Utilizing the Linked Directional Properties of Trajectories in Querying Spatial Data

Erik Mjaaland Skår

February 9, 2023

Abstract

insert abstract

Contents

List of Figures

List of Tables

0.1 Motivation

In the recent years, IoT has become more and more prevalent in our society. This has lead to an increase in the data generated every day, especially when it comes to spatial data. Companies like Strava have earned millions of dollars generating, storing and aggregating spatial data for their users [1]. This data is often presented as trajectories in the form of linked directional points in two or three dimensions showing the path of the user. Spatial databases have long been able to effectively query data on their spatial properties, but there does not seem to be much research on utilizing the linked directional properties of the trajectories for querying the data. This defines the motivation of this project: make queries of linked directional spatial data more effective.

0.2 Goal

The goal for this thesis is to test if the data structure presented is a viable data structure for storing spatial data with linked directional properties.

0.3 Objectives

To reach the goal, one would have to fulfill a set of objectives, which can be seen as subgoals for this thesis.

1. Implement the data structure in a programming language

2. Setup a benchmarking environment complete with baseline performances in other data structures to compare it with.

By completing these objectives the results can be compared and discussed, answering if the data structure compares against existing alternatives in querying spatial data with directional propertes.

0.4 Chapter overview

This paper will be divided into chapters, with the first chapter containing the background where existing data structures and indexing methods will be presented, discussed and compared. The context and gap in knowledge will be explained more in-depth. Related work will also be presented here. Chapter three encompasses the methodology used in this project. Here the new data structure will be presented, as well as how the benchmarking has been conducted, including the choice of technology and comparison data structures. In chapter four we will present the results of the benchmarking and look at how they stack up against the predicted outcome. Chapter five contains future work and chapter six contains the conclusion.

Disclaimer: chapter one may contain text already used in *IT3915* as that is the preliminary project for this thesis and is closely related.

Chapter 1

Background

Some context and knowledge is required for understanding the task at hand. It is assumed that the reader has some knowledge on the topic of spatial data, but a brief intro will be given in this chapter. For a more detailed explanation of spatial data and its underlying data structures, materials such as *The Design and Analysis of Spatial Data Structures* is recommended [2].

1.1 Spatial Data

Spatial data is simply data that contains spatial properties in some shape or form. These can include, but are not limited to, points, lines, rectangles, regions, surfaces and volumes [2]. There can be additional data present, but that is not a requirement. One of the most common data types to pair with spatial data is temporal data. This is known as spatio-temporal data, and is used extensively throughout the world. One interesting effect of storing spatial and temporal data together is that you can create trajectories, essentially drawing a line from the starting point, through all the intermittent points and to the end point. But how does one know when to generate a point? The two most common methods are to either save the location at a set time interval, or to save the location after a set distance interval. Another way, commonly used by apps such as Google Maps, is to save the location when the user stops and spends time close to a point of interest. These points of interests can be anything from statues to cafés. From this you can see how long each stop is, the average movement speed, where the user spends most of its time and generate patterns in user movements.

To limit the scope of this project, we will not focus on the temporal part of the data. We will instead only look at the spatial properties, such as the coordinates, but we can use the temporal properties to induce an ordering the the points, creating a linked directional property to each trajectory. We will also not focus on spatial data with more than two dimensions or look into different types of databases.

1.2 Dataset

There are quite a few different datasets publicly available that would be suitable for this project. To determine which datasets were relevant we would need to look at different aspects and see if they have everything we need. These aspects include:

- **Spatial properties.**
  For the dataset to be relevant at all it needs spatial properties. Either GPS coordinates or another X,Y system.
- **Be based on real-world data.**
  Real-world data makes the data points more genuine and realistic. Generating a new dataset could also be used, but that might not take into account different aspects one would get with real-world data.
- **Some kind of ordering.**
  This can either be through having links between data points, implicit ordering from the rows of the file, or inductive ordering through timestamps.
- **Sufficient data.**
  A dataset of a few kilobytes is not going to be enough data to make sure that the evaluation works. A few gigabytes would be optimal, but a a little less than one gigabyte should be sufficient as well.

1.2.1 Dataset Providers

There are several companies that use spatial data. Companies such as Strava have built empires using, manipulating and mining spatial data. Initially, Strava was looked at in case they had any anonymized datasets available. Being that largest fitness tracking app in the world, they would surely have some data available, right? Unfortunately, not without some caveats. To access their data, one would have to use their API. This is an open API, but it would require explicit consent from a lot of users to get a feasible amount of data. In addition, Strava does impose quite a lot of restrictions on how the dataset is used and distributed. These restrictions posed too much of a hassle to make it a viable option.

Kaggle is a community for data science and machine learning. They regularly host competitions where users are provided a dataset and have to create the best algorithm to solve a problem. These datasets are not created by Kaggle themselves, but rather provided by companies, users or researchers. Throughout Kaggle's history there have been several competitions, and as a result there are a lot of datasets available for download. As these are used for different competitions they consist of all types of data and sizes. Some datasets are only a few kilobytes in size whilst others are several gigabytes. Fortunately, they have a search functionality that allows you to filter datasets based on several different factors, including size, date, data type, and others. Table 1.2.1 shows some of the datasets that were considered. In the end, the Taxi Trajectory Prediction dataset was chosen. It fulfilled all the requirements, whilst also being really descriptive and easy to understand. This dataset was also recommended by Svein Erik Bratsberg during the initial meeting for this project.

