Undergraduate Thesis

An efficient external R-tree for very large datasets

Noah Nock

Examiner: Prof. Dr. Hannah Bast
Adviser: Johannes Kalmbach

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data Structures

January 05th, 2024

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Elzach, 3rd January 2024

Place, Date

Signature

Abstract

Modern information retrieval which is used in big search engines has a large impact on our lives. One of the many challenging tasks of information retrieval is spatial quering. Vast spatial datasets can be managed in different ways depending on the application. This work focuses on the task to retrieve every bounding box of a large dataset that intersects with a given bounding box. Among many different data structures which can be used to solve this problem, the R-tree is a popular choice.

The R-tree is a search tree specifically designed for spatial indexing. It works by dividing all elements of a certain node into a number of child nodes so that the bounding boxes of the elements in each child have a minimal overlap. Every node keeps track of the bounding box containing all its elements and continues to divide those elements into child nodes until a specific leaf condition is reached. When given a R-tree, you simply check for intersections of your query with the children of every expanded node starting from the root node. When an intersection is found, you expand the intersecting child node until the leaf nodes of every expanded nodes are reached. Every intersection in a leaf node delivers a result for the query.

In theory this works really well, since a balanced R-tree will answer your query in logarithmic time. However, the actual implementation poses significant challenges in terms of memory management when it comes to large datasets. Since the best-performing R-tree building algorithms require the knowledge about all data points to

work, previous implementations load the entire dataset into RAM and keep it loaded to execute queries. This is not possible for very large datasets.

This work provides the theoretical background of a R-tree implementation, which can build well-balanced trees in a very efficient way, while also being able to operate within a given RAM limit by using only the necessary data on demand. To address this object, this work delves into the fundamental concepts of R-trees, R-tree packing, the actual implementation and the analysis of the implemented R-tree.

Zusammenfassung

Moderne Technlogien aus dem Bereich Information Retrieval werden in großen Suchmaschinen verwendet und haben schon lange einen großen Einfluss auf unser Leben. Eine der vielen anspruchsvollen Aufgaben von Information Retrieval ist die räumliche Suche. Große räumliche Datensätze können je nach Anwendung auf unterschiedliche Weise verwaltet werden. Diese Arbeit konzentriert sich auf die Anwendung, alle Bounding Boxes eines großen Datensatzes zu finden, die sich mit einer gegebenen Bounding Box schneiden. Unter den vielen verschiedenen Datenstrukturen, die zur Lösung dieses Problems verwendet werden können, ist der R-Tree eine beliebte Wahl.

Der R-Tree ist ein Suchbaum, der speziell für die räumliche Suche entwickelt wurde. Er funktioniert, in dem alle Elemente eines bestimmten Knotens in eine Reihe von Unterknoten unterteilt werden, so dass die Bounding Boxes der Elemente in jedem Unterknoten eine minimale Überlappung aufweisen. Jeder Knoten merkt sich die Bounding Box, die alle seine Elemente enthält, und fährt damit fort, diese Elemente in Unterknoten aufzuteilen, bis eine bestimmte Terminalbedingung erreicht ist. Zum Suchen innerhalb des R-trees prüft man einfach, ob sich die gewählte Bounding Box mit den Boxen der Unterknoten jedes ausgewählten Knotens überschneidet, wobei man beim Wurzelknoten beginnt. Wenn ein Schnittpunkt gefunden wird, wählt man den überschneidenden Unterknoten aus und führt dies solang fort, bis die Blattknoten aller ausgewählten Knoten erreicht sind. Jede Überschneidung innerhalb der Blattknoten liefert ein Ergebnis für die Suchanfrage.

Theoretisch funktioniert dies sehr gut, da ein balancierter R-tree eine Suche in logarithmischer Zeit durchführen kann. Die tatsächliche Implementierung stellt jedoch eine große Herausforderung für die Speicherverwaltung dar, wenn es um große Datensätze geht. Da die besten Algorithmen zur Erstellung von R-trees das Wissen über alle Datenpunkte benötigen, laden bisherige Implementierungen den gesamten Datensatz in den Arbeitsspeicher und behalten ihn dort, um Abfragen auszuführen. Dies ist bei sehr großen Datensätzen nicht möglich.

Diese Arbeit liefert den theoretischen Hintergrund für eine R-tree-Implementierung, die sehr effizient balancierte Bäume bauen kann und gleichzeitig in der Lage ist, innerhalb einer bestimmten RAM-Grenze zu arbeiten, indem sie nur die erforderlichen Daten bei Bedarf verwendet. Um dieses Ziel zu erreichen, befasst sich diese Arbeit mit den grundlegenden Konzepten von R-trees, das Packen von R-trees, die eigentliche Implementierung und die Analyse des implementierten R-trees.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The use of spatial searching has increased drastically over the last years due to its many applications. The possibility to search based on locations within a certain dataset has paved the way for many new innovations. With increasing data to work with, it has become a tough task to maintain an efficient spatial index without the use of expensive hardware.
The R-tree, is a spatial index that divides the search space into smaller rectangles. When built correctly, it can make the search inside this space very efficient. Unfortunetely, there is a lack of fast R-trees that can handle datasets of arbitrary size, which makes it unsuited for the use on very large dataset.
Over the past months, I came up with a R-tree implementation that can handle large datasets in a very efficient way without the need of expensive hardware. In this thesis I will go into detail about the theory behind this R-tree.

I begin by introducing the fundamental concepts, which are essential to understand the problem and its solution. In Chapter 3 I go into detail about the idea behind the R-tree. I explain how my algorithm splits tree nodes into children while making sure to never pass a certain amount of working memory. I also address other algorithmic improvements that can be made to accerlate the building and searching time of a R-tree.
After explaining the theory in detail, I give a brief overview of the implementation in Chapter 4, which is followed by an analysis in Chapter 5. In this analysis I evaluate,

how successful my idea for an external R-tree is in regard to real world applications.

Finally, a conclusion is drawn in Chapter 6

2 Fundamental Concepts

This chapter provides a comprehensive overview of the fundamental underlying concepts of R-trees and their use in spatial searching. It begins by introducing the R-tree and its search performance together with a small example. The chapter then delves into different R-tree building algorithms where the focus will lay on the TGS algorithm, which is the underlying algorithm for all further improvements in this work.

2.1 R-tree structure

The R-tree was first introduced by Antonin Guttman in the year 1984 as an improvement of the B-tree for the use of spatial searching accross multi-dimensional spaces [1]. In this work, I will focus on R-trees working with two-dimensional datasets.
For the use of performing a range based search across geo data, you first have to prepare the data. Each of the n data points gets represented by a minimal rectangle, which fully contains the geometrical shape of the data. This rectangle is called a bounding box (BBOX). The leaf nodes of the R-tree contain a bounding box of their data and pointer to the actual data. All non-leaf nodes contain the bounding box which covers all its children and pointer to the children. The following set of properties must be satisfied for a structure to be a valid R-tree:

- Every leaf-node contains between m and M index records, with M being the branching factor of the R-tree and $m \leq \frac{M}{2}$.

