High dimensional NN search using Hilbert space filling curve

Mingpeiyu Zhang
June 2017

1 Introduction

Nowadays in the new data driven era nearest neighbour search for high dimensional data has become increasingly important. It has been applied broadly in areas such as data mining, image processing and pattern recognition, and is frequently used as a classification or clustering method in machine learning or data mining. When a new framework or algorithm is proposed, nearest neighbour is usually the first baseline method researchers want to compare with. So how to efficiently search the nearest neighbour given high dimensional data has been a popular research direction where a great amount of effort was devoted into over the past decades.

Since the data is high dimensional, most solutions can be extremely slow by using linear scan and compute the distances over and over again. Gast et al. [1] noted that while nearest neighbour searching algorithms are efficient for low dimensions, for a large number of dimensions even specialized algorithms can only give a minor performance given a sequence of search. And due to the fact that both the dimension of the data and the number of prototypes can be unlimited big, some researchers instead considered approximate methods [2, 3, 4, 5].

Other exact algorithms using data structures which have good performance on high dimensional data like k-d tree, ball tree can also be very efficient if we can make good use of GPU acceleration or other I/O efficient techniques to overcome the limitations due to conditional computations and suboptimal memory accesses. Ting et al. [6] introduced and evaluated two algorithms for more effectively exploiting spatial data structures during k-NN (k nearest neighbour) classification. However, as distance increases the performance of their solution degrades. Sproull et al. [7] presented a nearest neighbour search method by using k-d tree, which divides the training data exactly into half plane. However, it requires more computation and intensive search. Moreover, it blindly slices points into half which may miss data structure.

On the other hand, a space-filling curve orders points linearly to preserve the distance between two points in the space. This means that points which are close in space and represent similar data should be stored together in the linear sequence. Hilbert space filling curve has the good clustering property of preserving spatial proximity, so recently some researchers tried to apply it in high dimensional nearest neighbour searching. Marcelo et al. [8] presented a data-partitioning error-controlled strategy for solving the nearest neighbour search problem using spatial sorting as the basic building block where spatial sorting is a process of ordering d-dimensional points along a space-filling curve so that the result is still a unidimensional sequence. Although they got some good results, their approaches are limited in when the query point is already inside the dataset. Ge et al. [9] studied the k-NN operation in the context of MapReduce and used space filling curve to approximate the result. They used random shifts to the datasets so that z-values can preserve the spatial locality with proved high probability. Their results showed that the performance of this method is better than the baselines. However this is an approximation method and can only be applied in the context of MapReduce.

Recent years researchers proposed many nearest neighbour search methods based on a single curve. Gloria et al. [10] presented a new method for nearest neighbour search based on a single space-filling curve and the addition of artificially perturbed points to the data set and its results have been compared to a, more common, multiple space-filling method. Their results showed faster execution times in a memory based implementation and different real and artificial datasets. Meanwhile Fernando et al. [11] proposed an indexing scheme for very large multimedia descriptor databases based on the Hilbert curve. Swanwa et al. [12] presented an approach for approximate nearest neighbour queries for sets of high dimensional points. They used multiple shifted copies of the d-dimensional data points and stores them in up to $(d+1)$ B-trees, sorted according to their position along a space filling curve, they found that the approximation ratio in practice is better than theoretically derived bound.

Apart from single curve techniques, Multicurves [13] is an index for accelerating k-NN queries based on space-filling curves which uses several curves to mitigate boundary effects. One of the main challenges in the use of space-filling curves in similarity search regards the boundary effects that are a result of the existence of regions that violate the curves neighbourhood relation preserving property (i.e., the property that points that are close in the space should be mapped to points that are close in the curve). To overcome this problem, Multicurves uses multiple curves, expecting that in at least one of the curves the neighbourhood relations will be preserved. Each of the curves, however, is responsible for a subset of the dimensions, rather than all of the dimensions as is seen in the above mentioned methods. However, it is more effective to process several low dimensional queries than a single high-dimensional query.

In this work we focus on studying the practical performance of Hilbert space filling curve in high dimensional nearest neighbour search. We want to see how can we adapt the approach in order to get a competitive result compared to existing nearest neighbour search data structures. We will first show some preliminaries in Section 2 and then present the way we apply Hilbert space filling curve in searching the nearest neighbour in Section 3. In Section 4 we will introduce the dataset we use for the experiments and explain how we performed all the experiments as well as the experiment results. Finally in Section 5 is our conclusion from this work.

2 Preliminaries

Space filling curves are based on the assumption that any attribute value can be represented with some fixed number of bits. We define the number of bits as the level of refinement k. And we use d to denote the number of dimensions of the high dimensional vector.

