JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

# Umsetzung von Locality Sensitive Hashing in Java

BACHELORARBEIT
(PROJEKTPRAKTIKUM)

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

INFORMATIK

Eingereicht von:
Reinhold Taucher, 0455386

Angefertigt am:
Institut für Computational Perception

Betreuung:
Univ.-Prof. Dr. Gerhard Widmer
Dipl.-Ing. Klaus Seyerlehner

Linz, November 2009

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Einführung

Das Internet hat uns eine Fülle von Daten gebracht, die alle nur einen Mausklick entfernt sind [2, 3]. Mit Leichtigkeit ist es möglich, tausende von Liedern oder mehrere tausend Bilder oder sogar mehrere tausend Filme immer bei sich zu tragen. Doch trotz der mittlerweile sehr fortgeschrittenen Computertechnologie ist es nicht möglich, mit der vorhandenen Prozessorleistung all die vorhanden Daten effizient zu durchsuchen oder auf Ähnlichkeit zu überprüfen.
Diese Problematik zieht weite Kreise, denn sie ist auch von größter Bedeutung für eine Vielzahl an Programmen. Um nur einige Beispiele zu nennen: Data Compression, Database- und Datamining, Information Retrieval, Machine Learning, Pattern Recognition, Statistik und Datenanalyse und die bereits erwähnten Video- und Musikdatenbanken [2].
Das Problem der Suche nach ähnlichen Objekten wird allgemein als ”similarity search problem” [2] bezeichnet und spielt unter anderem eine wesentliche Rolle bei der Suche nach Ähnlichkeiten zwischen Webseiten, Dokumente, Dateien.... Da sehr oft sehr große Datenmengen auf Ähnlichkeiten durchsucht werden sollen, fordern die Anwender von Suchalgorithmen, dass sie in nur kurzer Zeit ein Ergebnis presentiert bekommen.
Diese Arbeit konzentriert sich auf einen speziellen Lösungsansatz für das ”similarity search problem”, in dem jedes Objekt durch einen Punkt in einem $n$-dimensionalen euklidischen Raum dargestellt wird. Als Baseline für die Suche wird die lineare Suche verwendet, bei der jedes Objekt, dargestellt durch einen Punkt, mit jedem anderen Punkt verglichen wird. Dieser Ansatz ist jedoch nicht sehr effizient und kann durchaus eine lange Suchzeit in Anspruch nehmen.
Für den Fall, dass es sich um Punkte von niederer Dimension handelt, existieren effiziente Verfahren wie Quicksort, KD-Tree (siehe Kapitel 1.2.1) bzw. des-

sen Abwandlungen oder auch Grid Files (siehe Kapitel 1.2.2). Handelt es sich bei den Objekten jedoch um Textdokumente oder Musikdateien, so werden diese oft durch sehr hochdimensionale Punkte beschrieben. Dadurch verlieren die herkömmlichen Verfahren, die bei niedrigen Dimensionen funktionieren, ihren Wirkungsgrad und sind in hochdimensionalen Räumen oft schlechter als die einfache lineare Suche.
Es gibt bereits einige effiziente Algorithmen für den Fall, dass es sich bei der Dimension *d* um die zweite oder die dritte handelt [4], jedoch gibt es noch keinen effizienten Ansatz für höhere Dimensionen. Somit ergibt sich ein grundsätzliches Problem, mit den großen Dimensionen zurecht zu kommen, der sogenannte 'curse of dimensionality' [2].
Trotz jahrzehntelanger Forschung sind die momentanen Lösungen noch nicht befriedigend [2]. Entweder sie benötigen zu viel Speicher oder die Suchzeit steigt exponentiell mit *d* an [4]. Tatsächlich, im Falle dass die Dimension groß genug ist, bieten bestehende Algorithmen weder in der Theorie noch in der Praxis eine Verbesserung gegenüber der einfachen linearen Suche. Dies geht sogar so weit, dass bei hochdimensionalen Räumen der Organisationsaufwand bei Verfahren, die auf Raumteilung basieren, so stark ansteigt, dass sie sogar oft ineffizienter als die einfache lineare Suche sind.
Um dennoch in hochdimensionalen Räumen effiziente Nachbarschaftssuche betreiben zu können, wurde vorgeschlagen [2, 4], die Definition von Nachbarschaft etwas aufzuweichen. Anstelle der tatsächlich nächsten Nachbarn versucht man nur mehr Punkte zu finden, die mit hoher Wahrscheinlichkeit die nächsten Nachbarn sind. Dadurch können weitaus effizientere Algorithmen verwendet werden, mit dem kleinen Nachteil, dass die Nachbarschaftssuche nicht garantiert die allernächsten Punkte zum Suchpunkt liefert, aber immerhin mit einer sehr hohen Wahrscheinlichkeit. Eines dieser Verfahren zum Auffinden der annähernd nächsten Nachbarn ist Locality-Sensitive Hashing (LSH).
Ziel dieser Arbeit war es, den $E^2$LSH Algorithmus, der von Alexandr Andoni und Piotr Indyk entworfen und in der Programmiersprache *C* umgesetzt wurde, auf *Java* zu portieren.

## 1.1 Problemstellung

Die Aufgabenstellung bei ”Similarity Search” [17] ist es, ausgehend von einem Suchpunkt *x* und einem Abstandmaß, alle Punkte ausfindig zu machen, die eine bestimmte Bedingung erfüllen. Dabei gibt es unterschiedliche Möglichkeiten für die Suche. Zum einen die der ”Range Queries” [17] und die der ”Nearest Neighbor Queries” [17].

### 1.1.1 Range Queries

Die Range Queries zeichnen sich dadurch aus, dass die Suche durch ein Suchobjekt $x \in \mathcal{P}$ und einen Radius *R* als Begrenzung gegeben ist. Die Suche würde in diesem Fall alle Objekte finden, die innerhalb einer bestimmten Distanz *R* von *x* liegen. $Rq(x, R) = \{y \in X, d(x, y) < R\}$ Für den Fall, dass der Radius für die Query Rq(x,0) ist, so handelt es sich um eine Punktsuche oder exakte Suche, das heißt, dass eine identische Kopie des Suchpunktes gesucht wird. Dieses Vorgehen kommt oft bei Löschfunktionen zum Einsatz.

#### R-near neighbor

Das Problem der R-near neighbor (R-NN) ist ein klassischer Vertreter der Range Queries. Dabei ist eine Menge an Punkten $\mathcal{P} \subset \mathbf{R}^d$ gegeben. Für einen Suchpunkt *x* sollen alle Punkte $y \in \mathcal{P}$ gefunden werden, für die $\parallel x - y \parallel_2 \leq R$ ist, siehe Abbildung 1.1, wobei $\parallel x - y \parallel_2$ die euklidische Distanz zwischen *x* und *y* ist und *R* der Radius ist innerhalb dessen die Ergebnisse geliefert werden sollen.

#### c-approximate near neighbor

Dabei handelt es sich um die R-NN Problematik mit Annäherung, durch welche die Suche nach einem ”near neighbor” beschleunigt werden soll. Dazu wird

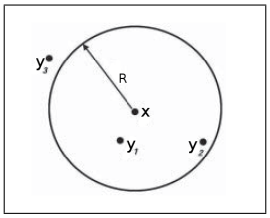


Abbildung 1.1: Schematische Darstellung der ”R-NN” Problematik [10]

ein Annäherungsfaktor *c* eingesetzt. Die angeforderte Datenstruktur soll einen Punkt zurückgeben, dessen Distanz vom Suchpunkt maximal *c* mal der Radius *R* ist. Der Grund für dieses Vorgehen ist, dass in vielen Fällen der angenäherte Nachbar genauso so gut ist wie der tatsächliche. Wenn ein c-approximate near neighbor (cR-NN), ausgehend von einem Suchpunktes *x*, existiert, siehe Abbildung 1.2, soll dieser mit der Wahrscheinlichkeit $1 - \delta$ gefunden werden.

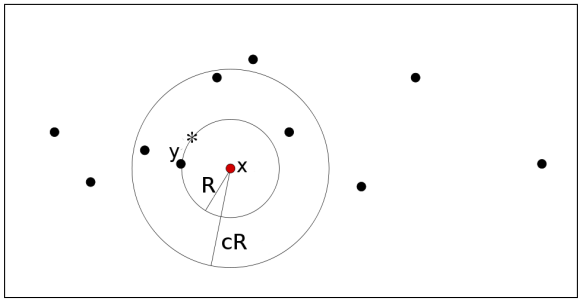


Abbildung 1.2: Schematische Darstellung der ”cR-NN” Problematik [2]

### 1.1.2 Nearest Neighbor Queries

Immer, wenn ein ähnliches Objekt mit Hilfe einer Range Query gesucht wird, ist es zwingend notwendig, eine maximale Distanz anzugeben. Jedoch, wenn die Datenmenge und die verwendete Distanzfunktion im Vorfeld nicht bekannt sind, kann es bei der Angabe eines Radius zu Schwierigkeiten kommen. Wird ein zu kleiner Radius gewählt, kann es vorkommen, dass kein Resultat erzielt wird und eine erneute Suche mit größerem Radius nötig ist. Wird der Radius zu groß

gewählt, kann die Berechnung lange Zeit in Anspruch nehmen und die Ergebnismenge könnte aufgrund der hohen Anzahl an Objekten nicht aussagekräftig sein.

### Nearest Neighbor (NN)

Die Basisversion dieses Suchansatzes findet den nächsten Punkt zu einem gegebenen Suchpunkt, welches folglich der nächste Nachbar des Suchpunktes sein soll. Auch hier kann wieder mit Annäherung gearbeitet werden, durch eine Auflistung aller ”approximate nearest neighbors” und die anschließende Auswahl des dem Suchpunkt am nächsten liegenden Objekts aus dieser Liste.

### k-Nearest Neighbors

Das Konzept der ”nearest neighbor search” (NNS) kann auch allgemeiner betrachtet werden und führt zur k-nearest neighbor search (k-NNS). Dabei werden die *k* nächsten Nachbarn ausgehend von einem Suchpunkt *x* gesucht. Wenn die ganze Sammlung an Objekten weniger als *k* Objekte enthält, gibt die Suche alle wieder.
Besitzen mehrere Objekte denselben Abstand wird beliebig ein Objekt ausgewählt. In Abbildung 1.3 werden die drei nächsten Nachbarn ausgehend vom Suchpunkt *x* gesucht. Als Ergebnis werden $y_1, y_4$ und $y_5$ geliefert. Das Objekt $y_3$ hat zwar den selben Abstand wie $y_1$ wird jedoch nicht mehr berücksichtigt, da nur drei Ergebnisse gefordert sind.

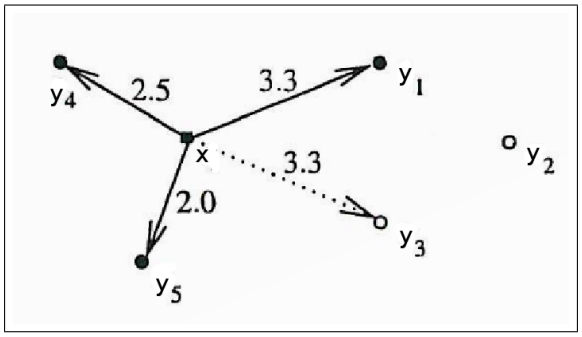


Abbildung 1.3: Schematische Darstellung der ”k-NN” Problematik [17]

## 1.2 Lösungsansätze in niedrig dimensionalen Räumen

### 1.2.1 kd-tree

Einer der bekanntesten Ansätze zur Verwaltung mehrdimensionaler Daten ist die Verwendung des kd-tree [8]. Hierbei handelt es sich um einen binären Suchbaum, der eine rekursive Unterteilung des Universums mithilfe von $(d-1)$ dimensionalen Hyperebenen repräsentiert. Bei diesen Hyperebenen handelt es sich um ”iso-orientierte” Ebenen, deren Richtungen sich zwischen den *d* möglichen Dimensionen unterscheiden. So ergibt sich z.B. für die Dimension 3 eine abwechselnde Aufteilung des Raumes durch Hyperebenen normal zu den x,y und z-Achsen.
Diese Hyperebenen werden zur Repräsentation im Baum herangezogen (siehe Abbildung 1.4) und müssen je mindestens einen Punkt beinhalten. Jeder innere Knoten des Baumes besitzt einen oder zwei Kinderknoten. Diese dienen bei der Suche nach entsprechenden Nachbarn als ”Diskriminatoren”[8]. Dabei leiten sie die Suche in die richtigen Bahnen. Ein Nachteil des kd-Baumes ist, dass die Datenstruktur sensibel, bezogen auf die Reihenfolge in der die Punkte angefügt werden, ist. Ein weiterer Nachteil ist, dass die Punkte auf den ganzen Baum verteilt sind. Mit dem ”adaptive kd-tree” [8] werden diese Probleme behoben, da die Aufteilung so vorgenommen wird, dass sich auf beiden Teilen des

Baumes annähernd die gleiche Anzahl an Knoten befinden. Die Hyperebenen verlaufen nach wie vor parallel zu den Achsen, jedoch ist es beim ”adaptive kd-tree” nicht mehr unbedingt nötig, dass sie einen Punkt enthalten und ihre Richtungen müssen nicht mehr zwingend abwechselnd sein.
Dadurch sind die Punkte, an denen geteilt wird, nicht mehr Teil der Eingabedaten. Die Datenpunkte werden alle in den Blättern des Baumes gespeichert. Darüber liegende Knoten beinhalten die Dimension und die Koordinate des jeweiligen Teilpunktes. Die Aufteilung wird rekursiv vorgenommen bis jeder Unterraum nur noch eine bestimme Anzahl an Punkten enthält. Der ”adaptive kd-tree” ist eine relativ statische Struktur, was die Balance beim Einfügen und Löschen teilweise beeinträchtigen kann. Vor allem beim Erstellen des Baumes ist die Reihenfolge, in der die Punkte eingefügt werden, von großer Bedeutung für die Balance des Baumes. Die Datenstruktur funktioniert am Besten, wenn das Universum, also die Datenmenge, schon zu Beginn feststeht und Einfügeoperationen selten sind.
Das Einfügen in einen kd-tree mit *n* Einträgen benötigt eine Zeit von $O(log(n))$, das Löschen des Wurzelknotens $O(n^{(k-1)/k})$, wobei das Löschen eines gewöhnlichen Knotens lediglich $O(log(n))$ benötigt. Die durchschnittliche Laufzeit für eine NNS ist $O(log(n))$ [6], was jedoch leider nur für eine niedrige Dimensionalität des Suchraumes gilt. Für Punkte in hochdimensionalen Räumen degeneriert die Suche zu einer linearen Suche.
Eine andere Variante des kd-Baumes ist die ”bintree” Version [8]. Bei diesem Ansatz wird das Universum, also die zu durchsuchende Menge an Objekten,

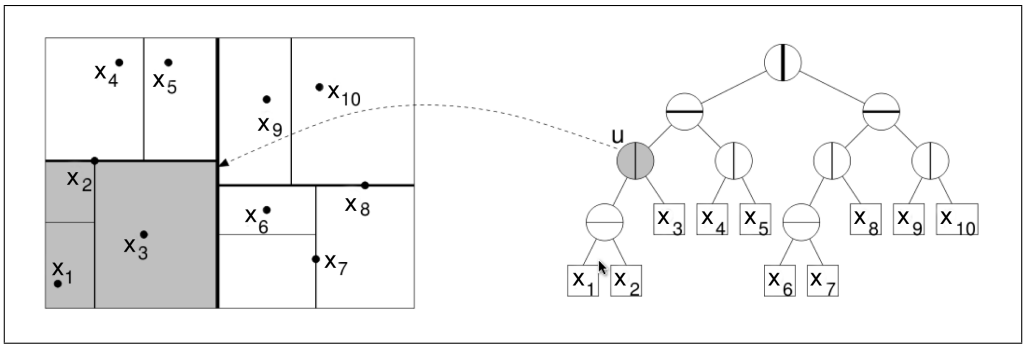


Abbildung 1.4: Schematische Darstellung eines kd-tree [10]

rekursiv in $d$-dimensionale Boxen gleicher Größe unterteilt. Dies wird solange durchgeführt, bis jede Box nur noch eine bestimmte Anzahl an Punkten des Universums enthält. Die Methodik ist zwar weniger anpassungsfähig, jedoch hat sie einige Vorteile wie z.B. impliziertes Wissen über die Aufteilung der Hyperebenen.

