High dimensional kNN-graph construction using space filling curves

Sami Sieranoja

ITÄ-SUOMEN YLIOPISTO

April 2015

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry,
Joensuu
School of Computing
Computer Science

Student, Sami Sieranoja: High dimensional k nearest neighbor graph construction
Master's Thesis, 59 p.
Supervisor of the Master's Thesis: Pasi Fränti
April 2015

Abstract:

The k nearest neighbor (kNN) graph has an important role in many computer science fields including machine learning and data mining. Although many fast methods exist for constructing kNN graph for low dimensional data, it is still an open question how to do it efficiently in high dimensional cases. We present a new method to construct approximate kNN graph for medium to high dimensional data. Our method uses space filling curves to construct initial graph and then continues to improve this using neighborhood propagation. It is targeted for Euclidean distance metric but has potential to be applied for general Minkowski distance metrics. Experiments show that the method is faster than compared methods with three different benchmark data sets which dimensionality ranges from 14 to 544.

Keywords: kNN graph, graph construction, nearest neighbor, space filling curves, neighborhood propagation

(ACM Computing Classification System, 1998 version): H.3.3 Information Search and Retrieval

Acknowledgements

I would like to express my gratitude to my supervisor, professor Pasi Fränti for the useful comments, remarks and guidance through the difficult process of writing this master's thesis. Additionally, I would like to thank all members of the Speech and Image Processing Unit (SIPU) where I was working while writing this thesis. I also thank my family for their encouragement, support and patience.

Abbreviations

NN	Nearest neighbor
kNN	k nearest neighbor
ANN	All nearest neighbor
SFC	Space filling curves
NNDES	Nearest Neighbor Descent
RLB	Recursive Lanczos Bisection
k-d tree	k -dimensional tree
PCA	Principal component analysis

Notations

P	Data set of points in some multidimensional space
p	point in data set P
$d(p_1, p_2)$	distance between points p_1 and p_2
q	query point
k	number of nearest neighbors in kNN and k-NNG
k	(in context of k-d trees) number of dimensions
N	size of data set P
D	dimensionality of the space
D_z	Target dimensionality in dimensionality reduction
$s(p)$	One-dimensional mapping of multidimensional point p
N_c	Number of different one-dimensional mappings (curves)
$L_p(x, y)$	$= \left(\sum_{i=1}^{n} \lvert x_i - y_i \rvert^p \right)^{1/p}$

Contents

1 Introduction

Given a set of N points $P = \{p_1, p_2..., p_N\}$ in some multidimensional space S, the *k-nearest neighbor* problem (kNN) is to find the k points in P that are closest to a given query point $q \in S$ according to some distance metric d.

When the k nearest neighbors are searched for all points in P, the result is a directed graph called *kNN graph* where the vertices correspond to points in the data set and edges are from each point to the k nearest points in the data set.

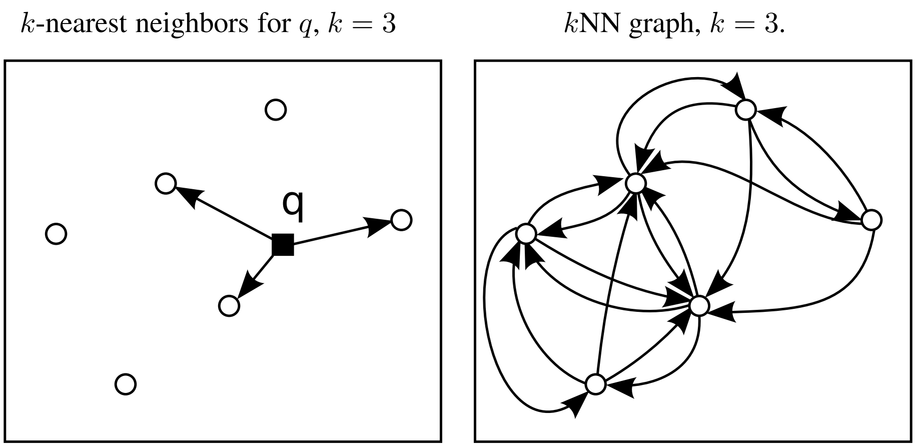


Figure 1: When k nearest neighbors are searched for all points in P, the result is a kNN graph.

Constructing a kNN graph is important for a wide range of applications including classification [1], agglomerative clustering [2], k nearest neighbor search [3], dimensionality reduction [4], image organization [5], outlier detection [6] and computer graphics [7]. For many of these applications, constructing the graph is a major bottleneck [8, 2].

Trivial brute force algorithm constructs kNN graph in $O(N^2)$ time by calculating distance for all possible pairs of points and selecting the k smallest distances for every point. This can be practical for small data sets consisting of up to tens of thousands of points, especially when utilizing parallel computing capabilities of modern GPUs [9]. However, for larger data sets, consisting of millions of points, the brute force algorithm becomes too slow.

Faster methods to construct the kNN graph can be roughly grouped into three categories: (1) *Space partitioning* methods construct tree shaped search indexes by recursively dividing the space. (2) *Space filling curves* project multidimensional points into proximity preserving one dimensional curves and perform local search along the curve. (3) *Neighborhood propagation*, given an approximate or random graph, finds neighbors of points by examining every point's neighbors' neighbors.

Fast methods such as k-d trees [10] and space filling curves [7] can construct an exact kNN graph for low dimensional data sets in $O(N \cdot \log(N))$ time. However, with high dimensional data these methods fail to provide speed-up over the brute force method. Therefore recent research has focused on constructing approximate kNN graphs [2, 11, 12, 13, 8].

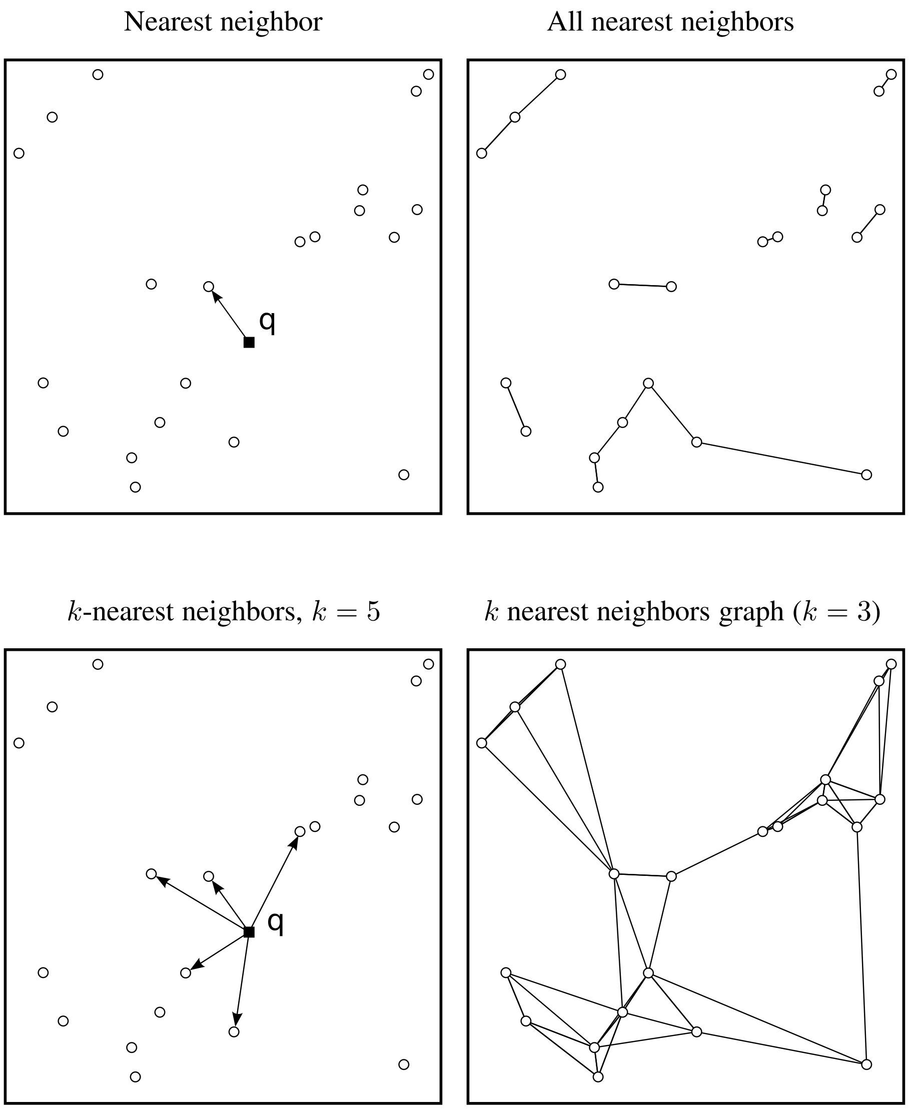
Among recent research, Dong *& al.* [12] and Zhang *& al.* [8] have reported significant improvements over previous methods. Their methods are designed to work in high dimensions and with many types of distance measures, including non-metric distance functions. We suspect that it might be possible to improve those methods even further by focusing on a specific distance measure. Many different distance measures have been developed, but the Euclidean distance is still the most widely used. In this work, we aim therefore to develop a method specifically for the Euclidean distance, but expect the developed method to generalize to Minkowski distance metrics as well.

Our method constructs a kNN graph by using a combination of space filling curves and neighborhood propagation. It aims at constructing an approximate kNN graph for the Euclidean distance metric in situations where dimensionality D ranges from medium to high ($D \geq 10$). We compare the method against two existing methods, NN-Descent [12] and Recursive Lanczos Bisection [11]. We show that our method provides better speed/quality ratio than the compared methods when using Euclidean distance metric with data sets where dimensionality ranges from 14 to 544.

We analyze the effect of increasing dimensionality on previous methods k-d trees [14, 10] and Neighborhood Propagation [11, 12, 13, 8]. We show that k-d trees become less efficient in higher dimensions because a smaller portion of the data has effect on the tree structure. Our experiments show that Neighborhood Propagation fails in higher dimensions because, at least with uniformly distributed data, the probability of neighbor of neighbor being a neighbor decreases as dimensionality increases.

In addition to decreasing computational performance, an increase in dimensionality may also cause even the exact results to become less meaningful. This happens because in many situations, such as with uniformly distributed data, all points become almost equidistant from the query point [15]. Additionally, choosing a right distance function has effect on the meaningfulness of the results [16]. Minkowski distance functions with small p have been shown to be more suitable for high dimensions than larger p [16].

2 Nearest neighbor problems



Nearest neighbor
All nearest neighbors
q
k-nearest neighbors, $k = 5$
k nearest neighbors graph ($k = 3$)
q

Figure 2: Solutions for different nearest neighbor problems. Graphs for all nearest neighbors and all k nearest neighbors are directed but drawn here as undirected for clarity.

The problem of construction the kNN graph belongs to a larger set of problems that involve finding points that are near to each other. We refer to these problems collectively as *nearest neighbor problems*. They vary on how many nearest neighbors are searched for and how many query points there are. For example, the *nearest neighbor problem* (NN) is to find the nearest neighbor for a point, and the *all nearest neighbor problem* is to find one nearest neighbor for all points in the data set. Solving these problems faces similar computational challenges and similar solutions are used to all these problems. We consider the following nearest neighbor problems:

1. Nearest Neighbor (NN)
2. All nearest neighbors (ANN)
3. k-nearest neighbors (kNN)
4. All k nearest neighbors (AkNN) or kNN graph

The *nearest neighbor* problem (NN) is to find the point $p \in P$ that is closest to a given query point q according to some distance metric d. This problem has also been called the *post office problem* because one instance of the problem is to find the closest post office for a given residence [17].

In the *all nearest neighbors* problem (ANN) the goal is to find the nearest neighbor for all points [18, 19]. In mathematical ecology all nearest neighbors have been used to study distributions of plant or animal populations [20, 18]. For example, a population can be determined to be randomly distributed if the distribution of distances to nearest neighbor in a population is the same as the distribution of distances from random points to the nearest individual in the population [20].

Given a set of N points $P = \{p_1, p_2..., p_N\}$, the *k-nearest neighbor* problem (kNN) is to find the k points that are closest to a given query point q according to some distance metric d [21]. The query point q may or may not belong to the set of data points P. All kNN search algorithms reviewed by us [10, 22, 21, 23, 24, 3] have been developed to work also in cases where $q \notin P$. This problem is encountered in a very wide range of practical applications. One example of this problem is a situation where a mobile phone user wants to find the nearest restaurants to his current position (query point q).

In another variant of k nearest neighbors problem, the goal is to find k nearest neighbors for all data points in the same P [25, 26]. In this thesis, we focus in this particular problem. The difference between k nearest neighbors problem and the all k nearest neighbors problem is the same as between the nearest neighbor problem and the all nearest neighbor problem. The nearest neighbors problems are special cases of the k nearest neighbor problem when $k = 1$.

If the results for the all k nearest neighbors problem are stored as a graph data structure, it is called *kNN graph*. Given a data set P and some distance metric d, the kNN graph for P is an unweighted directed graph where the vertices correspond to points in the data set and edges are from each point to the k nearest points in the data set [13]. More formally, $G = (V, E)$ is a directed graph, where $V = P$ and for all $p_i \in P$ there is an edge $\langle p_i, p_j \rangle \in E$ if and only if $d(p_i, p_j)$ is among the k smallest elements in $\{d(p_i, p_j) | p_j \in P \setminus \{p_i\}\}$. Sometimes edge weights are also stored to represent the distance between points [27, 28]. They can be useful in estimating densities of different areas.

A kNN graph can serve as $O(1)$ time search structure for accessing the neighbors of any points in P. However, kNN graph cannot be directly used for finding k neighbors for other query points outside this set ($q \notin P$) because kNN graph is constructed for a predefined set of data points ($p \in P$) [13]. Also, in k nearest neighbor searches k is taken as a parameter for the search [21, 24], and not for indexing, so it can be varied per query point, whereas kNN graph is constructed so that k is same for all points. Therefore $k + 1$ neighbors cannot always be efficiently found using a kNN graph.

When query point is outside the set of data points ($q \notin P$) it is also possible, although not trivial, to use kNN graph as a serch data structure [29, 3]. For example, Hajebi *& al.* [3] perform approximate kNN search by repeated hill climbing. The search starts from a random point in the kNN graph and continues by always choosing that neighbor which is closest to the query point while maintaining a list of the k visited points that are closest to q. The search stops when none of the neighbors are closer to the query point than the current node. To increase accuracy, the search is repeated multiple times by using a different random data point as a starting point.

3 Effect of dimensionality

There have been a great deal of theoretical studies analyzing the effect of dimensionality for the nearest neighbor search [15, 30, 16], but much less specifically about the k nearest neighbor search [31]. The analysis presented in this section is exclusively about the nearest neighbor search. Some of these results generalize to the k nearest neighbor problem. While this may seem intuitively quite clear, the relationship of these problems should be understood more formally.

Especially much analysis have been done on the computational efficiency of solving the nearest neighbor problem. It is a subproblem of the k nearest neighbor search, and the time complexity of kNN search is therefore lower bounded by the time complexity of NN search. If it takes $T(n)$ time to calculate NN, it will take at least $T(n)$ time to calculate kNN. Also, k nearest neighbors can be searched for by repeating the search for the nearest neighbor k times while removing the found neighbor from P after each search. Therefore, if nearest neighbor search takes $T(n)$ time, kNN search is expected to take at most $T(k \cdot n)$ time. Therefore theoretical results for the NN problem have relevance for the kNN problem.

There are two major issues in high dimensional nearest neighbor search. One is the *performance* issue where computational requirements of known exact nearest neighbor search methods increase exponentially in D [32], and therefore, are not practical for most high dimensional real life data sets.

