Friedrich-Alexander-Universität Erlangen-Nürnberg

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Master Thesis

Efficient Cross-Version Music Retrieval Using Dimensionality Reduction and Indexing Techiques

submitted by

Julian Brandner

submitted

November 14, 2019

Supervisor / Advisor

Prof. Dr. Meinard Müller
Frank Zalkow

Reviewers

Prof. Dr. Meinard Müller
Prof. Dr.-Ing. Andreas Maier

AUDIO LABS

International Audio Laboratories Erlangen
A joint institution of the
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and
the Fraunhofer-Institut für Integrierte Schaltungen IIS.

Fraunhofer
IIS

Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Erlangen, November 14, 2019

Julian Brandner

Abstract

Cross-version music retrieval is the task of finding all versions of the same piece of music as a given audio fragment. Earlier publications explained a robust method to overcome the pattern-recognition related obstacles. Using it on a potentially massive database is however algorithmically highly complex. In this thesis I will discuss various approaches to solve this problem, mainly R-trees and an approximation method called "hierarchical navigable small world graphs". There will also be reports of experiments on when to use those methods and how to optimize them.

Zusammenfassung

Digitale Audiodaten sind im 21. Jahrhundert von fundamentaler Bedeutung für die Musikwissenschaft. Datenbanken umfassen inzwischen große Sammlungen an Aufzeichnungen, Notensätzen und MIDI. Um eine derartige Fülle an Informationen noch sinnvoll nutzen zu können, werden effiziente Verfahren benötigt. Ein Anfrageszenario an eine solche Datenbank ist "cross version music retrieval". Hierbei sollen zu einem gegebenen Musikfragment Einträge in der Datenbank gefunden werden, die zu dem selben Stück gehören. Das Fragment kann dabei in Form von Audio-, MIDI-Daten oder ähnlichem vorliegen.
In früheren Veröffentlichungen wurde dieses Problem bereits angegangen. Ein etabliertes Verfahren ist es, sogenannte Schindeln aus der Datenbasis zu bilden. Eine Schindel betrachtet dabei einen kurzen Abschnitt eines Musikstücks (20s). Für jede Schindel wird der zeitliche Verlauf der 12 Grundtöne innerhalb ihres Abschnitts zu einem Featurevektor quantifiziert. Zwei so berechnete Schindeln werden als zusammengehörig betrachtet, wenn ihre Vektoren ähnlich sind. Ziel dieser Arbeit ist es, das Auffinden solcher ähnlicher Schindeln effizient zu lösen. Das Szenario wird dazu auf das bekannte Problem der k-Nearest-Neighbor-Search abgebildet. Für dieses existieren verschiedene Ansätze, von denen zwei näher betrachtet werden: R-Bäume und hierarchische Nachbarschaftsgraphen. Neben einer Erläuterung zur Funktionsweise der Algorithmen zeigen wir in nachfolgenden Experimenten ihre Laufzeit in diesem konkreten Szenario und schlagen Strategien vor, um sie effizient einzusetzen. Zusätzlich werden verschiedene Ansätze zur Featurereduktion zum Einsatz kommen und ihre jeweiligen Auswirkungen auf die Laufzeit der Anfragen gemessen und bewertet werden. Abschließend werden diese Ergebnisse zusammengefasst und es wird as Experimenten gefolgert, welche Methodik sich für welches Szenario eignet.

Contents

Part I

Introduction

Chapter 1

Introduction

Digital music data is widely available and databases collecting it comprise vast amounts of data. Managing and retrieving this content has to be automated to be feasible. For musicological research, the possibility of automatically retrieving music similar to a query is particularly interesting.
A meanwhile commercially available technique is “audio fingerprinting” (e.g. Cano et al. [1]). Given a short audio snippet, it aims to retrieve that exact recording from an audio database. The method is really well optimized and performs consistently on large databases. It has however a fundamental drawback: if the database does not contain that very recording, fingerprinting won’t find meaningful results at all.
In such cases a more general approach is needed. Musical features have to be robustly matched to decide, which database entries are relevant. This scenario is called “cross-version music retrieval” and is a topic of current research ([2, 3, 4, 5]). The potential application scenarios are wide ranging and can be relevant for research an consumer products alike. First of all, a use case like with fingerprinting is conceivable: Given a short audio recording the software identifies every recording of the same piece in the database. In this case, the recording doesn’t have to be present in the database, it could even be captured from a live performance. Furthermore cross-version music retrieval isn’t limited to audio data. It can be used to match audio and MIDI data in arbitrary combinations. Even sheet music (using Optical Music Recognition cf. [6, 7, 8]) could be used either as query or as the database.
Cross-version music retrieval is a complex pattern recognition task, but there are already known ways to solve this problem robustly, one of them will be outlined in the following chapter. However a bottleneck for such techniques is the algorithmic complexity of a query. As already stated, the databases can be very large and searching them has to be accelerated. Otherwise they can’t be made available to a large user basis. Examining the possibilities for fast cross-version music retrieval is the goal of this thesis.

The thesis will be structured as follows:

Chapter 2 summarizes earlier research and gives a general understanding of the approach. It is the basis for all optimizations and retrieval strategies in the remainder of the thesis.

In Chapter 3 we will formalize the problem. Then we will move on to some considerations about the different approaches and their desirable characteristics.

In Chapter 4 we will address the topic of KD-trees, a well known datastructure for multi-dimensional data. We will explain their functioning, and discuss their limitations.

In Chapter 5 we will move on to R-trees. R-trees are a conceptually similar structure to KD-trees and offer greater possibilities for optimization. The chapter contains a detailed description of the various ways to construct an R-tree and will focus on their respective characteristics.

Chapter 6 introduces the reader to the concept of hierarchical neighborhood graphs and explains their potential for cross version music retrieval. It includes a summary of the different parameters to consider when building these structures and explains the necessary algorithms.

Chapter 7 sets the methods for testing the above mentioned concepts. This includes a description of the precision measurement concepts and the used database.

For Chapter 8 we conducted various experiments on out own R-tree implementation and present the results. Special attention is payed to the camparison of the different approaches explained in Chapter 5. We will then introduce an optimization for R-trees and discuss when to use them.

Chapter 9 evaluates the graph structure from Chapter 6. It considers the tradeoffs between quality and speed and shows the impact of various parameters. It also conveys an understanding of the possibilities and drawbacks of the method.

Chapter 10 reviews two already existing implementations, that could be used for cross-version music retrieval. It is mainly intended to simply future research and show the potential of the already presented approaches.

Finally, in Chapter 11 we will recapitulate our findings and give concluding recommendations.

A main contribution of this thesis is the summary of the approaches applicable for efficient cross-version music retrieval. Therefor we explain their basics and explore their strengths and weaknesses.
We then conduct experiments on this topic, show the actual performance of the approaches, weight them against each other and give recommendations. The empirical observations are explained and used to develop an understanding on how to use the algorithms efficiently. Every approach has its drawbacks and possibilities and demonstrating them is a main goal of this thesis.

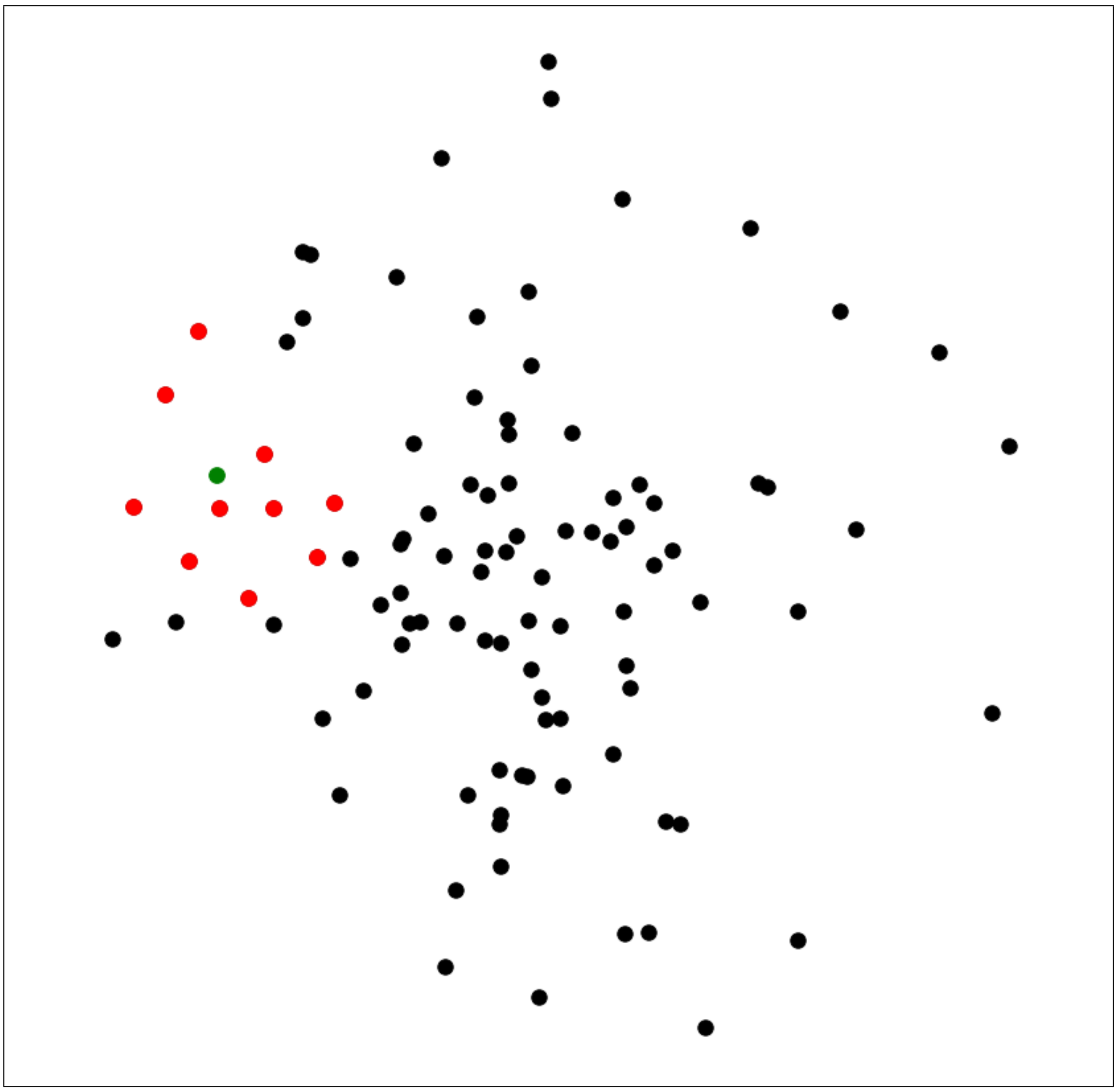
Chapter 2

Background

The basis of the methods presented in this paper is a shingling approach (cf. Casey et al. [2]) on cross-version music retrieval. It will be closely oriented along a method presented by Grosche and Müller [3]. Here the whole dataset gets converted into so-called shingles. A shingle represents a relatively long audio snippet (here 20 seconds) but gets generated at a significantly higher rate (here once per second). This leads to a massive overlap, hence the name shingle. The idea is to compare to shingles based on their chroma-features. Chroma features are computed from the frequency domain of the input and depict the energy of the 12 semitones. To increase robustness against temporal differences, instead of using chroma features directly, CENS features are calculated from them. This procedure leads to one 12-dimensional CENS feature per second, giving a total of 240-dimensions per shingle. The assumption is, that two shingles are similar, if the euclidean distance between their two feature-vectors is small. To classify an audio snippet one has to generate a shingle from it and retrieve the closest shingles from the database. In order to facilitate this a feature reduction can be performed. Aside from principal component analysis (PCA) Zalkow and Müller [9] used deep neural networks (DNN) as an option for this. However retrieving the most similar vectors from the database remains a time critical problem since the task becomes more complex for larger databases. This problem is called nearest neighbor search.