1.3 Existing Data Structures for Spatial Data

Spatial data is not a new concept. As early as the 1960's research began on studying geographic information systems (GIS) [3]. However, as the amount of spatial data has increased significantly over the past few years, spatial data has become more relevant than ever. The most common way to store and work with spatial data is through spatial databases. These are databases that are specifically designed to store and perform queries on spatial data. Although they are commonly referred to as spatial databases, none of the commercially available and well-known spatial databases are exclusively spatial-oriented. They are all plugins and extensions built on top of existing databases, or simply a small part of a DBMS. Perhaps the most

Table 1.1: Datasets that were considered from Kaggle.

Name	**Spatial Properties**	**Real-World Data**	**Ordering**	**Sufficient Data**
Microsoft Geolife GPS Trajectory Dataset	Yes	Yes	Timestamped	Yes - 1.67 GB
T-Drive Trajectory Dataset	Yes	Yes	Timestamped	No - 803 MB
GPS data from Rio de Janeiro buses	Yes	Yes	Timestamped	Yes - 6 GB
Wrocław public transport	Yes	Yes	Timestamped	Yes - 3.56 GB
ECML /PKDD 15: Taxi Trajectory Prediction	Yes	Yes	Polyline	Yes - 533.7 MB (compressed)

known spatial database is PostgreSQL with the PostGIS plugin.

For a spatial database to store and query spatial data there needs to be some properties present. The most important one being the spatial indexing method, as that is essential for querying spatial data effectively. Normal databases use indexes such as B^+-trees, but these are no good for querying spatial data. Most spatial database systems allow for using different indexing methods based on what the user needs, but the most common indexing method is R-trees or any of its variants. The different indexing methods all take different approaches and allows for different amount of dimensions to their data. Some prioritize dynamic data, data that will change during the usage, and some prioritize static data. When prioritizing dynamic data, an index method must easily be recreated or updated to accommodate the changes. If it takes a long time to adapt to the changes, the queries to the data will be slow and will result in a bad experience for the user. If one knows that there is not going to be any changes to the data, one can go for an indexing method that might use more time to generate the index, but has a faster time for querying the data after the index is up and running.

There are two main categories for the data structures used by spatial indexing methods: space-driven and data-driven structures. Space-driven structures divide the entire space into zones based on the data present. Some examples of this include quadtrees and kd-trees. Data-driven structures on the other hand do not divide the entire space, but rather uses the data to only create zones that are encompassing the data, using only the minimum amount of space required. This category contains R-trees and all its variants.

Different indexing methods can also be faster for some types of queries and slower for others. The *Open Geospatial Consortium* has defined some predicates which indicate how geometries interact with each other [4]. These, and others, are often built into the spatial databases such that users do not have to write custom functionalities for them.

1.3.1 Quadtrees

Quadtrees are a space-driven data structure used for indexing two-dimensional spatial data. They work by dividing the space into four squares, and then dividing those again until you only encompass a set amount of points in each

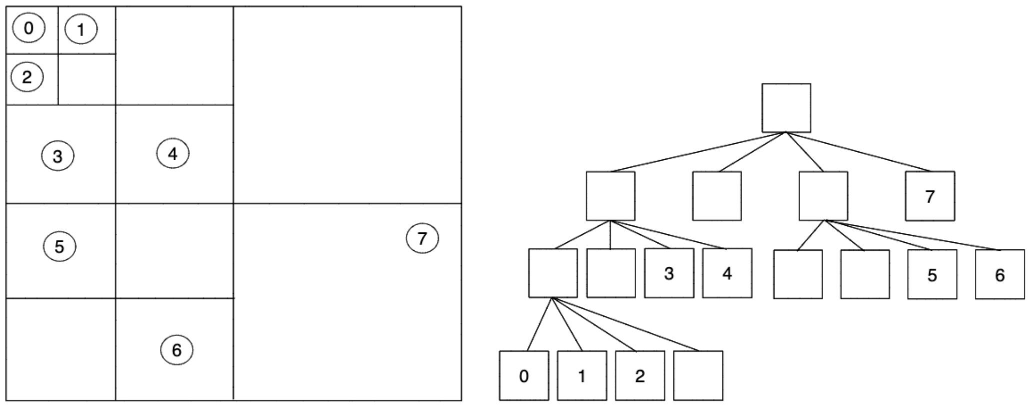

Figure 1.1: A PR quadtree with seven points and a maximum of one point per cell represented as both a grid and a tree.

square. There are several variations of quadtrees as they are used for different use cases such as image compression, storing polygons and several others. The different variations include point-, region, point-region-, PMR- and XBR quadtrees [5]. In figure 1.1 you can see a point-region (PR) quadtree represented as both a grid based on the position of the points as well as a tree. For connected directional spatial data the most relevant quadtrees to look at would be the point-region-, PMR- and XBR quadtrees.

1.3.2 R-trees

R-tree is a data-driven structure invented in 1984 by Antonin Guttman that is used for indexing multi-dimensional data [6]. The concept is closely related to B^+-trees, but the data is grouped by its spatial properties. It works by grouping items by using *minimum bounding rectangles* (MBR). MBRs are the smallest possible rectangles one can create for a set of objects whilst enveloping their coordinates. By grouping data with MBRs, one can easily query the data on their spatial properties by looking at the MBR and seeing if the rectangle satisfies the constraints of the query. With R-trees there are rectangles *all the way down*, meaning that several MBRs are grouped by a larger MBR with the data contained in the leaf nodes.

