Random HilbertTGS bulk-loading methods.

Varying the sampling rate to, in the first place, reduce the construction time of bulk-loading algorithms, seem not to have as great an impact as initially expected. For TGS bulk-loading, the running times are cut back by a small factor. In numbers, we have the following building times on the largest dataset available (see also figure 4.7:

Sampling rate	Construction time (processor ticks)
100% (normal TGS)	591620
50%	443523
25%	520720
10%	527641

We see that dropping 1 out of 2 possible splits beforehand is faster than considering all splits, as would be expected. The gain however, is not 50%; it is closer to 25%. More interestingly, lower sampling rates actually increase the needed construction time with respect to the 50% sampling rate. In figure 4.7, we even see that for the Random HilbertTGS bulk-loading method, the construction time of the original TGS method (100%) is actually the fastest one measured.

As a partial explanation for this, profiling the current experimentation software revealed that the bulk of the construction time needed is used for actual node construction and sorting; not for the actual recursion of the bulk-loading method. Cutting back on recursion steps would then not affect the final construction time that much; the only nodes that the algorithm will construct are nodes that will certainly be part of the resulting R-tree, and this number of nodes to be build is always the same, under equal numbers of input elements.

With respect to query efficiency (figure 4.8) we see that the true TGS method is less efficient than the random TGS method for each given sample rate. Apparently, randomisation has some merits over greedily choosing the best split available. There does not seem to be a clear winner among the different sampling rates, but going from the construction times, one could say that random TGS with a 50% sampling rate is preferable over the classic TGS method.

Random HilbertTGS shows a very different picture; adding in randomisation does not seem to change the eventual tree output. This may be due to a implementation technicality; since the TGS algorithm always has to consider at least 1 split, the program always considers the first available split applies randomisation only on the splits to be considered afterwards. If the first split is also the best one, then each 'random' Hilbert TGS method results in the same tree, regardless of the sampling rate.

Hence again going primarily from construction times, we have that the original HilbertTGS method is preferable to the random HilbertTGS variation.

5.1.4. Construction times. In figure 4.11 we see that the graphs belonging to the bisection method and the HilbertTGS method have about the same slope. This is conform theory; the construction time of the bisection method is of order $O(hn) = O(n \log n)$, and that of the HilbertTGS method is of order $O(n \log^2 n)$. In log-log scale, this results in graphs with shapes similar to $\tilde{n} + \log \tilde{n}$ and $\tilde{n} + 2 \log \tilde{n}$ (with $\tilde{t} = \log t$ and $\tilde{n} = \log n$); see the following derivations:

$$\begin{aligned} e^{\tilde{t}} &= e^{\tilde{n}} \log e^{\tilde{n}} = n e^{\tilde{n}}, \quad \text{and as such} \\ \tilde{t} &= \log n e^{\tilde{n}} = \log \tilde{n} + \tilde{n}, \end{aligned}$$

$$\begin{aligned} e^{\tilde{t}} &= e^{\tilde{n}} \log^2 e^{\tilde{n}} = \tilde{n}^2 e^{\tilde{n}}, \quad \text{and as such} \\ \tilde{t} &= 2 \log \tilde{n} + \tilde{n}. \end{aligned}$$

Hence for large n, the graph will be dominated by $\tilde{n}$. Indeed, the graphs for the bisection method and HilbertTGS show nearly linear graphs. The graph belonging to the RandomTGS method looks more curved, however. For larger number of grid elements, the slope of that graph still seems to converge to that of HilbertTGS and the bisection method. This can also be explained from theory; the construction time of RandomTGS is a few factors larger than that of the HilbertTGS method due to the extra quicksort calls; for each tree level, the entire input set of n elements is sorted $|S|$ times. This amplifies the logarithmic behaviour at the start of the graph, resulting in the curve seen in the figure.

Finally, notice from the tables that the time required for constructing an R-tree using RandomTGS is about 350 times slower for the largest dataset, while the HilbertTGS method is 22 times slower; it will take very good query times and a lot of queries for RandomTGS to become favoured above the HilbertTGS method.

5.1.5. Average number of touched nodes and average time per query. In the previous chapter, we claimed that measuring the numbers of nodes touched during a query is similar to measuring the wall-clock running time. The rationale is that when the query algorithm visits a node it always takes about roughly the same time to decide in which of the children to recurse or to decide which elements should be returned; the only variable not considered in this case is the number of children per node, which may vary. The effect of ignoring this should be small when $M - m$ is small, or non-existent when the tree is packed, for example in case of bulk-loading algorithms.

The only difference between time and node measurements would thus only arise from difference in either code-level optimisation, and differences of how the system architecture is exploited (cache effects for example), possibly unintendedly. This is the primary reason why node-level measurement was used; it enables us to measure the efficiency of spatial datastructures without having to worry about implementation details too much.

Figures 4.9 and 4.10 show a comparison between the average number of nodes touched during a single query and the average wall-clock running time of a single query. One can see the graphs are of similar shape, and as such, the number of touched nodes indeed seems to be a valid system and optimisation independent measurement of query efficiency.

In contrast of earlier implementations of the experimentation software, the current version performs quite well in terms of optimisation, with respect to the bisection method. There are still gains to be expected from implementing advanced memory management; optimising for cache and forcing contiguous memory use, for example. Due to this current high enough level of optimisation, most of the following experiments only display the wall-clock time graphs.

5.1.6. Query times. Figure 4.12 is pretty consistent about the relative query performance of each datastructure; RandomTGS is fastest, followed by HilbertTGS and with the bisection method as the slowest. This is untrue only for line queries on the smallest dataset; there, HilbertTGS performs slightly better. The slopes of the graphs for the point and $k$nn queries are noticeably steeper for the bisection method than for the TGS methods. Both TGS methods seem to perform similarly with respect to query time complexity, on all query types.

On line queries, the advantage of the R-trees over the bisection tree begins to fade; extrapolating the few data points would indicate the bisection tree would actually achieve better query times then HilbertTGS after datasets with 10^8 grid elements or more; this could however be solved by using a higher order Hilbert curve, which increases the HilbertTGS performance, as seen earlier. The asymptotic order of RandomTGS and the bisection tree still remain similar.

The same holds for box queries, although the performance of all three trees are nearly similar now. Note however that these plots are still all in log-log scale, hence even these seemingly small

difference between the graphs indicate a gain of a factor 2 to 4 for HilbertTGS and RandomTGS, respectively.

Concluding from these experiments, we can see that for queries which return few close-by elements, bulk-loaded R-trees can outperform the bisection method easily and consistently. For queries returning a lot of grid elements the gain of these R-trees over the bisection method diminishes but still does not perform worse than the bisection tree on any dataset used here.

The theory predicted a running time of fractional power ($O(n^t)$ with $0 \leq t \leq 1$). In figure 5.1 we manually tried to fit the data to a function $f(n) = n^t \cdot a$. Below the results are summarised. The p value corresponds to the probability the search algorithm would recurse in a given node; see equation 3.29; the lower, the better the "quality" of the tree with respect to the particular query in question.

Figure 4.12 shows two anomalies. The first one regards the difference in query times for each type of query from the second to the third dataset (8484 to 11352 non-void blocks). For the bisection tree, we see a steep increase, while for the R-trees we see a decrease, except for the *k*nn query. There the increase in query time is however still much less than the increase for the bisection tree. The difference between the two datasets is that the second one contains void blocks, whereas the third one does not; this would indicate R-trees perform better on "full" datasets than the bisection tree.

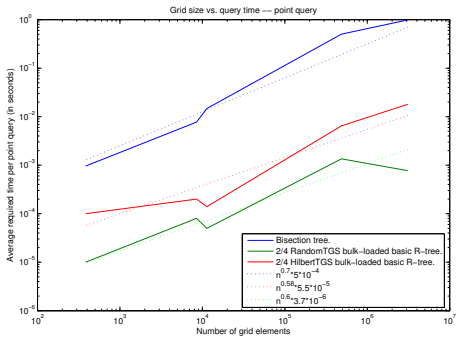
The second anomaly regards point queries. There, we observe a *decrease* in query time for RandomTGS when comparing the query times for the forth and the fifth dataset (486484 and 3106719 non-void blocks). It is unclear what causes this increase in efficiency; it seems too large to simply put it on the random character of the RandomTGS bulk-loading method.

Datastructure	Query type	t	a	p
Bisection	point	0.7	$5 \cdot 10^{-4}$	0.719
RandomTGS	point	0.58	$5.5 \cdot 10^{-5}$	0.559
HilbertTGS	point	0.6	$3.7 \cdot 10^{-6}$	0.574
Bisection	line	0.8	$3.5 \cdot 10^{-4}$	0.803
RandomTGS	line	0.8	$6 \cdot 10^{-5}$	0.758
HilbertTGS	line	0.55	$1.7 \cdot 10^{-5}$	0.536
Bisection	knn	0.93	$6 \cdot 10^{-5}$	0.926
RandomTGS	knn	0.49	$8 \cdot 10^{-3}$	0.493
HilbertTGS	knn	0.23	$4 \cdot 10^{-3}$	0.344
Bisection	box	0.93	$7 \cdot 10^{-4}$	0.926
RandomTGS	box	0.99	$4 \cdot 10^{-5}$	0.986
HilbertTGS	box	0.93	$5.5 \cdot 10^{-5}$	0.908

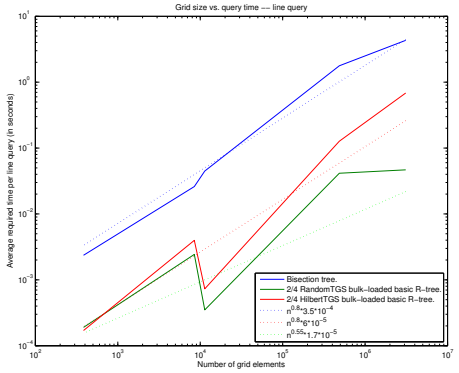
5.1.7. Turnover points. Suppose we have an application in which we have to use spatial data storage structures. Let q be the number of queries we have to perform, and for simplicity, let us assume the queries to be done are of a single type. Then, the best data structure for the job in terms of wall-clock execution time regarding data access, is the one for which the following is smallest:

$$t_{\text{data\_access}}(q) = t_{\text{construction\_time}} + q \cdot t_{\text{average\_single\_query\_time}} \tag{5.1}$$

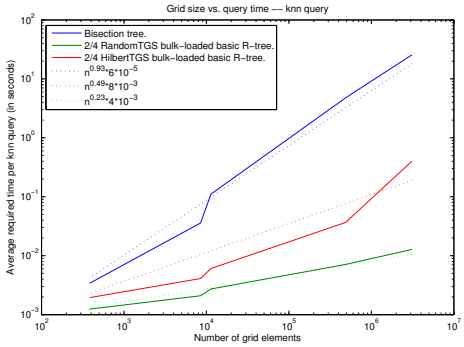
Since we know the datastructure currently at use is the Bisection tree, we can use the above formula to obtain a *turnover point* of an arbitrary other spatial datastructure S as follows. Let t_c^S denote the construction time of datastructure S and t_q^S the average query time of that datastructure. Let t_c^B and t_q^S be analogously defined for the bisection tree. Then, the turnover point is calculated as follows:



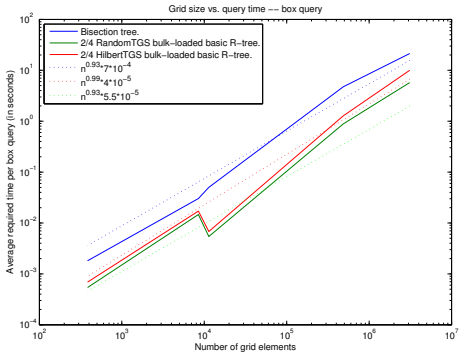
(a) Point query



(b) Line query



(c) knn query



(d) Box query

FIGURE 5.1. Results of fitting query time experiment data to functions of fractional power.

$$t = \frac{t_c^S - t_c^B}{t_q^B - t_q^S}. \tag{5.2}$$

t denotes here at which number of queries q we have that $t^B_{\text{data\_access}}(q) = t^S_{\text{data\_access}}(q)$, so that for each $q > t$ we have that $t^B_{\text{data\_access}}(q) > t^S_{\text{data\_access}}(q)$ iff $t_q^B > t_q^S$.

