Active Vertice Clusters – A Sophisticated Concave Cluster Shape Approach for Efficient High Dimensional Nearest Neighbor Retrieval

Sören Balko   Ingo Schmitt

Otto-von-Guericke-Universität Magdeburg
Institut für Technische und Betriebliche Informationssysteme
Postfach 4120, D–39016 Magdeburg, Germany
{balko|schmitt}@iti.cs.uni-magdeburg.de

December 2001

Abstract

In this paper, we advance the idea of concave cluster geometries introduced in [Sch01, BS01] for efficient high-dimensional nearest neighbor retrieval. We formulate substantial requirements for efficient cluster geometries and adapt the well known simplex model for data point alignments to these requirements.

Subsequently we propose the *Active Vertice* cluster as a preliminary milestone with respect to concave cluster geometries. An efficient clustering algorithm is introduced along with a analytic foundation of the utilized cluster parameters. Retrieval performance comparisons and a well-founded outlook on necessary advances of this approach conclude this paper.

Contents

List of Figures

Chapter 1

Introduction

In this introductive chapter we describe the problem of high-dimensional clustering and nearest neighbor retrieval. We present some assumptions and restrictions of our approach. The motivation for our method is gained by a retrospective review on previous work. Moreover, we present vital requirements for high-dimensional cluster geometries.

1.1 Foundations

Nearest neighbor retrieval of high-dimensional data points is required for a broad spectrum of applications. In particular, similarity search in multimedia databases benefits from efficient nearest neighbor retrieval methodologies. The most commonly used approach is a mapping of certain characteristics of the data objects to feature vectors. Without losing generality we assume these feature vectors to be scaled to an interval of $[0, 1]$ in each dimension. Therefore, we obtain data points p_i with:

$$p_i \in [0, \ldots, 1]^d$$

Most application scenarios have two major properties. Firstly, the number N of data objects is very large. For example, the number of frames in a video database "Hollywood Movies" would obviously be very large. Secondly, the dimensionality d (i.e. the *size*) of the feature vectors is typically quite high. Actually this dimensionality depends on the feature extraction method. For example, color histograms may easily exceed 1000 features whereas biometrical data like fingerprints or face pattern would need less features.

Similarity between two feature vectors (data points) $p, q \in P$ is expressed using distance functions. Please note, that P denotes the overall set of data points. Typically used distance functions are defined on L_k norms. In the scope of this paper, we use the Euclidean norm L_2:

$$L_2(p, q) = ||p - q||_2 = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}$$

The most similar object with a feature vector $p \in P$ (i.e. the *nearest neighbor*, NN) to a query object with a feature vector q is defined as follows:

$$\mathrm{NN}(q) = p \in P \text{ with } \forall p_i \in P : ||q - p||_2 \leq ||q - p_i||_2$$

Clearly there must not be another data point in P which is "closer" to q than p.

Within this paper, we restrict our considerations to uniformly distributed data points. In contrast to "real data" this approach permits us to appropriately compare different retrieval methods. Furthermore, in real data feature vectors we can detect and remove correlations between feature values in different dimensions. This transformation can be done by applying several dimension reduction methods like PCA [FK70, MKB79], MDS [KW78], and FastMap [FL95].

A naive nearest neighbor retrieval method is a sequential scan of all data points. Each of these points has to be retrieved from the database. We measure the retrieval costs as the amount of data which has to be fetched from the database. The sequential scan would cause linear costs corresponding to the number N of data points:

$$\mathrm{cost}_{\text{sequential scan}} = N \cdot d \cdot \mathrm{sizeof}(\mathsf{FLOAT})$$

Please note, that we assume feature values to be expressed using real numbers of type FLOAT[1]. Unfortunately existing indexing and retrieval methods (see [BBK01] for an overview) fail to perform better than sequential scan for high dimensions.

Most of these retrieval methods (e.g. the *R-tree* [Gut84], the *X-tree* [BKK96], or the *TV-tree* [LJF94]) are based on the assembly of a hierarchy of clusters. That is, a tree-like structure is utilized to store clusters of a certain geometrical shape.

In this paper we use the term *cluster* as (i) a geometrical description of a certain region in the vector space and (ii) as a "container" for a number of data points which are located in this region. Therefore, our comprehension of cluster differs from a semantics centric point of view where somewhat similar data objects are compound into one cluster. In contrast, the shape of a cluster which is derived from its geometrical description is the major characteristic in our cluster definition.

In this context, we mention the term *cluster efficiency* as it was introduced in [BS01]. A cluster is called "efficient", whenever the storage size of its comprised data points exceeds the size of the geometrical description.

In Figure 1.1, an adapted *Branch-and-Bound* algorithm for nearest neighbor retrieval is depicted. This algorithm differs from the original Branch-and-Bound algorithm [NG88] in several ways.

First of all, it does not recursively traverse a tree like graph which hierarchically contains clusters. We restrict our considerations to the *cluster geometry* which is the basis for every hierarchical access structure. Furthermore, we follow the argumentation from [BKK96] which states that index trees should *flatten* with rising dimensionality. For the time being, our algorithm works on a sequential list of clusters c_i.

[1] 4 byte single precision reals (IEEE 754).

Secondly, we included the concept of cluster efficiency. That is, the algorithm computes the minimal distance LB from a query point q the the cluster shape only for efficient clusters. Whenever the distance LB lies below the current minimal distance UpperBound, the data points inside this clusters are inspected. The variable NearestNeighbor contains the nearest neighbor to the query point found so far.

Finally, we put one heuristics in place. That is, we order the efficient clusters ascendingly by their distance LB to the query point q. This heuristics bases on the expectation, that "closer" clusters are more likely to contain nearer data points. As a result, the variable UpperBound shrinks quickly and less clusters have to be inspected. To ensure that we do not overlook a potential nearest neighbor candidate, UpperBound has to be initialized with $\sqrt{d}$. This distance is the diameter of the vector space and the largest possible distance by definition:

$$\|(\underbrace{0,\ldots,0}_{d}) - (\underbrace{1,\ldots,1}_{d})\|_2 = \sqrt{d}$$

```
procedure BranchAndBound(point q)
    initialization: UpperBound = √d
    heuristics: order efficient clusters by LB(q)
    for each cluster c_i do
        if c_i is efficient then
            begin
            compute lower bound distance LB(q)
            if LB(q) ≤ UpperBound then
                for each point p_j ∈ c_i do
                    begin
                    distance = ||p_j − q||_2
                    if distance ≤ UpperBound then
                        begin
                        NearestNeighbor = p_j
                        UpperBound = distance
                        end
                    end
            end
```

Figure 1.1: Adapted *Branch-and-Bound* Algorithm for Nearest Neighbor Retrieval

1.2 Requirements for Cluster Geometries

In [Sch01] we introduced a statistical analysis of point distances. Major results of this paper are the expecting value $E(X)_{\|p_i-p_j\|_2}$ and the standard deviation $\sigma_{\|p_i-p_j\|_2}$ for L_2

distances[2] of uniformly distributed points in d dimensional (vector) space:

$$\begin{aligned} E(X) &= \sqrt{\frac{d}{6}} \\ \sigma &= \text{const} \approx 0.24 \end{aligned}$$

This means, that average distances become larger for rising dimensionalities. However, their standard deviation remains constant.

All (cluster based) established retrieval methodologies like *minimal bounding rectangle* (e.g. [Gut84, BKSS90]) or *hyperball* (e.g. [WJ96, KS97]) based approaches are based on *convex* cluster geometries. Convex cluster geometries span a certain region in the vector space which typically comprises a non-zero volume. Depending on their geometrical shape they have an edge length and/or a diameter which specifies this region.

In general, a cluster is meant to precisely approximate the location of a set of data points in the vector space. Therefore, the difference (approximation error) between the distance of an arbitrarily located query point q to its nearest (virtual) point in the cluster geometry *and* the distance to its actual nearest neighbor data point inside the cluster should be as small as possible. However, as point distances grow, it does not make sense to enclose them in convex clusters any longer. This is due to the fact, that large "empty" regions are enclosed by the cluster which causes the approximation error to grow.

A further consequence of this statistical observation, is an increasing, *relative* similarity of distances between uniformly distributed data points:

$$p_1, p_2, q_1, q_2 \in [0, \ldots, 1]^d : \lim_{d \to \infty} \frac{\|p_1 - q_1\|_2}{\|p_2 - q_2\|_2} = 1$$

However, the only geometrical alignment of equally distant points is as vertices of a *regular simplex*. In [BS01] we introduced the concept of regular simplexes as an appropriate model for high-dimensional data point alignments.