Throughout time there have been many proposed variations of R-trees, with a few of them being used extensively. R*-trees were first proposed in

1990 and improves on the original concept. In essence, the R*-tree uses more computational power to create trees with less overlap and smaller MBRs. This allows for better query performance than the original R-tree. The R$^+$-tree does not allow overlapping MBRs on the same level, causing greater performance for point queries, but worse performance for window queries due to storing duplicates rather than overlapping MBRs. In addition to these variants there are several others that do their own optimizations and concessions in order to improve the performance in some regard. A more detailed and comprehensive list of R-tree variations can be found in [7].

1.3.3 Comparing R-trees and Quadtrees

A study done by Oracle in 2002 found that R-trees are faster than quadtrees for querying of almost all types. R-trees were faster for *anyinteract, inside, contains, touch, coveredby, covers, equals, overlapbydisjoint*, and *overlapby-intersect*. However, quadtrees were better for *touch, overlapbydisjoint,* and *overlapbyintersect* only when using larger query windows. They also found update times to be better for R-trees and storage requirements similar for both structure, but only when using points. For everything else R-trees outperform quadtrees. Essentially, they recommend using R-trees unless you are creating

> "[...] update-intensive applications using simple poly- gon geometries, high concurrency update databases, or when specialized masks such as touch are frequently used in queries." [8]

They also found that using quadtrees required a lot of fine-tuning to reach optimal performance. This is not required for R-trees and provides R-trees with another advantage.

Out of all this, we can gather that the different data structures have their own advantages and use-cases. For this project it can be beneficial to keep both in mind look at how the different aspects can be used for making an optimal data structure for querying linked directional spatial data.

1.3.4 Space-filling Curves

Space-filling curves (SFCs) are mathematical functions used to map multi-dimensional data points to a single dimension whilst preserving the spatial

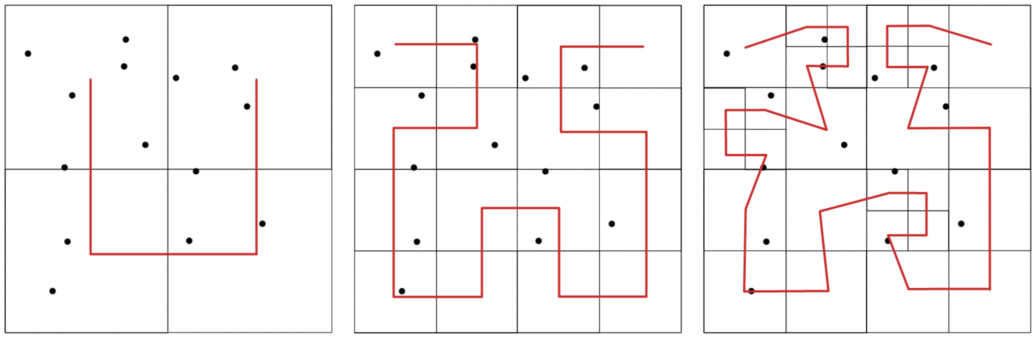

Figure 1.2: Hilbert space-filling curve used with a quadtree. Note how the depth of the quadtree affects the order of the curve.

relationships of the data points. They only access each location once, and never crosses itself [9]. SFCs have several applications including image compression, computer graphics and spatial indexing.

There are several different variations of SFCs, with the most well-known being the Hilbert curve which was described in 1891 by David Hilbert [10]. It is preferred over other curves, such as the Morton curve, as it better preserves the spatial relationships between points [2].

Space-filling curves can be utilized in spatial indexing either alone or accompanied by either a space-driven or data-driven structures such as the ones introduced previously. The Hilbert R-tree is a famous variation of the R-tree that utilizes the Hilbert space-filling curve to better preserve the locality of the data points than the original R-tree in case of splits. Space-filling curves can also be used in conjunction with quadtrees as can be seen in figure 1.2.

1.4 Linked Lists

Linked lists are a common data structure in computer science and are used in several. It consists of nodes linked together with pointers pointing from one node to the next. They do not use indexes as the nodes can be stored anywhere in memory, with only the pointers telling where the data is. Because of this, one can not access any given node directly, but you rather have to traverse the list from start to finish to read items. Why are linked lists relevant? Linked directional data is essentially just linked lists containing spatial properties. There are nodes which are logically connected to other nodes through some kind of ordering.Utilizing linked lists for spatial data is not an original thought and has been used in MX-CIF quadtrees previously all the way back in 1982, but luckily in a different way from what we are proposing here [2].

1.4.1 Expanding Linked Lists to Accommodate Spatial Data

Linked lists work fine, but up against data structures as R-trees they are quite simple. Using standard linked lists will not use the spatial component of the data, which is needed to make efficient queries. To differentiate this data structure from linked lists, we will refer to it as *linked MBR lists*. This name is temporary and should be revisited at a later date to find something more fitting that takes into account all aspects of the data structure.

1.4.1.1 Adding MBRs to Nodes

To expand the linked list we can add an MBR to each node, which also encompasses all of its children. For this to work, one would have to have a doubly linked list as the MBR will have to be created from the bottom up. A visualization of the data structure can be seen in figure 1.3.

Adding MBRs to each node will allow for easily determining if a query window intersects the MBR of the head, which tells us from the first node if the query can or can not be satisifed from this list. Also, by having the MBR of all later nodes in each node, we can terminate the traversal when the MBR no longer intersects a query window or encompasses a query point. This works well when the query window is close to the head of the list, but if only the last node is within the query window this would still result in traversal of the whole list.