### 1.2.2 Grid File

Das Grid File [8, 16] ist eine typische Struktur für hashbasierte Zugriffsmethoden. Das Prinzip dieser Struktur ist, die Zugriffszeit, die hauptsächlich von den Plattenzugriffen abhängig ist, so kurz wie möglich zu halten. Daher sollen bei einer Suchanfrage nicht mehr als zwei Plattenzugriffe durchgeführt werden. Ein weiteres Ziel der Datenstruktur ist, dass Datensätze, deren Attribute sich ähneln, auch nahe im physischen Speicher verweilen. Jeder Datensatz ist ein Punkt in einem mehrdimensionalen Raum, wobei jede Dimension ein Attribut des Datensatzes repräsentiert. Dieser mehrdimensionale Raum wird durch ein orthogonales Gitter aufgeteilt. Durch diese Aufteilung entstehen sogenannte Gridblöcke.
Die eigentlichen Daten, die mit einem Gridblock assoziiert sind, werden auf der Platte in ein sogenanntes Bucket gespeichert. Ein Bucket kann zwischen 10 und 1000 Datensätze aufnehmen und ist intern linear organisiert. Jeder Gridblock verweist nur auf ein Bucket, jedoch kann ein Bucket die Daten mehrerer Gridblöcke beinhalten.
Zur Verwaltung von Buckets existieren Gridverzeichnisse (sog. ”Grid Directories”), welche die Assoziationen beinhalten. Das Grid Directory besteht aus einem $k$-dimensionalen Array, welches die Zeiger auf die Buckets beinhaltet, und $k$ eindimensionalen Arrays, sogenannten Skalierungsvektoren, von denen jeder Vektor eine Koordinatenachse des Datenraumes beschreibt.
Verweist mehr als ein Grid Directory auf ein Bucket, so werden diese Grid Directories zu einer ”Region” zusammengefasst. Eine Verschmelzung von Grid Directories ist nur möglich, wenn diese ebenfalls wieder die Form eines Rechtecks aufweisen.

Um nun auf Basis dieses Konstruktes eine Suche zu tätigen, muss zuerst mit Hilfe der Skalierungsvektoren die Zelle gefunden werden, welche den gesuchten Punkt enthält. Sollte sich das gewünschte Grid Directory nicht im Speicher befinden, so ist ein Plattenzugriff nötig. Das geladene Grid Directory beinhaltet eine Referenz zu jenem Bucket, in dem die möglichen Daten gefunden werden können. Um dieses Bucket jedoch zu erhalten, ist abermals ein Plattenzugriff nötig. Alles in allem sind jedoch nicht mehr als zwei Plattenzugriffe nötig, um eine Suchanfrage zu beantworten. Für eine Suchanfrage mit Auswahl an Ergebnissen müssen alle Zellen untersucht werden, welche die Suchregion überlappen. Nachdem die Duplikate ausgemustert wurden, werden die entsprechenden Buckets in den Speicher geladen, um diese genauer zu betrachten.
Damit ein Punkt eingefügt werden kann, muss vorab mittels einer Suchanfrage die Zelle und das Bucket festgestellt werden, in welches der Punkt eingefügt werden soll. Ist in dem entsprechenden Bucket genügend Platz vorhanden, so kann der Punkt einfach eingefügt werden. Sind die Kapazitäten des Buckets bereits ausgelastet, so wird dessen Inhalt auf zwei Buckets aufgeteilt, und somit wird auch dessen Region und folglich der ganze Datenraum geteilt. Der einzufügenden Punkt wiederum wird in das entsprechende Bucket gespeichert, in dem nun Platz vorhanden ist.
Beim Löschen eines Punktes wird ebenfalls entsprechend der verbleibenden Anzahl an Daten im Bucket verfahren. Wenn die sich im Bucket befindlichen Daten einen Schwellenwert unterschreiten, wird dieses gelöscht und dessen Region mit einer anderen verschmolzen. Andernfalls wird der Punkt ohne Umstrukturierung aus dem Bucket gelöscht.

## 1.3 Locality Sensitive Hashing

### 1.3.1 Einführung

Das *"locality sensitve hashing"* (LSH) [1, 2, 4, 10, 11, 13, 12] wurde zuerst von Indyk und Mowani als Ähnlichkeitssuche in sublinearer Zeit vorgestellt und kann

zum Lösen des R-NN Problems, bzw. des cR-NN Problems eingesetzt werden. Die wesentliche Idee des LSH ist, dass die Punkte mittels mehrerer verschiedener Hashfunktionen gehashed werden. Für jede Funktion ist die Wahrscheinlichkeit, dass zwei Punkte einen ähnlichen Hashwert aufweisen, höher, wenn die Objekte nicht weit voneinander entfernt sind. Im Gegensatz dazu ist die Wahrscheinlichkeit einer Kollision der Funktionen geringer, wenn sich die beiden Objekte weit voneinander entfernt befinden. Dadurch ist es möglich, den oder die entsprechenden Nachbarn zu finden, indem der jeweilige Punkt gehashed wird, und alle Elemente ausgelesen werden, die in demselben Bucket enthalten sind. Ein Bucket ist in diesem Zusammenhang lediglich eine Ansammlung, bzw. eine Art Container von Objekten, die den gleichen Hashwert aufweisen.

### Locality Sensitive Hash Function

Das LSH Schema basiert auf ”locality sensitive” Hashfunktionen. Eine Familie von Hashfunktionen $\mathcal{H}$ dient dazu, die sich in $\Re^d$ befindlichen Punkte auf ein Universum *U* zu projizieren [4] ($\mathcal{H} = \{h : S \rightarrow U\}$), wobei *S* die Domäne der Objekte ist [1]. Für zwei Punkte *x* und *y* wird gleichmäßig eine zufällige Funktion $h_i$ aus $\mathcal{H}$ gewählt, und die Wahrscheinlichkeit, dass $h(x) = h(y)$ analysiert. Jedoch darf sich eine Familie von Funktionen $\mathcal{H}$ nur ”locality sensitive”, bzw. ”(R, cR, $p_1$, $p_2$)-sensitive” nennen, wenn für alle Punkte *x,y* $\in \Re^d$ gilt [4]:

- falls $D(x, y) \leq R$ dann $Pr_H[h(x) = h(y)] \geq p_1$
- falls $D(x, y) \geq cR$ dann $Pr_H[h(x) = h(y)] \leq p_2$

Zusätzlich muss, damit eine ”locality-sensitve hash family” auch nützlich ist, $p_1 > p_2$ erfüllt sein. Die Definition zeigt, dass Punkte, die sich nah sind, innerhalb eines bestimmten Radius *R*, mit einer Wahrscheinlichkeit von $p_1$ kollidieren, wohingegen Punkte, deren Distanz größer als $cR$ ist, mit geringerer Wahrscheinlichkeit $p_2$ kollidieren [12]. Weiters können für diese ”locality-sensitive hash func-

tions" unterschiedliche Distanzfunktionen *D* gewählt werden. Die bekanntesten Vertreter sind die Hamming Norm und p-stable Distributions. In Kapitel 1.3.3 wird noch näher auf unterschiedliche LSH Familien eingegangen.

### LSH Index

Mit Hilfe dieser LSH Funktionen $\mathcal{H}$ ist es möglich, eine indizierende Datenstruktur zu generieren, auf deren Basis Ähnlichkeitssuchen gestartet werden können [13]. Bei dieser indizierenden Datenstruktur, in weiterer Folge nur noch als LSH Index bezeichnet, handelt es sich um eine Menge von Hash-Tabellen.

- Es werden unabhängig und zufällig *L* Funktionen $g_1, g_2, \ldots g_L$ aus der Funktionsfamilie $G = \{g : S \rightarrow U^k\}$ gewählt.
- Jede dieser *L* Funktionen wird zur Konstruktion einer Hashtabelle verwendet. Somit besteht die Datenstruktur aus *L* Hashtabellen.
- Jede dieser *L* Hashfunktionen ist eine Konkatenation von *k* Hashfunktionen $h_1, h_2, \ldots h_k$. Jede dieser *k*-Hashfunktionen $h_i$ wird wiederum unabhängig und zufällig aus einer Hashfamilie $\mathcal{H}$ ausgewählt.
- Jeder Punkt $x \in \mathcal{R}$, wird in $g \in G,\ g(x) = (h_1(x), h_2(x), \ldots h_k(x))$ eingefügt.

So wird ein Punkt mittels der Konkatenation von *k* unterschiedlichen Hashfunktionen in ein sogenanntes Hashbucket von *g* eingefügt, wie in Abbildung 1.5 schematisch dargestellt. Die Größe einer solchen Hashtabelle ist proportional zur Größe der verwendeten Datenmenge, da jeder Table so viele Einträge besitzt, wie es Datenobjekte in der Datenmenge gibt.
Wenn zwei Punkte eine Wahrscheinlichkeit $p_2$ haben, mit nur einer Hashfunktion nicht zu kollidieren, so verringert sich diese Wahrscheinlichkeit durch die Konkatenation von *k* unterschiedlichen Funktionen auf $p_2^k$. Bei einer sehr hohen Anzahl von Hashfunktionen wird damit die Wahrscheinlichkeit, dass zwei Punkte, die einen Abstand größer als *cR* vom Suchpunkt besitzen, kollidieren, vernachlässigbar klein. Daraus resultierend ist leicht erkennbar, dass eine geringe Anzahl an Hashfunktionen die Anzahl der Punkte, die fälschlicherweise

als ähnlich angenommen werden, wachsen lässt. Andererseits sinkt aber auch bei einer hohen Anzahl an Hashfunktionen (großes *k*) die Wahrscheinlichkeit, dass nahe gelegene Punkte kollidieren, da auch $p_1^k$ kleiner wird [13]. Abhilfe für diese Problematik verschafft die Anwendung mehrerer Hashtabellen [12].

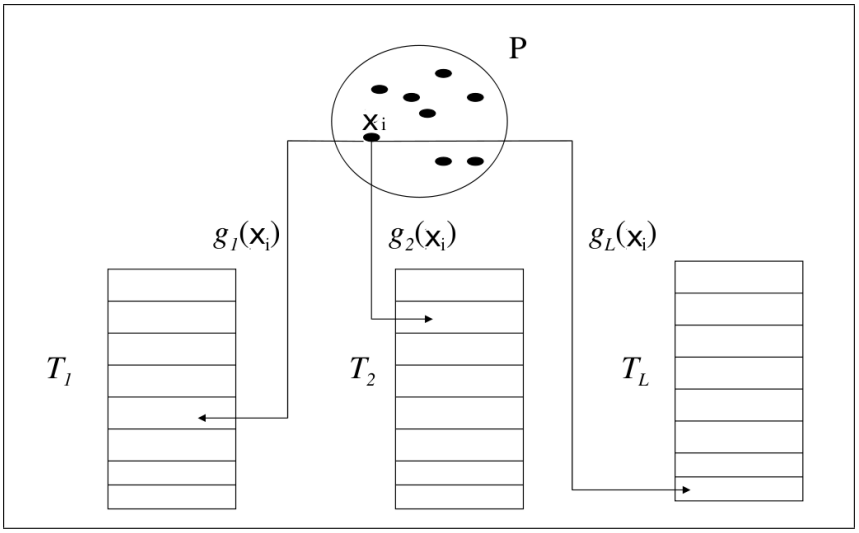


Abbildung 1.5: Schematische Darstellung der LSH Datenstruktur [7]

### 1.3.2 Suche in der LSH Datenstruktur

Um nun von einem Suchpunkt *x* aus eine Ähnlichkeitssuche anzustoßen, muss diese in zwei Schritten abgearbeitet werden. Zuerst wird für jede Hashtabelle $T_1$ bis $T_L$ ermittelt, in welches Bucket der Suchpunkt fallen würde, indem der entsprechende Hashwert $g_i(x)$ berechnet wird. Dann wird eine Kandidatenmenge der möglichen NNs gebildet, indem die Punkte aus den Buckets vereinigt werden. Im zweiten Schritt wird für jeden Punkt *y* in dieser Kandidatenmenge geprüft, ob die Distanz $D(x, y)$ zum Suchpunkt kleiner dem geforderten Suchradius ist. Wenn $D(x, y) \leq R$ ist, wird *y* in die Menge der nächsten Nachbarn aufgenommen, welche das Ergebnis bildet. Bei der Suche selbst kann zwischen zwei Strategien unterschieden werden [4]:

1. Die Suche wird abgebrochen, sowie die ersten *3L* Punkte, mit Duplikaten, gefunden wurden (die Variable *L* wird im Kapitel 1.4.3 noch näher beschrieben).
2. Die Suche wird solange fortgeführt, bis alle Punkte aus allen entsprechenden Buckets ausgelesen wurden.