- Every inner node contains between m and M children.

- The root node has at least 2 children, unless it is a leaf.

- All leaves must appear on the same level.

In Figure 1 you can see an example of 8 data points, represented by their bounding boxes $R_7 - R_{14}$. Those data points are first divided into the rectangles R_1 and R_2 and those are then divided into $R_3 - R_6$. The result is a R-tree with depth 3 and branching factor $M = 2$. Figure 2 visualizes the corresponding tree form of this R-tree structure.

2.2 Spatial searching in a R-tree

A R-tree can be used for various applications. One of the most popular applications is range search based on a query rectangle. That means that a R-tree can return every bounding box with its corresponding data in the dataset that intersects with a certain rectangle. The underlying concept is similar to other search trees:
You start with the whole dataset as your search space and then reduce the search space by eliminating nodes outside the range of the query until all results or no results are found. Since the elements of a subtree are divided into M subtrees, you can eliminate a subtree if its bounding box does not intersect with the query. That means that you only have to expand the nodes, which could possibly contain your desired elements. In the example visualized by Figure 1 and Figure 2 you could search for the query rectangle R_{11} by starting at the root node, expanding only R_2 and from R_2 only R_5. A check for intersection in R_5 will now return R_{11} as the only result.
The performance of this search highly depends on the way the R-tree is built and the

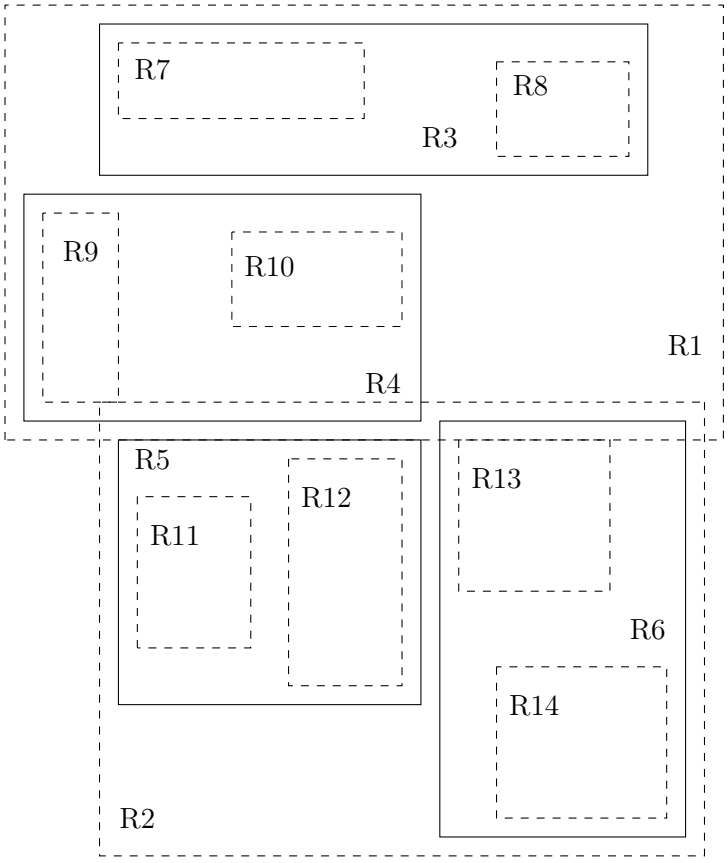

Figure 1: An example structure of a R-tree

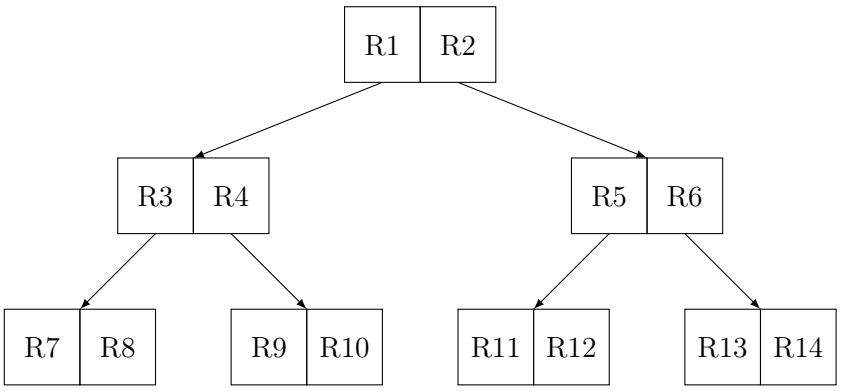

Figure 2: An example of a R-tree

query. If the query is a big rectangle covering all the datapoints or if the sub rectangles in the R-tree overlap a lot, a search could require every node to be expanded to get to the result, since no subtree can be eliminated. The runtime for such a search is

$O(n + number\_of\_nodes)$, which is even worse than just iterating through each element.
However, for most practical queries and R-trees with little to none overlap, the search eliminates most of the nodes and only expands a few paths down to the leaves. A search that expands exactly one path down to a single leaf, has to walk through the depth of the tree once. Therefore, such a search has the runtime $O(depth\_of\_Rtree)$. For a balanced R-tree, which divides each subtree into M subtrees of roughly the same size, the depth of the tree is $\lceil \log_M n \rceil$. This results in a runtime of $O(\log n)$.

2.3 R-tree building

Section 2.2 already stated that the performance of the spatial search in a R-tree highly depends on the quality of the tree. The key part of building algorithm is the node splitting. Each splitting algorithm has its own properties which can be useful depending on the application. Algorithms that produce a less overlapping and more balanced tree usually take more ressources to build. That means that you have a trade off between building and searching time.

2.3.1 Non-packing algorithms

Non-packing algorithms describe the basic R-tree building algorithms that can be used to build a tree and add, delete and update the elements. Some examples of non-packing algorithms are the linear- and the quadratic algorithm by Guttman [1]. Another well known example is the R^*-tree which was introduced by Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider and Bernhard Seeger in 1990 [2]. This improvement of the R-tree delivers better results in terms of searching than the previous ones, but takes a lot of time for building.

2.3.2 Packing algorithms

Packing algorithms perform a technique called bulk loading to build the R-tree. To use bulk loading, you have to know all of your data before building which is why this is intended for static trees, where the elements do not change during the lifespan of the tree. However, this method is faster and results in R-trees with better internal structure. This advantage over non-packing algorithms was shown by Gehreis, Lalande, Loskot, Wulkiewicz and Simonson on the official C++ boost R-tree entry [3]. Because of the fast builds and unbeatable query times due to the interal structure, this technique is suited for many applications. Even for non-static datasets of big size with relatively few inserts, deletes and updates it can be more efficient to build the whole dataset with bulk loading on a ragular basis, rather than modifying the existing tree built by non-packing algorithms. One of the most used packing algorithms is TGS.