The second is the *quality* issue where increase in dimensionality makes all points almost equidistant from the query point when using typical distance functions such as L_2 distance. This gives rise to the question if the concept of nearest neighbor is meaningful in high dimensions [15] or if other non-conventional distance functions would be more useful [16].

The quality issue has been investigated in [15, 30, 16]. Beyer *& al.* [15] discovered that with many types of high dimensional data sets all data points are almost equidistant from the query point. In other words, when dimensionality increases towards infinite the relative distance between the farthest (d_F) and nearest (d_N) data point (contrast) goes towards one. They also argued that the concept of nearest neighbor is no longer meaningful when all data points are almost equidistant from the query point. Consequently, the meaningfulness of the nearest neighbor concept would decrease when dimensionality increases.

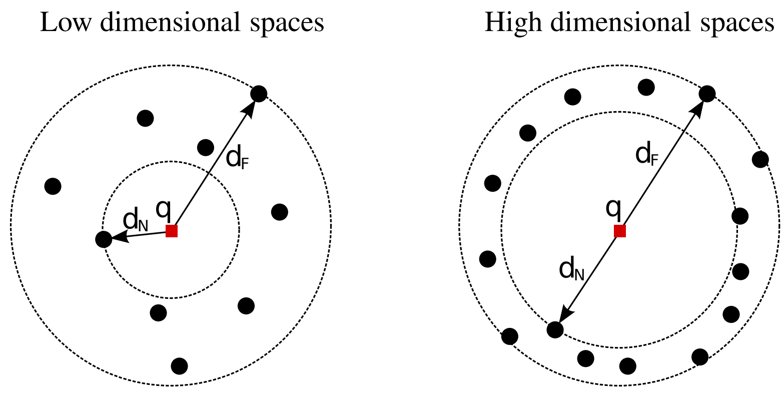


Figure 3: In high dimensional spaces ratio of the nearest and farthest data points (d_F/d_N) is close to one. The example on the right (high dimensional case) represents the distances only from query point q to data points (black circles).

$$contrast = d_F/d_N \tag{1}$$

Beyer *& al.* proved that this loss of contrast caused by increase in dimensionality happens in many different situations, most notably for cases when the data points are independent and identically distributed and the query point is chosen independently of the data points. One example of such a case is when the data and query points are uniformly distributed random points. In this case, with one dimension the distance to farthest point was an order of 10^7 times the distance to nearest point whereas with 20 dimensional data this figure was 4.

Beyer *& al.* also noted that there are many situations where increase in dimensionality does not cause a loss of contrast:

- *Exact match and approximate match queries.* If the query point matches almost exactly to one of the data points ($d_N \approx 0$), then contrast would be very large.
- *Data with clusters.* If the data contains clusters where groups of points are within some small δ from the cluster centroid and the query point falls within a cluster ($d_N \approx \delta$), this causes a high contrast. However, if the points inside the cluster are uniformly distributed, then they are likely to be almost equidistant from the query point and it does not matter which of those points is returned as the result.

- *Underlying inherent dimensionality of the data is low.* For example, when all data points are on the same line, the inherent dimensionality of the data is one regardless of the dimensionality of the space where those points reside.

Also selecting a suitable distance function is critical in providing meaningful nearest neighbor search results. Aggerwal *& al.* [16] investigated the effect of varying the value p for Minkowski distance metric L_p for high dimensional data. The conclusion was that fractional values ($p < 1$) generally provide better contrast than integral values ($p \geq 1$). They also showed with many real data sets that classification accuracy is best with small values of p such as 0.1 or 0.5.

There is one major difference between small and large p Minkowski distances that might explain the findings of Aggerwal *& al.* When p goes towards zero, the distance measure becomes less sensitive to large differences in individual dimensions. Consider for example a case with three 6-D vectors $q = (1, 1, 1, 1, 1, 1)$, $x = (1, 1, 1, 1, 1, 5)$ and $y = (1, 2, 3, 2, 3, 2)$. Notice that q is exactly the same as x except for the last dimension. According to L_2 -metric y would be closest to q, whereas with $L_{0.5}$ -metric x would be closest to q.

$$
\begin{array}{ll}
L_p(x,y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p} & \\
L_2(q,y) = 3.32 & L_2(q,x) = 4 \\
L_{0.5}(q,y) = 33.97 & L_{0.5}(q,x) = 4
\end{array}
$$

4 Solving methods

Many solving methods exists for kNN graph construction such as Divide and Conquer [28], Recursive Lanczos Bisection [11] and Locality Sensitive Hashing (LSH) [8]. In this section we focus on the following three approaches:

1. k-d trees [10]
2. Space filling curves [7]
3. Neighborhood propagation [12]

4.1 k-d trees

A k-d tree (k-dimensional tree) is a tree shaped search index formed by recursive subdivision of the data set [14]. It was first introduced for nearest neighbor searching in [14], but the same search principle have been adapted also to other problems such as k nearest neighbor searches [10].

According to Connor *& al.* [7] k-d tree is one of the best methods to construct kNN graph for low dimensional data sets. Especially the k-d tree implementation in ANN library[1] has been widely used for low dimensional data sets.

A k-d tree is formed by recursively dividing the data set into two halves. First the median of the points is calculated according to the first dimension and then the points are divided into two subsets according to the median: smaller and larger. These parts become the child nodes. The process is repeated recursively for each child, and in each recursive step alternating the dimension used to calculate the median value. The process is repeated until only one point is left at each child node. These are called terminal nodes.

[1]Mount D. M.: ANN: A library for approximate nearest neighbor searching. www.cs.umd.edu/ mount/ANN/

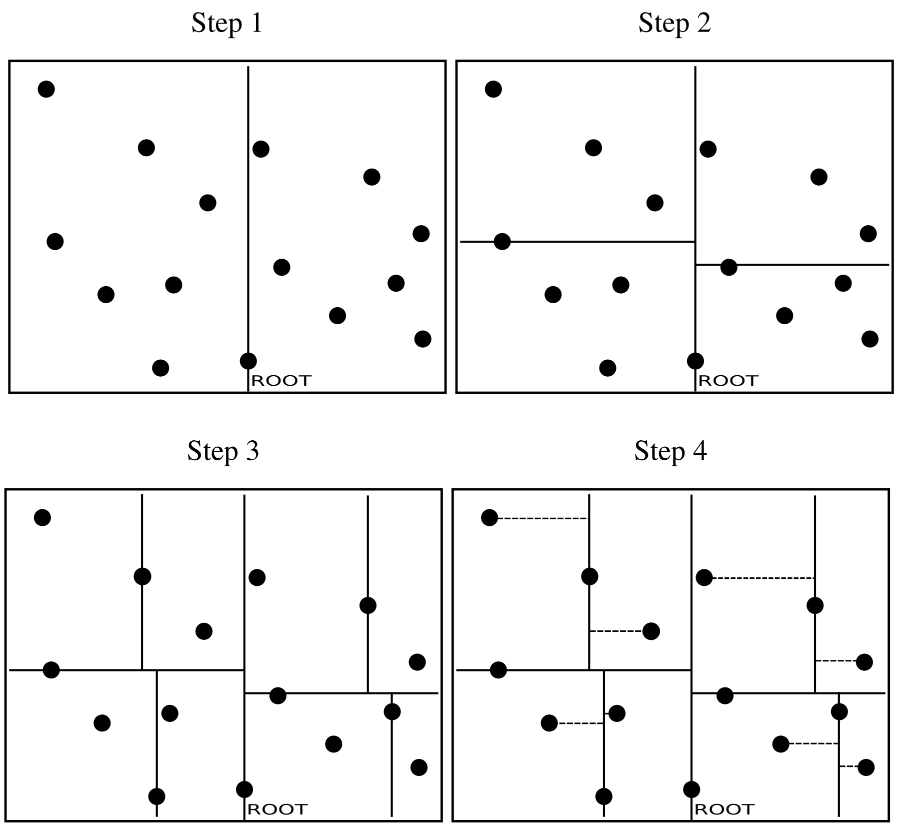


Figure 4: Four recursive steps to construct a k-d tree for a 2 dimensional dataset with 14 points (black circles).

The search for k nearest neighbors starts from the root node of the tree. In each iteration, the child node that is on the same side of the dividing line as the query point is chosen. This is continued downwards the tree until terminal node is reached. For every investigated node the search calculates the exact distance and keeps record of the k nearest points found so far. The minimum ball centered at q and containing all of the current k nearest neighbors is referred to as *kNN ball* and it contains also the exact k nearest neighbors.

After reaching terminal node, the search returns upwards towards root of the tree. In each iteration, we check if the current kNN ball overlaps with the geometric boundaries of the child nodes (ball-overlap-bounds). If it does, the search descends to those child nodes (if not visited). Otherwise it returns upwards. After the ball-overlap-bounds check has been performed for all child nodes, the search checks if the current node contains the kNN ball within its geometric boundaries (ball-within-bounds). If it does, the search ends and returns current k nearest neighbors as the exact results.

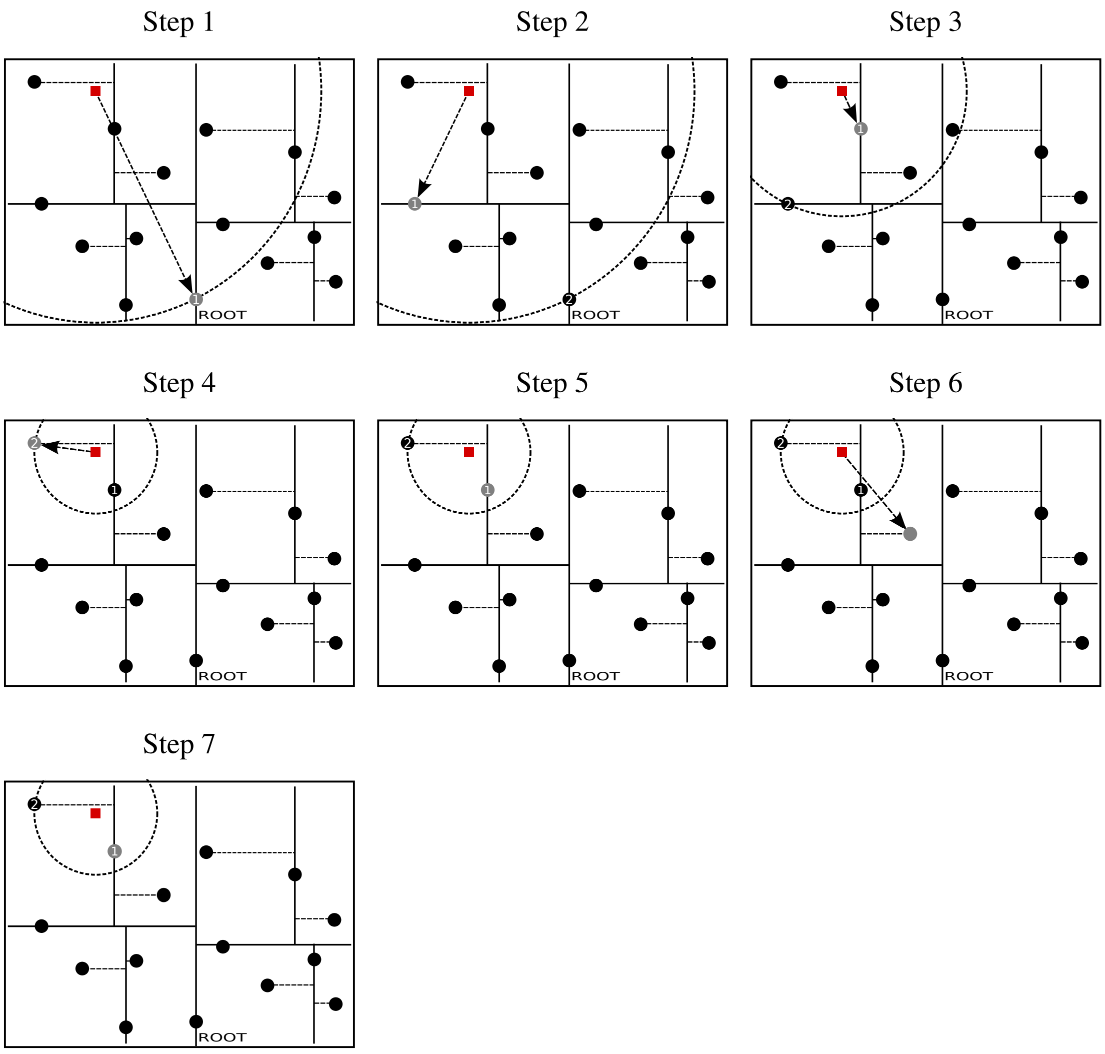


Figure 5: Finding exact $k = 2$ nearest neighbors for query point q (red rectangle) using k-d tree. The grey circle represents the current node. Numbers 1 and 2 represent current 1st and 2nd nearest neighbor candidates. The arrow is present in situations when a distance calculation is performed. Search terminates at step 7 when all children of the node have been investigated and the current kNN ball is within the geometric boundaries of current node.

Constructing a k-d tree takes $O(N \cdot \log N)$ time, and for low dimensional data, search takes $O(\log N)$ time [10]. However, for higher dimensional data they do not work as well. Sproull showed empirically that the running time increases exponentially with dimension [33]. Experiments by Arya *& al.* [34] showed that it is possible to keep the search time logarithmic, but this requires that N should be exponential with dimension.

To understand why k-d trees generally fail for higher number of dimensions, consider a data set of one million (1,000,000) points. Dividing it recursively into two halves according to each dimension, size of the child nodes is halved in each step when the depth of the tree increases. The terminal nodes are reached in about $\log_2(1000000) \approx 20$ iterations.

If there are more than 20 dimensions, only the first 20 dimensions would effect the structure of the k-d tree but all dimensions effect the actual distance calculations. In general, the proportion of dimensions, and thus the proportion of data, that effect the structure of the tree is

$$E_{data}(N, D) = \begin{cases} \frac{\log_2(N)}{D} & \log_2(N) \leq D \\ 1 & \log_2(N) > D \end{cases} \tag{2}$$

With $N = 1000000$ and $D = 1000$ this proportion would be 2%. Intuitively, if the tree represents only a small fraction of the data, then it is unlikely to be effective data structure for nearest neighbor search. It can be seen from Equation 2 that the efficiency of k-d trees depends both on the number of points and the number of dimensions. More specifically, N should be $O(2^D)$ to keep $E_{data}(N, D)$ constant. To the best of our knowledge, no formal proof for this have been given, but it was empirically shown in [34] that k-d trees perform better than brute force search only when N is larger than 2^D. So, k-d trees can be effective for any dimensionality if the number of data points is large enough.

Although single k-d trees usually perform badly for high dimensional data, it is possible to extend their use to higher dimensions by using alternative search methods on multiple k-d trees to calculate approximate results [35, 36]. For example, Silpa *& al.* [35] use priority search on multiple randomly rotated k-d trees to find nearest neighbor on 128-dimensional SIFT image feature data set.

4.2 Space filling curves

A Space filling curve is a way of mapping the discrete multidimensional space into the one dimensional space [37]. It imposes an linear order for points in multidimensional space. This order usually preserves the proximity of points so that points that are near to each other in the multidimensional space can be found by searching locally along the curve.

Space filling curves have been used for many types of problems that include the notion of distance between points. Such problems are range search [38, 39], searching for nearest neighbor [40, 41], k nearest neighbor [41, 42, 43, 24] and constructing the kNN graph [7]. Space filling curves have also been used in image compression [44], bandwidth reduction [45], representation of quadtrees [46] and as indexing method for faster disk reads [47].

