Constructing a High-Dimensional *k*NN-Graph Using a Z-Order Curve

SAMI SIERANOJA and PASI FRÄNTI, University of Eastern Finland

Although many fast methods exist for constructing a *k*NN-graph for low-dimensional data, it is still an open question how to do it efficiently for high-dimensional data. We present a new method to construct an approximate *k*NN-graph for medium- to high-dimensional data. Our method uses one-dimensional mapping with a Z-order curve to construct an initial graph and then continues to improve this using neighborhood propagation. Experiments show that the method is faster than the compared methods with five different benchmark datasets, the dimensionality of which ranges from 14 to 784. Compared to a brute-force approach, the method provides a speedup between 12.7:1 and 414.2:1 depending on the dataset. We also show that errors in the approximate *k*NN-graph originate more likely from outlier points; and, it can be detected during runtime, which points are likely to have errors in their neighbors.

CCS Concepts: • **Theory of computation** → **Nearest neighbor algorithms**; • **Mathematics of computing** → *Graph algorithms*; • **Information systems** → *Nearest-neighbor search*;

Additional Key Words and Phrases: kNN-graph, graph construction, nearest neighbor, Z-order curve, space-filling curves, neighborhood propagation

ACM Reference format:
Sami Sieranoja and Pasi Fränti. 2018. Constructing a High-Dimensional *k*NN-Graph Using a Z-Order Curve. *J. Exp. Algorithmics* 23, 1, Article 1.9 (October 2018), 21 pages.
https://doi.org/10.1145/3274656

1 INTRODUCTION

Given a set of N points $X = \{x_1, x_2 \ldots, x_N\}$ in some D-dimensional space S, the *k-nearest neighbor* problem (*k*NN) is to find the k points in X that are closest to a given query point $q \in S$ according to some distance metric d. A search for the *k*NN for all points in X yields a directed graph called *kNN-graph* where the vertices correspond to points in the dataset and edges connect each point to its k-nearest points in the dataset.

Constructing a *k*NN-graph is important for a wide range of applications including classification [39], agglomerative clustering [16], k-nearest neighbor search [21], dimensionality reduction [3], outlier detection [22], and computer graphics [10]. For many of these applications, constructing the graph is a major bottleneck [16, 38].

The trivial brute-force algorithm constructs a *k*NN-graph in $O(N^2)$ time by calculating distances between all pairs of points and selecting the k-smallest distances for every point. This can be practical for small datasets consisting of up to tens of thousands of points, especially when utilizing

Authors' addresses: S. Sieranoja and P. Franti; emails: {samisi, franti}@cs.uef.fi.

1084-6654/2018/10-ART1.9 $15.00
https://doi.org/10.1145/3274656

parallel computing capabilities of modern GPUs [19]. However, for larger datasets, consisting of millions of points, the brute-force algorithm becomes too slow.

Fast methods such as k-d trees [17] and space-filling curves [10] can construct an exact kNN-graph for low-dimensional datasets in $O(N \cdot \log(N))$ time. However, with high-dimensional data, these methods fail to provide speed-up over the brute-force method. Recent research has, therefore, focused on constructing approximate kNN-graphs [9, 12, 16, 18, 35, 38].

Most approximate methods [9, 35, 38] work by dividing the data into smaller groups and then using a brute-force method for each group. Chen [9] and Wang [35] do this by projecting to principal axis and hierarchically splitting by the projection. Zhang uses locality sensitive hashing with random linear projection [38]. A complementary approach called neighborhood propagation is used in all recent methods [9, 35, 38] but with different variations. The key idea is to consider the neighbors of neighbors, and update the current kNN list whenever a closer neighbor candidate is found. The method by Dong [12] starts from a random graph and improves it iteratively by neighborhood propagation as long as a better graph is found.

In this article, we propose a novel two-stage method to construct an approximate kNN-graph. The first step uses a Z-order curve to construct an initial, roughly 20–50% accurate graph. This step is similar to the Z-order sliding window method used by Connor et al. [10] for a low-dimensional exact kNN-graph, but we adapt it to high-dimensional data by using a novel fast neighborhood preserving dimensionality reduction method. The second step improves this approximate graph by using neighborhood propagation [12] jointly with the Z-order curve. The method is targeted especially for high-dimensional data.

We compare the method against NN-Descent [12], KGRAPH,[1] Recursive Lanczos Bisection [9], LSH [38], and Multiple PCA divide-and-conquer [35]. Experimental results with 14 to 784-dimensional data show that our method provides better speed/quality ratio than the compared methods when using Euclidean distance metric and other Minkowski distances between $p = 0.1$ to 10.

The new contributions can be summarized as:

- First time to apply space-filling curves to generate approximate kNN-graph.
- Introduce a new variant of z-curve, which allows us to use z-values also in high-dimensional spaces (Algorithm 3).
- We propose a novel interleaving of the Z-order search and neighborhood propagation that is less sensitive to the choice of parameters than the existing methods (Algorithm 2).
- We demonstrate that the errors in approximate kNN-graphs originate more often from outliers. This observation has potential to reduce the errors of any approximate kNN-graph -method by focusing more extensive search on outlier points (Section 4.6).
- We introduce a new metric to calculate the proximity preserving quality Q of one-dimensional mappings (Section 2.3). This Q-value correlates with the accuracy of the resulting kNN graph. Optimizing the Q-value of the projection could potentially improve any methods that use space-filling curves.

The rest of the article is organized as follows. In Section 2, we briefly discuss the effect of dimensionality and the choice of distance measure. We then present space-filling curves and neighborhood propagation, which are the basic building blocks of our method. The new method and its design choices are then introduced in Section 3. Experimental comparisons are presented in Section 4, and finally, conclusions are drawn in Section 5.

[1]http://kgraph.org/.

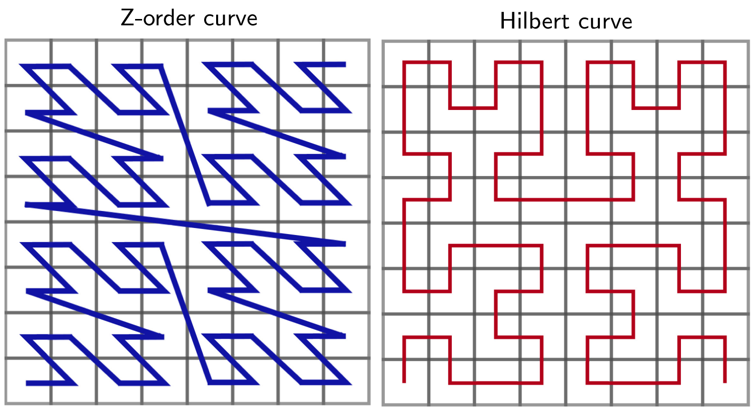

Fig. 1. Hilbert curve and the Z-order curve are the two most commonly used space-filling curves in computer science.

2 BACKGROUND AND RELATED WORK

2.1 Effect of Dimensionality

There have been a great deal of theoretical studies analyzing the effect of dimensionality for the nearest neighbor search [1, 4, 24], which is a subproblem of *k*NN-graph construction and is generally thought to share the same issues regarding dimensionality. These studies have noted especially two major issues regarding the rise of dimensionality: (1) the *quality* issue and (2) the *performance* issue.

In the *quality* issue, an 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 [4] or if other non-conventional distance functions would be more useful [1].

In the *performance* issue, computational requirements of known exact nearest neighbor search methods increase exponentially in D [25], and, therefore, are not practical for most high-dimensional real life datasets.

In this work, we focus on providing solutions for the performance issue.

2.2 Measuring Distance

We use Euclidean distance as primary distance measure but consider also other types of Minkowski distance, which is defined as follows:

$$d(x, y) = \left(\sum_{i=1}^{D} |x_i - y_i|^p \right)^{1/p} \tag{1}$$

Value $p = 2$ corresponds to Euclidean distance (L_2) and value $p = 1$ to Manhattan distance(L_1). According to Aggarwal [1], fractional values ($0 < p < 1$) are better suited for high-dimensional data. Still, the Euclidean distance is more widely used, and, for example, the SIFT features [28] are specifically developed for it.