1.4.1.2 Skipping Nodes Based on Distance

By adding MBRs to each node one can more efficiently query on the spatial properties of the data. However, this does not utilize the linked aspect of the

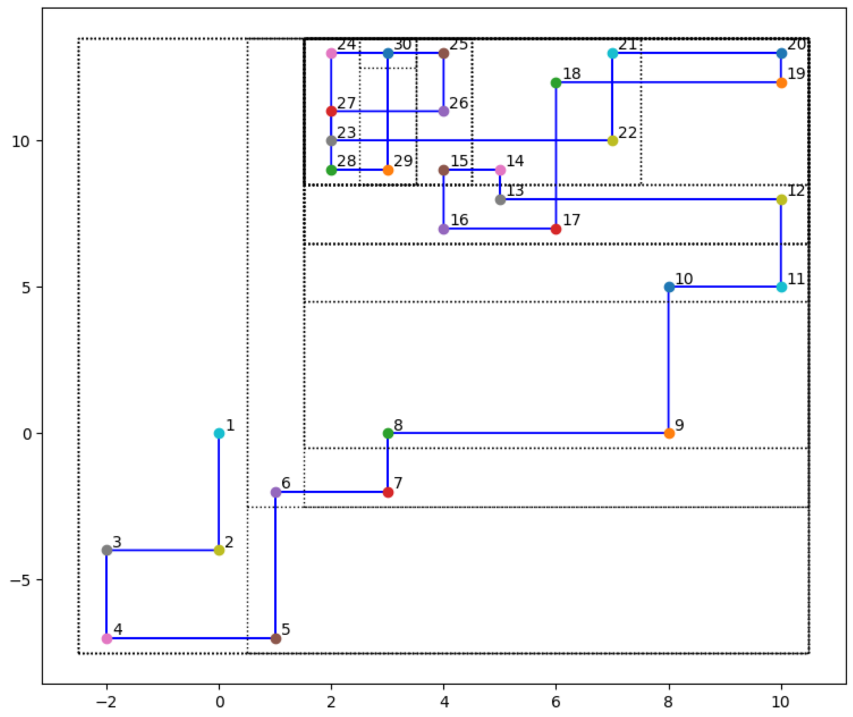

Figure 1.3: Visualization of MBRs added to each node. The MBRs are shown in dotted rectangles.

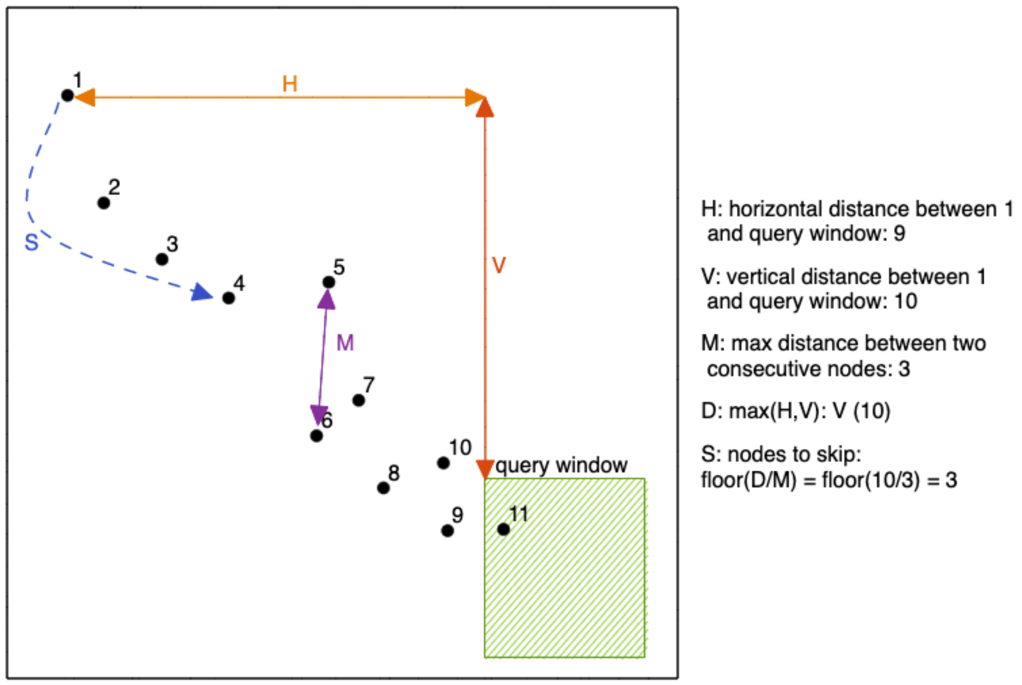
1
H
2
S
3
4
5
V
M
7
6
10
query window
8
9
11
H: horizontal distance between 1 and query window: 9
V: vertical distance between 1 and query window: 10
M: max distance between two consecutive nodes: 3
D: max(H,V): V (10)
S: nodes to skip: floor(D/M) = floor(10/3) = 3

Figure 1.4: Visualization of node skipping based on distance. Note that one would still have to traverse from node 1 $\rightarrow$ node 2 $\rightarrow$ node 3 $\rightarrow$ node 4, but one would not have to check their spatial properties or MBRs.

linked list. When querying one would have to traverse each node, checking if the node itself is within the query window, and then checking if the MBR intersects the query window. These are not heavy computational checks, but they do add up over a large amount of nodes. One way to skip these checks, only traversing until a specific node, could be done by calculating the maximum distance between any two consecutive nodes in the list M. By knowing the maximum distance that occurs between two consecutive nodes within the list, one could look at the head node, check the distance from it to the closest point of the query window D, and if $D > M$ one could traverse $S = floor(D/M)$ nodes without checking neither the node's location or their MBR. After having traversed S nodes, one could check the location of this node and do the calculation again. If $D > M$ still holds true, one could traverse S nodes again without checking. The cycle repeats until $D > M$ is false.