Note that typically, for a small number of queries, the bisection tree is better than most R-tree variants because of the low construction time of the bisection method. However, since for the R-trees treated in this section we have that the average single query time is much lower, we also have relatively low turnover points after which the R-tree is preferred above the bisection tree. These turnover points can be calculated from the experiments in section 4.6 and 4.7. These calculated turnover points are shown in the figures 5.2, 5.3, 5.4, 5.5.

	384	8484	11352	486484	3106719
2/4 RandomTGS bulk-loaded basic R-tree.	Always worse	1178.01	2859.09	70644.7	264320
2/4 HilbertTGS bulk-loaded basic R-tree.	Always worse	664.894	479.452	896.039	16326.6

FIGURE 5.2. Different combinations of datastructures and grid sizes and their turnover points on random point queries, wrt the Bisection tree.

	384	8484	11352	486484	3106719
2/4 RandomTGS bulk-loaded basic R-tree.	Always worse	236.756	387.418	7454.23	10212.9
2/4 HilbertTGS bulk-loaded basic R-tree.	Always worse	94.3693	85.649	94.1381	628.794

FIGURE 5.3. Different combinations of datastructures and grid sizes and their turnover points on random knn queries, wrt the Bisection tree.

	384	8484	11352	486484	3106719
2/4 RandomTGS bulk-loaded basic R-tree.	Always worse	380.872	968.468	20504.7	61099.5
2/4 HilbertTGS bulk-loaded basic R-tree.	Always worse	181.077	181.736	270.159	4394.8

FIGURE 5.4. Different combinations of datastructures and grid sizes and their turnover points on random line queries, wrt the Bisection tree.

	384	8484	11352	486484	3106719
2/4 RandomTGS bulk-loaded basic R-tree.	Always worse	515.464	939.597	9220.38	16687.4
2/4 HilbertTGS bulk-loaded basic R-tree.	Always worse	306.984	184.758	129.546	1455.77

FIGURE 5.5. Different combinations of datastructures and grid sizes and their turnover points on random box queries, wrt the Bisection tree.

5.2. Memory usage

From figure 4.13 we see that the results are conform theory; the higher the amount of children, the less high the resulting tree becomes, the less the number of internal nodes, and thus ultimately, the less memory it consumes. Following equation 3.48, we thus have a memory usage of order

$$O(M^{\lceil \log_M n \rceil - 1} - 1) = O(n).$$

In figure 5.13(b) we see a difference between RandomTGS bulk-loading and HilbertTGS; this is because the latter needs extra memory resources during construction to calculate Hilbert coordinates. The effects of this are less apparent as M becomes larger. Basic linear fitting has been performed on figure 5.13(b); the results are as follows:

Fit to: $f(x) = ax$

Datastructure	a
Bisection	278.69
4-RandomTGS	236.41
4-HilbertTGS	236.23
12-RandomTGS	182.9
12-HilbertTGS	183.03

Here we see confirmed that the amount of memory used by both bulk-loading strategies is similar when M is equal. We also see big gains in terms of memory efficiency when increasing M; the savings from $M = 4$ to $M = 12$ is about the same as switching from bisection to RandomTGS with $M = 4$.

As mentioned earlier, the current experimentation code does not exploit cache effects, nor does any memory management whatsoever; memory is allocated for each new tree node constructed and thus, memory used by the application is non-contiguous. Also, explicitly creating a new tree node is slower than allocating a bunch of tree nodes in a single run; this fact may be exploited to further speed up construction times. In particular, implementing a better memory management scheme would improve the RandomTGS bulk-loading method considerably (see also Section 5.1.3).

5.3. Recommended R-Tree variant

Given the preceding analysis, we would conclude the HilbertTGS bulk-loading method seems to be the method of choice for use with Shell datasets; the datastructure construction times are good, as are the wall-clock query times. The latter are of course not better then those obtained when using the RandomTGS bulk-loading method, however, the construction time for that particular variant is a whole lot higher; using formula 5.2 for comparing the RandomTGS and HilbertTGS methods for point queries on the largest dataset, we have that we need to do more than approximately $14 \cdot 10^6$ queries. Expected is that this amount of queries is not realistic for a typical Shell testrun.

While the third order Hilbert curve was used in the final experiments, it might be worthwhile to use a higher order curve, higher than 5, since one can see a small increase in query times for the largest dataset in the figure 4.12; if the extra building time is found to be worth the gain. Also, while the maximum number of children per node set at $M = 4$ delivers the best performance, one might opt to increase this amount to lower memory usage.

5.4. Future Work

Due to time limitations, many other ideas we have wanted to experiment with have not been researched in detail. These ideas are the following.

5.4.1. Research other TGS cost functions. Standardly, the TGS bulk-loading method uses the sum of the volumes of two MBRs as the cost function f_1. Some initial experiments have been performed by instead taking the volume of the intersection of the MBRs as cost function f_2, but this did not yield any improvements.

Different cost functions may be possible; for instance, a linear combination of f_1 and f_2 seems interesting to research, e.g.:

$$f = \alpha f_1 + \beta f_2 \tag{5.3}$$

with $-1 \leq \alpha, \beta \leq 1$. A good combination α, β may possibly yield better results than when only using f_1 as const function.

5.4.2. Improved Random TGS. Instead of using an uniform[3] distribution for deciding when a binary split should be considered or not, other distributions may result in trees which achieve higher query efficiency; for instance, one may intuitively think that splitting a set in two equal parts is probably better than splitting off only the first few elements, giving rise to an uniform

[3] i.e., each split has the same chance of being selected

distribution. Note however that which distribution is best will probably be dependent on the spatial distribution of the data objects to be stored.

5.4.3. Faster and dimension-independent Hilbert coordinate transforms. The Hilbert transformation algorithm outlined in this thesis and implemented in the experimentation software, only works for $d = 2$ and $d = 3$; it is worthwhile to implement an algorithm which works for any $d \geq 2$. This would also open up the possibility to research other Hilbert orderings; the current one transforms the centre coordinate of MBRs, but no experimentation has been done for any of the other possibilities outlined in section 2.5.3, because of the limitation of d.

While on basis of [KF93] we expect that using the centre coordinate was the best option in general, it may be true that one of the other options is a better choice for the type of grids we store in our case.

5.4.4. Cache and main memory optimisation. It is most certainly worthwhile to implement a better memory management scheme, as described in Section 5.2. In the case where the R-tree is fully packed and we apply some bulk-loading scheme (such as TGS), we can calculate the total number of required tree nodes beforehand. Hence it is possible to allocate one large array containing exactly the number required nodes in one go, so that the bulk-loading algorithm no longer actively has to allocate and construct new tree nodes. This will also increase the effect of the sampling rates in the RandomTGS method, since constructing new nodes no longer will be a bottleneck.

5.4.5. Differently shaped bounding boxes. We have consistently used the standard hyper-rectangular bounding boxes, but a differently shaped bounding box might have been more effective. Shapes to be considered could be, for example, a sphere or a cylinder. However, since the objects we wish to manage here are beam-like, we do not expect large gains, if any, by using a different shape.

5.4.6. Priority R-tree. One R-tree variant we did not experiment with but would have liked to, is the Priority R-tree introduced in [AdBHY04]. In its paper the claimed to outperform the Hilbert R-tree in terms of query time, and still hold similar construction times. Interesting for us would be to see how this variant would perform when measured against the HilbertTGS variant.

5.4.7. Tree-level dependent number of children. In our case, we did not use the R-tree for storing information on a physical system such as a hard-drive, and also did not do any cache optimisations; as such the number of children $n, m \leq n \leq M$ per node was free for us to choose. However, why would we hold m and M constant for each tree level? Depending on if the input objects were clustered, it might be worthwhile to use more children per node higher up the tree so to easily differentiate between those clusters, and fewer children per node when closer to the leaf nodes to speed up searching. The use of multiple children higher up the node would at least already result in less memory usage.

5.4.8. Hybrid methods. Another idea we had but could not investigate, were hybrid methods. These methods consist of using a datastructure for *groups* of objects; if S is the input set, we first build sets $S_1, \ldots, S_m$ each containing $|S|/m$ elements and use a datastructure to store these sets $S_1, \ldots, S_m$. Then, for each of those sets we use a *different* datastructure to store the actual objects.

Hence we could, for example use a TGS-based method to store the higher order sets S_i, and a bisection-based datastructure for the actual objects. TGS construction time may be quite fast if m is small enough, while the bisection method on $|S|$ elements still also has the fastest construction

time. The advantage is that query times may be improved a lot since TGS enables fast high-level searching while the slower bisection method is used for querying on smaller subsets of size $|S|/m$, which can thus still achieve acceptable query times.

Of course, different R-tree variations could be used as well, and more than two ”layers” of datastructures can be introduced; this is expected to be very interesting to research.

Bibliography

[AdBHY04] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. Technical report, 2004. [Online; accessed 26-July-2007. http://www.cs.duke.edu/~yike/prtree/].

[AS07] Houman Alborzi and Hanan Samet. Execution time analysis of a top-down r-tree construction algorithm. *Information Processing Letters*, 101(1):6–12, 2007.

[BG01] John J. Bartholdi III and Paul Goldsman. Vertex-labeling algorithms for the Hilbert spacefilling curve. *Software Practice and Experience*, 31(5):395–408, 2001.

[BO] Timothy Barth and Mario Ohlberger. Finite volume methods: foundation and analysis. *Encyclopedia of Computational Mechanics*.

[CLRS03] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2003.

