DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2020

A comparison of different R-tree construction techniques for range queries on neuromorphological data

JOHAN GRUNDBERG

AXEL ELMARSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

En jämförelse av R-trädskonstruktionstekniker för sökningar i neuronal morfologidata

JOHAN GRUNDBERG
AXEL ELMARSSON

Degree Project in Computer Science, DD142X
Date: June 2020
Supervisor: Alexander Kozlov
Examiner: Pawel Herman
School of Electrical Engineering and Computer Science

Abstract

The brain is the most complex organ in the human body, and there are many reasons to study it. One way of studying the brain is through digital simulations. Such simulations typically require large amounts of data on the 3-dimensional structure of individual neurons to be stored and processed efficiently. Commonly, such data is stored using data structures for spatial indexing such as R-trees. A typical operation within neuroscience which needs to make use of these data structures is the range query: a search for all elements within a given subvolume of the model. Since these queries are common, it is important they can be made efficiently. The purpose of this study is to compare a selection of construction methods (repeated R*-tree insertion, one-dimensional sorting packing, Sort-Tile-Recursive (STR) packing, Adaptive STR packing, Hilbert/Z-order/Peano curve packing, and binary splitting packing) for R-trees with respect to their performance in terms of building time, query page reads and query execution time. With reconstructions of neurons from the human brain, ten datasets were generated with densities ranging from 5,000 to 50,000 neurons/mm^3 in a $300\,\mu\mathrm{m} \times 600\,\mu\mathrm{m} \times 300\,\mu\mathrm{m}$ volume. Range queries were then run on the R-trees constructed from these datasets. The results show that the lowest query times were achieved using STR packing and Adaptive STR packing. The best performing construction techniques in terms of build time were Peano and Z-order curve packing.

Sammanfattning

Hjärnan är kroppens mest komplicerade organ och det finns många anledningar att studera den. Ett sätt att studera hjärnan är genom datorsimuleringar. Sådana datorsimuleringar kräver en stor mängd tredimensionell neurondata som behöver lagras och behandlas effektivt. R-träd är en vanlig datastruktur för att behandla sådan data, och en vanlig operation man inom neurovetenskapen vill genomföra är sökningar efter element i en given delvolym av modellen man arbetar med. Det är därför viktigt att dessa operationer kan genomföras effektivt. Syftet med denna studie är att jämföra ett urval av konstruktionstekniker (upprepad R*-trädsinsättning, endimensionell sorteringspackning, STR-packning, adaptiv STR-packning, Hilbert-packning, Z-ordningspackning, Peano-kurvpackning samt binär klyvpackning) för R-träd med avseende på prestanda i konstruktionstid, antal sidinläsningar och exekveringstid för sökningar. Tio datamängder genererades med nervcellsdensiteter mellan 5,000 och 50,000 celler/mm^3 i en volym på $300\,\mu\mathrm{m} \times 600\,\mu\mathrm{m} \times 300\,\mu\mathrm{m}$. Dessa användes sedan för konstruktion av olika R-träd i vilka en sekvens av delvolymssökningar gjordes. Resultaten visar att den lägsta söktiden erhölls med R-träd konstruerade genom STR-packning och adaptiv STR-packning, medan konstruktionstiden var som lägst för packning med Peanokurvan och Z-ordningskurvan.

Terminology

- **Neuromorphology**: The study of the spatial structure of neurons.
- **Spatial index**: Data structure used to store spatial data.
- **R-tree**: A spatial index which stores data inside rectangles (hence the R) or cuboids.
- **Range query**: A search for all elements within a given subvolume of a spatial index data structure (such as an R-tree or R*-tree). There are some slight variations in the definitions used by other authors. Some use the term solely for queries in which the sphere (using an arbitrary distance function) surrounding some point is searched[1]. Others also include cuboidal queries with varying aspect ratios [1]. In the experiments performed in this study, the first definition is used.
- **Packing/bulk loading**: Constructing an R-tree from a previously known dataset without repeated insertion, typically utilizing some form of sorting. A bulk-loaded R-tree is the same thing as a bulk-loaded R*-tree.
- **STR**: Sort-Tile-Recursive, a bulk-loading method.
- **A-STR**: Adaptive Sort-Tile-Recursive, a bulk-loading method based on STR.
- **MBR**: Minimum Bounding Rectangle/hyperrectangle. The minimal area/volume that encloses data points or MBRs. We use the term MBR in 2D- and 3D-space. Used to group elements in R-trees.
- **Rectangle**: Used to describe hyperrectangles including rectangles and cuboids.

[1] `https://elki-project.github.io/releases/current/doc/de/lmu/ifi/dbs/elki/database/query/range/RangeQuery.html`. Accessed 2020-05-27.

Contents

Chapter 1

Introduction

1.1 Neuroscience and brain simulations

The human brain consists of approximately 86 billion neurons [2] which communicate via electrical and chemical signals. Its complexity is one of the reasons we are yet to fully comprehend how the brain functions.

Understanding the brain has its primary goal in the treatment of brain disorders. According to the World Health Organization, approximately 6.3% of all disability-adjusted life years lost are due to neurological disorders (2005) [3]. Secondly, a better understanding of the structure and function of neurons in the brain could help us better explain how the brain can process complicated information through billions of relatively simple neurons. This is one of the important questions of computational neuroscience. Artificial neural networks has recently become a popular field of research, in which computer scientists and statisticians attempt to build computational models inspired by the apparent functionality of the nervous system. Advancements made within neuroscience can thus contribute to connectionist neural networks research.

As part of the effort by neuroscientists to understand the network structure of the human brain, several projects are ongoing in which researchers construct digital models of the brain in order to be able to simulate neuronal activity and thus study the brain in silico. Three examples of such projects are the Human Brain Project[1], the Blue Brain Project[2], and Project MindScope[3].

[1] `https://www.humanbrainproject.eu/en/`. Accessed 2020-05-27.

[2] `https://www.epfl.ch/research/domains/bluebrain/`. Accessed 2020-05-27.

[3] `https://www.frontiersin.org/10.3389/conf.fncom.2012.55.00033/event_abstract`. Accessed 2020-05-27.

Typically, large-scale digital reconstructions and simulations of neural tissue are computationally expensive tasks, prompting the use of supercomputers. As with any compute-intensive data processing task, improvements in efficiency of the methods used directly translate either to reduced resource usage or the possibility of handling even larger problems. This is clearly a good reason to use methods which are as computationally efficient as possible.

1.2 Brain simulations and data structures

This study is focused on one specific aspect of the simulations, namely the use of spatial indices (i.e. data structures) for storing 3-dimensional digital reconstructions of neuronal morphologies.

In particular, we will look at the creation of such data structures as well as range queries. Range queries are commonly executed in neuroscience [1], and a specific use case is found in [4]. It is clear from [1], [5], [6] and [7] that different spatial indices have different performance implications when used to store neuromorphological data.

One spatial index which can be used for these kinds of queries is the R-tree, a tree which stores data to speed up range queries. Different R-tree variants are commonly used for storing neuromorphological data [1]. An R-tree is generally built either by the repeated insertion of single elements or by a method known as bulk loading, in which prior knowledge of the data to be inserted is used to optimize the resulting data structure. An improved version of this data structure is known as the R*-tree. These data structures are further described in section 2.4.

One thing the aforementioned studies have in common is that they are focused on static datasets, in which an index is built and then queried against. That is, the data does not change over time. When this is the case, it becomes very important to build as efficient a data structure as possible from the beginning, using any and all information about the data to speed up future operations. In these situations, bulk loading (as explained in the terminology section) can be used. This study is concerned with the same problem, and compares a selection of methods some of which have been tested in previous studies and some of which have not.

1.3 Project aim

The aim of this study is to compare a selection of approaches for building efficient R-trees to be used as spatial indices. These approaches are compared by the time it takes to build the tree used, as well as their efficiency for a sequence of range queries.

1.4 Research Question

How do R-trees built using different methods (see list below) perform with respect to index build time, the number of page reads when executing a sequence of range queries, and the execution time of such a sequence, on neuromorphological data?

- Repeated R*-tree insertion
- One-dimensional sorting packing
- Hilbert curve packing
- Z-Order curve packing
- Peano curve packing
- Binary split sorting packing
- STR packing
- Adaptive STR packing (from ELKI)

Chapter 2

Background

2.1 The structure of the brain and nervous system

The human nervous system is a very large collection of communicating cells. These cells are divided into two types: neurons and glial cells (glia). The number of glial cells and neurons is similar [8]. Neurons are specialized cells that communicate with each other through electrical and chemical mechanisms in a very complex way. Like any other cell, a neuron has a cell body. It also has two kinds of outgrowths extending from the cell body: axons and dendrites, sometimes collectively referred to as neurites. Each neuron has at least one axon, which carries output signals to other neurons (for communication) or to muscles (for movement). The axon can branch off several times, enabling output connections to multiple other cells. Glial cells surround the neurons of the nervous system, providing supportive functionality but also affecting the communication of neurons in a seemingly complex way, something which was relatively recently discovered [8]. Due to limited data availability, glial cells fall outside the scope of this study and only neurons are discussed further.

Within each neuron, there is an electric potential which differs from the electric potential outside of it (in the extracellular space), called the membrane potential. When a neuron is sufficiently stimulated through inputs from other cells, the membrane potential rapidly increases in the whole cell. This is called a spike or an action potential. When an action potential occurs, it propagates through the axon and certain molecules called neurotransmitters are released at the specific locations, synapses [8], where the axon contacts the neurites of the receiving cells. This is a very rough simplification of how neurons communicate, but it explains the terminology used in this thesis. It also shows

why the geometrical structure matters for neural networks.

The human brain and the spinal cord make up what is known as the central nervous system. It consists of many different regions, most of which are present and functionally similar in all mammals. The use of animal model systems forms a fundamental part of neuroscience research efforts [9], though a human model is used in this study. Figure 2.1 shows the similarities between the human brain and the rat (rodent) brain.