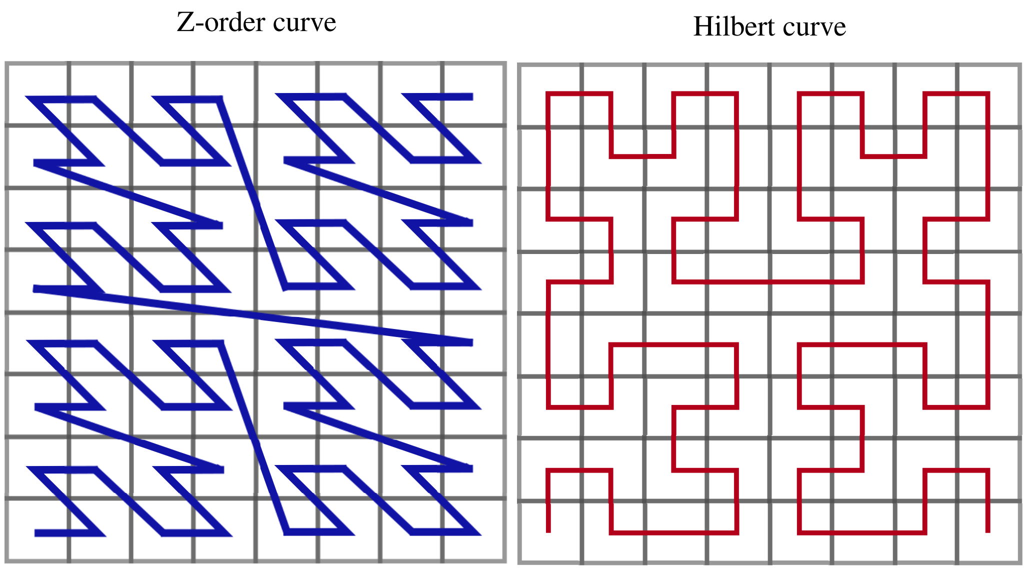


Figure 6: Hilbert curve and the z-order curve are two most commonly used space filling curves in computer science.

Properties that space filling curves may have

- *Proximity preserving*: Points which are close to each other in multidimensional space are also close to each other on the curve.
- *Self similar*: The same pattern is repeated on different scales. For example, in

Figure 6a the same mirror Z -shape is repeated in three different scales.

- *Non-self-crossing*: Curve does do not cross itself.
- *Bijective*: Every point in the discrete multidimensional space is mapped to one distinct value on the curve which can be converted back to the multidimensional value.

Proximity preserving qualities

The most important feature that space filling curves may have is their *proximity preserving quality*. This means that points that are close to each other in some multidimensional space are also likely to be close to each other on the curve. This quality has also been called *clustering property* [48] and *distance preserving quality* [40]. To the best of our knowledge, no formal definition for this quality has been given.

We give the following definition for the proximity preserving quality of one dimensional mappings. It is inspired by the definition given for locality sensitive hash functions in [32]. Our definition is not limited to space filling curves but can be applied to other one dimensional mappings. Let Pr denote probability, d distance function in the multidimensional space and s a mapping of point p to the curve. Then one dimensional mapping s is proximity preserving for a data set P if for any $q, p_1, p_2 \in P$

$$\begin{aligned} |s(q) - s(p_1)| &< |s(q) - s(p_2)| \\ \Rightarrow Pr(d(q, p_1) < d(q, p_2)) &> Pr(d(q, p_1) > d(q, p_2)) \end{aligned} \tag{3}$$

In other words, if q is closer to p_1 than p_2 on the curve s, it implies that q is more likely to be closer to p_1 in the multidimensional space. The higher the probability that q is closer to p_1 in the multidimensional space, the more proximity preserving s is.

The amount of the proximity preserving quality (PPQ) could be defined as the expected difference between these probabilities:

$$PPQ = E[Pr(d(q, p_1) < d(q, p_2)) - Pr(d(q, p_1) > d(q, p_2))] \tag{4}$$

Equations 3 and 4 are theoretical and not intended as practical ways to determine if a one dimensional mapping is proximity preserving. Calculating an absolute PPQ value would require considering all possible 3-tuples (q,p_1,p_2). However, random sampling might provide sufficient approximation of PPQ.

Z-order curve

The *z-order curve* (Figure 6a) is a function which maps multidimensional points to one dimension by interlacing the bits of the binary representation of the vector components. This one dimensional value is referred to as *z-value*. When multidimensional points are ordered by their z-values this order is called *z-order* and it is demonstrated in Figure 8.

The z-ordering has been independently discovered by several authors. According to Faloutsos and Roseman [40] and also Mokbel and Aref [49] the equivalent of the z-order curve was first introduced by Peano [50] in 1890. To the best of our knowledge, the first use of z-values in computer science was in 1966 by Morton [51] who used them for file sequencing of geographical data. Tropf and Herzog [38] used it for range searches, calling it *bitwise interlacing*. The term z-order was first introduced by Orenstein [39] who used the curve for range searches. Essentially the same concept has been also referred to as *quad code* [52].

The calculation of a z-value is shown in Figure 7. The vector components are first converted to binary representation. Then the bits of the binary representation are interleaved. Finally, the resulting binary string is interpreted as an integer number which we refer to as z-value. For example, the two dimensional vector $(3,5)$ can be converted to either z-value 27 or 39 depending on the permutation of dimensions in the bit interleaving.

$$\boxed{\begin{array}{lll} & & \text{interleaved bits} \downarrow \\ (3,5) = (011_2, 101_2) & \rightarrow & 01\ 10\ 11_2 = 27 \\ & \rightarrow & 10\ 01\ 11_2 = 39 \\ \uparrow \text{2D vector} & & \text{z-value} \uparrow \end{array}}$$

Figure 7: z-value calculation for a 2-dimensional vector

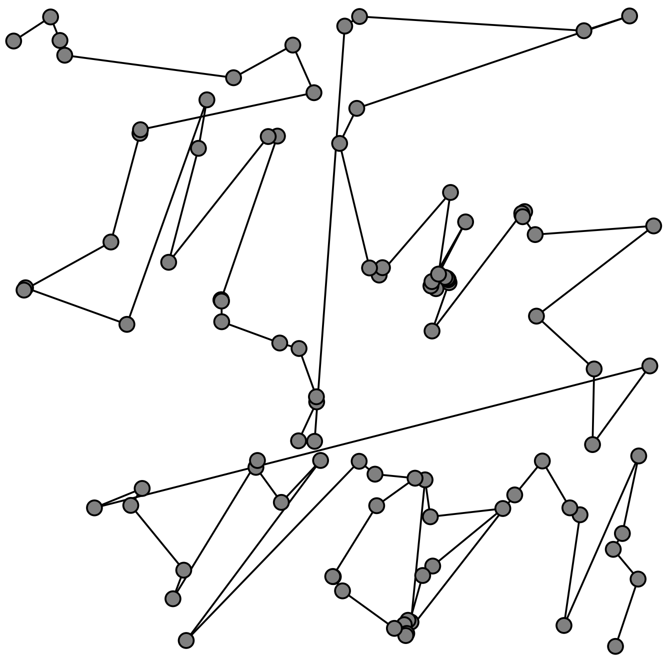


Figure 8: Two dimensional set of points ordered by their z-values.

Comparison of different curves

In the Hilbert curve (Figure 6b), consecutively ordered points are always adjacent in space [53]. In comparison, in the Z-order curve (Figure 6a) there exists "jumps" where the real distance between consecutively ordered points can be high. For example, in two dimensional case the worst case real distance between two consecutive points can be almost the width of the mapped area ($(4,1) \rightarrow 0111_2$ vs $(0,3) \rightarrow 1000_2$). These jumps can have significant effect on quality. However, this can be compensated by using multiple z-order curves.

Searching for k nearest neighbors using z-order

Several methods use space filling curves to find results for the k nearest neighbor problem [41, 42, 43, 24] and nearest neighbor search [40, 41]. The general principle used in these methods is described in Figures 9-12 and in Algorithm 1.

First in a preprocessing step a search index is created. This is done by generating z-values for all data points and sorting the points by their z-values. To improve accuracy of searches, multiple different z-orders can be used by first randomly shifting or rotating the point set. Multiple orderings are useful because one linear ordering can preserve proximity for some points but not for all points.

For a given query point q, the k nearest neighbor search is executed starting from the z-value of the query point q. The position of q on the curve can be found in $O(\log(N))$ time using binary search. For each linear ordering (Figures 9 and 10), the search finds on both directions $\alpha \cdot k$ points that are nearest to the query point q along the z-order curve. The results from these multiple searches are combined (Figures 11) to a set S, and the actual distance is calculated for all points in S. The k points that are closest to the query point q are selected from S.

This search takes $O(k \cdot \alpha \cdot m)$ distance calculations and gives approximate k nearest neighbors. The quality of the approximation can be increased by increasing the variable α or by increasing the number of different linear orderings m.

The z-order curve has also been used to find exact nearest neighbors [24, 7]. This can be done using a range search method very similar to that introduced by Tropf and Herzog [38]. The range search takes as input the query point q and range R which is the radius of the kNN ball of the approximate results (Figure 11).

The search (Figure 12) finds all points within a box with lower left and top right corners $L = q - (R, ..., R)$ and $T = q + (R, ..., R)$. The z-values of all points inside the box (which cointains also the exact results) are guaranteed to be between the z-values of the corners of this box. A proof for this is given in [24]. Therefore, the exact result can be found by a search which starts from the query point and continues forward along the curve until it finds a point p_t for which z_value(p_t) $\geq$ z_value(T). The same is repeated to the other direction.

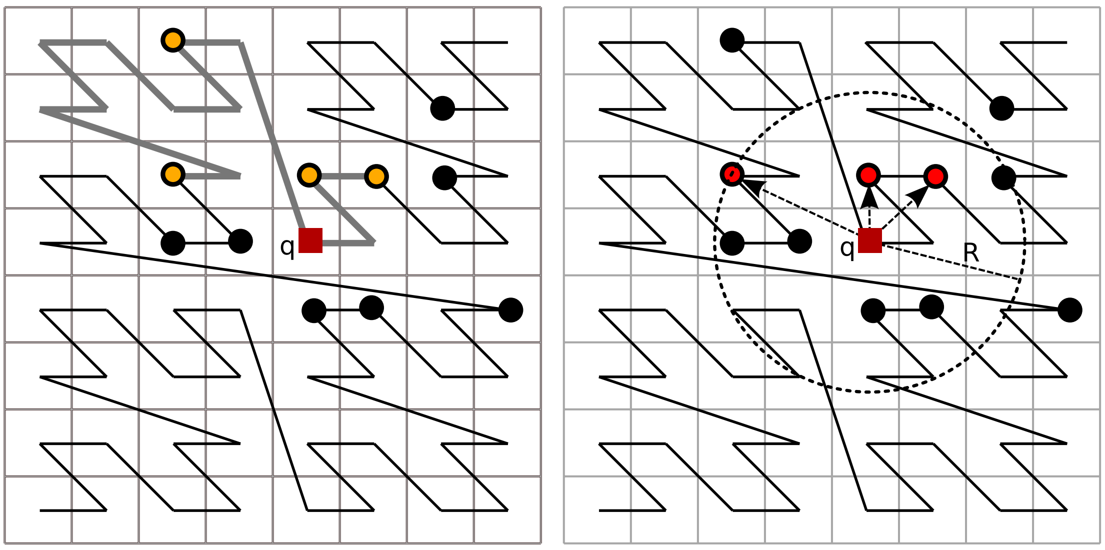


Figure 9: In this example the goal is to find $k = 3$ nearest points for the query point q (red rectangle) by searching for 2 nearest points ($\alpha = 2/3$) on both directions along the curve. The orange circles on the left represent candidate points. The red circles on the right represent the approximate results.

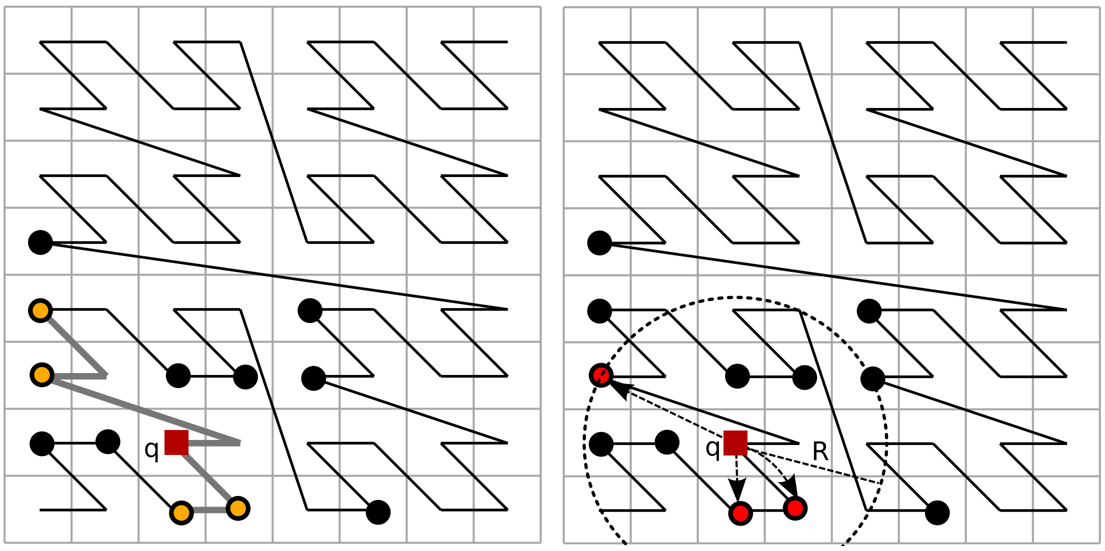


Figure 10: The process in Figure 9 is repeated here for the same data set after it has been shifted by adding random vector $(-2, -3)$ to the query point and to all data points.

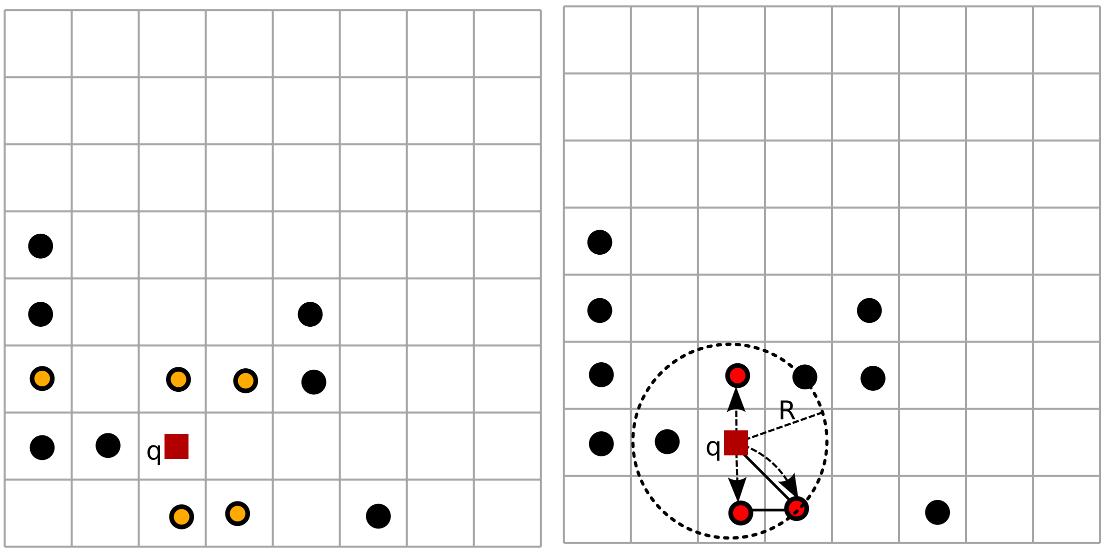


Figure 11: The results from Figures 9,10 are combined to provide better approximate result.