2.3.3 TGS Algorithm

The Top-down Greedy Splitting Algorithm (TGS) was first published by Yván J. García R, Mario A. López and Scott T. Leutenegger in 1998 [4]. TGS is a packing algorithm used to bulk load R-trees. As the name suggests, the algorithm performs a top-down approach, where the rectangles on the current node get split in one position along the x- or y-axis. The resulting splits then get split again until the stop condition is reached. The basic split step is shown in Algorithm 1.

When calculating the best split, the algorithm iterates through each possible split along all dimensions and sortings and chooses the split which minimizes a specific cost function. A possible split is at a position in a ordering which is a multiple of S, with S being the maximum number of rectangles per subtree. This property gurantees that splitting the subsets until the stop condition is reached, will produce fully packed subsets of size S. The only exception is the subset containing the last elements, since

Algorithm 1 TGS: Basic split step

```
Let n = number of input rectangles
Let S = maximum number of rectangles per subtree
Let M = maximum number of entries per node
Let f(b_1, b_2) be the cost function
if n ≤ S then
    return (stop condition)
end if
foreach dimension d do
    foreach ordering in dimension d do
        for i from 1 to ⌈n/M⌉ − 1 do
            Let B_0 = BBOX of first i ∗ S rectangles
            Let B_1 = BBOX of the other rectangles
            remember i if f(B_0, B_1) is the current best valued.
        end for
    end for
end for
Split input set and orderings at best position.
```

this subset could contain less rectangles if the input size was not divisible by S. Figure 3 shows a simplified example of choosing a split along an axis. Figure 4 and Figure 5 show the next splits until termination. The example clearly shows how you can achieve accurate and well fitting splits by only testing for certain splits on the x- and y-axis.