### 1.3.3 LSH Familien

Hier soll ein kurzer Überblick über bestehende LSH Familien gegeben werden, wobei auf die in weiterer Folge angewendete Familie der p-stable Distribution detaillierter eingegangen wird.

#### LSH mittels Hamming Distance

Um eine binäre Darstellung eines Vektors $\{0,1\}^d$ zu erhalten, wird jeder Punkt *x* mit Hilfe einer Funktion $h_i(x) = x_i$ umgeformt, wobei *i* für die *i*-te Koordinate des Punktes steht [7]. Um dies zu ermöglichen, wird unter den Koordinaten aller Punkte die größte Koordinate *C* ausgewählt. Mit dieser werden alle Punkte aus $\mathcal{P}$ in den Hamming Würfel $H^{d'}$ mit $d' = Cd$ umgeformt, indem jeder Punkt $x = (x_1, ....x_d)$ als Binärvektor dargestellt wird: $bin(x) = Unary_C(x_1) \ldots Unary_C(x_d)$. Dabei entspricht $Unary_C(x_i)$ der monadischen Repräsentation von *x*, und wird dargestellt durch eine Folge von $x_i$ Einsen (siehe Abbildung 1.6), gefolgt von $C - x_i$ Nullen. Somit kann die Distanz für die Punkte *x* und *y* wie folgt definiert werden [7]:

$$D(x, y) = D_H(bin(x), bin(y))$$

Durch dieses Einbetten in den Hamming Raum wird die Distanz zwischen den Punkten bewahrt. Daher kann in weiterer Folge auch die R-NN Problematik mit dem Hamming Raum $H^{d'}$ gelöst werden.
Die Hashfunktion wird wie folgt definiert. Es werden *l* Projektionen bestimmt,

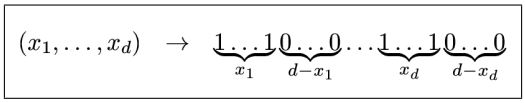


Abbildung 1.6: Schematische Darstellung der Unary Funktion [18]

die je nur eine Teilmenge $I_1, \ldots, I_l$ aller Koordinaten eines Vektors verwenden. Mit $x_{|I}$ wird die Projektion des Punktes *x* auf das Koordinatenset *I* bezeichnet. Dabei wird $x_{|I}$ nur auf Basis der gewählten Koordinate nach $I_i$ bestimmt und verknüpft die Bits in diesen Positionen.
Für die Vorverarbeitung wird jeder Punkt $x \in \mathcal{P}$ in ein sogenanntes Bucket $g_j(x)$ gespeichert, wobei für dieses Bucket $g_j(x) = x_{|I_j}$, für $j = 1, \ldots, l$ gilt.
Zur Realisierung der Buckets werden Standardhashtabellen verwendet, demnach werden zwei Levels von Hashfunktionen [7] angewendet. Zuerst die LSH Funktion, die einen Punkt *x* in ein Bucket $g_j(x)$ mapped und dann eine Standardhashfunktion, die den Inhalt der Buckets in eine Hashtabelle einer gewissen Größe mapped. Wird die maximale Größe eines Buckets der letzteren Hashtabelle überschritten, so wird mittels 'chaining' die Anzahl um die bereits vorhanden Menge erweitert .

### LSH mittels Min Hash

Mittels *Jaccard Koeffizient* [4, 19, 20], ist es möglich, die Ähnlichkeit zwischen zwei Mengen ($A, B \subset U$) zu bestimmen. Der Koeffizient ist definiert als $sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$ und ist im Gegensatz zur bereits erwähnten Hamming Distanz ein Ähnlichkeitsmaß. Je höher der Jaccard Koeffizient, desto ähnlicher sind die entsprechenden Mengen. Das entsprechende Distanzmaß ist durch $D(A, B) = 1 - sim(A, B)$ definiert, wodurch eine LSH Familie, min-hash genannt, definiert werden kann. Dazu wird eine zufällige Anordnung der Grundwahrheit *U* angenommen. Auf Basis dieser zufälligen Anordnung kann eine Hashfunktion $h(A)$ definiert werden, die eine Wahrscheinlichkeit einer Kollisionen, also die Wahrscheinlichkeit, dass Punkte in das selbe Bucket gehashed werden, mit $P\lfloor h(A) = h(B) \rfloor = sim(A, B)$ definiert.

### LSH mittels Arccos

Die Cosinus Ähnlichkeit [9] ist definiert als der Cosinus des Winkels zwischen zwei Vektoren $\vec{u}$ und $\vec{v}$ in einem *d*-dimensionalen Raum. Die Cosinus-Ähnlichkeit wird nach folgender Formel berechnet: $cos(\Theta(\vec{u},\vec{v})) = \frac{|\vec{u}.\vec{v}|}{|\vec{u}|.|\vec{v}|}$ [9].
Hier ist $\Theta(\vec{u},\vec{v})$ der Winkel zwischen $\vec{u}$ und $\vec{v}$ im Radiantenmaß. $|\vec{u}.\vec{v}|$ entspricht dem Skalarprodukt der Vektoren $\vec{v}$ und $\vec{u}$. Die Länge der Vektoren wird repräsentiert durch $|\vec{u}|$ und $|\vec{v}|$.
Bei der LSH Funktion für Cosinus Ähnlichkeit wird für eine Sammlung an Vektoren der Dimension *k* eine Hashfunktion wie folgt definiert. Zuerst wird ein zufällig Einheitsvektor $\vec{e}$ aus diesem *d*-dimensionalen Raum erstellt, mithilfe dessen die Hashfunktion $h_e$ wie folgt definiert wird [9]:

$$h_e(\vec{u}) = \begin{cases} 1 & : \vec{e}.\vec{u} \geq 0 \\ 0 & : \vec{e}.\vec{u} < 0 \end{cases}$$

Die Wahrscheinlichkeit, dass eine zufällige Hyperebene zwei Vektoren teilt, ist direkt proportional zum Winkel zwischen diesen beiden Vektoren.
Eine alternative Methode um die Cosinus Ähnlichkeiten zu finden, veranschaulicht die Gleichung $cos(\Theta(\vec{u},\vec{v})) = cos(1 - Pr\left[h_e(\vec{u}) = h_e(\vec{v})\right]\ \pi)$ [9]. Somit ist es möglich, eine große Anzahl *l* an zufälligen Vektoren zu erzeugen, um $P_e[h_e(\vec{u}) = h_e(\vec{v})]$ zu approximieren und in die Gleichung einzusetzen, um die Cosinusdistanz zwischen zwei Vektoren zu erhalten. Je mehr zufällige Vektoren erstellt werden, desto genauer kann die Cosiunsähnlichkeit zwischen zwei Vektoren berechnet werden. In der Praxis ist die Anzahl der Zufallsvektoren *l* in sehr hohem Maße von der Domäne abhängig, d.h. von der Gesamtzahl der Vektoren, der Dimensionalität und der Art und Weise, wie die Vektoren verteilt sind.
Werden *l* zufällige Vektoren verwendet, kann jeder Vektor mittels der definierten Hashfunktion durch einen Bitstream der Länge *l* dargestellt werden. Bei genauerer Betrachtung der zuletzt erwähnten Formel ist ersichtlich, dass $P_e\left[h_e(\vec{u}) = h_e(\vec{u})\right] = 1 - (hammingdistance)/l$. Daher kann das Problem, den Cosinusabstand zwischen zwei Vektoren zu finden, umgemünzt werden auf das

Problem, den Hammingabstand zwischen den Bitstreams zu finden. Die Berechnung des Hammingabstandes zwischen zwei Bitstreams ist wesentlich schneller und weniger speicherintensiv. Weiters wird dabei die Dimensionalität verringert, wobei die Cosinusdistanz zwischen den Vektoren erhalten bleibt.

## 1.4 LSH mittels p-stable Distribution

In diesem Kapitel wird die in der Implementierung angewandte LSH Familie, die der p-stable Distribution, vorgestellt. Diese Familie zeichnet sich dadurch aus, dass die Distanz mittels der $l_p$ Norm für ein $p \in (0, 2]$ ausgedrückt wird. Die Hashfunktionen sind besonders einfach, für den Fall dass $p = 2$, und bieten eine effiziente Lösung des "approximate nearest neighbor" Problems [14].

### 1.4.1 Mathematischer Exkurs

In diesem Unterkapitel soll ein kurzer Überblick über die mathematischen Hintergründe, die bei LSH mittels p-stable Distribution zum Einsatz kommen, gegeben werden.

#### p-stable Distribution

Sogenannte "Stable Distributions" [14] sind definiert als normalisierte Summe von unabhängigen, identisch verteilten Variablen. Die am häufigsten auftretende und damit auch bekannteste Verteilung ist die Gauss'sche oder Normalverteilung. Die genaue mathematische Beschreibung einer p-stabilen Verteilung lautet wie folgt:
Eine Verteilung *D* über $\Re$ wird p-stable genannt, falls ein $p \geq 0$ existiert, sodass für beliebige reellen Zahlen $w_1, w_2 \ldots w_n$ und unabhängige Zufallsvariablen $X_1, X_2 \ldots X_n$ die alle die gleiche Verteilung *D* besitzen, dann auch die

Zufallsvariable $\sum_i w_i X_i$ die gleiche Verteilung besitzt wie die Zufallsvariable $(\sum_i \mid w_i \mid^p)^{\frac{1}{p}} X$), wobei *X* eine zufällige Variable mit der Verteilung *D* ist. Für alle $p \in (0, 2]$ existieren p-stabile Verteilungen. Von besonderer Bedeutung sind:

1. Die *Cauchy Verteilung* $D_C$ wird definiert durch $D_C(x) = \frac{1}{\pi}\frac{1}{1+x^2}$ und ist 1-stable.
2. Die bereits erwähnte *Gauss'sche (Normal)-Verteilung* $D_G$ wird definiert durch $D_G(x) = \frac{1}{\sqrt{2\pi}} c^{\frac{-x^2}{2}}$ und ist 2-stable.

Praktisch gesehen können p-stabile Zufallsvariablen von zwei unabhängigen gleichmäßig über $[0, 1]$ verteilten Variablen generiert werden, obwohl geschlossene Dichte- und Verteilungsfunktionen fehlen [14, 1].

### Norm

In weiterer Folge wird die $l_p^d$ Norm [1, 15] als Bezeichnung für den *d*-dimensionalen reellen Raum $\Re^d$ unter Verwendung der $l_p$ Norm eingesetzt. Die $l_p$ Norm eines Vektors $\vec{v} \in \Re^d$ wird kurz als $\parallel \vec{v} \parallel_p$ geschrieben. Eine Norm ist eine Funktion $\Re^d \rightarrow \Re$, wobei zwischen folgenden Vektornormen unterschieden werden kann:

1. $L_1$ Norm: $\parallel \vec{v} \parallel_1 = \sum_{i=1}^{d} \mid X_i \mid$
2. $L_2$ Norm oder auch Euklidische Norm $\parallel \vec{v} \parallel_2 = \sqrt{\sum_{i=1}^{d} \vec{v}_i^2}$
3. Maximum Norm $\parallel \vec{v} \parallel_\infty = max_{1 \leq i \leq n} \mid \vec{v}_i \mid$
4. $L_p$ Norm $\parallel \vec{v} \parallel_p = \sum_{i=1}^{d} \vec{v}_i^p)^{\frac{1}{p}}$

Die Distanz zwischen zwei Vektoren $\vec{u}$ und $\vec{v}$ ist $\parallel \vec{u} - \vec{v} \parallel$ wobei $\parallel \cdot \parallel$ eine beliebige Norm ist. Die am häufigsten verwendete Norm ist die Euklidische Norm $\parallel \vec{u} - \vec{v} \parallel_2$ [15].

### 1.4.2 LSH Familie basierend auf der p-stabilen Verteilung

In diesem Unterkapitel wird die bei der Implementierung angewandte LSH Familie $\mathcal{H}$ im Detail vorgestellt. Da sich die Punkte, mit denen gearbeitet wird, in $l_p^d$ befinden, kann ohne Verlust von Generalität der Suchradius *R* mit eins angenommen werden. Wäre dies nicht der Fall, wäre es möglich, den Abstand zum Suchpunkt um den Faktor *R* zu verkleinern.
Jede Hashfunktion dieser Familie $h_{\vec{a},b}(\vec{v}) : \Re^d \rightarrow \mathbb{Z}$ bildet einen *d*-dimensionalen Vektor $\vec{v}$ auf eine Menge an ganzen Zahlen ab:

$$h_{\vec{a},b}(\vec{v}) = \lfloor \frac{\vec{a} \cdot \vec{v} + b}{\omega} \rfloor$$

Dabei ist $\vec{a}$ ein Zufallsvektor, dessen Einträge zufällig und unabhängig aus einer p-stabilen Verteilung ausgewählt werden. Die Zahl *b* ist eine reelle Zufallszahl, die aus einem Bereich zwischen $[0, \omega]$ ausgewählt wird.
Das Skalarprodukt $[\vec{a}.\vec{v}]$ projiziert jeden Vektor auf einen Gerade. Auf Grund der p-Stabilität ist die Distanz zwischen der Projektion ($\vec{a}.\vec{v} - \vec{a}.\vec{u}$) zweier Vektoren ($\vec{u}, \vec{v}$) wie $\parallel \vec{v} - \vec{u} \parallel_p X$ verteilt [14], wobei *X* eine Zufallsvariable einer p-stabilen Verteilung ist.
In weiterer Folge wird die Gerade, auf die die Vektoren zuvor projiziert wurden, in gleich lange Segmente unterteilt (sieh Abbildung 1.7), die je eine Länge $\omega$ besitzen [1, 14]. Ein solches Segment kann auch als Bucket angesehen werden [2]. Je nachdem, in welches Segment die Vektoren projiziert werden, wird ihnen ein Hashwert zugewiesen, der wiederum das Bucket identifiziert. So werden Punkte, die nahe beieinander liegen, mit größerer Wahrscheinlichkeit in dasselbe Bucket gehashed, als Punkte, die weit voneinander entfernt liegen.