2.3 Space-Filling Curves

A space-filling curve is a way of mapping a discrete multidimensional space into one-dimensional space [29] (see Figure 1). It imposes a linear order for points in multidimensional space. This order

$$\begin{array}{c} \text{interleaved bits} \downarrow \\ (3,5) = (\mathbf{011}_2, 101_2) \to \mathbf{0}1\ \mathbf{1}0\ \mathbf{1}1_2 = 27 \\ \uparrow \text{2D vector} \qquad\qquad \text{Z-value} \uparrow \end{array}$$

Fig. 2. Z-value calculation for a two-dimensional vector.

is used to preserve the proximity of points so that points that are near 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 include range search [31, 34], searching for nearest neighbor [14], *k*NN [37], and constructing an exact *k*NN-graph [10]. Space-filling curves have also been used in image compression [11], bandwidth reduction [5], representation of quadtrees [20], and as an indexing method for faster disk reads [13].

Proximity Preserving Qualities. The popularity of space-filling curves in computer science is primarily due to their *proximity preserving qualities*. 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* [7] and *distance preserving quality* [14]. These terms are usually used without formal definition, although some definitions have also been given in the context of search and sorting [23, 36].

We give the following definition for the proximity preserving quality, which is inspired by the definition given for locality sensitive hash functions in Ref. [25]. The definition is not limited to space-filling curves and can also be applied to other one-dimensional mappings. Let P denote probability, d distance function in the multidimensional space, and s a mapping of point x to the curve. Proximity preserving quality is then defined as:

$$Q = P(d(q,x) < d(q,y))\ \forall_{q,x,y \in X}\ |s(q) - s(x)| < |s(q) - s(y)| \tag{2}$$

In other words, for $Q > 0.5$, if q is closer to x than y on the curve s, q is also more likely to be closer to x in the multidimensional space. The mapping s is considered more proximity preserving the closer the value Q is to 1.

To calculate this value, in practice, one would need to loop over all $O(N^3)$ possible 3-tuples $(q,x,y) \in X^3$ (or use random sampling), and count the cases where the point x or y that is farther from q in absolute distance is also farther along the curve:

$$Q = \frac{1}{N^3} \sum_{(q,x,y) \in X^3} \begin{cases} 1 & (d(q,x) - d(q,y))(|s(q) - s(x)| - |s(q) - s(y)|) > 0 \\ 0 & \text{else.} \end{cases} \tag{3}$$

Z-order Curve. The *Z-order curve* (Figure 1) is a type of space-filling curve, a function which maps multidimensional points to one dimension by interlacing the bits of the binary representation of the vector components (see Figure 2). 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*. The Z-order has been discovered independently by several authors, including Morton [30], Tropf and Herzog [34], and Orenstein [31].

The calculation of a Z-value is shown in Figure 2. The vector components are first converted to binary representation. The bits of the binary representation are then 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 order of dimensions in the bit interleaving.

2.4 Neighborhood Propagation

Neighborhood propagation is a method to construct (by starting from a 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 a 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 [9, 12, 35, 38]. 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 the main algorithm. Chen et al. [9] uses neighborhood propagation to refine the graph after their divide-and-conquer method. Wang et al. [35] uses multiple random divide-and-conquer and Zhang et al. [38] uses locality sensitive hashing (LSH) before neighborhood propagation.

The basic principle is shown in Algorithm 1. The algorithm takes as parameter a set of points X and an initial approximation for the kNN of each point. The initial approximation can consist of just random points chosen from X or it can be output from another non-exact kNN-graph algorithm.

The algorithm iteratively improves the quality of the graph. In each iteration, the neighbors of neighbors are tested for each point $x \in X$. If any of them are closer than the furthest of the current neighbors, the neighbor list is updated accordingly. The algorithm is iterated until a specified stop condition is met. For example, it can be run just for a fixed number of iterations [38] or as long as the method is able to improve the graph [12].

ALGORITHM 1: PropagateNeighborhood

```
procedure PropagateNeighborhood($X$, $kNN$) → $kNN$
    repeat
        for all $x \in X$ do
            for all $Neighbor \in kNN(x)$ do
                for all $y \in kNN(Neighbor)$ do
                    UpdateNeighborList($x, y, kNN$)
                    UpdateNeighborList($y, x, kNN$)
                end for
            end for
        end for
    until stop condition met
    return $kNN$
end procedure
```

Since each point has k^2 neighbors of neighbors, 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.

3 Z-ORDER NEIGHBORHOOD PROPAGATION

In this chapter, we present our method called *Z-order neighborhood propagation (ZNP)* for constructing the k-NN graph (see Algorithm 2). It has two parts: First, an initial rough approximation is constructed by repeating a sliding window search for multiple Z-order projections (see Figure 3). Second, the graph is improved further by alternately running neighborhood propagation (NN-Descent algorithm) and the first method of Z-order sliding window search. Our method is targeted for high-dimensional datasets and the L_2 distance metric.

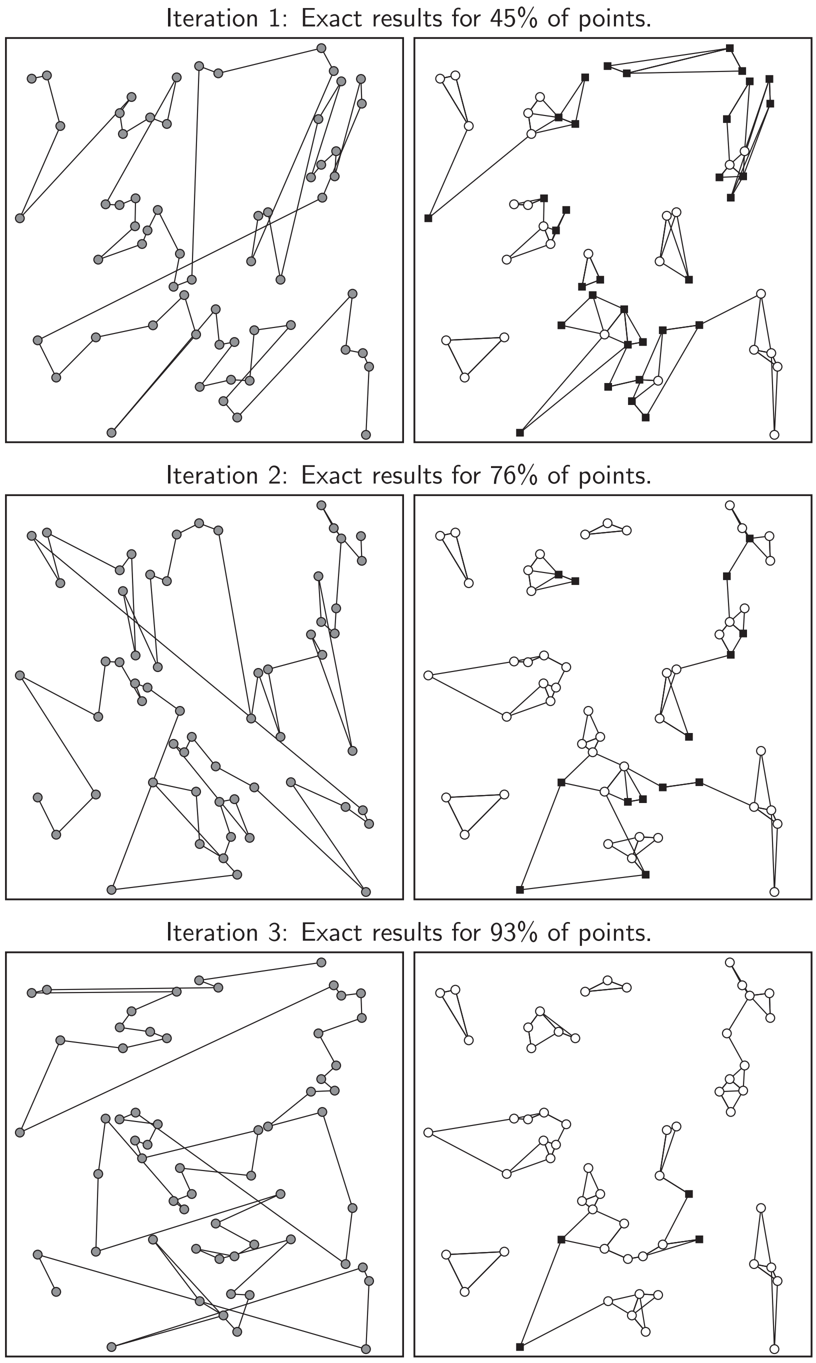
Iteration 1: Exact results for 45% of points.
Iteration 2: Exact results for 76% of points.
Iteration 3: Exact results for 93% of points.