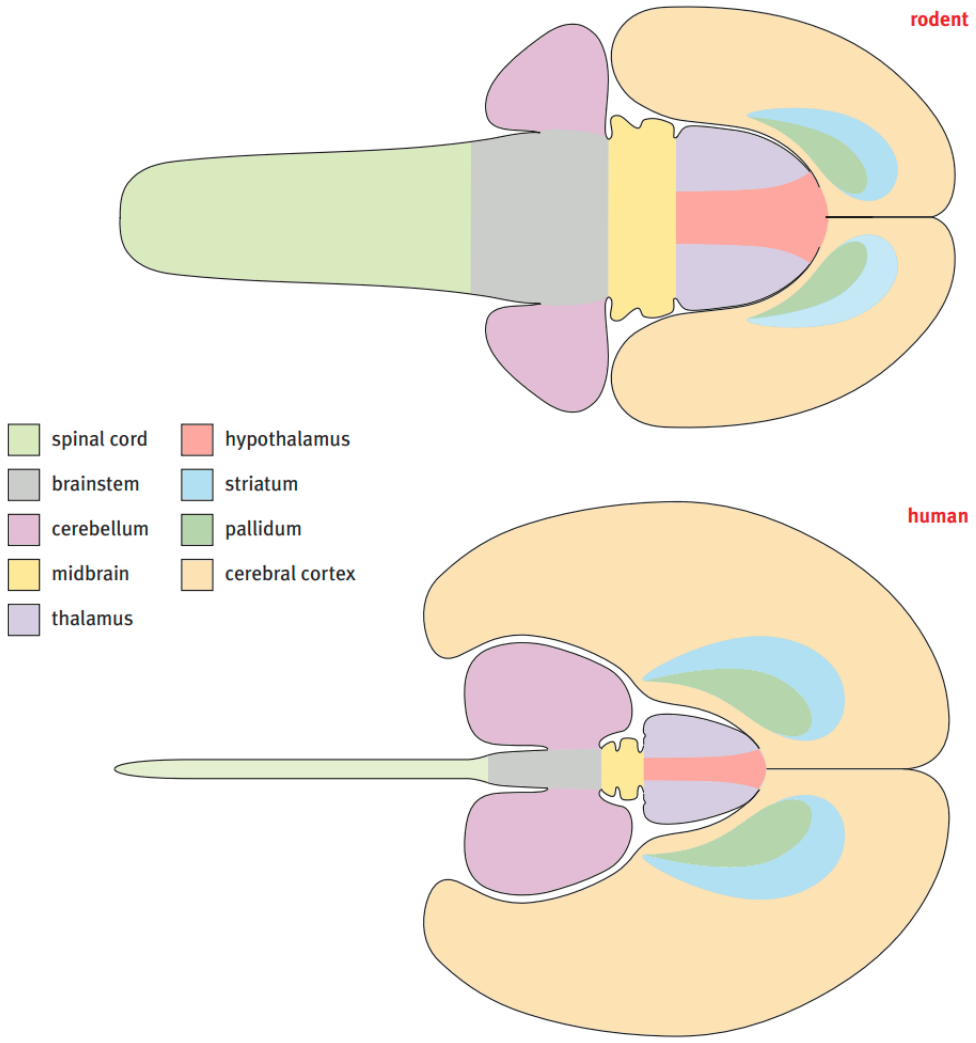


Figure 2.1: A two-dimensional visualization of the different regions of rat and human brains from [9] (with slight color changes). The brains are not equally scaled. The further-developed cerebral cortex is larger in the human brain.

A brain region which is relevant to this study is the cerebral cortex; the out-

ermost part of the brain. One of the most important features of the human brain is that the cerebral cortex is much more developed than in other mammals [9], and this region is associated with complex human behaviour [10], hence making it interesting to study. In the cerebral cortex, neurons appear in a number of different layers with different characteristics in terms of e.g. neuron types and neuronal density [11], which clearly impacts digital reconstructions. One region within the cerebral cortex is the neocortex; the most recently developed part [12]. A further description of the layers in the rat neocortex can be found in [11].

2.2 Digital 3D brain reconstructions

In order to build digital reconstructions of neuronal networks and study these, neuroscientists need to accurately reconstruct individual neurons in a digital representation. This has been done since at least the early 80's, and a very popular system for the reconstruction of neurons from biological samples is Neurolucida [13]. Traditionally, the digital reconstruction of neurons is a time-consuming, partially manual process [13], which eventually results in some sort of file representing the three-dimensional structure of the neuron.

Digitally reconstructing and simulating brains (or more realistically small pieces of nervous tissue) from individual digital models of neurons is a complicated process. It is not the point of this study to detail this process, but an example is briefly described here due to the context it provides. This section is largely a simplified explanation of the experimental process in [11], a very large study undertaken as part of the Blue Brain Project in which neuronal microcircuitry from the somatosensory cortex of young male rats is reconstructed and used for large-scale simulation. A similar reconstruction (of parts of the mouse striatum) is made in [14].

Before any neuronal microcircuit (functional computer model of nervous tissue) can be built, data must be available on the putative structure of it. Firstly, neurons are categorized into types based on their morphology (m-types) and their electrical characteristics (e-types). Combinations of these types are referred to as me-types. For each m-type, density is either determined experimentally or approximated for each different layer. Also, data and rules for synapse formations is collected for use in the synapse construction part of the reconstruction to ensure realistic results.

With the data needed, the reconstruction can begin. Firstly, neurons of all m-types are placed in some previously defined volume according to the densities for all layers in the reconstruction. Next, appositions are found be-

tween axons and dendrites (axo-dendritic appositions). These may or may not be converted to synapses, based on rules for synapse formation. The process of reconstructing the synapses used in this specific project (and thus the connectome of the model) is well described in [4]. When synapses have been constructed, neurons are categorized into different e-types and synapses are categorized based on me-types of the connected neurons. Lastly, synapses are added for thalamocortical fibers (nerves extending from the thalamus region outside the microcircuit) and then the model can be used for simulation.

While the example described above only reconstructs a specific part of the brain in one specific animal, any model of nervous tissue designed in this fashion will contain a large amount of three-dimensional data representing the morphology of a large number of individual neurons. While handling and experimenting with this data, it is likely that range queries will be made [1]. For example, they were used for counting synapses in [4] (mentioned above), using a method proposed in [1].

2.3 Neuromorphological data

In any realistic neuromorphological reconstruction, each neuron is represented by a large number (at least several thousands) of 3-dimensional cylinders or meshes. It is thus easy to see that any reconstruction of nervous tissue will result in large datasets and require efficient data processing.

There are many ways of digitally representing neurons, and there is no consensus on which file format should be used in the scientific community[1]. One popular file format for representing digital reconstructions of single neurons is the SWC file format. SWC files are made up of plain text and thus human-readable. In this format, neurons are represented as a set of points and corresponding radii. Each point is located on a neurite (unless it is part of the cell body), which in turn has some radius where the point is. The first part of the file, the header, is optional but contains metadata about the neuron itself. Each subsequent line of the file contains a list of 7 numbers [15]:

1. the point index
2. a number representing the type of the data point (e.g. cell body, dendrite, axon)
3. x-coordinate in μm

[1] `http://neuromorpho.org/StdSwc1.21.jsp`. Accessed 2020-05-27.

4. y-coordinate in μm
5. z-coordinate in μm
6. radius in μm
7. the index of the parent point

The data is constrained so that one point, which is the root point, has an index of -1. The other points have exactly one parent. By ensuring all parents have a lower index than their children, cycles are avoided and all points are connected [15]. Under these conditions, the SWC file format implicitly specifies a tree in which each node is a point in 3D-space with an associated radius for the neurite.

In order to advance the scientific field of neuroscience, the website `www.neuromorpho.org` aims to maintain a public and very large database of digital reconstructions of neurons. The NeuroMorpho project intends to provide as much coverage as possible of all such reconstructions that have been used in scientific publications, subject to three additional criteria [16]. As of 2012, data from the website had been used in over 200 peer-reviewed publications.

2.4 Data structures for neuromorphological data

2.4.1 The R-tree

The R-tree is a data structure originally proposed by Antonin Guttman in 1984 [17] and is a height balanced search tree similar to the B-tree (a generalized self balancing binary tree). The data structure is designed for database storage and has proven to be highly useful for storing different kinds of data (including neuromorphological), which is why extensions of the algorithm and further research into the algorithm has been performed, mostly to improve the construction algorithm of the tree to minimize overlapping MBRs, which increases query performance. An example of this is bulk loading which is discussed in section 2.5.

The R-tree stores data points in rectangles (MBRs), hence the name R-tree ("R" for rectangle). The rectangle can easily be generalized to higher dimensions: in 3D a rectangle would be represented as a cuboid. This makes

the R-tree usable for storage of both two- and three-dimensional data. The R-tree seeks to minimize the area of each MBR as much as possible.

All data is stored in nodes, either leaf nodes or non-leaf nodes. The original idea is to store each node on one page. A non-leaf node stores references to its children nodes and the minimum bounding rectangle that covers all children MBRs. Meanwhile a leaf node stores a reference to the corresponding data point in the database and the MBR of the data point. If the data point is a non-geometric object, like a point, the leaf node itself can be the point, but when storing geometric data like polygons, an MBR is required. By storing data in MBRs, all data inside an MBR benefit from spatial locality, which speeds up the search query process. A visualization of an R-tree can be seen in figure 2.2.

When inserting data it is added as a leaf to a parent node. If the node we try to append to overflows the maximum number of elements per node we perform a splitting algorithm (of which there are several different versions and implementations) between the node and a second (new) node. The splitting propagates up the tree and should preferably organize the data such that there is no need to examine both new nodes upon a later search query.

The searching algorithm of the R-tree is similar to that of a B-tree. It descends the tree from the root and branches down a subtree, in some cases several subtrees. By separating the dataset into different sized rectangles we can avoid searching in a child node if it is outside the query space, and therefore query faster. If the tree has an overlapping MBR, this is however not the case. The average time complexity of the search algorithm is $O(\log_M n)$ where M is the maximum number of elements in a node and n is the number of data entries.

The deletion algorithm deletes the desired data point and tries to condense the tree by adjusting the MBR which hold the data point and all its parent MBRs. The deletion algorithm also deletes nodes which has too few data entries, which means that it has to reinsert into the tree all data points whose parent has been deleted.

2.4.2 The R*-tree

The R*-tree is a similar but improved upon version of the R-tree proposed by Beckmann et al. in 1990 [18]. It uses the same query and deletion algorithm as the R-tree, but uses different algorithms and heuristics for insertion and splitting, as well as tries to avoid splits overall by reinserting nodes and subtrees into the tree. The R*-tree has a higher construction cost but in return usually

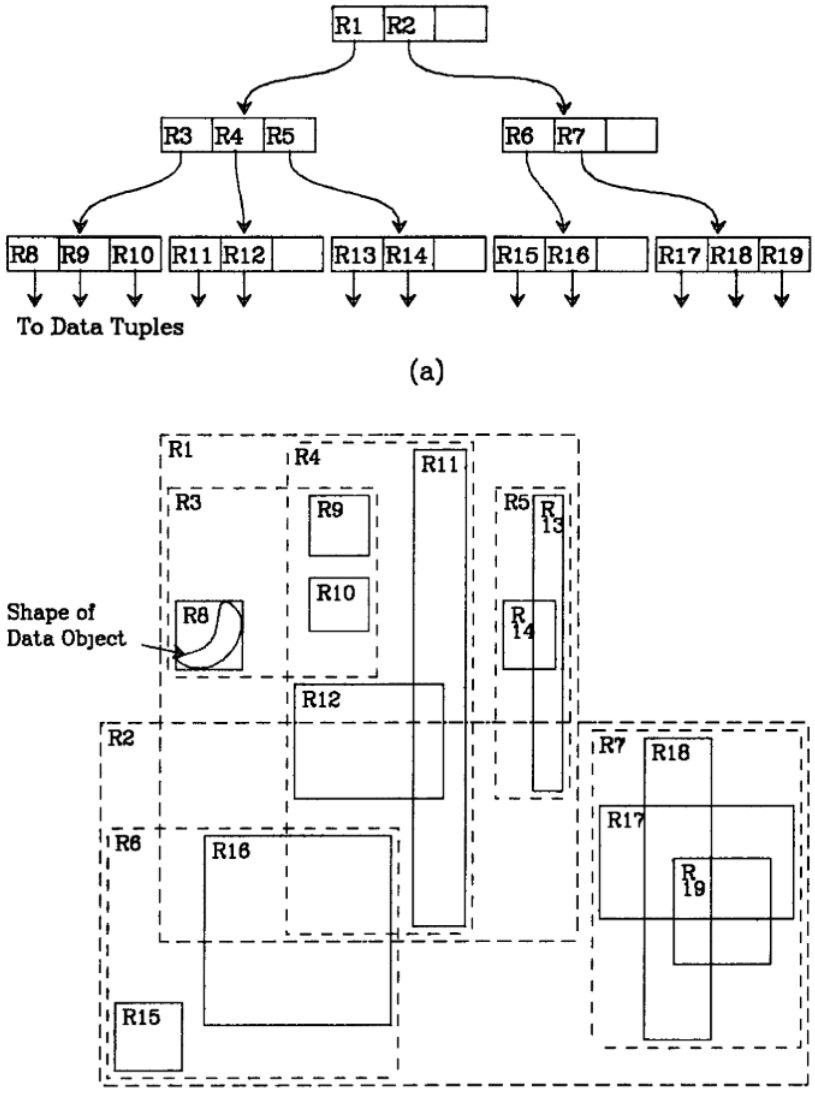


Figure 2.2: A visual representation of the structure of a 2D R-tree. [17] The tree in this figure stores geometric figures, one can be seen in rectangle R8.

has better query performance.