1.4.1.3 Skipping Nodes Based on MBR Change

One unfortunate aspect of the linked MBR list can be seen in 1.3. Node 1 and node 4 have the same MBR. Node 1 is laterally dominated by node 4 and node 20 and longitudinally dominated by node 4 and node 30. This would store the same MBR on several nodes, causing superfluous lookups. One way to prevent these redundant checks would be to add pointers between nodes where there are no changes in MBRs upon creation. This would allow for traversing the nodes, checking them against the query window, but without having to check the MBR for each one.

For instance, in figure 1.3 one could add a pointer from node 1 $\rightarrow$ node 4 as they both have the same MBR. The same could be done from node 12 $\rightarrow$ node 17 as the MBR does not change between those either. Adding these extra pointers would require expanding the data structure even further from a linked list, adding an optional pointer. For the example used here, it would not save much time or many steps, but for a large trajectory where the MBR does not change for a significant amount of nodes, this could save some time.

1.4.1.4 Indexing

Currently, if you were to query these lists you would have to check the MBR of each individual list as an initial filtration. This is obviously bad for performance, and an indexing method would be needed for mitigating this. Trees are effective as they effectively prune large amounts of data on the first split, with balanced binary trees pruning half the data initially. To prune data early we need to divide the search space into zones. We have already seen that existing methods use space-driven and data-driven structures for dividing the search space. Our current implementation is data-driven as the MBRs are generated out of the data itself. One idea would be to implement a space-driven structure on top of the linked MBR lists. This would create a hybrid structure where the initial splits would be done on the search space until there is only one MBR in each reactangle and further divisions would be done by the MBRs.

The big question when implementing the space-driven aspect of the structure is what algorithm is to be used. Although kd-trees are generally preferred for static data, the fact that splits the data in two, as it is a binary tree, makes it conflict with the MBRs as that would split the content of the MBR into separate buckets. Because of this, despite the fact that we are using static data, quadtrees are to be preferred here. A point quadtree would be impractical as it is a binary tree adaptation and would split the MBRs in the middle in the same manner as the kd-trees. A point-region quadtree would work.

1.4.1.5 Splitting Trajectories

Another optimizations that can be done is to split the MBRs of the trajectories. This will keep the pointers from one node to another, but the MBR will be reset at some point during the chain. To keep the MBR of the entire trajectory, an extra node will be prepended to the linked list, not containing any point, but rather just the MBR of the entire path and pointers to the first nodes in each split. What this allows for is splitting the trajectory on where the two resulting MBRs would be the smallest and removing the most amount of dead space where there are no points. How many times this should be allowed can be used as a tweakable parameter, allowing for tailoring it to each dataset. One could also have the amount of splits dependant on a set variable so that all splits would contain a set amount of nodes. A visual comparison of a list without trajectory splitting and with a single split can be seen in figure 1.5.

1.4.2 Querying Linked MBR Lists

How querying linked MBR lists would work is dependant on the top-level indexing method. For now we will assume that we are using a point-region quadtree and that the query is *is contained by* with a query window q. Querying would work by first narrowing down the search space to the smallest possible quadrant(s) in the quadtree. From there, each top-level MBR (MBR of the first node or the parent MBR p) would be checked against the query window and if they are disjoint it would be discarded. If there are any top-level MBRs that are not disjoint their lists will have to be traversed. If the MBR is a parent MBR, it will check the MBR of its first child first, and if it is disjoint it will skip to the next child without traversing the children of the first child. For instance, in figure 1.5 we assume that the MBR of p intersects the query window, but the MBR of node 0 does not. Then, node 0 through 2 would not be traversed, but one would rather go directly from p to