Given a two-dimensional square space of size $N \times N$, where $N = 2^k$ with level of refinement $k \geq 0$, the 2D Hilbert curve recursively divides the space into four equal-sized blocks. Each block is given a sequence number which ranges from 0 to $N^2 - 1$. For example, Figure 1 shows the 2D Hilbert curve of $k = 1$ which linearly orders four blocks by four sequence numbers ranged from 0 to 3. Figure 2 shows the 2D Hilbert curve of $k = 2$ in which the sequence numbers range from 0 to 15. It is derived from the curve of refinement level $k = 1$ in Figure 1 with the reflection and rotation on the first and last blocks of the previous curve. Figure 3 shows the 2D Hilbert curve of refinement $k = 3$ in which sequence numbers range from 0 to 63. It is derived from the curve of the previous refinement level $k = 2$ after the similar procedure.

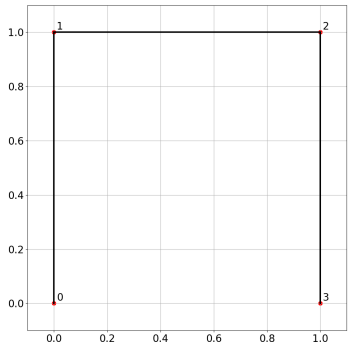

Figure 1: 2D Hilbert curve with 1 iteration.

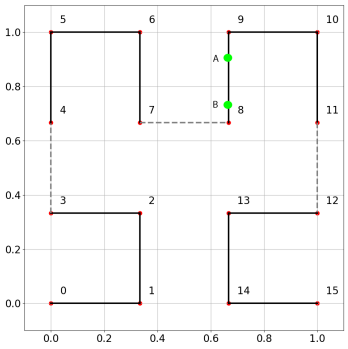

Figure 2: 2D Hilbert curve with 2 iterations.

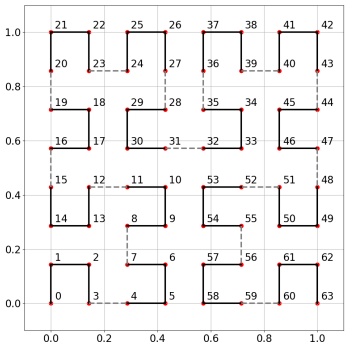

Figure 3: 2D Hilbert curve with 3 iterations.

However, the space filling curve has the limitation that two points close to each other may be mapped to different subcubes. As shown in Figure 2, point A and B are mapped to different blocks. This is the limitation one has to overcome when using it to query nearest neighbour.

Similarly, we can draw the 3D Hilbert curve given the level of of refinement k to be 1, 2 and 3, as shown in Figure 4, 5 and 6. And we can see that the bigger we set the level of of refinement k, the more subcubes we will end up with in the unit cube. And when mapping the unit cube into the unit interval, the more intervals we will divide the unit interval into. Generalising the concept into d dimensions, squares and quadrants are replaced by hyper-rectangles, successive hyper rectanglesshare common hyper-faces and a curve passes through 2^{kd} points.

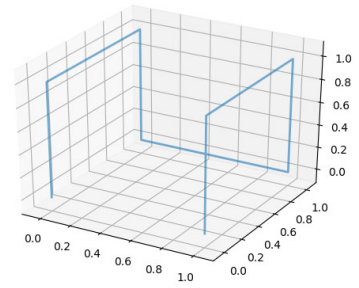

Figure 4: 3D Hilbert curve with 1 iteration.

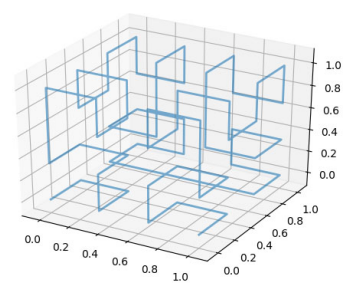

Figure 5: 3D Hilbert curve with 2 iterations.

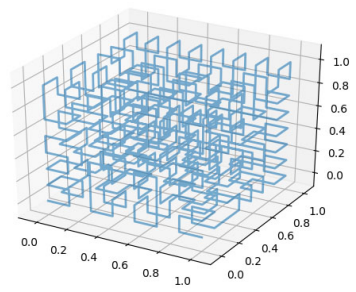

Figure 6: 3D Hilbert curve with 3 iterations.

When mapping high dimensional vector to the unit interval, we have to find out the exact subcube the corresponding point lies in, we can think of a traversal in the cube where each subcube we can map to a interval of the unit interval. For a given level of refinement k, we can divide the unit cue into 2^{kd} subcubes of equal size and divide the unit interval into 2^{kd} segments of equal length. It means that for each high dimensional vector, we can first find out the subcube id it lies in, then by using the mapping from the unit cube to unit interval, we can finally get a number representation of the vector which is between 0 and 1. Because the Hilbert curve has the better clustering property than the other curves, e.g., the Peano curve, we can then perform the nearest neighbour searching based on the Hilbert curve by using this mapped value.

3 Space filling curve based nearest neighbour search

The method we use is to transform the high dimensional vector to a number between 0 and 1 by using Hilbert space filling curve. Since the Hilbert curve has the good clustering property of preserving spatial proximity, points are close to each other are also close to each other on the curve, we can perform the nearest neighbour search by using the mapped value.