As mentioned earlier, the R-tree only seeks to minimize the area of each of its MBRs in order to reduce the number of paths pursued while querying. The R*-tree expands upon this and adds three other criteria which it tries to find the best possible combination of:

Minimization of overlap between MBRs: With smaller overlap the expected number of paths to follow in a query becomes smaller, increasing query speed, this criterion has the same aim as the one the R-tree pursues.

Minimization of MBR margin: By minimizing the MBRs into more cube-like shapes we can pack the MBRs more tightly and efficiently.

Maximization of storage utilization: With high storage utilization the tree height decreases, improving query time.

When inserting a new node the insertion algorithm has to decide which branch to follow at each level of the tree. It starts at the root and chooses the MBR which requires the least enlargement. For leaf nodes, it instead tries to pursue the overlapping minimization criterion, that is, it tries to find the MBR where enlargement leads to the least overlap increase as possible. If the insertion would overflow, i.e. the node has already reached the maximum amount of children nodes, the R*-tree does not immediately apply the splitting algorithm, which the R-tree would. The R*-tree instead takes a fraction of the entries whose centroid distances from the node centroid is the furthest, and reinserts them into the tree. This reinsertion is costly, therefore only one reinsertion round is allowed per tree level. When overflow cannot be fixed with reinsertion the algorithm instead applies node splitting.

Node splitting is done in two steps: it first chooses a split axis among all dimensions based on perimeter, which it then tries to minimize. When a split axis has been chosen the algorithm sorts the entries on their lower and upper boundaries in the selected dimension and examines all possible divisions. The one it chooses is the one with the least overlap between the MBRs of the resulting nodes. [18][19]

2.5 Bulk loading of R-trees

When using spatial index structures like R-trees to store spatial data, sometimes the data to be stored is known beforehand. However, algorithms for the original R-tree treat insertions as independent operations, using a greedy approach to try and minimize the area or volume enlargement introduced by each one [17]. This approach does not utilize any knowledge of previous or future insertions, doing which could result in a tree with less overlap and better storage utilization.

The solution to this problem is known as *bulk loading* or, equivalently, *packing* [19]. Bulk loading algorithms take as input some dataset $\mathcal{S}$ and produce an R-tree where each data point in $\mathcal{S}$ has been inserted, but not necessarily using conventional insertion algorithms. For example, they may sort the data points in some way and use certain criteria for distributing the points among

leaf nodes to produce a tree with low overlap and high storage utilization. This section describes two major bulk-loading approaches: Spatial sorting and STR (along with a variation called Adaptive STR). Below, we assume c is the capacity of a single node in the R*-tree and n is the number of rectangles to be packed. Figure 2.4 depicts the MBRs of R-trees built with most of the approaches listed below. Since R*-trees and R-trees differ only in terms of the insertion algorithms, there is no difference between a bulk loaded R*-tree and a bulk loaded R-tree.

Firstly, there are the approaches we refer to here as spatial sorting-based bulk loading. In these approaches, a total ordering is imposed on the set of MBRs (or points) to store, which are then sorted based on this ordering. Next, they are split into $\lceil \frac{n}{c} \rceil$ groups, each of which makes up a single node. If the number of nodes created fit in one parent node, the process is complete and all elements are put inside a node which becomes the root node. If not, it continues recursively until a root node is created.

The initial, somewhat naive approach to bulk loading is one-dimensional sorting, presented in 1985 [20]. In this approach, all elements are sorted along the x-axis (or some other axis). The problem with this approach is that if there is much data in the R-tree, the data is split into very thin slices, which is probably not what users of the R-tree will be querying for.

Other spatial sorting approaches sort the data based on their positions along a space-filling curve. One example is Hilbert packing, which sorts the data based on their position along the Hilbert curve [19]. It is also possible to use other space filling curves. In this paper, we examine spatial sorting-based bulk loading using the Hilbert curve, the Z-Order curve and the Peano curve. (With "the Peano curve", we mean the space-filling curve resembling, in early iterations, the letter "N" drawn using only right angles. This is also explained in the source code of the implementation used [2].) The Z-order curve is sometimes abbreviated as the Z-curve in this thesis. Of course, there are as many spatial sorting-based bulk loading strategies as there are methods of sorting MBRs. An additional method for spatial sorting we will examine is binary splitting. In this approach, the rectangles are approximated by their mean values. Initially, they are sorted along the first dimension. Then, the two halves of the resulting list are sorted based on the next dimension. This process is applied recursively until the sublists can no longer be split [3].

[2] `https://github.com/elki-project/elki/blob/master/elki-core-math/src/main/java/elki/math/spacefillingcurves`. Accessed 2020-05-27.

[3] Ibid.

STR (Sort-Tile-Recursive) is a packing method presented in 1997 by Leutenegger et al. [21]. The name of the algorithm is very fitting. Each MBR to store is approximated as a single point; the center of it. The STR algorithm then works by first sorting the data along the first dimension, and then dividing it into evenly sized slices. Next, the algorithm is applied recursively to each slice, but sorting along the next dimension instead. When the last dimension is reached, the elements are simply sorted by that dimension and split into leaf nodes.

In this approach, an equal number of slices is made for each dimension; this can cause non-cubically shaped rectangles if the dataset itself is not approximately cubical (by cubical, we mean each edge has the same length, be it in a rectangle or cuboid). In this study, we therefore also look at a variation of STR called Adaptive STR [4]. This implementation of STR attempts to compensate for non-cubical node shapes potentially resulting from the use of ordinary STR on non-cubical/wide datasets. For an example, compare the MBR thicknesses along the depth axis of figure 2.4E and 2.4F.

The result of the above procedures is that all rectangles are placed into nodes of an R-tree. As is done in the spatial sorting approach, the algorithm is applied recursively if the resulting number of nodes do not fit as children of a single root node [21]. This strategy avoids overlap well as seen in figure 2.3 [19].

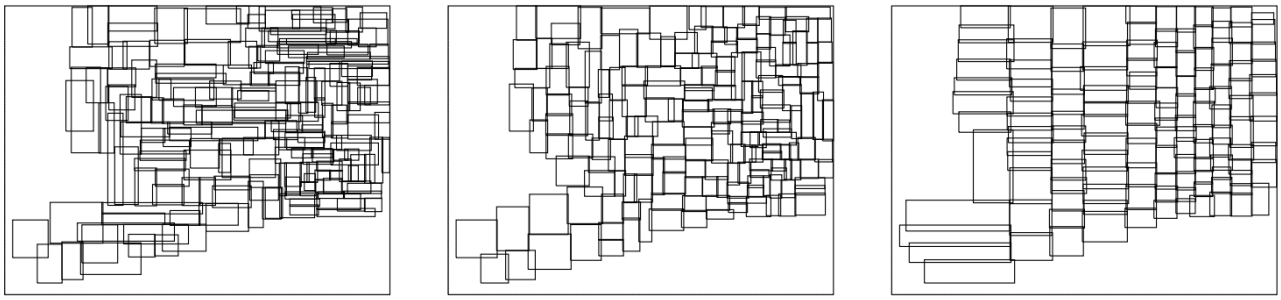


Figure 2.3: The leaf-level MBRs of three 2-dimensional R-tree variants built with the same data: a regular R-tree, a regular R*-tree, and an STR packed R-tree. [19]

[4] `http://elki.dbs.ifi.lmu.de/releases/release0.6.0/doc/de/lmu/ifi/dbs/elki/index/tree/spatial/rstarvariants/strategies/bulk/AdaptiveSortTileRecursiveBulkSplit.html`. Accessed 2020-05-27.

2.6 Software implementations and ELKI

ELKI is a free open source data mining project written in Java, which aims to be easy to use and extend for both students and researchers in the subject. It contains a large collection of algorithms and data structures (for example the R*-tree), but it also has decent and fair benchmarking capabilities and data visualization tools built into it [22]. ELKI separates the data mining algorithms and the data management[5], which makes it possible to use for testing different data structures. With ELKI's algorithms and data structures being highly parameterizable it is a very usable framework for our research task.

2.7 Related research

Many studies have been made on the relative performance of different spatial indices for data in two or more dimensions. Of course, most of them do not involve neuromorphological data. In the paper in which STR was first proposed, some benchmarking is made on R-trees built using STR, Hilbert packing and one-dimensional sorting packing (referred to by the authors as Nearest-X). In this experiment, the authors measured the number of disk accesses made to complete a range query with varying page cache sizes. This research shows that STR slightly outperforms Hilbert packing for the realistic datasets used. It also shows that both STR and Hilbert packing greatly outperform one-dimensional sorting packing for range queries on realistic datasets [21].

We have identified one study comparing the preservation of distance (i.e. the ability of a space-filling curve to keep objects that are close in higher dimensions close in one dimension) when using two different space-filling curves to store spatial data; the Hilbert curve and another curve which resembles the Z-order curve, rotated 90 degrees to the left. This study concludes that the Hilbert space-filling curve performs better when used for range queries on multi-dimensional data [23].

Known indexing approaches have proven to not perform well on dense models required in the neuromorphological field. The overlap in the tree structure increases with increasing level of detail, which slows down the query time [1]. A data structure which has been proven to be efficient on extreme datasets, like very dense neuromorphological models, is the PR-tree; a bulk loaded R-tree which has provably optimal time complexity of range queries [24]. It is

[5] `https://elki-project.github.io/`. Accessed 2020-05-27.

not discussed in detail because it is not used in the experiments of this study.

Tauheed et al. [1] presented a solution to this problem by designing a new scalable indexing approach for dense datasets with high level of detail, which they call FLAT. FLAT is designed to be highly efficient for range queries, which makes it suffer in both time and space when indexing (building the index). The high efficiency for range queries makes it a great algorithm to use for neuroscientists who work with range queries on neurmorphological data. The longer indexing time is countered by the fact that indexing is performed more rarely. In their experiments they have concluded that FLAT outperforms state-of-the-art bulk loaded R-trees, i.e. PR-trees, with up to a factor of eight [1].

Since we could not find a publicly available implementation of FLAT, it is not used for benchmarking in this study. Further, since the PR-tree is not available in ELKI, it is not used either.

