NTNU

Norwegian University of
Science and Technology

Hilbert-Geohash

Hashing Geographical Point Data Using the
Hilbert Space-Filling Curve

Tibor Vukovic

Master of Science in Computer Science
Submission date: June 2016
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

As geographic information systems become more and more widely used, an increasing amount of positional data is generated and shared between users. Sharing spatial data introduces challenges, with URLs containing lengthy coordinates needing to be sent through services with character limits. This problem has been solved with the usage of geospatially-specific URL-shortening systems, using properties of space-filling curves to reduce the multiple dimensions of point data, down to a shorter, URL-friendly, hash value.

An example of such a system is the Geohash.org service, using the Z-order space-filling curve to map between hash values and geographical points. While the Z-order curve is simple, alternative space-filling curves of higher complexity exist, possessing several theoretical advantages and disadvantages. One such alternative is the Hilbert curve, which presumably has superior hashing properties at the cost of being computationally more complex. Thus, the Hilbert-curve based point data hashing system *hilbert-geohash*, is implemented to determine whether these properties are worth the additional complexity they introduce.

The results show that the Hilbert curve is, to some extent, applicable within the context point data hashing, but that the complexity increment is not proportional to the practical benefits over the Z-order curve. While a point data hashing system is not fully capable of exploiting the specific advantages of the Hilbert-curve, a proposed distributed point data retrieval system should be capable of doing so. This proposed system would use the presented *hilbert-geohash* system as a crucial part of its hash-space addressing mechanism, taking advantage of the properties of the Hilbert curve to reduce communication distances between peers in the distributed network.

Sammendrag

Ettersom geografiske informasjonssystemer taes mer og mer i bruk, genereres og deles en økende mengde posisjonell data mellom brukere. Deling av geodata introduserer utfordringer ved at URL-er bestående av lange koordinater må sendes gjennom tegnbegrensede tjenester. Dette problemet har blitt løst ved å ta i bruk geodata-spesifikke URL-forkortingstjenester, som benytter romfyllende kurvers egenskaper til å redusere dimensjonene i punktdata til en kortere, URL-vennlig hash-verdi.

Et eksempel på et slikt system er tjenesten Geohash.org, som bruker den romfyllende Z-kurven til å oversette mellom hashverdier og geografiske punkter. Z-kurven er enkel, men det finnes samtidig alternative romfyllende kurver av høyere kompleksitet, som innehar teoretiske fordeler og ulemper. Et slikt alternativ er Hilbertkurven, som antas å ha overlegne hashingegenskaper på bekostning av høyere beregningskompleksitet. I denne masteroppgaven implementeres den Hilbertkurvebaserte hashingtjenesten for punktdata *hilbert-geohash* for å undersøke hvorvidt disse egenskapene skaper forbedringer som er verdt den medfølgende kompleksiteten.

Resultatene viser at Hilbertkurven til en viss grad er anvendelig i kontekst av hashing av punktdata, men at kompleksitetsøkningen ikke er proporsjonal med de praktiske fordelene sammenlignet med Z-kurven. Mens et hashingsystem for punktdata ikke er fullt i stand til å utnytte de spesifikke fordelene Hilbertkurven har, burde et foreslått distribuert gjenfinningssystem for punktdata være i stand til dette. Det foreslåtte systemet ville ta i bruk det presenterte systemet *hilbert-geohash* som en viktig del av sin mekanisme for å adressere hash-rom, og utnytte Hilbertkurvens egenskaper til å redusere kommunikasjonsavstanden mellom likemenn i det distribuerte nettverket.

Acknowledgments

First and foremost, I would like to express gratitude towards my family and friends for providing support and motivation throughout the entirety of my time at NTNU. I would like to thank my supervisor, Prof. Svein Erik Bratsberg, for giving me great encouragement and allowing me to freely shape my own research.

Furthermore, I would like to thank my fellow students Kristine Steine, Juul Arthur Ribe Rudihagen, and Hans Kristian Henriksen for their crucial feedback and advice, helping me complete this project.

Finally, a special thanks to my late grandfather, Rašid Hodžić, a true innovator, veteran, and generally an inspiration in life. You will be missed, but not forgotten.

Contents

1 Introduction **3**

1.1 Purpose . . . 3

1.2 Research questions . . . 4

1.3 Product . . . 4

1.4 Limitations . . . 5

1.5 Approach . . . 5

1.6 Structure . . . 5

2 Background and related work **7**

2.1 Definitions . . . 7

2.1.1 Multidimensional point data . . . 7

2.1.2 Queries . . . 8

2.1.3 Quadrant terminology . . . 9

2.2 A naive approach . . . 9

2.3 Tree structures . . . 10

2.3.1 Quadtrees . . . 11

2.3.2 K-d Trees . . . 13

2.4 Space filling curves . . . 15

2.4.1 The Morton curve . . . 16

2.4.2 The Hilbert curve . . . 18

2.4.3 Hilbert curve locality properties . . . 19

3 Architecture and Design **21**

3.1 System overview . . . 21

3.1.1 Motivation . . . 21

3.1.2 System structure 21
3.2 Tools and Technologies 22
3.2.1 Version control 22
3.2.2 The Go programming language 23
3.2.3 Go Gin 23
3.2.4 Node Package Manager and Webpack 24
3.2.5 Heroku 24
3.3 The core algorithm 24
3.3.1 Z-ordering 25
3.3.2 Hilbert-ordering 26

4 Implementation 29

4.1 System Modularity 29
4.2 Server 30
4.3 Geohash 33
4.3.1 Modules 33
4.3.2 The Hashing algorithms genZorder and genHilbert 37
4.3.3 The inverse algorithms genHilbertInverse and genPoint 40
4.3.4 Precision 42
4.4 Client 42
4.4.1 Client Application Bundling 44
4.4.2 Interface 45

5 Evaluation of the Hilbert curve 47

5.1 Motivation 47
5.2 Hash Distance Measurement 47
5.2.1 Curve distance 48
5.2.2 Levenshtein distance 48
5.3 Test Data Generation 51
5.4 Test execution 51
5.4.1 Hash Distance 51
5.4.2 Benchmarking 54
5.5 Results 54

5.6 Evaluation 55

6 Summary **57**

6.1 Summary 57

6.2 Conclusion 58

6.3 Further work 59

Bibliography **60**

Chapter 1

Introduction

1.1 Purpose

As the earth's population increases[1], the cities we live in are increasing in size and complexity[2]. Many of us have become dependent on systems to help us solve geo-related problems like city navigation or finding various points of interest in unfamiliar places. Today's smartphones are becoming more and more capable, especially with the steady increase in processing power[3], faster cellular connections[4], and inclusion of sensors like the compass sensor that can determine device direction or the usage of the Global Positioning System (GPS). This has made it possible for developers to create systems that take advantage of these device capabilities, giving users the ability to have real-time navigation systems on their mobile phones. However, even though mobile phone are primarily communication devices, they often introduce some challenges when it comes to communicating spatial data. Popular communication services, like the Short Messaging Service (SMS)[5] or Twitter [6], both limit the length of messages and thus set limitations on how much information can be sent. Geographical point data poses especially a challenge, since the data often has a multitude of digit-wise lengthy attributes that need to fit within the message, i.e. Usually both the latitude and the longitude coordinates of a point need to be sent, either alone or as a part of a URL. This problem also exists more generally on the Internet with URL-links being often too long for usage on certain services, but this has largely been solved by using various URL-shortening services like Bit.ly[7] or Goo.gl[8]. While these services can be used generally on all URLs, they are not always the optimal choice for positional point data. This is because positional data often relates to physical locations which are areas of various sizes e.g. continents, countries, cities, neighborhoods etc. and these services provide no way to customize locational precision e.g. shorter URLs for large areas where less precision is needed. In addition, there is often no indication of the relation between two shortened URLs and their location. Two geographically close positions could have totally different shortened URLs, e.g: *http://goo.gl/Mf3kVc* and *http://goo.gl/psuJ96* are two points within the same building at the Norwegian University of Science and Technology. There are currently few known services that possess these properties, one of them is Geohash.org. This service is based upon on the Z-order space-filling curve, a function that maps from a two-dimensional point down to a one-dimensional entity, or a hash, used to create the shortened URLs. While the Z-order curve is computationally simple, there are alternative space-filling curves with various properties. The Hilbert curve is an alternative space-filling curve that, in theory, has some superior properties when it comes to locality clustering preservation between the dimensions [9, Conclusions]. Thus, the purpose of this work is to test these properties and evaluate their applicability within this point-hashing context.

1.2 Research questions

The research questions of thesis are as follows:

- RQ1: Compared to the Z-order curve, can the superior clustering properties of the Hilbert-curve be used practically within the point-hashing system to gain better hash-similarity between geographically close points?
- RQ2: Compared to the Z-order curve, are the advantages of using the Hilbert-curve worth the additional computational complexity of generating the hash?

This requires us to create a working prototype of a system capable of hashing and retrieving geographical points, using Hilbert space filling curve.

1.3 Product

The product of this research is the construction and evaluation of a prototype spatial data retrieval system that can hash point data and retrieve point data from hash values. The hashing mechanism within this system is based upon the Hilbert-space filling curve to gain a theoretical advantage over the Z-order-curve based Geohash.org equivalent. This system is a two-part system, consisting of a server application and a separate client application. Both client and the server-systems are relatively independent of each other, and could with minimal effort, be altered to work with other systems. In addition, the server application is also split into multiple sub-modules, which are usable as 'building blocks' within other similar systems. A variety of technologies currently considered as 'modern', are used both for implementation and hosting purposes of this system. This is done to incentivise further development of both this particular system, and generally the field of spatial data processing, by taking advantage of the current popularity of the used technologies.

The source code is available on GitHub:*https://github.com/tiborv/hilbert-geohash*

1.4 Limitations

Due to time limitations, no other space-filling curve types, except the Hilbert and the Z-order curves, will be implemented or evaluated. The client-based application will be held to a minimum viable product-standard, keeping it simple and avoiding any JavaScript front-end frameworks.

1.5 Approach

The first step in this research would be to get an overview of different, already existing, solutions and their theoretical foundations. This requires a review of relevant papers and literature to create a conceptual framework for further work. When a basic conceptual framework is acquired, a prototype implementation can be created to see if these theoretical concepts are feasible/possible to implement and also to get a generally better understanding of them. There is a possibility that the initial phase of this research needs to be revisited multiple times. This is mainly due to it generally being hard to predict outcome of the implementation phases and therefore, in a case of a 'dead end' e.g. a theoretical concept can't be implemented due to time constraints, the initial phase is revisited to properly consider alternatives and possible changes needed to be done.

1.6 Structure

This research is mainly structured into five chapters: Theoretical background, system design, implementation details, evaluation of the system, and finally a conclusion. The background chapter introduces some basic concepts related to multidimensional data, tree structures, and space-filling curves. The following system design chapter outlines the intended system structure and functionality of a Hilbert curve-based point-hashing system, with the implementation details of the system presented in the succeeding implementation chapter. Thereafter, an evaluation part evaluates the system and presents the corresponding results. Finally, a conclusion chapter presents suggestions for further work and a conclusion to the research.

Chapter 2

Background and related work

This chapter is based on prestudy done in the fall semester of 2015, titled: *The Structuring and retrieval of two-dimensional spatial data*, and dealt with two-dimensional spatial data management in a distributed system context. This is considered relevant as it presents a general introduction to spatial data, with various methods of retrieval and management, as well as some practical usage examples. The final section of this chapter presents the background theory of space-filling curves, with focus on the two most relevant curve types for the implementation done in succeeding chapters.

2.1 Definitions

2.1.1 Multidimensional point data

Multidimensional data is, as its name implies, data with attributes in more than one dimension. These dimensions can be of any type of attributes, both spatial e.g. distances, lengths, and non-spatial e.g. telephone numbers and street addresses. Multidimensional point data differs from general multidimensional data by often having the attribute values be of the same type and represent units of space. This is in contrast to generic multi-attribute data which often can have completely uncorrelated attributes. A trivial example of this can be having a database of people where each entry has the attributes birth year and current weight. Executing range queries on this data set would yield the quite uncommon unit of year-kilograms, and probably not be very useful.[10, Chapter 1]. An example of what multidimensional point data actually is, can be illustrated by using point entities in two-dimensional Euclidean space, each having an X and an Y value that defines its

Table 2.1: Example data with attributes X and Y

Point id	X	Y
A	1.5	5
B	3.5	4
C	4.5	6
D	5.5	2
E	6.5	8
F	7.5	4
G	8.5	6
H	9.5	2

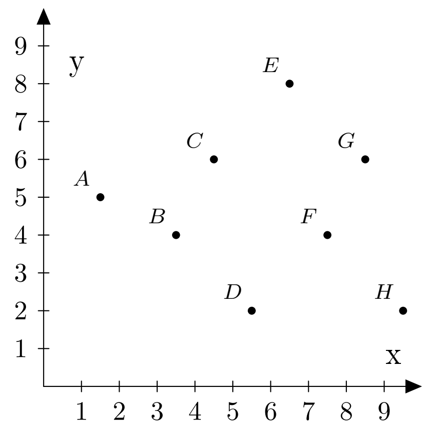


Figure 2.1: Example points from Table 2.1 plotted in a two dimensional graph

positions in a Cartesian coordinate system. A randomly generated set of such data can be seen in Table 2.1, and the same data plotted on a graph with a query example in Figure 2.1.

2.1.2 Queries

Querying spatial data can be done in various ways. For simplicity's sake, the queries in this chapter will be rectangular and defined by two points. An example of such a query is shown in Table 2.2 and a two-dimensional visualization in Figure 2.2

Table 2.2: Example query data points

Point id	X	Y
Q1	2.5	3.5
Q2	7.5	6.5

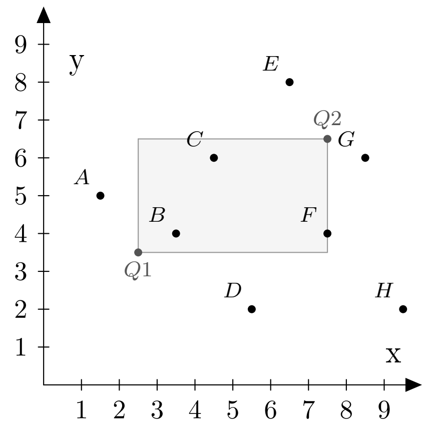


Figure 2.2: Example query points from Table 2.2 plotted on Figure 2.1. Points B, C and F are within the query.

2.1.3 Quadrant terminology

The term 'quadrant', used throughout this research, relates to one-fourth sized part of a spatial area i.e. a spatial area is divided into four sections equally sized quadrants. To be able to concisely describe which quadrant is in question, appropriate spatial abbreviations for each quadrant are introduced: NE(north-east), SE(southeast), NW(northwest) and SW(southwest). Figure 2.3 shows this mapping.

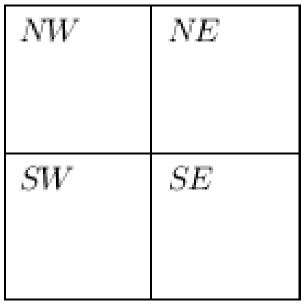


Figure 2.3: Mapping between quadrant and abbreviation.

2.2 A naive approach

In this section, an example of a retrieval problem is presented along with some naive solutions. This is first and foremost done to illustrate the complexity multidimensionality introduces, and also to show why one-dimensional approach become ineffective when handling these types of data.