In $\mathbb{R}^1$ a regular simplex is simply a line connecting two points, in $\mathbb{R}^2$ it is a triangle with equally long edges, and in $\mathbb{R}^3$ it is a regular tetrahedron. In higher dimensional space, these regular shapes are called regular hyper-polyhedra. Furthermore, we derived a formula to compute the radius ρ of the hyperball inscribed into the vertices of an n-simplex with an edge length l as:

$$\rho = l\sqrt{\frac{n-1}{2n}}$$

A complete simplex has $n = d + 1$ vertices in d-dimensional space. Therefore, its radius can be computed as $\rho = \sqrt{d/(2d+2)}$.

With respect to appropriate high-dimensional cluster geometries, there are two consequences. The *ideal* cluster geometry for high-dimensional points should

[2]Subsequently abbreviated with $E(X)$ and σ.

- locate the comprised data points as precisely as possible, and
- follow the shape of a regular simplex.

Since it is not possible to efficiently enough locate the data points precisely, the cluster should comprise small separated regions which approximate the alignment of a data point.

Obviously, a convex cluster geometry does not locate small separated areas which contain single data points but rather encloses large areas in the vector space. Therefore, it is essentially not suited for high-dimensional data points. Moreover, an appropriate cluster geometry is mandatorily concave. In [BS01] we proposed the intersection of two hyperballs with an enclosing box: the (totally) concave R-Shell cluster (see Figure 1.2). The concave R-Shell cluster unveiled superior retrieval performance in comparison to the

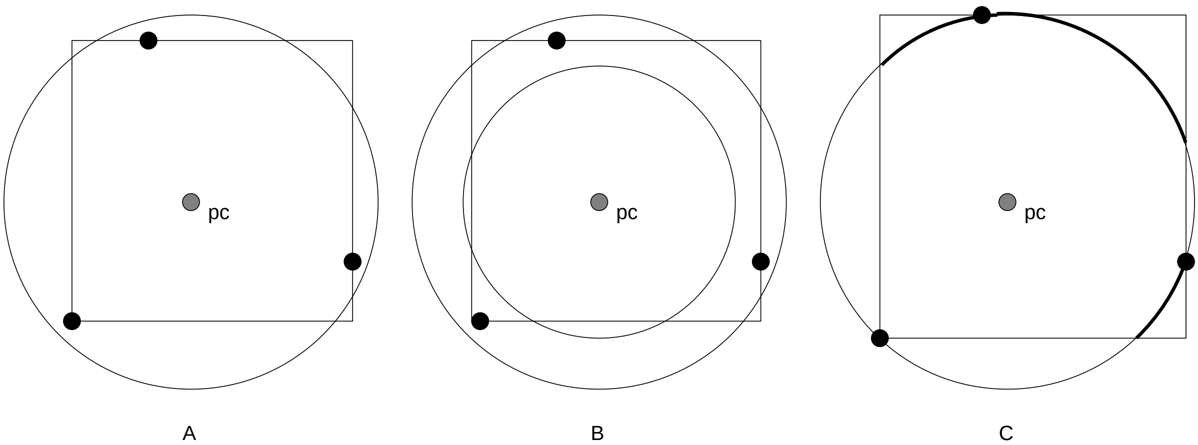


Figure 1.2: Convex (*A*), Concave (*B*) and Totally Concave (*C*) R-Shell Cluster Geometry

convex R-Shell cluster in higher dimensional space. However, retrieval performance still quickly deteriorates with rising dimensionality.

This is caused by some drawbacks of the concave R-Shell cluster. Firstly, the emerging concave regions in the R-Shell cluster insufficiently adapt to regular simplex shapes. Secondly, many empty regions can arise from an intersection of hyperballs and an enclosing box. That is, most separated regions do not contain any data points which, in turn, deteriorates the approximation error.

Therefore, a cluster geometry which follows the stated requirements more closely, is expected to yield better retrieval performance. In this paper, we introduce a cluster geometry that strictly obeys these requirements – the *Active Vertice Cluster*.

Chapter 2

The *Active Vertice* Cluster

In this chapter, we introduce the *Active Vertice* cluster geometry proposal. Besides a detailed geometrical description, we show its commitment to the ideal cluster requirements. Subsequently, we present a suitable clustering algorithm for this cluster shape and briefly explain the computation of the LB parameter in the Branch-and-Bound algorithm.

2.1 Geometrical Specification

In the light of the simplex alignment requirement, we would want to geometrically specify the simplex itself and exclude certain regions to make it concave. Unfortunately, regular simplexes may be arbitrarily aligned in the vector space. In consequence, a geometrical description is mandatorily very expensive. For a naive implementation which stores each vertice of that simplex as a single d-dimensional point we obtain the following cluster retrieval cost:

$$\text{cluster cost (naive)}_{\text{simplex}} = (d+1) \cdot d \cdot \text{sizeof}(\mathsf{FLOAT})$$

Please note, that this cost holds for a complete $(d+1)$-simplex. Actually, there is a cheaper description of a regular simplex. This idea utilizes the information about the fixed edge length:

$$\text{cluster cost (cheaper)}_{\text{simplex}} \approx (1 + \sum_{i=1}^{d} i) \cdot \text{sizeof}(\mathsf{FLOAT})$$

We abstain from a formal description of this method since it turns out it still needs approximately half of the cost required for a naive storage of the simplex. In Figure 2.1 the method is depicted by means of an example in $\mathbb{R}^2$.

One starting point (*here:* p_1) and the edge length L must be explicitely specified. The second point p_2 can be located relative to p_1 using one additional angle α between the two-dimensional hyperplane through p_1 and p_2 and one Cartesian coordinate axis.

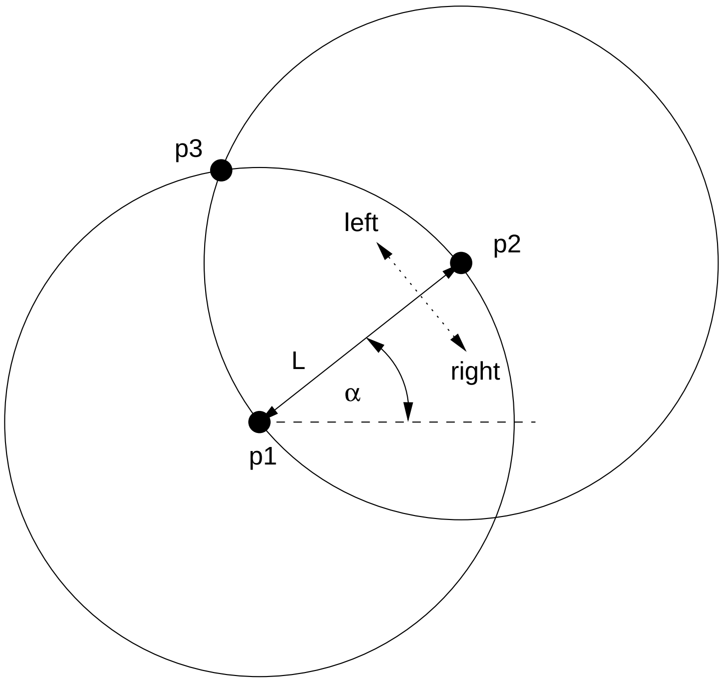


Figure 2.1: Cheap Construction of a Regular Simplex

Finally, there are only two possible locations for a third point p_3 which is equally distant from both p_1 and p_2. Therefore, we need the information on which side (*left* or *right*) of the hyperplane formed by the previous points p_1 and p_2 the last point is located. Overall, we have an amount of 4 FLOAT and one BOOLEAN values to precisely describe this simplex.

This *cheap(er) simplex* method can be generalized to work for higher dimensionalities. Nevertheless, its geometrical retrieval costs are not satisfactory. It takes approximately half of the costs for retrieving the data points sequentially. This ratio does not improve for higher dimensionalities. Whereas the first vertices has to be explicitely stored, each additional vertice incrementally saves the data volume of one FLOAT value. That is, this method pays off for *complete* simplexes (i.e. having $d + 1$ vertices). However, detecting complete simplexes turns out to be a complicated task of pattern recognition.