Fig. 3. 2NN-graph construction using three iterations of ZNP algorithm with parameter $W = 2$ and without NN-Descent. Three Z-curves (left) and the resulting, gradually improved 2NN graph (right) are shown. White circles represent points with only correct neighbors and black rectangles represent points with some incorrect neighbors.

The motivation for this approach is that, by using only the Z-order search, a rough approximation (around 10–50% recall rate) of the graph can be achieved very fast. However, when continuing the search further, the quality improves slowly. Conversely, the NN-Descent algorithm gives usually very bad results on the first 1–3 iterations, mainly because it starts from a random *k*NN-graph. 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. However, the NN-Descent algorithm will at some point converge to a state where no new neighbors are found. For this reason, we run the Z-order search and NN-Descent alternately so that new neighbors found by the Z-order search can be further propagated to neighboring points using the NN-Descent algorithm.

The Z-order curve was chosen as basis of our method over other alternatives such as Hilbert curve. Most space-filling curve-based methods use either the Z-order or Hilbert curves because they have the best proximity preserving qualities [27, 37].

Some sources claim that the Hilbert curve has better proximity preserving quality than the Z-order curve [8, 14], but it is unclear whether these results generalize to higher dimensions. We chose the the Z-order curve mainly because its simplicity, which allows fast implementation also in high dimensions (with little effort).

3.1 Calculation of Z-value

Different methods for Z-value generation have been compared in Ref. [33] and by J. Baert.[2] We use the lookup table-method, which we implemented for an arbitrary number of dimensions and varying bit-lengths.

Other methods using space-filling curves for *k*NN-graph construction are limited to a low number of dimensions [10]. 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 dataset with dimensionality $D = 1,000$, bit-length $b = 32$ bits per dimension, and size of $N =$ 1,000,000 points, the Z-values would need to be represented by $D \cdot b = 32,000$ bit integers. Calculating, storing, and comparing such large integers would become a bottleneck in the algorithm.

For example, 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.

3.2 Dimensionality Reduction

We introduce, next, a simple but effective linear dimensionality reduction technique to avoid the aforementioned problems with high dimensionality (see Algorithm 3). It is inspired by the mean-distance ordered partial search (MPS) method introduced in Ref. [32], which was used to construct the *k*NN-graph in Ref. [16]. It is also related to Johnson-Lindenstraus transform (JLT) [2] with the main difference that JLT aims to preserve the distance between points in the projected space, whereas we are concerned only on preserving the neighbor connections.

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

We reduce dimensionality of data points from D to D_z before projecting to Z-order curve. We divide the dimensions into D_z random subsets with roughly equal sizes and then construct new subvectors corresponding to the subsets of the dimensions. Each subvector is mapped to one dimension by projecting them to the diagonal axis. The one-dimensional mappings of the subvectors are 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 $x = (5, 4, 7, 0, 3, 2)$, $D_z = 3$ and permutation of dimensions $DP = (4, 5, 6, 1, 2, 3)$ coordinates of point x would be first reordered to $x' = (0, 3, 2, 5, 4, 7)$ 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 $001010111111_2 = 703$.

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 us to use 32-bit integers

ALGORITHM 2: ZNP

Inputs:
Dataset of points X
Number of nearest neighbors k
Minimum graph change $\delta \in [0, 1]$
NN-Descent start parameter (graph maturity) $\gamma \in [0, 1]$
Dimensionality of the Z-order curve D_z
Width of sliding window W.

Output: kNN graph.

```
1:  procedure ZNP( $X, k, \delta, \gamma, D_z, W$)
2:      repeat
3:          $c \leftarrow 0$                                   ▷ Number of successful updates to kNN
4:          $S \leftarrow$ ProjectTo1D($X, D_z$);
5:          $X \leftarrow$ Sort so that $\forall x_j \in X, \quad S[j] \leq S[j+1]$     ▷ Sort based on Z-values
6:          for all $x_j \in X$ do                          ▷ Scan points using a sliding window
7:              for all $y \in \{x_{j+1}, \ldots, x_{j+W}\}$ do
8:                  $d \leftarrow$distance($x_j, y$)
9:                  $c \leftarrow c+$UpdateNeighborList($x_j, y, d$, kNN)
10:                 $c \leftarrow c+$UpdateNeighborList($y, x_j, d$, kNN)
11:             end for
12:         end for
13:         if $c/Nk < \gamma$ then
14:             $c \leftarrow c$ + NN-Descent(kNN)            ▷ One iteration of NN-Descent
15:         end if
16:     until $c/Nk < \delta$
17: end procedure
18:
19: procedure UpdateNeighborList($x, y, d$, kNN)
20:     update $\leftarrow 0$
21:     if $d < max\{d_y | (y, d_y) \in$ kNN$(x)\}$ or size(kNN($x$)) $< k$ then
22:         Insert $(y, d)$ into kNN($x$)
23:         Remove item with largest distance from kNN($x$) if size(kNN($x$)) $> k$
24:         update $\leftarrow 1$
25:     end if
26: return update
27: end procedure
```

to represent vector components and to calculate Z-values for up to 32 dimensional points without using dimensionality reduction ($32 \times 32 = 1{,}024$).

ALGORITHM 3: ProjectTo1D

```
Inputs:
    Dataset of points $X$
    Dimensionality of the Z-order curve $D_z$
Output:
    One-dimensional projection $S$
1: procedure ProjectTo1D($X, D_z$)
2:     $h \leftarrow$ Random vector of size $D$
3:     $X \leftarrow$ Scale $X$ to positive integer values
4:     $DP \leftarrow (0, \ldots, D-1)$                    ▷ Create random permutation of dimensions
5:     $DP \leftarrow$ Shuffle($DP$)
6:     parfor $i \leftarrow 0, N-1$ do
7:         $x \leftarrow$ Zero vector of size $D_z$
8:         for $j \leftarrow 0, D-1$ do                   ▷ Reduce dimensionality from $D$ to $D_z$
9:             $x[j \bmod D_z] \mathrel{+}= X[i][DP[j]] + h[j]$
10:        end for
11:        $S[i] \leftarrow$ z_value($x$)
12:    end parfor
13: return $S$
14: end procedure
```

3.3 Neighborhood Propagation

In our method, we use the original NN-Descent algorithm [12] for neighborhood propagation, with one small difference. We do not use the actual value k of the graph in the operation, because NN-Descent requires $O(k^2N)$ distance calculations per iteration, and this quadratic time complexity makes the algorithm impractical for higher values of k. Instead, we choose another value $\mathrm{k_{nndes}}$ using the rule $\mathrm{k_{nndes}} = \sqrt{jk}$, where j is a small number. So, for fixed $j = 10$, for example, $k = 10 \Rightarrow \mathrm{k_{nndes}} = 10$, and $k = 100 \Rightarrow \mathrm{k_{nndes}} = 32$.

To clarify, we do not use all neighbors, but only the nearest k_{nndes} neighbors. In this way, the time complexity per iteration can be kept linear for k, at $O(\mathrm{K_{nndes}}^2N) = O(kjN)$. This way of limiting k is also apparently somewhat related to the operation of KGRAPH,[1] a newer version of NN-Descent, although this feature is not clearly documented.