Part II

Approaches to K-Nearest Neighbor Search



Chapter 3

K-Nearest Neighbor Search in General

3.1 Problem Definition and Notation

K-nearest neighbor search (kNNS, kNN-search) is an optimization problem that arises in numerous fields of computer science. Generally speaking, its target is to retrieve the k points from a dataset D ($D \subset \mathbb{R}^d$) that are closest to a point p ($p \in \mathbb{R}^d$) measured in a distance metric m. The members of D and p have to be of the same dimensionality d. Theoretically, any m: $\mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ could be used to formulate the problem, but the following chapters will assume that m fulfills the triangle inequality. If m is the Euclidean distance $d(p, q) = ||p - q||_2$, the problem is also known as the post-office problem [10].

3.2 Approaches

There is an obvious naive approach to the problem: for each member $x \in D$ calculate $m(x, p)$, and return the k nearest candidates. This approach runs in complexity $O(|D|)$ (neglecting the complexity of m and the bookkeeping of k entries), and is thereby unfit for most database-applications. A multitude of different algorithms and data-structures have been supposed to deal with this problem. Yet it has to be stated, that there is no “perfect” solution that is applicable to large datasets and guarantees a bounded, sub-linear complexity for any query (in contrast to the 1-dimensional case, here e.g. a binary heap [11] can solve the problem in $O(log(|D|)k)$ for any p). A high dimensionality of the search space worsens this even further. Nevertheless in practice some approaches can achieve a good-enough performance. In practice this task can even be further relaxed since a mathematically correct result for each query isn't always required.

This thesis will discuss some of those data structures and apply them to the proposed problem. It will focus on the complexity for data retrieval, not on the construction cost of those structures.

Chapter 4

KD-Trees

K-dimensional-trees (KD-trees) are one of the most well known spatial indexing structures. First introduced by Bentley [12] they offer a comparatively simple way of handling multi-dimensional data. While initially designed for partial-match-queries (find all points, where certain dimensions are equal to the query) and range-queries (find all points within a certain range), an elegant algorithm for kNNS was later shown by Friedman et al. [13]. Note that while the “K” in the name stands for the dimensionality, we will continue to refer to it as d contrary to what is often found in literature. In our case “k” refers to the k nearest neighbors.

4.1 Structure

KD-trees are a special form of binary search trees. Each node (Algorithm 1) is either a leaf, or it references one or two children. On every level of the tree a fixed dimension is used to divide the dataset according to a threshold. A node stores this value and its (up to) two children. Thereby, each node divides the current subset into two parts, one above the threshold, one below it and each one gets handled by one of its children (cf. Figure 4.1). To get a balanced tree, naturally the median of the subset is chosen as threshold. Given the whole dataset is already known, a tree can recursively be constructed using algorithm 2. There are methods of inserting additional datapoints into existing trees [12].

4.2 K-Nearest-Neighbor-Search

The algorithm proposed by Friedman et al. [13] offers a simple yet efficient approach to kNNS. Each node contains the entries from a specific region. This region can be determined from its predecessors and forms an axis-aligned-bounding-box. These boxes can be recursively searched

Algorithm 1 KD-tree node

We will assume a KD-tree-node (**KD_Node**) to have the following members:

- *leaf_node* set for leafs, contains a single data-point
- *threshold* a floating-point-number
- *child_one* the first child node
- *child_two* the second child node

Note: The dimension to which the threshold of a node refers does not need to be stored in the node itself. It can be derived from the height at which the node sits in the final tree.

Algorithm 2 Construction of a KD-tree

$construct(dataset, 0, d)$ returns the root-node

```
function CONSTRUCT(data, dimension_current, dimension_dataset):
    KD_Node $result$
    if $length(data) = 1$ then
        $result.leaf\_node \leftarrow data[0]$
        return $result$
    end if
    $data \leftarrow sort\_along\_axis(data, dimension\_current)$
    $m \leftarrow \lfloor length(data)/2 \rfloor$
    $threshold \leftarrow data[m]$
    $subset\_one \leftarrow []$
    $subset\_two \leftarrow []$
    for all $p \in data$ do
        if $p[dimension\_current] \leq threshhold$ then
            $add\_to\_set(subset\_one, p)$
        else
            $add\_to\_set(subset\_two, p)$
        end if
    end for
    $dimension\_next \leftarrow (dimension\_current + 1)\%dimension\_dataset$
    $result.threshold \leftarrow threshold$
    $result.child\_one \leftarrow construct(subset\_one, dimension\_next, dimension\_dataset)$
    $result.child\_two \leftarrow construct(subset\_two, dimension\_next, dimension\_dataset)$
    return $result$
end function
```

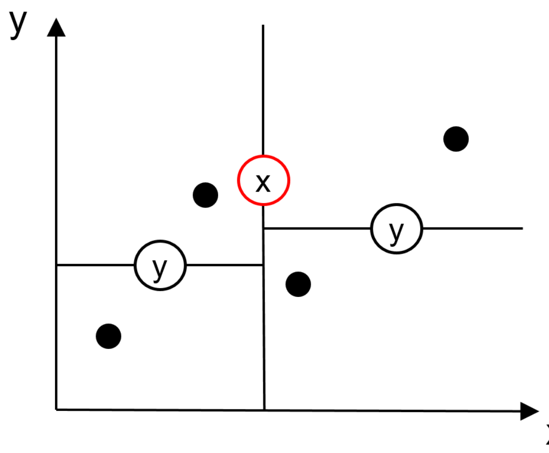


Figure 4.1: Sketch of a KD-tree. The root-node is highlighted in red. On the highest level the search space is divided using a threshold on the x-Axis. On the second layer the y-value is used.

until k neighbors have been found. To do this in the right order a data structure is required that supports fast access and removal of the first (according to a global order) element and insertion of new entries. Some well known structures provide these properties (cf. Algorithm 3). A region of the searchspace can be searched by "cutting" it into its two halves along the threshold. Then both halves get reinserted into the ordering structure. The order metric is the distance of the halve from the query-point. At leaf level single points get inserted into the structure the same way, using the point-to-point distance. Then the next region (or point) from the structure gets searched. Algorithm 4 provides a simple outline of the approach.

Algorithm 3 Prerequisites for a Heap-structure

Some algorithms in this chapter assume, that a **Heap** type exists, which supports the following operations (for the purpose of complexity-analysis n is the number of elements, that currently stored in the structure):

- $insert$: Insert a tuple of arguments into the heap. The entries in the heap get sorted by the first argument. $O(log(n))$
- pop_first: Return and remove the entry with the smallest first argument. $O(log(n))$

Such a structure could be implemented using the techniques described by Adel'son-Vel'skii and Landis [14] or Williams [11].

4.3 Variants

There are some variations that can be applied to the construction or structure of the tree. It is quite common to allow leafnodes to have more than two children. This creates "buckets" on the lowest level, which have to be searched in linear time. Sometimes this can be beneficial for

Algorithm 4 kNNS on KD-Tree

Required functionality:

- *divide_box*: divides the given box along the given axis at the given threshold into two and returns the resulting boxes
- *dist_point_point*: returns the distance between a given tree entry and a point
- *dist_box_point*: returns the distance between a given bounding-box and a point

bb is the axis-aligned-bounding-box of all entries currently present in the tree.

```
function KNNS(root_node, query_point,k,bb,d):
    $result \leftarrow []$
    Heap $todo\_list$
    $todo\_list.insert(0, root\_node, bb, 0)$
    while $length(todo) \neq 0$ and $length(result) \leq k$ do
        $dist, current\_node, current\_bb, current\_axis \leftarrow todo\_list.pop\_first()$
        if $current\_node.leaf\_node \neq$ NULL then
            $add\_to\_set(result, current\_node.leaf\_node)$
            continue
        end if
        $area\_one, area\_two \leftarrow divide\_box(current\_bb, current\_node.threshold, current\_axis)$
        $next\_axis \leftarrow (current\_axis + 1)\%d$
        if $current\_node.child\_one.leaf\_node \neq$ NULL then
            $next\_node \leftarrow current\_node.child\_one$
            $next\_distance \leftarrow dist\_point\_point(next\_node.leaf\_node, query\_point)$
            $todo\_list.insert(next\_distance, next\_node, area\_one, next\_axis)$
        else
            $next\_node \leftarrow current\_node.child\_one$
            $next\_distance \leftarrow dist\_box\_point(area\_one, query\_point)$
            $todo\_list.insert(next\_distance, next\_node, area\_one, next\_axis)$
        end if
        if $current\_node.child\_two.leaf\_node \neq$ NULL then
            $next\_node \leftarrow current\_node.child\_two$
            $next\_distance \leftarrow dist\_point\_point(next\_node.leaf\_node, query\_point)$
            $todo\_list.insert(next\_distance, next\_node, area\_two, next\_axis)$
        else
            $next\_node \leftarrow current\_node.child\_two$
            $next\_distance \leftarrow dist\_box\_point(area\_two, query\_point)$
            $todo\_list.insert(next\_distance, next\_node, area\_two, next\_axis)$
        end if
    end while
    return $result$
end function
```

performance. E.g. for range-queries the requested region and the low-level leafs won't match most of the time and decreasing the trees height saves some overhead of traversing the trees layers. Also the dimension used for splitting the dataset can be picked using different strategies. The original publication [12] uses one dimension after another, but also randomised picking or vaious heuristics can be used. KD-trees can be implemented as homogeneous (as in [12]) or inhomogeneous[1] trees (as in Algorithm 1 & 2).