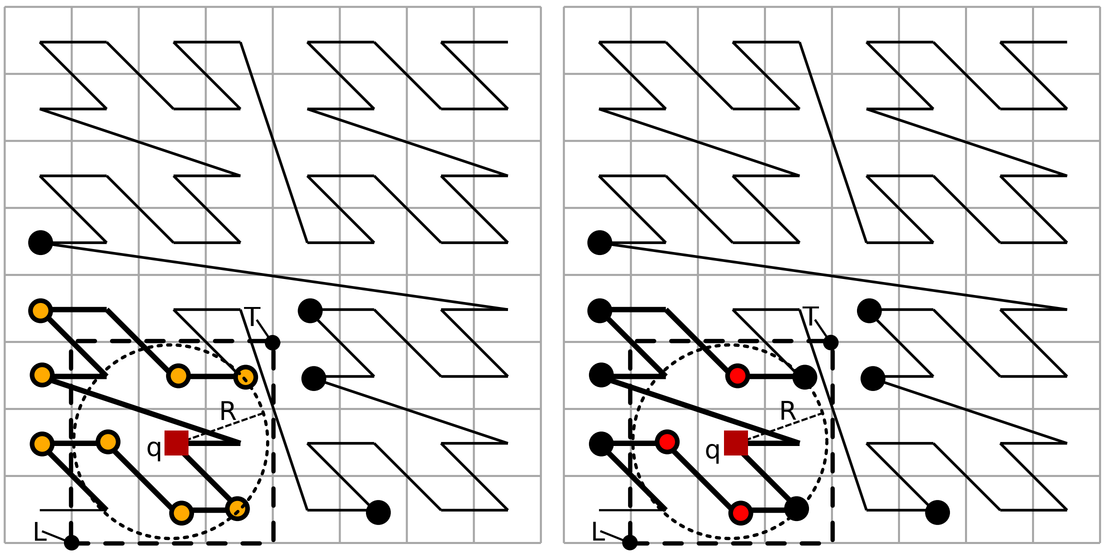


Figure 12: The exact results can be found by doing a range search for q with range (R) set to radius of the kNN ball of the approximate results (Figure 11). The coordinates of the lower left and top right corners of the bounding box are $L = q - (R, R)$ and $T = q+(R, R)$ respectively. The exact results are guaranteed to be found by examining all points between L and T along the curve.

Algorithm 1 Approximate kNN with z-order curve

```
 1: procedure CREATEZINDEX($P$,$m$)
 2:     Calculate $m$ different random vectors $H = \{h_1, h_2, ..., h_m\}$
 3:     for all $h_i \in H$ do
 4:         for all $p_j \in P$ do
 5:             $z \leftarrow$ Z_VALUE$(p_j + h_i)$                          ▷ See Figure 7
 6:             $Z[i][j] \leftarrow (j, z)$
 7:         end for
 8:         $Z[i] \leftarrow$ Sort so that $\forall p_j \in P, \quad Z[i][j][1] \leq Z[i][j+1][1]$  ▷ Sort based on
    z-values
 9:     end for
10: end procedure
11:
12: procedure SEARCHZORDERKNN($q$,$k$,$\alpha$)
13:     $S \leftarrow \emptyset$
14:     for all $h_i \in H$ do
15:         $q_z \leftarrow$ z_value($q + h_i$)
16:         Using binary search, find $l$ so that $Z[i][l][1] \leq q_z \leq Z[i][l+1][1]$
17:         for all $(j, z) \in \{Z[i][l - k \cdot \alpha], \ldots, Z[i][l + k \cdot \alpha]\}$ do
18:             $S \leftarrow S \cup \{p_j\}$
19:         end for
20:     end for
21:     $S \leftarrow$ Sort so that $\forall p_j \in S, \quad d(q, S[j]) \leq d(q, S[j+1])$
22: return $\{S[0], \ldots, S[k-1]\}$
23: end procedure
```

Faloutsos [40] used a method very similar to Algorithm 1 already in 1989 for the nearest neighbor problem using just one curve. They also compared Hilbert and Z-order -curves and found Hilbert to provide more accurate results.

Megiddo and Shaft [41] used Gray codes to provide distance preserving one dimensional mapping for kNN search in relational databases. They used multiple different mappings where each mapping was made different by randomly permuting the dimensions and shifting points by random vector. Instead of choosing fixed number of closest points along the curve, they selected points within a threshold δ. The δ variable was then gradually increased until the search returned enough results. Their method was tested with up to 324 dimensional data set.

The kNN graph can be constructed by executing SEARCHZORDERKNN for all points. However, this way all distance calculations would be executed twice. For example, if points p_1 and p_2 are consequtive points in the z-ordering, $d(p_1, p_2)$ would be calculated when running SEARCHZORDERKNN(p_1, k, α), and $d(p_2, p_1)$ when running SEARCHZORDERKNN(p_2, k, α). To avoid this problem, Connor *& al.* [7] used a sliding window technique which selects $k \cdot \alpha$ nearest points on the curve in only one direction.

4.3 Neighborhood propagation

Neighborhood propagation is a method to construct (by starting from random graph) or improve an approximate kNN graph by comparing each point of a graph with its neighbors' neighbors. It is based on the observation that if a point y is a neighbor of x and point z is neighbor of y, then z is also likely to be a neighbor of x.

Neighborhood propagation has been used in many different methods to construct the kNN graph [11, 12, 13, 8]. The *Nearest Neighbor Descent* (NN-Descent) [12] algorithm constructs the kNN graph solely by using neighborhood propagation. Other methods use neighborhood propagation only as a post processing step to improve the quality of graph after main algorithm. Chen *& al.* [11] uses neighborhood propagation to refine the graph after conquer step in their divide-and-conquer method. Wang *& al.* [13] uses multiple random divide-and-conquer. Zhang *& al.* [8] uses locality sensitive hashing (LSH).

The basic principle is shown in Algorithm 2, and a visual example is presented in Figure 13. The algorithm takes as parameter a set of points P and an initial approximation of k nearest neighbors for each point. The initial approximation can consist of just random points chosen from P or it can be output from some other non-exact kNN graph algorithm.

The algorithm iteratively improves the quality of the graph. On each iteration, the neighbors of neighbors are selected for each point $p \in P$. If any of the neighbors of neighbors of p is closer than the current neighbors for the point, its neighbor list is updated accordingly. The algorithm is iterated until a specified stop condition is met.

Algorithm 2 PropagateNeighborhood

```
procedure PROPAGATENEIGHBORHOOD($P$, $kNN$) $\rightarrow kNN$
    repeat
        for all $p \in P$ do
            for all $Neighbor \in kNN(p)$ do
                for all $x \in kNN(Neighbor)$ do
                    UPDATENEIGHBORLIST$(p, x, kNN)$
                    UPDATENEIGHBORLIST$(x, p, kNN)$
                end for
            end for
        end for
    until stop condition met
    return $kNN$
end procedure

procedure UPDATENEIGHBORLIST($p$, $x$, $kNN$)
    if $d(p, x) < max\{d(p, y)|y \in kNN(p)\}$ then
        Insert $x$ into $kNN(p)$
        Remove the item with largest distance from $kNN(p)$
    end if
end procedure
```

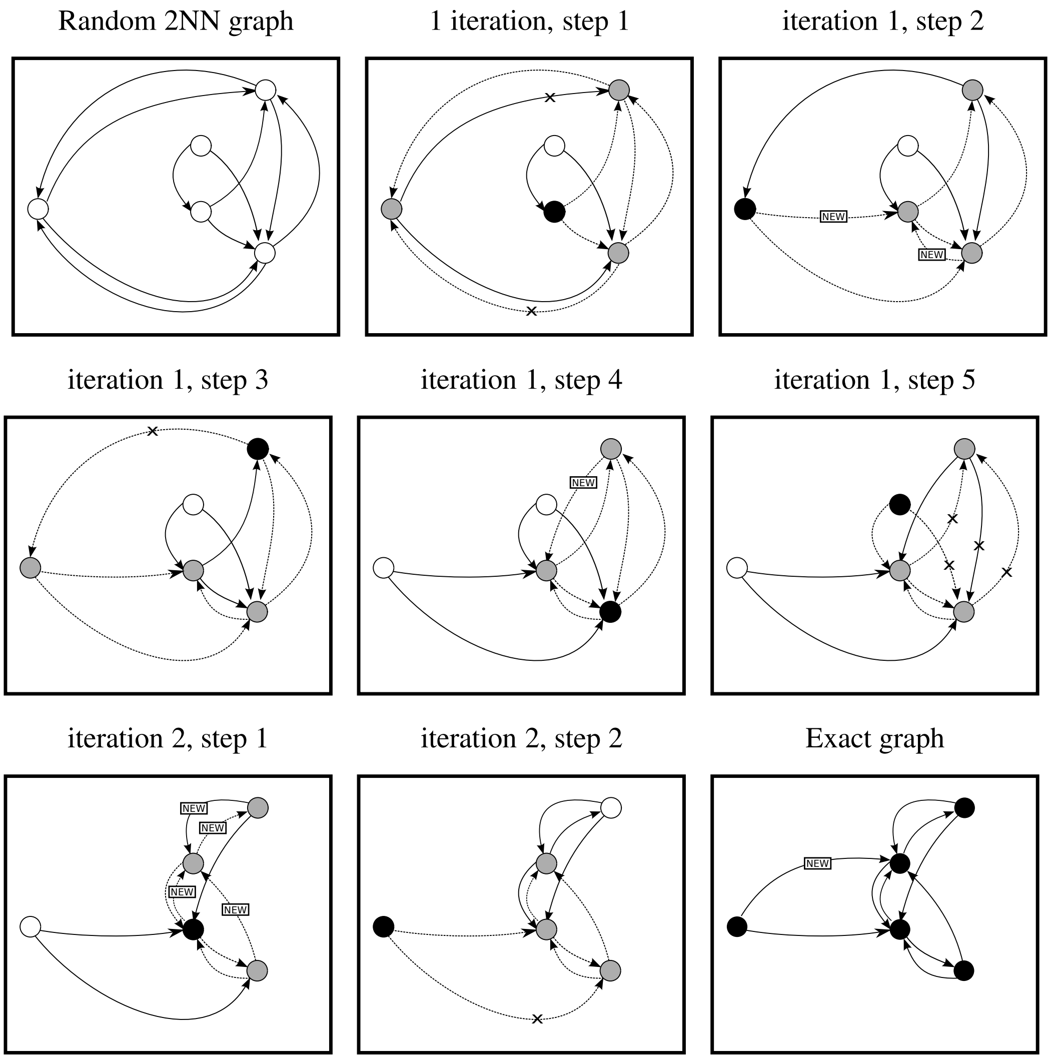


Figure 13: The nearest neighbor propagation creates an exact 2NN graph starting from a random graph. Black circle represents current node. Gray nodes represents current node's neighborhood which consists of its neighbors and neighbors of neighbors. White circle represents points outside current nodes neighborhood. On each step, the exact distance is calculated between current node (black cicle) and each of the gray circles, and the edges of those nodes are updated if the exact distance is smaller than largest known distance for that node. Edges that are removed in current step are marked with **X**. New edges created in last step are marked with **NEW**.

Since each point has k^2 neighbors of neighbors, and distance is calculated to each of them, the propagation requires $O(k^2N)$ distance calculations per iteration. Assuming that the time complexity of a distance calculation is linear with respect to the number of dimensions D, the total time complexity of the method is $O(k^2NID)$ where I is the number of iterations.

Each of the surveyed methods differ from the previously described basic neighborhood propagation method. Algorithms by Chen *& al.* [11] and Zhang *& al.* [8] use the most simple variant with just one iteration of neighborhood propagation where $kNN(x)$ is updated by selecting k smallest items from $kNN(x) \cup \{\cup_{y \in kNN(x)} kNN(y)\}$.

NN-Descent

NN-Descent algorithm [12] uses neighborhood propagation just for random points as the initial graph. The algorithm stops when the number of changed edges with respect to the number of all edges in the graph drops below user provided threshold parameter δ. The algorithm can also be limited to a specific number of iterations.

In addition to using the neighbors of a point ($\mathrm{kNN}(p)$), the NN-Descent algorithm also propagates to the reverse direction by using $\mathrm{kNN}_r(p) = \{y \in P | p \in \mathrm{kNN}(y)\}$. When neighbors of neighbors are selected, NN-Descent uses the union of kNN and kNN_r.

To allow control on the quality of resulting graph, the NN-Descent algorithm uses a sampling factor $\lambda \in [0, 1]$ so that only a fraction of λ randomly chosen neighbors are considered. In effect, this reduces the time complexity of the algorithm from $O(k^2 NID)$ to $O((\lambda k)^2 NID)$. Lowering the λ -value may, in theory, also increase the number of iterations needed for convergence, but in practice it reduces the running time at the cost of quality.

Wang *& al.*'s method

The method by Wang *& al.* [13] differs from other methods in that it takes advantage of the fact that a neighbor of a close neighbor is more likely to be a neighbor than a neighbor of a more distant neighbor. This is done by constructing a priority queue for a point p so that the nearest neighbor is at the top. Then a point is iteratively popped from the queue and all its unvisited neighbors are pushed into the queue.

Wang *& al.*'s method has a different stop condition. Because it uses priority queue, the algorithm will stop when the queue becomes empty, and a parameter T is used to limit the propagation to a maximum number of distance calculations.

In comparison to Algorithm 2, it will omit some distance calculations to neighbors of more distant neighbors which are less likely to be true neighbors. Therefore, it is expected to find true neighbors of p faster. However, no empirical comparison between this and other neighborhood propagation methods have been made, so it is unclear if it produces actual speed or quality benefits.

Dimensionality

To estimate the effect of dimensionality to the performance of the neighborhood propagation method we calculated the probability of a neighbor of a neighbor being a neighbor using a random uniformly distributed artificial data set of size $N = 10000$ for each dimension. Dimensionality was varied from 1 to 150, and neighborhood size was set to $k = 10$. The experiment was repeated with four different random data sets and the average is displayed in Figure 14.

The probabilities range from $p = 66\%$ ($D = 1$) to $p = 1.4\%$ ($D = 150$). As dimensionality increases the empirically measured probability seems to approach $k/N = 1\%$, which is also the probability that a randomly chosen point p is among the k nearest neighbors of another point q. In other words, the probability of a neighbor of a neighbor being a neighbor approaches the probability of a randomly chosen point being a neighbor when dimensionality increases.

This suggests that with uniformly distributed data, the efficiency of the neighborhood propagation method decreases towards the efficiency of the $O(n^2)$ brute force algorithm when dimensionality increases towards infinity. However, we did not test this with non-uniformly distributed data. Real data sets are usually not uniformly distributed, so further tests would be needed to before making conclusions about real data.

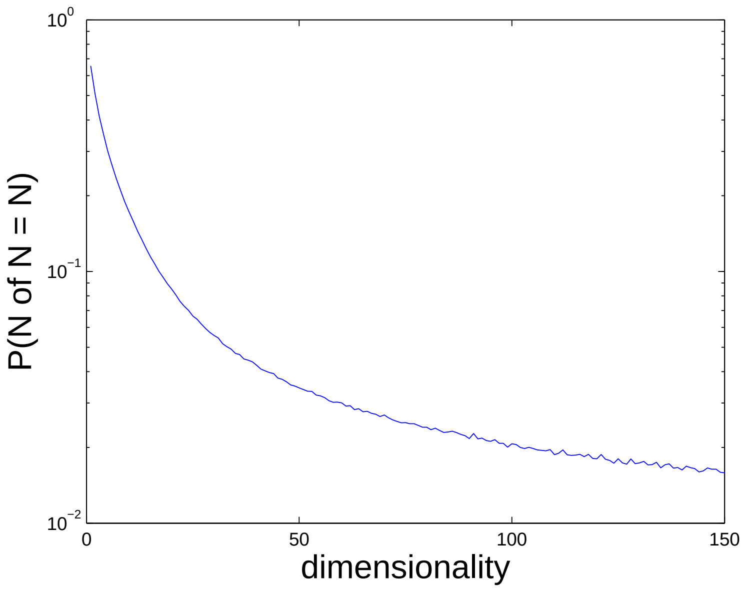


Figure 14: Probability of a neighbor of a neighbor being a neighbor as a function of dimensionality.