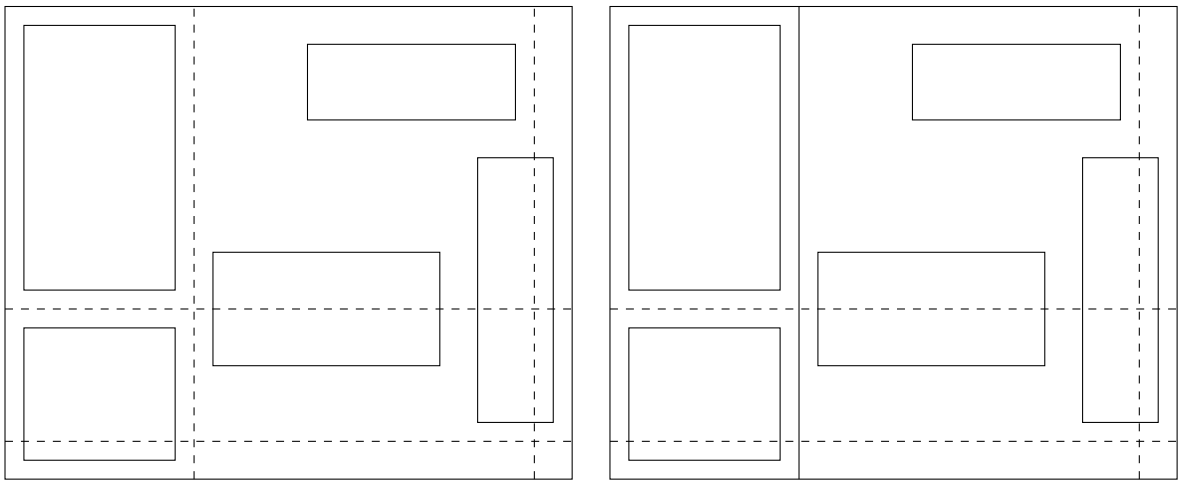

Figure 3: Chose the best first split with minimal overlap

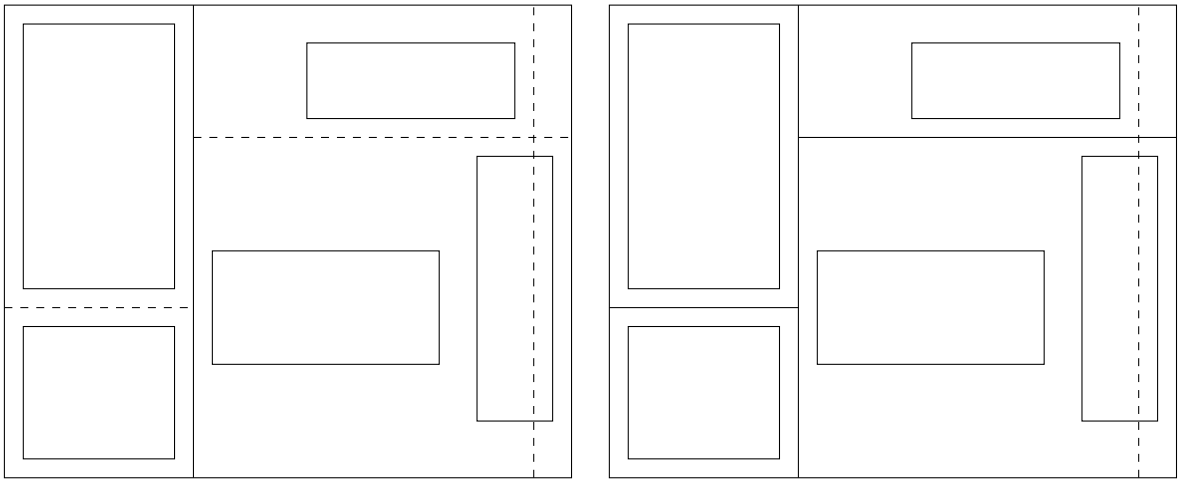

Figure 4: Computing both splits in the left and right subset in parallel

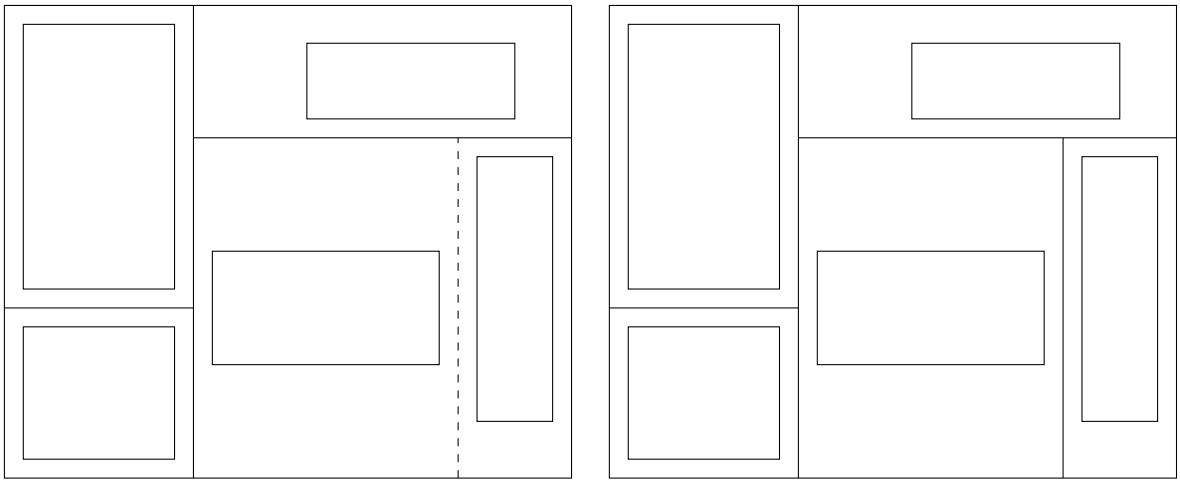

Figure 5

As shown in [4], this approach of building the R-tree results in trees with faster query times than previous packing algorithms. This is why the TGS algorithm is the underlying algorithm for my external R-tree implementation.

3 Approach

In this chapter, I present the approach of developing an external R-tree implementation that can handle arbitrary large datasets in an efficient way.
It starts with the defintion of the main problem followed by two sections which describe the idea behind the solution of this problem. Since my objective is to not only solve the problem, but also do it in an efficient way, the chapter then explains how different aspects of the formal defintion of the TGS algorithm from Section 2.3.3 can be improved. Lastly, I give a suggestion of a suitable cost function for the improved TGS algorithm.

3.1 Problem Definition

When trying to use a R-tree for the purpose of spatial searching in a large dataset, you quickly run into problems with your machine's working memory limit. One of the best R-tree packing algorithms is the TGS algorithm which needs all of the data to be loaded into RAM in any previous implementations. Furthermore, many implementations like the one of the C++ boost library have no native way of serializing the R-tree what requires the whole R-tree to be loaded into RAM during its entire lifespan [5]. For those reasons, any previous implementations are not suited for very large datasets.

3.2 Build and search with a depth first approach

The main goal of this work is to develop a R-tree which can be built and searched through with very little RAM consumption. For this, it is important to decide in which order the nodes are being processed.

Searching in a R-tree is an easy task, because you can perform depth first search on it. This means that the search algorithm first expands nodes along a branch as far as possible before backtracking. Every node is stored individually on disk and only gets loaded into RAM when it gets expanded. Also, the algorithm only has to keep track of the nodes along its path which reduces the memory consumption of traversing the whole tree to $O(depth\_of\_rtree)$ which is $O(\log n)$ for a balanced tree.

For the building, you can use the same approach. First split the elements of the root node into M children. Move one layer down to the first child and split the first child's elements into M children. By repeating this, you will eventually reach a leaf node while having only the single path to this leaf loaded into RAM. The leaf can then be serialized and the algorithm backtracks to its parent. This approach will write a complete R-tree to disk, while again only using space $O(\log n)$.

3.3 Working with external and internal data

The main problem of actually performing the building of the R-tree in a depth first approach like described in Section 3.2 is the splitting of the nodes. Because the TGS algorithm performs bulk loading, its basic split step (Algorithm 1) needs to know all elements to perform a split. However, this does not mean that the whole dataset needs to be loaded into RAM. TGS selects the best split along a number of potential splits. Those potential splits are evenly spaced along the orderings of both dimensions. To be more precise, the potential splits are at the positions which are a multiple

of S. That means that you can keep track of all possible split elements in linear time by looking at each Sth element. Therefore, there is no need to have the data loaded into RAM. The algorithm can evaluate each potential split by only loading the border elements of the potential split into RAM as well as the minimum and maximum element of the ordering. Based on these elements, the two bounding boxes of the resulting subtrees can be created and then evaluated by the cost function. The actual splitting of the dataset can be achieved by loading each element successively from each ordering and assign it to the correct subset and ordering. This procedure gets explained with more detail in Section 3.4.2.
This whole approach for splitting the nodes allows the algorithm to use as little RAM as possible.

The approach also is applicable for use cases where all of the data is loaded into RAM. For this, the algorithm accesses every Sth element of the data in RAM instead of loading it from disk to search for the best split. Also the splitting itself can be done in the exact same way by looping through the elements. That means that it is possible to let the algorithm switch between performing splits on internal data (data loaded into RAM) and external data (data saved on disk).
Based on this behaviour, I developed a way to speed up the building time while staying under a certain value RAM usage. For this, you give the algorithm a limit of RAM it can use (I will call this memory limit). While the algorithm can not perform the splits on internal data without exceeding this memory limit, it uses external data. Once the number of elements of a subtree get small enough, the algorithm can switch to finding the splits in this subtree using internal data. This mix of internal and external data accelerates the building time while also staying under the memory limit.

3.4 Sorting

3.4.1 Problem with the orderings

The most time consuming task in the TSG basic split step (Algorithm 1) is the looping through the different orderings. The idea is to sort all the elements in different ways according to the coordinate of the current dimension. For example, you could sort the rectangles based on their minimum corner, maximum corner, center or other desired anchor points. But as stated in the introduction paper of the TGS algorithm [4], different sortings of the same data make no significant difference in the quality of the R-tree. I will therefore only consider orderings based on the center of the rectangles. After deciding on a split, the algorithm has to split the data into the two subsets. The ordering of the dimension where the best split was found is trivial to split up by taking all elements up to the split for the first subset and all the other elements for the other subset. The other dimensions need to assign the same elements to the correct subsets while also maintaining them sorted. The easiest solution is to copy the subsets of the dimension of the split and sort them again in the other dimension(s). Even when you consider only the center-based ordering, this makes the algorithm run very slow.

3.4.2 Fixing the ordering in linear time

A simple solution for this problem is to assign each element its positions in the orderings of the dimensions. In my case I have two dimension, x and y. That means that I sort the elements first by x. I then assign each element its position $orderX$ which is trivial, since its sorted. Those elements then get sorted by y. After assigning each element its position in this ordering as $orderY$ I receive a finished y ordering with the right order positions. By copying and sorting this by x, I get the correct x ordering as well.