[CMed] Gary W. Cox and Richard D. McKelvey. A ham sandwich theorem for general measures. Working Papers 337, California Institute of Technology, Division of the Humanities and Social Sciences, undated. available at http://ideas.repec.org/p/clt/sswopa/337.html.

[DWT01] Alan Dennis, Barbera H. Wixom, and David Tegarden. *Systems analysis and design: an object-oriented approach with UML*. Wiley, 2nd; international edition, 2001.

[Eck00a] B. Eckel. *Thinking in C++*, volume 1. 2nd edition, 2000. [Online; accessed July 2007. http://www.ibiblio.org/pub/docs/books/eckel].

[Eck00b] B. Eckel. *Thinking in C++*, volume 2. 2nd edition, 2000. [Online; accessed July 2007. http://www.ibiblio.org/pub/docs/books/eckel].

[GRLL98a] Yván J. García R., Mario A. López, and Scott T. Leutenegger. A greedy algorithm for bulk loading r-trees. In *Proceedings of the 6th ACM international symposium on Advances in geographic information systems*, pages 163–164. ACM Press, 1998.

[GRLL98b] Yván J. García R., Mario A. López, and Scott T. Leutenegger. On optimal node splitting for r-trees. In *Proceedings of the 24th VLDB Conference*, 1998.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[KF93] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *CIKM '93: Proceedings of the second international conference on Information and knowledge management*, pages 490–499, New York, NY, USA, 1993. ACM Press.

[KF94] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th VLDB Conference*, pages 500–509, 1994.

[KSS90] Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference, 1990*, pages 322–331. ACM Press, 1990.

[LEL97] Scott T. Leutenegger, Jeffrey M. Edgington, and Mario A. Lopez. STR: A simple and efficient algorithm for R-tree packing. Technical Report TR-97-14, 1997.

[LMS94] Chi-Yuan Lo, Jirí Matousek, and William L. Steiger. Algorithms for ham-sandwich cuts. *Discrete & Computational Geometry*, 11:433–452, 1994.

[MAK02] Mohamed F. Mokbel, Walid G. Aref, and Ibrahim Kamel. Performance of multi-dimensional space-filling curves. In *GIS '02: Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 149–154, New York, NY, USA, 2002. ACM Press.

[MNPT06] Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*. Springer-Verlag, 2006.

[RK85] N. Roussopoulos and Y. Kotidis. Direct spatial search on pictorial databases using packed r-trees. In *SIGMOD Conference on Management of Data*, pages 17–31. ACM Press, 1985.

[RM03] Philip Romanik and Amy Muntz. *Applied C++: Practical Techniques for Building Better Software*. Pearson Education, 2003.

[Ros04] Eric Rosenberg. The expected length of a random line segment in a rectangle. *Operations Research Letters*, 32(2):99–102, 2004.

[SRF87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In *The VLDB Journal*, pages 507–518, 1987.

[ST42] A.H. Stone and J.W. Tukey. Generalized "sandwich" theorems. *Duke Mathematical Journal*, 9(2):356–359, 1942.

[Wik07] Wikipedia. Big o notation — wikipedia, the free encyclopedia, 2007. [Online; accessed 4-July-2007. http://en.wikipedia.org/w/index.php?title=Big_O_notation&oldid=142242017].

List of Algorithms

APPENDIX

A

ONE

Pseudocode

A.1. Search algorithms

Algorithm A.1.1 Point-Query Search Algorithm on R-trees

Finds a selection of candidate grid elements which may contain a given point x. The returned selection contains all elements containing a given point x, but may also contain some elements which do not. The algorithm is called on an initial tree node v, which usually initially the root node.
Note that this algorithm is a specialisation of the Generic Search Algorithm (Algorithm 2.1) we used to analyse up until now. PointSearch(v, x)

```
1:  Let S be an initially empty set of grid elements.
2:  if v is a leaf node then
3:      for all grid element e stored at v do
4:          if x ∈ MBR(e) then
5:              Add e to S
6:          end if
7:      end for
8:  else
9:      for all children v_i of v do
10:         if x ∈ MBR(v_i) then
11:             S = S ∪ PointSearch(v_i, x)
12:         end if
13:     end for
14: end if
15: return S
```

Algorithm A.1.2 Box-Containment Query Algorithm on R-trees

Finds a selection of candidate grid elements which may be contained in a given bounding box (or rectangle) BB. The algorithm is called on an initial tree node v, which usually initially the root node.

BoxSearch(v, BB)

```
1: Let S be an initially empty set of grid elements.
2: if v is a leaf node then
3:    for all grid element e stored at v do
4:       if BB ∩ MBR(e) ≠ ∅ then
5:          Add e to S
6:       end if
7:    end for
8: else
9:    for all children v_i of v do
10:      if BB ∩ MBR(v_i) then
11:         S = S ∩ BoxSearch(v_i, BB)
12:      end if
13:   end for
14: end if
15: return S
```

Algorithm A.1.3 Line-Query Search Algorithm on R-trees

Finds a selection of candidate grid elements which may intersect with a given line l. l is defined by its starting point s and its ending point e, both in $\mathbb{R}^3$. The algorithm is called on an initial tree node v, which usually initially the root node.

LineSearch(v, s, e):

```
1: Let S be an initially empty set of grid elements.
2: if v is a leaf node then
3:    for all grid element o_i stored at v do
4:       if LineIntersect(s, e, MBR(o_i)) (See Algorithm A.1.4) then
5:          Add o_i to S
6:       end if
7:    end for
8: else
9:    for all children v_i of v do
10:      if LineIntersect(s, e, MBR(v_i)) then
11:         S = S ∪ LineSearch(v_i, s, e)
12:      end if
13:   end for
14: end if
15: return S
```

Algorithm A.1.4 Subalgorithm used for the Line-Query search on R-trees

The following sub-algorithm checks if a line l given by two points s and e intersects with a given bounding box BB.

LineIntersect(s, e, BB):

if $s \in BB$ or $e \in BB$ **then**
  return true
end if
for $i = 1$ to d **do**
  Let b_i be the lower ith dimensional bound of BB
  Let B_i be the upper ith dimensional bound of BB
  if $[s_i, e_i] \cap [b_i, B_i] == \emptyset$ **then**
    return false
  end if
end for
return InfiniteLineIntersect(s, e, BB) (See Algorithm A.1.5)

Algorithm A.1.5 Subalgorithm used for the Line-Query search on R-trees

This sub-algorithm utilises Theorem 2.17 to complete intersection detection.

InfitineLineIntersect(s, e, BB):

Set $\mathbf{d} = \mathbf{s} - \mathbf{e}$
for $i = 1$ to d **do**
  if $\mathbf{d}_i \neq 0$ **then**
    Let b_i be the lower ith dimensional bound of BB
    Let B_i be the upper ith dimensional bound of BB
    Let $\mathbf{y}^1 = \mathbf{s} + \frac{b_i - \mathbf{s}_i}{\mathbf{d}_i}\mathbf{d}$
    Let $\mathbf{y}^2 = \mathbf{s} + \frac{B_i - \mathbf{s}_i}{\mathbf{d}_i}\mathbf{d}$
    if $\mathbf{y}^1 \in BB$ **then**
      return true
    end if
    if $\mathbf{y}_2 \in BB$ **then**
      return true
    end if
  else
    if $s_i \notin [b_i, B_i]$ **then**
      return false
    end if
  end if
end for
return false

Algorithm A.1.6 Hyperplane-Query Search Algorithm on R-trees

Finds a selection of candidate grid elements which may intersect with a given hyperplane. The algorithm is called on an initial tree node v, usually initially the root node.

HyperplaneSearch$(v, \mathbf{n}, b)$:

Let S be an initially empty set of grid elements.
if v is a leaf node **then**
  for all grid elements e stored at v **do**
    if IntersectHyperPlane$(\mathbf{n}, b, \text{MBR}(e))$ **then**
      Add e to S
    end if
  end for
else
  for all children v_i of v **do**
    if IntersectHyperPlane$(\mathbf{n}, b, \text{MBR}(v_i))$ **then**
      $S = S \cup \text{HyperplaneSearch}(v_i, x)$
    end if
  end for
end if
return S

Algorithm A.1.7 Subalgorithm for Hyperplane-MBR intersection detection

This algorithm checks for a hyperplane H as in equation 2.11 if it intersects with a given MBR.

IntersectHyperPlane$(\mathbf{n}, b, MBR)$

for each edge e_j of the MBR (See equation 2.12) **do** {Note that there exist multiple edges e_j for any j when $d \geq 2$}
  $x = \frac{b - n \cdot v}{n_j} + v_j$
  if $x \in [b_j, B_j]$ **then**
    return true
  end if
end for
return false

Algorithm A.1.8 Algorithm for k-nearest-neighbourhood query

Performs a k-nearest neighbourhood query on the R-tree with root r. It will return the k objects with their MBRs closest to a point $p \in \mathbb{R}^d$, with d the dimensionality of the (objects stored in the) spatial tree. This algorithm invokes the subalgorithms A.1.9 and A.1.10, and indirectly also Algorithm A.1.11.

knnSearch(r, p, k):

Let $BB = \Pi_{i=1}^{d}[p_i, p_i]$.
Let $v = \text{ChooseLeaf}(r, BB)$.
Let $S = \emptyset$
$(S, v) = \text{knnFill}(S, v, k)$
$S = \text{knnExtend}(S, v, k)$
return S

Algorithm A.1.9 Subalgorithm for the knn query.

Performs the first phase of the $k$nn-search; i.e., it fills up the set S using heuristics until it has k elements.

knnFill(S, v, p, k):

1: Set $T = \{\text{MBR}(w) | w \in v_O\}$.
2: **if** $|T| - |S| \geq k$ **then**
3: Calculate the distance between each element in T and p using Algorithm A.1.11, and sort T in ascending order using these distances.
4: Set $t = k - |S|$.
5: Drop the last $|T| - t$ entries of T
6: **end if**
7: Set $S = S \cup T$
8: **if** $|S| < k$ **then**
9: Find sibling leaf nodes v_i with their MBR closest to p and add objects from those nodes to S until $|S| = k$.
10: Let v denote the last sibling node v_i, where the last element in $|S|$ was added from.
11: **end if**
12: **return** (S, v)

Algorithm A.1.10 Subalgorithm for the knn query.

Performs the second phase of the $k$nn-search; i.e., it updates S until S surely consists of the closest objects (with respect to p).

knnExtend(S, v, p, k):

Let m be the maximum of the distance of each object in S compared to p (using Algorithm A.1.11).
if v is an internal node **then**
  Let W be the set of unvisited child nodes in v_C.
  Sort W according to the distance of their MBR to p.
  Exclude from W all nodes with their MBR distance to p larger than m.
  while $|S| < k$ **do**
    Let w be the first element of W and let $W = W \backslash w$.
    $S =$ knnExtend(S, w, p, k).
  end while
else
  Let W be the set of objects in v_O.
  Sort W according to the distance of their MBR to p.
  Exclude from W all nodes with their MBR distance to p larger than m.
  while $|S| < k$ **do**
    Let w be the first element of W and let $W = W \backslash w$.
    $S = S \cup \{w\}$.
  end while
end if
if $|S| < k$ **then**
  Let r be the parent of v.
  $S =$ knnExtend(S, r, p, k).
end if
return S

Algorithm A.1.11 Subalgorithm for the knn query.