Several previous algorithms [9, 12, 35, 38] use some variant of neighborhood propagation. In all of them, one algorithm is first used to initialize the graph and neighborhood propagation is then used to refine it. However, without increasing the value k, neighborhood propagation will always converge at some point to a situation where no improvements on the graph can be made. For this reason, we apply neighborhood propagation (NN-Descent algorithm) alternately with the Z-order search. In this way, new neighbors found by Z-order search can be propagated to neighboring points using the NN-Descent search, and the search can be continued beyond the convergence point of the NN-Descent algorithm.

3.4 Parallelization

The Z-order search has three bottlenecks in the order of time consumption:

- The sliding window
- Z-value calculation
- Sorting of Z-values

All of these steps can be easily parallelized. The main bottleneck, the sliding window (line 6 in Algorithm 2) can be parallelized by dividing the (sorted) dataset into equal parts for multiple non-overlapping sliding windows, one for each thread. This avoids the problem of concurrent memory write access to the graph.

The z-value calculation (line 6 in Algorithm 3) can be trivially parallelized by using a parfor statement. Sorting the z-values (line 5 in Algorithm 2) can be done in parallel using standard libraries.

For the number of threads n, parallel efficiency is commonly defined as the time taken by single thread T_1 divided by n times the time taken by parallel threads T_n:

$$\text{Parallel efficiency} = \frac{T_1}{nT_n} \tag{4}$$

According to our experiments with the Sift1M dataset and $n = 4$ threads, parallelizing these three steps in our implementation using OpenMP leads to parallel efficiency of 83%. In other words, if single thread took 100 seconds, the parallel version would require 30 seconds.

We expect the method to be efficient also for GPUs. The main bottleneck of the sliding window is only slightly more complex than the brute-force method, and it can be divided into sub-problems similarly as brute force, which is easy to parallelize [19].

4 EXPERIMENTS

This section presents the methodology and results for our experiments. We compare the performance of our ZNP method to seven existing algorithms on Euclidean distance metric and five data sets, the dimensionality of which ranges from 14 to 784. We use *recall rate* to measure the quality of the kNN-graph and program execution time to measure speed. Additionally, in Section 4.6, we analyze which kind of points cause inaccuracies in the graphs. In Section 4.7, we study the effect of our dimensionality reduction method on the quality of results.

4.1 Evaluated Algorithms

We compare our ZNP algorithm against seven different algorithms, shown in the following table.

ZNP	Z-order neighborhood propagation, without NN-Descent (proposed)
ZNP$^+$	Z-order neighborhood propagation, including NN-Descent (proposed)
NNDES	Nearest Neighbor Descent (NN-Descent) [12]
KGRAPH	Newer version of NN-Descent with additional optimizations
RLB	Recursive Lanczos Bisection, variant *glue* [9]
MPS-limited	A limited version [15] of mean-distance-ordered partial codebook search (MPS) [32]
PCADIV	Multiple random divide-and-conquer using PCA [35]
LSH	Locality Sensitive Hashing [38]
Bruteforce	A naive algorithm that calculates all $N(N-1)/2$ possible distances

For our ZNP algorithm, we used the configuration parameters $D_z = 32$, $W = 2k$, and $\delta = 0.0001$. The algorithm was run once with NN-Descent disabled ($\gamma = 0$, denoted as ZNP) and another time with NN-Descent enabled ($\gamma = 0.3$), denoted as ZNP$^+$). Time and recall values are reported for each iteration. We have made our implementation also available for others.[3]

[3]http://www.uef.fi/web/machine-learning/software.

NNDES [12] 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 changes on the graph. Increasing δ increases the number of iterations. The NNDES algorithm was run for $\lambda = 1$ and $\delta = 0.00001$. Performance is reported for each iteration, starting from the second. For NN-Descent benchmarks, we used the implementation made available by the authors.[4]

KGRAPH[1] is a newer version of the NN-Descent algorithm with some additional features and optimizations. Namely, according to the documentation,[5] the actual neighborhood size M is not fixed to k but varies depending on data point while staying below an upper limit L. To set suitable parameter L, we used the recommended $L = k + 50$.

The RLB algorithm [9] has two variants: *glue* for higher speed but lower quality and *overlap* for higher quality and lower speed. We use only the variant glue. We use the implementation previously made available by the authors,[6] although no longer online. The RLB algorithm has quality control parameter *overlap size* for which we used the following values $(0.05, 0.10, 1.5, 0.20, 0.25, 0.30)$. Increasing the overlap size increases both recall rate and program execution time.

MPS-limited is an approximate version of mean-distance-ordered partial codebook search (MPS) [32], first used for kNN-graph construction in Ref. [15]. It has parameter W to limit the maximum number of distance calculations for each point, and, consequently, time complexity $O(NW)$. We run the algorithm for varying values of $W \in \{100, 500, 1000, 5000, 10000, 20000\}$.

Brute-force algorithm single core running times and quality were extrapolated based on a sample of 100,000,000 distance calculations. An actual brute-force algorithm for ground truth generation was run on parallel threads.

Locality Sensitive Hashing (LSH) [38] calculates hash-values for each point and uses a random linear projection to map the hash bucket values to one dimension. Based on this value, the points are divided into equal size blocks and a brute-force algorithm is applied for each block. The process is repeated for m different projections and then single step neighborhood propagation is used to improve the graph. The authors recommend a block size 100, but we used a value of 200 as this provided better results, especially in the case of $k = 100$. We vary the number of projections $m \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25\}$. For this method, we used our own implementation as we found no sources available. We use FALCONN[7] implementation for the hash function.

Multiple PCA random divide-and-conquer (PCADIV) [35] hierarchically divides the dataset into smaller subsets until the size is smaller than 500, and then runs brute-force algorithm for each part. This process is repeated m times until effective rate $r_m < 0.05$ is reached. The approximate graph is then improved using a variant of neighborhood propagation that expands the search to the nearest unvisited point. The propagation is iterated by the maximum number of steps $T \in \{0, 50, 100, 500, 1000, 2000, 4000\}$. For this method, we used our own implementation as we found no sources available.

4.2 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-graph G' in relation to accurate graph G by taking the number of common edges (neighbors) relative to the number of all

[4]https://code.google.com/p/nndes/.
[5]https://github.com/aaalgo/kgraph/blob/master/doc/params.md.
[6]http://www.mcs.anl.gov/~jiechen/software/knn.tar.gz.
[7]https://falconn-lib.org/.

Table 1. Summary of Datasets and Time Taken by Brute Force ($k = 10$)

Dataset	N	D	time
MNIST	10,000	784	337s
Shape	28,775	544	295s
Audio	54,387	192	401s
Corel	662,317	14	9,018s
Sift1M	1,000,000	128	100,846s

Brute force times for Euclidean distance, except (slower) Minkowski for MNIST.

edges [9, 12]:

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

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 *error rate* = 1 − *recall*.

Program execution time is measured as single thread execution time. The time needed to load the dataset 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.

4.3 Datasets

We use five different datasets: Shape, Audio, Corel, Sift1M, and MNIST. These first three datasets were used in Ref. [12] and can be found on the web.[8] Sift1M was first used in Ref. [6] and can be found on the web.[9] MNIST was first used in Ref. [26] and is available online.[10]

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

The Audio dataset 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.

Sift1M dataset contains one million Scale-invariant feature transform (SIFT)[28] image feature vectors of dimensionality 128.

MNIST data contains the raw image files of 10,000 handwritten digits in 28x28 resolution. The grayscale value of each pixel is interpreted as a dimension, which makes the data 784-dimensional.

4.4 Empirical Process

We run the algorithms for two different values of $k \in 10, 100$. We expect this range to be representative of most practical applications. For comparison, $k = 50$ was the highest value used in Ref. [35] and $k = 100$ in Ref. [39]. For the MNIST dataset, we also vary the parameter p for Minkowski distance.