Retrieval

An example of retrieval of multidimensional data can be to consider a set of N points, each with coordinate values in D dimensions, and a range query like two points defining a rectangle. Using the example data from Table 2.1, N, in this case, becomes the set of points A-H (N=8), each having two attributes (D=2: X ,Y) and the query defined by the points Q1 and Q2 from Table 2.2. In this example, one solution is to iterate through every point, one by one and evaluate it against the query points. This will be the easiest solution to implement, and for a small data set like the example data, this can be considered a sufficient solution. If a bigger data set is used, the inefficiency of this approach becomes more obvious. To evaluate each of the N points, each having D attributes, and check if they are within the query range, yields an inefficient process with time complexity O(N*D)

[10, Section 1.1]. This naive, 'one-dimensional' retrieval approach (i.e. not using an appropriate data structure), is easily identified as an inefficient 'brute force' search that certainly is 'cursed by dimensionality' and is highly inefficient.

A better approach

Another approach may be to store each point across multiple sorted one-dimensional data structures, e.g: Two lists, one where the points are sorted by their X coordinates and the other by their Y coordinates. In this case, a range query can be done in each list separately: First finding the range of the query in one single dimension, filtering the list with the min and max X-coordinates of the two query points Q1 and Q2 in Table 2.2, and then doing the same on with the Y-coordinate list i.e. using the query Y-coordinates. These tasks can easily be done in parallel since there is no shared state and result aggregation is not complicated. Upon completion of these parallel tasks, an intersection can be done between them to get the final result, e.g. only entries that exist in both result lists are returned. While this approach with the usage of data redundancy may be efficient for retrieval, it also introduces several disadvantages: The first and most obvious one being space unscalability i.e. data size doubles on dimensionality increments. This is due to the redundant data structure of having one list for each dimensional attribute. Another disadvantage is if a point is updated or deleted, each redundant copy of it in all dimensional collections, also needs to be updated or removed, which can be costly to perform. While this structure is not write-optimal, it still can be considered somewhat useful in distributed systems, as the computation can be divided into non-overlapping parts, i.e. each server in a distributed system can manage points for one dimension, with an additional aggregation-server that aggregates results from all dimension-servers.

2.3 Tree structures

While the previously mentioned, somewhat naive, approaches for managing spatial data have some advantages, they become inefficient with larger data sets. In this section, a set tree structures, more appropriate for storing and processing spatial data, are presented. Tree structures, by definition, are hierarchically ordered directed graphs where nodes have edges that represent child-parent relations. Two types of tree structures will be presented: Quadtrees and K-d trees.

2.3.1 Quadtrees

A quadtree is a tree structure where each node has at most four child-nodes. When used as a space partitioning structure, it iteratively divides space into four quadrants, increasing granularity each iteration i.e. splitting the same space into smaller quadrants. This splitting process is performed to decrease the amount of data residing in each node, distributing the data across its four child nodes. There are mainly two types of quadtrees: Point quadtrees and trie-based quadtrees. These differ from each other by their children-node definitions. Point quadtrees use actual data points as children-nodes, while trie-based use predefined regions of space as nodes and act as 'buckets' capable of containing several points.[10, Chapter 1.4]

Point quadtrees

Point quadtrees are tree structures where each node is represented by a spatial point. Each node has four children, each child being either a new point or a null-node representing empty space. An example of this structure can be seen in Figure 2.4 and the corresponding Figure 2.5. The sequence of insertion shapes the overall tree structure. A structure where each child-tree (i.e. a child with its descendants) has less than half of the entire node count, yields an optimal tree structure with the most efficient search properties. This can achieve by choosing an insertion order that splits space as equally as possible on each level i.e. finding the median points after each insertion. Therefore, an optimal tree can only be guaranteed if the inserted points are known upfront i.e. in bulk and not one-by-one.[10, Section 1.4.1.1]

Trie-based quadtrees

Trie-based quadtrees are tree structures where space is divided into finite-sized regions. Each node in the tree contains all the points that reside in the bounding spatial area it represents i.e all the points that are in its quadrant of space. This implies that the insertion order does not affect the tree structure. There are mainly two types of quadtrees: Matrix(MX) and Point-Region(PR).[10, Section 1.4.2]

MX trees[10, Section 1.4.2.1] are quadtrees where every point entry resides at the leaf level. Each leaf node has a predefined spatial area which it represents and is static, i.e. the areas are atomic and no splitting occurs. These trees can be illustrated as matrices of 0 and 1, having 0 indicate empty quadrants and 1 non-empty quadrants. A structure like this is suitable for range queries since the

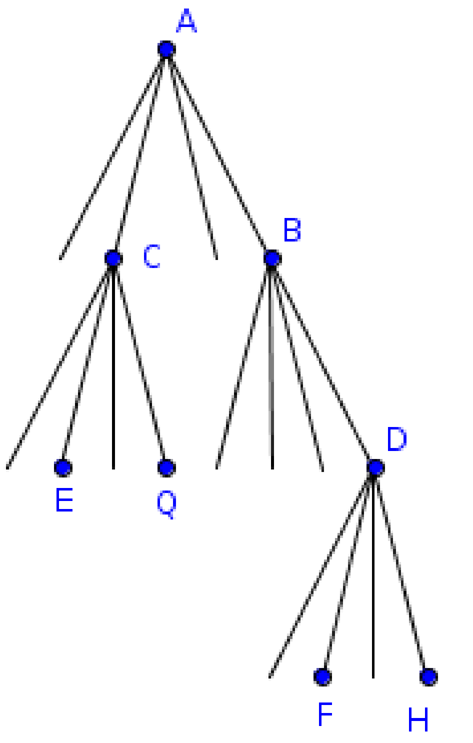


Figure 2.4: Point quadtree, with points from Table 2.1 inserted in alphabetical order. Children, from left to right, represent the quadrant sequence: NW,NE,SW,SE. E.g: Point C is the NE child of Point A.

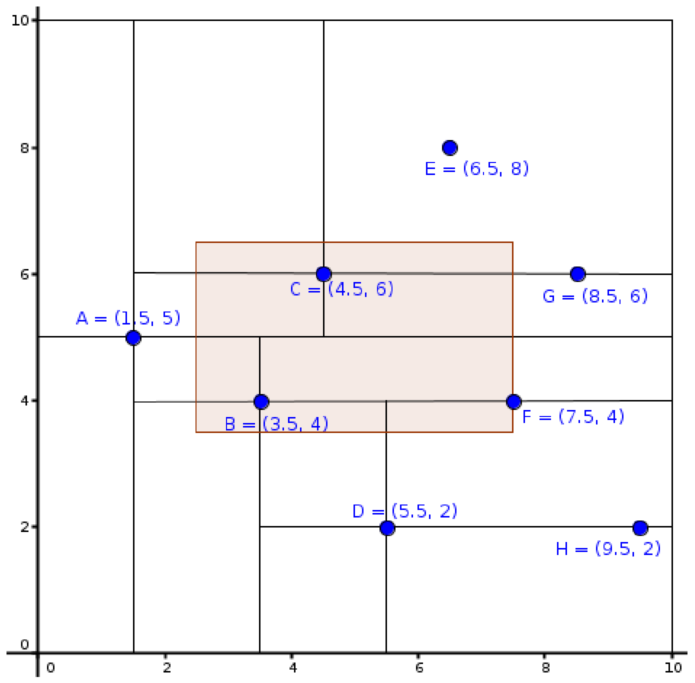


Figure 2.5: Spatial division overview of the quadtree from Figure 2.4 and the query from Table 2.2 marked.

size of the bounding area of each node is known in advance, and can be used to easily calculate which nodes intersect a given query. An example of an MX tree can be seen in Figure 2.7.

PR trees[10, Section 1.4.2.2] are quadtrees where space is recursively divided into four equally sized quadrants. Each node represents a quadrant area and has a 'bucket' that can contain a set of nodes. This implies that points can reside in non-leaf nodes, and can be stored higher in the tree. A split is usually initiated when a bucket is full and creates four children which the points are redistributed to. Figure 2.6 shows an example of such a tree.

A distributed quadtree index

Tanin et al.[11] describe a distributed system based on the MX quadtree. This system uses a load balancing hashing algorithm that distributes the ingested data uniformly across it peers in the network. Each peer manages a set of subsections of the entire hashing-space in-memory and the data

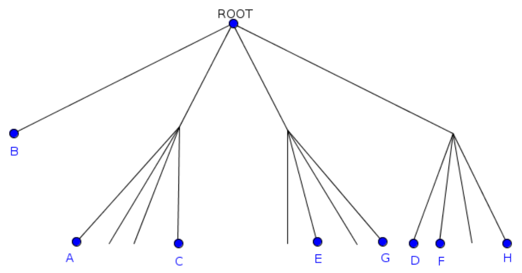


Figure 2.6: Point Region quadtree, with points from Table 2.1 and bucket size of 1

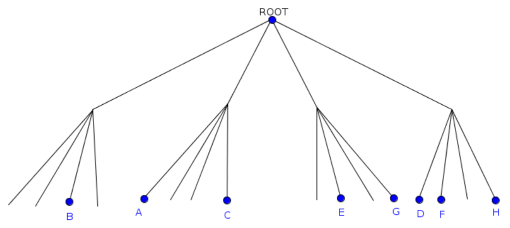


Figure 2.7: Matrix quadtree, with points from Table 2.1 inserted

is distributed accordingly. In this system, quadtree quadrants are represented by their centroids, also known as a control-points, and these are distributed randomly across all the peers. The peer associated with a control point is responsible for all computation done in its region of space. Queries, therefore, propagate through the system by spreading to all the control-point-holding peers of the query-intersecting underlying area, and since all control-points are uniformly distributed, the work is divided equally across the peers.

2.3.2 K-d Trees

A K-d tree, or a K-dimensional tree, is a space partitioning structure where data is ordered by iterating through its attributes in a predefined and constant order. This structure resembles a binary tree with each node having a maximum of two children, and insertions are distributed downwards in the tree based on their relation to a discrimination attribute i.e. data with a lower value than the discriminator propagates through the left child, while data with higher values are directed through the right child. For each level of descent, the discrimination attribute alternates e.g. when inserting two-dimensional point data, X coordinates are used for discrimination on one level and Y coordinates on the next. There are many forms of k-d trees and these differ from each other in mainly two ways: The choice of partitioning points and the dimension iteration sequence[10, Section 1.5.1]. The partitioning point, or 'discrimination' point, can either be a point from the input-data, or some arbitrary chosen point in space. The choice of these partitioning points influences the discrimination ability of each level within the tree. It is desired to choose discrimination points that divide the entire search space equally so that each level of descent excludes a maximum amount of nodes. Note that an optimal choice can only be made if all the points are provided beforehand (and not one-by-one) so that the optimal discriminators can be chosen (e.g: The median point or average of each subsection). The sequence of discrimination attribute alteration also affects the tree structure. In data sets where many points have attributes that are static, or with a small variation, choosing more discriminatory attributes is favorable. An example of a K-d tree can be seen in Figure 2.8 which represents the spatial division visible in Figure 2.9. These examples show the resulting tree of an alphabetical insertion order, which is a sub-optimal order. A more optimal order could be achieved by inserting the points that divide the underlying space equally with the current discriminator attribute. An example of a more optimal order is: D,A,F,B,C,G,E,H.

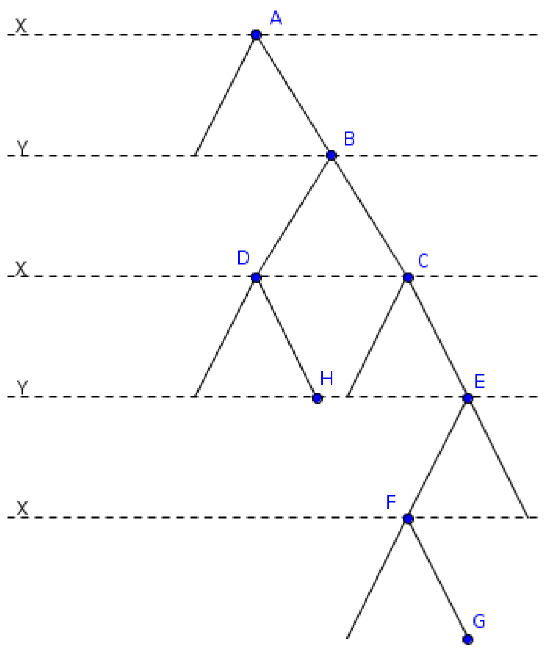


Figure 2.8: K-d tree, with points from Table 2.1 inserted in alphabetical order. The dotted lines indicate the discrimination attribute on that tree level.

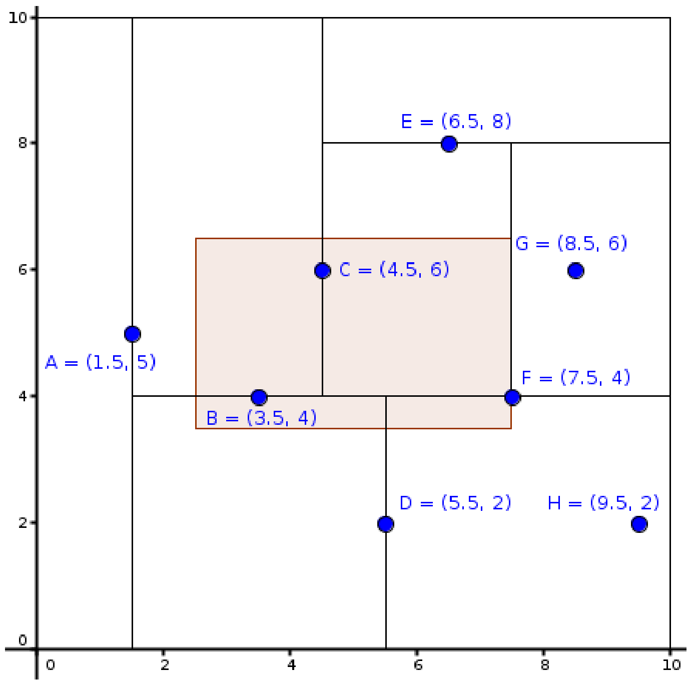


Figure 2.9: K-d tree spatial division overview of the graph from Figure 2.8 and the query from Table 2.2 marked

Distributed K-d Trees

The work done by Aly et al.[12] presents a distributed k-d tree structure used for managing very large image collections. Each peer in this distributed system contains a set of leaf nodes and their entire insertion paths from the root. There are two types of distributed structures presented: The independent structure and the distributed structure. The independent structure is a system where each peer contains a full-sized k-d tree, independent from the other peers. For this structure, queries need to be distributed across all the peers in the network and upon aggregation need to be gathered from all the peers. The distributed k-d tree structure is a structure where there is a single k-d tree structure for the entire system. One peer-node holds the tree-root and a vertical partition of the entire tree of variable size. Each peer manages a tree-leaf and the remains of the vertical partition, not residing in the root-peer. In this structure, the node holding the tree-root partition, decides which of its peers are intersected by a query. The affected peers then receive the queries and process it, returning the result to the root peer for aggregation. Benchmarking results presented by Aly et. al, shows that the distributed structure has 30 times more throughput and 30% more precision over its independently ordered counterpart.