3.1 Mapping high dimensional vector into unit interval

Algorithm 1 COMPUTE HILBERT CURVE REPRESENTATION

procedure HILBERT(p, k, d) ▷ p is a high dimensional vector, k is number of iterations, d is the dimension of p
  $direction \leftarrow 1$; $unsgnedPrm[0, ..., d] \leftarrow [0, ..., d]$; $sgnsInvPrm[0, ..., d] \leftarrow [1, ..., 1]$; $result \leftarrow \emptyset$
  for $j \leftarrow 1$ **to** k **do**
    $entrAxs \leftarrow unsgnedPrm[d]$; $extAxs \leftarrow unsgnedPrm[d-1]$
    $quartAxs \leftarrow unsgnedPrm[d]$; $sbcubeId \leftarrow 0$
    for $i \leftarrow 1$ **to** d **do**
      $axis \leftarrow quartAxs$; $quartAxs \leftarrow unsgnedPrm[d-i]$; $sbcubeId \leftarrow 2 \cdot sbcubeId$
      $p[axis] \leftarrow 2 \cdot p[axis]$; $pInTheBack \leftarrow \lfloor p[axis] \rfloor$; $p[axis] \leftarrow p[axis] \mod 1$
      $sgnsInvChldPrm[axis] \leftarrow 1 - 2 \cdot pInTheBack$
      if $pInTheBack = isneg(sgnsInvPrm[axis])$ **then**
        $unsgnedChldPrm[i-2] \leftarrow extAxs$; $extAxs \leftarrow axis$
      else
        $unsgnedChldPrm[i-2] \leftarrow entrAxs$; $entrAxs \leftarrow axis$
        $sbcubeId \leftarrow sbcubeId + 1$
        $sgnsInvPrm[quartAxs] \leftarrow -sgnsInvPrm[quartAxs]$
    $unsgnedChldPrm[d-1] \leftarrow unsgnedPrm[1]$
    $unsgnedChldPrm[d] \leftarrow entrAxs + extAxs - unsgnedPrm[1]$
    if $sbcubeId \in \{0, 2d-1\}$ **then**
      swap $unsgnedChldPrm[d-1]$, $unsgnedChldPrm[d]$
    if $sbcubeId \geq 3 \cdot 2^d$ **then**
      $unsgnedChldPrm[1] \leftarrow unsgnedPrm[d]$
    $sgnsInvChldPrm[unsgnedPrm[1]] \leftarrow -sgnsInvChldPrm[unsgnedPrm[1]]$
    $orientation = unsgnedChldPrm[d]$
    if $sbcubeId \notin \{0, 2d-1\}$ **then**
      $sgnsInvChldPrm[orientation] \leftarrow -sgnsInvChldPrm[orientation]$
    if $direction = -1$ **then** ▷ $sbcubeId$ is in binary representation.
      flip all bits of $sbcubeId$
    $unsgnedPrm = unsgnedChldPrm$; $sgnsInvPrm = sgnsInvChldPrm$
    if $extAxs = orientation$ **then**
      $direction \leftarrow direction$
    $result = result \cup \{sbcubeId\}$
  return $result/2^{kd}$

Algorithm 1 shows the procedure we use Hilbert space filling curve to map the high dimensional vector to a number between 0 and 1. It takes 3 parameters as input: the high dimensional vector p, the refinement level k and the dimension of the vector d. There are two loops in the procedure, the outer one loops over the refinement level, each time compute a more refined subcube id of the unit cube. i.e., the first iteration divides the d-dimensional unit cube into 2^d subcubes, and in the inner loop computes an id between 0 and $2^d - 1$ which indicates in which subcube the corresponding point lies in. Then during the second iteration, it divides the subcube from the first iteration into 2^d subcubes, and gives an id between 0 and $2^d - 1$ which indicates in which subcube the corresponding point lies in. Finally we will get k subcube ids, each id is a number between 0 and $2^d - 1$. We can append all the binary representation of the ids together and divide it by 2^{kd} then the result is a number between 0 and 1.

3.2 Nearest neighbour searching based on binary search tree

After getting all the values for the training set, we then construct a binary search tree for all the obtained values, since the are all individual numbers and by using binary search tree one can quickly search the nearest value. For each query in the test set, we can first transform it to a number between 0 and 1, then query in the binary search tree for the nearest value and give the result. However in practice the result can be really bad since the points close to each other in the space may be mapped to different subcubes. We used some random shift methods to improve this limitation which will be introduced in Section 4.2.2.

4 Experiments

4.1 Data Description

The data we use to perform the experiments is from the competition **Predictive Web Analytics**[1] which is a discovery challenge co-located with ECML/PKDD 2014. It is a time series data which is from real-time analytics engine Chartbeat, which contains for a sample of URLs on the web, a time series of the number of pageviews of those URLs and the number of messages posted on Twitter and Facebook that include those URLs. There's in total $30,000$ URLs in the dataset. Besides, in the dataset the number of pareviews is recorded every 5 minutes, in total it contains the pageview data for 48 hours. We use the first 16 dimensions for each URL as the high dimensional data we analyze on.

