Fast Hilbert Sort Algorithm Without Using Hilbert Indices

Conference Paper *in* Lecture Notes in Computer Science · October 2016

DOI: 10.1007/978-3-319-46759-7_20

CITATIONS
7

READS
2,381

4 authors, including:

Takeshi Shinohara
Kyushu Institute of Technology
104 PUBLICATIONS **2,117** CITATIONS
SEE PROFILE

Kouichi Hirata
Kyushu Institute of Technology
164 PUBLICATIONS **591** CITATIONS
SEE PROFILE

Tetsuji Kuboyama
Gakushuin University
138 PUBLICATIONS **701** CITATIONS
SEE PROFILE

Fast Hilbert Sort Algorithm Without Using Hilbert Indices

Yasunobu Imamura[1(✉)], Takeshi Shinohara[1], Kouichi Hirata[1], and Tetsuji Kuboyama[2]

[1] Department of Artificial Intelligence, Kyushu Institute of Technology, Kitakyushu, Japan
imamura.kit@gmail.com
[2] Computer Centre, Gakushuin University, Toshima, Japan

Abstract. *Hilbert sort* arranges given points of a high-dimensional space with integer coordinates along a Hilbert curve. A naïve method first draws a Hilbert curve of a sufficient resolution to separate all the points, associates integers called *Hilbert indices* representing the orders along the Hilbert curve to points, and then, sorts the pairs of points and indices. Such a method requires an exponentially large cost with respect to both the dimensionality n of the space and the order m of the Hilbert curve even if obtaining Hilbert indices. A known improved method computes the Hilbert index for each point in $O(mn)$ time. In this paper, we propose an algorithm which directly sorts N points along a Hilbert curve in $O(mnN)$ time without using Hilbert indices. This algorithm has the following three advantages; (1) it requires no extra space for Hilbert indices, (2) it handles simultaneously multiple points, and (3) it simulates the Hilbert curve in heterogeneous resolution, that is, in lower order for sparse space and higher order for dense space. It, therefore, runs much faster on random data in $O(N\log N)$ time. Furthermore, it can be expected to run very fast on practical data, such as high-dimensional features of multimedia data.

1 Introduction

Hilbert curve [4], one of the space-filling curves, can be defined in any dimensionality. In Fig. 1 Hilbert curves in the 2-dimensional space of the first to the third order are shown. It is known that any interval of sorted objects along a Hilbert curve forms a relatively good cluster. For example, R-tree, which is one of the hierarchical spatial index structures, exhibits high performance when constructing such clusters [6]. In this paper we consider Hilbert sort problem which sorts given objects in high-dimensional space with integer coordinate values along a Hilbert curve.

This work was partially supported by Grant-in-Aid for Scientific Research 16H02870, 26280090, 15K12102, 26280085 and 16H01743 from the Ministry of Education, Culture, Sports, Science and Technology, Japan.

L. Amsaleg et al. (Eds.): SISAP 2016, LNCS 9939, pp. 259–267, 2016.
DOI: 10.1007/978-3-319-46759-7_20

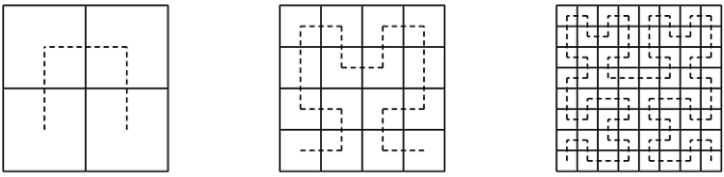

Fig. 1. Hilbert curves of the first order to the third order in two-dimensional space

A naïve method of Hilbert sort first draws the Hilbert curve of the m-th order with a sufficient resolution to separate all the points, associates integers called *Hilbert indices* representing the orders along the Hilbert curve to points, and then, sorts the pairs of points and indices. Such a method requires an exponential cost with respect to both the dimensionality n of the space and the order m of the Hilbert curve only for obtaining Hilbert indices [5]. A known improved method [1–3] computes the Hilbert index for each point in $O(mn)$ time. Thus, we have already known the Hilbert sort for N objects in n-dimensional space of m-th order can be solved in $O(mnN + mN\log N) = O(mN\log N)$ time.