5 Z-order neighborhood propagation

In this chapter we present our method for constructing the k-nearest neighbor graph. Our method is targeted for high dimensional data sets and the L_2 distance metric. It has two variants. We call the first variant *Z-order nearest neighbors* (ZNN). It uses a space filling curve called the z-order curve to construct the graph. We call the second variant *Z-order neighborhood propagation* (ZNP). It is a combination of the Z-order nearest neighbors -method and neighborhood propagation.

Algorithm 3 ZNP

```
Input: Data set of points $P$
       Number of nearest neighbors $k$
       Number of different 1-D mappings $N_c$
       Dimensionality of the z-order curve $D_z$
       Width of sliding window $W$.
Output: $k$NN graph
1: procedure ZNP( $P$, $k$, $N_c, D_z, W$)
2:     kNN ← ZNN($P$, $k$, $N_c, D_z, W$)
3:     kNN ← PropagateNeighborhood($P$, kNN)
4: end procedure
```

The ZNP algorithm first uses the Z-order nearest neighbors search to calculate a graph with a modest accuracy. Neighborhood propagation is then used to improve the graph. For this part we use the NN-Descent algorithm [12]. The NN-Descent algorithm continues iterating until the graph reaches a stable state where new iterations do not bring any improvements on the approximation quality.

The motivation for this approach is that, the Z-order nearest neighbors search gives rough approximation (of around 50% recall rate) very fast. However, when continuing the search further, the quality improves slowly. On the other hand, the NN-Descent algorithm, because it starts from random kNN graph, gives usually very bad results on the first 2-3 iterations. But after that the quality improves very quickly. Therefore, it makes sense to combine these two approaches so that we first execute search along the z-order -curve and then continue with the NN-Descent algorithm.

5.1 Z-order search

Figures 15 and 16 give a visual example of a 2NN-graph construction for a two dimensional data set P with $N_c = 7$, $N = 60$ and $W = 3$. On the left side is a one dimensional mapping calculated for the rotated set of points. On the right side there is the 3NN graph created (or improved) using the one dimensional mapping. White circles represent points which neighbors are correct, and black rectangles represent points for which the search did not find all k nearest neighbors correctly. In this example, there was no further improvement on iteration 6, and the exact 2NN-graph was calculated in the seventh iteration. However, there is no guarantee that the algorithm returns exact results on any amount of iterations.

Algorithm 4 ZNN

Input:

Data set of points P

Number of nearest neighbors k

Number of different 1-D mappings N_c

Dimensionality of the z-order curve D_z

Width of sliding window W.

Output: kNN graph

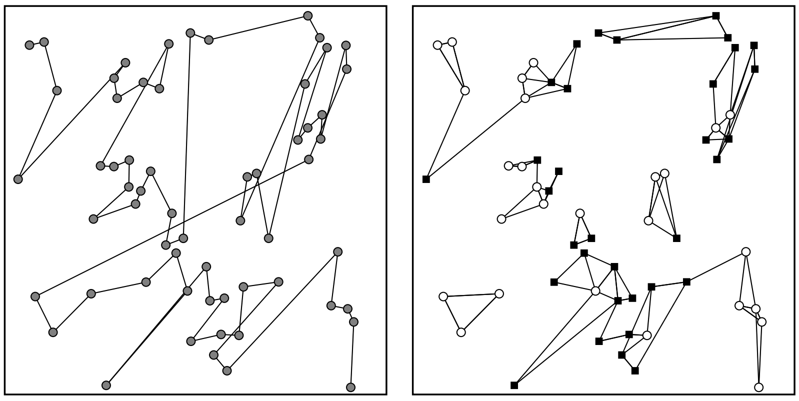
.

```
1:  procedure ZNN( $P$, $k$, $N_c, D_z, W$)
2:      for all $i \in \{1, \dots, N_c\}$ do
3:          $S \leftarrow$ ProjectTo1D($P, D_z$);
4:          $P \leftarrow$ Sort so that $\forall p_j \in P, \quad S[j] \leq S[j+1]$      ▷ Sort based on z-values
5:          for all $p_j \in P$ do                    ▷ Scan points using a sliding window
6:              for all $x \in \{p_{j-W}, \dots, p_{j+W}\} \setminus p_j$ do
7:                  UpdateNeighborList($p_j, x$, kNN)
8:              end for
9:          end for
10:     end for
11: end procedure
12:
```

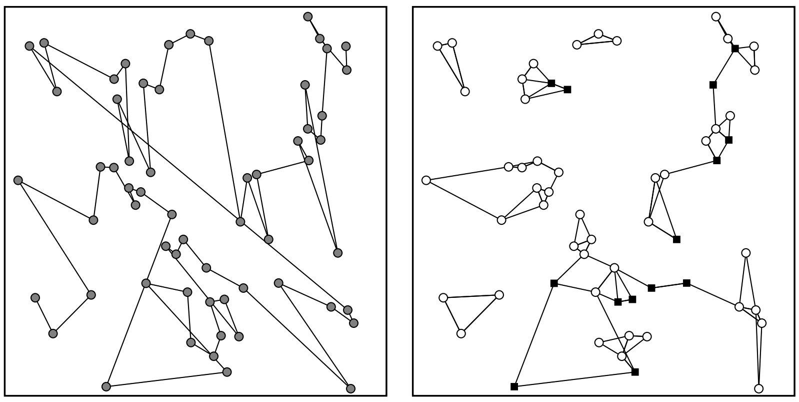
Algorithm 5 ProjectTo1D

```
Input:
  Data set of points $P$
  Dimensionality of the z-order curve $D_z$
Output:
  One dimensional projection $S$
  procedure ProjectTo1D($P, D_z$)
      $h \leftarrow$ Random vector of size $D$
      $P \leftarrow$ Scale $P$ to positive integer values
      $DP \leftarrow (0, ..., D-1)$                ▷ Create random permutation of dimensions
      $DP \leftarrow$ Shuffle$(DP)$
      for $i \leftarrow 0, N-1$ do
          $p \leftarrow$ Zero vector of size $D_z$
          for $j \leftarrow 0, D-1$ do                ▷ Reduce dimensionality from $D$ to $D_z$
              $p[j \bmod D_z] \mathrel{+}= P[i][DP[j]] + h[j]$
          end for
          $S[i] \leftarrow$ z_value($p$)
      end for
  return $S$
  end procedure
```

Iteration 1: Exact results for 45% of points.



Iteration 2: Exact results for 76% of points.



Iteration 3: Exact results for 93% of points.

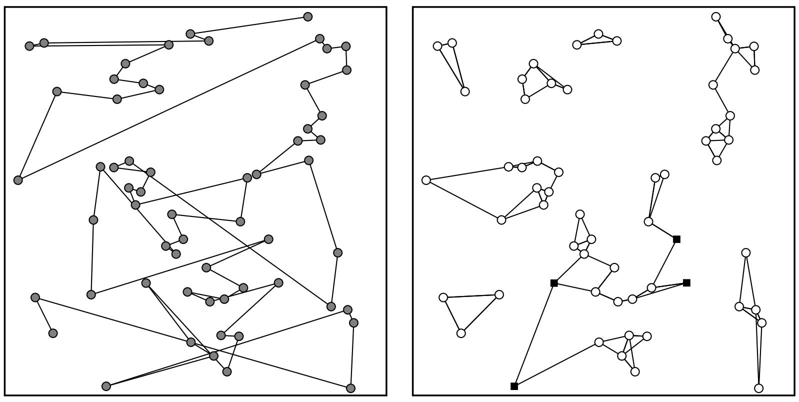


Figure 15: 2NN-graph construction using ZNN-algorithm with parameters $W = 2$ and $N_c = 7$ (part 1/2). On the left side is a one dimensional mapping calculated for the rotated set of points. On the right side is the 2NN graph created or improved using the one dimensional mapping. White circles represent points which neighbors are correct and black rectangles depict points with some errors in the edges.

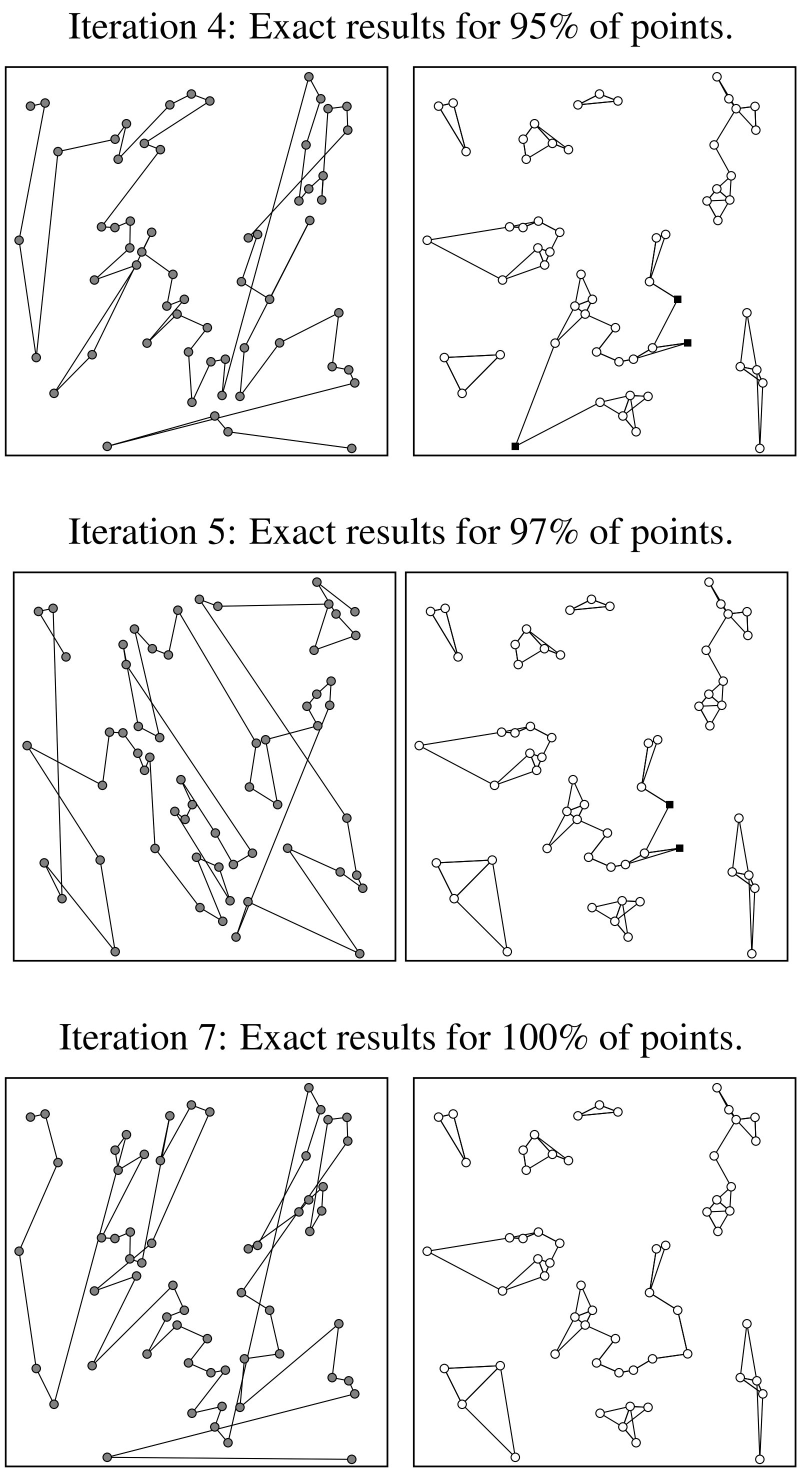

Iteration 4: Exact results for 95% of points.
Iteration 5: Exact results for 97% of points.
Iteration 7: Exact results for 100% of points.

Figure 16: 2NN-graph construction using ZNN-algorithm with parameters $W = 2$ and $N_c = 7$ (part 2/2).

The z-order -curve was chosen as basis of our search method over other alternatives such as Hilbert curve because of four reasons: (1) It provides good proximity preserving qualities. (2) It generalizes trivially for higher dimensions. (3) It is simple to implement (4) Generating z-values can be done fast in $O(D)$ time.

Most methods use either the z-order or Hilbert curves because they have the best proximity preserving qualities [24, 43]. Therefore, those were the main candidates for our method. Some sources have claimed the Hilbert curve to have better proximity preserving quality than the z-order -curve [40, 53]. However, it is unclear if these results generalize to higher dimensions. Also, to the best of our knowledge, no efficient method exists for mapping high dimensional points to Hilbert curve. For example, the state diagram method described by Lawder [54] requires at least $O(D \cdot 2^{D-1})$ memory. By contrast, projecting points on z-order -curve can be done simply by interleaving the bits of vector components, which takes $O(D)$ time and memory. Because we aim to provide solutions especially to high dimensional data sets, we therefore chose the z-order curve in our study.

Many different methods exists for calculating z-values for multidimensional points. We first used a simple method based on for loops where interleaving is done bit-by-bit for each dimension. However, this is very slow and became one of the bottlenecks of our algorithm. Different implementations for z-value generation have been compared in [55] and by J. Baert[1]. We use the lookup table -method implementation presented by J. Baert[1] for 3-dimensional points as a model for our implementation of the z-value generation. We made a more general implementation (see Appendix I) of the lookup table -method so that it works with arbitrary number of dimensions and varying bit-lengths. As a result, the speed of our algorithm improved considerably and z-value calculations were no longer a bottleneck.

Other methods that have used space filling curves for kNN graph construction were limited to a low number of dimensions [7]. Applying space filling curves for higher number of dimensions can be problematic. One of the problems in using the z-order curve is that the space and time constraints grow linearly with D. This is because the z-values are calculated by interleaving the bits of the binary representations of all vector dimensions. For example, a data set with dimensionality $D = 1000$, bit-length $b = 32$ bits per dimension and size of $N = 1000000$ points, the z-values would need to be represented by $D \cdot b = 32000$ bit integers. Since modern processors can handle up to 64 bit integers, additional multiple precision libraries are needed to handle such

[1]Baert, J., "Morton encoding/decoding through bit interleaving", October 2013, http://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/

large numbers.

Additionally, memory needed for generating and sorting D-dimensional z-values would be $N \cdot D \cdot b/8 = 4$ GB. Storing the z-values for all points is not necessary for sorting, but calculating them every time a comparison is made by the sorting algorithm would increase the number of z-value calculations from N to roughly $N \cdot \log(N)$ (number of comparisons in sorting). Consequently, this would increase the running time of the algorithm, especially for high dimensional data where z-value calculations are more time-consuming.

5.2 Dimensionality reduction

We introduce next a simple but effective dimensionality reduction technique to avoid the aforementioned problems with high dimensionality. It is inspired by the mean-distance ordered partial search (MPS) method introduced in [56], which was used to construct the kNN-graph in [2].

We reduce dimensionality of data points from D to D_z before projecting to z-order curve. We do this for each curve by dividing the dimensions into D_z random subsets with roughly equal sizes and then dividing each vector into subvectors corresponding to the subsets of the dimensions. Each subvector is then mapped to one dimension by projecting them to the diagonal axis. The one dimensional mappings of the subvectors are then combined to one D_z dimensional vector for which a z-value is calculated in the normal way of bit interleaving.