Finds the distance between a point p and a bounding box BB in d dimensions. Note that in implementation we can skip calculating the root, if we are interested only in relative distances (i.e., which is further away than...).

knnDist(p, BB):

Let $l = 0$
Let b^+ denote the supremum coordinate vector of BB
Let b^- denote the infimum coordinate vector of BB
for $i = 1$ to d **do**
  if $p_i < b_i^-$ **then**
    $l = l + (b^- - p_i)^2$
  else
    if $p_i > b_i^+$ **then**
      $l = l + (p_i - b_i^+)^2$
    end if
  end if
end for
return $\sqrt{l}$

A.2. Bisection tree construction

Algorithm A.2.1 The Bisection Algorithm.

This algorithm will build a spatial tree using the bisection principle. It will add n elements $o_1, o_2, \ldots, o_n \in \mathbb{R}^d$ to the tree. The tree will have a predefined height of $h \geq 1$. The o_i are assumed to be contained within some $W \subset \mathbb{R}^d$. Let O denote the set $\{o_1, o_2, \ldots, o_n\}$.

BISECT(O, W, h):

Let W be bisected into W_l and W_r by a hyperplane $p = \{x \in W | a \cdot x = b\}$, for some $a \in \mathbb{R}^d$ and $b \in \mathbb{R}$.
Let $W_m = \{x \in W | a \cdot x = b\}$.
Let $W_l = \{x \in W | a \cdot x < b\}$.
Let $W_r = \{x \in W | a \cdot x > b\}$.
Let $O_m = \{o \in O | o \in W_m\}$, and $O = O - O_m$.
Let $O_l = \{o \in O | o \in W_l\}$, and $O = O - O_l$.
Let $O_r = \{o \in O | o \in W_r\} = O$.
Let r be a new spatial tree node, with:

$$f_r(x) = \begin{cases} 1, & \text{if } x \in W \\ 0, & \text{otherwise.} \end{cases}$$

if (h==1) **then**
    Let n_l be a new child node of r with

$$f_{n_l}(x) = \begin{cases} 1, & \text{if } x \in W_l \\ 0, & \text{otherwise} \end{cases}$$

    Similarly construct child nodes n_m and n_r using W_m and W_r.
    Let n_l store the elements in O_l, n_m elements from O_m and n_r elements from O_r.
else
    Let n_l be a new child node of r containing the subtree BISECT$(O_l, W_l, h-1)$.
    Let n_m be a new child node of r containing the subtree BISECT$(O_m, W_m, h-1)$.
    Let n_r be a new child node of r containing the subtree BISECT$(O_r, W_r, h-1)$.
end if
return The spatial tree defined by the root r.

A.3. Basic R-tree construction and manipulation

Algorithm A.3.1 R-tree Insertion Algorithm

This algorithm inserts a spatial object o into the R-tree.

Insert(r, o):

1: Let $BB = \text{MBR}(o)$
2: $v = \text{ChooseLeaf}(r, BB)$ (See Algorithm A.3.2)
3: **if** v has room to store o **then**
4: Store o at v
5: **else**
6: Split v into v_1, v_2 and distribute all children together with o over v_1 and v_2
7: **end if**
8: Update the MBR of v (or v_1 and v_2 when a split occurred), and also update the MBRs of their ancestors.

Algorithm A.3.2 Leaf-Selecting Subalgorithm for the Object Insertion Algorithm

This algorithm selects a leaf node at which an object with MBR BB should be stored at. The location is determined by selecting descendants of r by criterion of least enlargement when o would be added at one of its descendant leaf nodes.

ChooseLeaf(r, BB):

1: **if** r is leaf **then**
2: **return** r
3: **end if**
4: Let $m = \infty$
5: **for all** children v_i of r **do**
6: Let $c = \text{Area}(\text{MBR}(v_i) \cup BB) - \text{Area}(\text{MBR}(v_i))$
7: **if** $c < m$ **then**
8: $m = c$
9: $j = i$
10: **end if**
11: **end for**
12: **return** ChooseLeaf(v_j, BB)

Algorithm A.3.3 Basic R-tree Deletion Algorithm

This algorithm deletes an object o, stored at the tree with root r.

Delete(r, o):

```
1: Find the leaf v which contains o
2: Remove o from v
3: Let S be an empty set
4: X = v
5: while X ≠ r do
6:    if X has less than m children then
7:       Add all children of X to S
8:       Remove X from its parent p
9:       X = p
10:   end if
11: end while
12: Re-insert all entries in S to the R-tree defined by r
13: if X has only one child c then {At this point, X is the root}
14:   Let X = c
15: end if
```

Algorithm A.3.4 Linear Split Algorithm

This algorithm adds an object o to a full leaf node v by splitting it in two, in linear time.

LinearSplit(v, o):

```
Let o_0 = o and o_i be the ith object stored at v.
for i = 1 to d do {Loop over all dimensions}
  Let M be a d by 3 matrix
  Let M[i][1] equal the lower bound coordinate of o on the ith dimension
  Let M[i][2] equal the lower bound coordinate of o on the ith dimension
  Let M[i][3] = M[i][4] = 0
  for all objects o_j stored at v do
    Let t equal the lower bound coordinate of o_j on the ith dimension
    if t < M[i][1] then
      M[i][1] = t
      M[i][3] = j
    else
      if t > M[i][2] then
        M[i][2] = t
        M[i][4] = j
      end if
    end if
  end for
  M[i][1] = M[i][2] − M[i][1]
end for
k = arg max_{1≤i≤d} M[i][1]
Add o_{M[k][3]} to a set S_1 and o_{M[k][4]} to a set S_2
while there are still objects o_i not added to S_1 or S_2 do
  Select randomly a not-added object o_k
  Let t_1 equal the MBR enlargement of S_1 if it would also contain o_k
  Let t_2 equal the MBR enlargement of S_2 if it would also contain o_k
  if t_1 < t_2 then
    Add o_k to S_1
  else
    Add o_k to S_2
  end if
end while
Create leaf nodes v_1 and v_2 from the sets S_1 and S_2
return (v_1, v_2)
```

Algorithm A.3.5 Quadratic Split Algorithm

This algorithm adds an object o to a full leaf node v by splitting it in two, in quadratic time.

QuadraticSplit(v, o):

```
Let o_0 = o and o_i be the ith object stored at v, with 1 ≤ i ≤ M
Select j and k such that the dead space of MBR({o_j, o_k}) is maximum
Add o_j to a set S_1 and o_k to a set S_2
while there are still objects o_i not added to S_1 or S_2 do
  Let p = −∞
  for all objects o_i not yet added do
    Let a be the dead space of MBR(S_1 ∪ {o_i})
    Let b be the dead space of MBR(S_2 ∪ {o_i})
    if ‖a − b‖ > p then
      p = ‖a − b‖
      j = i
    end if
    Add o_j to the set S_1 or S_2, depending on which set requires the least MBR enlargement
  end for
end while
Create leaf nodes v_1 and v_2 from the sets S_1 and S_2
return (v_1, v_2)
```

Algorithm A.3.6 Exponential Split Algorithm

This algorithm adds an object o to a full leaf node v by splitting it in two, in exponential time.

ExponentialSplit(v, o):

```
Let o_0 = o and o_i be the ith object stored at v, with 1 ≤ i ≤ M
for all Possible combinations of splits S_1, S_2, where S_1 ∪ S_2 = {o_0, o_1, ..., o_M} do
  Calculate and remember the combination with the least dead space
end for
Create leaf nodes v_1 and v_2 from the remembered sets S_1 and S_2.
return (v_1, v_2)
```

A.4. TGS bulk-loading tree construction

Algorithm A.4.1 The TGS Bulk Loading Algorithm

The TGS algorithm takes as input a series of n spatial objects stored in the set O, a cost function f_c (as described above) and the maximum number of objects M per node. Also, since the algorithm is recursive, we want to keep track of the desired tree height h, which is thus also supplied. A set of imposed orderings S should also be available.
The algorithm returns a valid R-tree root, and uses subalgorithms A.4.2, A.4.3, and A.4.4. The main difference with the algorithm presented in [**AS07**], is that the input set is not pre-ordered before entering recursion; this would require an $O(n)$ amount of extra storage for each ordering in S after the first. For large datasets, this is simply too much.

TGSBulkLoad(O, f_c, M, S):

1: Let n be the number of objects in O
2: Set

$$h = \max(0, \lceil \frac{\log n}{\log M} \rceil - 1)$$

3: **return** TGSBulkLoadChunk(O, f_c, M, S, h)

Algorithm A.4.2 Subalgorithm for use with TGS bulkloading

This algorithm builds an R-tree with tree depth h. n is the number of input elements, and M the maximum number of children or elements per node. S is a set of imposed orderings on MBRs and f_c is a cost function taking as input two MBRs.

TGSBulkLoadChunk(O, f_c, M, S, h):

1: **if** h==0 **then**
2: Determine the MBR m of all objects in O
3: **return** An internal node containing all objects in O with m as its MBR
4: **else**
5: Set $m = M^h$
6: Set $P =$ TGSPartition(O, f_c, m, S) {O is now divided into M subsets $\{p_1, p_2, \ldots, p_k\} = P$, $k \leq M$}
7: Let N be an empty set of nodes
8: **for all** $p \in P$ **do**
9: $N = N \cup \{$TGSBulkLoadChunk$(p, f_c, M, S, h-1)\}$
10: **end for**
11: Determine the MBR m of all nodes in N
12: **return** An internal node containing the nodes in N as children, and m as its MBR
13: **end if**

Algorithm A.4.3 Subalgorithm for use with TGS bulkloading

This method partitions an input set O into $k \leq M$ subsets using the cost function f_c and the orderings contained in S. Note the difference between M and m; M is the number of maximum children per node which is *not* explicitly passed to this function. Instead, the maximum number of elements which may be stored in the subtree of one of those M children is given; this is m.

TGSPartition(O, f_c, m, S):

```
Let n be the number of objects in O
if n ≤ m then
    return O
else
    (L, H) = TGSBestBinarySplit(O, f_c, m, S)
end if
return TGSPartition(L, m) ∪ TGSPartition(H, m)
```

Algorithm A.4.4 Subalgorithm for use with TGS bulkloading

This method splits an input set O into two sets. This is done by cutting O in partitions of size m after first having sorted O for an ordering in S. By comparing the cost function applied to each pair of partition and selecting the pair resulting in the lowest value, we obtain a best binary split. The best binary split resulting from all different orderings in S is then returned in the variables L and H.