[8]https://code.google.com/p/nndes/downloads/list.
[9]Downloaded from http://corpus-texmex.irisa.fr/.
[10]http://yann.lecun.com/exdb/mnist.

Table 2. Time Taken to Reach 80% Accuracy Level

		$p = 0.1$		$p = 1$		$p = 2$		$p = 10.0$	
Dataset	Method	Time	Accuracy	Time	Accuracy	Time	Accuracy	Time	Accuracy
MNIST	ZNP$^+$	**107s**	85.4%	**21s**	91.7%	**20s**	92.1%	**86s**	92.3%
	ZNP	446s	80.1%	37s	80.0%	36s	80.4%	171s	80.4%
	NNDES	122s	83.4%	47s	**92.8%**	49s	**93.8%**	165s	**93.8%**
	PCADIV	278s	**91.4%**	37s	83.3%	36s	85.6%	160s	85.0%
	KGRAPH	147s	84.1%	34s	85.6%	36s	85.9%	118s	86.2%
	bruteforce	1361s	80.0%	271s	80.0%	270s	80.0%	1358s	80.0%
ZNP$^+$ speedup from bruteforce		12.7:1		12.9:1		13.5:1		15.8:1	

Varying parameter p for Minkowski distance ($L_{0.1}$, L_1, L_2, L_{10} metrics).

All experiments were run on a computer equipped with 8-Core 4.0GHz AMD FX-8350 CPU with 8MB of L2 cache and 8GB of RAM. The computer was running 64-bit Ubuntu Linux 16.04. Code compilation for all tested algorithms was done using gcc version 5.4.1 with -O3 optimization.

All algorithms were run on a single thread, with parallelization disabled. SSE2 vectorized distance functions were disabled and simpler implementations used instead.

The Recursive Lanczos Bisection-algorithm was run only for datasets Shape and Audio. For other datasets, the program didn't finish because of high memory consumption (over 6GB of RAM).

4.5 Time Versus Quality Tradeoff

The results are presented in graphical plots in Figures 4 and 5 and Table 2. The results for RLB are missing in cases where the program did not give results with any overlap values. The Minkowski experiments were run only for the best algorithms selected based on the results in Figure 4.

From Figure 4, it can be seen that the ZNP algorithm provides best results with $k = 10$. With the Corel dataset, ZNP reaches a 90% accuracy level taking only 73% of the time of the second best method (PCADIV).

For $k = 100$, the PCADIV is somewhat faster with all datasets, although ZNP is still the second or third best method. One explanation is that PCADIV uses a different type of neighborhood propagation method, which expands to the nearest unvisited point. We suspect that it requires higher connectivity of the graph, and increasing the k-value usually increases graph connectivity. It may also be that the default parameters suggested by the authors are more suitable for the $k = 100$ case.

All methods seem to perform in a similar way regardless of the distance metric $L_{0.1}$, L_1, L_2, L_{10}. The ZNP$^+$ method is the best in all tests where the metric was varied, although the performance gain is somewhat smaller with the $L_{0.1}$ distance.

ZNP$^+$ converges to near exact (98.0%–99.9%) results in all cases with the same configuration parameters ($W = 2k$, $\gamma = 0.3$) and stop condition ($\delta = 0.0001$), regardless of the choice of k or the dataset. For comparison, the quality of the NN-Descent algorithm varied much more (78–100%) when k changed from 10 to 100. This is important for practical situations where the users typically cannot calculate the recall rate of the end results and need to rely on the recommended parameters.

4.6 Predicting Errors

From the results in Figure 4, it can be seen that the quality versus time performance of the ZNP method degrades after roughly 90% recall rate, and further iterations provide only minor improvement. For Sift1M dataset and $k = 100$, ZNP reaches 96.0% accuracy in 1,274 seconds whereas the

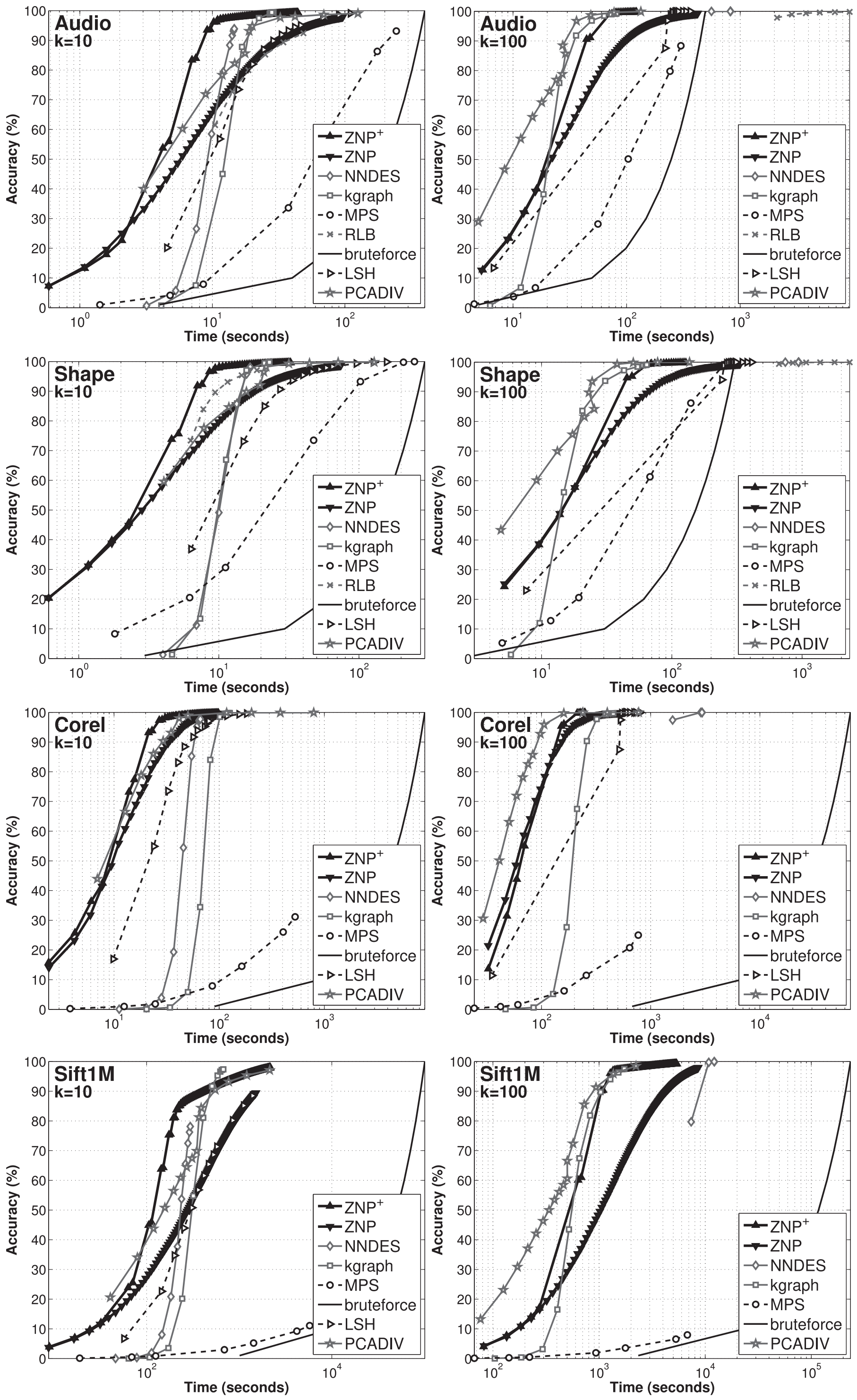

Fig. 4. Results: Time versus quality tradeoff.