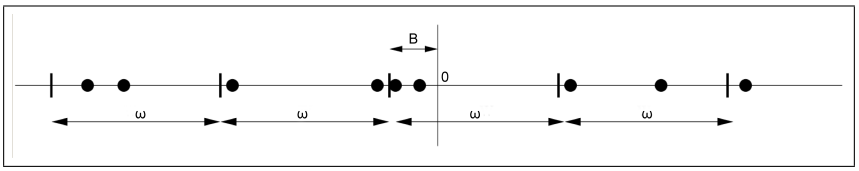


Abbildung 1.7: Schematische Darstellung der Projektion der Vektoren [2]

**Kollisionswahrscheinlichkeit**

In diesem Abschnitt soll die Wahrscheinlichkeit berechnet werden, dass zwei Vektoren mittels einer Hashfunktion, die gleichmäßig und zufällig von der zuvor genannten Familie stammt, kollidieren, also im selbigem Bucket landen. Angenommen $f_p(t)$ bezeichnet die Wahrscheinlichkeitsdichtefunktion des absoluten Wertes der p-stabilen Verteilung [1, 14].
Für zwei Vektoren $\vec{u}, \vec{v}$ bezeichnet $s = \| \vec{u} - \vec{v} \|_p$ die euklidische Distanz unter der $l_p$ Norm, und $p(s)$ die Wahrscheinlichkeit, dass beide Vektoren mittels einer Hashfunktion, die wieder gleichmäßig aus der Familie $\mathcal{H}$, wie zuvor beschrieben, ausgewählt wurde, in das selbe Bucket gehashed werden [1, 14].
Für einen zufälligen Vektor $\vec{a}$, dessen Einträge aus einer p-stabilen Verteilung ausgewählt werden, ist $\vec{a}.\vec{v} - \vec{a}.\vec{u}$ genauso verteilt wie $sX$, wobei *X* wieder eine zufällig aus einer p-stabilen Verteilung ausgewählte Variable ist [14]. Da, wie bereits erwähnt, *b* gleichmäßig aus dem Intervall $[0, \omega]$ ausgewählt wird [14], dann gilt für $s = l_p(\vec{u}, \vec{v})$ folgende Wahrscheinlichkeit [2]:

$$p(s) = Pr_{\vec{a},b}[h_{\vec{a},b}(\vec{v}) = h_{\vec{a},b}(\vec{u})] \leq \int_0^{\omega} \frac{1}{s} f_p(\frac{t}{s})(1 - \frac{t}{\omega})dt$$

Für einen fixen Parameter $\omega$ verringert sich die Wahrscheinlichkeit einer Kollision monoton mit steigender Distanz *s*. Daher ist die Funktion ”locality sensitive” (siehe Kapitel 1.3.1). Genau genommen ist sie (R, cR, $p_1$, $p_2$)-sensitive für $p_1 = p(R)$ und für $p_2 = p(cR)$ [14].
Der optimale Wert für $\omega$ ist von der Datenmenge und dem Suchpunkt abhängig. Da aber der Wert $\omega = 4$ gute Resultate bei [1] erzielte, wurde er auch in die Implementierung übernommen.

### 1.4.3 Wichtige LSH Parameter

Die zwei bedeutendsten Parameter sind die Anzahl der Hashfunktionen *k* [1, 14], die aus einer Familie $\mathcal{H}$ ausgewählt werden, und die Anzahl der unter-

schiedlichen Hashtabellen *L*. Mit nur zwei Schritten können die Parameter *k* und *L* ermittelt werden, die optimal für eine bestimmte Datenmenge sind. Zuerst müssen die Grenzen der beiden Variablen ausgelotet werden, um zu garantieren, dass der Algorithmus korrekt läuft. Im zweiten Schritt muss innerhalb dieser definierten Grenzen ein Wert für *k* und *L* bestimmt werden, der die beste Such- bzw. Laufzeit mit sich bringt.
Die Datenstruktur soll einen cR-NN mit einer Wahrscheinlichkeit von mindestens $1 - \delta$ liefern, wobei $\delta$ die Wahrscheinlichkeit ist, mit der ein cR-NN nicht gefunden wird. Um zu analysieren, welche Bedingungen erfüllt sein müssen, um dieses Verhalten zu veranlassen, wird ein Suchpunkt *x* und ausgehend davon ein wissentlicher "R-near neighbor", der Punkt *y* angenommen. Wie schon in Kapitel 1.4.2 erwähnt, wird zusätzlich angenommen, dass der Punkt mit der Wahrscheinlichkeit $p_1 = p(R)$ innerhalb des Radius liegt. Daraus folgt, dass $Pr_{g \in G}[g(x) = g(y)] \geq p_1^k$, wobei die Funktion $g \in G$ noch genauer in Kapitel 1.4.5 beschrieben wird. Dadurch werden die Punkte *x* und *y* in keiner der *L* Funktionen $g_i$ mit der Wahrscheinlichkeit von höchstens $(1 - p_1^k)^L$ kollidieren [14, 1].

### Wahl der Parameter

Durch die Tatsache, dass ein Suchpunkt *x* mit einem anderen Punkt *y* mittels einer Funktion $g_1$ kollidiert, siehe Kapitel 1.4.5, gilt für diese Wahrscheinlichkeit $1 - (1 - p_1^k)^L \geq 1 - \delta$. Dies hat zur Folge, dass der Parameter *L* wie folgt angenommen werden kann [14, 1]:

$$L \geq \frac{log\frac{1}{\delta}}{-log(1 - p_1^k)}$$

Da aber *L* für ein fix vorgegebenes *k* so klein wie möglich gewählt werden soll, ist die geeignetste Wertigkeit für $L = \lceil \frac{log\frac{1}{\delta}}{-log(1-p_1^k)} \rceil$ [14, 1].
Der Parameter *k* kann als einziger Parameter vom Benutzer frei gewählt werden und bei optimaler Wahl kann die Such- und Laufzeit minimiert werden. Um

das Verständnis für die Wahl des Parameter *k* zu verbessern, soll hier noch ein kurzer Überblick über dessen Einfluss auf das Zeitverhalten gegeben werden. Dafür wird die Laufzeit in zwei Terme, $T_g$ und $T_c$ unterteilt [14, 1].
Der erste Term, $T_g$, ist die Zeit, die benötigt wird, um die *L* Funktionen $g_i$ für einen Suchpunkt *x* zu berechnen, sowie zusätzlich die Zeit, die es benötigt, die Buckets $g_i(x)$ von den Hashtabellen zu erlangen. Somit ergibt sich für den Term $T_g = O(dkL)$ [14]. Zur Erläuterung sei hier angemerkt, dass für die Berechnung von *k* Hashfunktionen jedesmal jede Koordinate des Punktes *x* betrachtet wird, und somit eine Hashfunktionsberechnung $O(dk)$ ist. Durch den bereits erwähnten Einsatz von mehreren unterschiedlichen Hashtabellen *L* muss die Hashfunktionsberechnung *L*-mal durchgeführt werden und somit ergibt sich eine Teillaufzeit von $O(dkL)$.
Der zweite Term, $T_c$, steht repräsentativ für die Zeit, die benötigt wird, um die Distanz vom Suchpunkt zu allen anderen Punkten zu berechnen, die zuvor aus den entsprechenden Buckets erhalten wurden. Dabei ergibt sich für $T_c$ eine Laufzeit von $O(d \cdot \#collisions)$. Die Variable *#collisions* steht für die Anzahl der Punkte die in den selben Buckets $g_1(x), g_2(x) \dots g_L(x)$ für einen bestimmten Suchpunkt *x* gefunden wurden. Der Erwartungswert für $T_c$ ist wie folgt [14]:

$$E[T_c] = O(d \cdot E[\#collisions]) = O(dL \cdot \sum_{y \in P} y^k(\parallel x - y \parallel))$$

Dadurch ergibt sich, dass $T_g$ sich in Abhängigkeit von *k* vergrößert, wobei sich $T_c$ in Abhängigkeit von *k* verkleinert. Letzteres entspringt der Tatsache, dass, wenn *k* vergrößert wird, die Kluft zwischen der Kollisionswahrscheinlichkeit von sowohl nahen als auch weiter entfernten Punkten vergrößert wird. Jedoch, für geeignete Werte von *L*, wird die Wahrscheinlichkeit, dass weit entfernte Punkte kollidieren, verringert [14]. Daher existiert typischerweise ein optimaler Wert für *k*, der die Summe von $T_g + T_c$ für einen bestimmten Suchpunkt *x* minimiert. Jedoch kann es auch unterschiedliche optimale *k* Werte für unterschiedliche Suchpunkte geben. Das Ziel ist es hierbei, den Durchschnitt der Laufzeit für alle Suchpunkte zu optimieren [14]. Um den Wert *k* zu berechnen empfiehlt Indyk in [1], experimentell die Werte für $T_g$ als eine Funktion von *k* zu berechnen.

Um $T_c$ zu berechnen, welche von einem bestimmten Suchpunkt *x* abhängig ist, wird ein Menge *S* von Beispielpunkten herangezogen, welche zufällig aus der tatsächlichen Suchmenge extrahiert werden können. Der Wert für $T_c$ wird erhalten, indem der Durchschnitt aller erhaltenen $T_c$ Zeiten für *S* berechnet wird [1, 14].
Um die Terme genau zu schätzen, müssen die Konstanten der Funktionen für $T_g$ und $T_c$ bekannt sein. Um diese unbekannten Konstanten zu berechnen, wird eine Beispieldatenstruktur erstellt, auf Basis derer mehrere Suchanfragen abgewickelt werden um die tatsächlichen Zeiten von $T_g$ und $T_c$ zu messen [1]. Der Wert für *k* wird so gewählt, dass $T_G + \tilde{T}_c$ minimal ist. Der Term $\tilde{T}_c$ steht für den Mittelwert aller berechneten $T_c$ von allen Punkten in der Beispielmenge $S$ :$\tilde{T}_c = \frac{\sum_{x \in S} T_c(x)}{|S|}$ [1, 14].

### 1.4.4 Berechnung der Wahrscheinlichkeiten

Für die Spezialfälle der p-stabilen Verteilungen wenn $p = 1, 2$ ist, können die Wahrscheinlichkeiten $p_1$ und $p_2$, welche bereits in Kapitel [1.3.1] vorgestellt wurden, mit Hilfe der Dichtefunktion berechnet werden. Durch eine einfache Berechnung ergibt sich für $p = 1$ (Cauchy)

$$p_2 = 2\frac{tan^{-1}\frac{\omega}{c}}{\pi} - \frac{1}{\pi\frac{\omega}{c}}$$

und für den Fall dass $p = 2$, also eine Gauss'sche Verteilung vorliegt,

$$p_2 = 1 - 2norm(\frac{-\omega}{c} - \frac{2}{\sqrt{2\pi}\frac{\omega}{c}}(1 - e^{-(\frac{\omega^2}{ec^3})}$$

Die Norm $norm(\cdot)$ ist die kumulative Verteilungsfunktion (cumulative distribution function - cdf) für eine Zufallsvariable, die wie $N(1, 0)$ verteilt ist. Der Wert für die Wahrscheinlichkeit $p_1$ kann ebenfalls aus den zuvor erwähnten Formeln gewonnen werden, indem für den Approximationsfaktor $c = 1$ eingesetzt wird [2, 14].

### 1.4.5 LSH Datenstruktur

Da die Hashfunktionen $g_i$ auf den gesamten Integerwertebereich abbilden, kann nicht für jedes mögliche Ergebnis bereits im Vorfeld ein Bucket erstellt werden. Daher werden in der Datenstruktur nur nicht-leere Buckets gespeichert [1]. Jeder Vektor $\vec{v}$ wird, wie beschrieben, in jeder der *L* Hashtabellen in einem Bucket entsprechend ihrem Hashwert $g_1(\vec{v}) \dots g_L(\vec{v})$ gespeichert. Dabei werden die *L* Hashtabellen wir folgt realisiert:
Es existieren zwei weitere globale Hashfunktionen $h_1 : \mathbb{Z}^k \to \{0, \dots, tableSize - 1\}$ und $h_2 : \mathbb{Z}^k \to \{0, \dots, chainlength - 1\}$. Jede Funktion $g_i$ mapped einen Vektor auf $\mathbb{Z}^k$. Die Funktion $h_1$ hat allgemein die Rolle einer universellen Hashfunktion und die Hashfunktion $h_2$ dient der Identifikation der einzelnen Buckets in der entsprechenden Kette, da die Kollisionen innerhalb der Buckets mittels chaining aufgelöst werden [1].
Wenn ein Bucket $g_i(\vec{v}) = (x_1, \dots x_k)$ in der entsprechenden Kette angehängt wird, wird nicht der ganze Vektor $(x_1, \dots, x_k)$ gespeichert, um diesen zu identifizieren, sondern nur ein Wert $h_2(x_1, \dots, x_k)$. Daher besitzt ein Bucket $g_i(\vec{v}) = (x_1, \dots, x_k)$ lediglich den sogenannten Fingerprint $h_2(x_1, \dots, x_k)$ und die Punkte selbst [1]. Die Anwendung der zweiten Hashfunktion $h_2$ anstatt des Wertes $g_i(\vec{v}) = (x_i, \dots, x_k)$ hat zwei Gründe. Zuerst wird der Wert der zweiten Hashfunktion $h_2(x_1, \dots, x_k)$ als eine Art Fingerprint für den Vektor herangezogen, was die Laufzeit für die Identifikation im Bucket von $O(k)$ auf $O(1)$ verringert. Zum zweiten ist es mittels dieses Fingerprints einfacher und schneller möglich, einen Vektor im entsprechenden Bucket zu finden [1]. Die Domain der Funktion $h_2$ wird groß genug gewählt, um auch wirklich sicher zu gehen, dass mit einer hohen Wahrscheinlichkeit keine zwei unterschiedlichen Vektoren in einer Kette denselben $h_2$ Wert aufweisen [1].
Alle *L* Hashtabellen verwenden dieselbe primäre Hashfunktion $h_1$, welche zur Berechnung des Index in der Hashtabelle dient, und dieselbe sekundäre Hashfunktion $h_2$ [1]. Die zwei Hashfunktionen $h_1$ und $h_2$ lauten wie folgt:

$$h_1(x_0 1, x_2, \dots, x_k) = \left( \left( \sum_{(i=1)}^{k} r' x_i \right) mod \textit{ prime} \right) mod \textit{ tableSize}$$

$$h_2(x_1, x_2, \ldots, x_k) = \left( \sum_{i=1}^{k} (r'' x_i \right) mod \textit{ prime}$$

Die Parameter $r_i'$ und $r_i''$ sind zufällige Integer-Werte, *tablesize* ist die Größe der Hashtabelle und *prime* ist, wie der Name schon sagt, eine Primzahl [1]. Laut Indyk in [1] sollte $tablesize = |\mathcal{P}|$ sein und $x_i$ sollte ein 32-bit Integer sein, der das Ergebnis der Hashfunktion darstellt, wie in Kapitel 1.4.2 beschrieben. Die Primzahl *prime* ist $2^{32} - 5$ was eine schnelle Berechnung von Hashfunktionen ermöglicht, ohne dabei Modulo-Operationen zu verwenden [1]. Für den Fall dass $k = 1$ ist, kann für $h_2(x_1)$ die Berechnung etwas umgestaltet werden:

$$h_2(x_1) = (r_1'') \, mod \, (2^{32} - 5) = (\textit{low}[r_1'' x_1]) \, mod \, (2^{32} - 5)$$