Since arbitrarily aligned regular simplexes turn out to have unacceptable storage costs we propose to use (incomplete) *normalized* regular simplexes. In [BS01] we proposed the normalized simplex model to approximate point alignments and radii of inscribed hyperballs. That is, the edges of that simplex are fixed with respect to their orientations. An incomplete (d vertices) normalized regular simplex is composed from scaled unit vectors as follows:

$$\begin{aligned} v_1 &= (x, 0, 0, \ldots, 0) \\ v_2 &= (0, x, 0, \ldots, 0) \\ &\ldots \\ v_d &= (0, 0, \ldots, 0, x) \end{aligned}$$

A vertice v_i of the simplex has a value of x in its i-th coordinate and zero in the remaining coordinates[1]:

$$v_1 = x \cdot \mathfrak{e}_i$$

Obviously, the value of x is computed from an edge length L of the simplex as follows:

$$x = \frac{L}{\sqrt{2}}$$

Since we obtain d vertices $v_1, \ldots, v_d$, the emerging d-simplex is incomplete. That is, since there is no $(d+1)$-th unit vector in $\mathbb{R}^d$ the last (i.e. $(d+1)$-th) vertice of a *complete* regular simplex can not be expressed that simple:

$$v_{d+1} = (y, \ldots, y)$$

The value of y can be derived from the following equations:

$$\begin{aligned} \|v_1 - v_2\|_2 &= \|v_1 - v_{d+1}\|_2 \\ x \cdot \sqrt{2} &= \sqrt{(y-x)^2 + (d-1)y^2} \end{aligned}$$

For the sake of simplicity, we are restricted to normalized regular d-simplexes and leave their $(d+1)$-th vertice out of consideration.

In Figure 2.2 an (incomplete) normalized regular 3-simplex with its vertices v_1, v_2, and v_3 is depicted in $\mathbb{R}^3$. The identifier L labels the edge length of the simplex. In this context, we can make some important observations:

- Obviously, each vertice v_i has the same distance of x to the origin $\mathfrak{o}$[2].
- Any two disjoint vertices $v_i \neq v_j$ are orthogonal to each other: $\langle v_i, v_j \rangle = 0$

In turn, this means we can regard the vertices of a simplex as selected corner points of a d-dimensional hypercube with an edge length of x. Precisely, *any* corner point of a hypercube may act as the origin of a regular d simplex.

Therefore, we propose the geometry of a hypercube as an appropriate approximation for the vertices of a (normalized) regular simplex. Precisely, only the *corner points* of a hypercube are part of the cluster geometry. Unless stated otherwise, subsequently we relate the terms *vertice* and *corner point* to hypercubes and use them in equivalence.

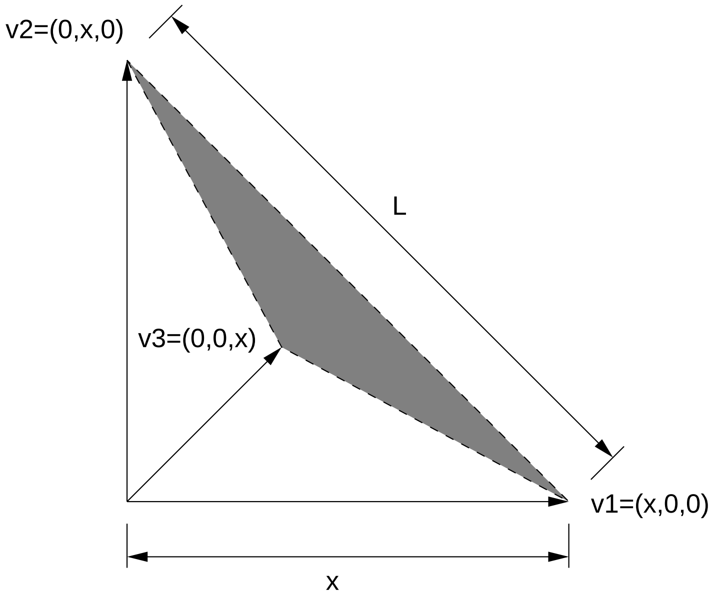


Figure 2.2: Normalized Regular Simplex in $\mathbb{R}^3$

In Figure 2.3 two exemplary corner points c_1, c_2 are selected as *origins* of the two depicted simplexes. Obviously, any corner point may be the origin of a simplex and therefore, any corner point may be part of a regular simplex. In consequence, we must face the possibility that any of the cube's corner points may be the vertice of a simplex formed by selected data points. Unfortunately, a hypercube in $\mathbb{R}^d$ comprises 2^d corner points.

Firstly, one regular simplex inscribed into a hypercube may comprise d vertices at most and, secondly, the overall number of data points is restricted by N. Since the number of corner points (2^d) exceeds both of these numbers we must tag those corner points that really correlate with a data point. In terms of data volume, the utilization of bit codes is a *cheap* mechanism. That is, every corner point is labeled with a bit code which corresponds to its location. To reduce data volume, we store the labels of those corner points which correlate to a data point. These corner points are named as *active vertices*. The remaining vertices are labeled as inactive.

Furthermore, since data points in high dimensions are ideally aligned in regular simplexes, in reality they vary from this location. This variation is reflected by hyperballs around the active vertices. Therefore, we must provide an additional radius for each active vertice. Alternatively, we can set up an upper limit for this radius globally. In Figure 2.4 an active vertice cluster is depicted in $\mathbb{R}^3$. The active vertices in Figure 2.4 are labeled with a grey point. The assigned data points (black dots) are located on hypershells around the active vertices. In this example, we have four active vertices labeled

[1]Unit vector $\mathfrak{e}_i = (\underbrace{0,\ldots,0}_{i-1},1,\underbrace{0,\ldots,0}_{d-1})$

[2]Zero vector $\mathfrak{o} = (0,\ldots,0)$

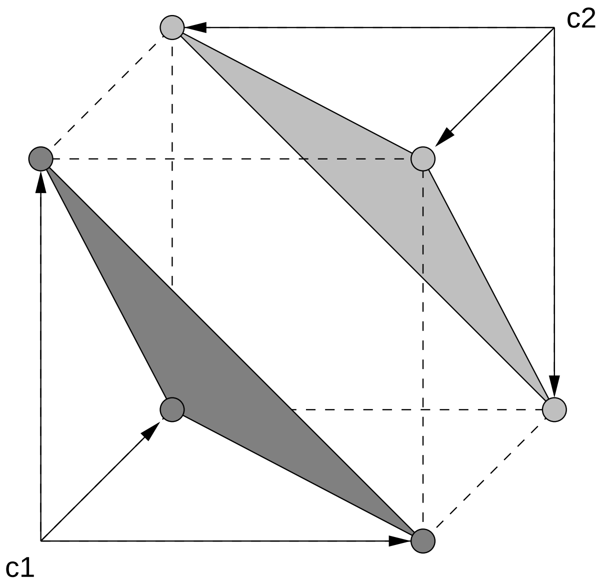


Figure 2.3: Simplexes Inscribed into a Hypercube

with the bit codes 001, 010, 100, and 111. These vertices are located close to four data points $p_1, \ldots, p_4$. Please note, that these active vertices actually form a *complete* regular simplex in $\mathbb{R}^3$. However, this "completeness" is a special case which only holds for this dimensionality.

The remaining vertices have no data points assigned and are, thus, not part of the cluster. However, the origin point (i.e. the vertice with the bit code $0 \ldots 0$) is stored with its "real coordinates". The edge length of the hypercube is labeled with the identifier x. Please note, that x differs from the edge length L of the inscribed regular simplexes.

The bit codes may actually be stored as a sequence of single bits. Be b the bit code (sequence of bits, $b = b_1 \ldots b_d$) assigned to an active vertice, o the origin point and x the edge length of the hypercube. The coordinates of the active vertice v can be computed as follows:

$$v_i = o_i + b_i \cdot x$$

That is, the i-th coordinate of the active vertice i is the sum from the corresponding coordinate of the origin point o and the product of the edge length x and the bit code at this position. In consequence, each active vertice can be determined by (i) the origin point, (ii) the edge length, and (iii) its bit code.