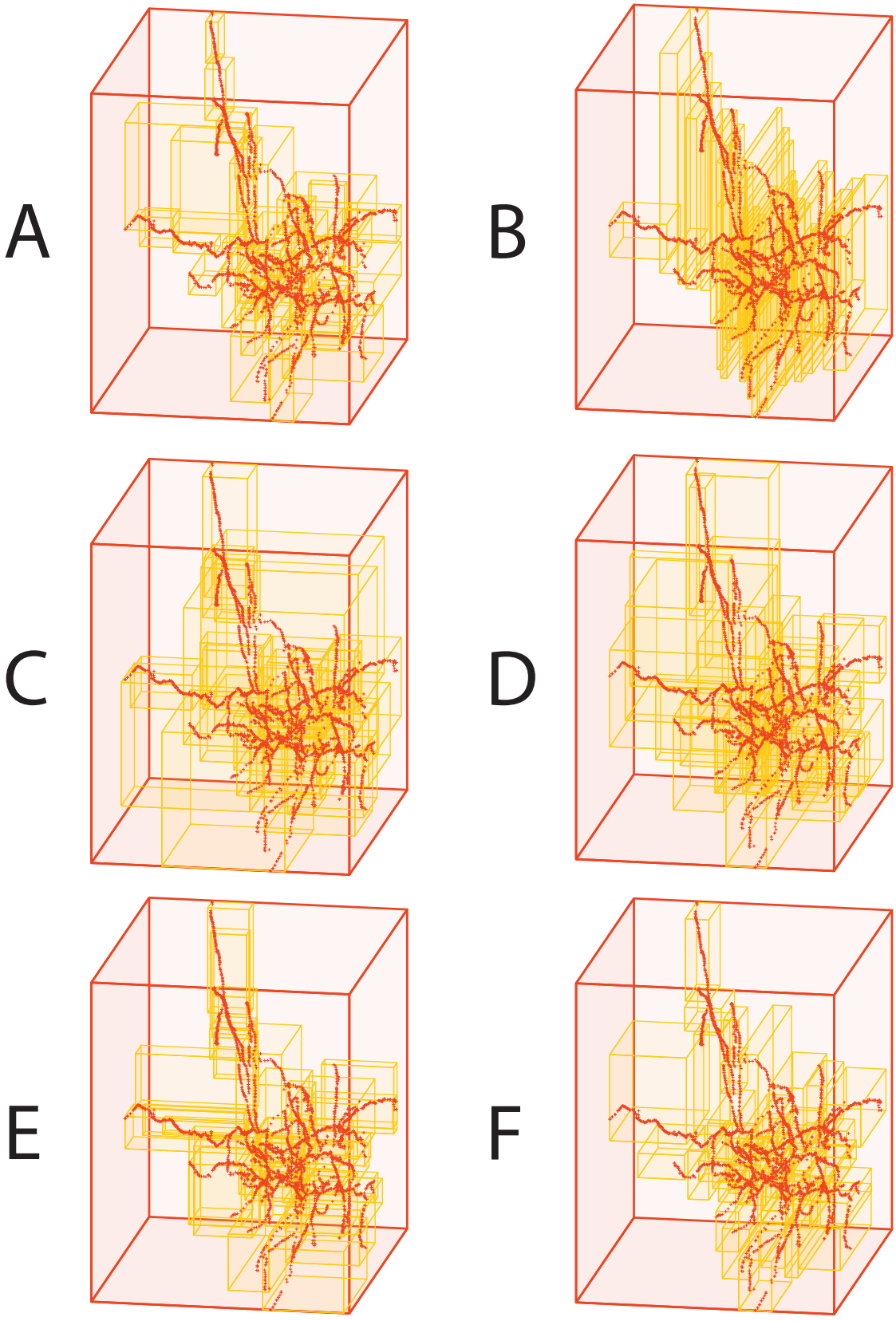


Figure 2.4: This figure depicts R-trees constructed with ELKI using different bulk loading approaches. In all pictures, the tree contains points describing a neuron from layer 1 of the neocortex of a young male rat. **A**: Repeated R*-tree insertion (no bulk loading). **B**: One-dimensional sorting. **C**: Hilbert sorting. **D**: Binary split sorting. **E**: STR. **F**: Adaptive STR.

Chapter 3

Methods

This chapter describes what methods we used to conduct our research experiments. To put it briefly, neuromorphological data is generated, R-trees are built, and range query sequences are run on these trees.

3.1 Data generation

The datasets used for benchmarking were constructed to resemble nervous tissue (excluding glial cells) from the human neocortex. To generate the datasets, we used a set of reconstructed pyramidal cells from layers two and three of the human neocortex, provided through `http://neuromorpho.org/` by the Allen institute [25]. The majority of the cells in the human neocortex are pyramidal cells [26], which makes these cells a good fit for use in the benchmark datasets. The neuron density in the six layers of the neocortex range from approximately 5,000 to 50,000 neurons/mm^3 [26], and these figures were used in the design of the different datasets as described below.

Ten datasets (A-J) within volumes of $300\,\mu\mathrm{m} \times 600\,\mu\mathrm{m} \times 300\,\mu\mathrm{m}$ were generated with neuron densities ranging (linearly spaced) from 5,000 to 50,000 neurons/mm^3. The sizes of the generated datasets on disk ranged increasingly from 54 to 551 MB. Each neuron was represented as a set of points (i.e. not MBRs), due to the scope of the project. These neurons where randomly rotated around their y-axis whereafter they were randomly scaled between a factor of 0.8 and 1.2. This method was chosen as it is similar to the methods used to generate datasets in [11] (see *Supplemental Procedures*) and thus as close as possible (given the scope of this project) to a realistic use case. The neurons were then randomly translated in the volume using a uniform distribution, such that the center of each neuron was placed within the volume. The model was

then trimmed so that all points residing outside the volume were removed. This saved memory and allowed for experiments with higher densities, but in turn removed parts of the model. An overview of the datasets is given in table 3.1.

The data generation was conducted with the help of Python scripts which were written using a neuron morphology processing package[1] by Alexander Kozlov. The source code can be found in Appendix A.

Dataset	**Volume** (mm^3)	**Neuron density** ($\mathrm{n/mm}^3$)	**No. of neurons**	**No. of points**
A	$0.3 \times 0.6 \times 0.3$	5,000	270	1,550,947
B	$0.3 \times 0.6 \times 0.3$	10,000	540	3,171,449
C	$0.3 \times 0.6 \times 0.3$	15,000	810	4,698,664
D	$0.3 \times 0.6 \times 0.3$	20,000	1,080	6,128,490
E	$0.3 \times 0.6 \times 0.3$	25,000	1,350	7,767,099
F	$0.3 \times 0.6 \times 0.3$	30,000	1,620	9,224,626
G	$0.3 \times 0.6 \times 0.3$	35,000	1,890	10,770,829
H	$0.3 \times 0.6 \times 0.3$	40,000	2,160	12,434,364
I	$0.3 \times 0.6 \times 0.3$	45,000	2,430	13,697,261
J	$0.3 \times 0.6 \times 0.3$	50,000	2,700	15,306,216

Table 3.1: Numerical description of all datasets used for the experiments.

3.2 Performance measurements

We compared the performance of the different data structures in terms of three measurements:

1. The time required to build the index.
2. The number of page reads required for the sequence of queries.
3. The total execution time of the sequence of queries.

3.3 Experimental methodology

All of the measurements mentioned above were obtained and easily compiled through features available in ELKI. ELKI was chosen because of its fair and

[1] `https://github.com/a1eko/morphon`. Accessed 2020-05-27.

solid benchmark capabilities and because of its readily available algorithms which made it easy to use for answering the research question reliably.

For the experiments performed, a range query benchmark which is built into ELKI was used. This evaluates a sequence of range queries on a dataset [22]. ELKI defines a range query as a search for all points in the dataset within some given distance from another point using a particular distance function[2]. In the experiments, two three-dimensional distance functions were used: the standard Euclidean norm given by $d(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{3}(\vec{x}_i - \vec{y}_i)^2}$ as well as the Chebyshev distance or maximum distance: $d(\vec{x}, \vec{y}) = \max_{i\in\{1,2,3\}} |\vec{x}_i - \vec{y}_i|$. This is because the Euclidean distance produces spherical queries, whereas the Chebyshev distance produces cubical queries, all centered at the query points.

The experiments consisted of two different tests that were both run on the 10 datasets described in section 3.1. These tests were largely based on the use cases described in [1], with one for small spherical queries and one for larger cubical queries. For each test and dataset, 20,000 unique points were randomly selected to be used as the query centers of the corresponding range query sequence.

The first test was performed as follows. For all construction techniques and datasets, a file containing all the points of the dataset was loaded into ELKI and used to build an R-tree using the corresponding technique. Additionally, a file was loaded into ELKI containing the test queries, each written as a sequence of coordinates followed by the query radius, $5.0\,\mu\mathrm{m}$. Next, the range query benchmarking algorithm of ELKI was run using spherical (Euclidean distance) range queries. For each of the 20,000 query points for the corresponding dataset, a range query was made. The second test was performed in the same way, except with cubical (Chebyshev distance) shape and a radius (half cube side) of $20.0\,\mu\mathrm{m}$. The tests are referred to in the remainder of this report as test case S (small) and test case L (large), respectively.

The index build times of the indices were obtained when running test case S. Since the queries used have nothing to do with index build time, only the index build times of test case S were measured.

[2] `https://elki-project.github.io/releases/current/doc/de/lmu/ifi/dbs/elki/database/query/range/RangeQuery.html`. Accessed 2020-05-27.

3.4 Testing environment

The experiments were run on a computer with the following specs:

- **Operating system:** Ubuntu 18.04.4 LTS 64-bit
- **CPU:** Intel Core i7-2600 3.4GHz Quad Core (8 threads)[3]
- **RAM:** 15.6 GiB
- **Storage:** A hard disk drive (HDD)

All tests were run in the main memory of the computer, no page swapping was performed.

[3] `https://ark.intel.com/content/www/us/en/ark/products/52213/intel-core-i7-2600-processor-8m-cache-up-to-3-80-ghz.html`. Accessed 2020-05-27.

Chapter 4

Results

In this chapter, the results of the two benchmarks run on each of the different datasets are presented in figures. The diagrams are grouped based on the different measurements: index build time, page reads for the query sequence and execution time for the query sequence. For each construction technique, the color used is consistent across all figures. All figures appear in pairs displaying the same data, but with one figure including methods that performed much worse than the others. This makes it possible to differentiate between and compare all methods.

4.1 Index build times

In figures 4.1-4.2 we can see that the regular R*-tree (no bulk loading strategy) performed worst of all strategies: its building time was about 14 times longer than the second worst strategy for the largest dataset. Peano curve and Z-curve performed similarly and outperformed the rest.

Figure 4.1: Index build times in test case S.