Dabei sind $\textit{low}[r_1'' x_1]$ die niederwertigen 32 Bits von $r_1'' x_1$ das eine 64-bit Zahl bildet. Somit sind $\textit{high}[r_1'' x_1]$ die hochwertigeren 32 Bits von $r_1'' x_1$ [1]. Wenn der Wert für $r_i''$ in einem Intervall von $[1, \ldots, 2^{29}]$ gewählt wird, gibt es immer ein $\alpha$ das folgende Eigenheiten aufweist:

$$\alpha = \textit{low}[r_1'' x_1] + 5 * \textit{high}[r_1'' x_1] < 2 * (2^{32} - 5)$$

Dies hat zur Folge, dass gilt: [1]:

$$h_2(x_1) = \begin{cases} \alpha & , wenn \; \alpha < 2^{32} - 5 \\ \alpha - (2^{32} - 5) & , wenn \; \alpha \geq 2^{32} - 5 \end{cases}$$

Für den öfter auftretenden Fall, dass $k > 1$ ist, kann die Summe $\left( \sum_{i=1}^{k} r_i'' x_i \right) mod \, prime$ berechnet werden. Dabei wird der Wert immer modulo der Teilsumme $2^{32}$, nach dem bereits beschrieben Prinzip, berechnet. Somit ist der Wertebereich den die Funktion $h_2$ annehmen kann nur noch $[1, \ldots, 2^{32} - 6]$

### Erstellen der Datenstruktur

Zuerst müssen die Parameter bestimmt werden, die als Ausgangsbasis für die Berechnung der LSH-Konstanten dienen. Insgesamt gibt es drei davon. Zum einen die Wahrscheinlichkeit $\delta$ mit der ein cR-NN gefunden werden soll, die wie in [1] beschrieben mit $0.90$ angenommen werden soll, und die Intervallslänge $\omega$, die für eine annähernd optimale Datenstruktur den Wert 4 besitzen soll. Für den Parameter *k* wurde von [1] kein konkreter Vorschlag gegeben, lediglich dass der Wert zwischen eins und zehn liegen soll. Wurden diese drei Parameter festgelegt, kann begonnen werden, die restlichen Konstanten zu berechnen und die Datenstruktur zu erstellen und zu befüllen. Der erste zu berechnende Wert ist *L*, die Anzahl der Hashtabellen, mit der in Kapitel 1.4.3 beschriebenen Formel. Da hierfür die Wahrscheinlichkeit $p_1$ benötigt wird, so muss diese im Vorfeld berechnet werden. Die dafür notwendige Formel befindet sich im Kapitel 1.4.4 und ist dort die Formel für $p_2$ für die Gauss'sche Verteilung. Um den gewünschten Wert $p_1$ aus dieser Formel zu berechnen muss lediglich $c = 1$ gesetzt werden. Zum Einfügen eines Punkts in die Datenstruktur müssen noch einige Zufallszahlen und Zufallsvektoren erstellt werden. Zur Berechnung der Hashfunktion, die in Kapitel 1.4.2 beschrieben ist, sind für jede der *L* Hashtabellen insgesamt *k* Zufallsvektoren $\vec{a}$ und *k* Zufallszahlen *b* notwendig. Weiters sind für die beiden Formeln aus Kapitel 1.4.5 jeweils eine Zufallszahl $r'$ und $r''$ notwendig. Die Ergebnisse der *k* Hashfunktionen aus Kapitel 1.4.2 dienen als Input für die Formeln $h_1$ aus Kapitel 1.4.5, zum Berechnen des Bucket in der Hashtabelle, und zur Berechnung von $h_2$ als Fingerprint im Bucket.
Zusammengefasst kann gesagt werden, dass für jeden einzufügenden Punkt zuerst die *k* Hashfunktionen aus 1.4.2 berechnet werden müssen und der k-dimensionale Ergebnisvektor als Input für die Hashfunktionen $h_1$ und $h_2$ aus Kapitel 1.4.5 dient, mit welcher die Punkte schlussendlich in die Datenstruktur eingefügt werden.

### 1.4.6 Optimierung des Algorithmus

#### Optimierung der Vorverarbeitung

Mit Hilfe von kleineren Veränderungen im Algorithmus ist es möglich, die Zeit $T_g$ welche bereits in Kapitel 1.4.3 beschrieben wurde, und zur Berechnung der $g_i$ Funktionen dient, zu verkürzen [1].
Bei der nicht optimierten Version des LSH [1] existieren *k* Funktionen $g_i = (h_1^{(i)}, \ldots, h_k^{(i)})$ von denen jede Funktion $h_j^{(i)}$ zufällig aus einer LSH Familie $\mathcal{H}$ gewählt wird [1]. Für jeden Punkt *x* wird in $O(d)$ Zeit die Hashfunktion $h_j^{(i)}$ berechnet und es benötigt $O(dkL)$ Zeit um alle Funktionen $g_1(x), \ldots g_L(x)$ zu berechnen.
Um nun diese Zeit zu verringern, werden einige der Funktionen $h_j^{(i)}$ mehrfach verwendet. In diesem Fall sind $g_i$ nicht total unabhängig. Zusätzlich zu der Funktion $g_i$ werden Funktionen $u_i$ mit folgendem Hintergedanken erstellt: Angenommen der Parameter *k* ist eine gerade Zahl und *m* ist eine fixierte Konstante. Dann kann die Funktion $u_i$ für jedes $i = 1\ldots, m$ definiert werden als $u_i = h_j^{(1)}, \ldots h_{\frac{k}{2}}^{(i)}$, wobei $h_j^{(i)}$ zufällig und unabhängig aus der bereits bekannten Familie $\mathcal{H}$ zu wählen ist [1]. Dies hat zur Folge, dass es sich bei $u_i$ um Vektoren handelt, deren Einträge von $\frac{k}{2}$ Funktionen erzeugt werden, welche wiederum zufällig aus einer LSH Familie $\mathcal{H}$ gewählt werden. Nun können folglich die Funktionen $g_i$ definiert werden mit $g_i = (u_a, u_b)$ wobei $1 \leq a \leq b \leq m$ ist. Insgesamt gibt es *L* dieser Funktionen $g_i$ mit $L = m\frac{m-1}{2}$ [1].
Da die Funktionen $g_i$ zwar unabhängig, aber nicht total unabhängig sind, muss eine alternative Wahrscheinlichkeit abgeleitet werden, mit der der Algorithmus einen Punkt innerhalb eines Radius, ausgehend von einem Suchpunkt meldet. Durch diese neuen Funktionen $g_i$ ist diese Wahrscheinlichkeit größer oder kleiner gleich $1 - \left(1 - p_1^{\frac{k}{2}}\right)^m - m * p_1^{\frac{k}{2}} * \left(1 - p_1^{\frac{k}{2}}\right)^{m-1}$. Um nun folglich eine Erfolgswahrscheinlichkeit von $1 - \delta$ zu erlangen, wird der Parameter *m* so beschränkt, dass gilt $\left(1 - p_1^{\frac{k}{2}}\right)^m + m * p_1^{\frac{k}{2}} * \left(1 - p_1^{\frac{k}{2}}\right)^{m-1} \leq \delta$. Durch diese Ungleichung wird zwar der Wert für $L = m\frac{m-1}{2}$ kleiner als für den Fall, dass die Funktionen $g_i$ unabhängig sind, aber die Berechnung von *L* ist dennoch in $O\left(\frac{log\frac{1}{\delta}}{p_1^k}\right)$ [1] Zeit

möglich.
Die Zeit, die für die Berechnung von den $g_i$ Funktionen benötigt wird, um für einen Suchpunkt *x* den oder die Nachbarn zu finden, wird verringert auf $T_g = O(dkm) = O\left(dk\sqrt{L}\right)$. Dies folgt aus der Tatsache, dass für die Berechnung von den $g_i$ Funktionen nur *m* Funktionen $u_i$ berechnet werden müssen.
Die Berechnung von $T_c$, welche die Laufzeit für die Berechnung der Distanz zu den Punkten in den Buckets $g_1(x), \ldots, g_L(x)$ angibt, bleibt unverändert, aufgrund der Linearität des Erwartungswertes.
Dadurch ergibt sich für die Berechnung der Parameter eine kleine Änderung. Zusätzlich muss der Parameter *m* berechnet werden, der in Abhängigkeit von *k* so gewählt wird, dass *m* die kleinste natürliche Zahl ist, welche folgende Gleichung erfüllt:

$$\left(1 - p_1^{\frac{k}{2}}\right)^m + m * p_1^{\frac{k}{2}} * \left(1 - p_1^{\frac{k}{2}}\right)^{m-1} \leq \delta$$

Eine weitere kleinere Änderung ist die Berechnung des Parameters *L* der nun mit $L = m\frac{m-1}{2}$ festgelegt ist. Somit ist aus beiden Berechnungen ersichtlich, dass sowohl *m* als auch *L* als Funktion von *k* berechnet werden, welches wieder zur Wahl des entsprechenden *k* führt, dessen genauere Berechnung bereits im Kapitel 1.4.3 unter die Lupe genommen wurde.

### Optimierung der Hashfunktionsberechnung

Um eine Suche zu tätigen, müssen, wie bereits mehrfach erwähnt, zuerst die Funktionen $g_i(x)$ berechnet werden. Im Falle, dass die Optimierung von Kapitel 1.4.6 angewendet wird, müssen lediglich, da $g_i = (u_a, u_b)$, $\frac{k}{2}$ Tupel $u_a(q)$ für $a = 1, \ldots, m$ berechnet werden [1]. Um einen Vektor $g_i(x)$ in einer Hashtabelle zu finden, müssen lediglich die zwei Hashfunktionen $h_1(g_i(x))$ und $h_2(g_i(x))$ berechnet werden. Diese beiden Hashfunktionen schon im Vorfeld zu berechnen, würde $O(Lk)$ Zeit in Anspruch nehmen. Da $g_i$ der Form $(u_a, u_b)$ ist, kann die Zeit für die Berechnung wie folgt vermindert werden. Jede Funktion $h_j, j \in \{1, 2\}$

ist von der Form $h_j(x_1, \ldots, x_k) = \left(\left(\sum_{i=1}^{k} r_i^j x_i\right) mod\ A_j\right) mod\ B_j$. Daher kann $h_j(x_1, \ldots, x_2)$ auch folgendermaßen berechnet werden:

$$\left(\left(\sum_{i=1}^{\frac{k}{2}} r_i^j x_i + \sum_{i=\frac{k}{2}}^{k}\right) mod\ A_j\right) mod\ B_j$$

Wenn $r_{left}^j = \left(r_1^j, \ldots, r_{\frac{k}{2}}^j\right)$, und entsprechend dazu $r_{right}^j = \left(r_{\frac{k}{2}+1}^j, \ldots, r_k^j\right)$ bezeichnet wird, können diese beiden Funktionen in die Hashfunktion integriert werden was,

$$h_j(g_i(x)) = \left(\left(r_{left}^j * u_a(\vec{v}) + r_{right}^j * u_b(\vec{v})\right) mod\ A_j\right) mod\ B_j$$

ergibt [1]. Daher ist es auch völlig ausreichend, nur $r_{side}^j * u_a(\vec{v})$ zu berechnen, wobei $j \in \{1, 2\}$ ist, $side \in \{left, right\}$ und $a \in \{1, \ldots, m\}$ definiert ist, was zu einer verkürzten Berechnungszeit von $O(km)$ führt [1].

### Filterung doppelter Ergebnisse

Werden keine entsprechende Maßnahmen ergriffen, kann es durchaus vorkommen, dass der Algorithmus einen Punkt $y \in \mathcal{P}$ öfter als nur einmal findet. Dies kommt vor allem dann vor, wenn ein Punkt $y \in \mathcal{P}$ in mehr als nur einem Bucket $g_1(x), \ldots, g_L(x)$ auftaucht [1]. Da es auch keine Notwendigkeit gibt, die Distanz zu einem Punkt $y \in \mathcal{P}$ öfter als nur einmal zu berechnen, kann vermerkt werden, zu welchen Punkten bereits die Distanz $\| x - y \|$ berechnet wurde. Daher wird eigens für jede Suche ein Vektor $\vec{e_i}$ erstellt. In diesem wird, wenn der Punkt $y_1 \in \mathcal{P}$ bereits in einem früheren Bucket gefunden wurde, $\vec{e_i} = 1$ gespeichert. Im Falle, dass die Distanz $\| x - y \|$ noch nicht berechnet wurde, ist $\vec{e_i} = 0$ [1]. Sowie ein Punkt *y* zum ersten Mal in einem Bucket gefunden wurde, wird dessen Distanz zum Suchpunkt berechnet, was $O(d)$ Zeit in Anspruch nimmt. Für alle folgenden Male, wenn der Punkt *y* gefunden wird, wird lediglich $O(1)$ Zeit aufgewendet, um im entsprechenden Vektor unter $\vec{e_i}$ nachzusehen [1].

Durch dieses kleine Optimierung ändert sich auch die Zeit $T_c$, welche Auskunft über die Dauer gibt, die benötigt wird, um die Distanz zu allen Punkten in den gefundenen Buckets zu berechnen. Mit Bedacht auf diese Änderung ergibt sich eine neue Formel für den Parameter $T_c$, bei dem auch gleichzeitig die Optimierung von 1.4.6 einfließt [1]:

$$E\left[T_c\right] = d * \sum_{y \in P} \left(1 - \left(1 - p\left(\| x - y \|\right)^{\frac{k}{2}}\right)^m - m * p\left(\| x - y \|\right)^{\frac{k}{2}} * \left(1 - p\left(\| x - y \|\right)^{\frac{k}{2}}\right)^{m-1}\right)$$

# 2 Eigene Implementierung

Ziel der Implementierung war es, eine wie in [1] beschriebene Datenstruktur in Java zu entwickeln, die das Problem des approximate-NN löst, da der originale Algorithmus von [1] in der Programmiersprache *C* implementiert wurde und noch keine Implementierung in Java existiert. Der in [1] beschriebene Algorithmus bzw. die in [1] beschriebene Datenstruktur wurden als Basis für die eigene Implementierung herangezogen.