In this paper, we propose an algorithm that directly sorts N points along a Hilbert curve in $O(mnN)$ time without using Hilbert indices, which was originally introduced by Tanaka [7][1]. This algorithm has three advantages; (1) it requires no extra space for Hilbert indices, (2) it handles simultaneously multiple points, and (3) it simulates the Hilbert curve in heterogeneous resolution, that is, in lower order for sparse space and higher order for dense space. As shown later, it can be observed to run very fast on practical data, such as multimedia data with high-dimensional features. It runs much faster on random data in $O(N\log N)$ time.

2 Outline of Proposed Algorithm

In this section, we explain the outline of the proposed algorithm by using an example. Let's consider 9 points in 2-dimensional space shown in Fig. 2. To separate all the points we have to draw the Hilbert curve of the third order, where the dimensionality n is 2, the order (resolution) m of Hilbert curve is 3, and the space is divided into $2^{mn} = 64$ subspaces. Without loss of generality we can deal with every point in the set represented by nm-bit unsigned integers. For example, the rightmost one in the 9 points is in the position (5, 6) represented by "101110".

Our algorithm simulates the Hilbert curve by dividing space into two in an axis. Here we trace it in a breadth-first manner, while it runs in a depth-first manner by a recursive call in a natural implementation.

We represent the start point, end point and crossing area of the Hilbert curve by an arc-like arrow. For example, Fig. 3 shows that the Hilbert curve crosses the square from the lower left corner to the lower right.

[1] We found that Tanaka's implementation has $O(mn^2N)$ running time.

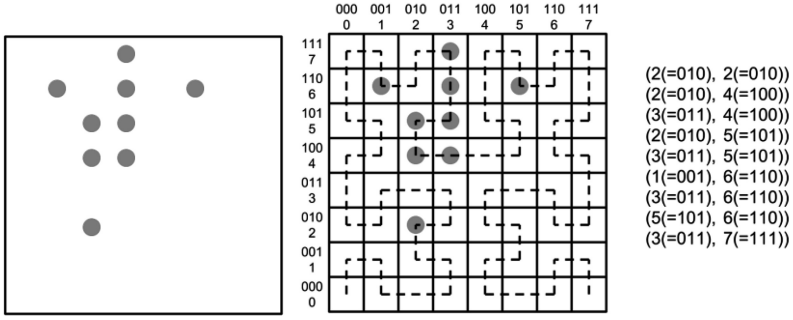

Fig. 2. 9 points separated by Hilbert curve of the third order

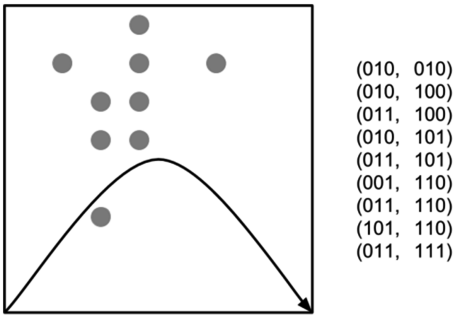

Fig. 3. Arc-like arrow of Hilbert curve

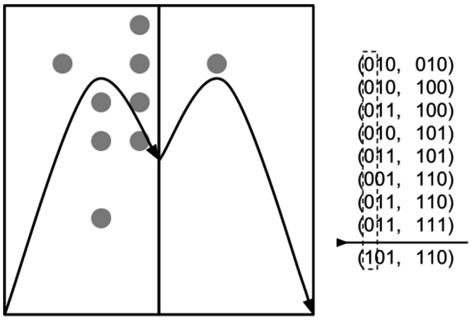

Fig. 4. Division in the horizontal axis

First, we divide the space into two in the horizontal axis as in Fig. 4. Note that this division is done by tests on the first bits, which are enclosed by a dashed line in Fig. 4. At this stage, 8 points in the left subspace are decided to precede a point in the right subspace. Since just one point exists in the right side, no more division is necessary,

The next division in the vertical axis is applied only to the left subspace (Fig. 5). Then the simulation of the first order of the Hilbert curve is complete.