In terms of cluster retrieval costs, an active vertice cluster does not require a fixed amount of data for its storage. Rather, its data volume depends on the number of data points n it comprises. Precisely, in dimensionality d, the cluster retrieval costs (in bytes) are computed as follows:

$$\mathrm{cost}_{\mathrm{cluster}} = \underbrace{(1 + d) \cdot \mathrm{sizeof}(\mathsf{FLOAT})}_{\mathrm{fixed}} + \underbrace{n(\frac{d}{8} + \mathrm{sizeof}(\mathsf{FLOAT}))}_{\mathrm{dynamic}}$$

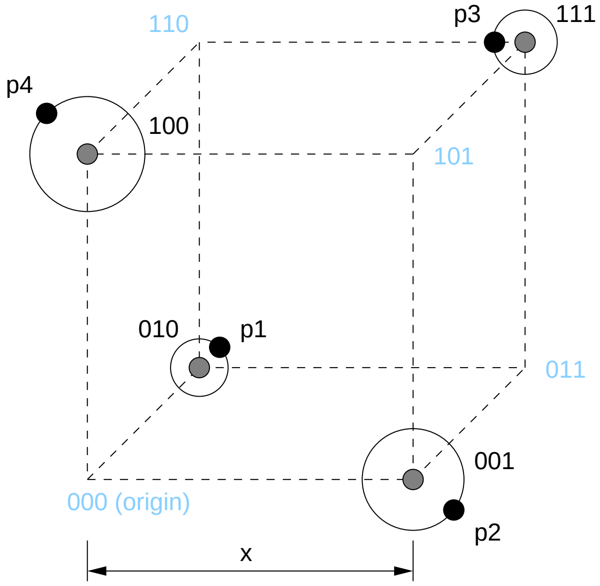


Figure 2.4: Active Vertice Cluster in $\mathbb{R}^3$

The cluster retrieval costs contain a fixed proportion which is the origin point and the edge length. In addition, there is a dynamic component which depends on the number n of active vertices. Each active vertice comprises a bit code of length d (bits) and a radius which determines the distance of the actual data point from the active vertice. Please note, that this radius is optional and may be globally provided for all clusters. Clearly, a sequence of d bits can encode 2^d vertices which corresponds to the number of corner points of a hypercube in $\mathbb{R}^d$. Since one byte comprises 8 bits the size of a bit code is $d/8$ bytes.

2.2 Clustering Algorithm

The *clustering algorithm* is a method to assign data points from a preexisting set P of data points or newly-inserted data points to one of the existing clusters. If necessary, the clustering algorithm has to create new clusters.

There are several requirements with respect to the clustering algorithm. First of all, it must provide an acceptable runtime behavior. Secondly, it must be aimed at an optimization of point assignment to clusters which yields a good approximation (i.e. low approximation error). That is, the retrieval performance should be optimal for the selected cluster geometry.

In terms of Active Vertice clusters, we have some variables which are subject to a clustering algorithm. That is, the origin point o (bit code $0 \ldots 0$) which has to be set up, an edge length x for each hypercube must be fixed, and some threshold value $r_{\max}$ for the maximal distance between one active vertice and the assigned data point must also be determined. This maximal distance forms the radius of a hyperball with the vertice point as its center.

Ideally, these parameters should not be provided from outside but rather from the data point distribution itself. For example, the edge length x can be provided in this way. Besides an universal value of x valid for all clusters, we may also assign separate values for each cluster. We believe this approach is superior because this way, we can smoothly adapt to particular (i.e. non-uniform) data point distributions. We propose two ideas for the determination of x.

The first proposal is based on the concept of simplexes inscribed into hypercubes (see Section 2.1). Therefore, we choose two arbitrary points from our point set P as "seed points", and label their distance with L. We assume these distance as the edge length of the inscribed simplex. Therefore, we obtain a value of

$$x = \frac{L}{\sqrt{2}}$$

In contrast, the second proposal is not based on the simplex model but rather on a simple statistical observation. A hypercube in $\mathbb{R}^d$ comprises 2^d corner points. Based on one arbitrary corner point (without loss of generality the origin point o) these points are located in different distances that range between 0 and $x\sqrt{d}$.

The idea of this approach is to set up the edge length x appropriately that most of the *other* corner points are located in a distance of L to o which is determined by the two seed points. In Figure 2.5 the distances of the corner points relative to the origin point are depicted in $\mathbb{R}^3$. We have one point which is $x\sqrt{0} = 0$ distant, three points which are

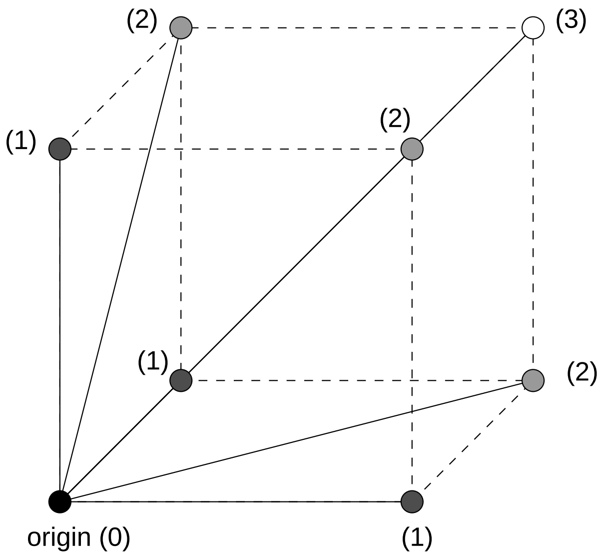


Figure 2.5: Distribution of Distances in $\mathbb{R}^3$

$x\sqrt{1} = x$ distant, another three points which are $x\sqrt{2}$ distant and one point which is

$x\sqrt{3}$ distant from the origin point. In Figure 2.5 equally distant points are labeled with equal colors. A simple combinatorial consideration may be used to compute the number of vertices that lie in a certain distance. The distances between the origin o and and the corner points c_i $(1 \leq i \leq 2^d)$ are classified as follows:

$$\|o - c_i\|_2 = x \cdot \sqrt{k}$$

Obviously there are exactly $k + 1$ distinct distances $k \in \mathbb{N} : 0 \leq k \leq d$.

In a hypercube, one arbitrary corner point c_i may only differ from the origin o as follows. In each dimension j $(0 \leq j \leq d)$ it may either take the same coordinate as o *or* the edge length x added to the coordinate of o:

$$c_{i_j} = \begin{cases} o_i & \text{if } b_i = 0 \\ o_i + x & \text{otherwise} \end{cases}$$

Obviously, the value of k is equivalent the number of dimensions where x is added to the coordinate of o. To compute the number C_k of corner points that lie in a distance of $x\sqrt{k}$ from the origin (or any other corner point) we need to know the number of disjoint combinations of k coordinates among d dimensions.

$$C_k = \binom{d}{k}$$

We can conclude that the maximal number of corner points is yielded for $k = d/2$. This means, relatively most corner points are located in a distance of

$$L = x \cdot \sqrt{\frac{d}{2}}$$

from the origin. We obtain the following edge length:

$$x = \frac{L}{\sqrt{d/2}}$$

The two remaining parameters – origin point o and radius threshold $r_{\max}$ – are determined very straightforwardly. An origin point must be settled by the time the cluster is created. A new cluster is formed, whenever an inserted data point p does not "fit" into any of the existing clusters. That new cluster assimilates this data point, i.e. it assigns it exactly to one of its corner points $c_i = p$. To obtain some variance in the position and extent of clusters, we randomly choose this vertice out of the clusters set of 2^d vertices. Subsequently this chosen corner point is *activated* and the origin point is computed by the parameter x (edge length).

In case, a data point has to be inserted into an existing cluster, the following procedure is conducted. First of all, the relative position of p to a (virtual) center point p_c of the hypercube is determined:

$$p_c = (o_1 + \frac{x}{2}, o_2 + \frac{x}{2}, \ldots, o_d + \frac{x}{2})$$

In this way, we obtain the appropriate corner point c_a for this data point:

$$c_{a_j} = \begin{cases} o_j & \text{if } p_j \leq p_{c_j} \\ o_j + x & \text{if } p_j > p_{c_j} \end{cases}$$

Please note, that the data point may also be located *outside* the convex shape of the hypercube. Next, the distance between the data point p and the assigned corner point c_a is computed. If this distance lies below the maximal radius threshold value $r_{\max}$, we can insert p into this cluster:

$$\|p - c_a\|_2 \leq r_{\max}$$

Subsequently, the corner point must be made active, i.e. its bit code is stored into the database. In Figure 2.6 the insertion of a data point p into a cluster in $\mathbb{R}^2$ is depicted. We sketched the point insertion algorithm as pseudo code in Figure 2.7.