4.4 Limitations

While some improvements can be made to the algorithm (some leading to the structure addressed in the next chapter) the whole concept shows one critical drawback. Going down one layer, only one dimension is divided into two. The tree as outlined in Algorithm 2 has the height $\lceil \log_2(n) \rceil$. This implies that on high dimensional data not all axes can be considered for indexing. Thereby all entries which are close to the query in the considered dimensions have to be checked in a brute force like manner. The variations from 4.3 reduce the height of the tree and thereby worsen the problem even further. This cannot be solved using KD-trees and limits their potential regarding high dimensional data.

[1]Homogeneous trees store data on their inner nodes, inhomogeneous store every element on leafe-level.

Chapter 5

R-Trees

R-trees can be interpreted as a more powerful improvement to KD-trees, offering a more elaborate data model. They were first proposed in by Guttman [15] and also support an exact solution for nearest neighbor search. For that reason some of the most widespread database operating-systems (such as SQLite) offer implementations of R-trees.

5.1 Structure

R-trees can be used to index boxes or simple points. Either way each such element is stored at leaf-level. Higher level nodes (Algorithm 5) store references to their respective children and an axis-aligned-bounding-box (Algorithm 6). The bounding-box (or point at leaf-level) of each child has to be fully contained in the bounding-box of its parent. There is a crucial parameter for the construction of R-trees: P, the maximum number of children per node. Its impact will be discussed later in this thesis. Any tree that satisfies those constraints will provide correct results, but certain structural properties can have a significant impact on performance (cf. Figure 5.1).

Algorithm 5 R-tree-node

We will assume a R-tree-node (**R_Node**) to have the following members:

- *box*: the axis-aligned-bounding-box
- *children*: a list like structure storing references to the children
- *is_leaf*: indicates whether the node is at leaf-level
- *parent*: reference to the parent of a node

parent is technically not necessary, but can simplify some algorithms.
If *is_leaf* is set to true, *children* refers to datapoints, instead of the next node level.

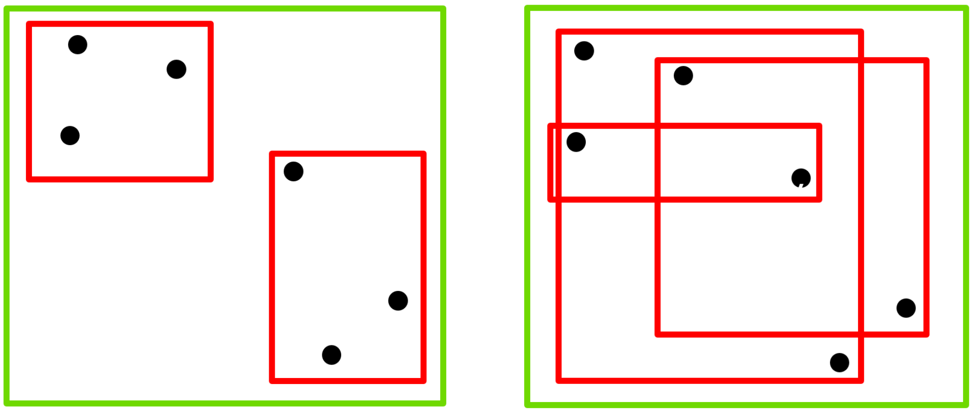


Figure 5.1: Two R-trees ($P = 3$) indexing the same set of points in 2D-space. The left one shows favorable partitioning properties, while the right one contains unnecessary high box-overlap.

5.2 Differences to KD-Trees

While the R- and KD-trees share the approach of hierarchically dividing the search-space, there are some differences. R-trees store their bounding-boxes explicitly, while KD-trees are stuck to a single threshold on each node. The former can be used to implement smarter strategies of dividing space than the one used by KD-trees. Even when the same choices of distributing the points are made, empty spaces between the boxes can randomly appear and lead to performance increases. The other major difference is the parameter P, which gives the maximum number of children per node. Nodes of KD-trees have a maximum of two children per node.

5.3 Construction

There is a variety of algorithms for R-tree construction. Their goal is to construct a tree, that yields a good performance for queries. E.g. overlapping and unnecessary large boxes can cause the complexity to degenerate to $O(n \cdot \log(n))$ (n is the number of elements stored in the tree). On the other hand, building good trees is algorithmically highly complex and “perfect” approaches are thereby not feasible for large datasets. So the following algorithms are mere heuristics that lead to well structured trees in most cases. Note, that all trees constructed by them are guaranteed to fulfill the constraints from 5.1.

5.3.1 Insertion using quadratic split

Guttman [15] initially proposed construction of R-trees by inserting the elements one by one. The most practical algorithm he provided for that is the quadratic split method (Algorithm 7).

Algorithm 6 Box Operations

The algorithms in this chapter need a structure representing axis-aligned-bounding-boxes. It will be assumed, that the following functions are supported:

- *box_from_point*: returns a box that lies on a given point and has an expansion of zero in every dimension.
- *extend_box*: takes the given box and extends its borders to contain the given point, returns the result. The initial box is unchanged.
- *fuse_boxes*: takes two boxes and returns a box containing both
- *box_volume*: returns the volume of the given box
- *box_margin*: returns the margin of the given box
- *distance_box_point*: returns the distance between a given box and a given point

The algorithm consists of two phases: first it chooses a suitable position in the tree and places the data-point there, second "overflowing" nodes (i.e. nodes with more than P children) are split using the quadratic split heuristic (cf. Algorithm 8). Choosing the right position is done as follows: The algorithm recursively descends from root to leaf level. On each level the child-node is chosen, whose bounding-box needs the least enlargement (measured by volume) to contain the inserted data-point. On tie, the smallest box is chosen. On leaf level the new data-point gets inserted into the tree. This may lead to an overflow of this leaf-node. If this is the case the node gets split into two by the following heuristic: Find the two children of the node that span the largest bounding-box on by themselves. Use them as initial elements for the two groups. Turn by turn choose for each group the child from the remaining children that enlarges its box the least. This operation has a complexity of $O(P^2)$ giving it the name quadratic split.

5.3.2 R*-Style Insertion

A conceptually similar algorithm was later shown by Beckmann et al. [16]. While the resulting structure is called R*-tree, it fulfils the same structural constraints as described by Guttman [15] and can thereby be considered a valid R-tree. However it offers a fairly elaborate insertion mechanism and thereby better results (cf. Figure 5.2). This time the algorithm distinguishes between leaf-level and high-level nodes, when picking a position for the data point. For high-level nodes it again picks the one, that needs the least volume increase to fit the point, but for the second lowest level the option is chosen, that leads to the least overlap between the (at most P) child-nodes. After this an overflow-treatment is started. If a node has more than P children, the $\lfloor \frac{P}{2} \rfloor$ children (/subtrees), that are the furthest away from the center of their parents bounding-box, get reinserted into the tree. Only if this triggers an overflow on the same tree-level

Algorithm 7 Insertion using quadratic split

insert_quad(root_node, point) inserts the given point into the tree. The root-node can change during the process.

```
 1: function INSERT_QUAD(node, point):
 2:     $node.box \leftarrow extend\_box(node.box, point)$
 3:     if $node.is\_leaf$ then
 4:         $add\_to\_set(node.children, point)$
 5:         $split\_quad(node)$
 6:         return
 7:     end if
 8:     $least\_enlargement \leftarrow \infty$
 9:     $best\_volume \leftarrow \infty$
10:     $best\_child \leftarrow -1$
11:     for all $i \in \{0, length(node.children) - 1\}$ do
12:         $chld \leftarrow node.children[i]$
13:         $old\_volume \leftarrow box\_volume(chld.box)$
14:         $new\_volume \leftarrow box\_volume(extend\_box(chld.box, point))$
15:         $enlargement \leftarrow new\_volume - old\_volume$
16:         if $least\_enlargement > enlargement$ or
17:         ($least\_enlargement = enlargement$ and $new\_volume < best\_volume$) then
18:             $least\_enlargement \leftarrow enlargement$
19:             $best\_volume \leftarrow new\_volume$
20:             $best\_child \leftarrow i$
21:         end if
22:     end for
23:     $insert\_quad(node.children[best\_child])$
24: end function
```

For the first point an initial leaf-node can simply be constructed using *box_from_point*.

Algorithm 8 Quadratic split heuristic

$vol(X)$: returns the volume of the minimal axis aligned bounding box for a given set of nodes

```
1: function SPLIT_QUAD(node):
2:     $W \leftarrow node.children$
3:     if $|W| \leq P$ then
4:         return
5:     end if
6:     $a, b \leftarrow argmax_{a,b \in W}(vol(\{a, b\}))$
7:     remove $a$ & $b$ from $W$
8:     $A \leftarrow \{a\}$
9:     $B \leftarrow \{b\}$
10:    while $|W| > 0$ do
11:        $i \leftarrow argmin_{i \in W}(vol(A \cup \{i\}))$
12:        remove $i$ from $W$
13:        $A \leftarrow A \cup \{i\}$
14:        $C \leftarrow A$
15:        $A \leftarrow B$
16:        $B \leftarrow C$
17:    end while
18:    set children of $node$ to $A$ and adjust bounding box
19:    create node $m$ with childern $B$ and fitting bounding box
20:    if $node.parent == -1$ then
21:        create new root $r$
22:        add $node$ and $m$ to $r$ and adjust bounding box of $r$
23:        set $r$ as new root of the tree
24:        return
25:    else
26:        $add\_to\_set(node.parent.childern, m)$
27:        adjust bounding box of $node.parent$
28:        $split\_quad(node.parent)$
29:    end if
30: end function
```

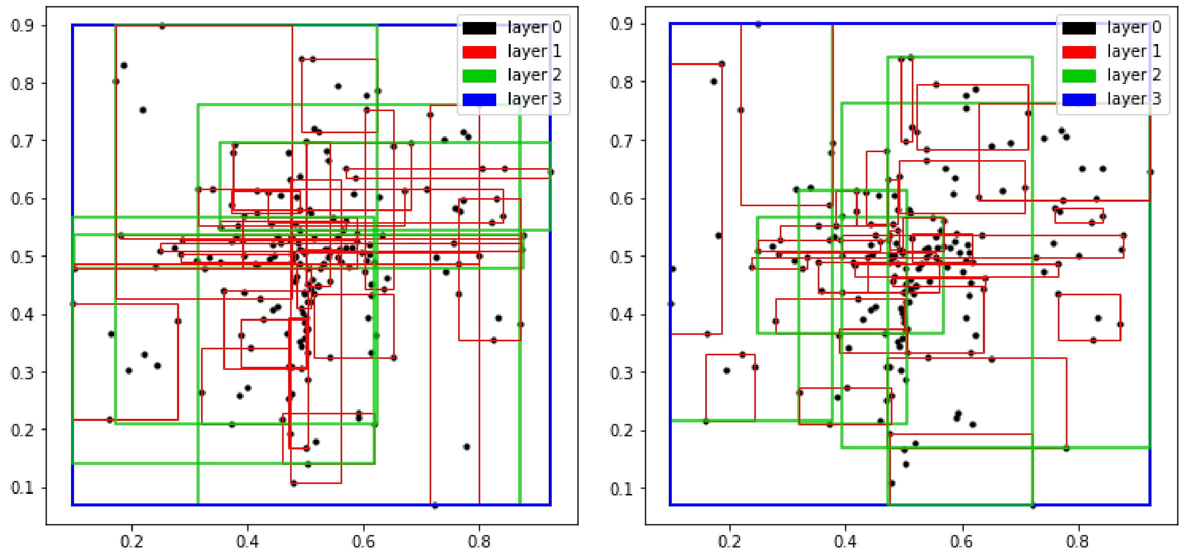


Figure 5.2: A randomly generated dataset loaded ($P = 20$) via quadratic-split (left) and R*-Style insertion (right).

an heuristic driven split is performed.