## 2.1 Datenstruktur

Durch die von [1] abweichende Programmiersprache ergaben sich einige kleine Eigenheiten und Änderungen, um sich der entsprechenden Programmiersprache, in diesem Falle Java, anzupassen. Folgendes Kapitel soll einen Überblick geben, wie die Eigenschaften des Algorithmus in Java realisiert wurden.
Im Wesentlichen besteht die Implementierung aus einer großen Klasse *LSH* (siehe Abbildung 2.1), die alle Funktionen aufweist, die für die Realisierung von Locality Sensitive Hashing notwendig sind. Jeder Vektor wird nur einmal für das System erstellt und als ein Array des Datentyps Float gespeichert. In weiterer Folge wird nur noch mit Referenzen, anstatt mit Pointern, die in Java nicht existieren, auf das jeweilige Array zugegriffen. Diese Referenzen werden in der Klasse *LSHEntry,java* mit dem jeweiligen Fingerprint (siehe Kapitel 1.4.5) gespeichert. Dieser Fingerprint ist der dort beschriebene $h_2$ Hashwert, der einem schnelleren Zugriff auf den Vektor, im Falle einer Löschoperation, dient. Hierbei

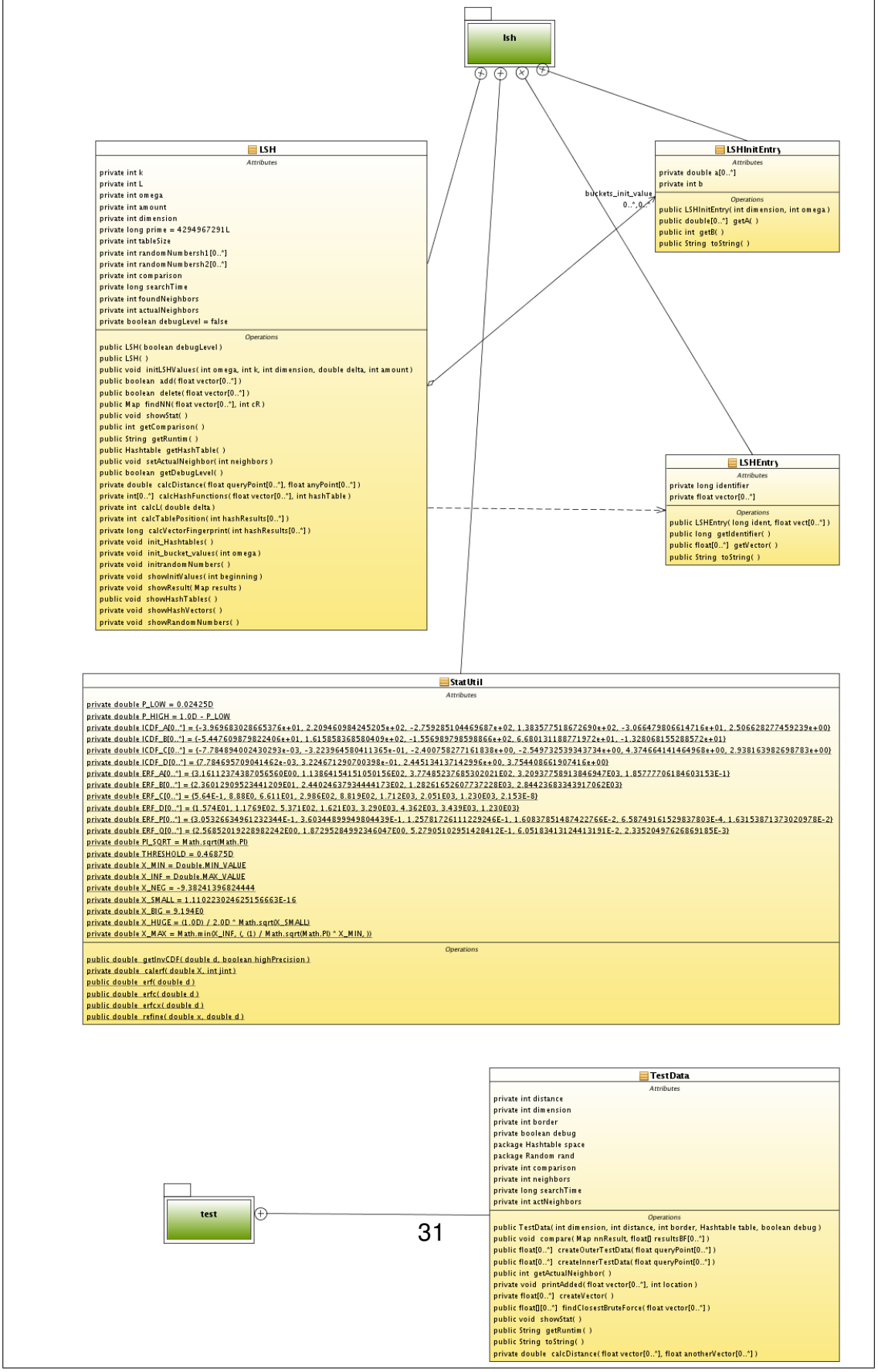




Abbildung 2.1: Klassendiagramm der LSH Implementierung

muss nicht mehr der ganze Vektor, sondern nur der Hashwert $h_2$ direkt verglichen werden, was die Laufzeit der Löschoperation erheblich verringert. Für andere Operationen ist im Gegensatz zur C-Implementierung der Fingerprint nicht wirklich von Nöten, da Referenzen direkt verglichen werden können.
Wird ein neues Objekt der Klasse LSH erstellt, stehen zwei leicht unterschiedliche Konstruktoren zur Verfügung.

```
LSH ()
LSH(int debugLevel)
```

Beim ersteren wird ein einfaches Standardobjekt ohne zusätzliche Parameter erstellt. Der zweite Konstruktor erstellt ein Objekt, bei dem der Debug-Modus, (siehe Kapitel 2.2), aktiviert ist. Im Debug-Modus wird zusätzlich ein detaillierter Output generiert. Im Gegensatz dazu werden bei einem Standardobjekt lediglich die Statistikwerte ausgegeben, die ebenfalls näher im Kapitel 2.2 und 2.3.1 beschrieben werden.
Für die Erstellung der Datenstruktur werden noch die in Kapitel 1.4.3 beschriebene Parameter benötigt. Diese werden dem Algorithmus mit der

```
initLSHValues(int omega, int k, int dimension, double delta, int amount)
```

Methode übergeben. Der Parameter $\omega$ gibt die Länge der Intervalle der Zahlengerade an (vgl. 1.4.2), auf die die Vektoren durch die Hashfunktion gemapped werden. Der Parameter *k* gibt die Anzahl der Hashfunktionen für jede Hashtabelle an. Die Dimension wird ebenfalls gleich mit übergeben, anstatt sie mithilfe der Länge der Arrays eines Vektors selbst zu eruieren. Denn die Dimension des Vektors wird bereits für die Initialisierung in der Methode
`init_bucket_values(int omega)` benötigt. In dieser sehr kleinen und unscheinbaren Methode werde gleich zu Beginn der für die Hashfunktion $h_{\vec{a},b}(\vec{v}) = \lfloor \frac{\vec{a} \cdot \vec{v} + b}{\omega} \rfloor$ sehr essentielle Vektor $\vec{a}$ und die Zufallszahl *b* erstellt. Die Aufgabe der Klasse `LSHInitEntry`, die in dieser Methode ein Objekt erstellt, wird im Laufe dieses Kapitels noch näher beschrieben.
Der Parameter $\delta$ steht, wie in Kapitel 1.1 beschrieben, für die Wahrscheinlichkeit, dass ein Nearest Neighbor gefunden wird. Will sich der Benutzer bei der

Wahl des Wertes von $\delta$ nicht festlegen, so wird, wie in [1] vorgeschlagen, standardmäßig eine Wahrscheinlichkeit von $0.10$ angenommen, wodurch ein Nearest Neighbor mit der Wahrscheinlichkeit von $0.90$ gefunden wird. Mithilfe dieses $\delta$ wird mittels einer kleinen Hilfsfunktion `calcL(double delta)` der Parameter *L*, der die Anzahl der verwendeten Hashtabellen widerspiegelt, (siehe Kapitel 1.4.3) berechnet. Der letzte Parameter, der der Methode `initLSHValues(...)` übergeben wird, ist die Anzahl der Vektoren insgesamt, d.h., $|\mathcal{P}|$, der für die Initialisierung der *L* Hashtabellen notwendig ist, da diese standardmäßig wie in [1] beschrieben mit der Größe $|\mathcal{P}|$ erstellt werden.
Wie bereits ersichtlich, ist die wesentliche Aufgabe dieser Methode die Berechnung aller nötigen Parameter und die Erstellung aller nötigen Objekte, die für die Verwendung der Datenstruktur notwendig sind.
Sowie der Wert für *L* bekannt ist, werden alle *L* Hashtabellen mit der Größe `amount` erstellt. Zur einfacheren Verwaltung dieser wird ein Array `hashTable_container` bestehend aus Hashtabellen erstellt. Somit kann in $O(1)$ auf jede entsprechende Hashtabelle zugegriffen werden. Durch die kleine Hilfsfunktion `init_Hashtables()` werden die Hashtabellenobjekte für das Array erstellt. Jede dieser Hashtabellen besteht aus den Datentypen `<Integer, ArrayList>` . Als `Integer` wird der Wert der Hashfunktion $h_1$ gespeichert (vgl. Kapitel 1.4.5) und in der entsprechenden Arrayliste die jeweiligen Vektoren, die denselben Hashwert aufweisen, sprich in dasselbe Bucket gehashed werden.
Da für die Hashfunktionen $h_1$ und $h_2$, wie in 1.4.5 beschrieben, für jede der *L* Tabellen unterschiedliche Zufallszahlen, $r_i'$ und $r_i''$, benötigt werden, werden diese im Vorfeld in Arrays gespeichert. Die Methode `initRandomNumbers` füllt die zwei entsprechenden Arrays mit zufälligen `Integer` Werten.
Ein weiteres interessantes Detail der Implementierung ist in der Methode `calcL(double delta)` versteckt, die den Parameter *L* berechnet. Um die Formel für *L* aus Kapitel 1.4.3 berechnen zu können, wurde eine externe Klasse `Statutil` dem Projekt beigefügt, die statistische Hilfsfunktionen beinhaltet. Sie übernimmt die Berechnung der komplementären Errorfunktion, und wird für die Berechnung des Wertes $norm(\cdot)$ aus der Formel für die Wahrscheinlichkeit $p_2$ aus Kapitel 1.4.4 benötigt. Die Klasse wurde von Sherali Karimov entwickelt und ist unter

http://home.online.no/~pjacklam/notes/invnorm/impl/karimov/StatUtil.java

verfügbar. Zum Einsatz kommt jedoch nur die Methode `erf()`.
Da bereits durch die Methode `initLSHValues` die nötige Datenstruktur erstellt und alle Parameter berechnet wurden, können folglich mit der Methode `boolean add(float[] vector)` die Vektoren an ihren entsprechenden Platz in der Datenstruktur eingefügt werden. Wie schon zu Beginn dieses Kapitels erwähnt, arbeitet die Datenstruktur ausschließlich mit `Float` Werten. Somit ist auch der Input für die `add()` Methode ein Floatarray, welches sequentiell in jede Hashtabelle eingefügt wird. Zuerst werden alle *k* Hashfunktionen für die jeweilige Hashtabelle mittels der Funktion `calcHashFunctions(vector, hashTable)` berechnet. Dabei wird eine Referenz des Vektors mitgegeben und ein Integer-Wert, für welchen der *L* Hashtabellen die *k* Hashfunktionen berechnet werden sollen. Dies ist wichtig, da der entsprechende Zufallsvektor $\vec{a}$ und die Zufallszahl *b* ausgewählt werden müssen. Diese sind in einem Objekt der Klasse `LSHInitEntry` gespeichert (siehe Abbildung 2.1). Die Objekte der Klasse `LSHInitEntry` werden in einem zweidimensionalen Array `buckets_init_values` verwaltet. Somit ergibt sich ein Array das *k* lang und *L* hoch ist. An die `add()` Methode wird ein Array zurückgegeben, welches alle *k* Ergebnisse der Hashfunktionen für die jeweilige Hashtabelle beinhaltet. Auf Basis dieser Werte wird dann der Index $h_1$ für die jeweilige Hashtabelle mittels der Funktion `calcTablePosition` wie in Kapitel 1.4.5 beschrieben, berechnet. Folglich wird in der Hashtabelle nachgesehen, ob dieser Wert mit der dazugehörigen `ArrayList` bereits existiert. Für den Fall, dass für den Hashwert $h_1$ bereits eine Liste mit Vektoren existiert, wird der aktuelle Vektor der Arrayliste angehängt (chaining). Der Anhang wird mittels eines neuem `LSHEntry` Objekts vollzogen (siehe Abbildung 2.1). Dieses dient lediglich zur Verwaltung der Vektorreferenzen und deren Fingerprint $h_2$. Das heißt, die Klasse `LSHEntry` besitzt eine Referenz auf ein Floatarray und eine `Long` Variable, die dafür zuständig ist, den Fingerprint zu speichern. Die Berechnung des Fingerprints $h_2$ erfolgt wie in Kapitel 1.4.5 beschrieben.
Im Falle, dass für den berechneten Hashwert $h_1$ noch kein Eintrag bzw. keine Arrayliste existiert, wird an der entsprechenden Stelle für den Hashwert ein Objekt einer neuen `ArrayList` eingefügt und dieser wie bereits beschrieben ein

`LSHEntry` Objekt mit den entsprechenden Werten angefügt.
Da es auch durchaus vorkommen kann, dass einmal ein Vektor aus der Datenstruktur entfernt werden soll, wurde eigens eine Methode `delete(float[] vector)` erstellt, die, wie der Name schon sagt, für das unwiderrufliche Löschen eines Vektors zuständig ist. Für das Löschen wird iterativ jeder Hashtabelle nach dem entsprechenden Vektor durchsucht. Da direkt mittels des Hashwertes $h_1$ auf die entsprechende Arrayliste zugegriffen werden kann, ist die Suchzeit in der Hashtabelle $O(1)$. In der Arrayliste muss folglich nicht mehr jeder Vektor verglichen werden, bis der zu löschende gefunden wurde, es genügt, den Fingerprint $h_2$ zu berechnen und auf Basis dessen den gewünschten Vektor aus der Liste zu selektieren und dessen `LSHEntry` Objekt zu entfernen. Jedoch wird dadurch nur die Referenz des Vektors in der LSH Datenstruktur gelöscht, nicht der Vektor selbst, der im Speicher verweilt.
Zu guter Letzt soll noch das Herzstück der ganzen Klasse beschrieben werden, die Methode `findNN(float[] querypoint)`, welche die annähernd nächsten Nachbarn, ausgehend von einem Suchpunkt, der der Methode übergeben wird, sucht, und mittels einer `Map` zurück gibt. Gleich zu Beginn wird ein Objekt einer Java-Map angelegt, in der die gefundenen Vektoren gespeichert werden. In diese Map werden schlicht zwei Referenzen des Vektors gespeichert, einer als Key und der zweite als Value. Somit kann immer direkt nachgesehen werden, ob ein gefundener Vektor bereits in der Ergebnismenge existiert oder nicht. Diese Überprüfung findet noch vor der eigentlichen Berechnung des Abstandes statt, da eine Referenzabfrage wesentlich schneller ist, als eine Distanzberechnung zwischen zwei Vektoren und die anschließende Prüfung ob das Ergebnis in der Map existiert. Dadurch wird die Zahl der Distanzberechnungen nochmals dezimiert. Für den Statistikoutput, der in Kapitel 2.3.1 beschrieben wird, zählt die Methode gleichzeitig, wie viele Distanzberechnungen nötig waren. Wie schon bei der `add()` und `delete()` Methode, wird auch hier wieder der Hashwert $h_1$ für den Suchvektor berechnet und, in Folge dessen, die Distanz zwischen Vektoren mit dem selben Hashwert $h_1$, also die in dasselbe Bucket gehashed wurden, berechnet. Die `ArrayList`, die als Bucket fungiert, wird iterativ durchwandert und wenn der entsprechenden Vektor noch nicht in der Resultat-Menge enthalten ist, wird die Distanz berechnet, verglichen mit dem maximalen Radius $cR$. Wenn

die Distanz kleiner gleich $cR$ ist, wird eine Referenz, wie bereits beschrieben, in die Map aufgenommen. Von den möglichen Optimierungen aus Kapitel 1.4.6 wurde lediglich die Filterung, welche ansatzweise in Kapitel 1.4.6 beschrieben wurde, durch die Verwendung der bereits erwähnten Map realisiert. Die anderen wurde einstweilen vernachlässigt.