For example, given a six dimensional ($D = 6$) vector $p = (5, 4, 7, 0, 3, 2)$, $D_z = 3$ and permutation of dimensions $DP = (4, 5, 6, 1, 2, 3)$ coordinates of point p would be first reordered to $p' = (0, 3, 2, 5, 4, 7)$ and then mapped to a three dimensional vector $(0 + 3, 2 + 5, 4 + 7) = (3, 7, 11) = (0011_2, 0111_2, 1011_2)$. After that, the bits of this vector $(0011_2, 0111_2, 1011_2)$ are interleaved to produce a z-value of 001010111111_2 $= 703$.

An additional benefit of using this kind of dimensionality reduction is that it can be used to avoid resorting to multiple precision arithmetic. Because single precision arithmetic is faster than multiple precision arithmetic, the use of dimensionality reduction can result in faster running time. Even when multiple precision arithmetic is used, the dimensionality reduction can be used to keep the bit-length of the data type on reasonable levels.

For multiple precision arithmetic we use Boost.Multiprecision C++ library. We use 1024-bit unsigned int data type (uint1024_t) to represent the z-values. This allows to use 32-bit integers to represent vector components and to calculate z-values for up to 32 dimensional points without using dimensionality reduction ($32 \times 32 = 1024$).

The ZNN algorithm uses three kinds of randomization. First, random shifting of points is performed by calculating a random vector for each curve and then adding this vector to each point. Secondly, in dimensionality reduction the dividing of vectors to subvectors is randomized. Thirdly, the order of dimensions in bit interleaving is randomized.

5.3 Configuration variables

The ZNN -algorithm has three different configuration variables:

N_c	Number of different 1d mappings.
W	Width of sliding window.
D_z	Dimensionality of the z-order curve.

Suitable values for these variables depend on many factors such as targeted quality and the characteristics of the data set. Choosing the values that give optimal performance may not be possible without first running the algorithm. To make the task of choosing the values easier, we propose the following simple rules.

Given a quality control parameter $\gamma \in]0, 1[$, k, dimensionality of data set D, maximum dimensionality allowed by implementation D_{max} and number of points N, the values for N_c, W and D_z can be chosen as follows:

$$N_c = \lfloor \log_{1/\gamma}(D) + 1 \rfloor$$
$$W = \lfloor k/2 + \log_{1/\gamma}(N) \rfloor$$
$$D_z = \min\{D, D_{max}\}$$

As will be shown in Section 6.5, the recall rate of our method increases logarithmically with respect to the running time of the algorithm. The quality control parameter γ was created to provide a parameter that would increase the running time exponentially so that the resulting recall rate would be approximately the same as γ.

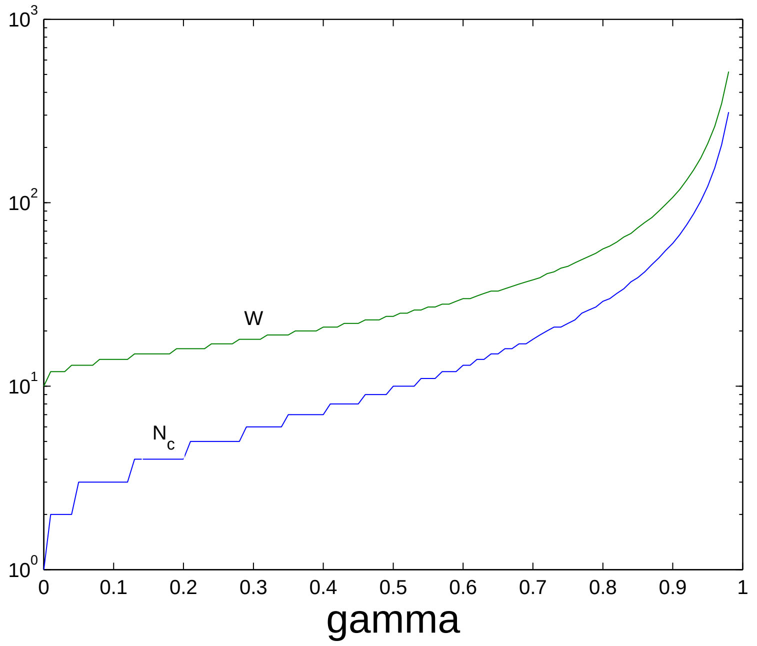


Figure 17: The effect of variable γ to N_c and W. $N = 28775, k = 20, D = 544$.

The rules were chosen based on random search experiments performed to find optimal configuration parameters. For the configuration variable D_z, the optimal value was the maximum value allowed by our implementation. The implementation was limited by our choice of using 1024 bit integers for z-values and 32 bit unsigned int to represent vector components. Therefore D_{max} was $1024/32 = 32$. With higher D_{max}, it might not be an optimal value for D_z because time and space constraints grow linearly with D_z.

Another goal of the rules is to provide control for the ZNP method on how much the ZNN algorithm is be applied and when to switch to use the neighborhood propagation method. When γ is set to 0.5 the rules provide roughly those values that gave the best performance for the ZNP algorithm in our optimal configuration parameter random search experiment.

Table 1: Example values for used data sets.

Dataset	N	D	γ	N_c	W	D_z
Corel	662317	14	0.5	4	29	14
Shape	28775	544	0.5	10	24	32
Audio	54387	192	0.5	8	25	32
Corel	662317	14	0.9	26	137	14
Shape	28775	544	0.9	60	103	32
Audio	54387	192	0.9	50	113	32

6 Experiments

This section presents the methodology and results for our experiments. We evaluate the performance of our ZNN and ZNP methods in comparison to two existing algorithms on Euclidean distance metric and three data sets with with dimensionality ranging from 14 to 544. We use *recall rate* to measure the quality of the kNN graph and program execution time to measure speed. Additionally, in Section 6.6, we study the effect of our dimensionality reduction method on the quality of results.

6.1 Evaluated algorithms

We compare our ZNN and ZNP algorithms against two existing algorithms: NN-Descent [12] and Recursive Lanczos Bisection [11]. We use the following abbreviations for the chosen algorithms:

ZNN	Z-order -curve nearest neighbors (proposed)
ZNP	ZNN + neighborhood propagation (proposed)
NNDES	Nearest Neighbor Descent (NN-Descent) [12]
RLB_g	Recursive Lanczos Bisection, variant *glue* [11]
RLB_o	Recursive Lanczos Bisection, variant *overlap* [11]

We used a NNDES -implementation made available by authors[2]. For the neighborhood propagation part of our ZNP algorithm we used the implementation of NNDES. The implementation of Recursive Lanczos Bisection algorithm was also made available by the authors[3]. It has two variants: *glue* for higher speed but lower quality and *overlap* for higher guality and lower speed.

6.2 Measurements

We measure *recall rate* and *program execution time* to evaluate the performance of the algorithms. Recall rate, or accuracy, measures the quality of the approximate kNN-

[2]https://code.google.com/p/nndes/

[3]http://www.mcs.anl.gov/ jiechen/software/knn.tar.gz

graph G' in relation to accurate graph G by dividing the number of common edges (neighbors) with the number of all edges [12, 11]. It is defined as:

$$recall(G', G) = \frac{|E(G) \cap E(G')|}{|E(G)|} \tag{5}$$

The recall rate ranges from 0 to 1 where value 1 means that the results are equal and 0 that the results are completely different. Instead of recall rate, one could also report $\textit{error rate} = 1 - \textit{recall}$.

Program execution time is measured as single thread execution time. The time needed to load the data set and save the results to file is excluded because it mostly depends on the system I/O performance and does not reflect the efficiency of the algorithm.

6.3 Data sets

We use three different data sets: corel, shape and audio. These data sets are used in [12] and can be found on web [4].

Table 2: Summary of data sets.

Data set	N	dimensionality
Corel	662,317	14
Shape	28,775	544
Audio	54,387	192

The corel dat set consists of features from 68,040 different images, each divided into 10 segments, thus, providing a total of 680,400 data objects. Each segment consists of 14 different features. The Corel data set has been used in [43, 12].

The shape data set contains 28,775 3-D shape models collected from various sources. Features were extracted from each 3-D model.

The audio data set contains features from the DARPA TIMIT collection. It was created by breaking recordings of 6,300 english sentences into smaller segments and extracting features from them. Each segment is treated as an individual object.

[4]https://code.google.com/p/nndes/downloads/list

6.4 Empirical process

We run the algorithms for varying number of k. Recursive Lanczos Bisection (RLB) and Z-order nearest neighbors (ZNP) was run for values $k = 1, 5, 10$ and 20. NN-Descent and ZNP uses neighborhood propagation which does not work for $k = 1$, so they were only run for values 5, 10 and 20. Higher k-values have also been considered, $k = 50$ in [13] and $k = 100$ in [1], but most others use $k \leq 20$.

Our ZNN algorithm has the quality control parameter $\gamma \in]0, 1[$, which is used to calculate appropriate window width and number of curves. We run the experiments for ten different γ values: $(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.925)$.

NNDES has sampling factor $\lambda \in [0, 1]$ and the parameter $\delta \in [0, 1]$. The program stops when the last iteration yielded less than δ amount of changes on the graph. Increasing the δ value decreases the number of iterations. The NNDES algorithm was run for seven different (δ, λ) values:

Table 3: NNDES parameter values selected for experiment

δ	0.82	0.684	0.547	0.411	0.274	0.137	0.001
λ	0.3	0.417	0.533	0.65	0.767	0.883	1.0

The ZNN part of our ZNP algorithm was run with a fixed quality parameter $\gamma = 0.5$. The (δ, λ) values for the neighborhood propagation part were varied in the same way as for the NNDES algorithm.

The RLB algorithm has quality control parameter *overlap size*. The RLB$_g$ and RLB$_o$ were run for six different overlap sizes: $(0.05, 0.1, 1.5, 0.2, 0.25, 0.3)$. Increasing the overlap size increases both recall rate and program execution time. For the the corel data set the program sometimes produced segfault errors with high overlap sizes and the experiment could be run only for small values.

In our experiments, for all algorithms, both speed and quality differed between multiple runs for the same input parameters. For NNDES, ZNN and ZNP this is clearly caused by randomization which is an important part of the algorithms. Therefore we run all experiments ten times for the same input parameters and report average values for the same input.

Every algorithm was run for three different data sets with varying (6-10) quality parameters and for 3-4 different values of k. In addition, each algorithm was run ten times for each distinct input parameters. These sum up to 3528 runs. The resulting text format kNN-graph files sum up to 57 gigabytes in size and were analyzed in GNU Octave for their correctness

All experiments were run on a computer equipped with 8-Core 4.0GHz AMD FX-8350 CPU with 8MB of L2 cache and 8 GB of RAM. The computer was running 64-bit Ubuntu Linux 14.04. Code compilation for all tested algorithms was done using gcc version 4.8.1 with -O3 optimization.

Remarks on algorithm implementations

Our method and the NN-Descent method have been designed for parallel processing. The Recursive Lanczos Bisection code however, has not been parallelized. Therefore, in order to get a fair comparison, we disabled the parallelization in those methods where it is included and run all experiments on a single thread. Parallelization was disabled during compilation and additionally each algorithm was restricted to a single processing core by using taskset command of the util-linux package.

NN-Descent included an implementation of L_2 distance function that was optimized using SSE2 vector extension, which allows to run four floating point operations in parallel. Therefore code run using SSE2 vectorization is up to 4 times faster. Distance calculations are a major bottleneck in the algorithms, so this could cause a significant bias in the results. We therefore disabled the SSE2 optimizations and used a non-optimized L_2 distance function instead.

The Recursive Lanczos Bisection -algorithm was run mainly for data sets SHAPE and AUDIO. For the corel data set we managed to run the RLB algorithm only with small values of k and low quality settings. With other settings the program produced a segfault error or was killed by kernel because of high memory consumption (over 6 gigabytes of RAM). In [12] the Recursive Lanczos Bisection -algorithm was successfully run on the corel data set. Therefore, we expect that there was some difference in the running environments that caused this problem.

The Recursive Lanczos Bisection -implementation reads data sets stored as double values. Our test data was stored as float values so we had to make small modifications to the RLB code to make it able to read the test data. There is a small possibility that these modifications contributed to the aforementioned segfault error although we verified with a debugger that the segfault happened in a part of code that was not directly related to the parts that we modified.

6.5 Recall vs. program execution time

This section presents the time and recall measurements of the selected algorithms and data sets. First the results are presented in graphical plots in Figures 18 - 20. Recall rates and program execution times are then shown in Table 5 after which we will provide detailed analysis of the results.

The results for RLB are blank in cases where the program did not give results with any overlap values. The NNDES and ZNP algorithms could not be run for $k = 1$ and even for $k = 5$ the highest recall rate was not very good (recall $\in [0.651, 0.896]$). The brute force time was only measured for $k = 20$ even when shown in graphs for $k \in \{1, 5, 10\}$.

data set	audio	shape	corel
time	417s	255s	18,889s

Table 4: Time taken for brute force results when $k = 20$.