2.4 Space filling curves

Space filling curves are functions that map from points in multidimensional space to points on a one-dimensional curve. These curves are definitions of exploration paths through space that cover the entire space with no overlapping. By dividing space into non-overlapping finite-sized partitions, the entire space can be represented by a set of nodes that each represents their surrounding spatial area, similar to the nodes in quadtree structures, outlined in Section 2.3.1. Each quadrant node gets its curve-value from the curve-visitation sequence i.e. the number of previously seen nodes is the curve-value of the current quadrant node. The order of a space filling curve describes the coverage granularity or 'resolution', i.e. higher order causes smaller quadrants, implying a greater number of quadrants are covering the area. Examples of order increment can be seen in the Figures 2.10 and 2.12, showing the first, second, and third orders of the Morton and the Hilbert curve. There are a wide variety of these space filling curves and each has its own advantages and disadvantages. In this research, two curves will be presented and compared: The Morton curve and the Hilbert curve. The reason why these particular curves are chosen is the prominent usage of the Morton curve within spatial indexing systems, and the Hilbert curves seemingly interesting locality preserving properties. This comparison is based on the curve properties described in [10, Section 2.1.1.2]:

- Complexity of the mapping function and its inverse.
- Ordering stability

  Does the relative order of nodes change when moving between curve resolutions?
- Preservation of locality.

  Are points close in spatial distance, close on the curve?

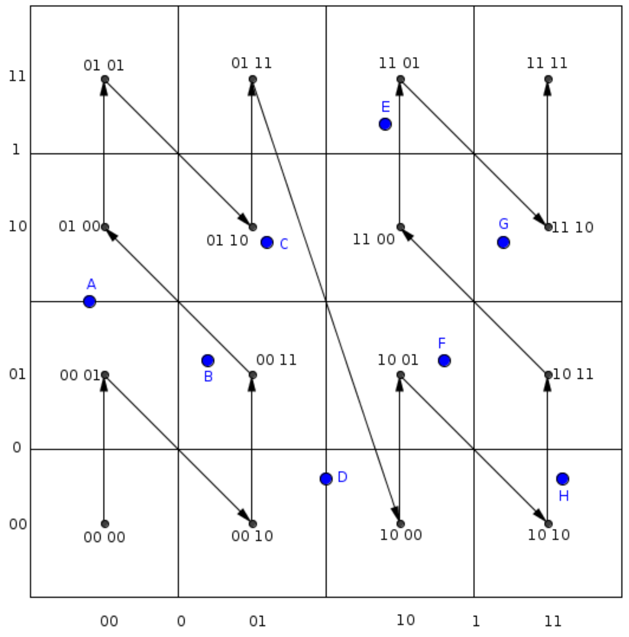
2.4.1 The Morton curve

The Morton curve, and also known as the Z-order curve, is a space filling curve that iterates through space in a 'Z' fashion. Figure 2.10b shows the first order of this curve with the corresponding curve-values. The mapping function is equivalent to bit interleaving the quadrant nodes coordinates, i.e. the pairwise concatenation of bits from each coordinate. Figure 2.10a show this encoding using the entire example data set. Ordering stability is present, with increments of curve order not changing the relative position of the points in relation to each other. The preservation of locality is to a certain extent present, with some exceptions. The most notable exception is located between quadrants 01 11 and 10 00 in Figure 2.10a. These are consecutive quadrants on the curve but distant spatially. This 'weakness' becomes more significant as the curve order increases i.e. the 'jump' is larger as the number of quadrants increases.

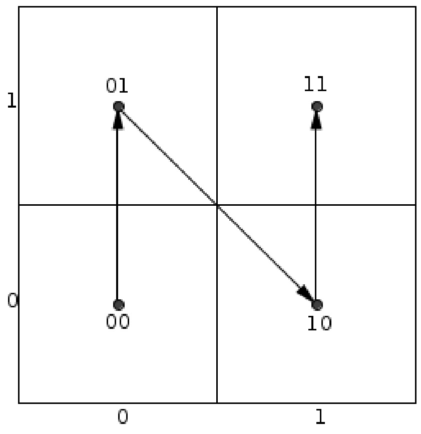
Geohash.org

Geohash.org is a service that maps from latitude and longitude points, to a 32-bit (0-z) unique hash. The service is based on the Morton curve, using the curve position of a point as its hash value, seen in Figure 2.11. One of the more useful properties of this technique is that bit length defines the accuracy of a point, i.e removing bits from the tail of the hashed output only decreases point precision while not changing its position in space. This can be used to balance precision against storage usage, for applications in which one is favored over the other.

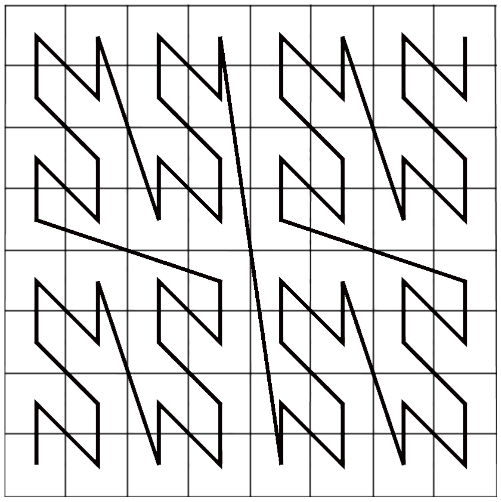
Geohash.org works by converting the latitude and longitude coordinates of a given point, to its geohash value, and use this as the URL for this point e.g: *http://geohash.org/u5r2u8wyptmf* is the URL for coordinates *(63.416891, 10.402666)*, with the hash value of this point: *u5r2u8wyptmf*. While this service is sufficient for many use cases, it suffers from the same shortcomings as the



(a) Second order Morton curve with points A-H from the example data in Table 2.1.



(b) First order Morton curve with curve-values for each quadrant: SW=00, NW=01, NE=11 and SE=10.



(c) Third order Morton curve.

Figure 2.10: First, second and third order Morton curves

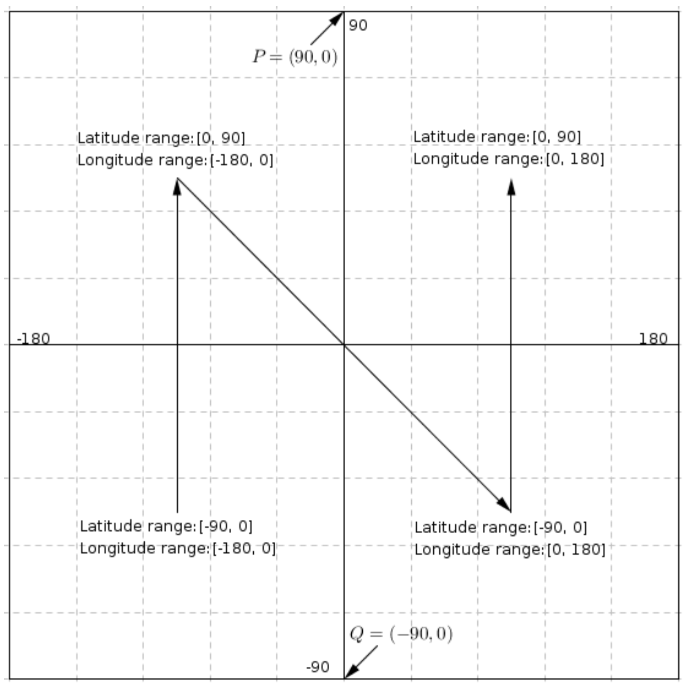


Figure 2.11: Example of Morton curve ordering and Geohash quadrant spatial division. Points P[13] and Q[14] are distant in space, but close in 32 bit hash values: *gzzzzzz*, *h00000*.

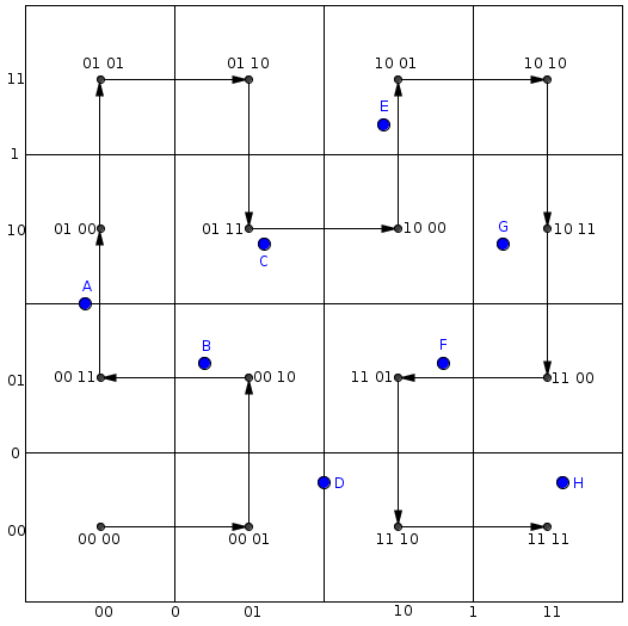
Morton curve regarding preservation of locality. This implies that two points with similar hashes, can be quite distant spatially, e.g: The hash *gzzzzz*[13] and the subsequent hash *h00000*[14] are close in base 32, while the points they represent $P = (90, 0)$ and $Q = (-90, 0)$ in Figure 2.11, are spatially distant.[15]

2.4.2 The Hilbert curve

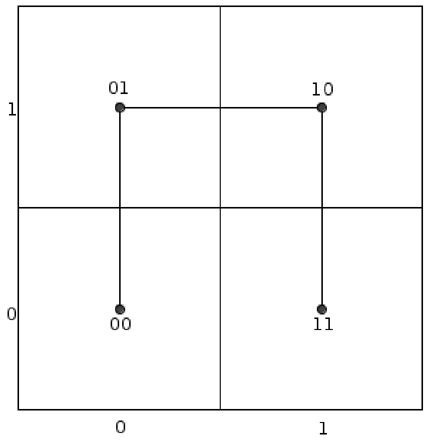
The Hilbert space filling curve, also known as the Peano-Hilbert curve, iterates through space in different orientations of a 'U' shape. This shape can be seen in Figure 2.12b. The second order curve, seen in Figure 2.12a, consists of four 'copies' of the first order curve, rotated in three distinct directions. These rotations are: 90 degrees clockwise (the 'Right' orientation in the SW quadrant), 0 degrees ('Up' orientation in quadrants NW and NE), and 90 degrees counter clockwise ('Left' orientation, quadrant SE), all seen in Figure 2.12a. The Hilbert curve is order-unstable: The relative locations on the curve are shifted as the curve's order increases due to its recursive 'rotational nature'. The third order curve is defined by the same rotation-pattern as the second order, except that the entire second order curve is used in each quadrant, as seen in Figure 2.12c. This recursive behavior continues as the curve order increases, making the mapping a more complex process and increases the locality preservation properties of the curve.

2.4.3 Hilbert curve locality properties

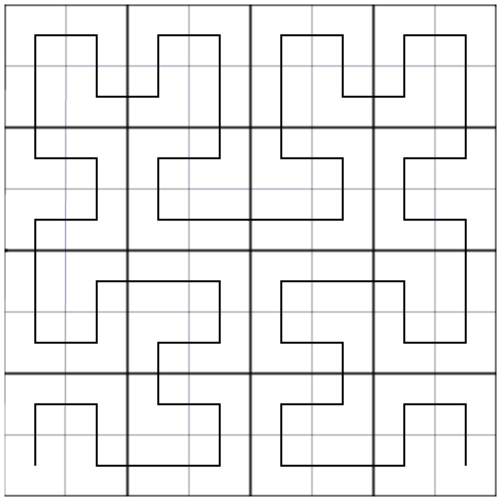
According to the conclusion section of the work done by Moon et al. [9, Conclusions], the Hilbert curve has superior preservation of locality of two and three-dimensional space over the Morton curve, based on disk access measurements upon range query retrievals.



(a) Second order Hilbert curve with points A-H from the example data in Table 2.1



(b) First order Hilbert curve with curve-values for each quadrant: NW=01, NE=10, SW=00 and SE=11.



(c) Third order Hilbert curve

Figure 2.12: First, second and third order Hilbert curves

Chapter 3

Architecture and Design

This chapter presents the design and architecture decisions made behind the Hilbert-curve based hashing system.

3.1 System overview

3.1.1 Motivation

The core properties of the hashing algorithm are mainly the properties held by the Hilbert-curve and described in Section 2.4.2. An important property is the Hilbert-curves locality preservation when comparing the similarity of two hashes and the geographical distance of the corresponding input points. This property is also present in a system simpler Z-ordering/Morton curve, like GeoHash.org described in Section 2.4.1, but is inferior when compared to the Hilbert-curve, as described in Section 2.4.3. Another important property, is the ability to use various lengths of hashes to describe points with various precision. This will enable a user to remove the least significant values of the hash without moving the position of the geographical point, only adding to its imprecision. All hashes with the same prefix are therfore geographically within the precision area of the prefix-point, i.e. the hash: 'aaab' and 'aaac' are within the area covered by 'aaa'.

3.1.2 System structure

The hilbert-geohash application consists of two parts: A client and a server. This separation is mainly done to be able to have a multitude of different client-applications connected to the same server, supporting more than just browser-based interaction. The interface between the client and the server is based upon using a JavaScript Object Notation (JSON)-API and any client-sided application that supports this, is potentially compatible. The server-sided system is built using the highly preformant programming language Google Go and the web-framework Gin. This choice is mainly based upon the assumption that the server-sided tasks could become quite computationally intensive and therefore would render the more popular languages such as Python or JavaScript i.e Django and NodeJs, as slower alternatives. The client is a simple JavaScript based application that does user interface related tasks, like sending and receiving request to and from the server. These tasks involve sending user-input from the client to the server and visualizing the responses on a geographical map. The Google Maps API is used for drawing the geographical map and the various shapes on top of it, like points and rectangles used for visualization of coordinate positions and areas. Git Version control hosted on GitHub[16] is used for managing the incremental changes to the code as well as open source distribution and availability. For deployment, the free app-hosting platform Heroku, is used witch easily integrates with GitHub and enables continuous deployment on every incremental code update to the Git repository.

3.2 Tools and Technologies

3.2.1 Version control

A version control system (VCS) is a system which keeps track of all changes made to a code base. This provides a way of tracking incremental changes done and often has roll-back functionality that can restore code from earlier points in time. By hosting a redundant copy of the code base on an external location, the risk of losing data when disk failures occur is vastly reduced. In this research, the distributed version control system git[17] is used along with the free git repository hosting service GitHub [16]. Git uses a decentralized tree structure and enables parallel development of the code base on a multitude of clients, with a (somewhat) automatic merge of code-changes upon a synchronization step also known as a "git pull". When the local synchronization step is finished, the merged version of the entire git-tree structure is then "git pushed" to an external, often centralized, repository making it available for git-pulls by other clients.

3.2.2 The Go programming language

The programming language chosen for the implementation of the system within this research is Go[18]. Go, often referred to as 'golang', is an open source programming language developed by Google and primarily designed for general systems programming. Some reasons for choosing this language are:

- Performance is similar to low-level languages such as C.
- Testing and benchmarking is built-in (i.e. no external frameworks or libraries are needed).

  It is easy to write tests and produce benchmark reports.[19]
- Supports most popular platforms: Linux, OSX and Windows etc.
- Simple and non-verbose syntax.