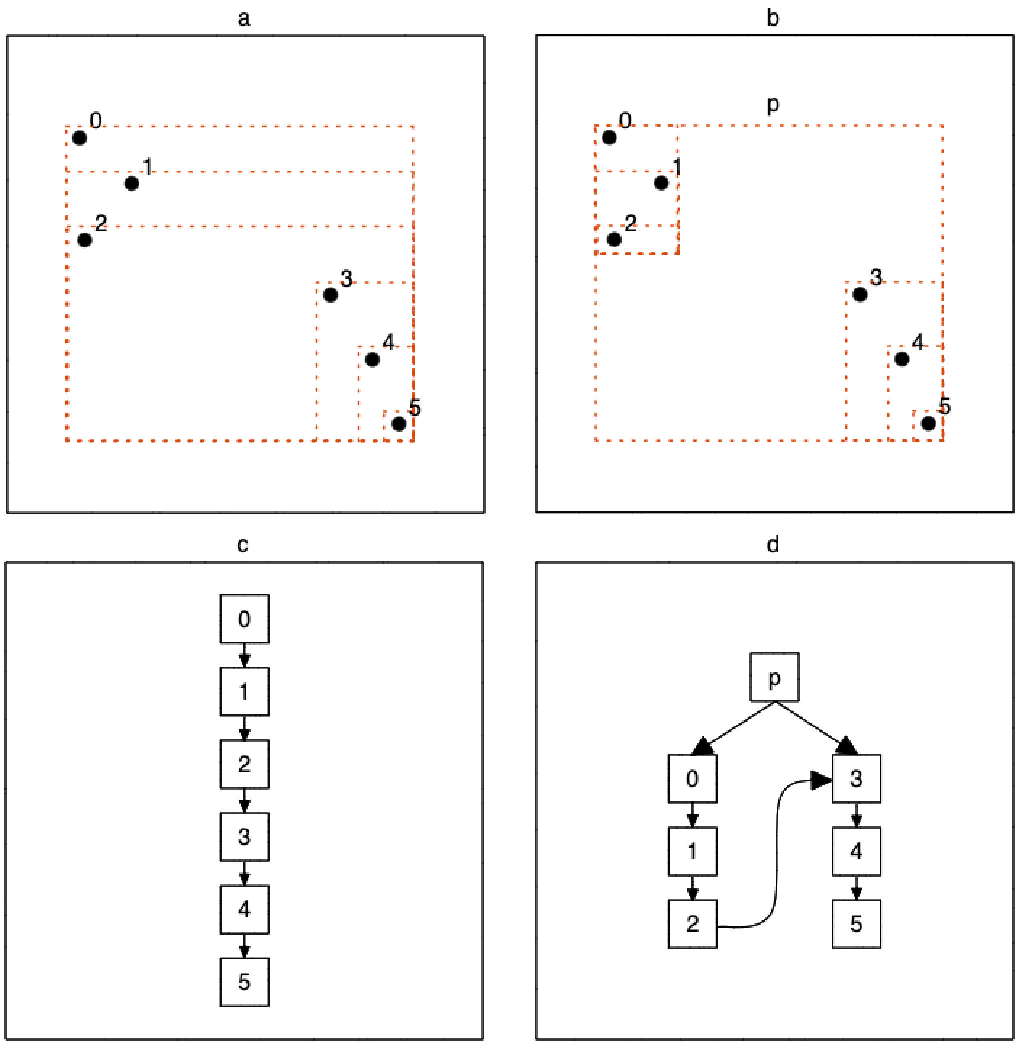

Figure 1.5: Visualization of splitting trajectories once. *a* and *b* show how the MBRs will be affected and *c* and *d* show how the linked list will be affected. Node p is a node that is not a point, but rather the parent MBR of the entire list.

node 3. This continues until one of the MBRs intersects the query window. Once an intersecting MBR is found the traversal of the list would begin. The distance from the first node to the query window would be calculated and if the distance is greater than 0 the corresponding amount of nodes would be traversed without checking the MBR. If any node in the traversal is enveloped by the query window it will be appended to the list of results. If the MBR of any node is disjoint with the query window the traversal would terminate and result list is either returned, or if there are more intersecting trajectories the next trajectory is traversed.

1.4.3 Limitations of linked MBR lists

As this data structure is focused on querying static data, no compromises have been made to ensure that it is fast to update. Due to the amount of MBRs that are dependent on each other, inserting, deleting or moving nodes is going to be inherently computationally expensive. Obviously, being slow to generate is going to be a drawback in most real-life scenarios, but in a perfect world where the data does not change and in theory this should be fine. If the generation time is heinous it will not be usable, but time will tell.

One other limitation is that geometric tests against MBRs can be done in constant time [11]. This means that improvements such as node skipping might have an almost insignificant impact on performance. If this proves to provide next to no performance boost it can easily be stripped from the implementation.

Another limitation is that of how the indexing method will behave based on the dataset. Using a quadtree for structuring the data can cause issues as the data might have significant overlaps due to the nature of how the traffic flow of a city behaves. However, this should be mitigated by the trajectory splitting, but it can be something to be aware of during benchmarking.

The possibly largest limitation of the linked MBR list is that the main bottleneck of databases is usually I/O. This means that optimizing locality of the data is going to have the most impact. That is why R-trees store their data the way they do. Each node is one page, meaning that all the data within that node is contained to a single page on disk, making it really fast to access other data within the same spatial area as they are also close on disk. Now, if everything is done in-memory, this will not be relevant, but if it is to be implemented in a database this might affect performance. Knowing if this is a limitation without having put it into practice is not easy.

Overall, this data structure has quite a lot of components and small moving parts that can impact performance. In addition, data structures can behave very differently based on datasets (bound by the time complexity). The easiest way to see how well the method performs is to test it out.

1.5 Evaluating Linked MBR Lists' Performance

Evaluation of the performance of linked MBR lists can be done in two ways: writing an extension to a spatial database that utilizes this data structure and see how it compares to existing indexing methods, or writing an implementation for both linked MBR lists and already proven data structures in some programming langauge and comparing the performance without using a database at all.

The queries that are of interest are the ones defined by *Open Geospatial Consortium* as briefly mentioned in section 1.3:

- **Disjoint**
- **Equals**
- **Touches**
- **Crosses**
- **Contains**
- **Within**
- **Covers**
- **Covered by**
- **Overlaps**

These are commonly known as *DE-9IM* and the explanation of each predicate can be found in [4]. Although it is supposed to be optimized for querying it can also be of value to see how it compares when it comes to generating the index and how much space the index takes up.