5.3.3 OMT Bulkload

A vastly different approach on R-tree construction is bulkloading (Algorithm 9). In contrast to the previous method all points have to be inserted at once an thereby be known at this point in time. The best-known approach, STR, was proposed by Leutenegger et al. [17]. An improvement on that was later developed by Lee and Lee [18]. They proposed an “overlap minimizing top-down bulk loading algorithm” (OMT). The approach is similar to KD-trees. Again the dataset is cut along one axis on each level (cf. Figure 5.3). In contrast to KD-trees however, each node yields up to P children instead of two. It is noteworthy, that this is by far the cheapest approach to construct.

5.4 K-Nearest-Neighbor-Search

Nearest neighbor search on R- and KD-trees is quite similar. A related technique was shown by Cheung and Fu [19], however the code used in this thesis (Algorithm 10) is implemented iteratively instead of recursively for better readability and performance. There is a set of currently considered boxes and points, initially containing the root-node. The elements of this set get “explored” in ascending order (distance from the query). Since every node explicitly stores a bounding box, distances between those boxes and the query-point can be directly computed.

Algorithm 9 Simplified OMT-Bulkload

$bulkload(dataset, 0, d, p)$ returns the root-node

```
1: function BULKLOAD(data, dimension_current, dimension_dataset, P):
2:     R_Node $result$
3:     if $length(data) \leq P$ then
4:         $result.is\_leaf \leftarrow$ true
5:         $result.box \leftarrow box\_from\_point(dataset[0])$
6:         $add\_to\_set(result.children, dataset[0])$
7:         for all $i \in [1, len(data) - 1]$ do
8:             $add\_to\_set(result.children, dataset[i])$
9:             $result.box \leftarrow extend\_box(result.box, dataset[0])$
10:        end for
11:        return $result$
12:    end if
13:    $result.is\_leaf \leftarrow$ false
14:    $result.box \leftarrow box\_from\_point(dataset[0])$
15:    $data \leftarrow sort\_along\_axis(data, dimension\_current)$
16:    $n \leftarrow len(datat)$
17:    $step \leftarrow \lceil \frac{n}{P} \rceil$
18:    for all $i \in [0, P - 1]$ do
19:        $from \leftarrow i \cdot step$
20:        $to \leftarrow min((i + 1) \cdot step, len(data))$
21:        $child\_data \leftarrow subset\_from\_to(data, from, to)$
22:        $next\_child \leftarrow bulkload(child\_data, (dimension\_current + 1)\%dimension\_dataset, P)$
23:        $next\_child.parent \leftarrow result$
24:        $add\_to\_set(result.children, next\_child)$
25:        $result.box = fuse\_boxes(result.box, next\_child.box)$
26:    end for
27:    return $result$
28: end function
```

Obviously not all nodes in the tree can have exactly P children (except if the size of the dataset is a potential of P). The above described algorithm leaves those empty spaces at leaf-level while the original OMT [18] makes additional calculations to distribute them equally.

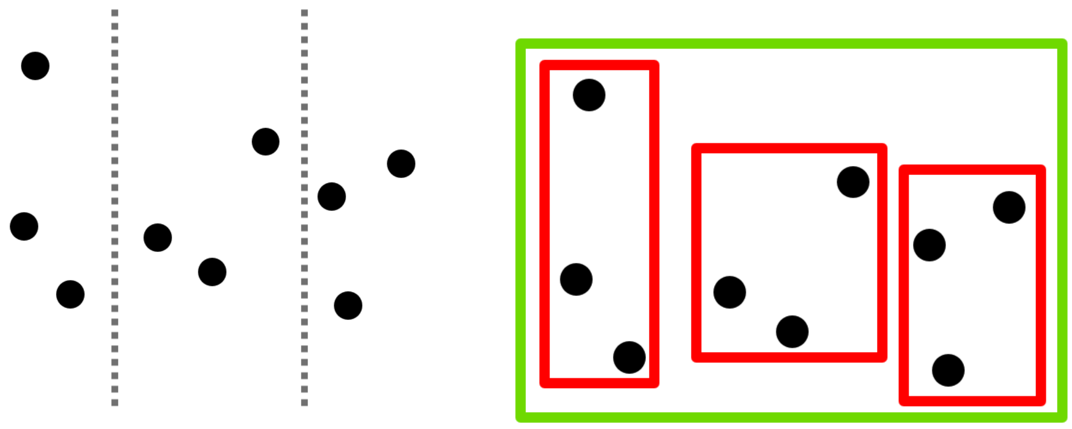


Figure 5.3: A set of points gets bulkloaded into an r-tree ($P = 3$). The set gets cut along the horizontal axis, leading to the tree on the right. If there were more than 9 points, an additional tree-layer would be needed. In this case the red boxes would be cut along the vertical axis after that.

Exploring a box leads to reinsertion of its children into the set. If a point gets explored, it means that it is the next result. The algorithm ends, if k results have been found or all nodes and points have been explored.

5.5 Variants

An advanced version of R-trees is the X-Tree. Berchtold et al. [20] developed this structure to allow for construction of better performing trees. To further decrease node-overlap, split operations get delayed until good splits can be made. These delays however can theoretically cause the tree to degenerate to a linear structure. While technically not R-trees, M-Trees share many concepts with them. Proposed by Ciaccia et al. [21], they organize their content in ellipsoids instead of boxes. Guhlemann et al. [22] later showed further optimization methods for kNNS on these structures.

5.6 Limitations

As previously mentioned the limiting factor for R-tree performance is the construction. While the datastructure theoretically allows for efficient organization of the search-space, this really powerful tool cant be fully utilized. One’s ability to invest computation time in more elaborate construction algorithms determines the performance of the resulting tree. For high dimensional data and large databases some strategies might not be feasible at all.

Algorithm 10 kNNS on R-tree

cf. Algorithms 3, 5 & 6

```
1: function KNNS(root_node, query_point,k):
2:     $result \leftarrow []$
3:     Heap $todo\_list$
4:     $todo\_list.insert(0, root\_node,$false$)$
5:     while $length(todo) \neq 0$ and $length(result) \leq k$ do
6:         $dist, current\_node, is\_result \leftarrow todo\_list.pop\_first()$
7:         if $is\_result$ then
8:             $add\_to\_set(result, current\_node)$
9:             continue
10:        end if
11:        for all $c \in current\_node.children$ do
12:            if $current\_node.is\_leaf$ then
13:                $dist \leftarrow distance\_point\_point(c, query\_point)$
14:                $todo\_list.insert(dist, c,$true$)$
15:            else
16:                $dist \leftarrow distance\_box\_point(c.box, query\_point)$
17:                $todo\_list.insert(dist, c,$false$)$
18:            end if
19:        end for
20:    end while
21:    return $result$
22: end function
```

Chapter 6

Hierarchical Navigable Small World Graphs

One of the most versatile datastructures in computer science are graphs. A large variety of problems can be mapped on a graph-like construct with the right properties. This holds true for the given problem of kNNS, where graphs can be used to approximate the solution. The naive approach are k-nn-graphs[1]. Greedily traversing them can lead to decent approximations, however the number of steps required to traverse the dataset is potentially high and global connectivity of the graph can easily be lost on clustered data. Those problems can somewhat be reduced by using small-world networks[2] (e.g. Malkov et al. [23]), but the results are still not ideal since there is no guarantee, that the search algorithms will use short (measured in number of traversed edges) paths (e.g. Kleineberg [24]). This leads to the usage of hierarchical graphs, which ensure fast query times.

6.1 Structure

The graphs that will ultimately be used as an indexing structure are hierarchical directed graphs. The method was proposed by Malkov and Yashunin [25]. On the lowest level each data-point from the indexed set is present as a node and connected to its M nearest neighbors. Higher levels are deduced by randomly removing nodes from the layer below and again connecting the remaining nodes with their respective M nearest neighbors on this level (Figure 6.1). The expected number of hierarchy levels is thereby logarithmic. Those connections are generally bidirectional. It is however possible for a node to have significantly more than M outgoing edges, since membership in the M-nearest neighbors of a node isn't commutative.

[1]A graph in which each node is connected to its k nearest neighbors

[2]A graph (G) in which for each two nodes $a, b \in G$ exists a path, that can be traversed in $O(log(|G|))$

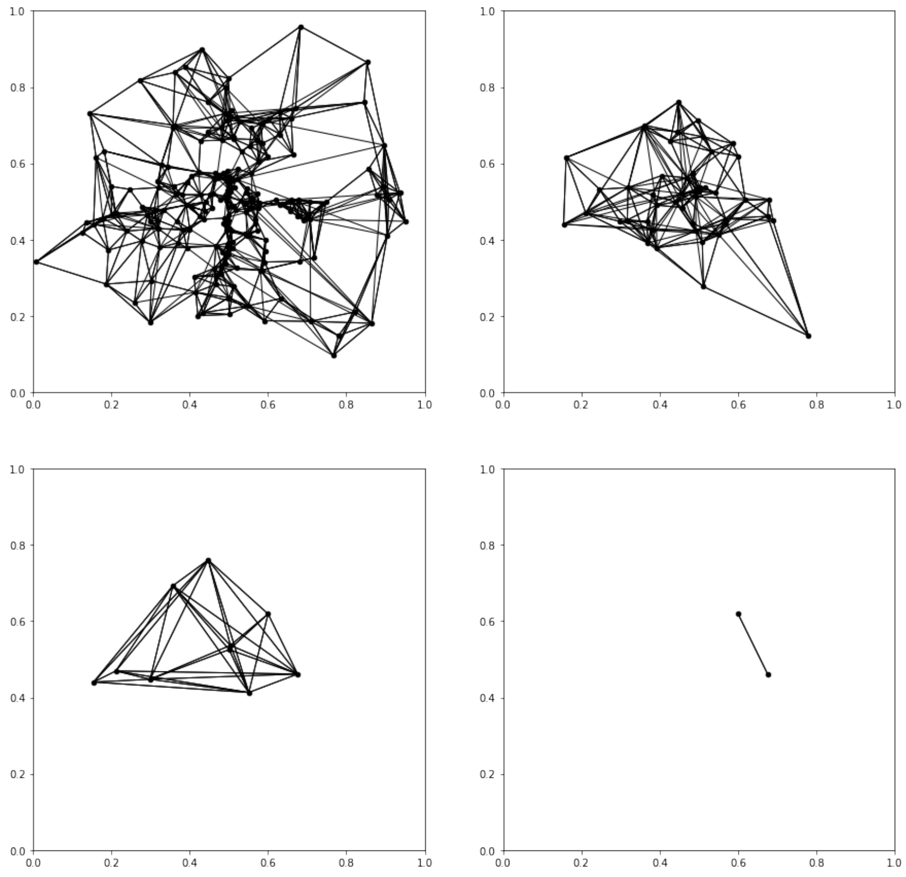


Figure 6.1: HNSWG indexing structure constructued on random 2D-data (M = 5).