In addition, Go has good integration with the Git version control and this enables easy module-style importing of code directly from Git repositories hosted externally (e.g GitHub). This feature also encourages having a modular system structure, which enables this project have sub-modules that are reusable in other projects on their own. An example of this would be to have a project that only needs the core algorithmic Hilbert-curve functionalities and not the entire server structure. In this situation, a simple import statement can be used to import the core module and this could be used together with or within other frameworks and libraries.

3.2.3 Go Gin

To handle the server functionality of this application the web framework Gin[20] is used. Among the various other choices of Go web frameworks, Gin is mainly chosen due to it having a relatively simple code base and being highly preformant. This framework mainly manages the mapping from different URL-routes to go-functions that preform various tasks like extraction of data from HTTP requests and formatting of responses e.g. JSON serialization/HTTP status codes etc. In addition, as with most of all modern web frameworks, it automatically handles thread management and spawns threads for each new connection, enabling it to serve more than one client simultaneously. These request handling-tasks are not elaborated upon further within this work, as they are deemed as outside of the scope, even thought they are be easily comprehensible from simple examination of the code.

3.2.4 Node Package Manager and Webpack

As mentioned in earlier, a client-sided JavaScript application is also present within this system. The Node Package Manager (NPM)[21] system is a highly popular JavaScript module-hosting system for the JavaScript-based runtime NodeJs, that enables easy installation and version management of various modules and packages. This gives the benefit of being able to access the largest module hosting repository on the Internet[22], and thus avoid re-implementing already existing functionality. The usage of the module 'superagent'[23] is an example of such a module, and it enables easy creation HTTP-request with proper formatting and serialization, used for client-sided communication with server APIs. Another important package used in the client application, is the 'google-maps' package. This package enables usage of the Google Maps API which can create a geographical map and draw various shapes and objects on top of it. With this package, it is easy to illustrate the various points and areas resulting from the hashing procedures done on the server side. The various NPM modules that are used need to be bundled with the rest of the client-side code. For this, the module bundler Webpack is used and this bundles the module-code together with the source code, effectively creating one single file that contains the entire client application. This bundled file can now be compressed and served as a static file by the server, before it is requested by a client browser or something similar.

3.2.5 Heroku

The application engine Heroku[24] is a modern hosting service that closely integrates with the version controls system Git and the Git hosting service GitHub. This enables quick and easy deployment of web applications without the need of any setup or configuration of a web server. A running instance of an application can simply be created by the provided Heroku command line interface witch detects an existing Git-repository and adds the Heroku service as a git-remote i.e a target that a git push can be executed towards. Any code pushed to this external target will be automatically built and deployed by Heroku.

3.3 The core algorithm

The entire algorithm is contained within a single, Go-importable, package, with a simple API used for conversion between hashes and geographic points. This package is usable on its own, but resides within the same repository as the entire server implementation. The algorithm mainly consist of two parts: Conversion from a spatial point with coordinates to a Z-ordered bit array and a conversion from this Z-order array to a hilbert-curve ordered bit array. The resulting Hilbert-ordered bit array is base-32 encoded, enabling a more compact representation of the value using an alphanumeric representation of the bits i.e. each set of five bits in the array is replaced by a single character ranging from zero to the letter v (0-9, a-v, 32 different characters in total). This will enable the final resulting string to be easily used for various purposes that have character length constrains e.g. URL's in Twitter messages, creating short URL's of geographical positions. To go back from a base 32 hash to a geographical point, all the steps are inverted and done in an reversed order: Base32-decoding, hilbert-to-zorder and finally zorder-to-point.

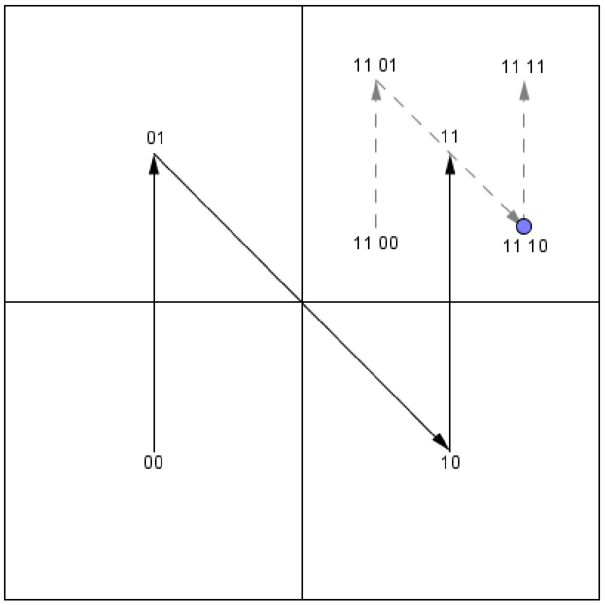


Figure 3.1: Example Z-order mapping of a spatial point. The point is mapped to the Z-order value 11 and then 10, yielding the final hash: 11 10

3.3.1 Z-ordering

Z-ordering, or also known as Morton encoding, is the process of generating an sequence of bits from a single spatial point. In more general terms, this process reduces the multiple dimensions of a spatial point down to a one-dimensional value. The algorithm takes a spatial point as input and produces an array of bits as output. This is done iteratively, computing two and two bits, appending each bit-pair to a result-array that is finally returned as the result. The output-bits are determined by finding witch of four quadrants, the location of the input-point is within. These four quadrant areas are defined by the current precision level, each resulting in a unique bit-pair output value as seen in Figure 3.1. Initially, this area contains the entire point-space, and divides the global map into four equally sized quadrants, each with its unique bit-pair identifying it.

The placement of a point within one of these four quadrants, yields its Z-oreder bit-pair and this is appended to the result bit-array. After the first iteration, the process repeats and again divides the area within that quadrant, into four sub-quadrants of equal size. The process continues until a desired amount of bits is obtained for the output array. This process is analogous to the creation of Quadtrees 2.3.1 and could be done both non-recursive and recursive.

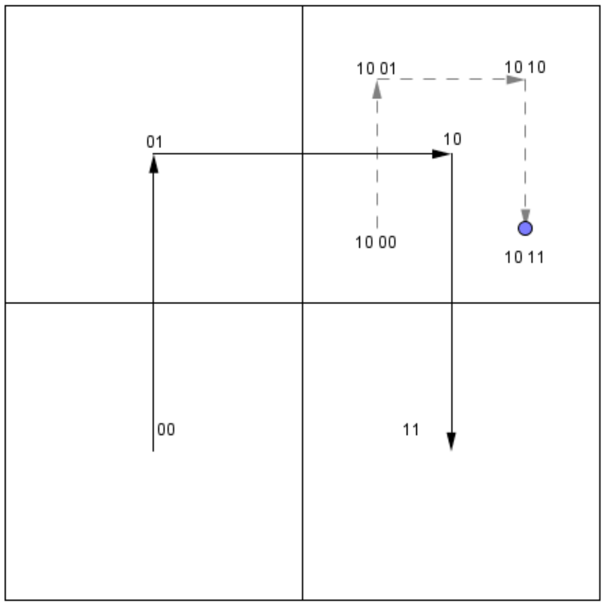


Figure 3.2: Hilbert-order mapping of the same point as in Figure 3.1. The final hash of the same point is: 10 11

3.3.2 Hilbert-ordering

The Hilbert curve hashing algorithm is mainly based on the algorithmic description by Lawder et al. [25]. The input is an array of bits representing a Z-ordered spatial point and the output is a Hilbert-curve based bit array. This is done by iterating through the input-sequence of bits, processing two bits at the time and appending the corresponding two Hilbert-bits to the output-sequence as seen in Figure 3.2. For each processed bit-pair, a Hilbert-curve mapping function is used to determine the result bit-pair. This function is one of four mapping functions, representing the four possible Hilbert-states or 'curve rotations'. The choice of mapping function is determined by a next-state variable returned form the previously used mapping function. Thus, each mapping function returns two values: The output bit pair appended to the result array and a next-state value used for determining the mapping function that is to be used for the next bit pair. These four mapping functions are illustrated in Figure 3.3.

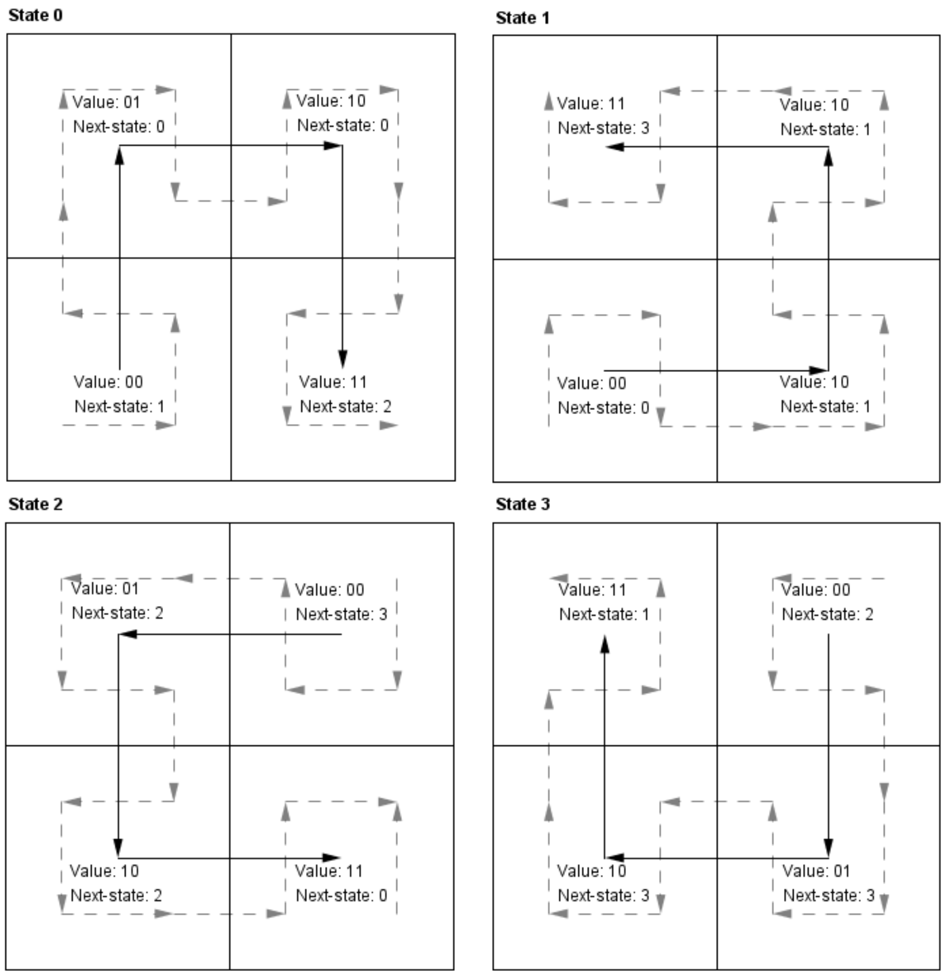


Figure 3.3: The four Hilbert-curve states. Each state maps to four Hilbert-values and next-states. The dashed arrows illustrate the resulting next-states for each mapping as well as the overall Hilbert-structure that is generated

Chapter 4

Implementation

In this chapter the implementation detalis of the *hilbert-geohash* system are outlined. The source code and a link to a running demo can be found on GitHub: *https://github.com/tiborv/hilbert-geohash*

4.1 System Modularity

The system is build from the ground up to be modular, with the core parts usable on their own. This is not only true with the client-server distinction, described earlier in Section 3.1.2, but also internally in both applications. The JavaScript client application uses the ECMAScript 6 (Section 4.4.1) import-statement, to import both the local and NPM-hosted modules (Section 3.2.4). The distinction between local and external modules is done by using a relative module path as opposed to just using the module name when importing, i.e. the module 'superagent' is external and causes a lookup on NPM for that module-name, while './superagent' causes a local file lookup. There are four locally imported modules and these reside in the 'js' folder within the client application:

- gui.js - The Graphical User Interface (GUI) module, the main controller module for the user interface.
- map.js - The geographical map module, used within the gui-module for initializing and using the Google Maps API.
- api.js - The Application Programming Interface (API) module, used for communication with the back end Go-server application.
- helpers.js - A small module consisting of a set of helper functions.

Both the external and internal modules are bundled with Webpack, the module bundler (Section 3.2.4), creating one single file containing the entire application, servable by the server. The server applications uses the standard built-in import statement of the Go programming language to import its modules. This supports direct importing of GitHub-hosted modules, using absolute GitHub-URL's as import paths for each module. In this project, all modules are imported by their absolute GitHub-URL, avoiding relative imports altogether. The structure within the server application consist of a Gin-framework enabled application which imports the Hilbert-hashing module within its route-handlers and uses this to answer client API requests for each URL-route. The Hilbert-hashing module resides in the 'geohash' folder and consists of four sub-modules each representing an object-type and the related functions for this type:

- bitarray - A module containg the BitAarray type definition and related functions.
- hash - The main hashing module containing the core hashing-functions.
- error - A module containing a simple Error-type used for error messages.
- point - A module containing the point-type used for representation of geographical points

4.2 Server

As mentioned previously, the server application is based upon the Go web framework Gin. This framework does the majority of the web-related tasks when a client request is processed. The main entry file for the server, 'app.go' resides in the root folder of the git repository and contains the basic configuration of the Gin framework, seen in Listing 4.1. This file declares usage of the Gin framework, sets a folder for static assets that should be served to clients, registers the URL routes for the server API and sets listening port for incoming requests. The static assets folder contains files that are externally accessible through the Gin web server i.e. using the URL: '<web-root>/static/<filename>'. This folder is also the target folder of the Webpack bundling process and the entire bundled JavaScript application becomes automatically accessible by URL after the bundling process. The listening port of server is set by an OS-specific environmental variable, this enables services like Heroku (Section 3.2.5) to set the port externally, while at the same time being also runnable locally without setting an environment variable, using the port 3000, i.e. a random port over 1024 avoiding privileged ports[26].

Listing 4.1: The file 'app.go' - The main entry file of the Go server application. Configures the Gin framework, setting a folder for static assets, registers URL routes and sets the listening port for incoming requests.

```go
package main

import (
  "os"

  "github.com/gin-gonic/gin"
  "github.com/tiborv/hilbert-geohash/handlers"
)

func main() {
  port := os.Getenv("PORT")

  router := gin.New()
  router.Use(gin.Logger())
  router.Static("/static", "static")
  handlers.Register(router)
  router.LoadHTMLGlob("templates/*")

  if port == "" {
    router.Run(":" + "3000")
  } else {
    router.Run(":" + port)

  }
}
```

Routing, the process of mapping incoming request to the functions that answer them, is done by registering these function to the Gin router. These registered functions, also known as handlers, are each registered with a corresponding URL-pattern that is the router uses on request to find which handler function to execute, as seen in Listing 4.2. Web frameworks often use regular expressions (regex) [27] for this URL-to-function matching process, but the Gin framework does not support full regex and does this in favor of performance and code simplicity. There are two registration-functions used in this project, and these determine what type of request[28] a handler should accept i.e. an HTTP-GET or an HTTP-POST. Each registration function takes two parameters: The URL-pattern used for request matching i.e. a string, and the corresponding handler function that is to be executed when a URL-match occurs. In addition, each handler function is defined having a Gin-context variable as a parameter. This parameter is the interface that can be used to access the deserialized content of a request from within the handler, e.g. post data if an HTTP-POST request received. In addition, the Gin-context variable has a serialization method for JSON encoding, which is used at the end of each handler function to answer the client request. There are three different handlers in this project: A handler for conversion from a geographical point to a Hilbert-hash, one for converting back from a Hilbert-hash to a point and one for serving the web page. The API routes all have the '/api' prefix as the URL-pattern, while all other valid URL's will only show index page from the 'templates' folder. Finally, there is a string reversing function present in this module. This is used to reverse the input and output of the Hilbert-hashing functions, changing the least-to-most significant value order from left to right, e.g: The hash: 11111_2 should be geographically close to the hash: 11112_2 and not to hash: 21111_2.