Data preprocess:

- We first traverse the data and then accumulate up the number of pageviews. e.g. the original data is $[p_1, \ldots, p_{16}]$, then the data after the accumulation transform would be $[p_1, p_1 + p_2, \ldots, \sum_{i=1}^{16} p_i]$. This version of the time series might provide more insights of the website visit flows.
- Since the Hilbert Space Filling curve works on a unit cube, we then have to normalize the data into the unit cube. We achieve this by performing a logarithm transform on the data. We first traverse the dataset (which only contains the first 16 dimensions), pick the maximum number we have, say p_{max}. Then we define the base of the logarithm transform to be $2p_{max} + 1$. Here we assume that there's no value from the dataset can be 2 times

[1] `https://sites.google.com/site/predictivechallenge2014/description`

bigger than the maximum value p_{max}. e.g. the original data is $[p_1, \ldots, p_{16}]$, then the data after the accumulation transform would be $[(\log_{2p_{max}+1} p_1), \ldots, (\log_{2p_{max}+1} p_{16})]$.

After preprocessing the dataset, we then end up with a 16 dimensional dataset, and we used this data to perform the following experiments.

4.2 Different representations of the data and approaches

In order to test our method in practice and make our method to get the best performance, we need take into consideration of some aspects such as the format of the data and the strategies we use in the method. The following is the aspects we primarily tested on.

1. There are two primary ways to use the time series data, one is use it as raw data, the other is use it process it into accumulated version. Since the dataset we use is website visit flow data, as mentioned above in the preprocess part, the accumulated version of it might provide more insights.

2. A time series vector usually is a high dimensional data, which would take a huge number of bits if one wants to represent it by a space filling curve, it also takes more time to compute the curve given a very high dimensional data. One way we can simplify the data is to reduce the dimension of it. We consider using a recursive way to reduce the dimension of the data.

3. Due to the limitation of space filling curve, we know that some points are close to each other in the space might be in different cubes of the unit square we use to build the curve. We then can apply some random shifts to overcome this limitation.

Since the first aspect mentioned above is pretty straightforward, only a few preprocessing on the data would make the experiment executable. Let's now have a look at how can we make the other two aspects practical.

4.2.1 Recursive Representation of time series data

We designed such a way that the high dimensional time series data can be reduced into 3 dimensions, where the first dimension represents the mean level of the data, the second represents the level of the first half of the data vector and the third represents the second half of the data vector. The method we used is shown in Algorithm 2, where we recursively compute the mean level of first half and the second half of the time series vector, at last feed the resulting 3 dimensional vector to the Hilbert procedure as described above, and finally we will get a recursive representation of the time series data.

Algorithm 2 Dimension Reduction For the Time Series Data

procedure $\mathrm{DR}(q, k)$ ▷ q is a high dimensional vector
  if $k \geq 1$ **then**
    $length \leftarrow |q|$ ▷ $\mu(q)$ is the mean of the vector
    return $\mathrm{Hilbert}\left(\begin{bmatrix} \frac{1}{2}\mu(q) + \frac{1}{2} \\ \mathrm{DR}(q[0, \ldots, length/2], k/3) \\ \mathrm{DR}(q[length/2, \ldots, length], k/3) \end{bmatrix}, k, 3 \right)$
  else return 0

4.2.2 Apply random shifts

As space filling curve has the limitation that the points close to each other in the space may be allocated to different cubes during the construction of the space filling curves, the most common way to overcome it is to apply random shifts on the data. For the normal data (without recursive representation), we can directly apply some random shifts both on the training data and the test data and construct different data structure for each shift. Algorithm 3 shows the details of the normal random shift procedure and the query procedure.

As shown in Algorithm 3, during the construction of the data structure, we first construct 10 random shifts, then for each shift, we construct a binary search tree to store the value returned from the HILBERT procedure. So for 10 random shifts we will get 10 binary search tree structures. Then, during the query procedure, we first apply the shift on the query and search it in the corresponding binary search tree. Finally return the minimum distance we can find, since the time series vector of different web page could be duplicated. We can then compare this distance with the ground truth distance and check whether it is correct.