6.2 K-Nearest-Neighbor-Search

A key component to construction and usage of this structure is the *search_layer* operation (Algorithm 11, Malkov and Yashunin [25]). It traverses a certain layer of the graph greedily towards a query-point, starting from a set of entry-points. It returns a given number of results found this way. For kNNS any point at the highest level can serve as an entry-point. The algorithm starts by using the entry-point to find a single close point on the highest layer. This result is then used as an entry-point for the search of the level directly below. This pattern is repeated until the lowest level is to be searched. Theoretically it would be sufficient to search for k results here, however in practice at least e_f =∼100 results get retrieved and the k best are returned to ensure stable quality for small k. The influence of the search-parameter e_f will be discussed later.

Algorithm 11 Search a given layer for nearest neighbors

This function searches a given layer for the nearest neighbors of point q by greedily descending towards it. At any given moment a maximum e_f points are considered and as soon as no greedy improvement can be made on them, the set is returned.

```
1: function SEARCH_LAYER($q, entry\_points, e_f, layer$):
2:     $visited \leftarrow []$
3:     Heap $candidates$
4:     Heap $considered$
5:     for all $p \in entry\_points$ do
6:         $dist \leftarrow distance\_point\_point(q, p)$
7:         $add\_to\_set(visited, p)$
8:         $candidates.insert(dist, p)$
9:         $considered.insert(-1 \cdot dist, p)$
10:    end for
11:    while $length(candidates) > 0$ do
12:        $best\_cand \leftarrow candidates.pop\_first()$
13:        $worst\_cons \leftarrow considered.get\_first()$
14:        $distance\_cand \leftarrow distance\_point\_point(q, best\_cand)$
15:        $distance\_cons \leftarrow distance\_point\_point(q, worst\_cons)$
16:        if $distance\_cand > distance\_cons \cdot (-1)$ then
17:            break
18:        end if
19:        for all $n \in$ [neighborhood of $best\_cand$ at $layer$] do
20:            if $n \notin visited$ then
21:                $add\_to\_set(visited, n)$
22:                $worst\_cons \leftarrow considered.get\_first()$
23:                $distance\_cons \leftarrow distance\_point\_point(q, worst\_cons)$
24:                $distance\_neigh \leftarrow distance\_point\_point(q, n)$
25:                if $distance\_neigh < distance\_cons \cdot -1$ or $length(considered) < e_f$ then
26:                    $candidates.insert(distance\_neigh, n)$
27:                    $considered.insert(-1 \cdot distance\_neigh, n)$
28:                    if $length(considered) > e_f$ then
29:                        $considered.pop\_first()$
30:                    end if
31:                end if
32:            end if
33:        end for
34:    end while
35:    return $to\_list(considered)$
36: end function
```

Parameter	Meaning
M	minimum number of outgoing edges per node
M_{max}	maximum number of outgoing edges per node
m_L	survival chance per node per level
e_f	minimum number of elements to search for during queries
$e_{fconstruction}$	elements to search for during insertion

Table 6.1: Overview over HNSWG parameters

6.3 Construction

The graph is constructed by iteratively inserting data into it. For each entry the highest hierarchy level it appears on gets chosen by random (using an exponential distribution). If this level is higher than the current hight of the graph, the new node becomes the new entry point for searches. Otherwise the graph gets searched level by level like in 6.2 until the chosen target level is reached. From there Algorithm 11 is used on each level to retrieve $e_{fconstruction}$ nearest neighbors and bidirectionally connect the new node to the M nearest of them. The results of the last search are used as an entry point for the next search, one level lower.
This procedure can however lead to a node having more than M outgoing edges on single layer. Consequently an insertion pattern can be constructed that connects a single node with the entire remaining graph. As soon as Algorithm 11 reached this node, search time would degenerate greatly. Thus it is necessary to introduce another construction parameter: the maximum number of outgoing edges per node M_{max}. If a node surpasses this predefined constant, outgoing edges are unilaterally removed. An easy way to do that, is to simply remove the longest ones. There are however more elaborate ways to select edges for removal.

6.4 Parameters

As implied earlier in this chapter, there are multiple parameter choices influencing the performance of the structure. For a brief compilation see Table 6.1. Their impact will be discussed now.

6.4.1 M

The minimum number of outgoing edges per node is maybe the most influential parameter and applies to the whole graph. Higher values for M worsen the query times since Algorithm 11 has to consider at least M neighbors per node that it visits. However lower values decrease result quality, since greedily descending becomes less probable to “get lost” if the connectivity of the graph increases. So a tradeoff has to be made. Malkov and Yashunin [25] named a range from 5 to 48 as for that. Generally speaking higher values of M are required as dimensionality increases.

6.4.2 $\mathbf{M}_{max}$

Choosing values for M_{max} is simpler. Too high values lead to a degeneration of runtime, too low ones destroy graph connectivity. However fixing it to $2 \cdot M$ shows to be a reasonable tradeoff.

6.4.3 $\mathbf{m}_L$

The hierarchical structure is crucial for query performance. m_L decides the height of this structure. On the first glance one might see an analogy to tree-like structures and think high structures offers the highest speedup. However here the referencing of neighbors is by itself somewhat independent of the height. But it is still beneficial to select it contrary to M. Doing otherwise would lead to unnecessary redundancy between the levels since most of the neighborhoods on two consecutive layers would overlap if both parameters are chosen high. A reasonable choice is $m_L = \frac{1}{ln(M)}$.

6.4.4 $\mathbf{e}_f$

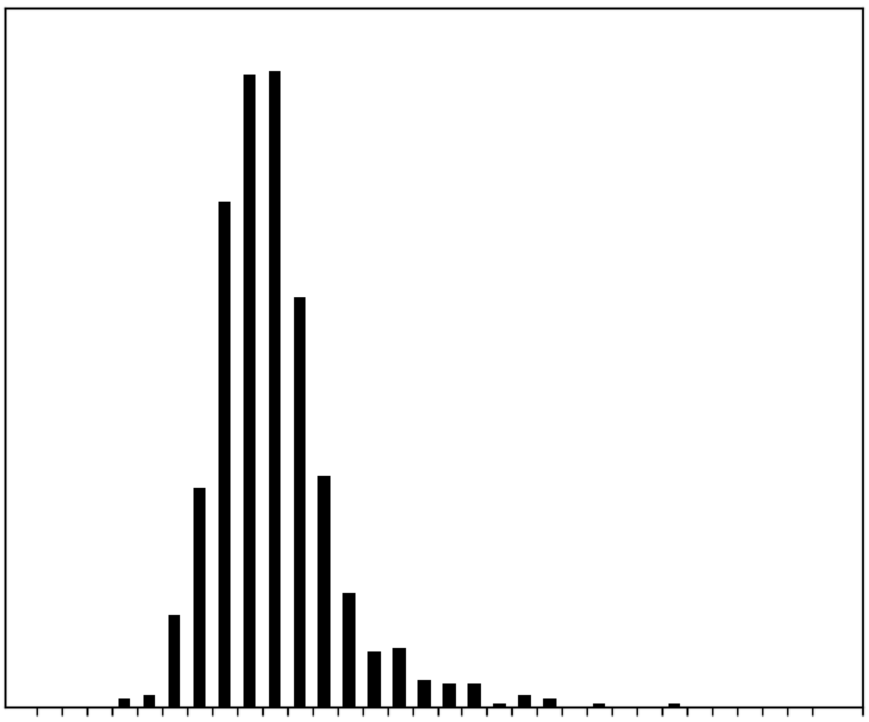
This parameter is only relevant for queries. Instead of searching for k neighbors directly, $max(k, e_f)$ elements get retrieved from the structure. This tries to provide consistent results for small queries. E.g. for $k = 1$ a single path of descend could "get lost" easily. However its impact is limited. A reasonable value is between 50 and 200.

6.4.5 $\mathbf{e}_{fconstruction}$

While being similar to e_f it controls the number of candidates during insertions. Remarkably it only impacts construction cost negativly while increasing indexing quality. If one is only considering query times, this value could theoretically be selected arbitrary high. Its impact however is again limited. Even a perfect graph won't provide perfect results, since even descending greedily along the actual nearest neighbors does not necessarily solve the problem.

Part III

Results



Chapter 7

Performance Measurement

7.1 Dataset

The dataset used for testing the described algorithms is a collection of classical pieces (Table 7.1). It has a total duration of 32h. As described by Grosche and Müller [3] audio shingles were generated from this data resulting in roughly 120 thousand shingles, each having 240 dimensions. To accelerate retrieval, principal component analysis (PCA) and deep neural networks (DNN, Zalkow and Müller [9]) were used to reduce the feature dimensionality. This lead to 3 new versions of the database for each of two reduction methods. The dimensionality per shingle was reduced to 6, 12 and 30.

7.2 Time Comparison

To get a better understanding for the query performance of various R-tree (Chapter 5) approaches compared to the graph-based algorithm (Chapter 6) both methods were implemented in Python 3. However it has to be stated this is not practical for real-life database applications since Python, being a non-compiled language, is inherently slower than other programming languages. So additionally we will compare already existing, better optimized implementations of nearest neighbor search. In return those can't be compared to python implementations with respect to query response times.

Composer	Piece	Versions
Beethoven	Op. 55, Mov. 1	4
	Op. 55, Mov. 2	4
	Op. 55, Mov. 3	4
	Op. 55, Mov. 4	4
	Op. 67, Mov. 1	10
	Op. 67, Mov. 2	10
	Op. 67, Mov. 3	10
	Op. 67, Mov. 4	10
Chopin	Op. 7, No. 1	53
	Op. 17, No. 4	62
	Op. 24, No. 1	61
	Op. 24, No. 2	64
	Op. 30, No. 2	33
	Op. 33, No. 2	67
	Op. 33, No. 3	50
	Op. 63, No. 3	86
	Op. 68, No. 3	51
	Op. 64, No. 4	62
Vivaldi	RV 269, Mov. 1	7
	RV 269, Mov. 2	7
	RV 269, Mov. 3	7
	RV 315, Mov. 1	7
	RV 315, Mov. 2	7
	RV 315, Mov. 3	7

Table 7.1: Pieces in the database.