Listing 4.2: The file 'handlers/handlers.go' - The module that registers handler-functions to the Gin-router.

```go
package handlers

import (
  "net/http"

  "github.com/gin-gonic/gin"
  "github.com/tiborv/hilbert-geohash/geohash/hash"
  "github.com/tiborv/hilbert-geohash/geohash/point"
)

func Register(r *gin.Engine) {
  api(r.Group("/api"))

  r.GET("/", showIndex)
  r.GET("/h/:_", showIndex)
  r.GET("/p/:_", showIndex)

}

func showIndex(c *gin.Context) {
  c.HTML(http.StatusOK, "index.html", gin.H{
    "title": "Hilbert GeoHash",
  })
}

func api(api *gin.RouterGroup) {
  api.POST("/point", func(c *gin.Context) {
    p := point.Point{}
    err := c.Bind(&p)
    hash, err := hash.NewHashPoint(p)
    if err != nil || !p.IsValid() {
      c.JSON(http.StatusBadRequest, "Invalid point")
    } else {
      c.JSON(http.StatusOK, gin.H{
        "point":  p,
        "hash":   reverseString(hash.String),
        "zorder": reverseString(hash.GetZorder().HashString(32)),
      })
    }
  })

  api.POST("/hash", func(c *gin.Context) {
    hash := &hash.Hash{}
    err := c.Bind(&hash)
    hash.String = reverseString(hash.String)
    if err != nil || hash.InitFromString() != nil {
      c.JSON(http.StatusBadRequest, "Invalid hash")
    } else {
      c.JSON(http.StatusOK, gin.H{
        "hash":   reverseString(hash.String),
        "point":  hash.GenPoint(),
        "zorder": reverseString(hash.GetZorder().HashString(32)),
      })
    }
  })
}

func reverseString(s string) string {
  runes := []rune(s)
  for i, j := 0, len(runes)-1; i < j; i, j = i+1, j-1 {
    runes[i], runes[j] = runes[j], runes[i]
  }
  return string(runes)
}
```

4.3 Geohash

The core hashing modules reside in the 'geohash' folder of the git repository and consist of four modules. Each module contains a Go-type definition and related exported and non-exported functions. This structure is analogous to the object-oriented programming design pattern, except that instead of having an explicit keyword for 'private' and 'public' functions, Go uses first letter capitalization of function names for public functions and small letters for private functions. The main hashing algorithm resides in the hash module and contains the hash-type definition and the related functions used to initiate the hashing processes. The hashing process is done in a two-step fashion: First, the creation of a Z-order bit array and then a conversion from the Z-order array to a Hilbert-order array. The inverse procedure is done similarly, with a backward procedure order and using inverse functions.

4.3.1 Modules

As mentioned earlier, each hash-module within the geohash folder contains a type definition. From these type definitions objects (known as 'structs' in Go) can be instantiated and further used. One of these modules, the error module, contains a simple error type definition for creating error-objects based on strings and this module will not be further discussed within this research. The three other modules on the other hand: the 'bitarray' module, the point module and the hash module are all relevant to the core hashing procedures.

BitArray module

The Hashing methods described in this research, often do various operations on finite sequences of bits. This has introduced the need of having a BitArray type with a simple interface that supports all the needed operations. The main supported operations are:

Listing 4.3: Excerpt from the file 'geohash/bitarray/ba.go' showing the BitArray type definition and some related functions.

```
type BitArray struct {
  arr uint64
  len uint64
}

func NewBitArray(val uint64, len uint64) BitArray {
  return BitArray{arr: val, len: len}
}

func (ba *BitArray) SetB(b, pos uint64) uint64 {
  ba.arr = ba.arr | (b << pos)
  return b
}

func (ba *BitArray) AppendPair(b uint64) uint64 {
  ba.SetB(b, ba.len)
  ba.len += 2
  return b
}

func (ba BitArray) HashString(base int) string {
  return strconv.FormatUint(ba.arr, base)
}
```

- Set - Set bits within the array.
- GetPair - Get the value of two bit at a given position.
- AppendPair - Add two bits at the end of current length of the array.
- HashString - Create a string representation of the bit array using some base e.g. base 32.

The BitArray simple type definition can be seen in Listing 4.3. Unsigned integers of 64-bit lengths are used to store both the bit sequence, the 'arr' field, and the current amount of bits in use, the 'len' field. Keeping track of how many bits are actually in use, is needed since the bit length of 'arr' is always 64 with the 'unused' bits set to 0. By keeping track of current array length, the AppendPair function can easily insert a pair of bits at the correct position using the 'SetB' function. The 'SetB' function uses Go bit operators[29] to left-shift the input bits to the appropriate position and use OR-operator to create a new array with the bits inserted. Finally, the 'HashString' function uses the Go built-in string formatting function 'FormatUint' to generate hash-string using various bases.

Listing 4.4: Excerpt from the file 'geohash/point/point.go' showing the Point type definition and related functions.

```go
type Point struct {
  Lat    float64 `form:"lat" json:"lat" validate:"exists"`
  Lng    float64 `form:"lng" json:"lng" validate:"exists"`
  ErrLat float64 `form:"latErr" json:"latErr"`
  ErrLng float64 `form:"lngErr" json:"lngErr"`
}

const (
  MAX_LATITUDE  = 90
  MIN_LATITUDE  = -MAX_LATITUDE
  MAX_LONGITUDE = 180
  MIN_LONGITUDE = -MAX_LONGITUDE
)

func NewPoint(lat, lng, errLat, errLng float64) Point {
  return Point{Lat: lat, Lng: lng, ErrLat: errLat, ErrLng: errLng}
}

func (p Point) WithinErr(other Point) bool {
  return p.Lng-p.ErrLng <= other.Lng+other.ErrLng &&
    p.Lng+p.ErrLng >= other.Lng-other.ErrLng &&
    p.Lat-p.ErrLat <= other.Lat+other.ErrLat &&
    p.Lat+p.ErrLat >= other.Lat-other.ErrLat
}

func (p Point) IsValid() bool {
  return p.Lat <= MAX_LATITUDE &&
    p.Lat >= MIN_LATITUDE &&
    p.Lng <= MAX_LONGITUDE &&
    p.Lng >= MIN_LONGITUDE
}
```

Point module

Geographical points used within this research are simple two-dimensional entities. A point can, therefore, be represented by a Go type using two fields of type float, one for each coordinate value. The Point definition and related functions can see in Listing 4.4. The point type definition declares two floats of bit length 64 for storing the latitude and longitude coordinates, with two additional floats for the precision errors for each coordinate. The definition also includes deserialization-rules used by the Gin framework to map how the JSON-fields from the incoming request, should be parsed into the Go type definition. This enables a handler function to use Gin-context Bind function to automatically parse the request payload into a Go object instance. The precision errors are introduced because points generated from the inverse-hashing procedure can be of variable precision. As described in Section 3.1.1, the number of values used in a hash determines the size of the area the hash points towards .i.e. Hashes with few values point to large areas while hashes with many values have more precision and point to smaller areas. This effect is observable and described further in Section 4.3.4.

Hash module

The hash module contains the core hashing Hilbert-curve hashing functions. The hash-type definition uses two BitArrays types and a string as the underlying data types, as seen in Listing 4.5. The BitArray fields are used for intermediate storing of Z-order and Hilbert-curve bit arrays, while the string is used to store the base 32 representation of the Hilbert-curve bit array. When a point is to be hashed, the 'NewHashPoint' function is called with the point provided as a parameter. The function checks if this particular point has valid coordinates by using the Point function: 'isValid' and proceeds to generate a Z-order BitArray of this point using the 'genZorder' function. Finally, a Hilbert curve based BitArray can be made using the 'genHilbert' function, supplying the zorder array as a parameter. The reverse procedure of going from a hash to a point is done using the 'NewHashString' function. This function is used to instantiate a new Hash object from a hash-string, using the 'InitFromString' function, which uses 'genHilbertInverse' to generate a Z-order BitArray. The function GetPoint can now be called on this instantiated Hash object to generate the corresponding geographical point, and this is done externally by the hash handler-function shown in Listing 4.2.

Listing 4.5: Excerpt from the file 'geohash/hash/hash.go' showing the Hash type definition and initialization functions.

```go
import (
  "strconv"
  "unicode/utf8"

  "github.com/tiborv/hilbert-geohash/geohash/bitarray"
  "github.com/tiborv/hilbert-geohash/geohash/error"
  "github.com/tiborv/hilbert-geohash/geohash/point"
)

type Hash struct {
  String  string `form:"hash" json:"hash" validate:"exists"`
  zorder  ba.BitArray
  hilbert ba.BitArray
}

func NewHashPoint(p point.Point) (Hash, error) {
  if !p.IsValid() {
    return Hash{}, errors.New("Invalid Point")
  }
  zorder := genZorder(p)
  hilbert := genHilbert(zorder)
  return Hash{hilbert: hilbert,
      zorder: zorder,
      String: hilbert.HashString(32)},
    nil
}

func NewHashString(hash string) (Hash, error) {
  h := Hash{String: hash}
  err := h.InitFromString()
  return h, err
}

func (h *Hash) InitFromString() error {
  val, err := strconv.ParseUint(h.String, 32, 64)
  bitArraylen := uint64(0)
  if h.String != "" {
    if err != nil {
      return err
    }
    //Each base32 value adds 5 bits
    bitArraylen = uint64(utf8.RuneCountInString(h.String)*5 - 1)
    if bitArraylen > 64 {
      return errors.New("Hash too long")
    }
  }
  h.hilbert = ba.NewBitArray(val, bitArraylen)
  h.zorder = genHilbertInverse(h.hilbert)
  return nil
}
```

4.3.2 The Hashing algorithms genZorder and genHilbert

As mentioned earlier, the core hashing algorithms reside in the hash-module and are called 'genZorder', seen in Listing 4.6 and 'genHilbert' seen in Listing 4.7. The genZorder function takes a point-object as a parameter and produces a z-ordered BitArray object, while the genHilbert function takes a Z-ordered BitArray and returns a Hilbert-ordered BitArray. The hashing process is, therefore, a two-step procedure: The conversion from a Point to a Z-order BitArray and the conversion from Z-order curve to Hilbert curve. 'genZorder' uses a simple for-loop with a switch-statement to iteratively generate bit-pairs which are appended to a BitArray later returned as the result. For each iteration, a search space is defined by using two variables 'latdevide' and 'lngdevide'. These variables divide the space vertically at the 'latdevide' value and horizontally at the value of 'lngdevide'. Initially, the search space is the entire point-space, and this is divided at the latitude and longitude position 0 i.e. at the center. Then, the 'setZorderQuadrant' function uses these division positions in a two-dimensional binary search manner: If the input points latitude value is bigger than the division value, a 1 becomes the first bit of the result bit-pair, else this bit is a 0. This is done for the longitude value as well, comparing it to the 'lngdevide' variable and determining the last bit of the bit-pair. The numerical values 0,1,2 and 3 are used to represent these bit pairs since they yield the desired bit-pairs when used with binary operators, i.e. 3 becomes the bit-pair 11_2, 2 becomes 10_2 etc. For each iteration, the search space is halved in both dimensions, creating smaller and smaller areas for which the point is evaluated over. The for-loop is repeated 32 times, each time generating 2 bits i.e. in total, 64 bits are appended to the resulting BitArray.

Listing 4.6: Excerpt from the file 'geohash/hash/hash.go' showing the 'genZorder' function and the related 'setZorderQuadrant' function.

```go
func genZorder(p point.Point) ba.BitArray {
  zorder := ba.NewBitArray(0, 0)
  latdevide, lngdevide := float64(0), float64(0)
  lat, lng := float64(point.MAX_LATITUDE), float64(point.MAX_LONGITUDE)
  for i := 0; i < 32; i++ {
    lat /= 2
    lng /= 2
    switch setZorderQuadrant(&zorder, p, latdevide, lngdevide) {
    case 3:
      latdevide += lat
      lngdevide += lng
    case 2:
      latdevide −= lat
      lngdevide += lng
    case 1:
      latdevide += lat
      lngdevide −= lng
    case 0:
      latdevide −= lat
      lngdevide −= lng
    }
  }
  return zorder
}

func setZorderQuadrant(zorder *ba.BitArray,
  p point.Point, latdevide, lngdevide float64) uint64 {
  if p.Lng > lngdevide {
    if p.Lat > latdevide {
      return zorder.AppendPair(3)
    }
    return zorder.AppendPair(2)
  }
  if p.Lat > latdevide {
    return zorder.AppendPair(1)
  }
  return zorder.AppendPair(0)

}
```

After the creation of the Z-order BitArray, a Hilbert ordering of this array can be initiated. This is done by calling the 'genHilbert' function and passing the newly created Z-order BitArray as a parameter. This function uses a state-based algorithm to map a Z-ordered bit array to a Hilbert-ordered one. There are four different states, each state with its own state number, 0-3, representing the Hilbert states seen in Figure 3.3. The iterative process uses an index-value 'i' to keep track of current evaluation pair of the Z-order array and this value is increased by 2 each iteration. This is done because the bits are processed in pairs, both when retrieved from the Z-order input array and when used as keys in the Hilbert mapping. Each iteration uses the 'getHilbertMap' function seen in 4.7, to get Hilbert-mapping result. This function takes two parameters: A mapping-key i.e. the Z-order bit pair, and a state value determining which Hilbert-map should be used with this mapping-key. The four Hilbert-maps saw in Listing 4.9, map from a Z-order bit pairs to both the Hilbert-curve result pairs and next-state values. Thus, the 'getHilbertMap' function returns two values: The Hilbert-mapping result pair and the next Hilbert state. The Hilbert-mapping result is appended to the result BitArray, and this array is in the end returned upon loop-completion.