Algorithm 3 SHIFT ON NORMAL DATA

$trees \leftarrow \emptyset$, $shifts \leftarrow \emptyset$ ▷ Initialize the data structure
$k \leftarrow 100$ ▷ Initialize the number of bits we use
procedure CONSTRUCT($data$) ▷ $data$ is a collection of time series vectors
    for i **in** $[1, \ldots, 10]$ **do**
        $shift \leftarrow$ random_vector($|dt|$) ▷ random_vector() returns a random $|dt|$ dimensional vector
        $shifts = shifts \cup \{shift\}$
    for i **in** $[1, \ldots, 10]$ **do**
        $tree \leftarrow$ **new** $bstree$ ▷ Initialize a new binary search tree
        $shift \leftarrow shifts[i]$
        for dt **in** data **do**
            $hilbert\_curve \leftarrow$ Hilbert$(dt/2 + shift/2, k, |dt|)$
            $tree$.insert$(hilbert_curve)$
        $trees = trees \cup \{tree\}$
procedure QUERY(q) ▷ q is a high dimensional vector
    $results \leftarrow \emptyset$
    for i **in** $[1, \ldots, 10]$ **do**
        $tree \leftarrow trees[i]$
        $shift \leftarrow shifts[i]$
        $hilbert\_curve \leftarrow$ Hilbert$(q/2 + shift/2, k, |q|)$
        $result \leftarrow tree$.query$(hilbert_curve)$
        $result\_dis \leftarrow$ dis$(q, result)$ ▷ Compute the distance from the query vector
        $results = results \cup \{result\_dis\}$
    return min$(results)$ ▷ Return the minimum distance found

Apart from the normal representation of the data, we also want to apply the random shift on the recursive representation of it. Since the construction of the data is a recursive procedure, at each recursive level we apply the Hilbert procedure twice, we have to construct a random shift for each call of the Hilbert procedure. For simplicity, we choose $k = 81$, which is 3^4, which

means that the level of the recursive tree is 5. Furthermore, since in each level we need to call the Hilbert procedure twice, so in total we need $2^5 = 32$ shifts for the construction of the recursive representation. Note that each shift is 3 dimensional this time, since at last we will reduce the dimension of the time series data into 3.

In order to achieve this, the procedure DR which we use to construct the recursive representation should also be changed to adapt the random shift. Algorithm 4 shows the details of the recursive shift procedure and the query procedure.

Algorithm 4 SHIFT ON RECURSIVE DATA

$trees \leftarrow \emptyset$, $shifts \leftarrow \emptyset$ ▷ Initialize the data structure
$k \leftarrow 81$ ▷ Initialize the number of bits we use
procedure DR_SHIFT($q, kk, count, idx$) ▷ q is a high dimensional vector, *count* is the index of shift
  if $k \geq 1$ **then**
    $length \leftarrow |q|$
    $new\_vector \leftarrow \begin{bmatrix} \frac{1}{2}\mu(q) + \frac{1}{2} \\ \text{DR}(q[0, \ldots, length/2], kk/3, count, idx + 3) \\ \text{DR}(q[length/2, \ldots, length], kk/3, count, idx + 6) \end{bmatrix}$
    for i **in** $[1, 2, 3]$ **do**
      $shifted\_vector[i] \leftarrow new\_vector[i]/2 + shifts[count][idx + i]/2$
    return Hilbert($shifted\_vector$, kk, 3)
  else return 0
procedure CONSTRUCT($data$) ▷ *data* is a collection of time series vectors
  for i **in** $[1, \ldots, 10]$ **do**
    $shift \leftarrow$ random_vector$(3 * 32)$ ▷ random_vector() returns a random $3 * 32$ dimensional vector
    $shifts = shifts \cup \{shift\}$
  for i **in** $[1, \ldots, 10]$ **do**
    $tree \leftarrow$ **new** *bstree* ▷ Initialize a new binary search tree
    for dt **in** data **do**
      $hilbert\_curve \leftarrow$ DR_Shift$(dt, k, i, 0)$
      $tree$.insert$(hilbert_curve)$
    $trees = trees \cup \{tree\}$
procedure QUERY(q) ▷ q is a high dimensional vector
  $results \leftarrow \emptyset$
  for i **in** $[1, \ldots, 10]$ **do**
    $tree \leftarrow trees[i]$
    $shift \leftarrow shifts[i]$
    $hilbert\_curve \leftarrow$ DR_Shift$(dt, k, i, 0)$
    $result \leftarrow tree$.query$(hilbert_curve)$
    $result\_dis \leftarrow$ dis$(q, result)$ ▷ Compute the distance from the query vector
    $results = results \cup \{result\_dis\}$
  return min($results$) ▷ Return the minimum distance found

4.2.3 Results

All the experiments were performed on a x64 Linux PC with an AMD Athlon^{tm} X4 760K 3.8GHZ processor, 8.0GB memory.

We applied the above mentioned methods both on the accumulated data and the raw time series data from the dataset we chose. Figure 7 and Figure 8 show the results we obtained. We can see that in both accumulated data and the raw time series data, the original data representation with random shifts gets a higher accuracy. Besides, the accumulated version of the data gets a higher accuracy in general. The best performance we can get out of all the 8 scenarios is the one with accumulated data and normal representation with shift on the data, which gets an accuracy of 73.9%.