TGSBestBinarySplit(O, f_c, m, S):

```
Let n be the number of elements in O
Set p = ⌈n/m⌉ − 1
c = ∞
Let L and H be empty sets
for all Ordering s ∈ S do
    Sort O according to s
    for i = 1 to p do
        Compute the bounding box b^1 of the MBR containing o_1, o_2, ..., o_mi
        Compute the bounding box b^2 of the MBR containing o_m(i+1), o_m(i+2), ..., o_n
        Set t = f_c(b^1, b^2)
        if t < c then
            Set c = t
            Set L = {o_1, o_2, ..., o_mi}
            Set H = {o_m(i+1), o_m(i+2), ..., o_n}
        end if
    end for
end for
return (L, H)
```

A.5. HilbertTGS bulk-loading

Algorithm A.5.1 The HilbertTGS Bulk Loading Algorithm

Adaptation of Algorithm A.4.1 for efficient use with Hilbert coordinate orderings. This leads to the HilbertTGS bulk-loading method.
The algorithm returns a valid R-tree root, and uses subalgorithms A.4.2, A.4.3, and A.5.2; the latter one used instead of Algorithm A.4.4 for the normal TGS method. Note that the S parameter in Algorithms A.4.2 and A.4.3 are unused in the case of bulk-loading with HilbertTGS.

HilbertTGSBulkLoad(O, f_c, M):

1: Let n be the number of objects in O
2: Set

$$h = \max(0, \lceil \frac{\log n}{\log M} \rceil - 1)$$

3: Sort O according to the Hilbert coordinate of the centre coordinate of each bounding box $o \in O$.
4: **return** TGSBulkLoadChunk$(O, f_c, M, \emptyset, h)$

Algorithm A.5.2 Subalgorithm for use with HilbertTGS bulkloading.

This method adapts Algorithm A.4.4 for use with only the Hilbert ordering. This lessens this methods running time by several linearithmic orders.

HilbertBestBinarySplit$(O, f_c, m, \emptyset)$:

1: Let n be the number of elements in O
2: Set $p = \lceil n/m \rceil - 1$
3: $c = \infty$
4: Let L and H be empty sets
5: **for** $i = 1$ to p **do**
6: Compute the bounding box b^1 of the MBR containing $o_1, o_2, \ldots, o_{mi}$
7: Compute the bounding box b^2 of the MBR containing $o_{m(i+1)}, o_{m(i+2)}, \ldots, o_n$
8: Set $t = f_c(b^1, b^2)$
9: **if** $t < c$ **then**
10: Set $c = t$
11: Set $L = \{o_1, o_2, \ldots, o_{mi}\}$
12: Set $H = \{o_{m(i+1)}, o_{m(i+2)}, \ldots, o_n\}$
13: **end if**
14: **end for**
15: **return** (L, H)

A.6. Hilbert coordinate calculation

Algorithm A.6.1 Algorithm for transforming a coordinate in $\mathbb{R}^d$ to $[0,1]$.

This algorithm transforms some point $p \in \mathbb{R}^d$ to a real number in $[0,1]$, by calculating its Hilbert distance. p is assumed to be contained in the unit cube $[0,1]^3$. The approximating Hilbert curve is given to be of order $L \in \mathbb{N}([1,\infty])$.
This algorithm employs the subalgorithms A.6.2, A.6.3, and A.6.4.

toHilbert3D(p):

1: Let M be a 3 by 8 matrix containing all 8 vertex coordinates (i.e., $(0,0,0)$, $(0,0,1)$, $(0,1,1)$, etc.) of the unit cube on each column
2: Let C be an empty set
3: Determine for which jth column of M (M_j) the coordinate M_j is closest to p using Algorithm A.6.2
4: Set $C = C \cup \{j\}$
5: **for** $i = 0$ to L **do**
6: $M = \text{RefineCoors3D}(M, j)$
7: Determine for which jth column of M (M_j) the coordinate M_j is closest to p using Algorithm A.6.2
8: Set $C = C \cup \{j\}$
9: **end for**
10: **return** fromBase8To10(C)

Algorithm A.6.2 Subalgorithm for use with the toHilbert3D algorithm.

Algorithm used to calculate the distance for use in Hilbert coordinate transformation. The function simply returns the square of the euclidean norm to save calculation time. Let $x, y \in \mathbb{R}^d$.

Distance(x, y) :

1: Set $r = 0$
2: **for** $i = 1$ to d **do**
3: Set $r = r + (x_i - y_i)^2$
4: **end for**
5: **return** r

Algorithm A.6.3 Subalgorithm for use with the toHilbert3D algorithm.

Refines the coordinates in M to reflect the jth subcell of M; see equation 2.18 for a single cell transformation in 2D. See [**BG01**, 7-8] for the full tables.

RefineCoors3D(M, i) :

1: Let M_i denote the coordinate in $\mathbb{R}^d$ at the ith column of M.
2: Let A be a 3 times 8 matrix.
3: **for** $i = 1$ to d **do**
4: Extract the two coordinates M_i, M_j of the original M for which we have to take the middle value according to the table.
5: Set $A_i = \frac{M_i + M_j}{2}$
6: **end for**
7: **return** A

Algorithm A.6.4 Subalgorithm for use with the toHilbert3D algorithm.

This algorithm assumes the input set C contains base-8 digits representing the following base-8 number:

$$0.C_1C_2C_3\ldots C_n, \tag{A.1}$$

where n is the set C size.
This algorithm converts the number in equation A.1 to its decimal (base-10) counterpart.

fromBase8To10(C) :

1: Set $d = 1$.
2: Set $r = 0$.
3: **for** $i = 0$ to n **do**
4: Set $d = 8 \cdot d$
5: Set $r = r + \frac{C_i}{d}$
6: **end for**
7: **return** r

A.7. Hilbert R-tree modification algorithms

Algorithm A.7.1 Inserts a given object into a Hilbert R-tree.

This algorithm inserts the object o into an Hilbert R-tree with r as its root. This function uses Algorithm A.7.1, A.7.4 and A.7.3.

HilbertInsert(r, o) :

1: Let h be the Hilbert coordinate of the centre coordinate of MBR(o)
2: Let $v =$ HilbertChooseLeaf(r, h)
3: Insert o at v
4: **if** v is overflowed **then**
5: HilbertHandleOverflow(v)
6: **end if**
7: HilbertUpdate(v)

Algorithm A.7.2 Selects a leaf node to store an object in with a given Hilbert key value.

This function selects a leaf node descending from v which has the lowest LHV-value higher than h.

HilbertChooseLeaf(v, h) :

1: **if** v is leaf **then**
2: **return** v
3: **end if**
4: Let $c = 1$ and t be an initially empty variable containing a tree node
5: **for all** Children $w \in v_C$ **do**
6: **if** $LHV_w \leq c$ **then**
7: $t = w$
8: $c = LHV_w$
9: **end if**
10: **end for**
11: **return** HilbertChooseLeaf(t, h)

Algorithm A.7.3 Updates an R-tree, starting at a modified Hilbert R-tree node.

Updates a given node v with respect to the MBR and LHV values. Is recursively called on the parent to update all ancestors as well.
This algorithm uses the Hilbert-transformation algorithm as in A.6.1, assuming the tree dimensionality is three. Storing objects of other dimensions require a specialised Hilbert transformation algorithm.

HilbertUpdate(v) :

1: **if** v is leaf **then**
2: Set MBR$(v) = \cup_{o \in v_O}$MBR(o)
3: Set $LHV_v = \max_{o \in v_O}$ toHilbert3D(getCenterCoordinate(MBR(o)))
4: **else**
5: Set MBR$(v) = \cup_{w \in v_C}$MBR(w)
6: Set $LHV_v = \max_{w \in v_C} LHV_w$
7: **end if**
8: **if** v is not root **then**
9: HilbertUpdate(v_{parent})
10: **end if**

Algorithm A.7.4 Hilbert R-tree overflow handling algorithm.

This method handles overflows occurring at an given Hilbert R-tree node v. Note that we count the elements in a set S from 1 to n, where n is the set size. The ith element of S, $i \in [1, n]$ then is denoted S_i.

HilbertOverflow(v) :

1: **if** v is root **then**
2: Construct a new Hilbert R-tree node n with its MBR and LHV-value equal to those of v
3: Let v become a child of n, and thus n become the parent node of v
4: **end if**
5: Let $p = v_{\text{parent}}$
6: Construct a set $\tilde{S}$ containing at most s nodes from p_C with their LHV-values as close to LHV_v as possible
7: Move all entries of the nodes in $\tilde{S}$ into a set C (emptying the nodes in $\tilde{S}$
8: Sort the elements in C according to their Hilbert values
9: **if** All nodes in $\tilde{S}$ are overflowing **then**
10: Construct a new Hilbert R-tree node w with p as its parent, and also add this node to p_C
11: Set $\tilde{S} = \tilde{S} \cup \{w\}$
12: **end if**
13: Let $q = \frac{|C|}{|\tilde{S}|}$
14: **for** $i = 1$ to $|\tilde{S}| - 1$ **do**
15: Add the elements $C_{i \cdot q+1}, C_{i \cdot q+2}, \ldots, C_{(i+1) \cdot q}$ to $\tilde{S}_i$
16: Update the MBR and the LHV-value of $\tilde{S}_i$ accordingly.
17: **end for**
18: Add the elements $C_{(|\tilde{S}|-1) \cdot q+1}, \ldots, C_{|C|}$ to $\tilde{S}_{|\tilde{S}||}$
19: Update the MBR and the LHV-value of $\tilde{S}_{|\tilde{S}||}$ accordingly.
20: **if** p is overflowed **then**
21: HilbertOverflow(p)
22: **end if**

Algorithm A.7.5 Object deletion algorithm for use with Hilbert R-trees.

Object deletion algorithm for Hilbert R-trees. The to-be deleted object o is assumed to be in the tree structure with root node r. This algorithm uses the algorithms A.7.6 and A.7.3.

HilbertDelete(r, o) :

1: Perform a standard R-tree search for MBR(o) on r to find the leaf node v containing o
2: Remove o from v
3: **if** v contains less than the minimum amount of elements allowed **then**
4: HilbertUnderflow(v)
5: **end if**
6: HilbertUpdate(v)

Algorithm A.7.6 Hilbert R-tree underflow handling algorithm.

Resolves an underflow occurred at a given node v. This algorithm performs a merge between s nodes according to the Hilbert values of the elements stored at those nodes. Note that this algorithm is largely similar to the overflow algorithm A.7.4.