Listing 4.7: Excerpt from the file 'geohash/hash/hash.go' showing the genHilbert function.

```go
func genHilbert(zorder ba.BitArray) ba.BitArray {
  hilbertValue := uint64(0)
  nextState := 0
  ba := ba.NewBitArray(0, 0)
  for i := uint64(0); i < zorder.Len(); i += 2 {
    hilbertValue, nextState = getHilbertMap(zorder.GetPair(i), nextState)
    ba.AppendPair(hilbertValue)
  }
  return ba
}
```

Listing 4.8: Excerpt from the file 'geohash/hash/state.go' showing the getHilbertMap function, using the Hilbert-maps defined seen in Listing 4.9.

```go
func getHilbertMap(zvalue uint64, currentState int) (uint64, int) {
  switch currentState {
  case 0:
    return map0[zvalue].v, map0[zvalue].s
  case 1:
    return map1[zvalue].v, map1[zvalue].s
  case 2:
    return map2[zvalue].v, map2[zvalue].s
  case 3:
    return map3[zvalue].v, map3[zvalue].s
  default:
    panic("Unkown HilbertMapper")
  }
}
```

Listing 4.9: Excerpt from the file 'geohash/hash/state.go' showing the State-type definition, and four hilbert-mappers.

```go
type HilbertMapper struct {
  v uint64 //Value
  s int    //Next HilbertMapper
}

var map0 = map[uint64]HilbertMapper{
  0: HilbertMapper{v: 0, s: 0},
  1: HilbertMapper{v: 1, s: 1},
  2: HilbertMapper{v: 3, s: 2},
  3: HilbertMapper{v: 2, s: 1},
}

var map1 = map[uint64]HilbertMapper{
  0: HilbertMapper{v: 0, s: 1},
  1: HilbertMapper{v: 3, s: 3},
  2: HilbertMapper{v: 1, s: 0},
  3: HilbertMapper{v: 2, s: 0},
}

var map2 = map[uint64]HilbertMapper{
  0: HilbertMapper{v: 2, s: 2},
  1: HilbertMapper{v: 1, s: 2},
  2: HilbertMapper{v: 3, s: 1},
  3: HilbertMapper{v: 0, s: 3},
}

var map3 = map[uint64]HilbertMapper{
  0: HilbertMapper{v: 2, s: 3},
  1: HilbertMapper{v: 3, s: 0},
  2: HilbertMapper{v: 1, s: 3},
  3: HilbertMapper{v: 0, s: 2},
}
```

4.3.3 The inverse algorithms genHilbertInverse and genPoint

As with the hashing procedure, the inverse procedure is also a two step process: A hilbert inverse function followed by a point generation function. The hilbert inverse function, ’genHilbertInverse’ seen in Listing 4.10, take a Hilbert ordered BitArray-type object and generates a Z-ordered BitArray. This function is smilar to the ’genHilbert’ function described in Section 4.3.2, using a state-based algorithm to generate a Z-ordered result. The only difference between the ’genHilbert’ and ’genHilbertInverse’ functions is the usage of the mapping functions: ’getHilbertMap’ and ’getHilbertInverseMap’. These mapping functions are also quite similar, but differ in the Hilbert mapping functions. The ’genHilbertInverseMap’, seen in Listing 4.11, uses the inverse Hilbert-mappers, inverse0 - inverse3, to map back from a Hilbert-value to a Z-order. Both mapping

Listing 4.10: Excerpt from the file ’geohash/hash/hash.go’ showing the ’genHilbertInverse’ function.

```
func genHilbertInverse(hilbert ba.BitArray) ba.BitArray {
  zorder := ba.NewBitArray(0, 0)
  nextState := 0
  zorderValue := uint64(0)
  for i := uint64(0); i < hilbert.Len(); i += 2 {
    zorderValue, nextState = getHilbertInverseMap(hilbert.GetPair(i), nextState)
    zorder.AppendPair(zorderValue)
  }
  return zorder
}
```

Listing 4.11: Excerpt from the file 'geohash/hash/state.go' showing the 'getHilbertInverseMap' and the inverse Hilbert mappers: inverse0 - inverse3.

```
func getHilbertInverseMap(hilbertvalue uint64, currentState int) (uint64, int) {
  switch currentState {
  case 0:
    return inverse0[hilbertvalue].v, inverse0[hilbertvalue].s
  case 1:
    return inverse1[hilbertvalue].v, inverse1[hilbertvalue].s
  case 2:
    return inverse2[hilbertvalue].v, inverse2[hilbertvalue].s
  case 3:
    return inverse3[hilbertvalue].v, inverse3[hilbertvalue].s
  default:
    panic("Unkown HilbertMapper")
  }
}

var inverse0 = map[uint64]HilbertMapper{
  0: HilbertMapper{v: 0, s: 0},
  1: HilbertMapper{v: 1, s: 1},
  2: HilbertMapper{v: 3, s: 1},
  3: HilbertMapper{v: 2, s: 2},
}

var inverse1 = map[uint64]HilbertMapper{
  0: HilbertMapper{v: 0, s: 1},
  1: HilbertMapper{v: 2, s: 0},
  2: HilbertMapper{v: 3, s: 0},
  3: HilbertMapper{v: 1, s: 3},
}

var inverse2 = map[uint64]HilbertMapper{
  0: HilbertMapper{v: 3, s: 3},
  1: HilbertMapper{v: 1, s: 2},
  2: HilbertMapper{v: 0, s: 2},
  3: HilbertMapper{v: 2, s: 1},
}

var inverse3 = map[uint64]HilbertMapper{
  0: HilbertMapper{v: 3, s: 2},
  1: HilbertMapper{v: 2, s: 3},
  2: HilbertMapper{v: 0, s: 3},
  3: HilbertMapper{v: 1, s: 0},
}
```

Listing 4.12: Excerpt from the file 'geohash/hash/hash.go' showing the 'GenPoint' function.

```
func (h Hash) GenPoint() point.Point {
  reslng, reslat := float64(0), float64(0)
  lat, lng := float64(point.MAX_LATITUDE), float64(point.MAX_LONGITUDE)
  errLat, errLng := float64(point.MAX_LATITUDE), float64(point.MAX_LONGITUDE)
  for i := uint64(0); i < h.zorder.Len(); i += 2 {
    lng /= 2
    lat /= 2
    errLat /= 2
    errLng /= 2
    switch h.zorder.GetPair(i) {
    case 3:
      reslat += lat
      reslng += lng
    case 2:
      reslat -= lat
      reslng += lng
    case 1:
      reslat += lat
      reslng -= lng
    case 0:
      reslat -= lat
      reslng -= lng
    }
  }
  return point.NewPoint(reslat, reslng, errLat, errLng)
}
```

4.3.4 Precision

The size of the area represented by the Point-type seen in Listing 4.4, defines the precision of the point. This area is defined by its position i.e. latitude and longitude coordinates, and the error values for these coordinates, 'errLat' and 'errLng'. If the error values for both coordinates are 0, the point has infinite precision and covers an infinitely small area. This opposite of this, a point with error values bigger than zero, creates a rectangular area which represents the geographical space this point covers. This is useful within the inverse hashing procedure, where a point is generated from a hash string with variable length. The more values a hash string has, the smaller the error values of the generated point becomes. This procedure is seen in Listing 4.12, showing the 'genPoint' function. In this function, the variables 'errLat' and 'errLng' are sequentially computed, starting from their maximum amount, 90 and 180, and halved for each iteration. Thus, the initial point has error values ±90 latitude and ±180 longitude, covering the entire point space. After one iteration, these error values are halved to 45 and 90, continuing to increase the precision. The process stops once there are no more hash-values to process. This effect can be seen in the GUI of the client application in Section 4.4.2, where red rectangles are used to represent the error-area of each point. The error areas of the points increase in size as the hash-input decreases in length. This property of the system can be used to shorten the hash in situations where high geographical precision is not needed e.g. location of a city.

4.4 Client

The client application resides in the 'client' folder of the git-repository of this project. The main functionality of this application is to provide an interface for the user to interacting with the back-end server application. This is achevied using a JavaScript application, enabling the usage of other JavaScript modules hosted on NPM, as described in Section 3.2.4. The functionality of the client application can be summarized in four steps:

1. Load the Google Maps API and the geographical map.
2. Check the current browser URL to figure out which URL-route is accessed.
3. Use the URL-data to send a request to the server application.
4. Display result of request onto the geographical map.

The main entry file for the client application, 'client/app.js' seen in Listing 4.13, shows how these steps are executed. When a URL is requested, the server application answers with the same HTML-page on all requests, containing the same JavaScript client application file. It is then up to the client to figure out what URL-has been used to access this route, by checking the browser input. The URL-input is extracted from the browser using the 'getUrlParams' function, within the main routing function 'route', and this loads both the path and the parameters of the entered URL, e.g.: The url '/p/12.34,56.78' becomes the path '/p/' and the paramteres '12.34,56.78'. From this a request can be generated, depending on what URL-route was used, and sent to the correct sever API endpoint. As with the server application, the client supports two types of input data: Hashes and geographical points. The URL-prefix for hashes is '/h/', while for points '/p/'. The file 'client/js/gui.js' contains the GUI-controller functions 'showHash' seen in Listing 4.14, and 'showPoint' seen in Listing 4.15, which are called by the 'route' function when it has decided what URL-route has been accessed. When 'showHash' or 'showPoint' has been called, the URL-parameters are provided as parameters for these functions. 'showHash' and 'showPoint' then use the API-functions 'getHash' or 'getPoint' to communicate withe API of server application, seen in Listing 4.16. The result from the API-requests are returned back to the GUI-controller functions, which use various functions update the client GUI. The entirety of the client application can be found in the 'client' folder of the root repository folder.

Listing 4.13: The main entry file for the client application 'client/app.js'

```
"use strict";
import './styles/main.css';
import { selectElement as $, setPushStateListener, getUrlParams } from './js/helpers';
import { init, showHash, showPoint } from './js/gui';

let route = ()=> {
  let [path, param] = getUrlParams(3); //Gets the current browser URL splitted: Path and params
  switch (path) {
    case '/h/': //Url's starting with /h/ are hashes
      showHash(param); //Parmeter is a hash
      break;
    case '/p/': //Url's starting with /p/ are points
      let [lat,lng] = param.split(',');//Split params into lat,lng coordinates
      showPoint(lat,lng);
      break;
    default: //Everything else: Show home.
      showPoint(63.41678857590608,10.402773320674896);
  };
}
//Init the GUI.
init($('hashInput'),
  $('pointInput'),
  $('hash'),
  $('point'),
  $('pathButton'),
  $('map')).then(route);

// Update view when URL has changed.
setPushStateListener(route);
```

Listing 4.14: Excerpt from the file 'client/js/gui.js' showing the 'showHash' function.

```
  getPoint(hash)
    .then(res=>{
      setMarker(res.point);
      setInfo(res);
      setZoom(res.point);
      pushState(`/h/${res.hash}`,`Hash:${res.hash}`);
    });
}

export function showHashPath(hash){
```

Listing 4.15: Excerpt from the file 'client/js/gui.js' showing the 'showPoint' function.

```
    .then(res=>{
      setMarker(res.point);
      setInfo(res);
      setZoom(res.point);
      pushState(`/p/${lat},${lng}`,`Point:${lat},${lng}`);
    });
}
```

Listing 4.16: Excerpt from the file 'client/js/api.js' showing the functions 'getPoint' and 'getHash'.

```
export function getPoint(hash) {
  return new Promise(resolve=>{
    request.post('/api/hash')
      .send({hash})
      .end((err,res)=>err ? alertify.error("Invalid hash"):resolve(res.body));
  });
}


export function getHash(lat,lng) {
  lat = typeof lat == "string" ? Number(lat):lat;
  lng = typeof lng == "string" ? Number(lng):lng;
  return new Promise(resolve=>{
    request.post('/api/point')
      .send({lat,lng})
      .end((err,res)=>err ? alertify.error("Invalid point"):resolve(res.body));
  });
}
```

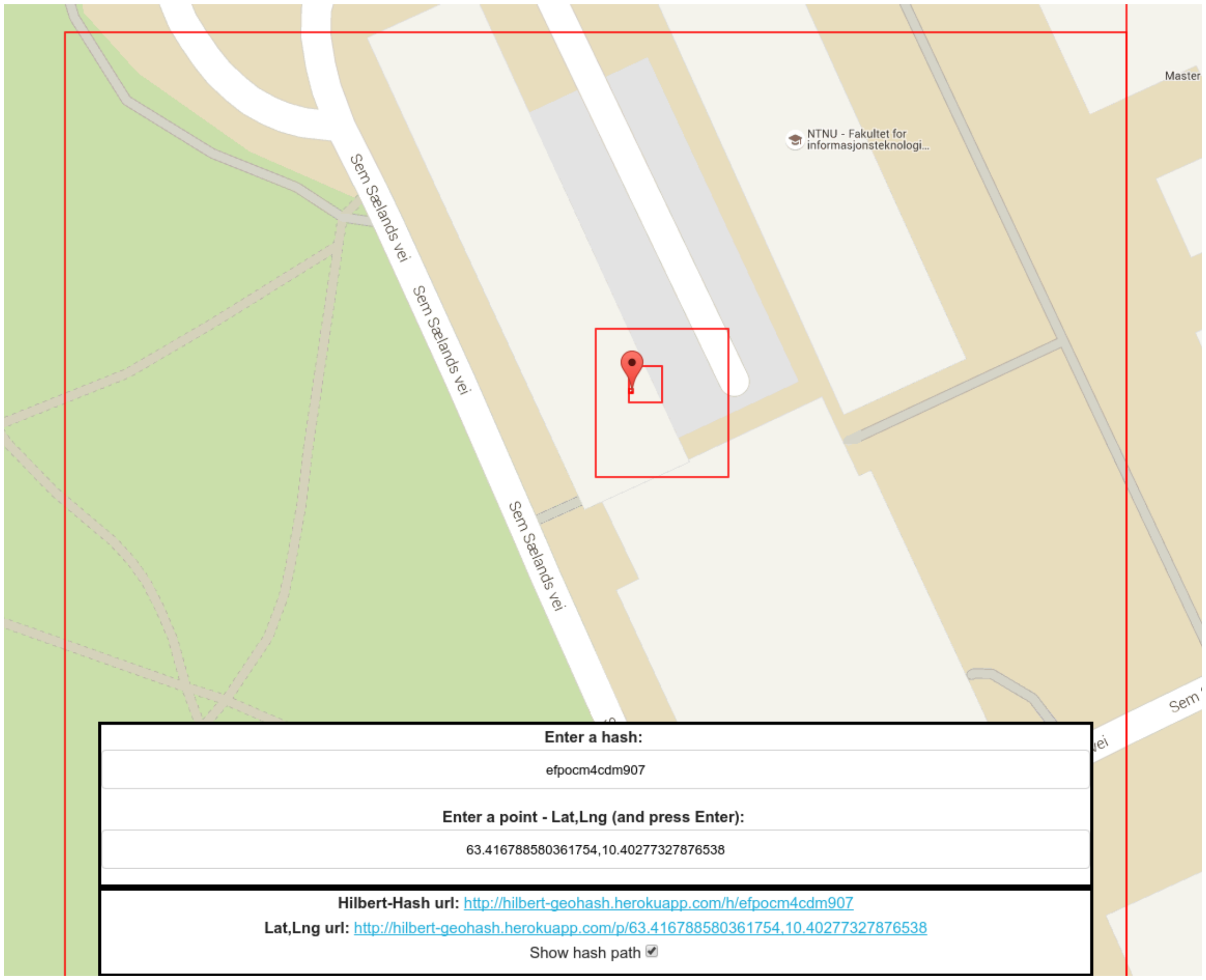
4.4.1 Client Application Bundling

As mentioned earlier, Webpack bundles the entire JavaScript client application into one single file. This file is served by the Go-server as a static assets i.e. placed in the Gin-configured static folder: '/static'. To future-proof the application, a modern JavaScript syntax specification is used. ECMAScript 6 (ES6) is the 6th, and currently newest, specification of the JavaScript programming language by the European Computer Manufacturers Association (ECMA) [30]. This specification is currently not fully supported by all the major Internet browsers[31], which introduces compatibility problems. This problem is solved by using the NPM-module Babel[32], a JavaScript ES6-to-ES5-compiler that compiles the code back to the previous, widely supported, ECMAscript 5 standard.