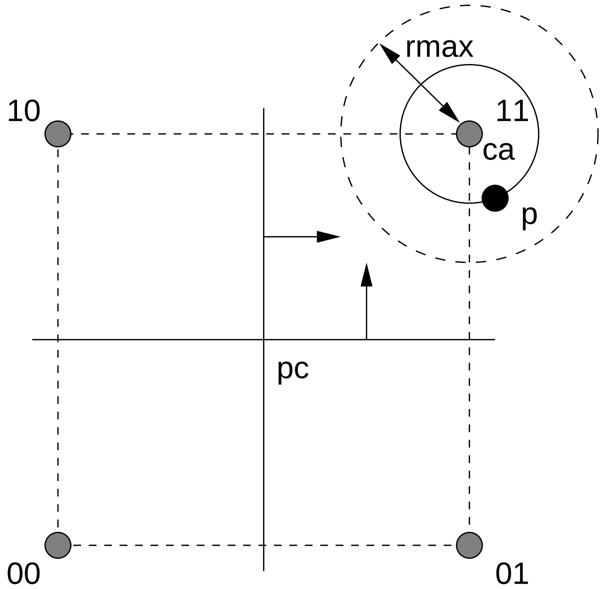


Figure 2.6: Insertion of a Data Point in $\mathbb{R}^2$

As the last parameter the maximal radius threshold $r_{\max}$ remains to be specified. We expect this parameter depend on both the dimensionality d and the overall number of data points N. On the one hand, $r_{\max}$ directly influences the approximation error of this cluster geometry approach. Therefore, it should be as small as possible. On the other hand, there should be few emerging clusters to reduce the cluster retrieval cost. To obtain a small number of clusters, this parameter should be set up more generously. Since this cluster geometry handles active vertices as small hyperballs, no pair of these hyperballs should overlap. Unfortunately, besides our combinatorial estimation of corner point occupations, we can hardly predict the actual average distance of two active vertices in advance. However, this would result in a maximal radius threshold of:

$$r_{\max} \leq \frac{x}{2} \cdot \sqrt{\frac{d}{2}}$$

```
procedure insertpoint(point p ∈ P)
  for each cluster cl do
    begin
    compute the appropriate vertice c_a for p
    if ||p − c_a||_2 ≤ r_max then
      begin
      store the bit code b of c_a in the cluster description
      stop
      end
    end
  randomly choose another point q ∈ P
  compute an edge length x = ||p − q||_2/√(d/2)
  cl = new cluster(p,x)
  store cl into the database
```

Figure 2.7: Algorithm insertpoint

Unfortunately this may prevent overlap *inside* one cluster. But it does not yet consider overlap between active vertices of several clusters. It is up to an experimental validation to confirm this value.

2.3 Computation of *LowerBound*

Finally, we briefly sketch how to compute the *lower bound* (LB) cluster distance. These value reflects the distance of an arbitrarily located query point q to the nearest virtual cluster point. This distance is required in the Branch-and-Bound algorithm (see Figure 1.1).

In particular, LB must approximate the actual distance of the query point to the nearest data point to obtain a low approximation error. In Figure 2.8 one example for two query points q_1 and q_2 is depicted. For the computation of $\mathrm{LB}(q)$, all active vertices of a cluster have to be visited. Each of those active vertices yields a "local" distance $\mathrm{LB}_i(q)$. For a cluster that comprises m active vertices, the "global" distance can be computed as follows:

$$\mathrm{LB}(q) = \min\{\mathrm{LB}_1(q), \ldots, \mathrm{LB}_m(q)\}$$

The computation of the local distance depends on whether we use a global threshold value $r_{\max}$ which specifies equally sized hyperballs for all active vertices. Or in contrast, we may also use distinct local radii (here: r_i) which create variably sized hyperballs for the active vertices. Moreover, in the second case, we know that the data point (here p_i) is precisely located on the surface of the hyperball. Provided that exactly one data point is assigned to an active vertice, we can exclude the inside of the hyperball from the cluster area.

For the first case (global radius $r_{\max}$), the distance $\mathrm{LB}_i(q)$ of one active vertice c_i is

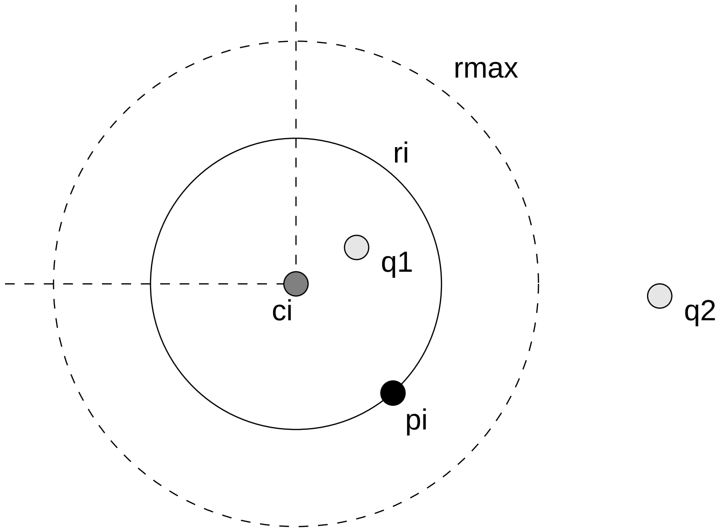


Figure 2.8: Computation of LB(q) and UB(q)

computed as follows:

$$\mathrm{LB}_i(q) = \begin{cases} \|q - c_i\|_2 - r_{\max} & \text{if } \|q - c_i\|_2 \geq r_{\max} \\ 0 & \text{otherwise} \end{cases}$$

In the second case (local radii r_i), the value $\mathrm{LB}_i(q)$ of one active vertice c_i is computed as follows:

$$\mathrm{LB}_i(q) = \begin{cases} \|q - c_i\|_2 - r_i & \text{if } \|q - c_i\|_2 \geq r_i \\ r_i - \|q - c_i\|_2 & \text{otherwise} \end{cases}$$

Please note, that the computation of $\mathrm{LB}_i(q)$ differs due to the fact that the “inside” of the hyperball is part of the cluster in the first case but excluded from the cluster in the second case.

In our example (see Figure 2.8), q_1 located outside the hyperball around the vertice c_i. Therefore, LB is the smallest distance from q_1 to the surface of that hyperball. The second query point q_2 is located inside the hyperball. If this hyperball is part of the cluster shape, i.e. the assigned data point p_i may be located somewhere inside this hyperball, q_2 might have the same coordinates as p_i. This yields a value of zero for the lower bound. In the second case, this hyperball is not part of the cluster region. Therefore, the lower bound is the distance from q_2 to the (inner) surface of the hyperball.

Chapter 3

Experiments

In this chapter, first we experimentally verify the values for the parameters we analytically derived in Chapter 2. Subsequently, we run retrieval performance measurements in different dimensionalities. Furthermore, we compare this cluster geometry with the *R-Shell* clusters in terms of retrieval costs. Finally a thorough discussion concludes this chapter.

3.1 Active Vertice Distribution

At first, we experimentally verify the distribution of active vertices in one cluster. Following our combinatorial analysis from Section 2.2, the most often distance L between two vertices is:

$$L = x \cdot \sqrt{\frac{d}{2}}$$

where x is the edge length of the hypercube. Correspondingly, we conclude that a hypercube should be sized in a way that L corresponds to the distance between two arbitrarily selected data points.

Therefore, we assembled active vertice clusters corresponding to the algorithm sketched in Chapter 2. We ran these experiments for different dimensionalities ($d = 10, 20, 30$) with a number of $N = 1000$ data points and scaled the parameter $r_{\max}$ respectively. Subsequently, based on the starting point p_0 we counted the number of active vertices in a distance of $x\sqrt{0}, x\sqrt{1}, \ldots, x\sqrt{d}$ from p_0. In Figure 3.2 (see Page 21) the experimentally obtained *real* distribution is depicted besides an *ideal* distribution[1].