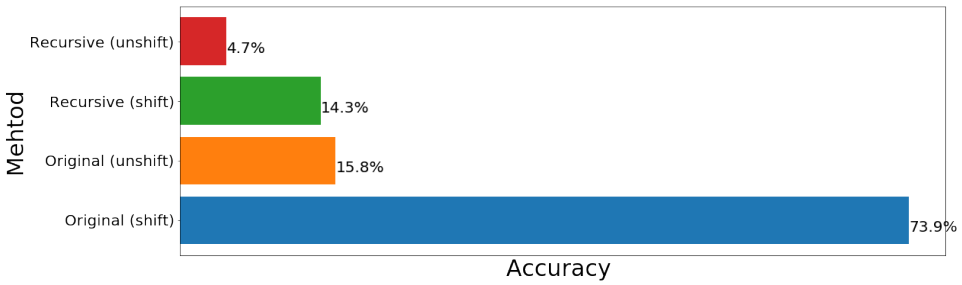

Figure 7: Experiment results on the accumulated data.

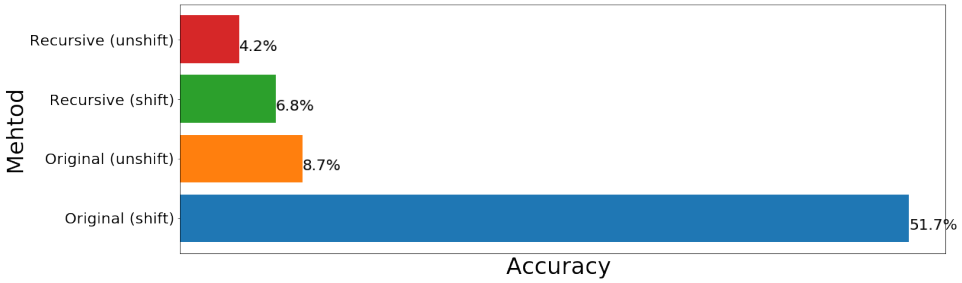

Figure 8: Experiment results on the raw time series data.

It is a little disappointed to see the recursive representation of the data does not get a better result. It might be because that when we transform the data recursively, some features of the data gets lost, so when querying the nearest neighbours the recursive representation of the data can be not really useful.

4.3 Performance analysis

From the previous experiments we know that the accumulated data and normal representation with shift on the data gets the best performance out of the 8 scenarios. However, it still gets 26.1% missed queries. Now the question remains how good are these 26.1% missed queries as an estimation of the nearest neighbour? To answer this question, we have to perform some checks on the results we obtained from the missed queries. We checked the approximation ratio of these missed queries as well as the number steps we need to take in the binary search tree in

order to get to the true nearest neighbour.

Figure 9 and Figure 10 show the results of the above mentioned question (for both missed and non missed queries). We can see that on average the number of steps we need to take to get to the true nearest neighbour is 0.8 and the approximation ratio of the missed queries on average is pretty close to 1, and all the values are smaller than 2. The reason for this result still need to be confirmed theoretically.

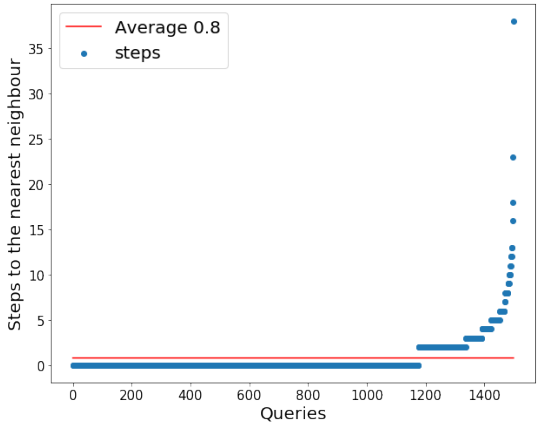

Figure 9: Steps it takes for queries.

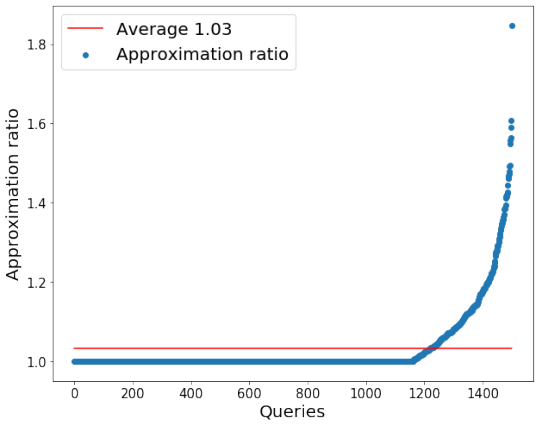

Figure 10: Approximation ratio for queries.

4.4 Comparison with other data structures

We used 12 existing nearest neighbour query benchmarks[2] we can find to evaluate the performance of our method. We first used these methods to query the 1-NN of each query point then compare the run time for each query among all the algorithms. Figure 11 shows the run time for each query of each method. We can observe that our method has a run time below 120 ms for all the queries. Compared to other methods, the run time is a little bit slow primarily because for each query we need to compute 10 random shifts of it and for each random shift we need to compute the Hilbert representation of it, which would take a huge amount of iterations.