In Table 1 you can see an example of 5 points that are sorted once in x direction and once in y direction. Each point also is assigned its positions in both orderings. If you find the best split between point $(4.3, 3.8)$ and $(5.6, 5.5)$ on the x-axis, you remember the index of point $(4.3, 3.8)$ on the x ordering, which is 3. Splitting the x ordering is already trivial, since the the first 3 elements belong to the left split and the remaining 2 to the right split. You then loop through the y ordering and assign all elements with $orderX \leq 3$ to the left split and all the other to the right split. The result are two new lists which contain the same elements as the two x ordered lists while also maintaining the y ordering.

centerX	centerY	orderX	orderY
2.3	4.6	1	4
2.7	1.4	2	1
4.3	3.8	3	3
5.6	5.5	4	5
5.9	2.3	5	2

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Table 1: Example data sorted in both dimensions (x: left, y: right)

After a split, the absolute number of the position for both orderings is not necessarily representing the actual position of an element anymore. The second element in the y ordering could become the first element in the new split after splitting based on the x axis while still having $orderY = 2$. But this does not influence the algorithm, because the relative order of the indices does not get violated due to assigning them to their split in order.

This is why you only have to assign the order positions once in the beginning. After this initial sorting, every split can be calculated in $O(n)$ time.

3.5 Cost function

The cost function of any kind of R-tree packing algorithm plays an important role, since its quality determines the quality of the whole R-tree. A poorly chosen cost function could lead to bad internal structure and therefore to slow query times. When splitting a set of bounding boxes into two subsets there are many properties one could look at:

- The areas of the resulting bounding boxes in relation to each other.
- The overlapping area.
- The number of elements in both splits in relation to each other.
- Some sort of weighted combination of the previous properties.

For the TGS algorithm, the choice of the optimal cost function is quite easy. Due to the way the splits gets chosen, it is guranteed to receive a fully packed and therefore balanced tree, as explained in Section 2.3.3. For this reason, the number of elements is irrelevant for the cost function. Because the tree is evenly packed, the areas of the resulting bounding boxes also do not provide any useful additional information.
The only significant metric is the area of the overlap. When a query intersects with an overlap, it has to expand all nodes that are part of the overlap. Expanding multiple nodes of the same layer increase the running time of the search drastically, since there is a physical limit of how many threads can handle those nodes in parallel. Expanding two nodes of the same layer one after the other will double the running time compared to expanding only one node, assuming that the following checks and expansions take the same time in both subtrees.

A cost function which returns the total area of the overlap between the two split bounding boxes is therefore simple, yet very effective.

4 Implementation

This chapter provides a brief overview of how my ideas for the external R-tree could be implemented.

4.1 Building the R-tree

4.1.1 Sorting

My implementation of the TGS algorithm requires all of the data to be sorted by both coordinate axes before starting the actual algorithm. Simply sorting the data with any kind of RAM-based sorting algorithm would defeat the goal of being able to restrict the RAM usage of the algorithm.

An external sorting algorithm sorts a dataset in chunks rather than all at once. This allows large datasets to be sorted within a certain memory limit. In my C++ implementation of the R-tree, I used stxxl as an external sorting algorithm [6]. If the dataset is small enough to be handled without exceeding the given memory limit, the data can be sorted using any sorting algorithm.

While sorting the data I assign the order indices like described in Section 3.4.2. This is also a good time to precompute a list of all possible splits, since the assigning of the order indices loops through each elements in the right order for both dimensions.

I can therefore check every item if its position is a multiple of S and save it and its following element to a list called $possibleSplitsX$ (or $possibleSplitsY$). I also make sure to save the minimum and maximum of the dataset to this list. These lists of the elements which represent the boundaries of each possible split can easily be stored in RAM since the number of splits per node is at most M.

The sorting should return the following things:

- All elements of the dataset sorted by x (called $elementsByX$)
- All elements of the dataset sorted by y (called $elementsByY$)
- The list $possibleSplitsX$
- The list $possibleSplitsY$

All four items are packed into a data structure called $OrderedBoxes$.

$elementsByX$ $elementsByY$ can be either lists loaded into RAM or pointer to the location of the elements on disk.

4.1.2 Building depth first

Algorithm 2 shows the depth first building of the R-tree. A $Node$ gets initialized with an unique id and the $OrderedBoxes$ of the elements in its subtree. Based on the $OrderdBoxes$, the TGS function call (Section 4.1.3) splits the elements into M children.

Algorithm 2 Building the R-tree depth first

Let *dataset* be the user given dataset containing bounding boxes with an unique id.
Let M be the branching factor.
Let *stack* be an empty stack.
Let $id = 1$ be the increasing id of non-root nodes
$orderedBoxesInitial \leftarrow SortAllElements(dataset)$
$root \leftarrow Node(0, orderedBoxesInitial)$
Add *currentNode* on top of *stack*
while
    stack is not empty **do**
    $currentNode \leftarrow stack.pop()$ ▷ Take the top element of the stack as the currentNode and remove it
    if #elements in currentNode $\leq M$ **then** ▷ Reached a leaf node
        SaveNode(currentNode)
    else
        $S \leftarrow \lceil \frac{\#\text{elements in currentNode}}{M} \rceil$
        $orderedBoxesOfSplits[] \leftarrow TGS(currentNode.OrderdBoxes, M, S)$
        foreach element *split* of $orderedBoxesOfSplits[]$ **do**
            $newNode \leftarrow Node(id, split)$
            $id \leftarrow id + 1$
            Add *newNode* on top of *stack*
            $currentNode.AddChild(newNode)$
        end for
    end if
end while

4.1.3 TGS

Algorithm 3 shows the basic behaviour of the TGS algorithm implemented as a recursive function. The actual splitting step is happening in the $PerformSplit$ function which is described in Section 4.1.4

4.1.4 PerformSplit

Since *possibleSplitsX* and *possibleSplitsY* are already pre-computed. It is trivial to find the best split by looking at the resulting bounding boxes of each possible split

Algorithm 3 TGS: The improved TGS implementation as a recursive function

Let $currentOB$ be the OrderedBoxes of the node that called TGS
Let M be the branching factor
Let S be the maximum number of elements per subtree
if #elements in currentOB $\leq S$ **then return** currentOB
end if
$splits \leftarrow PerformSplit(currentOB, M, S)$
Call TGS again for both splits:
$resultLeft \leftarrow TGS(split.left, M, S)$
$resultRight \leftarrow TGS(split.right, M, S)$
return $resultLeft$ and $resultRight$

and picking the split that minimized the cost function.

Actually splitting the data is done by performing the procedure described in Section 3.4.2.

Similar to Section 4.1.1, the new $possibleSplitsX$ and $possibleSplitsY$ can be computed, while the data gets split.

4.2 Searching in the R-tree

Algorithm 4 shows the depth first search algorithm that performs the searching of my R-tree. To keep the memory usage down, each node only knows the id and bounding box of its children. To actually get the child node, it needs to be loaded from disk by $LoadNode$.