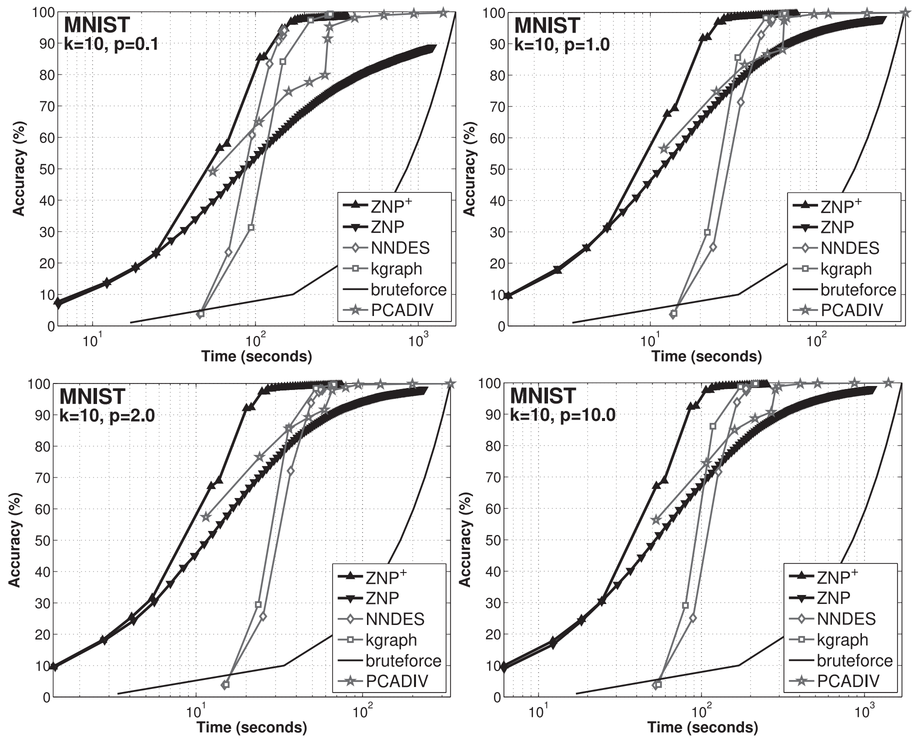

Fig. 5. Results: Time versus quality tradeoff for 784-dimensional MNIST dataset. Varying Minkowski distance parameter *p*.

final result of 99.3% takes 5,393 seconds in total. The same phenomenon happens also with other algorithms [35, 38]. The reason for this diminishing improvement has so far not been studied. We next provide some insight to this problem by analyzing which kind of points cause inaccuracies.

We calculate two statistical properties of every point:

$$\textbf{Cohesion}: \quad P_{N=NN}(x) = \frac{1}{k^2} \sum_{y \in \mathrm{kNN}(x)} |\mathrm{kNN}(x) \cup \mathrm{kNN}(y)| \tag{6}$$

$$\textbf{Inlierness}: \quad P_{x=NN}(x) = \frac{1}{k} \sum_{y \in \mathrm{kNN}(x)} \begin{cases} 1 & \text{if } x \in \mathrm{kNN}(y) \\ 0 & \text{else} \end{cases} \tag{7}$$

Cohesion measures the probability that a neighbor's neighbor of point x is among the neighbors of x. This relates to the neighborhood propagation method, which constructs a *k*NN-graph by testing the neighbors of neighbors of every point. Therefore, this method is dependent on a reasonable probability for $P_{N=NN}$.

Inlierness measures the probability that x is among the neighbors of its neighbor. In other words, it is the number of mutual neighbors of x relative to k. This measure also relates to node *indegree number* measure previously used for outlier detection in Ref. [22]. The difference is that we disregard edges from non-neighbors and normalize to [0, 1] range. Still, we consider both inlierness and cohesion to indicate how likely a point is an outlier.

We study next how the errors are distributed in the graph provided by ZNP and KGRAPH for the datasets Audio and Sift1M, using neighborhood size $k = 10$. We divide the points into two equal size classes based on their cohesion: points whose cohesion score is higher than median cohesion (condensed) and others (sparse). Another categorization is made based on their inlierness: points whose inlierness score is higher than the media score (inliers) and others (outliers).

We then study how the errors in the graph are distributed among the high and low classes. Since their sizes are equal, the number of errors is expected to be distributed 50%–50% between the classes. Figure 6 reveals that this is not the case. Only when there are lots of errors in the graph, they are equally distributed. However, the less there are errors, the more often they appear in the outlier class. For example, with 99% accurate KGRAPH results for the Audio dataset, there are 5,551 errors in total, of which 4,847 (88%) are within the outlier class. The phenomenon is similar with the cohesion measure.

From the results, we conclude that both sparsity and outlierness of the point highly correlate with the probability of whether it has errors. This has two consequences:

— It is expected that many of the approximation errors appear with the outlier points, which are less important for many applications.
— The cohesion and inlierness scores could be used to indicate how much longer the neighborhood propagation should be continued. For points with higher score, the search could be terminated earlier.

4.7 Effect of Dimensionality Reduction on Z-order Search

In Section 3.2, we introduced a dimensionality reduction method to be used with the Z-order curve. Here, we study how this dimensionality reduction affects the performance of the ZNP algorithm. We vary target dimensionality D_z from *two* to full dimensionality of the data and record the corresponding accuracy of the graph. The results are shown in Figure 7.

Window width of $W = 20$ was used in these experiments. The algorithm was run separately for (a) only one iteration and (b) five iterations. The experiments were repeated 10 times for each D_z while other parameters remained constant.

From the results, we can see 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 is increased beyond the selected value of 32, the results improve only marginally. Additionally, when D_z grows, the program execution time also grows.

It can also be seen from Figure 7 that there is a moderate amount of variation in the quality of results between program runs. The variations originate 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.

4.8 Proximity Preserving Qualities of the Z-order

In Equation (3), we introduced a formal definition for the proximity preserving quality (Q). Here, we test how well the Z-order curve preserves the proximity of points on the datasets Shape, Audio, Corel, and Sift1M when using Euclidean distance. We take a subset of the first D dimensions of each dataset, and then project it to one dimension using Algorithm 3. We measure the quality Q for the curve using 100,000 random samples and 100 different projections. The mean and standard deviations of Q are reported in Figure 8.

The results show that Q decreases with dimensions but remains better than random change $p = 0.5$. The proximity preserving quality depends both on the data and the dimensions. The Q value is higher for the Shape dataset than for Sift1M, despite Shape having higher dimensionality. Proximity preserving quality also correlates with the accuracy of the kNN graph. For example, the