1.5.1 Comparison Within a Database

Comparing the performance of linked MBR lists in a spatial database would require writing an extension that allows for using alternative indexing methods. The spatial database with the most support for writing extensions is PostgreSQL. Extensions in Postgres is one of the main draws to the DBMS. Extensions can be used for creating new data types, modifying how a process is performed and, essentially adding functionality to the DBMS. These extensions can be written in different languages depending on what part of the DBMS one is altering. Adding new data types or operators can be done with SQL, whilst more advanced alterations must be done in low-level languages such as C or Rust with C being the standard. Even though Postgres has quite and extensive documentation on extensions, there is no official documentation on how to create an extension that adds an indexing method as it is only referenced in the *CREATE INDEX* documentation:

> "*PostgreSQL provides the index methods B-tree, hash, GiST, SP-GiST, GIN, and BRIN. Users can also define their own index methods, but that is fairly complicated.*" [12]

Other sources on the topic are scarce, and it seems the general consensus on the topic is that it is not something that can be easily accomplished. One way to peek into what the process of creating a new indexing method looks like can be found by reading through the source code of available extensions that do create its own indexing methods. The source code of PostGIS is mirrored on GitHub[1], and is publicly available. Taking a look at the file *postgis/gserialized_gist_nd.c* we can look through what seems to be the implementation of R-trees. This implementation is only one of many files in the repository required for making spatial indexing work. Now, it is hard to distinguish what is the minimal amount of work is required for getting a custom indexing method up and running without having to understand all the code in the repository and having a deep understanding of the Postgres source code.

[1]`https://github.com/postgis/postgis`

1.5.2 Comparison Outside a Database

The alternative to writing a custom indexing method in Postgres is to implement linked MBR lists in a programming language and comparing its performance to an already proven indexing method outside of a database. There are several programming languages that have indexing data structure implementations through libraries such as the R-tree package in Python [13]. However, implementing linked MBR lists in Python and comparing it to the performance of such a package would not be a fair comparison as this package is a wrapper for a C implementation, and C is a much faster language than Python [14]. One solution would be to write both data structures from scratch, making sure that the playing field is even when it comes to underlying technology. Deciding on which language to use depends on what the goal is. If both indexing methods are implemented in the same way without using any performance gains through means such as compiling to lower-level languages, this will show how the data structures compare relative to each other. This can be done in any programming language, either high-level or low-level with the same result. If the data structure is to be used in a database at a later stage it can be beneficial to write it in a low-level language such as C++, C or Rust, but if that is not a priority a higher level language can be used for development speed.

Comparison outside of a database would also require writing the queries for the benchmarks.

1.5.3 Result

Both these comparison methods come with their own advantages and drawbacks. One one hand, an in-database implementation would be beneficial to see how each of them perform in a real-world scenario. This would allow for seeing if the algorithm has any benefit at all compared to how the currently implemented spatial indexing methods. It would also allow for seeing how I/O affects the performance. The drawback of this method is that it would require a significant amount of time and work to only get the indexing method implemented into a spatial database to start with before starting to benchmark the performance. If the indexing method is useless, this would be a waste of time.

Writing the algorithms and comparing them outside of a database would allow for faster results, but without benefit of seeing how it would be affected by I/O as it would only run in memory. It also requires either finding a library for existing indexing methods in a programming language, or spending time writing new implementations.

Overall, the best choice is to implement the method outside of a database as that is the most feasible within the time period of the project. Implementing it in a database can either be chalked up to future work or left as an exercise to the reader.

Chapter 2

Plan for Master's Thesis

This chapter aims to provide a pointer for how we are planning to conduct the work for the Master's Thesis. Everything is subject to change, but this will work as a baseline to help to know where to begin.

2.1 Research and Report Writing

2.1.1 Research

More research beyond what is contained in this project is needed. There is still more knowledge to be gained on the subject of spatial data structures. Expanding linked lists may not be the silver bullet to querying linked directional spatial data and that there are more improvements to be done. Looking at more spatial data structures and seeing what makes them good can help find inspiration for further improvements.

The two weeks allocated to research before starting implementing might easily turn into three or four weeks if not paying attention and staying on schedule. Setting a hard limit on two weeks for initial research (from the first day of research) will help not lock into over-researching, causing a delay in other parts of the process. If there is going to be more research done it should be conducted in week 5 and outwards. This will allow for trying out the data structure early on, making sure that it is possible go back and improve it before writing too much on the thesis. Spending too much time researching before implementing the method might result in having to double up on research as there is probably something that will need to be changed with the data structure.

During the report writing there will be more research conducted as well, especially on the background section, making sure that everything that is written is correct.

2.1.2 Report Writing

Most of the time during this thesis is going to be spent writing the report. Being as efficient as possible when writing is going to speed up the process quite a lot seeing that this is the part that is going to take the most time. To hit the ground running from the very start there has been prepared a LaTeX-template from this project that will be usable from the very start. One idea would be to create the skeleton or structure of the thesis before next semester as well, as that would help outline what the different parts from the very start.

Throughout the work done on this project it has been proven that working early in the morning is not effective. Doing activities in the morning can help alleviate this as one will not be as restless when writing. Because of this, the plan is to start the day early, making sure that enough activity and exercise is performed before the writing starts.

It has also shown that beginning to write without having a plan can cause large portions of the text to be rewritten later on. This is not very effective, especially if noticed early enough before having spent a long time writing something irrelevant. Because of this, all sections should be outlined with bullet points before any writing is done. In addition, making sure that another set of eyes is involved throughout the entire process will make sure that the report does not step outside its boundaries or scope. By preparing something along a "Status, Problem, Plan" at a set interval one can more easily know what to get feedback on. This will be utilized for checking in with the supervisor to ensure that the project is going along nicely.