Algorithm 4 Depth first search of the R-tree

```
Let query be the bounding box of the search query.
Let stack be an empty stack.
Let results be an empty list.
Add currentNode on top of stack
while
    stack is not empty do
    currentNode ← stack.pop()                    ▷ Take the top element of the
                                                   stack as the currentNode and
                                                   remove it
    foreach child of root do
        if intersects(query, child.boundingBox) then
            if child is leaf then
                Add child to results
            else
                newNode ← LoadNode(child.id)     ▷ Load whole node of the child
                                                   from disk
                Add newNode on top of stack
            end if
        end if
    end for
end while
```

5 Analysis

In Chapter 3 I explained how my proposed R-tree works. In this chapter I will focus on an analysis of the R-tree. The analysis inputs building and searching time meassurements on different datasets and branching factors, the runtime of the building algorithm in relation to the number of elements n and a comparison to the widely used R-tree of the C++ boost library [5]. The results of the analysis will give you a better understandment of the R-tree and its efficiency.

5.1 Background

5.1.1 Sample data

For the analysis of the R-tree I used three different datasets. The first file consists of real data. It contains the bounding boxes of all objects in switzerland that are recorded by OpenStreetMap [7]. This file contains 33266131 bounding boxes and I will call it SwissFile.
The other two datasets are synthetic data files. The generation of these synthetic files was inspired by Azzam Sleit and Esam Al-Nsour [8]. Both files consist of 1,000,000 bounding boxes which were randomly generated by a C++ program. The maximum area of any bounding box was set to 0.001× the area of the world space, which is also switzerland. One file generated these entries normally distributed and the other file generated the entries uniformly distributed. I will call the first file NormFile and

the second UniFile.

5.1.2 Sample queries

To analyze the searching times of the R-tree, I chose different bounding boxes as sample queries that intersect with various areas of switzerland. The size and location of these queries vary a lot so that different aspects about the R-tree and its structure get tested. Since the world space of NormFile and UniFile was set to be switzerland, the queries will also be applicable to these datasets.
To compare the searching times of R-tree configurations, I meassure the time to complete all the sample queries and divide them by the number of results multiplied by 1000. With this metric I can compare the time per 1000 results of two R-trees. The metric might not be a good choice for comparing searching time between two datasets since it depends on the amount of entries found within the query. But it is very representative when comparing two R-trees or R-tree configuration on the same dataset.

5.1.3 Hardware

Every experiment in this chapter was run on an Apple MacBook Pro with M1 chip and 16GB of RAM. To get a better understandment of the performance compared to any non-external R-tree building algorithm, all of the experiments were completely performed in RAM, except Section 5.3.

5.2 Impact of branching factor M

The only adjustable parameter of my R-tree implementation is the branching factor M. It determines the number of children of any fully packed node in the R-tree. It is therefore interesting to see, how the performance of both building and searching changes with different branching factors.

In Figure 6 you can see the building and searching times of my R-tree with different M. The values for M are all powers of two and range from 2^1 to 2^{11}. The building times of SwissFile are plotted in another graph than the times of UniFile and NormFile, since SwissFile takes much longer to build due to its size.
It is clear to see that the optimal choice for M is somewhere between 16 and 128 in terms of building and searching. Additionally, the memory consumption of the R-tree building algorithm depends on M since a higher branching factor means that more children need to be loaded into RAM, when a node gets expanded.

5.3 Building externally

Even though most of the analysis assumes to have no strict memory limit, it is important to have a look on the performance when a tight memory limit is given. In this case, the initial sorting happens externally and the improved TGS algorithm accesses the data from disk rather than directly from RAM. Because of that, the runtime of the algorithm itself only increases by a constant factor which is determined by the speed of the machine's disk. The increase in the time it takes to sort the data depends on the choice of the external sorting algorithm. Finally, the memory limit itself has the most influence in the speed of the building time, since subtrees automatically get handled internally once the memory limit can be adhered to.

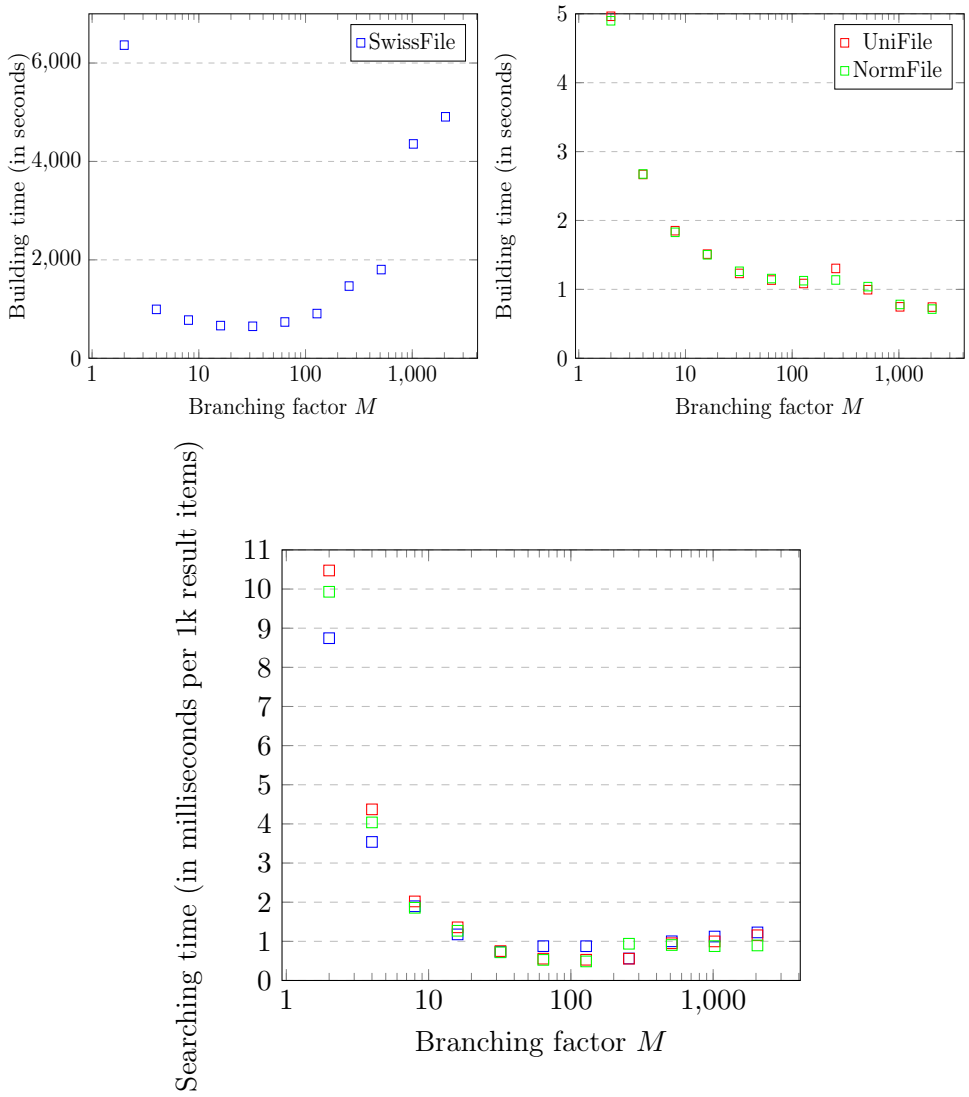

Figure 6: Building and searching with different M on all three datasets

Figure 7 shows the building time of the R-tree on the data from SwissFile with a memory limit of 500MB compared to the same R-tree built with no memory limit. The size of SwissFile is 1.33GB and the interal build needs as much as 4 times the file size of RAM. This means that the internal build of the SwissFile takes up to 5.32GB of RAM usage while the external build does not exceed the limit of 500MB. Both builds were performed with $M = 64$. In this experiment, it takes only a few even splits for the bounding boxes of a subtree to become small enough so that the