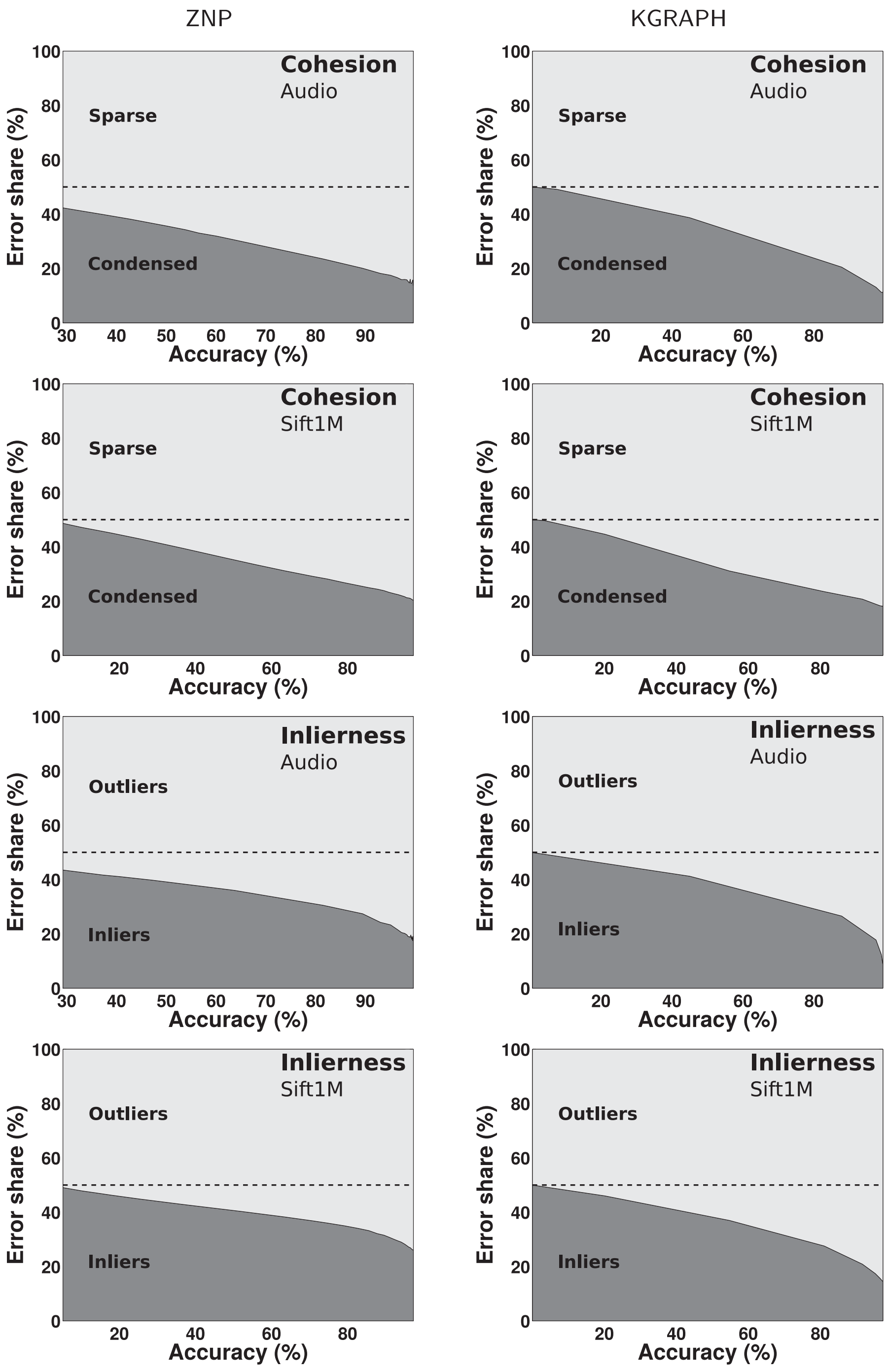

Fig. 6. The error distributions are shown for the results of ZNP (left) and KGRAPH (right). We use two measurements: inlierness and cohesion, and divide the points into two equal-size classes based on measurement scores. The share of errors for these classes is shown on the *y*-axis. The accuracy of the graph is shown on the *x*-axis.

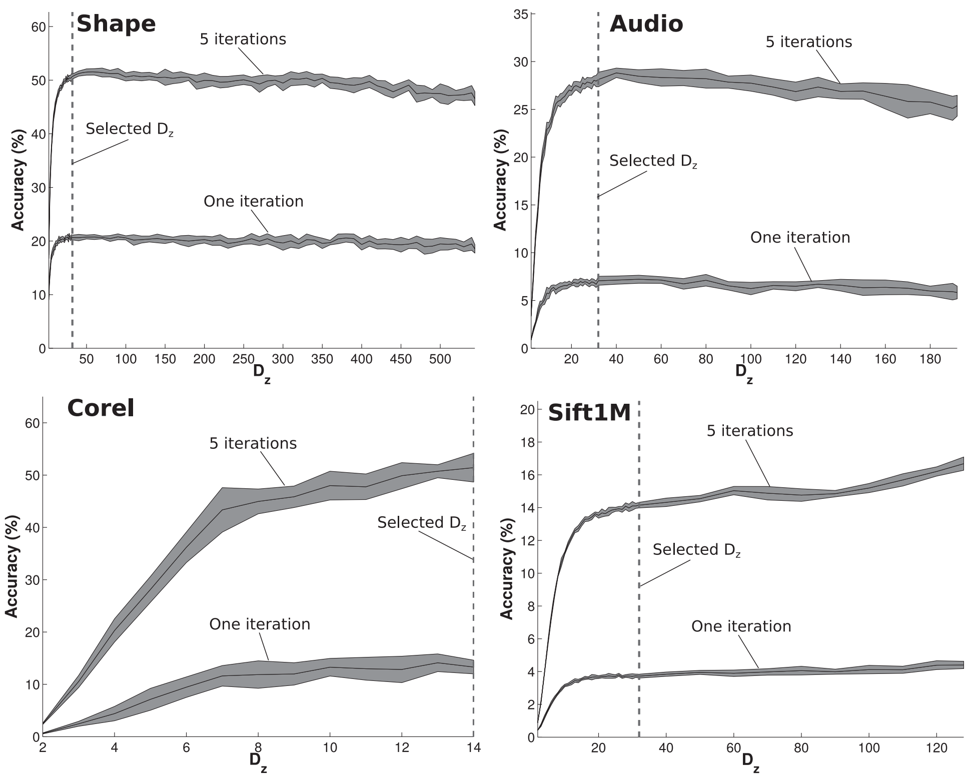

Fig. 7. The effect of dimensionality reduction (D_z) on recall rate of the ZNP algorithm illustrated for each dataset. Run for (a) one iteration and (b) five iterations. The area around the mean illustrates the standard deviation of the 10 repeats. The fixed D_z value used in other experiments is shown as a dashed line.

Corel dataset has the highest Q values, and the proposed Z-order search also performed best for this dataset (see Figure 4).

4.9 Space Complexity

In all of the compared algorithms, the data and the graph take a majority of the memory consumption. The space complexity is $O(Nk + Nd)$, and there should not be any signicant differences between the algorithms. In practice, KGRAPH consumes 51% more memory than the proposed method in the case of Sift1M data, but it is not clear whether the difference is due to the algorithms or their implementations.

5 CONCLUSION AND FUTURE WORK

In this work, we introduced a new method called ZNP for constructing a kNN-graph by using a combination of space-filling curves and neighborhood propagation. We compared it with five other methods using 14- to 784-dimensional datasets and different Minkowski distance metrics.

ZNP performed well, especially for a smaller neighborhood size ($k = 10$) and for the 14-dimensional Corel dataset consisting of 662,317 points, using the L_2 metric. In this case, it reached 90% quality level using only 73% of the time taken by the second best method PCADIV. ZNP performs well with even higher dimensional datasets. For the 784-dimensional MNIST data and L_1 metric, it reached a 90% quality level using only 45% of the time required by the second best

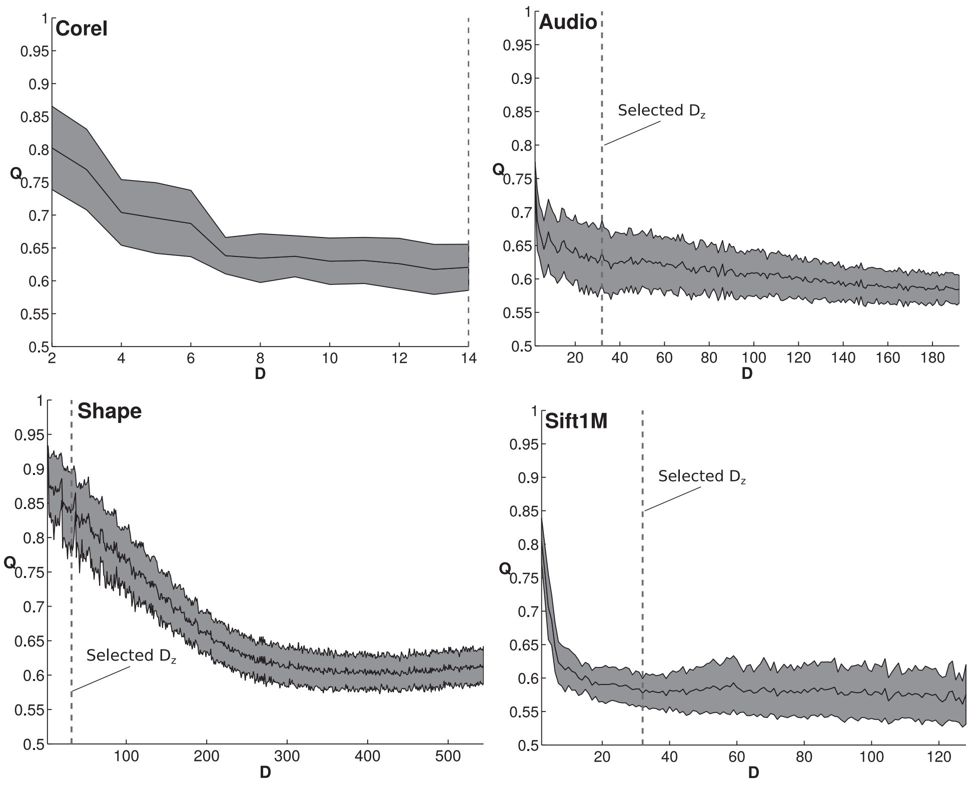

Fig. 8. The proximity preserving quality (Q) of the Z-order projection calculated on four datasets. We take a subset of the data with first D dimensions before projecting to the curve and vary D between 2 and the original dimensionality of the data. We use parameter $D_z = \min\{D, 32\}$ (see Algorithm 3). The gray area around the mean illustrates standard deviation of the 100 repeats.