The simulation of the second order is applied only to the upper left subspace (Fig. 6). The simulation of the third order is shown in Fig. 7.

Finally, in this example, to sort 9 points, 9 space divisions are done to generate 10 subspaces.

If we adopt the algorithm in [2, 3] to compute Hilbert indices for 9 points, then the number of simulations at the smallest level amounts to $mnN = 54$, where $m = 3$ is the order of Hilbert curve, $n = 2$ is the dimensionality and $N = 9$ is the number of points. In contrast, our algorithm requires only 9 simulations, which is the same as the number of divisions. Thus, the proposed algorithm can reduce the cost to sort objects.

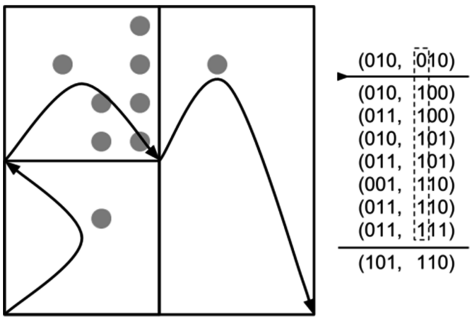

Fig. 5. Division in the vertical axis

It is obvious that the number of testing bits to compute Hilbert indices is $mnN = 54$, which is the same as the number of simulations. On the other hand, the number of testing bits by our sort algorithm without using Hilbert indices is $9 + 8 + 7 + 6 + (2 + 4) + (2 + 2 + 2) = 42$, which is the sum of the number of points in subspaces except leaves as shown in Fig. 8. Here the difference is not large, because m is small and many points need to be simulated in the largest order. However, the larger the order m is, the larger the difference is.