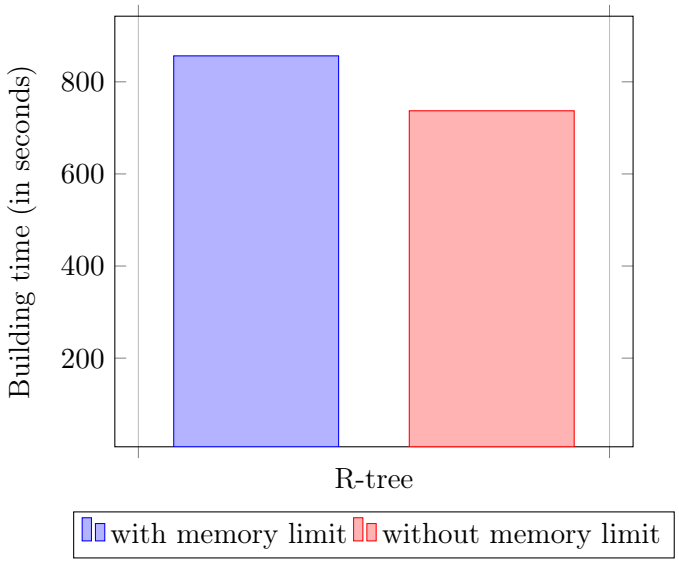

Figure 7: Comparison of building a R-tree with and without a memory limit

memory limit can be adhered to. For this reason, only the first few splits have to be performed on disk and the many smaller splits afterwards can be done in RAM. This results in the building time of the R-tree with memory limit being only a bit longer than the building time without a memory limit.
In general, the higher levels of the R-tree perform only a few splits, which drastically decrease the size of the resulting subsets. The lower levels, which have to perform the most splits and therefore have to access the data more often are a lot smaller and will probably fit into the memory limit.

This behaviour applies to most practical use cases of a R-tree, since you often already have a reasonable amount of RAM available for the task of building and a dataset that is even bigger than that. That means that it is not possible to build the whole R-tree in RAM but with the use of my R-tree the build can succeed while having only a reasonable increase in building time compared to the theoretical RAM building time.

5.4 Runtime

Based on the information provided in Chapter 3 the R-tree seems to have a good runtime. Every aspect of the improved TGS algorithm and the node building is performed in linear time. Only the initial sorting of the input data has a runtime of $O(n \log n)$ which makes the total runtime $O(n \log n + O(n)) = O(n \log n)$.
Because of this deterioration of the asymptotic runtime, it is interesting to see how the actual relation between the data size n and the time to build behaves.

In Figure 8 you can see how the time to build the R-tree increases with more bounding boxes as input. The data used for this experiment is from UniFile. As

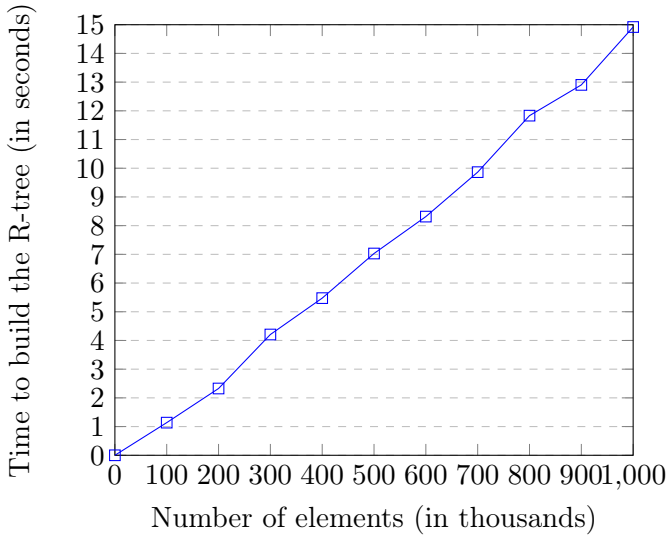

Figure 8: Relation between building time and number of elements of R-tree

you can see, the logarithmic factor of the asymptotic runtime has a very little impact on the actual runtime, since the building time increases mostly linear with increasing n. This makes the R-tree suitable for the use of very large datasets.

5.5 Comparison to C++ boost

In the previous sections of this chapter I presented an analysis of my R-tree and its unique mechanics in regard to different datasets and configurations. In this section I will compare the building and searching times of my R-tree to the R-tree of the C++ boost library to get a better view of its relevance for real world applications. The boost R-tree is highly optimized and the most popular R-tree implementation for C++ applications. Its comparison to my R-tree therefore delivers a suited metric for evaluating the usability of my R-tree on real data.

Figure 9 shows the building and searching times of both R-trees. The building was done for $M = 64$ and the building times are being plotted into two graphs because SwissFile is a much larger dataset than UniFile and NormFile which results in longer building time. The results from Figure 9 indicate that the boost algorithm produces the R-tree faster than my algorithm. For the large SwissFile, boost is around 60% faster while for the smaller UniFile and NormFile boost is only around 45% faster than my R-tree algorithm. This relative increase in building time over the file size can be explained by the logarithmic factor in the asymptotic runtime of my R-tree as explained in Section 5.4. The difference in building times of both algorithms is not a surprising results since the boost algorithm is highly optimized for building the R-tree in RAM, while my algorithm is designed to build the tree in RAM and/or on disk.