7.3 Retrieval Quality Estimation

To estimate the quality of certain retrieval approaches mainly three measures were used: $P@1$, P_R and P_{3R}. To compute those, the following experiment was performed on the various indexing methods.

1. We chose a recording by random and select a random shingle from it (both equally distributed). Let this shingle be q.

PCA:

	$D = 6$	$D = 12$	$D = 30$
$P@1$	1.000	1.000	1.000
P_R	0.838	0.953	0.977
P_{3R}	0.774	0.903	0.961

DNN:

	$D = 6$	$D = 12$	$D = 30$
$P@1$	1.000	1.000	1.000
P_R	0.865	0.956	0.988
P_{3R}	0.828	0.919	0.970

Table 7.2: Performance measures evaluated on the datasets

2. For the query q, we determine the number of recordings of the same piece in the data base. Let this number be R.
3. Search the structure for the $3 \cdot R$ nearest neighbors of q.
4. We call results meaningfull, if they belong to a recording of the same piece as q.
5. $P@1$ is 1, if the highest ranking result is meaningful, 0 otherwise.
6. P_R is the amount of meaningful results in the R highest ranking results divided by R.
7. P_{3R} is the amount of meaningful results in the $3R$ results divided by $3R$.
8. We repeat this process a set number of times (1000) and compute the average values for $P@1$, P_R and P_{3R}.

Since all algorithms presented in this thesis use euclidean nearest neighbor search, a maximum possible score can be assigned to each of the six versions of the dataset (Section 7.1) by evaluating those metrices on an exact approach. This gives a measurement for the quality of the various feature reduction methods (cf. Table 7.2).

Chapter 8

R-Tree Performance

As already stated in Chapter 5, the strategy used to construct an R-tree and the selection of parameter P have a vast influence on the performance of the resulting structure. However any R-Tree will solve nearest neighbor search exactly, so we don't have to consider retrieval quality in this case. In fact, the expected performance measures for each experiment in this chapter can be taken directly from Table 7.2.

8.1 Quadratic Split Insertion

Quadratic split insertion is a method originally proposed by Guttman [15]. A summary of query times can be seen in Table 8.1. Two meaningful conclusions can be taken from that. First of all, increasing the pagesize P doesn't give an actual benefit beyond a value of around 5. Moreover data that is reduced using PCA is easier to index. We will get back to a similar effect later. A histogram of this best setting is shown in Figure 8.1. The query times are fairly widely spread. Its crucial for the performance how "well constructed" the tree is at the position that gets queried. If all the k correct results are contained in the theoretically minimum required $\lceil \frac{k}{P} \rceil$ leaf-level boxes and there is no overlap, query complexity is reduced to $O(log(n))$, where n is the size of the database. This way queries on trees constructed that way can (at least in theory) be really fast. Construction is quite expensive since the complexity of this method is proportional to P^2. For the given dataset the Python implementation took an average of 80 seconds to construct the structure for 12-dimensional data.

	PCA	DNN
$P = 2$	213ms	219ms
$P = 5$	118ms	141ms
$P = 10$	119ms	141ms

Table 8.1: Query times for quadratic split $d = 12$ and $k = 30$

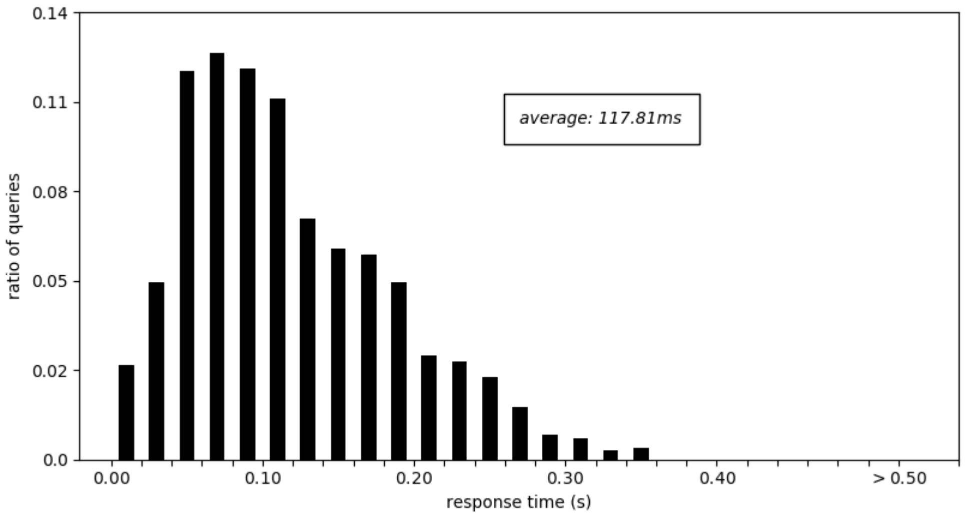


Figure 8.1: Histogram of query times using an R-tree constructed with quadratic split ($d = 12$, $P = 5$, $k = 30$).

8.2 R*-Style Insertion

For the performance of R*-trees cf. Table 8.2. On the first glance R*-style insertion does even worse than the traditional method. This is quite surprising, since it is the more elaborate and expensive method. The effect is caused by the comparatively high dimensionality of the data. However the query times can be decreased by increasing P. Unfortunately this is only practical up to certain degree, since this again increases complexity for constructing the tree quadratic with P. On top of this, R* trees are already computationally expensive to construct. The histogram (Figure 8.2) shows some bad outliers. In this case the worst one took 420 ms to complete a query. To put that in relation, a brute force approach implemented on a similar level takes around 700 ms to complete, showing that complexity has indeed degenerated severely.

	PCA	DNN
$P = 5$	$143ms$	$160ms$
$P = 10$	$138ms$	$147ms$
$P = 15$	$121ms$	$142ms$

Table 8.2: Query times for R*-trees $d = 12$ and $k = 30$

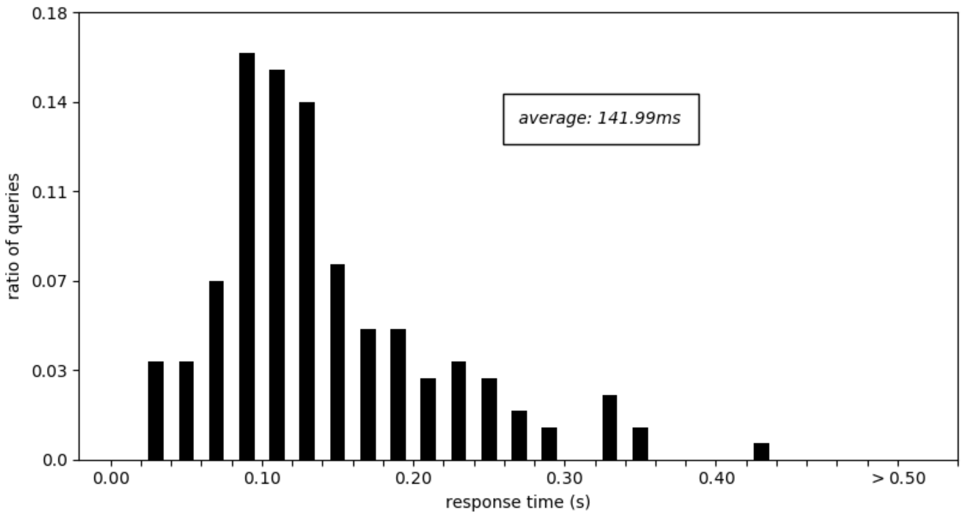


Figure 8.2: Histogram of query times using an R*-Tree ($d = 12$, $P = 15$, $k = 30$).

8.3 OMT Bulkload

Bad performance and degeneration of complexity of the last two approaches were caused by failures of the respective heuristics during the over 120 thousand insertion operations. Bad decisions at the beginning tend to get even worse later since both approaches will expand unnecessary large boxes even more on future insertions. Bulkloading approaches don't suffer from this problem. They use their knowledge of the whole dataset to produce a well ordered structure at once. And while the strategy described earlier is quite simple, the performance gain of this concept is huge. In this case overlap is prevented completely. Another benefit is the comparatively really fast construction of the structure (on our database less than 4 seconds).

Choice of parameter P

The used bulkloading algorithm excels for small P (as seen in Table 8.3). In fact, it could reasonably be set to the smallest possible value ($P = 2$). However this would lead to preventable

	PCA	DNN	DNN + base exchange
$P = 2$	19ms	23ms	14ms
$P = 5$	20ms	29ms	14ms
$P = 10$	23ms	51ms	22ms

Table 8.3: Query times for bulkloaded R-trees $d = 12$ and $k = 30$

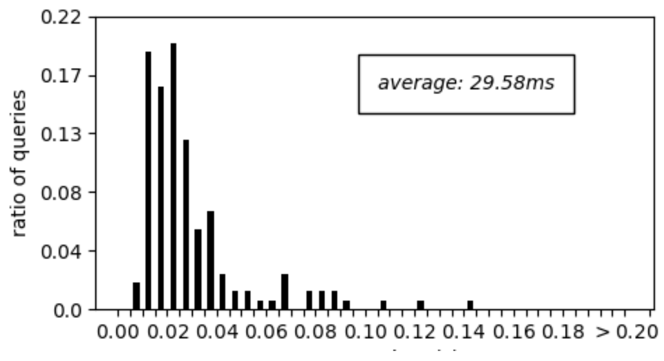
memory overhead and worse performance on larger queries (magnitude of k). $P = 5$ seems like a better tradeof for the considered usecase.

Choice of Reduction Method and Optimization

The method of reduction has great impact on the performance of the resulting structure (cf. Table 8.3). This effect is caused by the special properties that a dataset after PCA shows. Principal component analysis changes the base of the dataset linearly so that the new axes are sorted after the variance of the given dataset along them. This is normally used to construct axes, that show only little significance and can be omitted safely (feature reduction). This sorting of axes synergizes really well with OMT bulkloading since the latter divides the dataset along one axis after the other. The first axis is the one with the greatest variance after PCA, so the bulkloading algorithm is guaranteed to make a meaningful decision here. The resulting box structure is also more likely to even have gaps in between the boxes, reducing the probability that multiple high-level boxes have to be opened to process a query. This continues on lower levels. This insight can now be used, to gain this performance benefit on arbitrary data. Simply changing the base of the coordinate-system in the way PCA does, doesn't alter the results of euclidean kNNS. Combining this with the data reduced with a DNN, gives the best of both worlds. For $k = 30$ and $d = 12$ this could cut the query cost in half (cf. Figure 8.3 and Table 8.3).