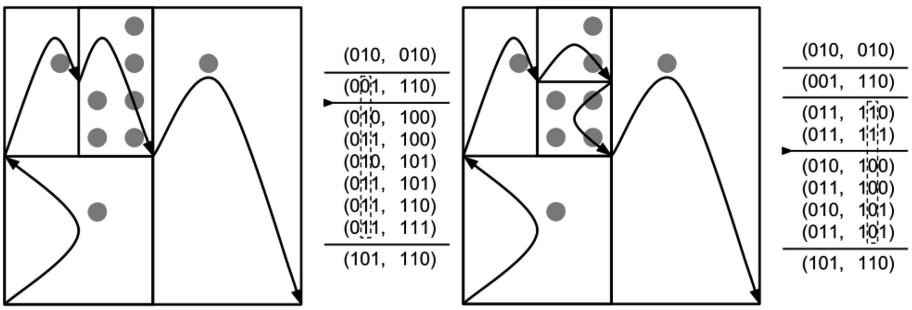

Fig. 6. Simulation of the second order

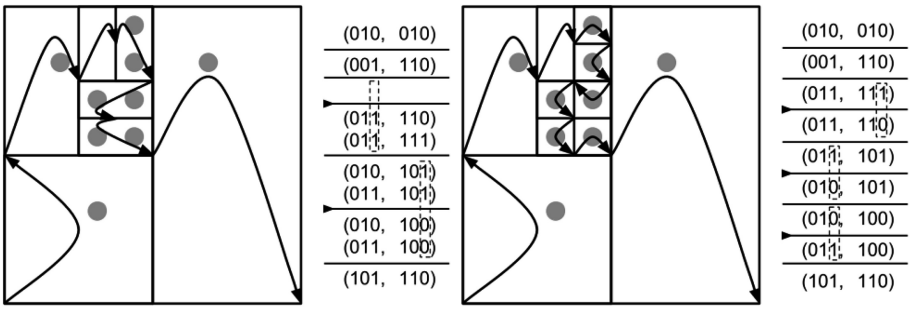

Fig. 7. Simulation of the third order

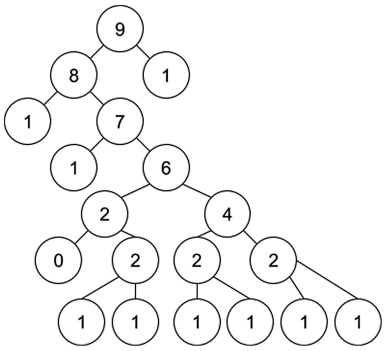

Fig. 8. The number of points in subspaces

3 Pseudo-code of Proposed Algorithm

In Fig. 9, we present C++ like pseudo-code of the proposed algorithm *Hilbert-Sort*, which consists of two functions `partition` and `HSort`. The constants `m` and `n`, which are the order of Hilbert curve and the dimensionality of data, are defined externally, such as macros. For clarity, here we use `bitset<m>` to represent coordinate values. In practice, we may use an integer type, such as `unsigned char` for $m = 8$, or `unsigned long` for $m = 32$. To sort N objects, we call `HSort` with parameters: `st = 0`, `en =` N `- 1`, `od = m - 1`, `c = 0`, `e = bitset<m>()`, `d = 0`, `di = false`, `cnt = 0`.

The function `partition` arranges data in a similar way as the famous partition function in quick sort. The function `HSort` is the main sorting function, which simulates Hilbert curve in almost the same way as in Hilbert-Index [2, 3]. Therefore we omit the formal proof of the correctness of simulation. The key difference of Hilbert-Sort from Hilbert-Index is the bit-wise simulation, which avoids redundant simulations to provide practical high speed.

4 Experiments

In this section, we demonstrate effectiveness of proposed algorithm Hilbert-Sort by running experiments. We use a library function sort in C++STL to sort using indices calculated by Hilbert-Index.

We use three kinds of data to be sorted, (1) random data, (2) random pair data, and (3) feature data extracted from images. Every bit of random data is expected to divide uniformly. A set of random pairs data is made by duplicating the half size random data, which is expected to derive the worst running time by the proposed algorithm because the deepest order simulation of Hilbert curve is necessary and the advantage of simultaneous simulations for multiple points is hard to preserve. For feature data, we prepare about 7 million image data extracted from video by processing grayscale transformation, down scaling and 2-dimensional FFT. Dimensionality of image features is 64. Each axis is represented by an 8 bits unsigned char.

```
int partition(
    bitset<m> *A[],     // array of points to be sorted
    int st, int en,     // represent range A[st], ... , A[en]
    int od,             // order of interest
    int ax,             // axis number to be divided
    bool di             // direction, false -> ascending, true -> descending
    )
{
  int i, j;
  i = st - 1;
  j = en + 1;
  while(true) {
    do i = i + 1; while(A[i][ax].test(od) == di);
    do j = j - 1; while(A[j][ax].test(od) != di);
    if(j < i) return i;     // partition is completed
    swap(A[i], A[j]);
  }
}

void HSort(
   bitset<m> * A[], // array of points to be sorted
   int st, int en,  // represent range A[st], ... , A[en]
   int od,          // current order of Hilbert curve
   int c,           // axis counter in current order
   bitset<m> & e,   // start positions of axes
   int d,           // axis number of first division on current order
   bool di,         // direction of previous division:
                    // false -> forward,  true -> backward
   int cnt          // number of continuous occurrences of di at same order
   )
{
  int p, d2;
  if(en <= st) return;  // nothing to do for empty or singleton
  p = partition(A, st, en, od, (d + c) % n, e.test((d + c) % n));
  if(c == n - 1) {         // simulation done, goto next order
    if(b == 0) return;  // no more order to simulate
    d2 = (d + n + n - (di ? 2 : cnt + 2)) % n;
    e.flip(d2); e.flip((d + c) % n);
    HSort(A, st, p - 1, b - 1, 0, e, d2, false, 0);
    e.flip((d + c) % n); e.flip(d2);  // undo of flips (2 before line)
    d2 = (d + n + n - (di ? cnt + 2 : 2)) % n;
    HSort(A, p, en, b - 1, 0, e, d2, false, 0);
  } else {
    HSort(A, st, p - 1, b, c + 1, e, d, false, di ? 1 : cnt + 1);
    e.flip((d + c) % n); e.flip((d + c + 1) % n);
    HSort(A, p, en, b, c + 1, e, d, true, di ? cnt + 1 : 1);
    e.flip((d + c + 1) % n); e.flip((d + c) % n);  // undo of flips
  }
}
```