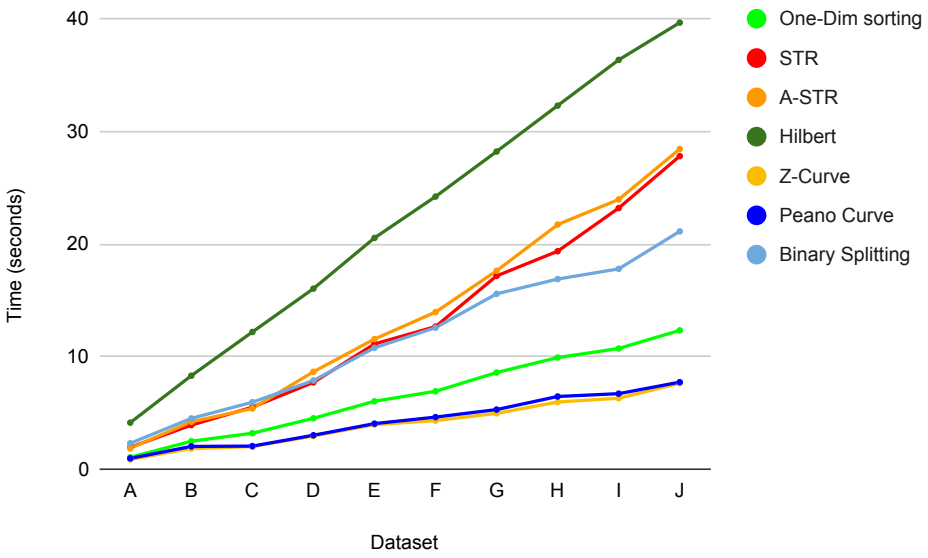
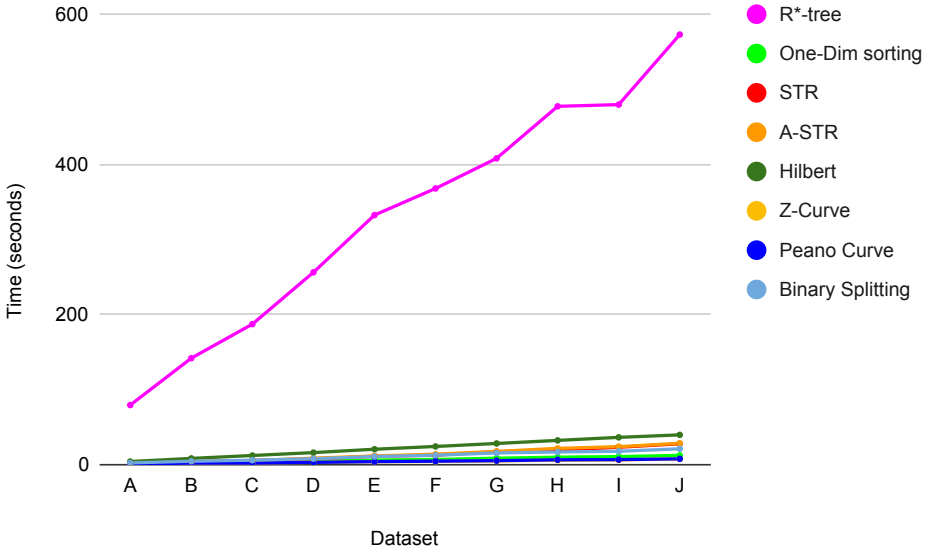


Figure 4.2: Index build times in test case S.



4.2 Page reads

In figures 4.3-4.4 we can distinguish three groups of similarly performing methods. Firstly, the R*-tree (no bulk loading strategy) and one-dimensional sorting bulk loading performed worst with the most page reads for small queries. The Z-curve and binary splitting packing approaches performed similarly to each other, but not the best overall. The third similarly performing group is the remaining methods, which performed best. The best performing method was Adaptive STR.

Figure 4.3: Page reads in test case S.

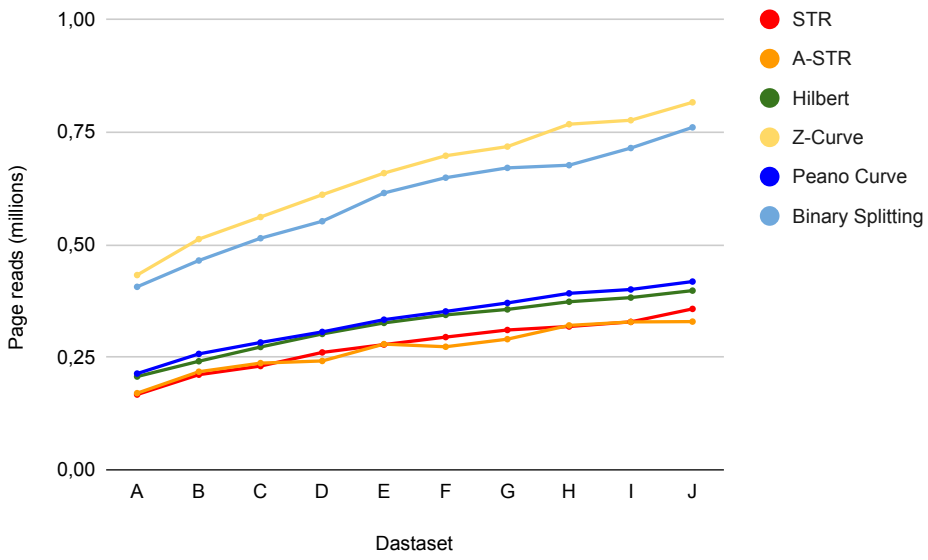
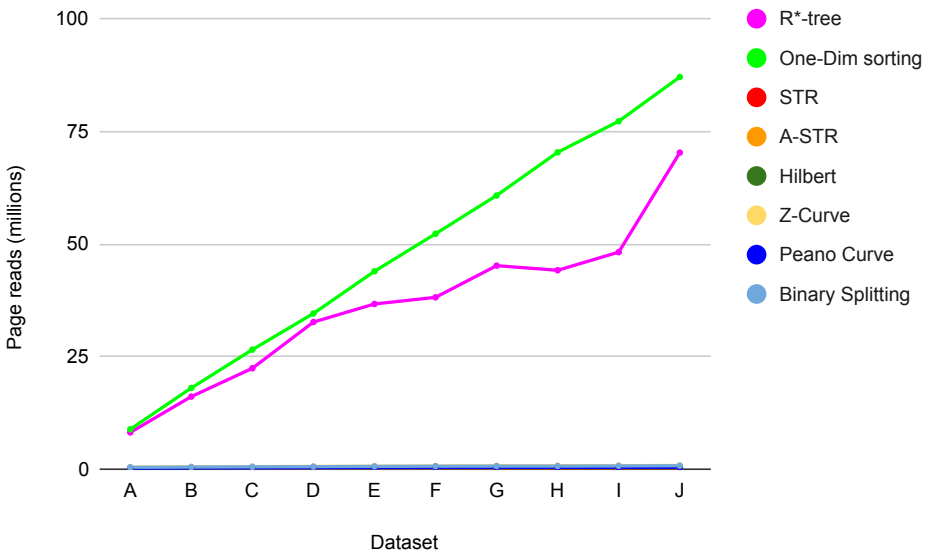


Figure 4.4: Page reads in test case S.



For larger, cubical queries, we see a similar pattern as for small queries with three groups. However, the difference in performance between R*-tree insertion and one-dimensional sorting bulk loading is larger for bigger queries. Additionally, the difference in performance between Z-curve packing and binary splitting packing versus the remaining methods is smaller than for smaller queries. This can be seen in figures 4.5-4.6. Again, the best performing method is Adaptive STR.

Figure 4.5: Page reads in test case L.

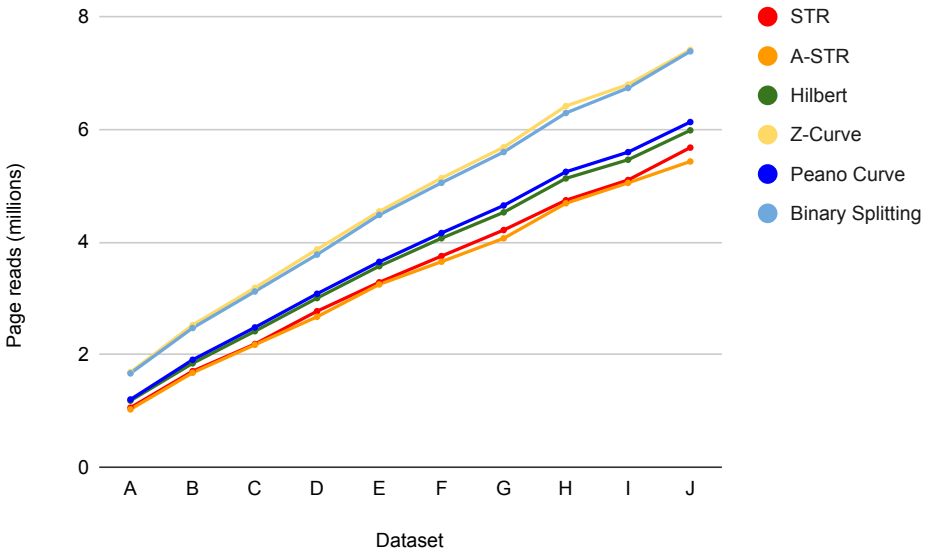
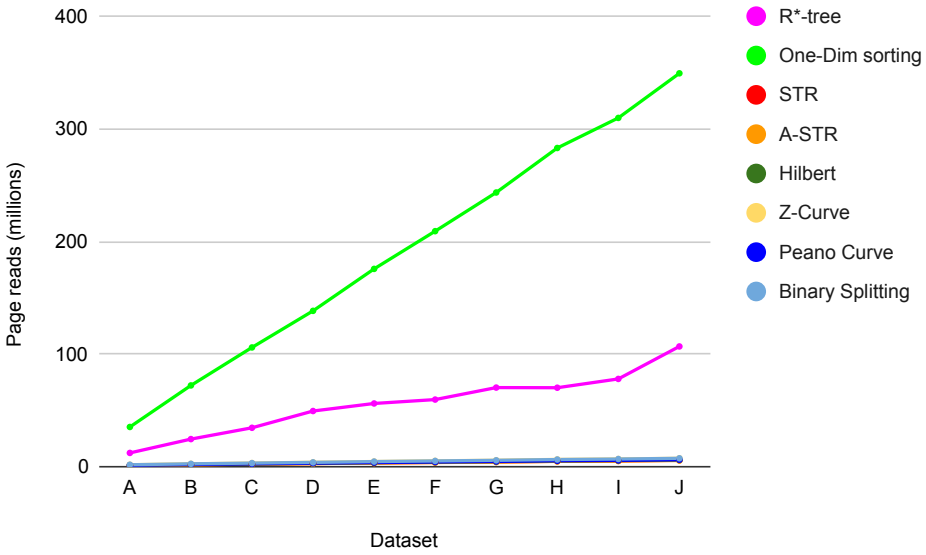


Figure 4.6: Page reads in test case L.



4.3 Execution times

As we saw in section 4.2 there are three different performing groups. R*-tree and one-dimensional sorting bulk loading belong to the worst performing one. Z-curve and binary split belong to the mid performing group and the rest belong to the best performing group. There is a larger performance difference between the mid performing group and the best performing group for smaller queries than for large ones, and the difference between R*-tree insertion and one-dimensional sorting packing is greater for larger queries.

A clear observation is that all methods but Hilbert and R*-tree dipped in execution time in dataset H, which can be seen in figure 4.7 and 4.8.

Figure 4.7: Execution times in test case S.