## 2.2 Debugmodus

Wie bereits im Kapitel 2.1 erwähnt, kann bei der Erstellung des LSH Objektes entschieden werden, ob der erweiterte Output "eingeschaltet" sein soll oder nicht. Dies geschieht mit Hilfe des Konstruktors, dem ein Integer Wert übergeben wird oder auch nicht. Dies hat zur Folge dass der Output in unterschiedlichen Detaillierungsgrad angezeigt werden kann. Bei aktiviertem Output in der höchsten Stufe, zwei, werden sämtliche Vektoren ausgegeben, die eingefügt wurden und deren Hashwerte. Der Output wird in mehrere unterschiedliche Sektionen unterteilt. Damit dieser auch noch einigermaßen überschaubar dargestellt werden kann, wurde im folgendem Beispiel, sowohl die Dimension der Vektoren als auch deren Anzahl sehr gering gewählt:

- Gleich zu Beginn des Outputs werden alle wesentlichen Parameter, mit der die Datenstruktur erstellt wird, ausgegeben. Somit ist ersichtlich, ob der Algorithmus mit der gewünschten Ausgangsbasis arbeitet.

```
+++ Basic Values +++
Debug level: true
Amount of vectors (n): 10
Amount of dimension (d): 3
Amount of Hashtables (L): 6
Amount of Hashfunktions per Table (k): 4
Size of Hashtable (s): 10
Primzahl fuer H1 und H1 (prime): 4294967291
+++ End Basic Values +++
```

- Da gleich zu Beginn die *k* Zufallsvektoren $\vec{a}$ für jeden der *L* Tabellen berechnet werden, werden diese, so wie sie erstellt sind, sofort ausgeben, zusammen mit dem entsprechenden Zufallswert *b*. Die folglich angeführte

Liste wurde aus Gründen des Umfangs gekürzt und zeigt lediglich die Zufallsvektoren und Zufallszahlen für die ersten beiden der *L* Hashtabellen an.

```
+++Vectors for Hashfunction Calculation+++
Initial Values for Hashtable: 0
  [ 0 - (-0.10850206010665014 -1.173226077450258 -0.49656059504498035 )]
  [ 1 - (0.26104349593388215 -2.208823803185684 0.33618289966658127 )]
  [ 0 - (-1.2382138531963351 -1.81021392055545 -1.3043291050424202 )]
  [ 1 - (0.9422412118471787 0.43874163125664634 0.367761956171561 )]
Initial Values for Hashtable: 1
  [ 3 - (-0.3567136324009582 -1.1331601023996551 0.7803054865654738 )]
  [ 0 - (0.1286892543130848 0.8529528997122544 -0.5649989764086719 )]
  [ 0 - (-0.6310339749005259 -0.273927029262733 0.02161181586608646 )]
  [ 2 - (-0.3174368524989149 -0.030712628350917967 -0.9840933830317287 )]
Initial Values for Hashtable: 2
...
```

- Für die Berechnung der Position $h_1$ in der Hashtabelle und dem Fingerprint $h_2$ ist ebenfalls jeweils ein Vektor mit Zufallszahlen der Länge *k* notwendig, da nach der Berechnung der Hashfunktionen, die den jeweiligen Vektor auf eine Zahlengerade mappen, jeweils ein *k*-dimensionaler Vektor entsteht.

```
+++ Random Numbers H1 and H2 +++
Values for H1:
 950441840 -2039127258 137283973 -1662391496
Values for H2:
 -1052087407 -1424448050 -696435583 -1834468510
+++ End Random Numbers H1 and H2 +++
```

- Zuletzt wird noch ein detaillierter Output der tatsächlichen Hashtabellen gegeben, welcher hier ebenfalls wieder nur auszugsweise wiedergegeben wird. Er zeigt auf, welcher Vektor bei welcher Hashtabelle in welches Bucket gehashed wurde. Zur besseren Übersicht bei einer sehr großen Anzahl an Vektoren wird außerdem noch die Anzahl der Elemente und der Hashwert $h_1$ beim entsprechenden Bucket angeführt.

```
+++ Hashtable Output +++
 Values of Hashtable 0:
 Bucket: -4 with 1 elements
  (-24.924362 -9.2995615 -3.188014 )
 Bucket: -5 with 1 elements
```

```
  (-6.680074 -20.649195 -14.538515 )
 Bucket: -6 with 1 elements
  (15.588769 15.406525 -22.82398 )
 Bucket: 7 with 1 elements
  (-16.274424 -5.0794115 6.86293 )
 Bucket: 6 with 1 elements
  (9.02572 2.7674289 8.56702 )
...

 Values of Hashtable 1:
 Bucket: 8 with 2 elements
  (-0.9142071 1.6519278 -14.738962 )
  (-6.680074 -20.649195 -14.538515 )
 Bucket: 5 with 1 elements
...
```

- Natürlich werden auch, nach vollendeter Suche, diejenigen Vektoren ausgegeben, die vom Algorithmus gefunden wurden, falls eine Debugstufe eingestellt ist.

```
+++ NN Results Found +++
(6.104716 -21.088932 23.39313 )
(5.8860707 -21.077438 23.60028 )
(6.1047974 -21.181828 23.485943 )
(5.9863033 -20.91604 23.33865 )
+++ END of NN Result List +++
```

Um diesen Debug-Output zu ermöglichen, erhielt jede Klasse eine eigene `toString()` Methode, in der die wesentlichen Daten so übersichtlich wie möglich aufbereitet werden.
Ist der Debug-Modus auf der geringst möglichen Stufe, also eins, so wird lediglich eine kurze Information darüber ausgeben, welche Methode der Algorithmus ausführt und ein grober Überblick über die berechneten Parameter unabhängig davon, ob die statistischen Methoden aufgerufen wurden oder nicht. Auch die Klasse `TestData` kann im Debug-Modus ausgeführt werden, jedoch nicht in unterschiedlichem Detaillierungsgrad, wodurch dennoch ein detaillierter Output über alle in dieser Klasse erstellten Vektoren möglich ist. Mehr dazu im Kapitel 2.3.

## 2.3 Validierung und Performance -Analyse

Um die Datenstruktur und den implementierten LSH - Algorithmus ausgiebig zu testen, wurde eine eigene Testklasse, `TestData` (siehe Abbildung 2.1) erstellt. Diese Klasse beinhaltet einen Zufallsgenerator für Vektoren und die Implementierung einer einfachen linearen Suche, sodass ohne Zweifel alle tatsächlichen nächsten Nachbarn innerhalb eines Radius ermittelt werden können. Zur Verdeutlichung soll hier noch in wenigen Worten die Funktionsweise des "Bruteforce" Algorithmus beschrieben werden. Wie in der Methode
`findClosestBruteForce(float[] vector)` ersichtlich, wird aus einer Hashtabelle, die aus dem Hashtabellen Array der Klasse `LSH` stammt, iterativ zuerst jedes Bucket ausgelesen und aus der resultierenden Arrayliste jeder einzelne Vektor. Von diesen wird dann die Distanz zum Suchpunkt berechnet und entsprechend wenn die Distanz kleiner als $cR$ ist, der Vektor in die "Bruteforce" Ergebnismenge aufgenommen. Wie schon bei der `initLSHValues` Methode der Klasse LSH muss auch die Testklasse mit gewissen Parametern, die gleich mit dem Konstruktor übergeben werden, ausgestattet werden, damit der Bruteforce dieselbe Ausgangsbasis besitzt wie die NN-Suche.
`TestData(int dimension,int distance,int border,Hashtable table,boolean debug)`
Hier wird schon im Vorfeld die Dimension festgelegt, die in Folge für die Erzeugung von Zufallsvektoren herangezogen wird, und der künftige Radius für beide Suchen. Der Parameter Border dient, wie in [1] vorgeschlagen, als Grenzwert der bei der Erstellung von Zufallszahlen für die Vektoren weder im positiven noch im negativen Bereich überschritten werden soll. D.h. die Zufallszahl soll in einem Intervall von $[-Border, Border]$ liegen. Die Hashtabelle dient als Referenz einer Tabelle der LSH Datenstruktur. Daher besitzt auch die Klasse `LSH` eine Funktion, die eine Referenz auf die ganze Datenstruktur, bzw. eine Referenz auf das Array, welches die *L* Hashtabellen verwaltet, zurück gibt. Somit hat die Testklasse immer Zugriff auf alle Vektoren, da bereits in einer Hashtabelle Referenzen auf alle Vektoren enthalten sind.
Da die Erstellung der Zufallsvektoren nicht ganz dem Zufall überlassen werden kann, können kontrolliert Zufallsvektoren erstellt werden, die außerhalb eines bestimmten Radius von einem Suchpunkt, der entweder "manuell" oder auch durch einen Zufallsgenerator erstellt werden kann, liegen, und Zufallsvektoren, die sich innerhalb befinden. Hierfür stehen die beiden Methoden

`createOuterTestData(float[] queryPoint)` und `createInnerTestData(float[] queryPoint)` zur Verfügung. Die erste erstellt, wie der Name schon so trefflich beschreibt, Vektoren, die sich außerhalb eines bestimmten Radius befinden. Hierfür wird die Methode `createVector()` zur Hilfe genommen, welche einen willkürlichen Floatvektor erstellt, unabhängig von Suchpunkten und Radien. In der Methode `createOuterTestData()` werden Vektoren, die sich außerhalb des entsprechenden Radius vom Suchpunkt befinden, generiert. Allerdings kann die Methode nicht einfach auf Basis von `createVector()` realisiert werden, da die Wahrscheinlichkeit, dass ein Vektor innerhalb eines Radius zufällig erstellt wird, bei Dimensionen größer als 5, schon sehr gering ist. Hierfür musste ein anderes Verfahren gefunden werden.
Zuerst wird, in der Methode `createInnerTestData` wie bisher mithilfe von `createVector()` ein zufälliger Vektor erstellt und gleichzeitig die Summe über alle seine Einträge berechnet. Jeder Eintrag des neu erstellten Vektors wird durch die zuvor berechnete Summe dividiert wodurch wieder ein neuer Vektor entsteht, der kleiner als ein Einheitsvektor ist. Dieser Vektor wird zu dem Suchpunkt addiert, was einen zufälligen Vektor innerhalb des Radius 1 zur Folge hat.
Weiters besitzt die Klasse `TestData` eine Methode, die mittels Bruteforce die Nachbarn innerhalb des Radius $cR$ sucht und zusätzlich eine Methode, die die tatsächlichen Ergebnismengen mit anderen Suchergebnissen, wie z. B. der Ergebnismenge der cR-NN Suche, vergleicht und entsprechende Vektoren ausgibt, die vom tatsächlichen Ergebnis abweichen. Dies wären zum Beispiel Vektoren die von cR-NN irrtümlich gefunden wurden und somit ein falsches Suchergebnis bilden.

### 2.3.1 Ergebnisse

Um die Qualität der Ergebnisse, die mit Hilfe der in Kapitel 2.3 vorgestellten Testdaten von der cR-NN Suche und von Bruteforce gefunden wurden, zu evaluieren, wurden noch einige Methoden erstellt, die statistische Werte der jeweiligen Suchfunktion erheben.
Für jede der beiden Klassen existieren eigene Variablen, die die statistischen Werte mitprotokollieren, die dann mittels einer eigene Methode aufbereitet werden, und wenn erwünscht, angezeigt werden können. Somit ist der statistische Output unabhängig vom Debug-Modus und kann individuell angezeigt werden. Ein typischer statistischer Output sieht wie folgt aus.