Fig. 9. Pseudo-code of Hilbert-Sort

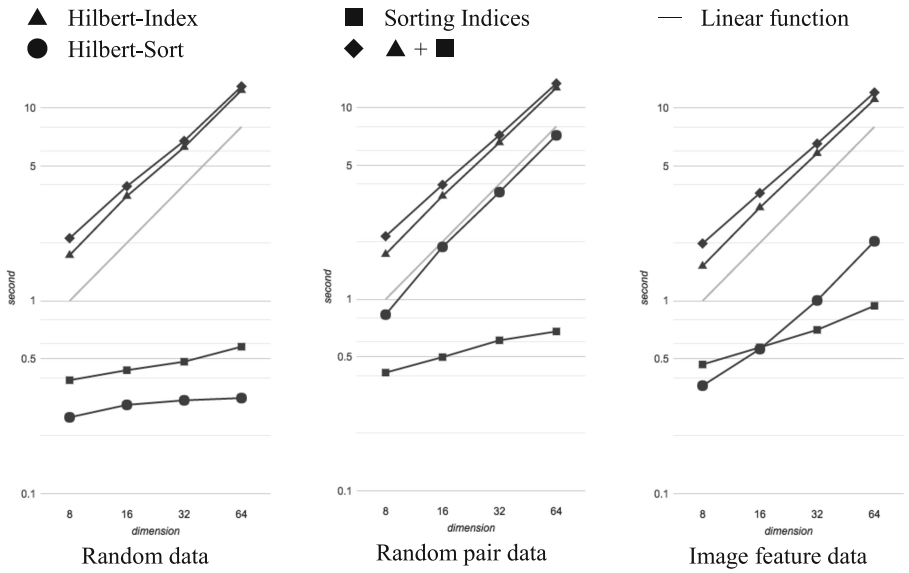

Fig. 10. Influence of dimensionality n

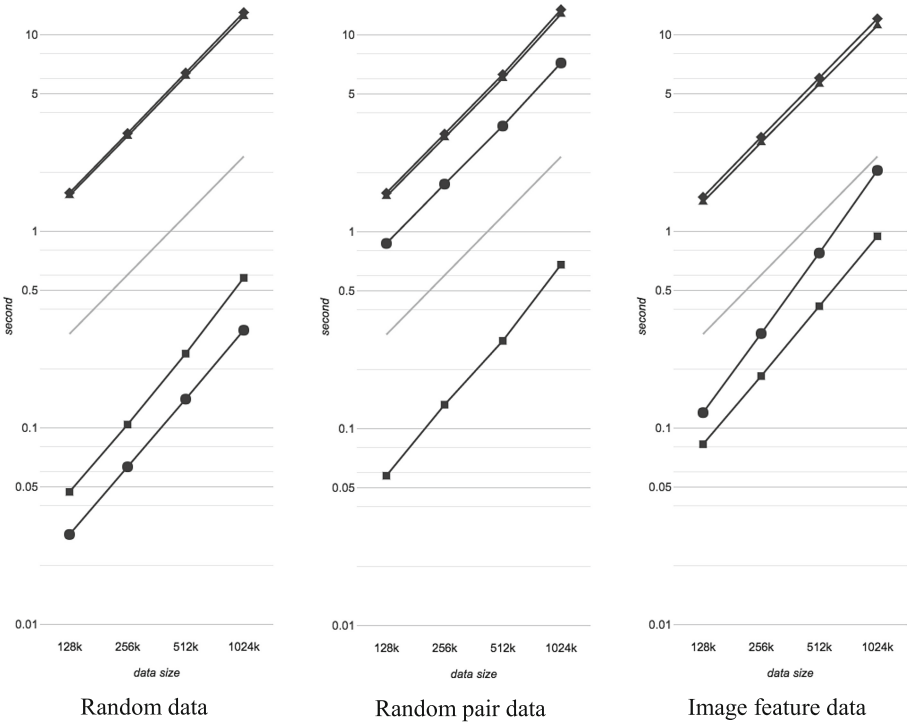

Fig. 11. Influence of data size N

In all the experiments, we fix $m = 8$, $n = 64$, $N = 1{,}048{,}576$ (=1024k) unless we explicitly vary the values. We present two graphs for each data to observe the effect of the dimensionality n and the number N of data. Every graph is plotted on a log-log scale.

As shown in Fig. 10, for random data, Hilbert-Sort runs extremely faster than Hilbert-Index in sublinear time with respect to n, whereas Hilbert-Index runs in just linear time. On the other hand, as shown in Fig. 11, for random data, both algorithms run in linear time with respect to the number N of data.

For random pair data, Hilbert-Sort has no advantage compared with Hilbert-Index in computational complexity, however, in our implementations, Hilbert-Sort runs about twice as fast as Hilbert-Index. (Figs. 10 and 11). For feature data, Hilbert-Sort runs about 5 times as fast as Hilbert-Index (Figs. 10 and 11).

From Fig. 11, the running time of Hilbert-Sort for feature data looks like worse than linear. The feature data are extracted from videos, which contain many similar data. Since we prepare the different sizes of images by random selection from 7 million data, the larger the size of data set is, the higher the probability of similar data is. Thus, the results for them are similar as ones for random data in smaller size, while they have many similar pairs like random pair data. Therefore, we can conclude that the running time of Hilbert-Sort is linear even for feature data.

Additional experiments on colors, one of SISAP databases, also show a similar behavior as on the image feature data.

5 Conclusion