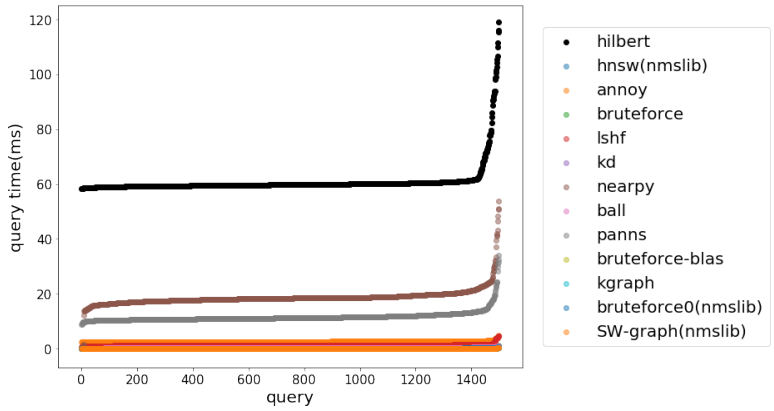

Figure 11: Comparison of run time (sorted)

Besides, for each query result we checked if it is the true nearest neighbour, and then compute

[2]Source code from `https://github.com/erikbern/ann-benchmarks`.

the number of missed queries for each algorithm. From Figure 12, we can see that many of the benchmarks do have a very good performance, they barely have missed queries. However the results from "nearpy" and "lshf" seem not inline with other methods. Figure 13 shows the approximation ratio of the queries against the query time. It also contains the correct queries of which the approximation ratios are 1. It's not clear in the figure but our method and most of the benchmarks have an approximation ration lower than 2 for all the queries.

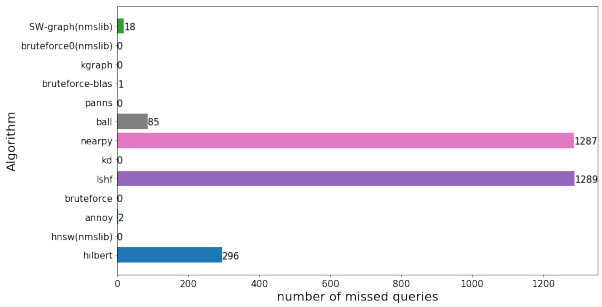

Figure 12: Number of missed queries.

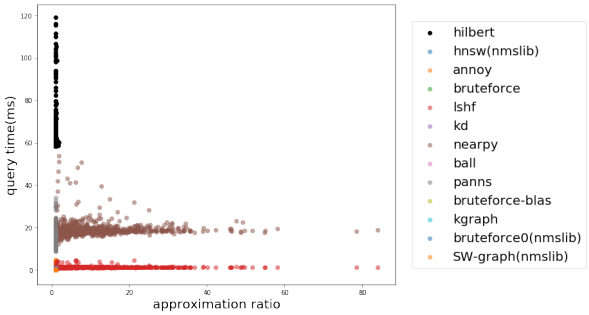

Figure 13: Approximation ratio vs query time for 1-NN.

Since the results from "nearpy" and "lshf" seem not inline with other methods, we then repeated the experiments without these two methods. The results are shown in Figure 14 and Figure 15 respectively. We can observe that when querying 1-NN, most of the methods have a low run time and good approximation ratio. The method "panns" has a good accuracy but the run time of it is slow compared to other methods. The accuracy of our method is disappointing compared to other benchmark methods. Our method takes more time, as discussed above, it is due to the huge amount of iterations. However the approximation ratio of it is always below 2.

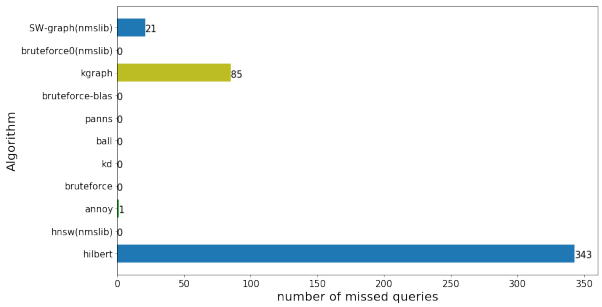

Figure 14: Number of missed queries for 1-NN.

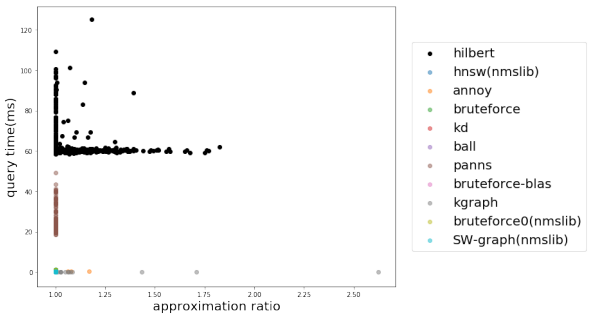

Figure 15: Approximation ratio vs query time.