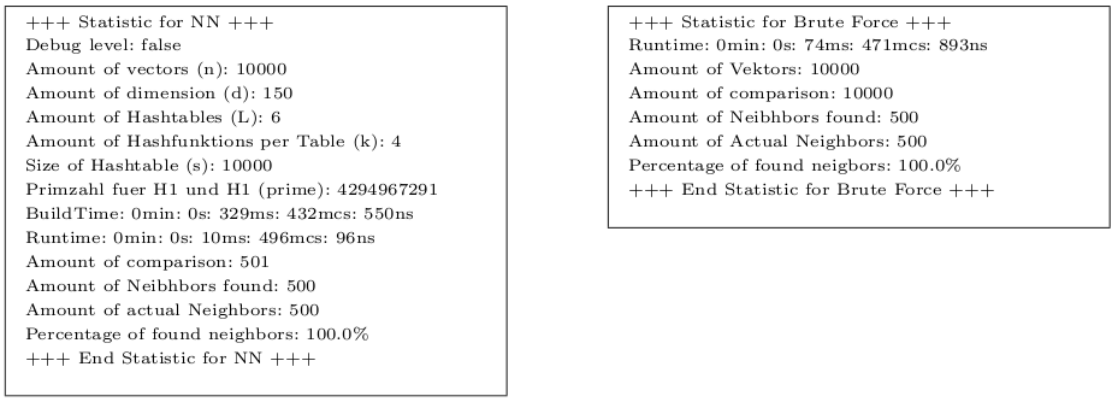


Abbildung 2.2: Statistische Output für cR-NN und Lineare Suche

Der Output (siehe Kapitel 2.3.1) der LSH-Klasse fällt dabei üppiger aus, da zusätzlich die Parameter, die auch im Debug-Modus ausgegeben werden, angezeigt werden. Somit ist ein Gesamtüberblick möglich. Doch die wichtigsten Werte sind die Laufzeiten beider Algorithmen. Die Laufzeit der LSH-Klasse wird gemessen ab dem Zeitpunkt, an dem die Suche begonnen wird, bis zu dem Zeitpunkt, wenn alle *L* Hashtabellen nach den entsprechenden Nachbarn durchsucht wurden. Hierbei sei angemerkt, dass die Suche nicht wie in Kapitel 1.3.2 beschrieben nach gefundenen *3L* Nachbarn abbricht, sondern die ganze Datenstruktur durchsucht. Die jeweiligen Zeitpunkte werden mit `System.nanoTime()` zwischengespeichert. Aus der Differenz zwischen der erhaltenen Endzeit und der Startzeit wird die tatsächliche Zeitdauer berechnet. Zum besseren Verständnis sei hier angeführt, dass die Abkürzung *mcs* bei der Zeitdarstellung für $\mu$s steht. Bei der *BuildTime* handelt es sich um die Zeit, die der Algorithmus benötigt, alle entsprechenden Parameter zu berechnen und die einzelnen Vektoren in die Datenstruktur einzufügen. Die Zeit wird bei jedem Aufruf der `add()` aufaddiert und am Ende, wenn die Methode für den statistischen Output aufgerufen wird, berechnet. Diese *BuildTime* wird jedoch nur bei NN ausgegeben, da der Bruteforce nur die Vektoren von der Datenstruktur der LSH Klasse übernimmt.
Ein weiterer wichtiger statistischer Wert ist die Anzahl der Vergleiche, d.h.. wie oft die Distanz zwischen zwei Vektoren berechnet werden musste, um festzustellen, ob sich dieser innerhalb eines Radius befindet. Wie schon bei der Laufzeit fällt die Anzahl der Vergleiche bei LSH wesentlich geringer aus (vgl. Kapitel 2.3.1), da nur diejenigen Elemente verglichen werden, die in das selbe Bucket gehashed wurden (siehe Kapitel 2.1). Weiters wird noch angegeben, wie viele der tatsächlichen Nachbarn gefunden wurden, die ja bekannt sind, da wie schon in Kapitel 2.3 erwähnt, eine eigene Methode existiert, die Zufallsvektoren innerhalb eines Radius berechnet, und die Anzahl der erstellten

Vektoren zählt. Dadurch ist es auch möglich ein prozentuelles Maß anzugeben wie viele der tatsächlichen Nachbarn gefunden wurden. Dies muss bei der LSH-Klasse nicht immer zwingend 100% sein wie bei der linearen Suche, sondern kann durchaus abweichen, da es sich um ”approximate nearest neighbors” handelt [1]. Unterscheidet sich das Ergebnis von den tatsächlichen Nachbarn, kann mittels der Methode `compare(Map nnResult, ArrayList<float[]> resultsBF)` analysiert werden, welche Vektoren nicht gefunden wurden.

```
99 Elements found by both, 1 elements not found by NN
and 0 were found by NN but not BruteForce

+++ Results not found by NN +++
(-5.404906 11.255481 13.86927 -4.377128 17.69853 24.53709 1.5832355 -5.492511
3.3361716 9.059968)
+++ End Results not found by NN +++
```

Um den Unterschied zwischen der Laufzeit der LSH-Implementierung und des Bruteforce Algorithmus zu verdeutlichen, wurde ein Diagramm erstellt (siehe Abbildung 2.3 und 2.4), welches die entsprechenden Suchlaufzeit beider Algorithmen darstellt.
Bei noch sehr wenigen Vektoren kann es durchaus vorkommen, dass Bruteforce schnel-

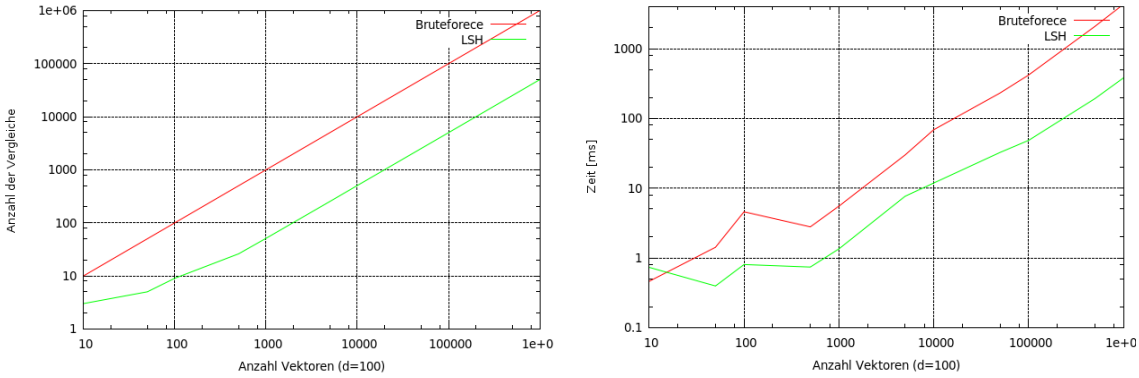


Abbildung 2.3: Darstellung der Suchzeit und Anzahl der Vektorvergleiche

ler die gewünschten Ergebnisse findet als LSH, jedoch bei einer größeren Menge ($n > 100$) ist die Suchzeit von LSH erheblich kürzer. Bei einer Anzahl von $10^6$ Vektoren ist der LSH Algorithmus beinahe zehn mal schneller als Bruteforce. Auf die Anzahl der Vergleiche hat die Dimension keine Auswirkung, jedoch sehr wohl auf die Laufzeit, die

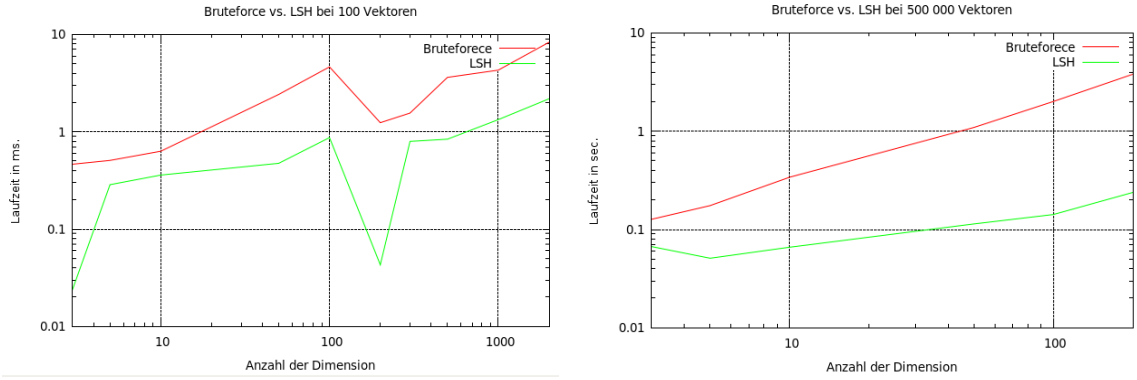


Abbildung 2.4: Darstellung der Suchzeit mit unterschiedlichen Dimensionen

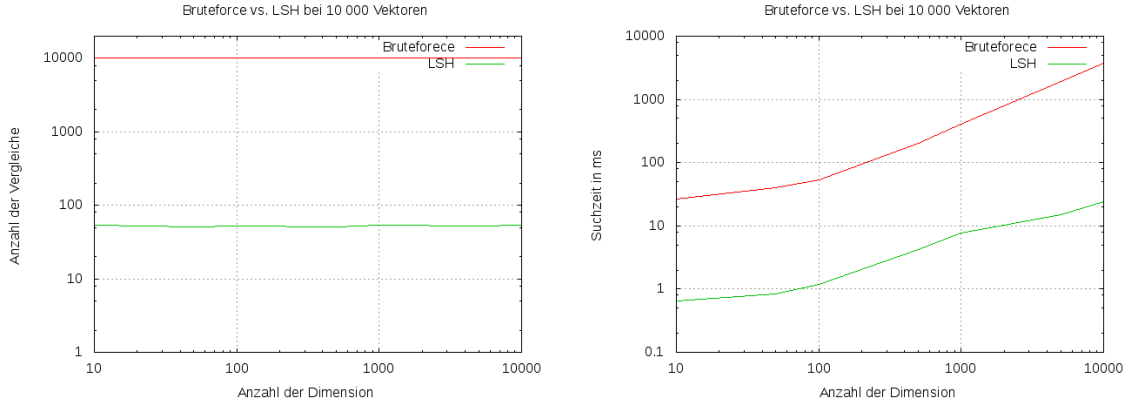


Abbildung 2.5: Anzahl der Vergleich und Laufzeit in Abhängigkeit der Dimension

beinahe stetig ansteigt. Ersichtlich ist dies in den Grafiken der Abbildung 2.5 bei denen eine statische Anzahl an Vektoren als Universum mit ebenfalls statischer Anzahl an wissentlicher ”cR-NN” angenommen wurde.

Zur Vollständigkeit soll hier noch erwähnt werden, dass, um $10^6$ Vektoren zu erstellen, der Java Virtual Machine mehr Speicher zugeschrieben werden muss, was mittels $-Xms128m - Xmx2024m$ möglich ist.

# 3 Zusammenfassung und Schlussfolgerung

In dieser Arbeit wurde ein Überblick über die Technik des ”Locality Sensitive Hashing” mit Fokus auf LSH mittels p-stable Distributions gegeben. Es wurden auch andere Hashfamilien $\mathcal{H}$ vorgestellt, die alternativ angewandt werden können. Weiters wurden bereits etablierte alternative Möglichkeiten vorgestellt nieder-dimensionale Datensätze zu durchsuchen. Der Algorithmus ist mit nicht allzu viel Aufwand zu implementieren und kann für jede $l_p$ Norm für $p \in (0, 2]$ verwendet werden [4].
Es wurde ein Überblick gegeben wie eine mögliche Implementierung in Java aussehen kann, die allerdings sicherlich noch weiter optimiert werden könnte. So könnten z.B. noch alle Optimierungen wie in Kapitel 1.4.6 beschrieben eingebaut werden.
Es wurde der implementierte LSH Algorithmus einer Bruteforce Implementierung, bezogen auf die Suchzeit und der Menge der Vektoren, die verglichen werden müssen, gegenüber gestellt, abhängig von der Anzahl der Vektoren.
Abschließend wurde der implementierte LSH-Algorithmus mit einer linearen Suche im Bezug auf Laufzeit in Abhängigkeit der Dimension verglichen (siehe Abbildung (2.4). Dabei stellte sich heraus, dass LSH-Implementierung verlässlich ist, in dem Sinne, dass mit der erwarteten Wahrscheinlichkeit auch die tatsächlichen NN gefunden wurden. Weiters konnte festgestellt werden, dass LSH sowohl was die Laufzeit, als auch was die Anzahl der Vergleiche betrifft, etwa 10 mal schneller ist als die lineare Suche. Überraschend war jedoch, dass der Geschwindigkeitsvorteil auch für sehr viele Vektoren etwa konstant bei einem Speedup Faktor von 10 liegt.

# 4 Literaturverzeichnis


[1] Alexandr Andoni, Piotr Indyk $E^2$*LSH 0.1 User Manual*, June 21, 2005.

[2] Mayur Datar,Nicole Immorlica, Piotr Indyk, Vahab S. Mirrokni, *Locality-Sensitive Hashing Scheme Based on p-Stable Distributions*, Brooklyn New York USA 2004.

[3] Malcolm Slaney, Michael Casey, *Locality-Sensitive Hahsing for Finding Nearest Neighbor*, IEEE Signal Processing Magazine, March 2008.

[4] Alexandr Andoni, Piotr Indyk *Near-Optimal Hashing Algorithm for Approximate Nearest Neighbor in High Dimensions*, Proceedings of the Symposium on Foundations of Computer Science (FOCS'06).

[5] Andrew W. Moore, *An introductory tutorial on kd-trees*, Carnegie Mellon University

[6] Jon Louise Bentley, *Multidimensional Binary Search Trees Used for Associative Searching*, Stanford University, 1975 ACM Student Award.

[7] Aristides Gionis, Piotr Indyky, Rajeev Motwaniz *Similarity Search in High Dimensions via Hashing*, Department of Computer Science Stanford University, 1999.

[8] Volker Gaede, Oliver Günther, *Multidimensional Access Methods*, Humboldt-Universität Berlin, Institut für Wirtschaftsinformatik, Germany

[9] Deepak Ravichandran, Patrick Pantel, and Eduard Hovy *Randomized Algorithms and NLP: Using Locality Sensitive Hash Function for High Speed Noun Clustering*, Information Sciences Institute University of Southern California, June 2005

[10] Alexandr Andoni, Piotr Indyk , *Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions*, Stanford MMDS'06

[11] Piotr Indyk, Rajeev Motwani, *Approximate Nearst Neighbors: Towards Removing the Curse of Dimensionality* Department of Computer Science, Stanfort University

[12] Mayank Bawa, Tyson Condie, Prasanna Ganesan, *LSH Forest: Self-Tuning Indexes for Similarity Search*, IW3C2

[13] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, Kai Li, *Mult-probe LSH: Efficient Indexing for High Dimensional Similarity Search* Department of Computer Science , Princeton University.

[14] Gregory Shakhnarovich, Trevor Darrell, Piotr Indyk, *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*

[15] Alexander Hinneburg *9 Vorlesung Lineare Algebra, SVD und LSI* Marthin-Luther Universität Halle/Wittenbwerg.

[16] Markus Apell *The Grid File - An Adaptable, Symmetric Multikey File Structure*

[17] Pavel Zezula, Guiseppe Amato, Vlastislav Dohnal, Michal Batko *Similarity Search, The Metric Space Approach*

[18] Aristides Gionis, Piotr Indyky, Rajeev Motwaniz *Similarity Search in High Dimensions via Hashing (Presentation)*

[19] Rajeev Motwani *Supplemental Min-wise Hashing Slides*

[20] Piotr Indyk *A small approximately min-wise independent family of hash functions* Stanford University 1998