Please note, that the ideal distribution computes the actual number of vertices in a certain distance from p_0 whereas in the real distribution a number of $N = 1000$ points is separated onto several clusters. The major outcome of these experiments is that the real distribution only loosely follows the ideal distribution. In particular, most active

[1]The value *distance* depicted in Figure 3.2 corresponds to an actual distance of $x\sqrt{\text{distance}}$

vertices are rather located in a distance of $x\sqrt{d/3}$ than $x\sqrt{d/2}$. So we might want to set x to a value of:

$$x \approx \frac{L}{\sqrt{d/3}}$$

Furthermore, we also have to consider a value of

$$x = \frac{L}{\sqrt{2}}$$

which follows the ideal of an inscribed simplex.

To choose the appropriate edge length computation for the scope of this paper, we conducted a further experiment. We measured the overall retrieval costs (in $d = 10, 20, 30$) for the following edge lengths:

- Simplex Method: $x = \frac{L}{\sqrt{2}}$
- Combinatorial Method: $x = \frac{L}{\sqrt{d/2}}$
- Experimental Method: $x = \frac{L}{\sqrt{d/3}}$

In Figure 3.1 the emerging retrieval costs are depicted. It turns out that among the chosen methods the *combinatorial* method best reflects the actual distribution of high dimensional data points. Despite a deviation of point distances from an ideal distribution (see Figure 3.2) we do not obtain superior retrieval costs for the *experimental* method. Active Vertice clusters based on hypercubes with an edge length computed with the *simplex* method reveal a remarkably worse retrieval performance than both other methods.

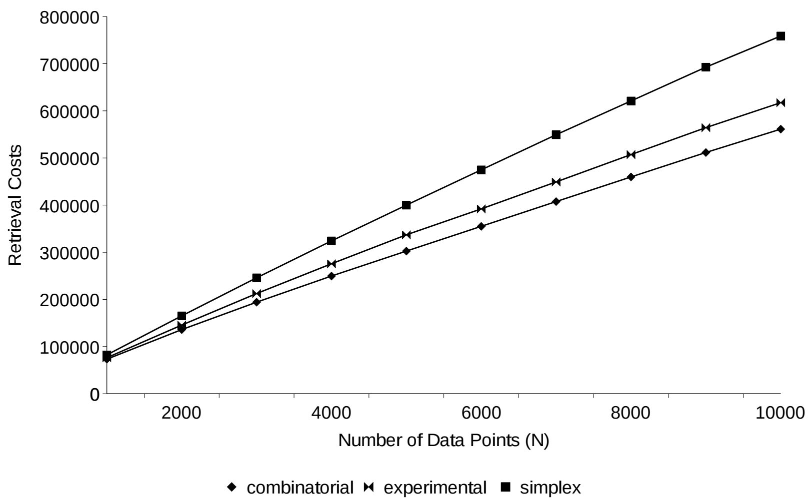
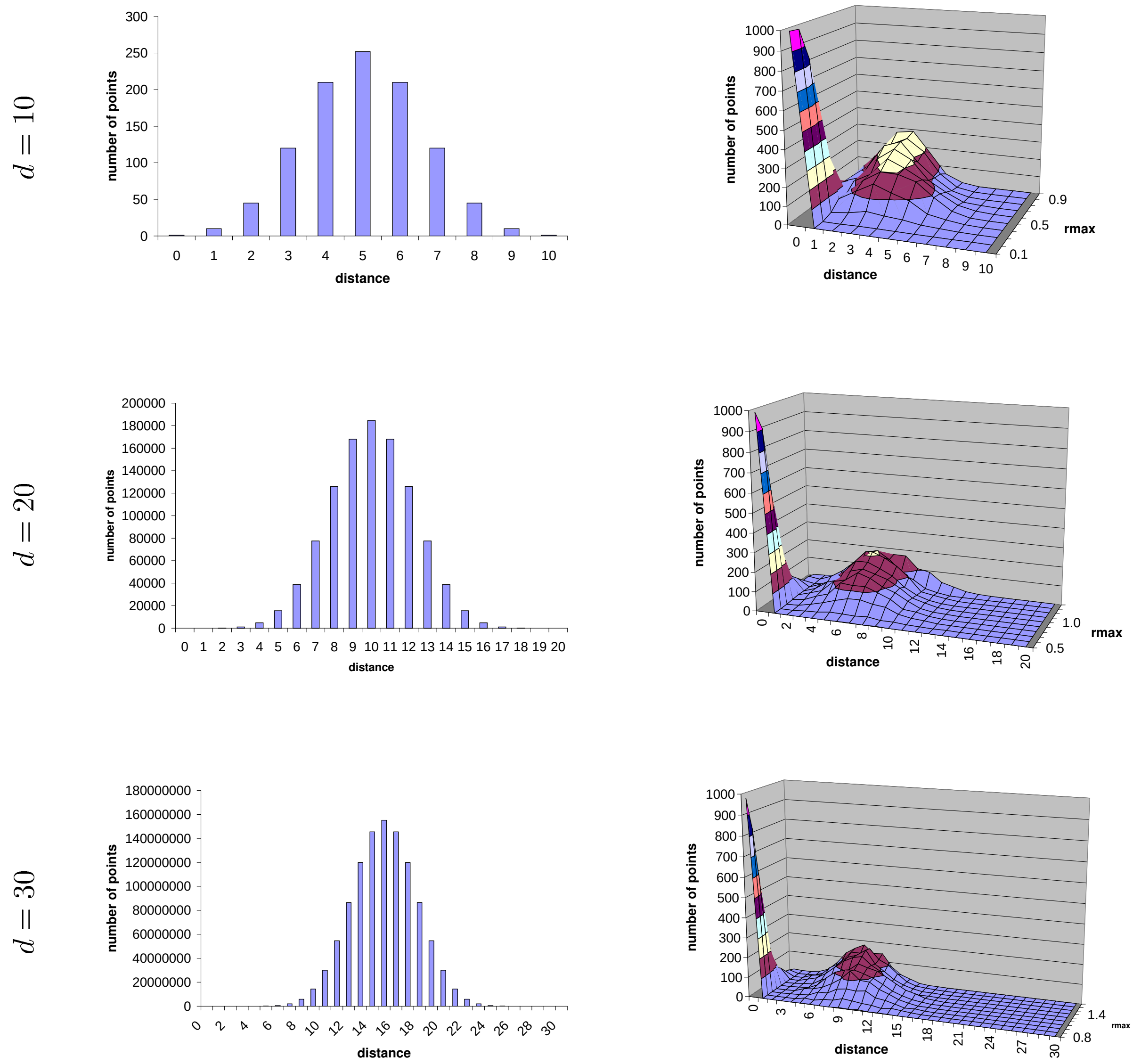


Figure 3.1: Retrieval Costs for Distinct Edge Length Computation Methods



Ideal Distribution *Real Distribution*

Figure 3.2: Distribution of Distances between Active Vertices from a Starting Point

3.2 Retrieval Performance

In this section, we present the retrieval costs measured in several experiments with Active Vertice clusters in $d = 10, 20, 30$. To obtain the (near) minimal retrieval costs for a given dimensionality, we scaled the threshold value of $r_{\max}$ respectively:

- $d = 10$: $r_{\max} = [0.1, \ldots, 1.0]$
- $d = 20$: $r_{\max} = [0.4, \ldots, 1.3]$
- $d = 30$: $r_{\max} = [0.6, \ldots, 1.5]$

These experiments were conducted for $N = 1000, \ldots, 10000$ data points. In this way, we obtain an optimal value of $r_{\max}$ for each dimensionality d and number of data points N. In the left side of Figure 3.4 (see Page 24) the overall retrieval costs for the different threshold values (increased in steps of 0.1) are depicted. It turns out that the optimal threshold values of $r_{\max}$ are

- ≈ 0.4 for $d = 10$,
- ≈ 0.7 for $d = 20$, and
- ≈ 0.8 for $d = 30$

for all numbers of data points. The experimental determination of an optimal threshold value for $r_{\max}$ is not satisfactory. To avoid time-consuming experiments an analytic determination is favorably. To analytically find out appropriate values for this parameter, we point out its two purposes:

On the one hand, it has to be sufficiently large to "enclose" (i.e. approximate) as many data points as possible in one Active Vertice cluster. On the other hand, a properly sized $r_{\max}$ should generate as few overlaps between hyperballs around other active vertices as possible to reduce the approximation error.