HilbertUnderflow(v) :

1: Let $p = v_{\text{parent}}$
2: Construct a set $\tilde{S}$ containing at most s nodes from p_C with their LHV-values as close to LHV_v as possible
3: Move all entries of the nodes in $\tilde{S}$ into a set C (emptying the nodes in $\tilde{S}$
4: Sort the elements in C according to their Hilbert values
5: **if** All nodes in $\tilde{S}$ are underflowing **then**
6: Delete an arbitrary node from $\tilde{S}$
7: **end if**
8: Let $q = \frac{|C|}{|\tilde{S}|}$
9: **for** $i = 1$ to $|\tilde{S}| - 1$ **do**
10: Add the elements $C_{i \cdot q+1}, C_{i \cdot q+2}, \ldots, C_{(i+1) \cdot q}$ to $\tilde{S}_i$
11: Update the MBR and the LHV-value of $\tilde{S}_i$ accordingly.
12: **end for**
13: Add the elements $C_{(|\tilde{S}|-1) \cdot q+1}, \ldots, C_{|C|}$ to $\tilde{S}_{|\tilde{S}||}$
14: Update the MBR and the LHV-value of $\tilde{S}_{|\tilde{S}||}$ accordingly.
15: **if** p is underflowed **then**
16: HilbertUnderflow(p)
17: **end if**

APPENDIX

B

TWO

Software Architecture

In this chapter we will briefly discuss the rationale of the C++ software written for obtaining the experimental results. Since providing a faster alternative than the bisection method in essence was the main goal of this thesis, the software written had to be of good quality. Also, since Alten has interest in continuing development and finding more applications for R-trees, the software furthermore had to be as generic as possible and good documentation was necessary as well.

To achieve this, initially a framework was set up which supported 'plug-and-play'-like support for R-tree variations as well as support for the original Shell bisection code. Also, a parser was developed that was able to read the so-called .inc-files which essentially describes a reservoir discretisation. From this point on the framework evolved into a true generic library with easy-to-use interfaces and excellent encapsulation.

What follows now is first a display of the object-oriented structure, followed by a description of the interfaces a developer will need to use when employing this R-tree library for his or her own application. This is then followed by more advanced options for using alternatively shaped bounding boxes or metrics.

B.1. Object-Oriented Structure

Figure B.1 shows a very basic Unified Modelling Language (UML) diagram denoting the overall structure of the library. Each box denotes a separate object, or rather, a *class*. The basic arrows denote generalisations; the arrow pointing from *Spatial Tree* to *Tree* denotes that the Tree class is a generalisation of the Spatial Tree class, or in more technical terms, the Spatial Tree class extends the Tree class. The dotted arrows denote which class is dependent on which other class; for example, the *R-tree* class cannot work without a *Bounding Box*.

Class names which are printed italic denote which classes are *abstract*. Abstract classes represent objects which are not completely specified; for example, the *Ordering* class which main purpose is to define some (semi-)ordering on other objects. However, it is impossible to supply a generic ordering which works for any object; we cannot specify such an order and as such the Ordering class is abstract. On the other hand, it is undeniable that ordering arbitrary objects is a general case of, for example, ordering cubic bounding boxes by their Hilbert coordinates. In implementation, this abstraction can be useful when we want to implement multiple different orderings; we can for example design a sorting algorithm which works with any subclass of the Orderings class.

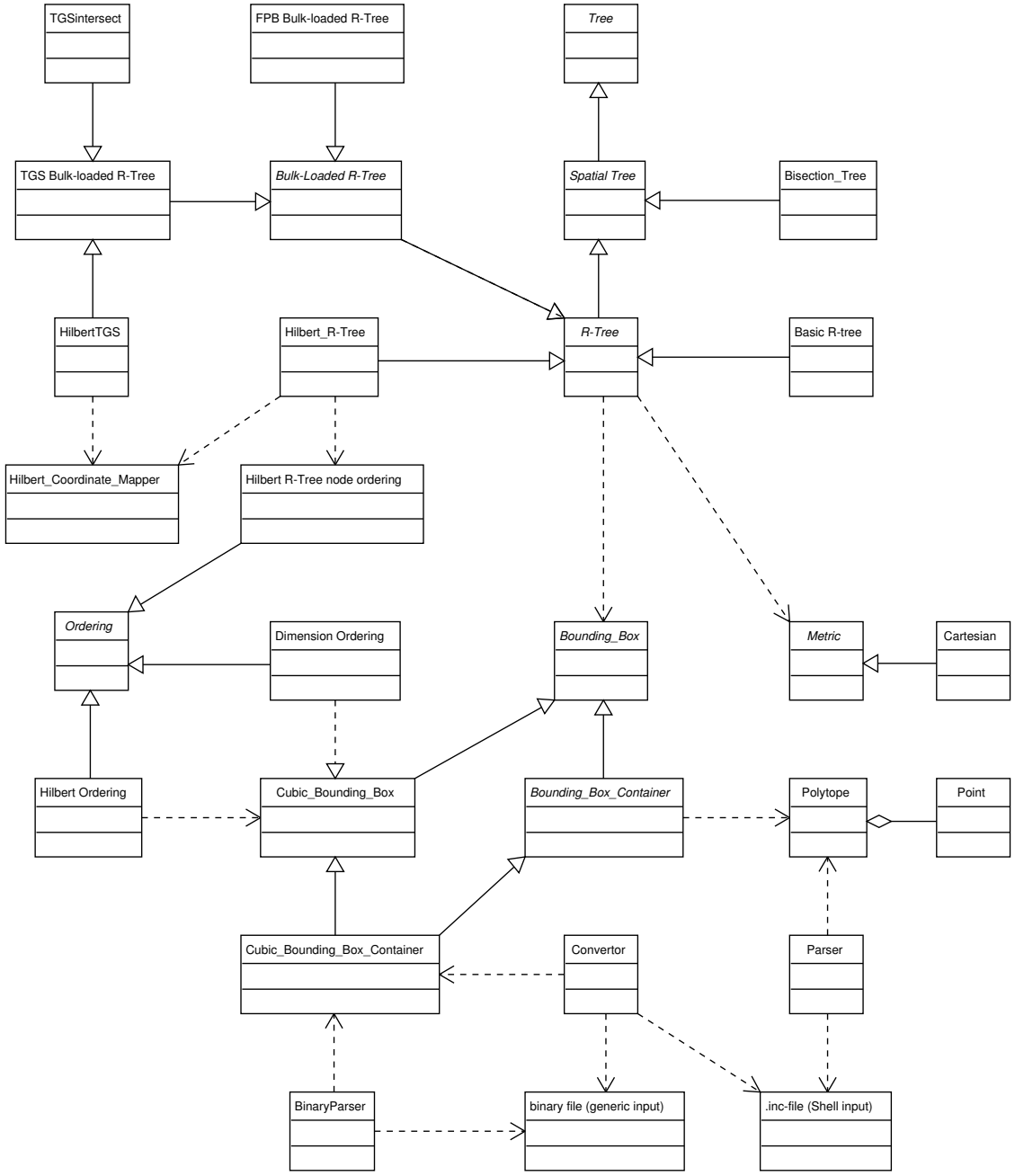


FIGURE B.1. Class diagram of the R-tree library. Italic classnames denote abstract classes.

For more information about Object Oriented programming in C++ we refer to [Eck00a] and [Eck00b]. For more information on UML, see for example [DWT01]. Also a good read is [RM03], which among others contains extensive examples using templates.

B.2. Interfaces

Interfaces are the most important aspect of classes for a developer who want to *use* the class. An interface basically consists of those functions of a class which are publicly visible. We will proceed with all relevant library classes and will describe the public functions relevant for typical

spatial storage use. For a more detailed description of all public functions of these classes, we refer to the doxygen documentation. Note that this section is meant for those who want to use an R-tree (as implemented in our experimentation software) in their own software, and thus is quite technical. Availability of the software code and doxygen documentation is assumed.

B.2.1. Trees. This is the base class of all trees in the current library. It defines the basic methods one would expect from a general Tree datastructure.

isRoot(). Returns a Boolean indicating if the current tree instance is the root node.

getRoot(). Returns a node of the same type as the current node, which is the root node of the tree the current node is in.

isEmpty(). Returns a Boolean indicating if the current tree is empty; i.e. contains only the root which does not store anything.

insert(newElement). Inserts an element into the tree. Note that the type of element is set fixed upon the tree declaration, using the templated variable *stored_type*. However, also note that this template variable is hidden from the user using R-trees; there, the stored type is a *Cubic_Bounding_Box_Container*.

insertElements(begin, end). Inserts all elements from the iterator at begin, until (not including) the iterator end.

B.2.2. Spatial Trees. This is the base class of all trees allowing storage and querying of spatial data.

checkReady(). Returns a Boolean indicating if the current spatial tree instance is ready for querying.

ensureReady(). Function which makes sure the current spatial tree is ready for querying. This method should always be called before the first query is executed. Once a tree is ready for querying it remains ready, even if the tree structure is modified (e.g., insertion, deletion); hence this function does not have to be called more than once.

neighboursOf(point, k). Searches the k closest stored object to a point in $\mathbb{R}^d$, where d is the spatial tree dimension. Returns a vector containing the indexes (*unsigned int*s) of the stored objects satisfying this query, as all query functions below do.

containedIn (box). Searches which objects are contained in a given box (of type *Cubic_Bounding_Box*). Returns a vector of indexes.

intersects (point). Searches which objects intersect with a given point (of type *Point*). Returns a vector of indexes.

intersects (begin, end). Searches which objects intersect with a line segment starting at point begin and ending at end. Begin and end are of type *vector¡double¿*. Returns a vector of indexes.

B.2.3. R-trees. The R-trees themselves inherit the interfaces from the Spatial Tree and the more general Tree classes. New functions introduced at this level are tree saving and loading. Note that these functions are not very advanced; files are written and read as simple text files and information on the R-tree variation used is not stored in the file; the user should make sure the files being read in to a given R-tree are written by an R-tree of similar variation. If not, errors may occur.

writeTreeToFile(filename). Writes the current tree to a text file.

readTreeFromFile(filename). Loads a tree from a given text file.

B.2.4. Bulk-loading. Bulk-loading an R-tree results in a so-called static R-tree. However, we have implemented bulk-loading methods in such a way that a given dynamic R-tree is bulk-loaded; after bulk-loading, the R-tree behaves as a dynamic R-tree and insertion and removal operations thus are still supported. Hence when one wants to use bulk-loaded R-trees, a template

parameter must be given denoting the type of dynamic R-tree which should be bulk-loaded. For example, one can construct a TGS bulk-loaded Hilbert R-tree as follows:

```
//declare
TGS_tree< Hilbert_R_tree > my_tree;
//get begin and end iterators from some source
source::iterator begin = source.begin();
source::iterator end = source.end();
//insert elements
my_tree.insertElements( begin, end );
//we're done inserting and want to do querying
my_tree.ensureReady();
//do our queries
my_tree.intersects(....
```

B.3. Alternative bounding box shapes and metrics

As mentioned earlier, the bounding boxes used are the *Cubic_Bounding_Box* (internally and for box queries) and the therefrom derived *Cubic_Bounding_Box_Container*, used for storing the objects in the leaf nodes. As the name indicates, these boxes are cubically shaped. Of course, a different shape could also be used, for example, a spherical bounding box; to achieve this, one can simply interchange the aforementioned bounding box classes with other classes; for example *Spherical_Bounding_Box* and *Spherical_Bounding_Box_Container*. The interface these classes should implement is the interface of the abstract class *Bounding_Box*; see the doxygen documentation.

