IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 16, 2020, accepted February 27, 2020, date of publication March 9, 2020, date of current version March 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2979250

MI-HCS: Monotonically Increasing Hilbert Code Segments for 3D Geospatial Query Window

YUHAO WU, XUEFENG CAO, AND WANZHONG SUN
Institute of Geospatial Information, Information Engineering University, Zhengzhou 450001, China

Corresponding author: Xuefeng Cao (cao_xue_feng@163.com)

This work was supported in part by the National Natural Science Foundation of China under Grant 41401465, and in part by the National Natural Science Foundation of China under Grant 41371384.

ABSTRACT Window queries are basic but important query tasks in geospatial databases. The Hilbert curve has good clustering properties, which can be used to effectively improve the execution efficiency of window queries. The ideal goal of a window query using Hilbert curve code is to quickly convert the query window to the corresponding monotonic continuous Hilbert code segments. However, existing algorithms have shortcomings in conversion calculations and Hilbert code segments properties. We propose the state vectors that are used to describe the filling rules of a three-dimensional Hilbert curve. In addition, we designed a direct generation algorithm for monotonically increasing Hilbert code segments (MI-HCS) for a three-dimensional window query. The MI-HCS algorithm is characterized by the direct generation of a monotonically increasing code segment set without the need to traverse all grid elements or the requirement of separate sorting steps. For a given query window $W(x, y, z, l, w, h)$ and a Hilbert curve of size $T \times T \times T$, the maximum complexity of our MI-HCS algorithm is $O\left(\alpha_1 \times \alpha_2 \times \left(\log_2 T + 1\right)\right)$, where $\alpha_1 = \text{median}(l, w, h)$ and $\alpha_2 = \max(l, w, h)$. The experimental results of the MI-HCS algorithm complexity are consistent with the specific theoretical analysis. The experimental results show that the Hilbert code segment generation efficiency of the proposed MI-HCS algorithm is 260.5% to 423.9% higher than that of existing algorithms.

INDEX TERMS Hilbert curve, octrees, spatial ordering code, window query.

I. INTRODUCTION

Spatial ordering can be defined as a reversible one-to-one correspondence between consecutive integers or key values and elements in a spatial entity set [1]. It has also been referred to as a spatial ordering code because these key values can decide the transversal order or address of the target [2]. Generally, spatial ordering codes divide a continuous space into regular grid elements to construct a one-dimensional curve traversing all grid elements without omission or repetition.

A Hilbert code describes the sequence in which a one-dimensional curve occupies the grid elements of a spatial target. The Hilbert curve is a one-dimensional curve that can recursively traverse the cells of a specified area [3] and is an important method for mapping spatial targets to one-dimensional spatial ordering codes [4]. Additionally, it has good spatial clustering [5], [6], meaning that spatial objects that are adjacent or close to each other in multidimensional space are mapped to the filling curve while retaining their original proximity relationship, which can effectively improve the access efficiency of multidimensional data in the physical one-dimensional storage of a disk [7]. These advantages have led to Hilbert codes being widely used in spatial database indexes [8], [9].

Window querying is an important basic query task in spatial databases [10], [11]. A Hilbert code can be used as the spatial target index to accelerate the window query rate. The process of performing a window query with a Hilbert code is generally divided into the following two steps [12], [13]:

1) Given a query window W, map all grid elements corresponding to the query window W to Hilbert codes, and concatenate the continuous Hilbert codes into code segments to produce a monotonically increasing code segment set as output;
2) The code segment set is put into the where condition of the database query, and searching is performed with

The associate editor coordinating the review of this manuscript and approving it for publication was X. Huang.

the help of one-dimensional indexing techniques such as B-tree.

Hilbert code generation efficiency and Hilbert code segment monotonicity are essential for efficient window queries.