Apart from the querying for 1-NN, we also checked if the query result is within the 10 nearest neighbour. Again we check the run time, the missed queries and the approximation ratio. The run time for each query is pretty much the same as the result shown in Figure 11. While for the number of missed queries, as shown in Figure 16, our method has a pretty high accuracy, which is inline with other algorithms. It is probably because our method has a good approximation ratio each time, and from Figure 17 we can see that the 3 missed queries actually all have a high approximation ratio compared to 1. Again, as shown in Figure 17, our method always has

an approximation lower than 2, however it takes more time to compute all the random shift representations.

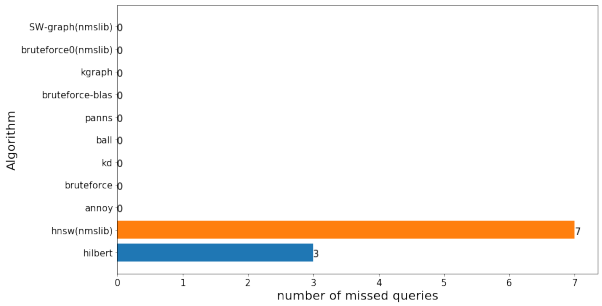

Figure 16: Number of missed queries for 10-NN.

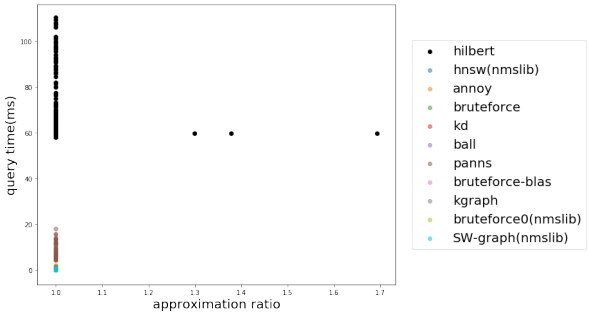

Figure 17: Approximation ratio vs query time for 10-NN.

5 Conclusion

In this work we practically studied the nearest neighbour searching on high dimensional data (time series data) based on Hilbert space filling curve. Due to the limitation of the space filling curves, two points close to each other in space may be mapped to different subcubes. The result of the method without random shift can be 60% accuracy lower than the shifted version. The result shows that by using random shift, the space filling curve is actually inline with the existing nearest neighbour searching methods when searching 10-NN. Besides, the approximation of our method is inline with other methods for both 1-NN search and 10-NN search. However, by performing the random shifts, the run time for each query gets higher as a result of the huge amount of iterations, which means that one has to make a trade off between the performance and the run time when using this technique.

Some future work would be theoretically study the behaviour of the Hilbert space filling curve, since the results show that the approximation ratio of the missed queries is always below 2, a good direction would be proving this result. Due to the limitation of time, we only performed the experiments on one dataset, another direction would be doing more detailed experiments on several datasets and try more possible changes on the method to make it more efficient (i.e., shift on the queries).

References

[1] Erik Gast, Ard Oerlemans, and Michael S Lew. Very large scale nearest neighbor search: ideas, strategies and challenges. *International Journal of Multimedia Information Retrieval*, 2(4):229–241, 2013.

[2] Sunil Arya and David M Mount. Approximate nearest neighbor queries in fixed dimensions. In *SODA*, volume 93, pages 271–280, 1993.

[3] Jeffrey S Beis and David G Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 1000–1006. IEEE, 1997.

[4] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[5] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[6] Ting Liu, Andrew W Moore, and Alexander Gray. New algorithms for efficient high-dimensional nonparametric classification. *Journal of Machine Learning Research*, 7(Jun):1135–1158, 2006.

[7] Robert F Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(1):579–589, 1991.

[8] Marcelo de Gomensoro Malheiros and Marcelo Walter. Spatial sorting: an efficient strategy for approximate nearest neighbor searching. *Computers & Graphics*, 57:112–126, 2016.

[9] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 38–49. ACM, 2012.

[10] Gloria Mainar-Ruiz and Juan-Carlos Perez-Cortes. Approximate nearest neighbor search using a single space-filling curve and multiple representations of the data points. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 2, pages 502–505. IEEE, 2006.

[11] Fernando Akune, Eduardo Valle, and Ricardo Torres. Monorail: A disk-friendly index for huge descriptor databases. In *Pattern Recognition (ICPR), 2010 20th International Conference on*, pages 4145–4148. IEEE, 2010.

[12] Swanwa Liao, Mario A Lopez, and Scott T Leutenegger. High dimensional similarity search with space filling curves. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 615–622. IEEE, 2001.

[13] Eduardo Valle, Matthieu Cord, Sylvie Philipp-Foliguet, and David Gorisse. Indexing personal image collections: a flexible, scalable solution. *IEEE Transactions on Consumer Electronics*, 56(3), 2010.