Successful implementation of a bounding box class also involves implementing a default ordering; see the class *Ordering* for an interface descriptions, and the file `orderings.h` for example orderings. Note that typically, implementing an ordering consists of only overloading the `getVal()` function.

Operations such as calculating the distance between two points, subtraction and additions of two vectors, and other functions, a specialised metric class has been implemented. The standard Euclidean metric has been implemented in the class *Cart*. Other metrics (polar, spherical, etc.) can easily be implemented as well by extending the abstract *Metric* class. This modularity can be useful if we want to store data on roads on a global level, for which a coordinate system different than the Euclidean system may be preferable.

APPENDIX

C

THREE

Detailed experiment results

C.1. Number-of-children experiment

C.1.1. Settings. The following parameters have been kept constant during this experiment.

Constant parameter	Value
Number of queries	1000
Hilbert curve order	4
Compiler	GNU
Optimisation level	debug
Machine	Rivium

C.1.2. Tables.

	384	8484	11352	486484	3106719
Bisection tree.	0	1	2	153	1080
2/4 non-bulk-loaded basic R-tree.	0	19	28	1326	9266
2/6 non-bulk-loaded basic R-tree.	0	20	24	1291	9214
4/6 non-bulk-loaded basic R-tree.	1	18	22	1130	7793
2/12 non-bulk-loaded basic R-tree.	1	22	32	1507	10329
4/12 non-bulk-loaded basic R-tree.	0	23	30	1600	10764
8/12 non-bulk-loaded basic R-tree.	1	19	24	1217	8366
12/12 non-bulk-loaded basic R-tree.	0	10	19	645	32763
2/24 non-bulk-loaded basic R-tree.	1	34	47	2128	14976
4/24 non-bulk-loaded basic R-tree.	1	34	44	2179	15597
8/24 non-bulk-loaded basic R-tree.	1	36	49	2229	15708
12/24 non-bulk-loaded basic R-tree.	1	34	48	2208	14053

TABLE C.1.1. Building times in processor ticks. Horizontally we have the grid size (number of grid elements), while vertically we have the different data structures used.

	384	8484	11352	486484	3106719
Bisection tree.	0.001	0.007	0.017	0.434	1.422
2/4 non-bulk-loaded basic R-tree.	0.001	0	0.004	0.098	0.209
2/6 non-bulk-loaded basic R-tree.	0	0	0.003	0.069	0.137
4/6 non-bulk-loaded basic R-tree.	0.001	0.002	0.007	0.126	0.223
2/12 non-bulk-loaded basic R-tree.	0	0	0.001	0.042	0.065
4/12 non-bulk-loaded basic R-tree.	0	0	0.002	0.085	0.152
8/12 non-bulk-loaded basic R-tree.	0	0.002	0.005	0.121	0.23
12/12 non-bulk-loaded basic R-tree.	0.002	0.004	0.01	1.023	5.628
2/24 non-bulk-loaded basic R-tree.	0	0.002	0	0.052	0.063
4/24 non-bulk-loaded basic R-tree.	0	0.002	0	0.052	0.098
8/24 non-bulk-loaded basic R-tree.	0	0	0.004	0.063	0.174
12/24 non-bulk-loaded basic R-tree.	0.002	0.002	0.005	0.137	0.241

TABLE C.1.2. Different combinations of datastructures and grid sizes and their average query time on random point queries.

	384	8484	11352	486484	3106719
Bisection tree.	0.005	0.042	0.138	3.656	25.636
2/4 non-bulk-loaded basic R-tree.	0.012	0.031	0.084	0.447	3.116
2/6 non-bulk-loaded basic R-tree.	0.01	0.02	0.056	0.298	2.045
4/6 non-bulk-loaded basic R-tree.	0.01	0.034	0.105	0.53	3.04
2/12 non-bulk-loaded basic R-tree.	0.006	0.018	0.035	0.219	1.427
4/12 non-bulk-loaded basic R-tree.	0.007	0.024	0.051	0.328	2.021
8/12 non-bulk-loaded basic R-tree.	0.008	0.034	0.105	0.52	3.809
12/12 non-bulk-loaded basic R-tree.	0.01	0.057	0.134	2.278	14.245
2/24 non-bulk-loaded basic R-tree.	0.008	0.025	0.046	0.265	1.486
4/24 non-bulk-loaded basic R-tree.	0.008	0.026	0.047	0.262	1.692
8/24 non-bulk-loaded basic R-tree.	0.007	0.03	0.071	0.288	2.551
12/24 non-bulk-loaded basic R-tree.	0.008	0.031	0.097	0.505	3.636

TABLE C.1.3. Different combinations of datastructures and grid sizes and their average query time on random knn queries.

	384	8484	11352	486484	3106719
Bisection tree.	0.003	0.033	0.06	1.574	5.146
2/4 non-bulk-loaded basic R-tree.	0	0.011	0.013	0.708	2.628
2/6 non-bulk-loaded basic R-tree.	0.001	0.014	0.007	0.603	1.873
4/6 non-bulk-loaded basic R-tree.	0.002	0.022	0.022	0.9	2.997
2/12 non-bulk-loaded basic R-tree.	0.001	0.016	0.007	0.526	1.29
4/12 non-bulk-loaded basic R-tree.	0.001	0.018	0.012	0.736	2.109
8/12 non-bulk-loaded basic R-tree.	0.001	0.022	0.028	0.931	3.34
12/12 non-bulk-loaded basic R-tree.	0.001	0.035	0.042	2.329	10.606
2/24 non-bulk-loaded basic R-tree.	0.001	0.022	0.004	0.644	1.325
4/24 non-bulk-loaded basic R-tree.	0	0.018	0.003	0.64	1.689
8/24 non-bulk-loaded basic R-tree.	0.002	0.022	0.014	0.694	2.508
12/24 non-bulk-loaded basic R-tree.	0.001	0.024	0.026	1.014	3.49

TABLE C.1.4. Different combinations of datastructures and grid sizes and their average query time on random line queries.

	384	8484	11352	486484	3106719
Bisection tree.	0.003	0.037	0.059	3.999	21.737
2/4 non-bulk-loaded basic R-tree.	0.004	0.049	0.029	3.044	15.503
2/6 non-bulk-loaded basic R-tree.	0.003	0.048	0.024	2.598	13.904
4/6 non-bulk-loaded basic R-tree.	0.003	0.058	0.047	3.104	15.291
2/12 non-bulk-loaded basic R-tree.	0.001	0.061	0.023	2.931	12.849
4/12 non-bulk-loaded basic R-tree.	0.003	0.064	0.027	3.071	13.903
8/12 non-bulk-loaded basic R-tree.	0.005	0.065	0.064	3.45	15.441
12/12 non-bulk-loaded basic R-tree.	0.008	0.098	0.151	4.959	20.145
2/24 non-bulk-loaded basic R-tree.	0.002	0.08	0.031	3.413	13.059
4/24 non-bulk-loaded basic R-tree.	0.005	0.081	0.029	3.42	13.439
8/24 non-bulk-loaded basic R-tree.	0.005	0.079	0.051	3.517	14.573
12/24 non-bulk-loaded basic R-tree.	0.005	0.083	0.093	3.79	15.918

TABLE C.1.5. Different combinations of datastructures and grid sizes and their average query time on random box queries.

C.2. Twelve children experiments

Note: experiments with RandomTGS with $M = 12$ have been skipped due to too large construction times.

C.2.1. Settings. The following parameters have been kept constant during this experiment.

Constant parameter	Value
Number of queries	1000
Hilbert curve order	3
Compiler	GNU
Optimisation level	O3
Machine	Qire
Minimum number of children	2
Maximum number of children	12
TGS cost function	Sum of volumes

C.2.2. Tables. The following tables have been derived from the experiments using a maximum number of children and objects per node of 12. The datastructures tested are:

- Bisection tree
- Hilbert R-tree
- RandomTGS
- HilbertTGS

	384	8484	11352	486484	3106719
Bisection tree.	0	1	1	103	736
2/12 non-bulk-loaded Hilbert R-tree.	1	38	53	2475	16062
2/12 RandomTGS bulk-loaded basic R-tree.	0	8	47	36387	-
2/12 HilbertTGS bulk-loaded basic R-tree.	0	5	8	809	18842

TABLE C.2.1. Building times in processor ticks. Horizontally we have the grid size (number of grid elements), while vertically we have the different data structures used.

	384	8484	11352	486484	3106719
Bisection tree.	94.432	407.386	1204.72	5284.73	17629.5
2/12 non-bulk-loaded Hilbert R-tree.	27.171	53.049	56.532	466.556	796.614
2/12 RandomTGS bulk-loaded basic R-tree.	10.388	27.506	13.455	86.662	-
2/12 HilbertTGS bulk-loaded basic R-tree.	30.752	60.056	45.681	472.104	1558.75

TABLE C.2.2. Different combinations of datastructures and grid sizes and their average number of touched nodes on random point queries.

	384	8484	11352	486484	3106719
Bisection tree.	324.282	2432.48	8155.96	54215.8	459380
2/12 non-bulk-loaded Hilbert R-tree.	86.174	168.435	417.308	938.828	7998.3
2/12 RandomTGS bulk-loaded basic R-tree.	65.353	105.001	148.874	111.343	-
2/12 HilbertTGS bulk-loaded basic R-tree.	91.626	188.13	407.456	1064.86	14369.9

TABLE C.2.3. Different combinations of datastructures and grid sizes and their average number of touched nodes on random knn queries.

	384	8484	11352	486484	3106719
Bisection tree.	171.851	1176.74	2633.34	16304.3	59755.4
2/12 non-bulk-loaded Hilbert R-tree.	40.846	394.632	143.064	5429.47	18332.5
2/12 RandomTGS bulk-loaded basic R-tree.	20.804	258.902	42.097	1600.44	-
2/12 HilbertTGS bulk-loaded basic R-tree.	42.456	422.41	107.348	5836.97	32482.8

TABLE C.2.4. Different combinations of datastructures and grid sizes and their average number of touched nodes on random line queries.

	384	8484	11352	486484	3106719
Bisection tree.	198.645	2338.82	3838.6	63668.8	426667
2/12 non-bulk-loaded Hilbert R-tree.	63.544	1385.73	564.782	49396.4	326511
2/12 RandomTGS bulk-loaded basic R-tree.	42.304	1209.84	405.483	36398.5	-
2/12 HilbertTGS bulk-loaded basic R-tree.	64.884	1403.66	517.143	49459.2	357921

TABLE C.2.5. Different combinations of datastructures and grid sizes and their average number of touched nodes on random box queries.