The expecting value for point distances (see Section 1.2 and [Sch01]) tends towards $\sqrt{d/6}$ in dimensionality d. If we build hyperballs around two arbitrarily selected data points their common radius must not be larger than $(\sqrt{d/6})/2$ to avoid an overlapping of these hyperballs. In Figure 3.3 we depicted this computed threshold value for $r_{\max}$. In addition, we measured the retrieval performance in the chosen dimensionalities for $N = 5000$ data points.

We scaled the threshold value for $r_{\max}$ in intervals of 0.025 to find out the optimal value. That is, the threshold value for $r_{\max}$ that yields the best retrieval performance. The results of this experiment is also depicted in Figure 3.3.

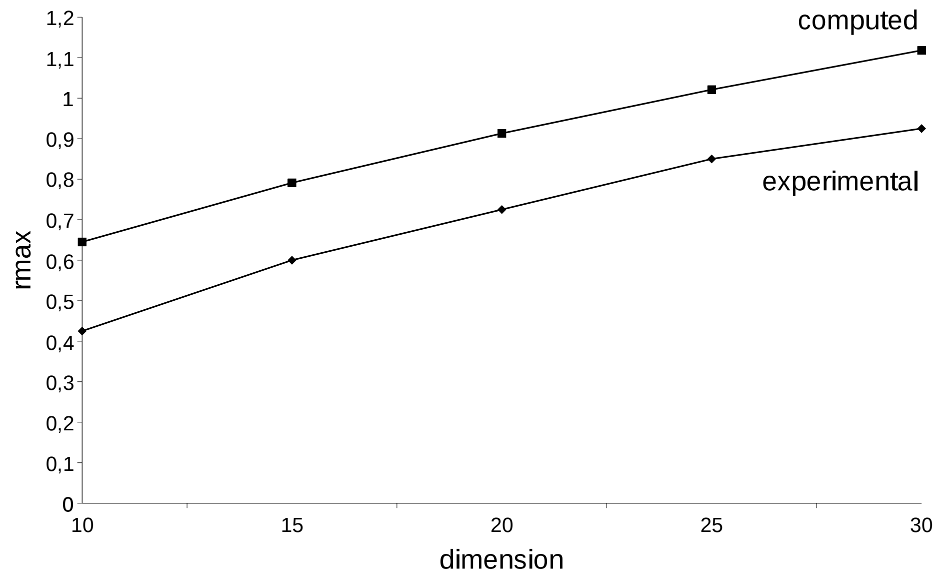


Figure 3.3: Computed and Experimental Threshold Values for $r_{\max}$

There seems to be a constant difference between the computed and the experimentally determined threshold values for $r_{\max}$:

$$r_{\max} = \frac{\sqrt{d/6}}{2} - c$$

Since the expecting value for point distances reflects the mean but not the minimal distance, the experimentally identified values are certainly smaller. It is up to further research to find out if c is really a constant, even for larger numbers N of data points.

On the right hand side of Figure 3.4 we compare the retrieval performance of Active Vertice clusters with sequential scan costs and convex and totally concave R-Shell cluster retrieval performance. For lower dimensionalities ($d = 10$) all considered clusters unveiled a nearest neighbor retrieval performance superior to sequential scan. Active Vertice clusters rank between convex (better) and totally concave (worse) R-Shell clusters.

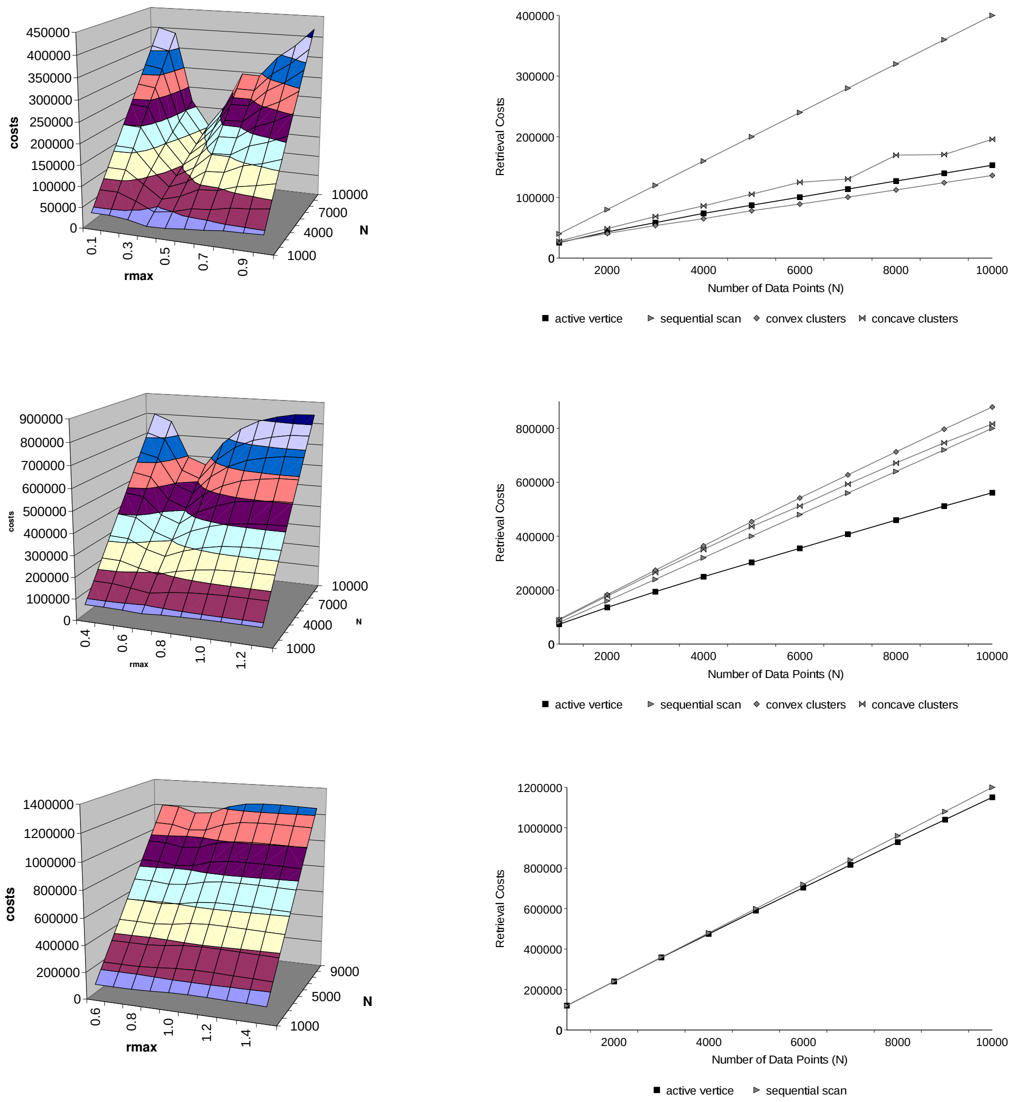
For $d = 20$ this situation changes. Both types of R-Shell clusters rank above sequential scan costs. However, at least the totally concave R-Shell cluster exhibits a sublinear growth. Nevertheless, Nearest Neighbor retrieval based on Active Vertice clusters performs remarkably better than all other clustering methods. In fact, for $N = 10000$ data points, Active Vertice retrieval takes approximately 70% of sequential scan costs.

In comparatively high dimensions ($d = 30$), R-Shell clusters suffer from the so-called "curse of dimensionality". That is, they do not outperform sequential scan. However, Active Vertice clusters still perform slightly better than sequential scan.

$d = 10$

$d = 20$

$d = 30$



Active Vertice Clusters *Comparison*

Figure 3.4: Retrieval Performance of Active Vertice Clusters

3.3 Discussion

The retrieval performance obtained in the experiments indicates that *Active Vertice* clusters outperform index structures based on convex cluster geometries. In particular, a precise location of *single* data points pays out in the retrieval process.

Nevertheless, Active Vertice clusters are just a first step in concave cluster geometries. For dimensionalities above ≈ 30 they exhibit a deteriorated retrieval performance that approaches sequential scan costs. For the optimal threshold value of $r_{\max}$ this deterioration is *not* caused by a worsen approximation error as it is for convex clusters. Instead, Active Vertice clusters do not sufficiently adapt to the location of high dimensional data points any longer. In turn, the cluster utilization shrinks in rising dimensionality which causes larger cluster retrieval costs.

In Figure 3.5 the average number of data points in one cluster is depicted for different dimensionalities. Since this value decreases we obtain larger cluster retrieval costs.