The Babel compiler is used in the bundling process together with Webpack, described in Section 3.2.4, and can simply be removed in the future when the ES6 standard becomes more natively supported by the major browsers.

4.4.2 Interface

The interface of the client application can be seen in Figure 4.1. It is a simple interface, having mainly two input areas: One for hashes and one for points. The hash-input field, auto updates the view upon key-entry and moves the position of the center map-marker, while the point-input field only updates the view when the user presses enter. When a hash is entered, the interface draws a rectangle on the geographical map to illustrate its precision. As mentioned earlier, short hashes generate large error-rectangles while long ones yield small error-rectangles. The checkbox 'Show hash path', when activated, show the error-rectangle of all shorter hashes with the same prefix. An example of this is the hash 'efpocm4cdm907' seen in Figure 4.1. Here the smallest centered rectangle, represents the error of the entire hash string. The second smallest rectangle represents the hash with one less value: 'efpocm4cdm90'. The one after this: 'efpocm4cdm90', 'efpocm4cdm9', each with one less hash-value than the previous. This illustrates the algorithm's search path towards this point.



Enter a hash:

efpocm4cdm907

Enter a point - Lat,Lng (and press Enter):

63.416788580361754,10.40277327876538

Hilbert-Hash url: http://hilbert-geohash.herokuapp.com/h/efpocm4cdm907

Lat,Lng url: http://hilbert-geohash.herokuapp.com/p/63.416788580361754,10.40277327876538

Show hash path ☑

Figure 4.1: A screenshot of the interface showing the hash 'efpocm4cdm907' with path illustration activated.

Chapter 5

Evaluation of the Hilbert curve

In this chapter, the implemented Hilbert-curve-based hashing algorithm is compared to the Z-order-based algorithm. Techniques of measuring the curve locality properties are proposed and used together with three datasets to generate the results. In addition, a simple benchmarking test is done to measure the complexity differences between the curves.

5.1 Motivation

As mentioned in Section 2.4.3, the locality properties of the Hilbert-curve should be superior to the Z-order curve. To confirm that this also holds for the implemented system, an evaluation needs to be done which involves comparing the hash-space properties of both curves. The assumption is that better curve-locality yields closer hash value-prefixes between two geographically close points, i.e. there are fewer situations where the hash values of two points are close, but their geographical position is distant. Thus, distance measurement methods need to be introduced that are able to tell distances between the hash values of two points. In addition, some simple benchmarking is done to measure the additional processing costs of executing the Hilbert curve, over using purely the Z-order curve.

5.2 Hash Distance Measurement

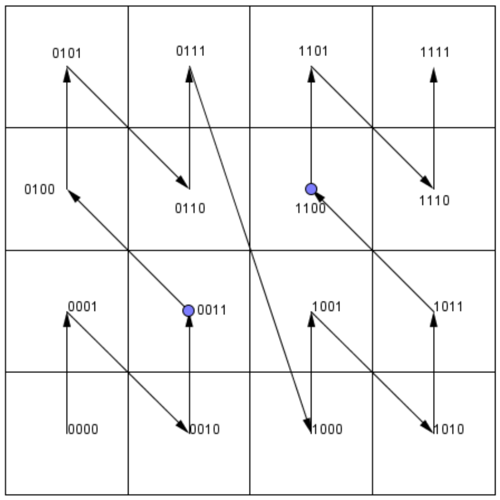
To generate more conclusive results, two separate ways of measuring distance are introduced: Curve distance and Levenshtein distance, also known as edit distance. The Curve distance measurement is based on measuring how many steps along the space-filling curve there are between two point-hashes, while the edit distance focuses purely on string-similarities between these hashes. The edit distance measurement has more of a practical motivation as it directly compares the hash value strings produced by both curves, checking if one curve produces more string similarities than the other between points of equal geographical distance. The curve distance measurement is theoretically motivated, as there is a possibility that the clustering capabilities of the curves, do not directly translate to string-similarities.

5.2.1 Curve distance

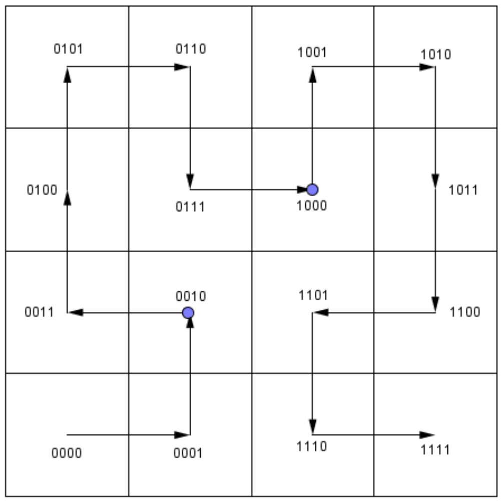
The Section 4.3.1 describes the BitArray-type, used for representing the hash values. The underlying Go base type for this structure, the 'arr' variable seen in Listing 4.3, is an unsigned integer of 64-bit length (uint64) and is used for storing the hashes. An unsigned integer is a non-negative integer datatype, using all its internal bits for representation of value and not signedness. Comparing curve types can, therefore, be done in the manner seen in Figure 5.1, doing a simple subtraction while making sure that no unsigned integer overflows happen i.e. avoiding negative results. This is done using the comparator function 'Diff' seen in Listing 5.1, which avoids overflow and returns the difference between two BitArrays. The 'Diff' function is then used in both Hash-comparator functions, one for each curve type, seen in Listing 5.2.

5.2.2 Levenshtein distance

Levenshtein distance, also known as edit distance, is a string comparison measurement indicating how many edits need to be done to change one string into another. There are three types of string-edits allowed: Insertions, deletions, and substitutions. An example of this is the comparison of the strings 'aaab' 'and 'bb', where going from one to the other requires two 'a's to be removed/prepended and one 'b' to be changed to an 'a'. The edit distance between 'aaab' and 'bb' is therefore: $2 + 1 = 3$. Using this, the hash values of two points can be compared between curves, checking if the Hilbert curve has shorter edit-distances between points than the Z-order curve. The function used, named 'DistanceLevenshtein' residing in 'geohash/stat.go', is a copy of the Go implementation of the Levenshtein distance algorithm found on rosettacode.org[33].



(a) Z-order hash-space distance:
$1100_2 - 0011_2 = 1001_2$



(b) Hilbert hash-space distance:
$1000_2 - 0010_2 = 0110_2$

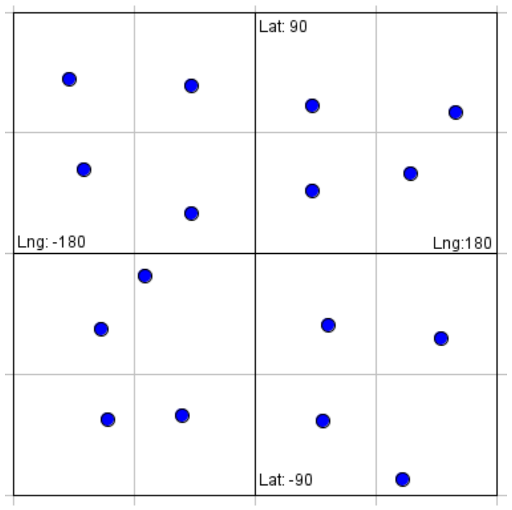
Figure 5.1: The same points mapped on both curves, illustrating the hash-space difference between them. For this point-pair the Hilbert-curve has the smallest distance: $1001_2 > 0110_2$. This can also bee seen by just counting the steps between the points, adding up to 9 for the Z-order curve and 5 for the Hilbert curve.

Listing 5.1: Excerpt from the file 'geohash/bitarray/ba.go' showing the 'Diff' function.

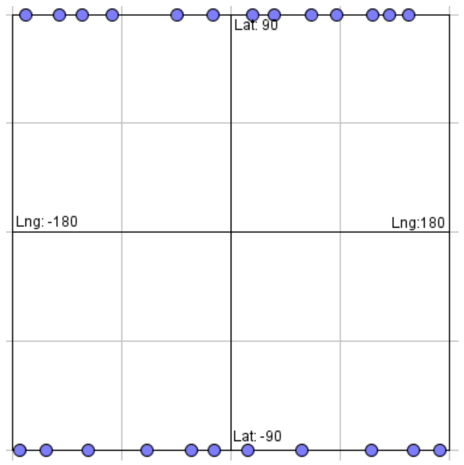
```
func (ba BitArray) Diff(other BitArray) uint64 {
  if ba.arr > other.arr {
    return ba.arr − other.arr
  }
  return other.arr − ba.arr
}
```

Listing 5.2: Excerpt from the file 'geohash/hash/hash.go' showing the 'DistanceZorder' and the 'DistanceHilbert' functions.

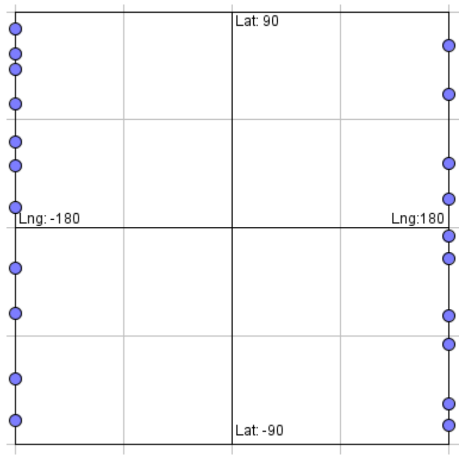
```
func (h Hash) DistanceZorder(to Hash) uint64 {
  return h.zorder.Diff(to.zorder)
}

func (h Hash) DistanceHilbert(to Hash) uint64 {
  return h.hilbert.Diff(to.hilbert)
}
```



(a) Randomly placed points



(b) Maximum Latitude: Random max and min latitude points.



(c) Maximum Longitude: Random max and min longitude points.

Figure 5.2: Illustrations of the three data types used.

5.3 Test Data Generation

There are three types of randomly generated data sets used for testing hash distances:

- Randomly positioned points.
- Maximum Latitude points: Points with fixed latitude values 90 and -90, and random longitude values.
- Maximum Longitude points: Points with fixed longitude values 180 and -180, and random latitude values.

These three data set generation types are illustrated in Figure 5.2. The points in Figure 5.2b and Figure 5.2c should be compared creating maximum distances between point comparison pairs i.e. one point from each side are compared with each other. The points in Figure 5.2a can be compared without considering their position.

5.4 Test execution

As mentioned earlier, two types of tests are executed: Hash distance and performance benchmarking. The hash distance tests are executed without any changes to the core geohashing source code, while the benchmarking tests are executed on two different versions of the code. The benchmarking results are therefore not directly reproducible, without manually removing the Hilbert-curve related functions from the hashing procedure to generate Z-order-only results.

5.4.1 Hash Distance

The randomly positioned points are generated within the function 'RunRandomPoints' seen in Listing 5.3. This function takes a number of runs, 'n', and a number of points to generate in each run, 'p', as parameters. One run has mainly three steps: The generation of 'p' points, the creation of hashes from these points and the comparison of these hashes. Each run uses the 'compareHashes' function seen in Listing 5.5, to compare two arrays of hashes against each other. This function simply iterates trough both hash arrays provided as parameters, h1, and h2 counts the amount of instances where one curve is closer than the other and uses the Levenshtein distance comparison function to compare hash edit distances. Finally, the function returns an average for both of these measurements. When running the random points dataset, one array of hashes is created and compared with itself i.e. providing the 'compareHashes' function two incidental hash arrays as parameters. For the other data sets, Maximum Lat and Maximum Lng, the hashes provided are two distinct arrays, each array representing points on one side. This enables comparisons of points on opposite sides of each other i.e. no two points on the same side in Figures 5.2c and 5.2b are compared. The Maximum Latitude and Maximum Longitude data sets are similarly generated as the random dataset, by the functions 'RunMaxDistancepLat' seen in Listing 5.4 and 'RunMaxDistancepLng', except that they don't use random values of both coordinates when they create points i.e. only one coordinate is random. The 'RunMaxDistanceLng' function will not be explicitly shown in this work, as this is considered redundant to show very similar code twice, but can be found in the same file as the other functions: 'geohash/stat.go'. Upon completion of the data generation function, a 'PrintResults' function seen in Listing 5.6, is used to average all 'n' results for both curve types, and finally, print the results.

Listing 5.3: Excerpt from the file 'geohash/stat.go' showing the 'RunRandomPoints' function.

```
func RunRandomPoints(n, p int) {
  fmt.Printf("\nRunning %v iterations, using %v RANDOM points...", n, p)
  resultsHilb := make([]float64, n)
  resultsZ := make([]float64, n)
  levenshteinDiff := make([]float64, n)
  for i := 0; i < n; i++ { //Repeat n times
    hashes := make([]hash.Hash, p)
    rand.Seed(time.Now().UTC().UnixNano())

    for i := 0; i < p; i++ { // Use p amount of points
      randLat := rand.Float64() * point.MAX_LATITUDE
      randLng := rand.Float64() * point.MAX_LONGITUDE
      p := point.NewPoint(randLat, randLng, 0, 0)
      hash, _ := hash.NewHashPoint(p)
      hashes[i] = hash
    }

    resultsHilb[i], resultsZ[i] = compareHashes(hashes, hashes, true)
    levenshteinDiff[i] = compareLevenshtein(hashes, hashes, true)

  }
  PrintResults(resultsHilb, resultsZ, levenshteinDiff)
}
```

Listing 5.4: Excerpt from the file 'geohash/stat.go' showing the 'RunMaxDistancepLat' function.

```go
func RunMaxDistancepLat(n, p int) {
  fmt.Printf("\nRunning %v iterations, using %v MAX DISTANCE LAT points...", n, p)
  resultsHilb := make([]float64, n)
  resultsZ := make([]float64, n)
  levenshteinDiff := make([]float64, n)
  for i := 0; i < n; i++ { //Repeat n times
    h1 := make([]hash.Hash, p/2)
    h2 := make([]hash.Hash, p/2)

    rand.Seed(time.Now().UTC().UnixNano())

    for i := 0; i < p/2; i++ { // Create p/2 amount of points on one side
      randLat := 1.0 * point.MAX_LATITUDE // Max latitude
      randLng := rand.Float64() * point.MAX_LONGITUDE
      p := point.NewPoint(randLat, randLng, 0, 0)
      hash, _ := hash.NewHashPoint(p)
      h1[i] = hash
    }
    for i := 0; i < p/2; i++ { // Create p/2 amount of points on other side
      randLat := -1.0 * point.MAX_LATITUDE //Opposite side maximum
      randLng := rand.Float64() * point.MAX_LONGITUDE
      p := point.NewPoint(randLat, randLng, 0, 0)
      hash, _ := hash.NewHashPoint(p)
      h2[i] = hash
    }
    resultsHilb[i], resultsZ[i] = compareHashes(h1, h2, false)
    levenshteinDiff[i] = compareLevenshtein(h1, h2, false)
  }
  PrintResults(resultsHilb, resultsZ, levenshteinDiff)
}
```

Listing 5.5: Excerpt from the file 'geohash/stat.go' showing the 'compareHashes' function.

```go
func compareHashes(h1, h2 []hash.Hash, equalInput bool) (float64, float64) {
  hilbertIsShorter, zordertIsShorter, checked := 0, 0, float64(0)
  for i := 0; i < len(h1); i++ {
    for j := 0; j < len(h2); j++ {
      if equalInput && i == j {
        continue //Skip comparison of same point when h1 == h2
      }
      hilbertDistance := h1[i].DistanceHilbert(h2[j])
      zorderDistance := h1[i].DistanceZorder(h2[j])
      if zorderDistance > hilbertDistance {
        hilbertIsShorter++
      }
      if zorderDistance < hilbertDistance {
        zordertIsShorter++
      }
      checked++
    }
  }
```

Listing 5.6: Excerpt from the file 'geohash/stat.go' showing the 'PrintResults' and the related Sum functions.

```go
func PrintResults(hilbRes, zorderRes, levenshteinRes []float64) {
  hilbAvg := 100 * Sum(hilbRes) / float64(len(hilbRes))
  zorderAvg := 100 * Sum(zorderRes) / float64(len(zorderRes))
  equal := 100 - (hilbAvg + zorderAvg)
  levenshteinAvg := Sum(levenshteinRes) / float64(len(levenshteinRes))
  fmt.Printf("\nThe Hilbert curve had a shorter distance %.2f%% of the time ", hilbAvg)
  fmt.Printf("\nThe Z-order curve had a shorter distance %.2f%% of the time ", zorderAvg)
  fmt.Printf("\nThe curves had eual distance %.2f%% of the time ", equal)
  fmt.Printf("\nThe Hilbert curve was on average %.2f shorter in Levenshtein distance\n",
    levenshteinAvg)
}
```

5.4.2 Benchmarking

The benchmarking tests are done using the built-in benchmarking and memory profile Golang tools. The two benchmarking functions, 'BenchmarkHash' and 'BenchmarkPoint', are used to execute the benchmarking and both functions reside in the 'geohash/hash/hash_test.go' file. These functions are simple, using randomly generated points and hashes, as parameters for the benchmarking-target functions: 'NewHashPoint' and 'NewHashString' from Section 4.3.1. Before each benchmark-target function is run, the benchmarking timer is reset. This is done to avoid having the creation of the random input data, affect the benchmarking results. The benchmarking tool decides by itself how many iterations are needed to get a stable reading of the performance of the functions, usually resulting various amounts of iterations. The benchmarks are run two times: One measuring the performance with the Hilbert-curve functions included and one without the Hilbert-function i.e. Z-order function only.