method NNDES. With high neighborhood size ($k = 100$), the method PCADIV performed best in most cases. However, it was not clear if the differences between high and low neighborhood sizes was due to fundamental differences in the performance of the methods, or if the recommended default parameters just worked better in different situations.

We showed that errors in approximate *k*NN-graphs produced by our method and KGRAPH are not randomly distributed, but they appear more likely for outlier points. This outlier estimation was performed during runtime, without using ground truth. Therefore, it could also be used for improving search methods in future studies by, for example, running a more extensive search for outlier points.

REFERENCES

[1] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. 2001. On the surprising behavior of distance metrics in high dimensional spaces. In *Proceedings of the 8th International Conference on Database Theory (ICDT'01)*. Springer-Verlag, London, UK, 422–434.

[2] Nir Ailon and Bernard Chazelle. 2009. The fast Johnson-Lindenstrauss transform and approximate nearest neighbors. *SIAM J. Comput.* 39, 1 (2009), 302–322.

[3] Mikhail Belkin and Partha Niyogi. 2003. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation* 15, 6 (2003), 1373–1396.

[4] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1999. When is nearest neighbor meaningful? In *International Conference on Database Theory*. Springer, 217–235.

[5] Theodore Bially. 1969. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory* 15, 6 (1969), 658–664.

[6] Jonathan Brandt. 2010. Transform coding for fast approximate nearest neighbor search in high dimensions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 1815–1822.

[7] Hue-Ling Chen and Chang. 2011. All-nearest-neighbors finding based on the Hilbert curve. *Expert Systems with Applications* 38, 6 (2011), 7462–7475.

[8] Hue-Ling Chen and Ye-In Chang. 2005. Neighbor-finding based on space-filling curves. *Information Systems* 30, 3 (May 2005), 205–226.

[9] Jie Chen, Haw-ren Fang, and Yousef Saad. 2009. Fast approximate k NN graph construction for high dimensional data via recursive Lanczos bisection. *The Journal of Machine Learning Research* 10 (Sep. 2009), 1989–2012. http://www.jmlr.org/papers/v10/.

[10] Michael Connor and Piyush Kumar. 2010. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics* 16, 4 (2010), 599–608.

[11] Revital Dafner, Daniel Cohen-Or, and Yossi Matias. 2000. Context-based space filling curves. In *Computer Graphics Forum*, Vol. 19. Wiley Online Library, 209–218.

[12] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*. ACM, 577–586.

[13] Christos Faloutsos and Yi Rong. 1991. DOT: A spatial access method using fractals. In *Proceedings of the 7th International Conference on Data Engineering*. IEEE Computer Society, Washington, D.C., 152–159.

[14] Christos Faloutsos and Shari Roseman. 1989. Fractals for secondary key retrieval. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'89)*. ACM, New York, NY, 247–252.

[15] Pasi Fränti, Olli Virmajoki, and Ville Hautamäki. 2003. Fast PNN-based clustering using k-nearest neighbor graph. In *Proceedings of the IEEE International Conference on Data Mining (ICDM'03)*. IEEE, 525–528.

[16] Pasi Fränti, Olli Virmajoki, and Ville Hautämaki. 2006. Fast agglomerative clustering using a k-nearest neighbor graph. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28, 11 (2006), 1875–1881.

[17] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)* 3, 3 (Sept. 1977), 209–226.

[18] Cong Fu and Deng Cai. 2016. EFANNA: An extremely fast approximate nearest neighbor search algorithm based on kNN graph. *CoRR* abs/1609.07228 (2016).

[19] Vincent Garcia, Eric Debreuve, and Michel Barlaud. 2008. Fast k nearest neighbor search using GPU. In *Proceedings of the 2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. IEEE, 1–6.

[20] Irene Gargantini. 1982. An effective way to represent quadtrees. *Commun. ACM* 25, 12 (Dec. 1982), 905–910.

[21] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. 2011. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, Vol. 22. 1312.

[22] Ville Hautamäki, Ismo Kärkkäinen, and Pasi Fränti. 2004. Outlier detection using k-nearest neighbour graph. In *ICPR (3)*. 430–433.

[23] Herman Haverkort. 2010. Recursive tilings and space-filling curves with little fragmentation. *arXiv preprint arXiv:1002.1843* (2010).

[24] Alexander Hinneburg, Charu C. Aggarwal, and Daniel A. Keim. 2000. What is the nearest neighbor in high dimensional spaces? In *Proceedings of the 26th VLDB Conference*, Cario, Egypt.

[25] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC'98)*. ACM, New York, NY, 604–613.

[26] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[27] Swanwa Liao, Mario A. Lopez, and Scott T. Leutenegger. 2001. High dimensional similarity search with space filling curves. In *Proceedings of the 17th International Conference on Data Engineering (ICDE'01)*. 615–622.

[28] David G. Lowe. 1999. Object recognition from local scale-invariant features. In *Proceedings of the 7th IEEE International Conference on Computer Vision*, Vol. 2. IEEE, 1150–1157.

[29] Mohamed F. Mokbel and Walid G. Aref. 2011. Irregularity in high-dimensional space-filling curves. *Distributed and Parallel Databases* 29, 3 (2011), 217–238.

[30] Guy M. Morton. 1966. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company.

[31] Jack A. Orenstein and Tim H. Merrett. 1984. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. ACM, 181–190.

[32] S.-W. Ra and J. K. Kim. 1993. 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* 40, 9 (1993), 576–579.

[33] Rajeev Raman and David S. Wise. 2008. Converting to and from dilated integers. *IEEE Trans. Comput.* 57, 4 (2008), 567–573.

[34] Herbert Tropf and H. Herzog. 1981. Multidimensional range search in dynamically balanced trees. *ANGEWANDTE INFO.* 2 (1981), 71–77.

[35] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. 2012. Scalable k-NN graph construction for visual descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 1106–1113.

[36] Jens-Michael Wierum. 2002. Logarithmic path-length in space-filling curves. In *CCCG*. 22–26.

[37] Bin Yao, Feifei Li, and Piyush Kumar. 2010. K nearest neighbor queries and kNN-joins in large relational databases (almost) for free. In *Proceedings of the 2010 IEEE 26th International Conference on Data Engineering (ICDE)*. IEEE, 4–15.

[38] Yan-Ming Zhang, Kaizhu Huang, Guanggang Geng, and Cheng-Lin Liu. 2013. Fast kNN graph construction with locality sensitive hashing. In *Machine Learning and Knowledge Discovery in Databases*. Springer, 660–674.

[39] Xiaojin Zhu, John Lafferty, and Ronald Rosenfeld. 2005. *Semi-supervised Learning with Graphs*. Ph.D. Dissertation. Carnegie Mellon University, Language Technologies Institute, School of Computer Science.

Received March 2017; revised March 2018; accepted August 2018