Whereas we can observe no improvement in the order of computation time for practical feature data of images as expected unfortunately, we can achieve sufficient speed-up in practice. One of reasons for improvement can be explained by the advantage of Hilbert-Sort that simulates Hilbert curve in heterogeneous resolution. A similar technique for calculating Hilbert indices is possible with introducing *variable length* indices. However, when a new data set is added, even if we use variable length indices, we have to re-calculate indices for old data. Thus, Hilbert-Sort has more advantage in dynamic situations. We have already implemented Hilbert-Merge to add a bulk data set to sorted data without using Hilbert indices [8]. Finally, our laboratory can realize *compact Hilbert R-trees* as online versions of Hilbert R-tree [6] without Hilbert indices, which can also be used even in dynamical situations.

References

1. Butz, A.R.: Alternative algorithm for Hilbert's space-filling curve. IEEE Trans. Comput. **20**, 424–426 (1971)
2. Hamiltonm, C.: Compact Hilbert indices. Technical report CS-2006–07, Faculty of Computer Science, Dalhousie University (2006)

3. Hamilton, C.: Compact Hilbert indices: space-filling curves for domains with unequal side lengths. Inf. Process. Lett. **105**, 155–163 (2008)
4. Hilbert, D.: Uber die stetige Abbildung einer Linie auf ein Flachenstuck. Math. Ann. **38**, 459–460 (1891)
5. Kamata, S., Perez, A., Kawaguchi, E.: A computation of Hilbert's curves in N dimensional space. IEICE **J76-D-II**, 797–801 (1993)
6. Kamel, I., Faloutsos, C.: Hilbert R-tree: an improved R-tree using fractals. In: The 20th International Conference on Very Large Data Bases (VLDB), pp. 500–509 (1994)
7. Tanaka, A.: Study on a fast ordering of high dimensional data to spatial index. Master thesis, Kyushu Institute of Technology (2001). (in Japanese)
8. Tashima, K.: Study on efficient method of insertion for spatial index structure by using Hilbert sort. Master thesis, Kyushu Institute of Technology (2011). (in Japanese)