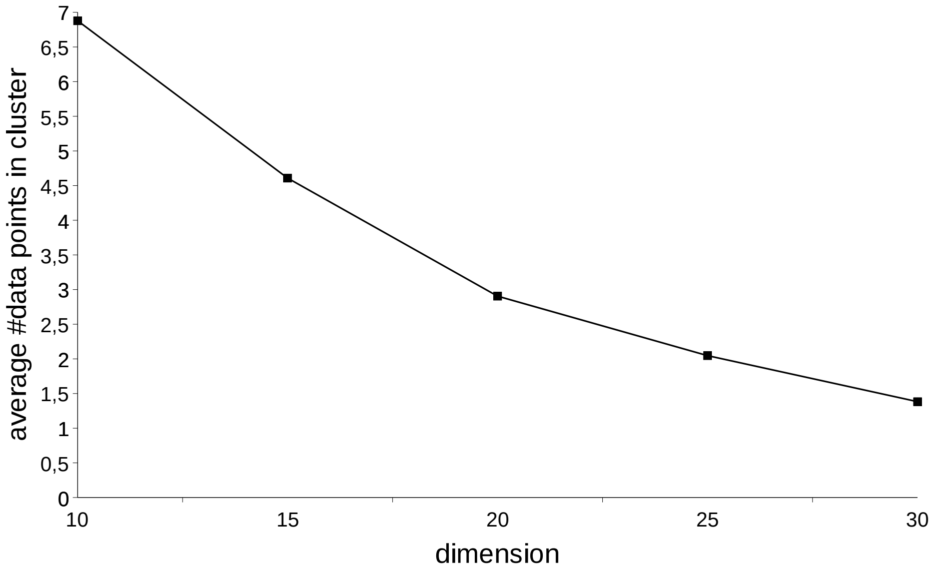


Figure 3.5: Average Number of Data Points in one Cluster

Even, if we neglect the number of emerging clusters, we can state one lower bound for the storage size each data points requires in an Active Vertice index structure. That is, besides edge length and starting points (not considered here) we need at least one bit vector (d bits) for each data point. In this ideal case (lower bound), we have $d/8$ bytes for each data point. In contrast, the storage space of a data point is $d \cdot \text{size of}(\mathsf{FLOAT})$. Though this *is* a considerable difference, it still means linear costs in the ideal case. No clusters are pruned during the retrieval process, so the complete set of cluster geometries has to be retrieved.

Furthermore, as Figure 3.5 indicates, the number of clusters quickly increases for rising dimensionalities. This means, we *additionally* have $d \cdot \text{size of}(\mathsf{FLOAT})$ bytes for the starting point in each cluster which drastically deteriorates the actual cluster retrieval costs for each cluster. Therefore, future progresses of the Active Vertice cluster should aim at avoiding the additional costs for the starting point. In addition, they must superiorly approximate the location of higher dimensional data points while maintaining the low approximation error.

Finally, the *Branch-and-Bound* algorithm is probably less appropriate for concave cluster geometries. In particular, we currently do not know how to *prune* certain clusters from consideration in high dimensionalities which is the actual idea of a Branch-and-Bound algorithm. Therefore, a better suited algorithm should (i) retrieve the index structure (i.e. the concave clusters) *without* their comprised data points, (ii) compute the approximated locations of data points out of the index structure, and (iii) only retrieve *necessary* data points from the data points to figure out the nearest neighbor point.

Chapter 4

Summary and Conclusions

In this paper we introduced the *Active Vertice* cluster geometry – a concave cluster shape for high-dimensional nearest neighbor retrieval.

4.1 Summary

In *Chapter 1* we described the problem of high-dimensional clustering and nearest neighbor retrieval. Moreover, we introduced basic foundations like the chosen data point distribution and distance function, presented a simple retrieval algorithm and stated general requirements for high dimensional cluster geometries.

Chapter 2 motivates the specific Active Vertice cluster geometry by means of a regular simplex model for data point alignments. Subsequently, we thoroughly specify the actual cluster geometry and make some statements about specific cluster retrieval costs. Following, we propose an efficient clustering algorithm and discuss the computation of several geometrical parameters. Finally, we explain the computation of approximated distances from a query point.

In the subsequent *Chapter 3* we experimentally verify the analytically determined cluster parameters. Following, we conduct several retrieval performance measurements in different dimensionalities. We relate the obtained results to other concave and convex cluster geometry proposals. A thorough discussion points out the advantages of the Active Vertice cluster geometry. We also name the drawbacks and formulate some arising requirements for future advancements of this approach.

4.2 Conclusions

Active Vertice clusters are a consequential step towards concave cluster geometries. They combine numerous advancements compared to previous concave geometries like the R-Shell cluster. In fact, Active Vertice clusters feature a structurally simple geometry which can cheaply be stored in a database. They comprise very precisely specified regions in the vector space. These advantages pay out in terms of a superior retrieval performance.

The second major outcome is that concave cluster geometries reveal as another *data compression* approach. Many recently proposed nearest neighbor index structures like the *VA-File* [WB97, WSB98] or the *IQ-tree* [BBJ+00] tend towards data compression. Our current concave cluster geometries like the *R-Shell* cluster or the *Active Vertice* cluster should be regarded as preliminary milestones which are based on partially different theoretical foundation. Clearly, further research has to be done with respect to even higher dimensions where Active Vertice clusters do not seem to sufficiently precisely describe the alignment of data points. Furthermore, it is up to future research to establish possible correlations to other data compression approaches.

Bibliography

[BBJ+00] Stefan Berchtold, Christian Böhm, H. V. Jagadish, Hans-Peter Kriegel, and Jörg Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *ICDE*, pages 577–588, 2000.

[BBK01] C. Böhm, S. Berchtold, and D.A. Keim. Searching in High-dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 2001.

[BKK96] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proc. of the 22nd Int. Conf. on Very Large Data Bases, VLDB'96, Bombay, India, September 3–6, 1996*, pages 28–39. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 322–331. Atlantic City, NJ, May 1990.

[BS01] S. Balko and I. Schmitt. Concave Cluster Shapes for Efficient Nearest Neighbor Search in High Dimensional Space. Preprint, Fakultät für Informatik, Universität Magdeburg, 2001. To appear.

[FK70] K. Fukunaga and W. L. G. Koontz. Representation of random processes using the finite Karhunen-Loève expansion. *Information and Control*, 16(1):85–101, 170.

[FL95] Christos Faloutsos and King-Ip Lin. FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 163–174, ACM Press, 1995.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In B. Yormark, editor, *Proc. of the 1984 ACM SIGMOD Int. Conf. on Man-*

agement of Data, Boston, NJ, ACM SIGMOD Record, Vol. 14, No. 2, pages 47–57. ACM Press, New York, 1984.

[KS97] Norio Katayama and Shin'ichi Satoh. The SR-tree: an Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proc. of the 1997 ACM SIGMOD Int. Conf. on Management of Data*, pages 369–380, 1997.

[KW78] Joseph B. Kruskal and Myron Wish. *Multidimensional Scaling.* Sage Publications, Beverly Hills, Calif., USA, 1978.

[LJF94] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree – an index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–549, October 1994.

[MKB79] K.V. Mardia, J. T. Kent, and J. M. Bibby. *Multivariate Analysis.* Academic, 1979.

[NG88] H. Niemann and R. Goppert. An efficient Brand-and-Bound Nearest Neighbour Classifier. *Pattern Recognition Letters*, 7:67–72, 1988.

[Sch01] I. Schmitt. Nearest Neighbor Search in High Dimensional Space by Using Convex Hulls. Preprint No. 6, Fakultät für Informatik, Universität Magdeburg, 2001.

[WB97] R. Weber and S. Blott. An Approximation-Based Data Structure for Similarity Search. Technical Report No. 24, ESPRIT project HERMES (no. 9141), 1997.

[WJ96] D. White and R. Jain. Similarity Indexing with the SS-tree. In *Proc. of the 12th Int. Conf. on Data Engineering, New Orleans, Louisiana*, pages 516–523, 1996.

[WSB98] R. Weber, H.-J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In A. Gupta, O. Shmueli, and J. Widom, editors, *Proc. of the 24st Int. Conf. on Very Large Data Bases, VLDB'98, Ney York City, August 24–27, 1998*, pages 194–205. Morgan Kaufmann Publishers, San Francisco, CA, 1998.