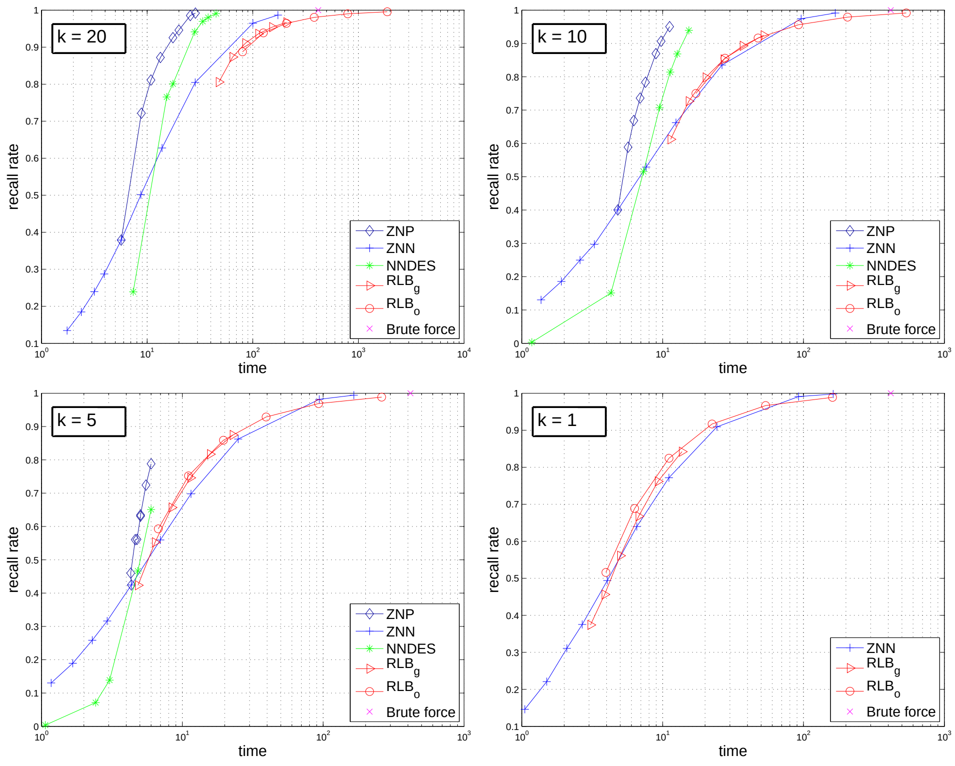


Figure 18: Data set audio

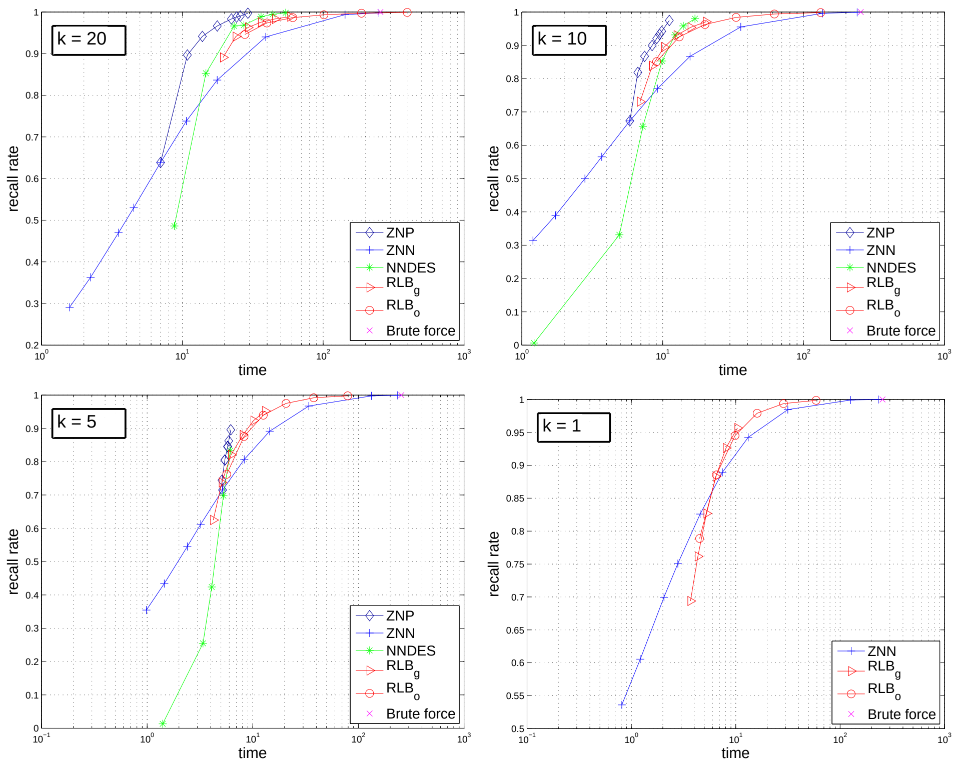


Figure 19: Data set shape

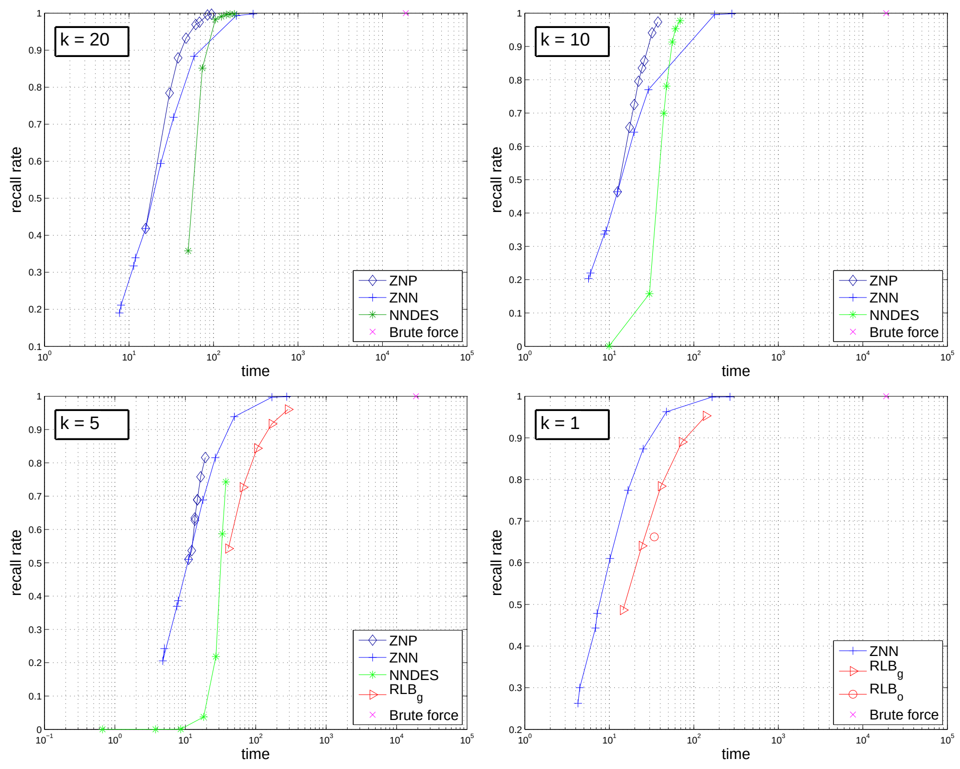


Figure 20: Data set corel

Table 5 show results with high recall rate. For NNDES and ZNP the results reflect quality settings where sampling factor $\lambda = 1.0$, $\delta = 0.001$, and for ZNN quality setting $\gamma = 0.9$. For RLB overlap was 0.25 except for the data set corel, $k = 1$ for which the program crashed with overlap higher than 0.05. For ZNN and RLB chosen quality settings were not the highest tested because increasing the quality settings beyond those points seems to give very little quality increase with quite a large increase in program execution time.

From Figures 18 - 20 and Table 5 it can be seen that the ZNP algorithm is the fastest in almost all tested cases. This is especially clear with $k = 20$, where almost the same recall rate is achieved by both NNDES and ZNP but 36-46% faster with ZNP. With $k = 10$ the results are not as clear. In that case ZNP is still faster, but there is difference in recall. NNDES gives a noticeably better recall rate for data sets corel (0.998 vs. 0.997) and shape (0.998 vs. 0.997) but worse for audio (0.991 vs. 0.992) than ZNP.

With high recall rates the standard deviation σ was very small for all algorithms. For recall > 0.9 the standard deviation was at most 0.0012.

		$k = 20$			$k = 10$		
Dataset	Method	Time	recall	σ(recall)	Time	recall	σ(recall)
audio	ZNP	**29s**	**0.992**	0.00011	**11s**	0.951	0.00051
	ZNN	100s	0.965	0.00112	96s	0.974	0.00045
	NNDES	45s	0.991	0.00013	15s	0.939	0.00031
	RLB$_g$	153s	0.954	0.00025	37s	0.893	0.00065
	RLB$_o$	793s	0.990	0.00010	205	**0.979**	0.00028
shape	ZNP	**29s**	0.997	0.00009	**11s**	0.975	0.00047
	ZNN	143s	0.994	0.00028	137s	**0.996**	0.00032
	NNDES	54s	**0.998**	0.00005	17s	0.980	0.00020
	RLB$_g$	46s	0.983	0.00027	16s	0.954	0.00120
	RLB$_o$	187s	0.997	0.00016	62s	0.994	0.00022
corel	ZNP	**95s**	0.997	0.00006	**38s**	0.973	0.00050
	ZNN	186s	0.993	0.00069	176s	0.996	0.00073
	NNDES	176s	**0.998**	0.00001	69s	0.977	0.00007
	RLB$_g$	-	-	-	-	-	-
	RLB$_o$	-	-	-	-	-	-

		$k = 5$			$k = 1$		
Dataset	Method	Time	recall	σ(recall)	Time	recall	σ(recall)
audio	ZNP	**6s**	0.788	0.00204	-	-	-
	ZNN	94s	**0.982**	0.00054	92s	**0.991**	0.00054
	NNDES	**6s**	0.651	0.00214	-	-	-
	RLB$_g$	16s	0.817	0.00143	**9s**	0.762	0.00247
	RLB$_o$	93s	0.969	0.00036	54s	0.967	0.00080
shape	ZNP	6s	0.896	0.00124	-	-	-
	ZNN	133s	**0.998**	0.00018	126s	**0.999**	0.00029
	NNDES	**6s**	0.833	0.00192	-	-	-
	RLB$_g$	10s	0.923	0.00086	**8.07s**	0.926	0.00105
	RLB$_o$	38s	0.992	0.00035	29s	0.994	0.00064
corel	ZNP	**19s**	0.816	0.00407	-	-	-
	ZNN	170s	**0.997**	0.00042	165s	**0.998**	0.00012
	NNDES	38s	0.743	0.00037	-	-	-
	RLB$_g$	287s	0.960	0.00017	139s	0.953	0.00020
	RLB$_o$	-	-	-	**34s**	0.662	0.00092

Table 5: Results for high accuracy setting.

In addition to the main data sets (shape, corel and audio) we did an experiment with a small data set called DIM064[5], which has 1024 points with dimensionality 64. With this set the running time of our algorithm was somewhat slower than that of the $O(N^2)$ brute force algorithm even for a modest recall rate of 0.8. We therefore recommend the method only for relatively large data sets consisting of at least 10,000 points. The smallest data set where we successfully tested our algorithm was the shape dataset with 28,775 points. In that case our ZNP algorithm gave 0.997 recall rate with only 11% of time in comparison to the brute force algorithm.

6.6 Effect of dimensionality reduction

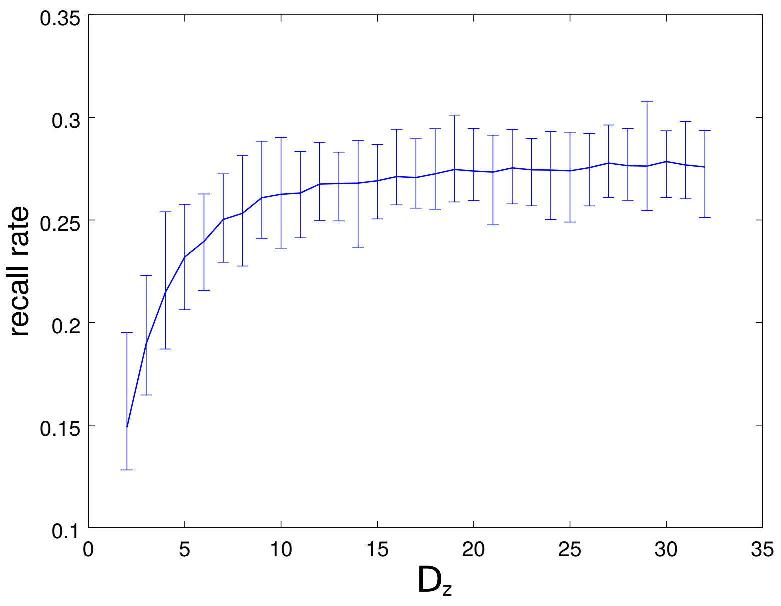


Figure 21: The effect of dimensionality reduction (D_z) on recall rate with 544 dimensional shape data set. Only one curve used ($N_c = 1$).

[5]http://cs.uef.fi/sipu/datasets/

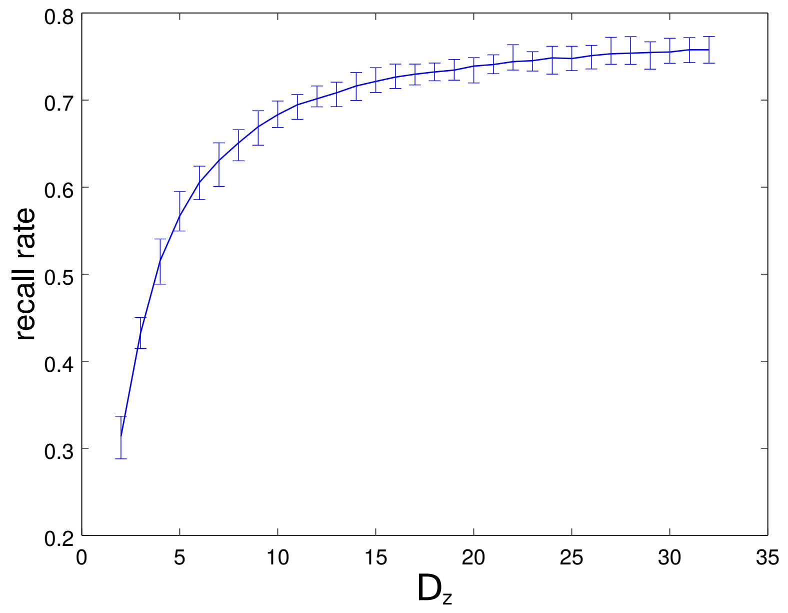


Figure 22: The effect of dimensionality reduction (D_z) on recall rate with 544 dimensional shape data set. Seven curves used ($N_c = 7$).

Figures 21, 22 shows the effect of dimensionality reduction on the quality in the ZNN-algorithm. D_z ranges from 2 to 32 because the z-order -curve is defined for a minimum of two dimensions and 32 was the maximum value allowed by our implementation. Window width of $W = 100$ was used in this experiment. In Figure 21, only one curve was used and 7 curves in Figure 22. The experiment was repeated 50 times for each D_z while other parameters remained constant.

It can be seen from the results that the recall rate decreases rapidly when $D_z < 10$, so sufficiently large D_z is needed to provide good results. However, when $D_z > 10$, there are only small differences in the quality of results. Additionally, when D_z grows, the program execution time also grows. We therefore suspect that there is an optimal value for D_z smaller than D, but more research is needed to determine this.

It can be also seen from Figures 21,22 that there is a significant amount of variation in the quality of results between program runs. The variations originates from three kinds of randomization in the algorithm: (1) random shifting of points, (2) random division to subvectors in dimensionality reduction and (3) random ordering of dimensions in bit interleaving.

7 Discussion

Constructing the kNN-graph has many important practical applications in a wide variety of fields. Many good solutions have been developed for this problem. Typical fast solutions like k-d -trees work well in the simple cases where the number of dimensions is relatively low ($D < 10$). For higher number of dimensions these methods begin to fail and provide no significant improvements over the trivial brute force method. Many kNN-graph construction methods have been developed specifically for higher number of dimensions, but they can still be slow or inaccurate, and their results vary unpredictably depending on the data set.

We introduced a new method called ZNP algorithm which uses space filling curves and neighborhood propagation. The performance of this algorithm was compared against two other leading fast kNN-graph construction methods using 3 different medium to high dimensional data sets. The results show that with $k \in \{10, 20\}$ our method outperforms existing methods with all of the tested data sets while providing moderate (27-46%) speed improvements for the same quality. With $k \in \{1, 5\}$ our method also performed well although no method outperformed the others.

Our method provides speed improvements in kNN graph construction for medium to high dimensional data sets but the speed still remains too slow for several practical applications especially for high dimensional ($D > 500$) data. Therefore, it remains an open question whether an efficient solution exists for data sets where the number of intrinsic dimensions is very high. We believe that the current method based on z-order curve is not yet up to its full potential and further research could provide significant further improvement.

Future research

Several kNN-graph construction methods use one dimensional proximity preserving projection. Our method and [7] do this using space filling curves. Zhang *& al.* [8] use vectors of locality sensitive hash values which were projected to random lines. In all these methods, one projection can preserve the proximity well for some points but not for others. Therefore, multiple different projections are often used. There are major differences between these algorithms, but they all face a common problem: how to generate a set of one dimensional projections that best preserve the proximity of points.

In all of the surveyed algorithms some kind of randomization is used to produce different proximity preserving projections. In our method, randomization was used because it is an easy way to generate the variation for the z-order curve. However, it is not an optimal way and some deterministic way of constructing the one dimensional mappings might be more efficient. Figure 21 shows that there is a large variance in the quality of results depending on which random projection is used. This suggests that it would be possible to improve the quality of the results if better projections could be chosen in a computationally efficient way.

We used a combination of the z-order curve and a simple dimensionality reduction technique to create a set of proximity preserving one dimensional mappings. The dimensionality reduction technique we used did not take into account properties of the data such as data distribution. It might be possible to improve our method by using a more advanced dimensionality reduction technique such as *principal component analysis* (PCA) that adapts to the internal structure of the data. The problem with PCA, however, is that it is computationally expensive. To the best of our knowledge the fastest exact method for PCA take $O(D^2h + D^2N)$ time where h is the number of target dimensions [57]. So doing a full principal component analysis would likely increase the computation time more than it would save. For our purposes, however, the accuracy of PCA is not very important. The axes chosen for projection need only to provide better proximity preserving qualities than our current method which projects sub-vectors to the diagonal axis. Therefore some approximation method might be a good solution. For example, a trivial sampling technique for the calculation of principal axes might work.