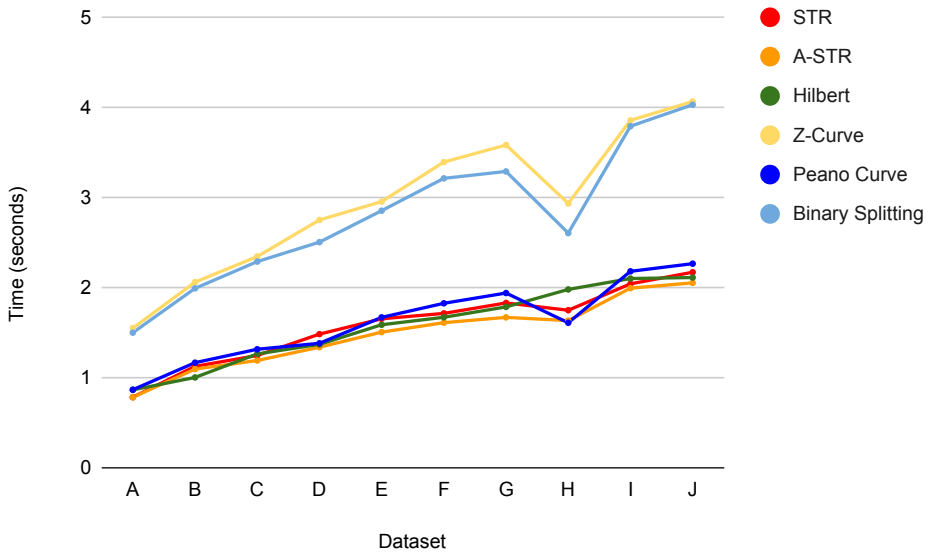
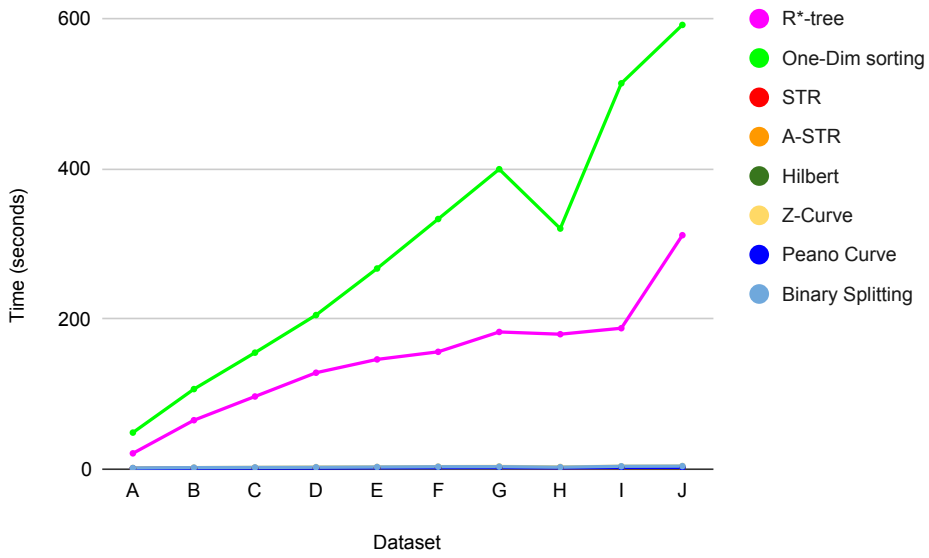


Figure 4.8: Execution times in test case S.



In figures 4.9-4.10, we see the execution times for test case L. For these larger queries, the execution time appears more predictable for standard R*-trees and Hilbert bulk loading. The curves corresponding to other methods have a more jagged shape, and in several cases the execution time decreases in spite of increased density in the dataset used. One-dimensional sorting performs worst, clearly outperformed by the R*-tree, which in turn is outperformed by all other methods. The relative differences in performance between the remaining methods is small.

Figure 4.9: Execution times in test case L.

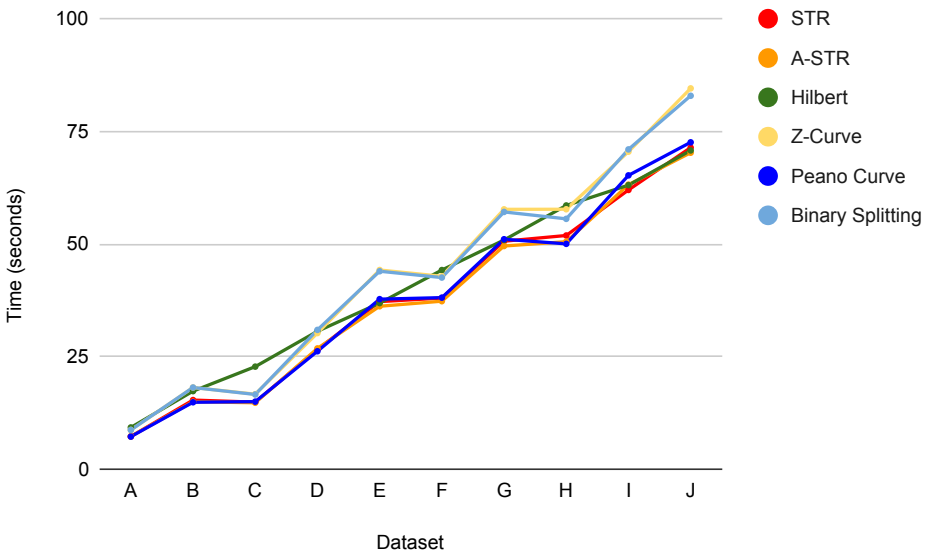
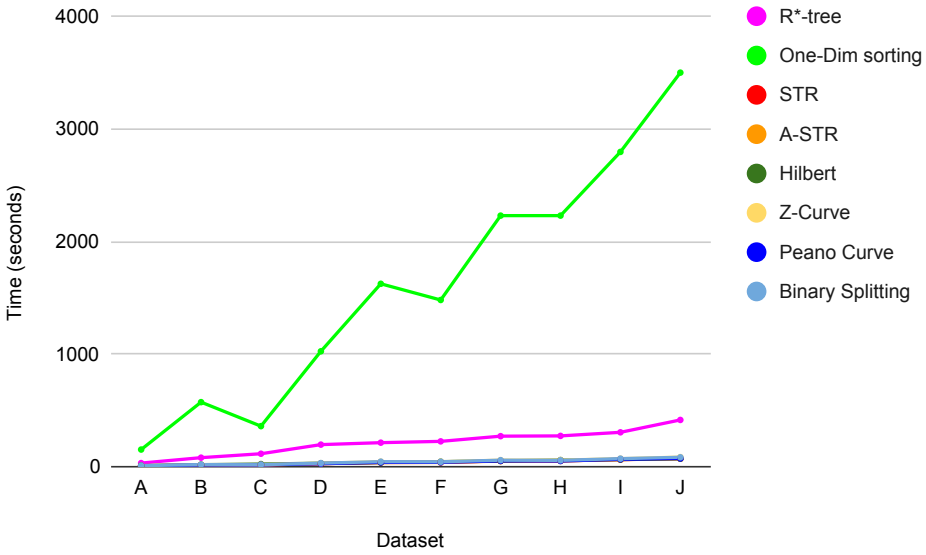


Figure 4.10: Execution times in test case L.



Chapter 5

Discussion

5.1 Results

One of the most conclusive results of the experiments made was that the R*-tree and the one-dimensional sorting bulk loading strategies were massively outperformed by other methods when it comes to query time. For one-dimensional sorting, this was expected. Because the dataset used is very dense, containing millions of elements in a small volume, the one-dimensional sorting approach will split the dataset into very thin slices along the x-axis. Thus, when queries are made that are wider than such slices, a very large amount of slices need to be searched despite the fact that most of the points within these slices are not in the result. For the standard R*-tree built with repeated insertion, we cannot propose an equally simple explanation. It is however clear that the query performance is much worse for this method than for bulk loading approaches.

As one gathers from figures 4.4 and 4.6, the performance of one-dimensional sorting deteriorates linearly with increasing query size. The number of page reads for one-dimensional sorting in test case L is consistently around 4 times the number of page reads for one-dimensional sorting in test case S across all datasets. This is because an increase by a factor of 4 in query radius results in approximately 4 times more slices being fetched (i.e. pages being read) when processing a query.

The volume of each query made in test case L is approximately 122 times that of the queries in test case S. Interestingly, for the standard R*-tree, the increase in the number of pages fetched per query between test case S and L was as low as a factor of about 1.3, as is also evident from figures 4.4 and 4.6. This suggests there is a large problem with overlap in the R*-trees for the given datasets. If there was not, the increase in pages fetched per query would

be closer to the increase in volume per query.

For the other methods, the increase in page reads between test case S and test case L was larger. For STR and A-STR, this increase was by a factor of between 10 and 20. This suggests the bulk-loading methods (save one-dimensional sorting) result in a much smaller overlap than R*-trees for these datasets.

Another interesting observation is that the difference between performance in the R*-tree and the bulk-loading approaches (again, ignoring one-dimensional sorting) seems to decrease with larger queries. As is stated by the authors of the R*-tree in [18], larger queries are generally less affected by geometrical optimization of the MBRs. This is likely the reason for this increase in similarity between the different methods between test case S and L. All in all, this suggests that the smaller the queries a neuroscientist is working with, the more important the choice of data structure becomes.

For all of the range query experiments done, the Z-curve and binary splitting sorting approaches performed similarly. This could be due to the fact that the Z-curve splits the space into half repeatedly, whereas the binary splitting sorting splits the data itself in half repeatedly.

One unexpected result was that the index build time of one-dimensional sorting was much higher than other approaches despite the very simple algorithm. This could be explained by the fact that spatial sorting algorithms which better preserve spatial locality may require fewer swaps in the data given its initial structure (SWC).

Using our results we can estimate the range query performance on a $1\,\mathrm{mm}^3$ cube, which is 18 times bigger than the volume we performed our tests on. The execution time to perform 20,000 queries on this volume bulk loaded with the Adaptive STR algorithm would be about 11 minutes, assuming linear scaling.

5.2 Previous research

The results obtained clearly illustrate the very large difference in index build time between repeated R*-tree insertion and bulk loading. As several authors point out [21] [27], one of the benefits of bulk loading spatial indices is the reduced build time compared to repeated dynamic insertion. This was thus expected.

In a previous study, STR and Hilbert bulk loading (among other methods) were compared for range queries on neuromorphological data [1]. In that study, it is concluded that the performance difference between STR and Hilbert

bulk loaded R-trees is very similar to the performance difference measured in this study.

In the paper where STR bulk loading was first presented [21], STR is experimentally compared to Hilbert bulk loading and slightly outperforms it on most datasets. While that study does not test neuromorphological datasets, the difference in performance between these methods is roughly similar to that observed in this study.

One paper by Faloutsos et al. [23] investigates the use of Hilbert curves and another space-filling curve very similar to the Z-order curve (it is the Z-order curve but rotated 90 degrees counterclockwise). In this study, the authors conclude that the Hilbert curve is better than the Z-order curve for preserving spatial locality for range queries in 3 dimensions. These authors do not experiment with R-trees in particular, but rather with the curves themselves. However, our results also show that Hilbert bulk loading performs better than Z-order curve bulk loading, which is consistent with their results.

A somewhat unexpected result was that Peano curve bulk loading performed similarly to the Hilbert curve bulk loading approach. We have not managed to find any mentions of this particular curve in previous database research, but consider this an interesting find. While the Peano curve bulk loading approach is not as good as Hilbert curve bulk loading for range queries, the index build time for the Peano curve bulk loading was far smaller. This suggests it might be a good idea to use this curve for neuroscientific experiments if the index build time is very important.

5.3 Methodology

A conclusion drawn from the results is that the execution times are very unreliable: they oscillate a bit which was expected. The running time of a process in a computer varies depending on, to name a few, the operating system, background processes, cache hits and cache misses, which is well-known. In our case the oscillating execution times could also be caused by the warmup time of the JVM (Java Virtual Machine) since we are performing our benchmark tests in ELKI, which is written in Java. For every datapoint (color and dataset in our figures) we restart the JVM, which could explain why the execution times are different.

Due to the experimental setup, there is no way of knowing the expected variation in any of the results. While a very large number of queries were made, thus reducing the influence of random variations, the page reads and execution times were measured only for the entire query sequence and not for

individual queries. This makes it impossible to obtain a standard deviation for either of these performance measurements. For the page reads, the majority of the data structures (all but the R*-tree) followed predictable patterns suggesting a low standard deviation. For the execution times, however, the fluctuations in the curves were more apparent and the standard deviation was likely higher in these results.