Influence of Dimensionality

Irrespective of the used construction method, the dimensionality d of the dataset has a large impact on the performance of R-trees. They fall victim to the infamous "curse of dimensionality". During kNNS, boxes have to be considered, if their distance to the query is smaller than the distance of the k-th neighbor. For high dimensional boxes, this becomes a less and less effective way of pruning parts of the dataset. On the other hand Table 8.4 shows, that this logic applies in both ways. If bulkloading and the associated optimization are used, R-trees become a highly



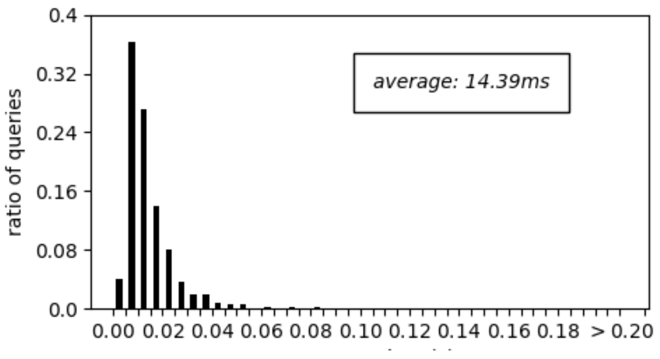


Figure 8.3: Comparison of querytimes after OMT-bulkloading with (lower) and without (upper) changing the base (DNN, $d = 12$, $p = 5$, $k = 30$).

	DNN ($d = 6$)	DNN ($d = 12$)	DNN ($d = 30$)
Query Time	$2.7ms$	$14.4ms$	$97.0ms$

Table 8.4: Query times for bulkloaded R-trees after changing the base ($k = 30$. $p = 5$).

competitive datastructure for low-dimensional use-cases. For $d = 1$ and $P = 2$ this method in fact is equivalent to an inhomogeneous binary search tree which has a guaranteed complexity of $O(log(n) + k)$ per query.

8.4 Conclusion

With the right optimization R-trees offer a fast solution for kNNS. Bulkload is the by far best option here. The other techniques can be used, if additional data has to be inserted later. Solving the problem directly for high dimensional data ($d > 12$) is however not really feasible for large query numbers. But if the dimensionality can be drastically reduced without sacrificing too much retrieval quality, R-trees are already a satisfactory solution.

Chapter 9

HNSWG Performance

In contrast to R-trees, HNSWG only offers an approximation of kNNS. Every choice directly impacts retrieval quality, hence the latter has to be considered constantly. On multiple occasions we will see, that we can "buy" retrieval quality with run time and vice-versa. Those tradeoffs will be evaluated against R-trees as a baseline.

9.1 Parameter Choices

For the large part of the parameters we will stick to the recommendations by Malkov and Yashunin [25] (cf. Section 6.4). However two parameters deserve a closer look.

M

The parameter M is a prime example for a tradeoff between retrieval quality and performance. Table 9.1 gives a feeling for this. With higher values of M the performance measures converge relatively fast towards those of the exact solution. This point however depends strongly on the dimensionality of the dataset. For $M = 10$ this convergence seems to have happened sufficiently (Figure 9.1), offering nearly the quality of the 30-dimensional dataset, while still being faster than the best R-tree variant on a 12-dimensional one. A histogram of the query times for this setup is given in Figure 9.2.

$\mathbf{e}_{fconstruction}$

For all experiments conducted earlier, the parameter $e_{fconstruction}$ was fixed to 100. In the-

	Query Time ($k = 30$)	$P@1$	P_R	P_{3R}
R-Tree DNN30	97.0ms	1.000	0.988	0.970
R-Tree DNN12	14.4ms	1.000	0.956	0.920
HNSWG DNN30 $M = 3$	7.5ms	0.939	0.910	0.889
HNSWG DNN30 $M = 5$	8.7ms	0.982	0.959	0.936
HNSWG DNN30 $M = 7$	9.5ms	0.991	0.970	0.952
HNSWG DNN30 $M = 10$	10.0ms	0.998	0.984	0.969

Table 9.1: Performance of HNSWG ($e_f = 100$) for various values of M compared to R-trees.

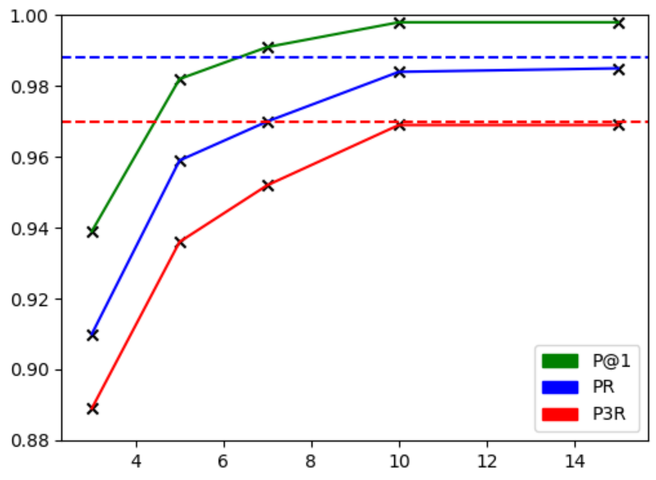


Figure 9.1: Course of HNSWG precision with increasing M (DNN, $d = 30$, $e_f = 100$, $k = 30$)

.

ory it should be able to increase retrieval quality, without sacrificing time after construction. However Table 9.2 shows, that this only holds true up to a certain degree. The parameter aims to prevent “mistakes” during construction, but for reasonably large values harmful effects caused by the approximative nature of the algorithm become rare anyway. The effect is measurable but for the real world use case probably not relevant. It is also limited, since for $e_f construction = \infty$, a perfect nearest neighbor graph will be constructed, which still doesn't solve the problem exactly.

9.2 Influence of Dimensionality

A huge difference between HNSWG and R-trees is how they behave for different dimensionalities. While R-trees gain a huge speedup as d decreases, the search complexity for HNSWGs changes

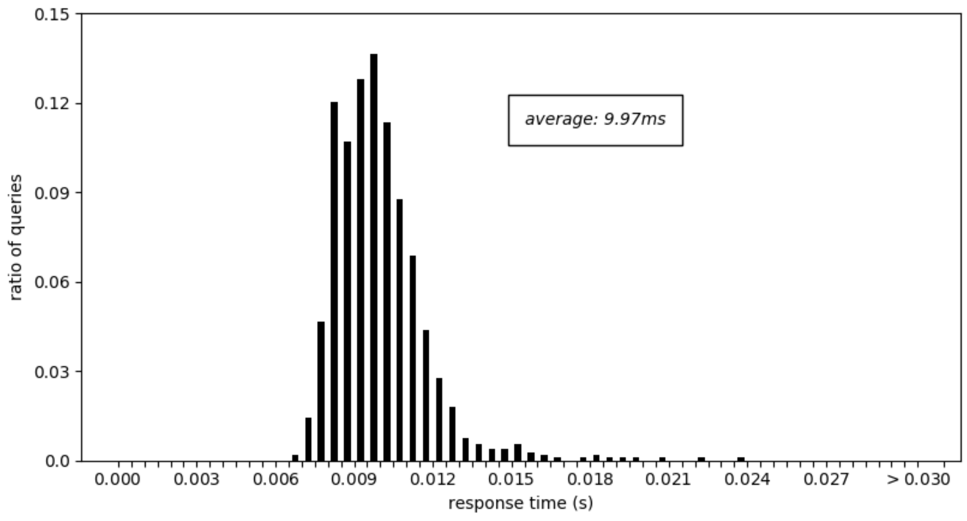


Figure 9.2: Histogram of HNSWG query times (DNN, $d = 30$, $M = 10$, $e_f = 100$, $k = 30$).

	$P@1$	P_R	P_{3R}
HNSWG DNN30 $e_{f construction} = 50$	0.990	0.966	0.946
HNSWG DNN30 $e_{f construction} = 100$	0.991	0.970	0.952
HNSWG DNN30 $e_{f construction} = 150$	0.992	0.975	0.955

Table 9.2: Performance of HNSWG ($M = 7$, $e_f = 100$) for various values of $e_{f construction}$ compared to R-trees.

	DNN ($d = 6$)	DNN ($d = 12$)	DNN ($d = 30$)
Query Time	$4.81ms$	$6.15ms$	$8.69ms$

Table 9.3: Query times for HNSWG ($M = 5$, $e_f = 100$, $k = 30$).

only marginally (cf. Table 9.3). However lower dimensionalities tend to need smaller values of m to yield satisfying results. For example in Table 9.4 a value of $M = 3$ was suffiecient to produce a good approximation.

9.3 Construction Time

A disadvantage of the method is however the time needed for index construction. The structure is build by performing a query for each element iteratively. Thereby the construction cost increases proportionally to the query cost. For our case constructing a single graph took between 10 and

	Exact Solution	HNSWG($M = 3$)
$P@1$	1.000	0.997
P_R	0.865	0.864
P_{3R}	0.828	0.823

Table 9.4: Precision of HNSWG ($e_f = 100$) compared to an exact solution on 6-dimensional data (DNN).

20 minutes. Here however lies potential for further optimization via parallelisation. The only operation needed for construction, that must not be executed parallel, is the insertion of new edges on the same node. This step is not only cheap but also should rarely occur several times at once on the same node. If a fine-grained, optimistic synchronization approach is used to cover this case, a nearly perfect speedup could be reached on massive parallel execution.

9.4 Conclusion

Despite being an approximation algorithm, HNSWG can deliver nearly accurate results if the parameters are chosen correctly. Query response times aren't hurt badly if a high enough value for M is selected. Where this point is, has to be determined depending on the use-case. It might be even feasible to increase the dimensionality even further. However for low dimensional cases, there are better alternatives.

Chapter 10

Existing Implementations

Finally we take a look at two already existing implementation of kNNS. Both are opensource and freely available. They are implemented in better suited languages and therefore much more practical than the Python implementations used for research.

10.1 Scikit-Learn KD-Trees

Scikit-learn (`https://scikit-learn.org/stable/index.html#`) is a machine-learning toolkit for Python. It offers a wide variety of algorithms from this field. Since kNNS is a frequently reappearing problem here, it comes with its own implementation of KD-trees for this purpose. Version 0.21.2 of Scikit-learn was used for testing. The package is implemented using Cython, a Python dialect designed to be translated to C-code and compiled afterwards. Depending on the exact usage, the speedup factor of this technique can be huge (> 100) compared to standard Python code. The resulting times can be seen in Figure 10.1. It behaves similar to an R-tree, which seems logical, since the approaches are conceptually similar (cf. Figure 10.1).