To increase the difference between multiple proximity preserving one dimensional projections, we used random shifting of points and random permutation of dimensions. Also rotations have been successfully used in a k-d tree based method [36] for a similar purpose. Rotation would be possible to use also with space filling curves, but it is unclear if it would bring enough benefits to justify the $O(D) \rightarrow O(D \cdot \log(D))$ per point increase in time complexity.

Our algorithm ordered the data points by calculating and storing z-values for each point and then sorting them according to the z-values. Another approach used in [7] is to calculate the z-values only when the points are compared by the sorting algorithm. This results in $O(N \cdot \log(N))$ z-value calculations instead of $O(N)$ in our method. The good point is that the memory requirements are reduced by $O(ND)$ bytes because the z-values are not stored. In addition, their method does not calculate the full z-values. Instead they iteratively calculate the z-values for compared points down from most significant bit towards least significant bit until a difference between the points is found. In their experiments, this method was faster than calculating the full z-values. Therefore it might be possible to use this method to provide a further speedup to our algorithm. However, their method was tested only with low dimensional ($D \leq 3$) data sets and it is not clear if it would give the same speed benefits for high dimensional data sets.

The recall rate we used to measure the quality of results has been used in many studies of the same subject [11, 12, 8]. However, there are other quality factors that could be considered. For example, for some applications it may be more important that the first nearest neighbor in the graph is correct than the kth nearest neighbor. The recall rate does not reflect this as it depends only on the total number of errors.

There is some evidence in our experiments that the errors in the results produced by our ZNN algorithm are more likely to be on small number of n for nth nearest neighbor ($n \in [1, k]$) than for the NNDES algorithm. In the NNDES algorithm, the errors were more likely to be for large n. However, our experiments on this were not quite extensive and more research should be done to find conclusive evidence.

8 Conclusion

In this thesis we have analyzed different methods for high dimensional kNN graph construction, and introduced a new method for solving this problem by using a combination of space filling curves and neighborhood propagation. We compared our method with two other methods called Nearest Neighbor Descent [12] and Recursive Lanczos Bisection [11] using 14 to 544 -dimensional data sets and Euclidean distance metric.

Experiments showed that our method is, in most cases, faster than the other tested methods. It performs well especially for large values of k and for the 14 dimensional corel data set consisting of 662,317 points. In this case, it is 46% faster than the 2nd best method. Our method also performs well even with higher dimensional data sets. With 544 dimensional shape data set it was 36% faster than the 2nd best method.

However, due to lack of having code available, we were unable to compare our method against two other methods [13, 8] which have reported similar performance increases in comparison to Nearest Neighbor Descent, which in our experiments was the second best method. One limitation of our method is that it has so far been confirmed to work only with the Euclidean distance metric while other methods work with larger variety of distance functions [12, 13, 8]. Therefore the performance of our method in comparison to other fast methods still remains somewhat unclear. We also believe that our current solution is not optimal and could be further improved by one of those ideas discussed in Section 7.

9 References

[1] X. Zhu, J. Lafferty, and R. Rosenfeld, *Semi-supervised learning with graphs*. PhD thesis, Carnegie Mellon University, Language Technologies Institute, School of Computer Science, 2005.

[2] P. Fränti, O. Virmajoki, and V. Hautämaki, “Fast agglomerative clustering using a k-nearest neighbor graph,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 11, pp. 1875–1881, 2006.

[3] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, “Fast approximate nearest-neighbor search with k-nearest neighbor graph,” in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, p. 1312, 2011.

[4] M. Belkin and P. Niyogi, “Laplacian eigenmaps for dimensionality reduction and data representation,” *Neural computation*, vol. 15, no. 6, pp. 1373–1396, 2003.

[5] J. Philbin, J. Sivic, and A. Zisserman, “Geometric latent dirichlet allocation on a matching graph for large-scale image datasets,” *International journal of computer vision*, vol. 95, no. 2, pp. 138–153, 2011.

[6] V. Hautamäki, I. Kärkkäinen, and P. Fränti, “Outlier Detection Using k-Nearest Neighbour Graph.,” in *ICPR (3)*, pp. 430–433, 2004.

[7] M. Connor and P. Kumar, “Fast construction of k-nearest neighbor graphs for point clouds,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 4, pp. 599–608, 2010.

[8] Y.-M. Zhang, K. Huang, G. Geng, and C.-L. Liu, “Fast kNN Graph Construction with Locality Sensitive Hashing,” in *Machine Learning and Knowledge Discovery in Databases*, pp. 660–674, Springer, 2013.

[9] V. Garcia, E. Debreuve, and M. Barlaud, “Fast k nearest neighbor search using GPU,” in *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1–6, IEEE, 2008.

[10] J. H. Friedman, J. L. Bentley, and R. A. Finkel, “An Algorithm for Finding Best Matches in Logarithmic Expected Time,” *ACM Trans. Math. Softw.*, vol. 3, pp. 209–226, Sept. 1977.

[11] J. Chen, H.-r. Fang, and Y. Saad, "Fast approximate k NN graph construction for high dimensional data via recursive Lanczos bisection," *The Journal of Machine Learning Research*, vol. 10, pp. 1989–2012, 2009.

[12] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th international conference on World wide web*, pp. 577–586, ACM, 2011.

[13] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li, "Scalable k-NN graph construction for visual descriptors," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pp. 1106–1113, IEEE, 2012.

[14] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[15] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When Is "Nearest Neighbor" Meaningful?," in *In Int. Conf. on Database Theory*, pp. 217–235, 1999.

[16] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the Surprising Behavior of Distance Metrics in High Dimensional Spaces," in *Proceedings of the 8th International Conference on Database Theory*, ICDT '01, (London, UK), pp. 422–434, Springer-Verlag, 2001.

[17] D. Dobkin and R. J. Lipton, "Multidimensional searching problems," *SIAM Journal on Computing*, vol. 5, no. 2, pp. 181–186, 1976.

[18] M. I. Shamos, *Computational geometry.* PhD thesis, Yale University, 1978.

[19] P. M. Vaidya, "An O(n log n) algorithm for the all-nearest-neighbors problem," *Discrete & Computational Geometry*, vol. 4, no. 1, pp. 101–115, 1989.

[20] E. Pielou, "A single mechanism to account for regular, random and aggregated populations," *The Journal of Ecology*, pp. 575–584, 1960.

[21] J. McNames, "A fast nearest-neighbor algorithm based on a principal axis search tree," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 9, pp. 964–976, 2001.

[22] T. Seidl and H.-P. Kriegel, "Optimal multi-step k-nearest neighbor search," in *ACM SIGMOD Record*, vol. 27, pp. 154–165, ACM, 1998.

[23] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, “iDistance: An Adaptive B+-tree Based Indexing Method for Nearest Neighbor Search,” *ACM Trans. Database Syst.*, vol. 30, pp. 364–397, June 2005.

[24] B. Yao, F. Li, and P. Kumar, “K nearest neighbor queries and knn-joins in large relational databases (almost) for free,” in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pp. 4–15, IEEE, 2010.

[25] E. Chavez, K. Figueroa, and G. Navarro, “A fast algorithm for the all k nearest neighbors problem in general metric spaces,” 1997.

[26] L. A. Piegl and W. Tiller, “Algorithm for finding all k nearest neighbors,” *Computer-Aided Design*, vol. 34, no. 2, pp. 167–172, 2002.

[27] G. Karypis, E.-H. Han, and V. Kumar, “Chameleon: Hierarchical clustering using dynamic modeling,” *Computer*, vol. 32, no. 8, pp. 68–75, 1999.

[28] O. Virmajoki and P. Fränti, “Divide-and-conquer algorithm for creating neighborhood graph for clustering,” in *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, vol. 1, pp. 264–267, IEEE, 2004.

[29] R. Paredes and E. Chávez, “Using the k-nearest neighbor graph for proximity searching in metric spaces,” in *String Processing and Information Retrieval*, pp. 127–138, Springer, 2005.

[30] A. Hinneburg, C. C. Aggarwal, and D. A. Keim, “What is the nearest neighbor in high dimensional spaces?,” in *Proc. of the 26th VLDB Conference, Cario, Egypt*, 2000.

[31] N. Kouiroukidis and G. Evangelidis, “The Effects of Dimensionality Curse in High Dimensional kNN Search,” in *Informatics (PCI), 2011 15th Panhellenic Conference on*, pp. 41–45, Sept 2011.

[32] P. Indyk and R. Motwani, “Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality,” in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC ’98, (New York, NY, USA), pp. 604–613, ACM, 1998.

[33] R. F. Sproull, “Refinements to nearest-neighbor searching in k-dimensional trees,” *Algorithmica*, vol. 6, no. 1-6, pp. 579–589, 1991.

[34] S. Arya, D. M. Mount, and O. Narayan, “Accounting for boundary effects in nearest-neighbor searching,” *Discrete & Computational Geometry*, vol. 16, no. 2, pp. 155–176, 1996.

[35] C. Silpa-Anan and R. Hartley, “Optimised KD-trees for fast image descriptor matching,” in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pp. 1–8, IEEE, 2008.

[36] P. W. Jones, A. Osipov, and V. Rokhlin, “Randomized approximate nearest neighbors algorithm,” *Proceedings of the National Academy of Sciences*, vol. 108, no. 38, pp. 15679–15686, 2011.

[37] M. F. Mokbel and W. G. Aref, “Irregularity in high-dimensional space-filling curves,” *Distributed and Parallel Databases*, vol. 29, no. 3, pp. 217–238, 2011.

[38] H. Tropf and H. Herzog, “Multidimensional Range Search in Dynamically Balanced Trees.,” *ANGEWANDTE INFO.*, no. 2, pp. 71–77, 1981.

[39] J. A. Orenstein and T. H. Merrett, “A class of data structures for associative searching,” in *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pp. 181–190, ACM, 1984.

[40] C. Faloutsos and S. Roseman, “Fractals for Secondary Key Retrieval,” in *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS ’89, (New York, NY, USA), pp. 247–252, ACM, 1989.

[41] N. Megiddo and U. Shaft, “Efficient nearest neighbor indexing based on a collection of space filling curves,” *IBM Almaden research center, San Jose, CA, Tech*, 1997.

[42] J. A. Shepherd, X. Zhu, and N. Megiddo, “Fast indexing method for multidimensional nearest-neighbor search,” in *Electronic Imaging’99*, pp. 350–355, International Society for Optics and Photonics, 1998.

[43] S. Liao, M. Lopez, and S. Leutenegger, “High dimensional similarity search with space filling curves,” in *Data Engineering, 2001. Proceedings. 17th International Conference on*, pp. 615–622, 2001.

[44] R. Dafner, D. Cohen-Or, and Y. Matias, “Context-based Space Filling Curves,” in *Computer Graphics Forum*, vol. 19, pp. 209–218, Wiley Online Library, 2000.

[45] T. Bially, “Space-filling curves: Their generation and their application to bandwidth reduction,” *IEEE Transactions on Information Theory*, vol. 15, no. 6, pp. 658–664, 1969.

[46] I. Gargantini, “An Effective Way to Represent Quadtrees,” *Commun. ACM*, vol. 25, pp. 905–910, Dec. 1982.

[47] C. Faloutsos and Y. Rong, “DOT: A Spatial Access Method Using Fractals,” in *Proceedings of the Seventh International Conference on Data Engineering*, (Washington, DC, USA), pp. 152–159, IEEE Computer Society, 1991.

[48] H.-L. Chen and Chang, “All-nearest-neighbors finding based on the Hilbert curve,” *Expert Systems with Applications*, vol. 38, no. 6, 2011.

[49] M. Mokbel, W. Aref, and I. Kamel, “Analysis of Multi-Dimensional Space-Filling Curves,” *GeoInformatica*, vol. 7, no. 3, pp. 179–209, 2003.

[50] G. Peano, “Sur une courbe, qui remplit toute une aire plane,” *Mathematische Annalen*, vol. 36, no. 1, pp. 157–160, 1890.

[51] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[52] S.-X. Li and M. H. Loew, “The quadcode and its arithmetic,” *Communications of the ACM*, vol. 30, no. 7, pp. 621–626, 1987.

[53] H.-L. Chen and Y.-I. Chang, “Neighbor-finding Based on Space-filling Curves,” *Information Systems*, vol. 30, pp. 205–226, May 2005.

[54] J. Lawder, *The application of space-filling curves to the storage and retrieval of multi-dimensional data*. PhD thesis, University of London, UK, 2000.

[55] R. Raman and D. S. Wise, “Converting to and from dilated integers,” *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 567–573, 2008.

[56] S.-W. Ra and J. Kim, “A fast mean-distance-ordered partial codebook search algorithm for image vector quantization,” *IEEE transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 40, no. 9, pp. 576–579, 1993.

[57] A. Sharma and K. K. Paliwal, “Fast principal component analysis using fixed-point algorithm,” *Pattern Recognition Letters*, vol. 28, no. 10, pp. 1151–1155, 2007.

Appendix 1: Source code for z-value generation

```
// zvalue.cpp
#include <stdio.h>
#include <stdlib.h>
#include <boost/multiprecision/cpp_int.hpp>


#define LONG uint1024_t
#define SIZE_LONG 1024
using namespace boost::multiprecision;
#include "zvalue.h"

// To compile:
// g++ -O3 -Wall -static -I. -o zvalue zvalue.cpp

// Usage: ./zvalue 3 5
// Output: 39

int main (int argc, char *argv[]) {

    unsigned int z_dimensions = unsigned(argc - 1);
    unsigned int P[z_dimensions];

    LONG ** lookup_table = generate_lookup_table(z_dimensions);
    for(unsigned int i=0; i < z_dimensions; i++) {
        P[i] = unsigned(atoi(argv[i+1]));
    }
    LONG z = z_value(lookup_table,P,z_dimensions);

    std::cout << "" << z << std::endl;
}
```

```
// zvalue.h
#ifndef ZVALUE_H_
#define ZVALUE_H_

#include <stdint.h>
#include <limits.h>

#define LOOKUP_SIZE 256
#define LOOKUP_UNIT unsigned int

LONG ** generate_lookup_table(unsigned int dimensions) {

    LONG **lookup_table;

    LONG answer = 0;
    LONG one = 1;

    unsigned int limit = SIZE_LONG/dimensions;
    if(limit > 32) { limit = 32;}

    // Allocate memory
    lookup_table = new LONG*[dimensions];
    for (unsigned int i = 0; i < dimensions; ++i)
        lookup_table[i] = new LONG[LOOKUP_SIZE];

    // Generate lookup table
    for (unsigned int i_dim = 0; i_dim < dimensions; ++i_dim) {
        for (unsigned int j = 0; j < LOOKUP_SIZE; ++j) {
            answer = 0;
            for (unsigned int k = 0; k < limit; ++k) {
                answer |= ((j & (one << k)) << (k*(dimensions-1) +
                   i_dim) );
            }
            lookup_table[i_dim][j] = answer;
        }
    }

    return lookup_table;
}

inline LONG z_value(
        LONG ** lookup_table, unsigned int * point, unsigned int
            dimensions){

    LONG answer = 0;
    unsigned int rightshift = 32; // == sizeof(unsigned int)*8

    // Once for each 8 bytes in unsigned int
    for(unsigned int i_depth = 1; i_depth <= 4; i_depth++) {
        answer = answer << dimensions*8;
            for(unsigned int i_dim = 0; i_dim < dimensions; i_dim++)
                {
                answer |= lookup_table[i_dim][
                    (point[i_dim] >> (rightshift -8*i_depth)) & 0xFF
                        ] ;
            }
    }

    return answer;
}

#endif // ZVALUE_H_
```