The index build time measurements were only performed once per algorithm and density. A complete measurement of all algorithms for all densities would take about 3 hours (where R*-tree takes the longest time of about 2 hours). To measure the index build times several times would therefore take a lot of time (30 hours for 10 experiments), which is the reason we only performed these tests once. The results show that the performance ordering of the different methods remains stable, which suggests they are still useful.

What is surprising with our results however is the fact that all strategies (except for Hilbert and the R*-tree) dipped in execution time on dataset H in test case S. Why this is the case we can merely speculate about. It could have something to do with the dataset itself: perhaps the queries are constructed in such a way that they optimize cache-hits, which lowers the execution time? You would expect that if a time anomaly was to happen, it would be slower, not faster than usual. Hardware or software faults could probably therefore be ruled out. Perhaps we should run the tests on dataset H again (with the same data input) and see if we get rid of the anomaly or not. Then we could conclude whether it has something to do with the dataset or if something else is strange.

In contrast to the dip in execution times, the page reads do not dip in dataset H. The amount of page reads we have to perform affects the execution time: the more page reads, the longer the execution time. The number page reads should also be unrelated to the computer used, in contrast to execution time it is the same across computers. It is therefore odd to see that the execution time dips below the previous test when the dataset is more dense and the amount of page reads are higher than the previous dataset test. It is therefore relieving to see that the page reads does not dip in dataset H. Perhaps we should have run the same tests several times to see if the amount of page reads are consistent.

ELKI proved to be a nice and easy tool to use for our experiment. It of course comes with the disadvantage of being written in Java, which would most likely be beaten by any decent C or C++ implementation. But the fact that it is a complete package with a lot of different algorithms written in the same language, with the same optimization level and using the same data structures ensures it provides fair benchmarking.[1] Fair benchmarking is very important

[1] `https://elki-project.github.io/benchmarking`. Accessed 2020-05-

for this kind of research to get reliable results.

The neurons in our datasets are represented as sets of points rather than sets of small MBRs. This is different from the models used in the only previous study we have identified on bulk loaded R-trees for neuromorphological data [1]. In that study, the authors use cylinders instead of points, so that each cylinder is represented by a small MBR. While comparing that to our own study, we did not find any reason to believe using points would lead to very different results. But, it is a simplification and may have had an impact on the results.

27.

Chapter 6

Conclusions

From the results of the experiments made in this study, it is possible to answer the research question as initially stated in section 1.4:

"*How do R-trees built using different methods perform with respect to index build time, the number of page reads when executing a sequence of range queries, and the execution time of such a sequence, on neuromorphological data?*"

Starting with the index build times, we see that the ordinary R*-tree (built by repeated insertion) performs much worse than the packed trees. For the packed trees, the trees built using Peano curve and Z-order curve packing are the quickest, whereas the Hilbert packed tree takes approximately 5 times as long to build as the Peano packed trees. In between these, we find the variations of STR, binary splitting, and one-dimensional sorting.

For the query page reads, the results were found to be slightly dependent on query size. The ordering in performance from best to worst was the same for both small and larger queries, though the difference in performance between different methods was smaller for larger queries. The R*-tree and one-dimensional sorting packing performed much worse than the other methods. Out of these other methods, adaptive STR and STR performed the best, with adaptive STR requiring slightly less page reads than STR. The Hilbert curve and Peano curve packing methods performed similarly, and slightly worse than the STR variants. This can be considered somewhat surprising given how little the Peano curve is mentioned in the literature, and the low index build time of the Peano packed trees. The Z-order curve and binary splitting packing methods also performed similarly, and significantly worse than the other packing methods (except for one-dimensional sorting), especially for smaller queries.

Lastly, for execution time, the results obtained are generally similar to those

obtained for page reads, although there appears to be much more variance in the results. As is stated in the discussion section, one can conclude that the number of page reads are likely a better indicator of data structure performance than the hardware-dependent execution times. Ultimately execution time is what matters though.

All in all, the study shows that out of the methods tested, the Peano curve and Z-order curve packing methods lead to the lowest index build time. For range query page reads and execution times, adaptive STR and STR perform the best.

For query performance, none of the tested approaches greatly outperform ordinary STR packing, which in turn is not very close to state-of-the-art methods [1], and so it is highly unlikely that the methods tested in this study will be widely adopted within neuroscience. However, this study clarifies the performance differences between existing bulk loading techniques as well as the performance difference between dynamic R*-tree insertion and bulk loading methods. This could help future researchers rule out options like the R*-tree or one-dimensional sorting packing for neuromorphological applications. This study also provides readers with an idea of the relative build times of the different construction techniques, which might be useful in neuroscientific applications where indices must be built quickly.

In future work, more methods could be used for similar benchmarks. One could try using different data structures, including M-trees and possibly X-trees and comparing these to different R-tree variants. Another approach would be to try and compare experimental approaches, including FLAT (proposed in a paper commonly cited in this thesis) [1], spatial indices making use of machine learning methods as discussed in [28] as well as a novel R-tree packing approach potentially outperforming state-of-the-art methods discussed in [29]. The last one could possibly even be implemented in ELKI.

Bibliography

[1] Farhan Tauheed et al. “Speeding Up Range Queries For Brain Simulations”. In: (2011).

[2] Suzana Herculano-Houzel. “The human brain in numbers: A linearly scaled-up primate brain”. In: *Frontiers in Human Neuroscience* 3.NOV (2009), pp. 1–11. ISSN: 16625161. DOI: 10.3389/neuro.09.031.2009.

[3] World Health Organization. “Estimates and projections”. In: *Neurological disorders: public health challenges*. 2006. Chap. 2. ISBN: 978-9241563369.

[4] Michael W. Reimann et al. “An algorithm to predict the connectome of neural microcircuits”. In: *Frontiers in Computational Neuroscience* 9.OCT (Oct. 2015). ISSN: 16625188. DOI: 10.3389/fncom.2015.00120.

[5] Carolina Westlin and Andreas Mårtensson. “An estimation of scalability when using a k-d tree as the data structure for neuron touch detection”. PhD thesis. 2018.

[6] Jenny Norelius and Antonello Tacchi Mondaca. “Evaluating data structures for range queries in brain simulations”. PhD thesis. 2018.

[7] Markus Adamsson and Aleksandar Vorkapic. “A comparison study of Kd-tree, Vp-tree and Octreefor storing neuronal morphology data with respect to performance”. PhD thesis. 2016.

[8] Charles Watson, Matthew Kirkcaldie, and George Paxinos. “Nerve cells and synapses”. In: *The Brain*. Elsevier, 2010. Chap. 1, pp. 1–10. ISBN: 978-0-12-373889-9. DOI: https://doi.org/10.1016/C2009-0-01579-7.

[9] Charles Watson, Matthew Kirkcaldie, and George Paxinos. "Central nervous system basics–the brain and spinal cord". In: *The Brain*. Elsevier, 2010. Chap. 2, pp. 11–24. ISBN: 978-0-12-373889-9. DOI: https://doi.org/10.1016/C2009-0-01579-7.

[10] Charles Watson, Matthew Kirkcaldie, and George Paxinos. "The human cerebral cortex". In: *The Brain*. Elsevier, 2010. Chap. 7, pp. 97–108. ISBN: 978-0-12-373889-9. DOI: https://doi.org/10.1016/C2009-0-01579-7.

[11] Henry Markram et al. "Reconstruction and Simulation of Neocortical Microcircuitry". In: *Cell* 163.2 (Oct. 2015), pp. 456–492. ISSN: 10974172. DOI: 10.1016/j.cell.2015.09.029.

[12] Charles Watson, Matthew Kirkcaldie, and George Paxinos. "A map of the brain". In: *The Brain*. Elsevier, 2010. Chap. 3, pp. 25–42. ISBN: 978-0-12-373889-9. DOI: https://doi.org/10.1016/C2009-0-01579-7.

[13] Maryam Halavi et al. "Digital reconstructions of neuronal morphology: Three decades of research trends". In: *Frontiers in Neuroscience* 6.APR (2012), pp. 1–11. ISSN: 16624548. DOI: 10.3389/fnins.2012.00049.

[14] J. J. Johannes Hjorth et al. "The microcircuits of striatum in silico". In: *Proceedings of the National Academy of Sciences* (2020), p. 202000671. ISSN: 0027-8424. DOI: 10.1073/pnas.2000671117.

[15] R. C. Cannon et al. *An on-line archive of reconstructed hippocampal neurons*. Tech. rep. 1-2. 1998, pp. 49–54. DOI: 10.1016/S0165-0270(98)00091-0. URL: www.neuro.soton.ac.uk..

[16] Maryam Halavi et al. "NeuroMorpho.Org implementation of digital neuroscience: Dense coverage and integration with the NIF". In: *Neuroinformatics* 6.3 (2008), pp. 241–252. ISSN: 15392791. DOI: 10.1007/s12021-008-9030-1.

[17] Antonin Guttman. "R-Trees - A dynamic index structure for spatial searching". In: *ACM SIGMOD Record* June (1984), pp. 47–57. DOI: https://doi.org/10.1145/971697.602266.

[18] Norbert Beckmann et al. "The R*-tree: An efficient and Robust Access Method for Points and Rectangles+". In: *Acm* (1990), pp. 322–331.

[19] Yannis Manolopoulos et al. *R-Trees: Theory and Applications*. 1st ed. London: Springer-Verlag, 2006, pp. 35–48. ISBN: 978-1-85233-977-7. arXiv: arXiv:1011.1669v3.

[20] Nick Roussopoulos and Daniel Leifker. "Direct Spatial Search on Pictorial Databases Using Packed R-Trees". In: *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*. SIGMOD '85. New York, NY, USA: Association for Computing Machinery, 1985, pp. 17–31. ISBN: 0897911601. DOI: 10.1145/318898.318900. URL: https://doi-org.focus.lib.kth.se/10.1145/318898.318900.

[21] Scott Leutenegger, Mario Lopez, and Jeffrey Edgington. "STR: A Simple and Efficient Algorithm for R-Tree Packing". In: *Proc. VLDB Conf.* 1997, pp. 497–506. ISBN: 0-8186-7807-0. DOI: 10.1109/ICDE.1997.582015.

[22] Erich Schubert and Arthur Zimek. "ELKI: A large open-source library for data analysis-ELKI Release 0.7. 5" Heidelberg"". In: *arXiv preprint arXiv:1902.03616* (2019).

[23] C Faloutsos and S Roseman. "Fractals for Secondary Key Retrieval". In: *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '89. New York, NY, USA: Association for Computing Machinery, 1989, pp. 247–252. ISBN: 0897913086. DOI: 10.1145/73721.73746. URL: https://doi.org/10.1145/73721.73746.

[24] Lars Arge et al. "The Priority R-tree: A practically efficient and worst-case optimal R-tree". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2004), pp. 347–358. ISSN: 07308078.

[25] Christof Koch and Allan Jones. "Big Science, Team Science, and Open Science for Neuroscience". In: *Neuron* 92.3 (2016), pp. 612–616. ISSN: 10974199. DOI: 10.1016/j.neuron.2016.10.019. URL: http://dx.doi.org/10.1016/j.neuron.2016.10.019.

[26] Javier DeFelipe, Henry Markram, and Kathleen S. Rockland. *The neocortical column*. June 2012. DOI: 10.3389/fnana.2012.00022.

[27] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. “Improving the query performance of high-dimensional index structures by bulk load operations”. In: *Advances in Database Technology — EDBT’98*. Ed. by Hans-Jörg Schek et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 216–230. ISBN: 978-3-540-69709-1.