10.2 Hnswlib

Hnswlib is an implementation of the method described by Malkov and Yashunin [25] (cf. Chapter 6). It is currently (November 14, 2019) available at `https://github.com/nmslib/hnswlib`. The library was tested using the included Python interface to ensure equal testing conditions. The algorithms itself are open source and implemented using C++ 11.

Performance

	$d = 6$	$d = 12$	$d = 30$
Query Time	335μs	562μs	3051μs

Table 10.1: Query times of Scikit-learn KD-trees (DNN, $k = 30$).

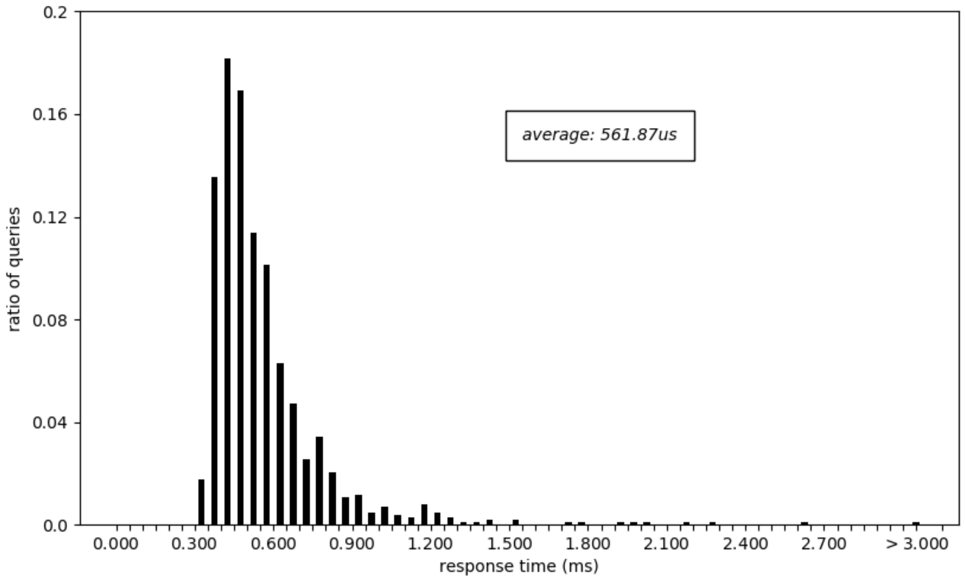


Figure 10.1: Histogram of Scikit-learn KD-tree query times (DNN, $d = 12$, $k = 30$).

	$d = 6$	$d = 12$	$d = 30$
$M = 3$	53μs	55μs	72μs
$M = 5$	55μs	61μs	79μs
$M = 7$	58μs	68μs	84μs
$M = 10$	61μs	68μs	98μs

Table 10.2: Query times of Hnswlib (DNN, $k = 30$, $e_f = 100$)

The query times of Hnswlib can be seen in Table 10.2. Being directly compiled, the C++ code is naturally significantly faster than the Python implementation. This performance improvement was the most noticeable during the index construction. On our database it didn't take longer than 10 seconds for any tested setup. An exemplary histogram is shown in Figure 10.2. It should be stated, that measuring response times that fast from a Python interface is somewhat imprecise. The delay caused by the interface itself starts being a relevant factor.

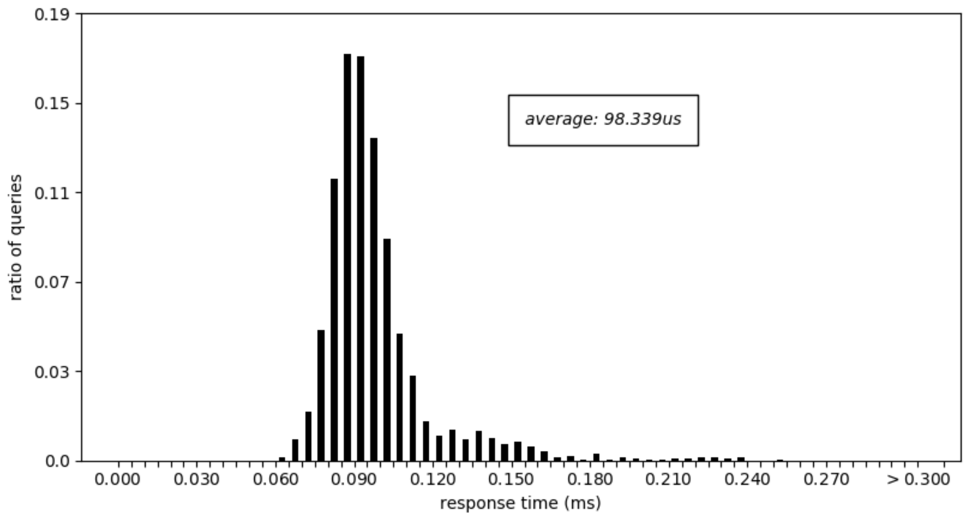


Figure 10.2: Histogram of hnswlib query times ($d = 30$, $M = 10$, $e_f = 100$, $k = 30$).

	$P@1$	P_R	P_{3R}
$M = 3$	0.995	0.974	0.955
$M = 5$	0.998	0.982	0.960
$M = 7$	0.999	0.985	0.964
$M = 10$	0.999	0.985	0.965

Table 10.3: Precision of Hnswlib (DNN, $d = 30$, $e_f = 100$) for various values of M.

Precision

The precision of Hnswlib is, as seen in Table 10.3, actually even better than the one achieved by our implementations (cf. 9.1). This is caused by the usage of a more elaborate heuristic for edge selection during construction. Since in this case construction times of the graph are negligible anyway, this is a good tradeoff. Interestingly it causes the precision to converge for smaller M. This opens an additional option for saving time.

Chapter 11

Conclusions

It is a known fact, that no exact indexing method exists, that guarantees sub-linear complexity for the algorithmic scenario that is cross version music retrieval. As shown in this thesis, there are however options that are satisfying in practice.
R-Trees are the superior a approach, if little precision is needed (in our case $P_R < 90\%$). This might seem counterintuitive, since R-Trees are an exact algorithm, but it can be achieved by proper feature reduction. If the dimensionality can be reduced to a value < 10 while the precision is still sufficient for the use-case, R-tree performance benefits greatly. With the optimizations presented in this thesis, the query times become really fast. The performance of those indexing structures however degenerates, if additional data has to be inserted later.
On the other hand, a nearly optimal precision can be reached, by combining high dimensional data with the graph-based approximation approach presented earlier. If adjusted correctly, it sacrifices nearly no precision while gaining a huge performance benefit over exact approaches. This makes HNSWG a valid option, even if high query correctness is needed. The structure is build iteratively, which makes later insertions easy. Insertions could even be processed massively parallel, since very little synchronization would be needed. The synchronization of this graph structure is however a complex topic on its own and deserves further considerations. This could make this algorithm an ideal approach for cloud-computing applications.
Further research on the field of cross-version music retrieval could however change the requirements to indexing fundamentally. For example further improvements in feature reduction could render approximation approaches obsolete.

Bibliography

[1] Pedro Cano, Eloi Batlle, Ton Kalker, and Jaap Haitsma. A review of algorithms for audio fingerprinting. In *Proceedings of the IEEE International Workshop on Multimedia Signal Processing (MMSP)*, pages 169–173, St. Thomas, Virgin Islands, USA, 2002.

[2] Michael A. Casey, Christophe Rhodes, and Malcolm Slaney. Analysis of minimum distances in high-dimensional musical spaces. *IEEE Transactions on Audio, Speech, and Language Processing*, 16(5):1015–1028, 2008.

[3] Peter Grosche and Meinard Müller. Toward characteristic audio shingles for efficient cross-version music retrieval. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 473–476, Kyoto, Japan, 2012.

[4] Frank Kurth and Meinard Müller. Efficient index-based audio matching. *IEEE Transactions on Audio, Speech, and Language Processing*, 16(2):382–395, 2008.

[5] Meinard Müller. *Fundamentals of Music Processing.* Springer Verlag, 2015.

[6] Pierfrancesco Bellini, Ivan Bruno, and Paolo Nesi. Assessing optical music recognition tools. *Computer Music Journal*, 31(1):68–93, 2007.

[7] Christopher Raphael and Jingya Wang. New approaches to optical music recognition. In *Proceedings of the International Society for Music Information Retrieval Conference (ISMIR)*, pages 305–310, Miami, Florida, USA, 2011.

[8] Ana Rebelo, Ichiro Fujinaga, Filipe Paszkiewicz, Andre R. S. Marcal, Carlos Guedes, and Jaime S. Cardoso. Optical music recognition: state-of-the-art and open issues. *International Journal of Multimedia Information Retrieval*, 1(3):173–190, 2012.

[9] Frank Zalkow and Meinard Müller. Learning low-dimensional embeddings of audio shingles for cross-version music retrieval. submitted for publication, under review.

[10] Donald E. Knuth. *The Art of Computer Programming.* Vol. 3: *Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.

[11] J. W. J. Williams. Heapsort. *Commun. ACM*, 7(6):347–348, June 1964.

[12] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[13] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[14] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[15] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In Shamkant B. Navathe, editor, *Proceedings of the 10th ACM International Conference on Management of Data (SIGMOD)*, pages 47–57. ACM Press, 1985.

[16] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ, 23–25 May 1990.

[17] S. Leutenegger, J. Edgington, and M. Lopez. STR : A simple and efficient algorithm for R-tree packing. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 497–507, Washington - Brussels - Tokyo, April 1997. IEEE.

[18] Taewon Lee and Sukho Lee. Omt: Overlap minimizing top-down bulk loading algorithm for r-tree. In Johann Eder and Tatjana Welzer, editors, *The 15th Conference on Advanced Information Systems Engineering (CAiSE '03), Klagenfurt/Velden, Austria, 16-20 June, 2003, CAiSE Forum, Short Paper Proceedings, Information Systems for a Connected Society*, volume 74 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[19] Cheung and Fu. Enhanced nearest neighbour search on the R-tree. *SIGMODREC: ACM SIGMOD Record*, 27, 1998.

[20] S. Berchtold, D. A. Keim, and H.-R Kriegel. The X-tree: An index structure for high-dimensional data. pages 28–39, San Francisco, Ca., USA, September 1996. Morgan Kaufmann.

[21] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. pages 426–435, East Sussex - San Francisco, August 1998. Morgan Kaufmann.

[22] Steffen Guhlemann, Uwe Petersohn, and Klaus Meyer-Wegener. Optimizing similarity search in the m-tree. In Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland 0001, editors, *BTW*, volume P-265 of *LNI*, pages 485–504. GI, 2017.

[23] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst*, 45:61–68, 2014.

[24] J. Kleinberg. Navigation in a small world. *Nature*, 406:845, 2000.

[25] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2018.