But the building time is not the most important metric. In general, a longer building time of a search index is acceptable as long as the searching time is fast. In the bottom graph of Figure 9 you can see the searching times in both R-trees. For the SwissFile, the boost R-tree is only $\sim 9\%$ faster than my R-tree. For the UniFile, the boost R-tree is $\sim 29.62\%$ faster than my R-tree. For the NormFile however, my R-tree is $\sim 46.31\%$ faster than the boost R-tree. This is because the boost R-tree somehow searches $\sim 60.74\%$ faster on the UniFile than on the NormFile, while my

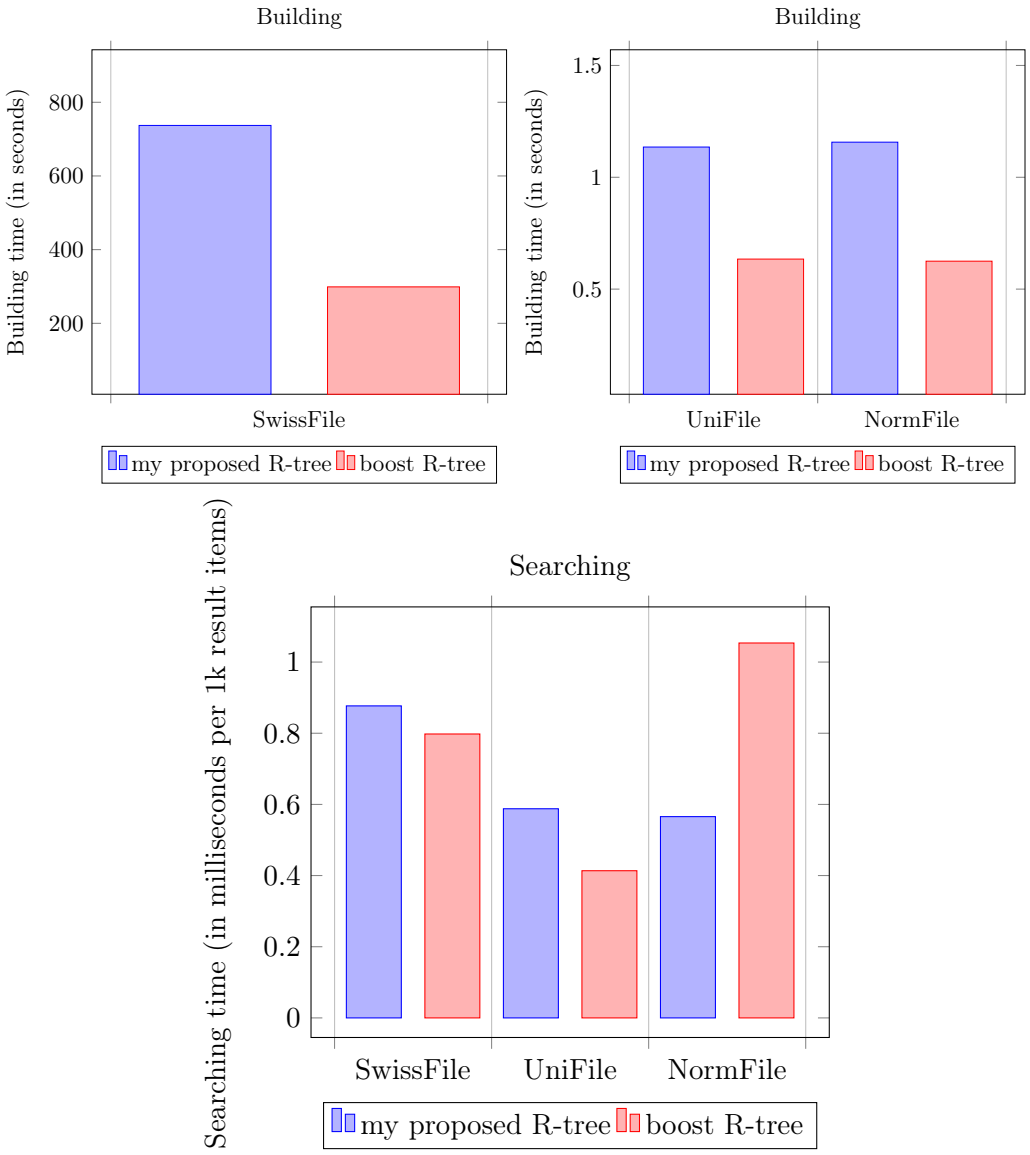

Figure 9: Comparison of my R-tree to the C++ boost R-tree

R-tree stays consistent with its searching times over these two datasets.

The results of this comparison are very satisfactory. The idea of my R-tree was not to outperform the established and well optimized previous R-trees. The idea rather was to maintain a similar and consistent performance while being able to handle much larger datasets with little RAM available.

6 Conclusion

The objective of this thesis was to present and analyse an efficient and external R-tree, which can be used to handle very large datasets on ristriced memory.

After introducing the underlying concepts of the R-tree as a spatial search index in Chapter 2, I explained the problem of previous R-tree implementations and delivered an approach for the solution in Chapter 3. This approach dived deep into the inner workings of the widely used Top-down Greedy Splitting Algorithm where I made several adjustments to the original algorithm to improve the runtime. I also introduced the concept of using the data externally to minimize the use of RAM during R-tree construction. Chapter 4 then showed how one could approach the implementation of the suggested R-tree

In Chapter 5 I conducted an analysis which confirmed the usability of my R-tree on real world applications. I started by analysing the impact of the branching factor M on the performance of the R-tree, in order to choose the suitable M for the tasks in the following experiments. The main finding of these experiments was that my R-tree is very much suited for the use of very large datasets. As demonstrated in Section 5.4 the R-tree has a reasonable time increase with the increase of data points. Futhermore, the addtional effort of externalizing the data is reasonable and the overall speed of the R-tree is within reach of the state of the art C++ boost R-tree.
Based on the results of this work it can be concluded that my proposed external R-tree is a good choice when computing a large spatial index.

7 Acknowledgments

First and foremost, I would like to thank Johannes Kalmbach for advising me during the span of this work.
I also want to thank Prof. Dr. Hannah Bast for agreeing to be my examiner.
I would also like to give special thanks to my family and friends for the support during the span of this work and my time at the University of Freiburg.

Bibliography

[1] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, p. 47–57, jun 1984.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," *SIGMOD Rec.*, vol. 19, p. 322–331, may 1990.

[3] B. Gehrels, B. Lalande, M. Loskot, A. Wulkiewicz, and L. Simonson, "Boost r-tree algorithm." `https://www.boost.org/doc/libs/1_84_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html` [Accessed: (15th December 2023)].

[4] Y. J. García R, M. A. López, and S. T. Leutenegger, "A greedy algorithm for bulk loading r-trees," pp. 163–164, 1998.

[5] B. Gehrels, B. Lalande, M. Loskot, and A. Wulkiewicz, "Boost r-tree documentation." `https://beta.boost.org/doc/libs/1_82_0/libs/geometry/doc/html/geometry/reference/spatial_indexes/boost__geometry__index__rtree.html` [Accessed: (15th December 2023)].

[6] "Stxxl documentation." `https://stxxl.org` [Accessed: (15th December 2023)].

[7] "Openstreetmap." `https://www.openstreetmap.org` [Accessed: (20th December 2023)].

[8] A. Sleit and E. Al-Nsour, "Corner-based splitting: An improved node splitting algorithm for r-tree," *Journal of Information Science*, vol. 40, pp. 222–236, 04 2014.