[28] Haixin Wang et al. “Learned index for spatial queries”. In: *Proceedings - IEEE International Conference on Mobile Data Management*. Vol. 2019-June. Institute of Electrical and Electronics Engineers Inc., June 2019, pp. 569–574. ISBN: 9781728133638. DOI: 10.1109/MDM.2019.00121.

[29] Jianzhong Qi et al. “Theoretically optimal and empirically efficient rtrees with strong parallelizability”. In: *Proceedings of the VLDB Endowment* 11.5 (2018), pp. 621–634. ISSN: 21508097. DOI: 10.1145/3177732.3177738.

Appendix A

Source code

The source code is also available at `https://github.com/elmaxe/kexjobb`

dataBuilder.py

```
"""
This script generates neuromorphological data in the form of points. It needs to
be run from the directory in which it is located (i.e., using 'python3 dataBuilder.py')
"""
import random as r
import functions as f
import sys
from os import listdir
from os.path import isfile, join
from morphon import Morph

# See Kozlov's email (he refers to Markram 2015)
LOW_SCALE = 0.8
HIGH_SCALE = 1.2

# Path where data is located
INPUT_FOLDER_PATH = sys.argv[1]
# Folder where output is saved, MUST NOT END WITH A SLASH /
OUTPUT_FOLDER_FILE_PATH = sys.argv[2]
# Length in micrometers
X_LENGTH = float(sys.argv[3])
Y_LENGTH = float(sys.argv[4])
Z_LENGTH = float(sys.argv[5])
# Neurons per cubic millimeter
```

```
25 DENSITY = float(sys.argv[6])
26
27 # Function applied to every copied neuron.
28 # All neurons are origin-centered, so new pos
     obtained by randomly rotating, scaling, and
     translating.
29 def processSingleMorph(m, id):
30   fp = OUTPUT_FOLDER_FILE_PATH+("/data-%d.txt"%
     DENSITY)
31   f.randomlyRotate(m)
32   f.randomlyScale(m, LOW_SCALE, HIGH_SCALE)
33   f.randomlyTranslate(m, X_LENGTH, Y_LENGTH,
     Z_LENGTH)
34   f.appendMorphToFileTrimmed(m, id, fp, 0, X_LENGTH
     , 0, Y_LENGTH, 0, Z_LENGTH)
35
36 # Get all SWC files in a folder and convert them to
      list of Morph objects
37 print("Using files: ", ", ".join([join(
     INPUT_FOLDER_PATH, filepath) for filepath in
     listdir(INPUT_FOLDER_PATH) if isfile(join(
     INPUT_FOLDER_PATH, filepath)) and filepath.
     endswith(".swc")]))
38 print("Building Morph objects from files... ", end=
     "")
39 inputMorphs = [f.morphFromFile(join(
     INPUT_FOLDER_PATH, filepath)) for filepath in
     listdir(INPUT_FOLDER_PATH) if isfile(join(
     INPUT_FOLDER_PATH, filepath)) and filepath.
     endswith(".swc")]
40 print(" done.")
41
42 # volume is converted to mm3 by 10^-9
     multiplication, then multiplied by density, to
     produce number of neurons in model
43 n = int(DENSITY*X_LENGTH*Y_LENGTH*Z_LENGTH
     /(10.0**9))
44
45 for i in range(n):
46   m = r.choice(inputMorphs).copy()
47   processSingleMorph(m, i)
48   if(i > 0 and (i+1) % 100 == 0):
49     print("%d/%d neurons placed" % (i+1, n))
50 print("All neurons placed.")
```

functions.py

```
1 import re
```

```
import math
from morphon import Morph, Nodes
import random as r

"""
Randomly translates a morphon Morph placing the center of the neuron inside the cuboid
defined by x in [0, xLen), y in [0, yLen), z in [0, zLen).
"""
def randomlyTranslate(m, xLen, yLen, zLen):
  xd, yd, zd = r.random()*xLen, r.random()*yLen, r.random()*zLen
  m.translate((xd, yd, zd))

# Takes a filepath, returns morph object
def morphFromFile(filepath):
  m = Morph()
  m.load(filepath)
  return m

"""
Takes a list of Morph objects and converts them to a single string
of the form 'x y z label' where label is 'n'+the index of the neuron.
"""
def morphListToStringList(morphs):
  return [morphToStrings(m, id) for (i, m) in enumerate(morphs)]

"""
Randomly rotates a morphon Morph object around the y-axis (0, 1, 0) with uniform distribution of rotation angles.
Adapted from Kozlov's sample code.
"""
def randomlyRotate(m):
  axis = (0, 1, 0) # the y-axis, as is appropriate for the Allen institute human pyramidal cells
  angle = math.pi*2.0*r.random() # value in range [0, 2*math.pi)
  m.rotate(axis, angle)

"""
Randomly scales a neuron by a random factor in the range [lowScale, highScale), uniform
```

```
    distribution
"""
def randomlyScale(m, lowScale, highScale):
  factor = r.uniform(lowScale, highScale)
  m.scale(factor)

"""
Converts a morphon Morph object into a list of strings of the form
'x y z label', where label is an n followed immediately by the id.
Implementation based on the save function in morphon/swc. If xMin, xMax, etc
are specified, points not satisfying these constraints are discarded.
Note: Newlines at the end of each string.
"""
def morphToStrings(m, id, xMin=-math.inf, xMax=math.inf, yMin=-math.inf, yMax=math.inf, zMin=-math.inf, zMax=math.inf):
  data = []
  for item in m.traverse():
    (t, (x, y, z), r) = m.value(item)
    if(xMin <= x and x <= xMax and yMin <= y and y <= yMax and zMin <= z and z <= zMax):
      data.append("%f %f %f n%d\n" % (x, y, z, id))
  return data

# Returns a random tuple of (x, y, z) present in a morphology
def randomPointInMorph(m):
  item = r.choice([i for i in m.traverse()])
  (t, (x, y, z), radius) = m.value(item)
  return (x, y, z)

"""
Writes a morph to a file
"""
def appendMorphToFile(m, id, filepath):
  f = open(filepath, "a")
  f.write("".join(morphToStrings(m, id)))

"""
Writes (appends) a morph to a file s.t. only points within the
specified volume are actually included in the file. Other points
```

```
are discarded.
"""
def appendMorphToFileTrimmed(m, id, filepath, xMin,
    xMax, yMin, yMax, zMin, zMax):
  f = open(filepath, "a")
  f.write("".join(morphToStrings(m, id, xMin, xMax,
    yMin, yMax, zMin, zMax)))
```

elkinoigui.sh

```
#!/bin/sh
exec java -Xmx12G -Xms12G -cp "elki/elki-0.7.5.jar:
   elki/*:dependency/*" de.lmu.ifi.dbs.elki.
   application.KDDCLIApplication "$@"
```

experiment.sh

```
#!/bin/bash
# $1 Path to data.
# $2 Path to query points.
# $3 Distance function: "maximum" or "euclid".
# $4 Folder to save output files to.
if [ "$3" == "maximum" ]; then
    dist="minkowski.MaximumDistanceFunction"
else
    dist="minkowski.EuclideanDistanceFunction"
fi

strategyIndices=( str astr onedim binsplit hilbert
   peano zcurve nobulk)

strategies[0]="-spatial.bulkstrategy
   SortTileRecursiveBulkSplit"
strategies[1]="-spatial.bulkstrategy
   AdaptiveSortTileRecursiveBulkSplit"
strategies[2]="-spatial.bulkstrategy
   OneDimSortBulkSplit"
strategies[3]="-spatial.bulkstrategy
   SpatialSortBulkSplit -rtree.bulk.spatial-sort
   BinarySplitSpatialSorter"
strategies[4]="-spatial.bulkstrategy
   SpatialSortBulkSplit -rtree.bulk.spatial-sort
   HilbertSpatialSorter"
strategies[5]="-spatial.bulkstrategy
   SpatialSortBulkSplit -rtree.bulk.spatial-sort
   PeanoSpatialSorter"
strategies[6]="-spatial.bulkstrategy
   SpatialSortBulkSplit -rtree.bulk.spatial-sort
   ZCurveSpatialSorter"
```

```
strategies[7]=" "

mkdir -p $4

for ((i = 0; i < ${#strategies[@]}; i++))
do
    echo "Starting ${strategyIndices[$i]} $3"
    ./elkinogui.sh -dbc.in $1 -db.index tree.
   spatial.rstarvariants.rstar.RStarTreeFactory -
   pagefile.pagesize 4096 ${strategies[$i]} -time -
   algorithm benchmark.RangeQueryBenchmarkAlgorithm
    -algorithm.distancefunction $dist -rangebench.
   query FileBasedDatabaseConnection -dbc.in $2 -
   evaluator NoAutomaticEvaluation -resulthandler
   ResultWriter | grep --invert-match '#' > $4/${
   strategyIndices[$i]}.txt
    echo "Finshed ${strategyIndices[$i]} $3"
done
```

all_densities_experiment.sh

```
#!/bin/bash
# $1 Path to root directory of data. MUST NOT END
   WITH A SLASH.
#
   ------------------------------------------------------------

    # The directories and files must be structured
   and named in the following way
    # root
    #   5000
    #       data-5000.txt
    #       queries-5000-euclid.txt
    #       queries-5000-cheb.txt
    #   10000
    #       data-10000.txt
    #       queries-10000-euclid.txt
    #       queries-10000-cheb.txt
    #   15000
    #       .
    #       .
    #       .
    #   35000
    #       ...
#
   ------------------------------------------------------------

# $2 Path to output folder, doesn't need to exist
```

```
   beforehand. MUST NOT END WITH A SLASH.

densities=( 5000 10000 15000 20000 25000 30000
   35000 40000 45000 50000 )

for d in "${densities[@]}"
do
    echo "Starting experiments for density of $d"
    ./experiment.sh $1/$d/data-$d.txt $1/$d/queries
   -$d-euclid.txt euclid $2/$d/euclid
    ./experiment.sh $1/$d/data-$d.txt $1/$d/queries
   -$d-cheb.txt maximum $2/$d/cheb/
    echo "Finished experiments for density of $d"
done
```

generate_queries.sh

```
#!/bin/bash
# Prints queries to stdout
#
# $1 Path to root directory of data. MUST NOT END
   WITH A SLASH.
#
   ------------------------------------------------------------

    # The directories and files must be structured
   and named in the following way
    # root
    #   5000
    #       data-5000.txt
    #   10000
    #       data-10000.txt
    #   15000
    #       .
    #       .
    #       .
    #   35000
    #       ...
#
   ------------------------------------------------------------

# $2 Query range, euclid
# $3 Query range, chebyshev
# $4 Number of queries

densities=( 5000 10000 15000 20000 25000 30000
   35000 40000 45000 50000 )

```

```
25 for d in "${densities[@]}"
26 do
27     shuf $1/$d/data-$d.txt > temp.txt
28     head -$4 temp.txt > temp2.txt
29     rm temp.txt
30     awk -v var="$2" '$4=var' temp2.txt > $1/$d/
   queries-$d-euclid.txt
31     rm temp2.txt
32 done
33
34 for d in "${densities[@]}"
35 do
36     shuf $1/$d/data-$d.txt > temp.txt
37     head -$4 temp.txt > temp2.txt
38     rm temp.txt
39     awk -v var="$3" '$4=var' temp2.txt > $1/$d/
   queries-$d-cheb.txt
40     rm temp2.txt
41 done
```

TRITA -EECS-EX-2020:338