5.5 Results

The test results can be seen in Table 5.1. These results have been generated using 1000 iterations, each iteration using 1000 generated points. The results show that the Hilbert-curve based hashing method had a shorter hash-space distance between two randomly placed points 53.15% of the time. Using the random-skewed datasets, Maximum Latitude, and Maximum Longitude, the results are a bit more distinct: 58.14% of all checks using the Maximum Latitude data set had the Hilbert-based hash with the shortest distance, while with the Maximum Longitude data set the Hilbert curve was shorter 75.45% of all checks. Note that there are instances where the Hilbert path is equal to the Z-order path, but these were too few to survive two-decimal rounding. Looking at the Levenshtein distances, the results show the average edit distance is between 0 and 0.58 in favor of Hilbert-curve. This implies that the hash strings created by both curves have quite similar distances between points, with no curve being vastly superior in this regard. The benchmark results, in Table 5.2, show significant differences between running the algorithm with or without the Hilbert-curve functions. This difference is clearest when looking at the operations per nanosecond measure, where three times as many operations per second are executed when using the Hilbert-curve. The memory usage is the same with or without using the Hilbert-curve algorithms and does not appear to be a distinguishing factor.

Data set	Iterations	Points	Curve: $Hilbert < ZOrder$	Curve: $Hilbert > ZOrder$	Levenshtein: $ZOrder - Hilbert$
Random	1000	1000	53.15%	46.85%	0.00
Maximum Latitude	1000	1000	58.14%	41.86%	0.55
Maximum Longitude	1000	1000	75.45%	24.55%	0.58

Table 5.1: Distance measurements test results: Amount of iterations executed, points generated in each iteration, average number of times the curve distance was shorter or larger between the curves types and the average difference in Levenshtein between two curve types.

Benchmark Function	Iterations	$Operation/ns$	$Bytes/op$	$Allocations/op$
BenchmarkHash	3000k	1563	16	1
BenchmarkHash, Z-order only	3000k	490	16	1
BenchmarkPoint	5000k	402	64	2
BenchmarkPoint, Z-order only	5000k	353	64	2

Table 5.2: Benchmarking results showing: Amount of iterations executed, operations per nanosecond, allocated bytes per operation, and memory allocations per operation.

5.6 Evaluation

Evaluating the testing done in this research, it is clear that some aspects of it could be improved. As mentioned earlier, the motivation of the testing is to check if the Hilbert-curve produces better hash value-similarities between points than the Z-order curve. While this aspect can be considered as a relevant testing criterion, it could also be argued that this is quite a synthetical measurement designed more to test the theory, than the practical system capabilities. However, these tests can still be considered useful for illustrating the clustering performance of the space-filling curve, regardless of if this implementation is capable of fully taking advantage of this. The more practically relevant test of string-edit distance comparisons, also has some weaknesses. This is mainly due to the fact that the edit distance does not take into account the value-difference of substitutions i.e. the edit distance between 'aaa' and 'aab' is the same as between 'aaa' and 'aaz'. A more reasonable measurement would be using a mixture of both hash and edit distance, creating a distance measurement that takes into account both the value of substitutions, with bit-significance, and the number of edits. Another relevant aspect of these tests is the usage randomly generated test data. There could be made an argument that real-world data could produce different results, having a few areas of highly clustered points with vast distances between them. While this type of data would produce results closer to the practical usage of the system, the main goal of this testing phase is to test the theoretical properties of the Hilbert curve. For this reason, the randomly generated data is deemed sufficient.

Chapter 6

Summary and Conclusion

This chapter presents a short summary and the conclusion to the work done within this research.

6.1 Summary

This research has mainly consisted of four sections, representing four phases: A research phase, a design phase, an implementation phase, and finally a testing phase. The goal of the research phase was to provide a general introduction to spatial data by describing its relation to ordinary multi-attribute data, and the ways they differ from each other. This is closely related to the following section of why naive approaches of managing spatial data, become quite unfit when queried for. Thus, more efficient ways of processing this type of data are described, with various tree structures and ordering techniques. The final section of this phase introduces the ordering structure of space-filling curves and the way they map from multiple dimensions, down to one. The Z-order and Hilbert curve-types are presented and compared, finding the locality preservation abilities of the Hilbert curve to be superior. This lays the basis of the following design phase, motivating the creation of a system that takes advantage of this Hilbert curve-property. The design phase introduces a set of various, currently considered, trendsetting technologies that are to be used for developing this system. These technologies range from using a modern programming language to various tools, frameworks, and cloud-based deployment services. In the following implementation phase, all these technologies are used together, to create a Hilbert curve based hashing system for point data. This system has a server-client structure, with two relatively independent applications as server and client. The internals of these applications is modular components, creating complexity abstractions similar to an object-oriented pattern. The last practical phase of this research was testing. The objective of this phase was to confirm the theoretical clustering advantages of the Hilbert curve over the Z-order curve, as well as explore the processing complexity costs both of these curves. Various measurement techniques were introduced to emphasize the theoretically expected system properties, and the results of these measurements were presented subsequently.

6.2 Conclusion

As outlined in Section 1.2, the research questions in this study relate to two aspects of the Hilbert space-filling curve: Are the theoretical properties of the curve practically applicable within the context of point data hashing (RQ1) and is the computational complexity of this curve, relative to the Z-order curve, worth the theoretical advantages. (RQ2). To answer the first question, the Hilbert-geohash system presented in the Design and Implementation chapters, was constructed based on the work done by Lawder et al. [25]. While the implementation of this system exemplifies the practical applicability of the Hilbert-curve for point data hashing, the question of whether the Hilbert-specific properties actually provide any advantages, is answered in the succeeding Evaluation chapter. The assumed practical benefit of the Hilbert-curve's theoretically superior clustering properties, was improved hash-prefix similarities between points. Using two types of measurements, together with three types of randomly generated test data, the test where executed and the results obtained. The results showed that these superior clustering properties were to some extent present, beating the Z-order curve on average in all but one case. In this one case, the test data was uniformly randomly positioned points, the measurement used was edit distance, and the result yielded that no curve was superior. Thus, the results indicate that while the Hilbert-properties are indeed applicable in this point-hashing context, the extent of their superiority over the Z-order curve is limited. Looking at the benchmarking results, the amount of operation per second almost tripled when using the Hilbert-hashing algorithms as opposed to using only the Z-order curve. The overall conclusion is therefore that while the were some applicable advantages of using the Hilbert-curve, these are arguably not proportional to the complexity increment gained.

6.3 Further work

As concluded earlier, the theoretical clustering superiority of the Hilbert curve is not fully exploited in the context of the point-hashing system presented. In spite of this, there where some findings that inspired the proposal of a related point-retrieval system. This would be a distributed system which uses the Hilbert-geohash system, presented in the previous chapters, as a core part of the distribution mechanism for distributing data among the peers within the system. The global increase of urbanization, mentioned in Section 1.1, can be assumed to generate relatively small areas of highly dense data (e.g the cities, towns etc.) and vast empty spaces with sparse data (e.g oceans, deserts etc.). Therefore, a distributed solution that can scale sections of itself accordingly, adding additional computation-nodes to manage the data of more densely populated areas. This can be done using a PR quadtree, described in Section 2.3.1, as the overall system structure, analogous to the system in Section 2.3.1. For this PR quadtree structure, the Hilbert-curve could be used as the distributed hashing algorithm for the ingested point data, distributing the points in a Hilbert order across the physical quadtree computation-nodes. This system would then be able to dynamically scale, since each node can be split into four sub-nodes, each representing a quadrant of the physical space contained within that node. Each sub-node is assigned a Hilbert curve based-hash, computed using the Hilbert-state of its parent node. Point insertion can, therefore, be done by using the Hilbert-hash of a point, to insert the point into the PR-tree node with the closest prefix, i.e. descending the PR-tree, following the Hilbert-hash until a leaf node is reached.The homogeneity of this structure, i.e. the non-dynamical structure independent of the point insertion order, enables the usage of simple caching methods that don't have complex cache invalidation techniques otherwise needed by dynamic trees. The system could cache the Hilbert-hash of the nodes containing the most popularly requested points, decreasing the amount of descent steps needed down the tree. This will counteract the bottleneck of needing all requests to go trough the single PR-tree root node, avoiding 'hotness'. In addition, the internal structure of each computational node within the distributed system, can have its own optimal in-memory tree structure. This structure can be chosen to fit the type of load this node receives, e.g. a read-optimized k-d tree or troughput-optimized quadtree. The creation of such a system appears to be worth exploring further.

Bibliography

[1] M. Roser, "World population growth." http://ourworldindata.org/data/population-growth-vital-statistics/world-population-growth, 2015. [Online; accessed 23-Nov-2015].

[2] World Health Organization, "Urban population growth." http://www.who.int/gho/urban_health/situation_trends/urban_population_growth/en/, 2015. [Online; accessed 23-Nov-2015].

[3] Texas Instruments, "The evolution of mobile technology series - webinar." http://www.ti.com/pdfs/wtbu/smp_webinar.pdf, 2009. [Online; accessed 23-Nov-2015].

[4] Qualcomm, "Qualcomm's 5g vision." https://www.qualcomm.com/documents/qualcomm-5g-vision-presentation, 2015. [Online; accessed 23-Nov-2015].

[5] "Wikipedia - Short Message Service." https://en.wikipedia.org/wiki/Short_Message_Service. [Online; accessed 29-May-2016].

[6] "Wikipedia - Twitter." https://en.wikipedia.org/wiki/Twitter. [Online; accessed 29-May-2016].

[7] "Wikipedia - Bitly." https://en.wikipedia.org/wiki/Bitly. [Online; accessed 29-May-2016].

[8] "Google url shortener." https://goo.gl/. [Online; accessed 29-May-2016].

[9] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 13, no. 1, pp. 124–141, 2001.

[10] H. Samet, *Foundations of multidimensional and metric data structures*. Amsterdam Boston: Elsevier/Morgan Kaufmann, 2006.

[11] E. Tanin, A. Harwood, and H. Samet, "Using a distributed quadtree index in peer-to-peer networks," *The VLDB Journal*, vol. 16, no. 2, pp. 165–178, 2007.

[12] M. Aly, M. Munich, and P. Perona, "Distributed kd-trees for retrieval from very large image collections," in *Proceedings of the British Machine Vision Conference (BMVC)*, 2011.

[13] Geohash.org, "geohash.org/h000000." http://geohash.org/h000000. [Online; accessed 26-Nov-2015].

[14] Geohash.org, "geohash.org/gzzzzzz." http://geohash.org/gzzzzzz. [Online; accessed 26-Nov-2015].

[15] Wikipedia, "Geohash — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Geohash, 2015. [Online; accessed 04-Dec-2015].

[16] GitHub, "Github." https://github.com/. [Online; accessed 04-Dec-2015].

[17] Git, "Git." https://git-scm.com/. [Online; accessed 04-Dec-2015].

[18] Google, "The Go Programming Language." https://golang.org/. [Online; accessed 04-Dec-2015].

[19] Google, "The cover story." https://blog.golang.org/cover. [Online; accessed 09-Dec-2015].

[20] "GitHub - Gin Web Framework." https://github.com/gin-gonic/gin. [Online; accessed 29-May-2016].

[21] "Node Package Manager." https://www.npmjs.com. [Online; accessed 29-May-2016].

[22] "Module Counts." http://www.modulecounts.com/. [Online; accessed 29-May-2016].

[23] "GitHub - SuperAgent." https://github.com/visionmedia/superagent. [Online; accessed 29-May-2016].

[24] "Wikipedia - Heroku." https://en.wikipedia.org/wiki/Heroku. [Online; accessed 29-May-2016].

[25] J. K. Lawder and P. J. H. King, "Querying multi-dimensional data indexed using the hilbert space-filling curve," *ACM Sigmod Record*, vol. 30, no. 1, pp. 19–24, 2001.

[26] I. E. T. F. (IETF), "RFC 6335: Service Name and Port Number Procedures." https://tools.ietf.org/html/rfc6335#section-6. [Online; accessed 16-May-2016].

[27] Microsoft, "Regular Expression Language - Quick Reference." https://msdn.microsoft.com/en-us/library/az24scfc(v=vs.110).aspx. [Online; accessed 16-May-2016].

[28] T. W. W. W. C. (W3C), "HTTP/1.1: Method Definitions." https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html. [Online; accessed 19-May-2016].

[29] Google, "The Go Programming Language Specification - The Go Programming Language - Operators." https://golang.org/ref/spec#Operators. [Online; accessed 19-May-2016].

[30] ECMA International, "Standard ECMA-262, ECMAScript 2015 Language Specification." http://www.ecma-international.org/ecma-262/6.0/index.html. [Online; accessed 16-May-2016].

[31] "ECMAScript 6 compatibility table." https://kangax.github.io/compat-table/es6/. [Online; accessed 16-May-2016].

[32] "Babel · The compiler for writing next generation JavaScript." https://babeljs.io/. [Online; accessed 12-Dec-2015].

[33] "Levenshtein distance - Rosetta Code." http://rosettacode.org/wiki/Levenshtein_distance#Go. [Online; accessed 19-May-2016].