	384	8484	11352	486484	3106719
Bisection tree.	0.001	0.006	0.015	0.451	1.273
2/12 non-bulk-loaded Hilbert R-tree.	0	0.001	0	0.009	0.015
2/12 RandomTGS bulk-loaded basic R-tree.	0	0	0	0	-
2/12 HilbertTGS bulk-loaded basic R-tree.	0	0	0	0.008	0.023

TABLE C.2.6. Different combinations of datastructures and grid sizes and their average query time on random point queries.

	384	8484	11352	486484	3106719
Bisection tree.	0.003	0.033	0.116	3.715	24.701
2/12 non-bulk-loaded Hilbert R-tree.	0.001	0.004	0.009	0.036	0.268
2/12 RandomTGS bulk-loaded basic R-tree.	0	0.003	0.002	0.007	-
2/12 HilbertTGS bulk-loaded basic R-tree.	0.002	0.005	0.007	0.033	0.378

TABLE C.2.7. Different combinations of datastructures and grid sizes and their average query time on random knn queries.

	384	8484	11352	486484	3106719
Bisection tree.	0.003	0.025	0.044	1.521	4.434
2/12 non-bulk-loaded Hilbert R-tree.	0	0.003	0	0.193	0.514
2/12 RandomTGS bulk-loaded basic R-tree.	0	0.004	0.001	0.043	-
2/12 HilbertTGS bulk-loaded basic R-tree.	0	0.002	0.002	0.142	0.732

TABLE C.2.8. Different combinations of datastructures and grid sizes and their average query time on random line queries.

	384	8484	11352	486484	3106719
Bisection tree.	0.002	0.033	0.048	4.326	20.592
2/12 non-bulk-loaded Hilbert R-tree.	0	0.019	0.007	1.38	8.829
2/12 RandomTGS bulk-loaded basic R-tree.	0	0.017	0.006	0.809	-
2/12 HilbertTGS bulk-loaded basic R-tree.	0.001	0.018	0.005	1.206	8.405

TABLE C.2.9. Different combinations of datastructures and grid sizes and their average query time on random box queries.

	384	8484	11352	486484	3106719
2/12 non-bulk-loaded Hilbert R-tree.	1000	7200	3533.33	5384.62	12570
2/12 RandomTGS bulk-loaded basic R-tree.	Always worse	1000	3066.67	80452.3	-
2/12 HilbertTGS bulk-loaded basic R-tree.	Always worse	500	533.333	1645.6	15162.4

TABLE C.2.10. Different combinations of datastructures and grid sizes and their turnover points on random point queries, wrt the Bisection tree.

	384	8484	11352	486484	3106719
2/12 non-bulk-loaded Hilbert R-tree.	500	1310.34	495.327	643.653	627.266
2/12 RandomTGS bulk-loaded basic R-tree.	Always worse	266.667	403.509	10223.3	-
2/12 HilbertTGS bulk-loaded basic R-tree.	Always worse	142.857	82.5688	190.657	752.498

TABLE C.2.11. Different combinations of datastructures and grid sizes and their turnover points on random knn queries, wrt the Bisection tree.

	384	8484	11352	486484	3106719
2/12 non-bulk-loaded Hilbert R-tree.	333.333	1681.82	1181.82	1843.37	3955.61
2/12 RandomTGS bulk-loaded basic R-tree.	Always worse	333.333	1093.02	25657	-
2/12 HilbertTGS bulk-loaded basic R-tree.	Always worse	217.391	166.667	517.041	4890.6

TABLE C.2.12. Different combinations of datastructures and grid sizes and their turnover points on random line queries, wrt the Bisection tree.

	384	8484	11352	486484	3106719
2/12 non-bulk-loaded Hilbert R-tree.	500	2785.71	1268.29	876.782	1390.89
2/12 RandomTGS bulk-loaded basic R-tree.	Always worse	437.5	1071.43	10538	-
2/12 HilbertTGS bulk-loaded basic R-tree.	Always worse	266.667	162.791	230.449	1489.13

TABLE C.2.13. Different combinations of datastructures and grid sizes and their turnover points on random box queries, wrt the Bisection tree.

C.2.3. Graphs. The following graphs visualise the data in the previous tables.

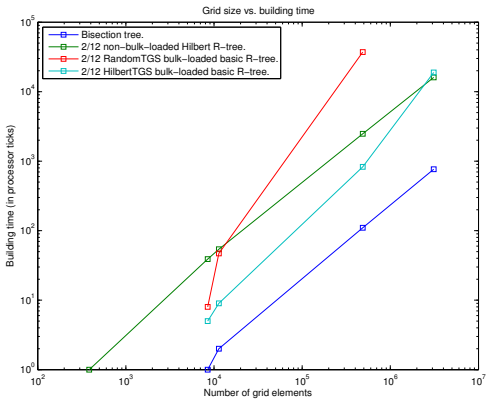
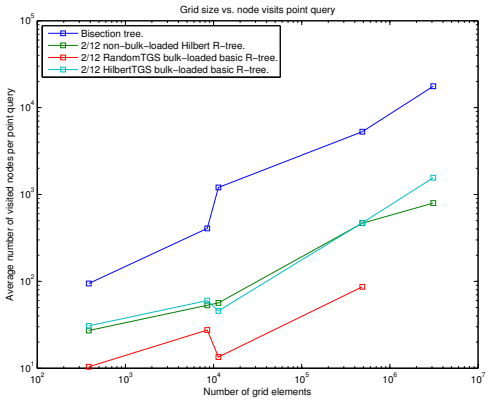
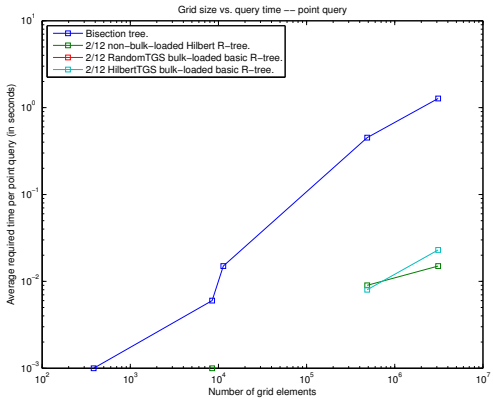


FIGURE C.1. Build times of different datastructures

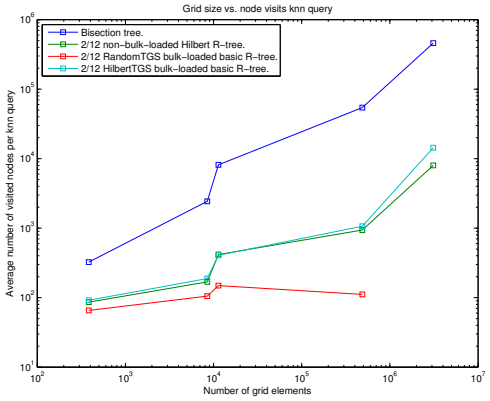


(a) Number of touched nodes versus grid size, for point queries

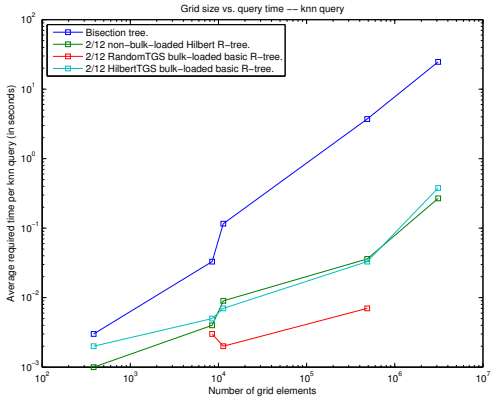


(b) Wall-clock query time versus grid size, for point queries

FIGURE C.2. Graphs detailing performance on point queries

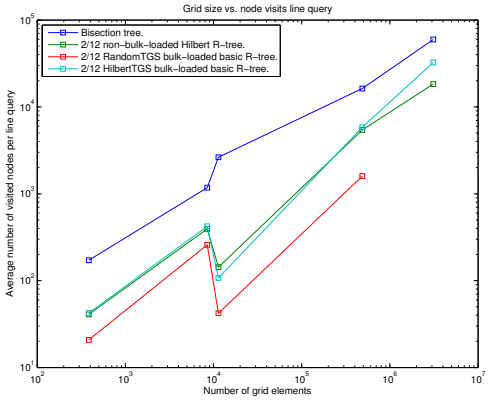


(a) Number of touched nodes versus grid size, for knn queries

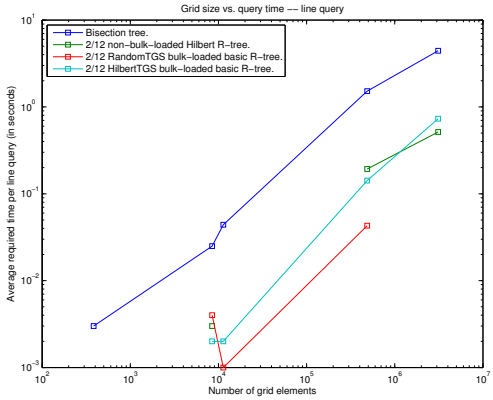


(b) Wall-clock query time versus grid size, for knn queries

FIGURE C.3. Graphs detailing performance on knn queries

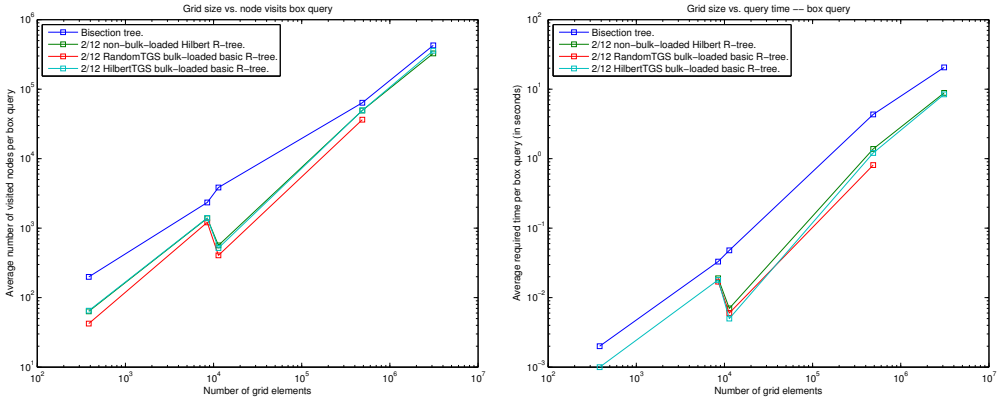


(a) Number of touched nodes versus grid size, for line queries



(b) Wall-clock query time versus grid size, for line queries

FIGURE C.4. Graphs detailing performance on line queries



(a) Number of touched nodes versus grid size, for box queries

(b) Wall-clock query time versus grid size, for box queries

FIGURE C.5. Graphs detailing performance on box queries