2.2 Implementation

To produce the results, the data structure will have to be programmed, as well as an already proven data structures to compare it to. This will be what the entire thesis revolves around. If the data structure turns out the be inefficient, a week has been allocated to look at the results, and improve it based on those. This can be seen under week 5 in table 2.1. If it still is not performing as intended the plan will continue as usual, not differing from if the results are positive.

2.2.1 Programming Language

The choice of programming language is something to consider at a later date. It is not present in the table as it is not presumed to take a significant amount of time to research thoroughly enough to make an informed decision. Some key factors to consider are:

- **Programming speed**
  The more efficient one is with a language, the faster the implementation stage is going to go. Having to learn an entirely new language is a barrier that might cause a large setback and that is not a preferable outcome.

- **Ease of visualization**
  To show the results in the thesis, the language should have good support for visualizing the results of the benchmarks. Now, most languages have extensive visualization libraries, so this likely will not be a huge factor.

- **Benchmarking**
  Benchmarking the different algorithms is going to be key for this thesis to be written. Again, most languages probably have a benchmarking feature or library, but there will be differences and being able to see both time and memory statistics would be key for analyzing the results.

- **Libraries**
  Libraries can help speed up the programming process by being able to re-use code that other people have already written. If there is a good open source R-tree implementation already written, it could save many hours trying to implement it by hand. However, there is always

the risk that the libraries can impact the comparisons. For instance, if one is using Python and using the R-tree library, nothing written in pure Python would come close as that library uses C under the hood, making it way faster. Libraries are also prone to having implementation errors that could affect performance or outcome. Making sure that the libraries implement the control data structures properly is going to be key to making sure that the comparisons are as fair as possible.

2.3 Action Plan

An action plan has been written for the work on the thesis. It can be seen in table 2.1. January, even after the 13th, can be hectic due to personal reasons, but even if the plan is set back two weeks it should still be within a good margin of the time limit. As when working on any plan not everything is going to go as smooth as outlined. Being aware of this will allow for using the plan as more of a baseline that can be continously updated to keep track of the work, whilst also making sure that every aspect of the project is taken into account.

Table 2.1: Action plan for the Master's Thesis.

Week	**Focus**	**Goal**	**Initial Dates**
1	Research	Comprehensive list of spatial data structures and their properties	13.01.2023 - 20.01.2023
2	Research	Potential improvements to linked MBR lists	21.01.2023 - 28.01.2023
3,4	Programming	Implementation of linked MBR lists and comparison data structures	29.01.2023 - 12.02.2023
5	Research and programming	Look at results, find improvements	13.02.2023 - 20.02.2023
6,7	Report writing	Write background	21.02.2023 - 08.03.2023
8	Report writing	Write method	09.03.2023 - 16.03.2023
9	Report writing	Write results	17.03.2023 - 24.03.2023
10	Report writing	Write introduction	25.03.2023 - 01.04.2023
11	Report writing	Write conclusion	02.03.2023 - 09.04.2023
12 and beyond	Report writing	Improve on feedback	10.03.2023 - 10.06.2023

Bibliography

[1] D. Curry, "Strava Revenue and Usage Statistics (2022)." Available online at `https://www.businessofapps.com/data/strava-statistics/`; Last accessed: 06-12-2022, 2022.

[2] H. Samet, *The Design and Analysis of Spatial Data Structures.* USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[3] Esri, "History of GIS." Available online at `https://www.esri.com/en-us/what-is-gis/history-of-gis`; Last accessed: 06-12-2022.

[4] Open Geospatial Consortium, "Spatial Relations Defined." Available online at `http://docs.safe.com/fme/html/FME_Desktop_Documentation/FME_Transformers/Transformers/spatialrelations.htm#DE9IM_Matrix`; Last accessed: 01-12-2022.

[5] M. Vassilakopoulos and T. tzouramanis, *Quadtrees (and Family)*, pp. 2219–2225. Boston, MA: Springer US, 2009.

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

[7] A. N. Papadopoulos and Y. Manolopoulos.

[8] R. K. V. Kothuri, S. Ravada, and D. Abugov, "Quadtree and r-tree indexes in oracle spatial: A comparison using gis data," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, (New York, NY, USA), p. 546–557, Association for Computing Machinery, 2002.

[9] I. Kamel and C. Faloutsos, “Hilbert r-tree: An improved r-tree using fractals,” tech. rep., 1993.

[10] D. Hilbert, “Ueber die stetige abbildung einer linie auf ein flächenstück,” *Mathematische Annalen*, vol. 38, pp. 459–460, 1891.

[11] P. Rigaux, M. Scholl, and A. Voisard, “6 - spatial access methods,” in *Spatial Databases* (P. Rigaux, M. Scholl, and A. Voisard, eds.), The Morgan Kaufmann Series in Data Management Systems, pp. 201–266, San Francisco: Morgan Kaufmann, 2002.

[12] PostgreSQL, “CREATE INDEX.” Available online at `https://www.postgresql.org/docs/current/sql-createindex.html`; Last accessed: 28-11-2022, 2022.

[13] The Toblerity Project, “Rtree: Spatial indexing for Python.” Available online at `https://rtree.readthedocs.io/en/latest/`; Last accessed: 28-11-2022, 2019.

[14] I. M. Wilbers, H. P. Langtangen, and Å. Ødegård, “Using cython to speed up numerical python programs,” *Proceedings of MekIT*, vol. 9, pp. 495–512, 2009.