Most existing Hilbert code generation research has focused on the generation of Hilbert codes for a single grid element [14], [15]. There are few studies specifically about generating Hilbert code segments for all grid elements in the query window. These generation algorithms can be made roughly iterative and recursive. The strategy of the iterative algorithm proposed in [16] and [17] is as follows. First, segment the entire Hilbert curve and calculate the smallest cell of the Hilbert code in the query window. Next, starting from the smallest cell of the Hilbert code, repeatedly determine whether the next grid element on each segment is in the query window. Finally, all Hilbert codes are combined, and the resulting code segment is given as output. The algorithm in [16] adopts a traversal strategy. When the size of the Hilbert curve and the query window increases, the number of grid elements to be calculated increases exponentially, and the efficiency decreases significantly. In [18], the concept of the maximum block in the quadtree [19] was introduced into the solution of the Hilbert code segment for two-dimensional Hilbert curves. First, a recursive method is used to generate all the maximum blocks of the quadtree in the range of the query window W, and then the Hilbert code correspondence to the quadtree maximum block is calculated. Finally, all code segments are assembled and sorted. The Hilbert code segment generation algorithm based on the maximum block of the quadtree has been applied to efficient image compression [20] and fast window querying [21]. After studying the filling order of the two-dimensional Hilbert curve [22], a code segment generation algorithm for recursive quad splitting of the query window is proposed. The efficiency of this algorithm is greatly improved compared with that in [18]. However, the algorithms in [16] and [22] only discuss two-dimensional Hilbert curves. Because the construction methods and hierarchical evolution laws of two- and three-dimensional Hilbert curves are completely different, it is impossible to combine the algorithms in [16] and [22] for direct application to the generation of a three-dimensional Hilbert curve. In [23], when studying the organization and management of the Hilbert-code-based large-scale point cloud data, the Hilbert curve was taken as a multidimensional tree; starting from the root node, child nodes at different levels were searched under the breadth-first policy until all child nodes corresponding to the multidimensional windows were located, after which the Hilbert code segment containing all child nodes was exported. However, the algorithm does not specify the order of searching for the child nodes, and it cannot guarantee that the output code segment set has a monotonically increasing nature. Therefore, it needs to sort the code segments before performing the continuous code segment merging step, which inevitably consumes additional $O(n)$ time.

To realize the efficient conversion from a three-dimensional query window to a Hilbert code segment, this paper proposes a direct algorithm for a monotonically increasing three-dimensional Hilbert code segment (MI-HCS) based on the study of the filling rule of three-dimensional Hilbert curves and the topological relationship between the query window and the space grid element. In the MI-HCS algorithm, a window query process is considered as the combination of the query window and the corresponding Hilbert curve. The process of the MI-HCS algorithm can be broadly described as follows:

1) Determine whether the current window size is equal to the curve size. If they are equal, the code segments corresponding to the current curve are added to the code segment set; if they are not equal, the window and the curve are divided to obtain sub-windows and sub-curves.
2) A new window query process is obtained by combining the sub-windows and sub-curves in pairs.

The whole flow of the algorithm is shown in Fig. 1 The rest of this paper is organized as follows. In Section II, we briefly describe the window query and the Hilbert curves filling rule. In Section III, we present the MI-HCS algorithm with no requirement of additional sorting steps. In Section IV, we analyze the performance complexity of the MI-HCS algorithm and compare the performance of our algorithm against existing algorithms. Section V presents concluding remarks.

II. HILBERT SPACE ORDERING CODE

A. HILBERT CODE SEGMENT CORRESPONDING TO THE QUERY WINDOW

The Hilbert curve first divides the n-dimensional space rule into seamless and non-overlapping $(2^m)^n$ hypercube grid elements. It then traverses every grid element with a one-dimensional curve without traversing any grid element more than once. This can also be stated as a one-to-one mapping between an n-dimensional space R^n and a one-dimensional space R^1. The three-dimensional Hilbert curve realizes a one-to-one mapping from three-dimensional space to one-dimensional space. The three-dimensional cube space with side length $T = 2^m$ is nested octrees. After m iterations of segmentation, 8^m sub-grid elements are obtained, and the length of each sub-grid element is $T/2^m$. The m-level sub-grid elements correspond to a curve of order m. The Hilbert code specifies the filling order of grid elements of the Hilbert curve. The value range of the mth order Hilbert code is $(0, 1, 2, \ldots, T \times T \times T - 1)$, as shown in Fig. 2 (a).

In three-dimensional space, the query window $W(x, y, z, l, w, h)$ can be regarded as a parallelepiped, of which the starting corner point P_{min} closest to the coordinate origin is (x, y, z), and the end corner P_{max} farthest from the coordinate origin is $(x + l, y + w, z + h)$. For a Hilbert curve of size $T \times T \times T$,the window query needs to take all Hilbert codes in window W, and integrate all codes into a code segment set $HRange = \{HR_1, HR_2, \ldots, HR_n\}$. Among them, HR_n is represented by the Hilbert filling start and end codes of the code segment, including the Hilbert codes of all grid elements between the start and end grid elements, namely

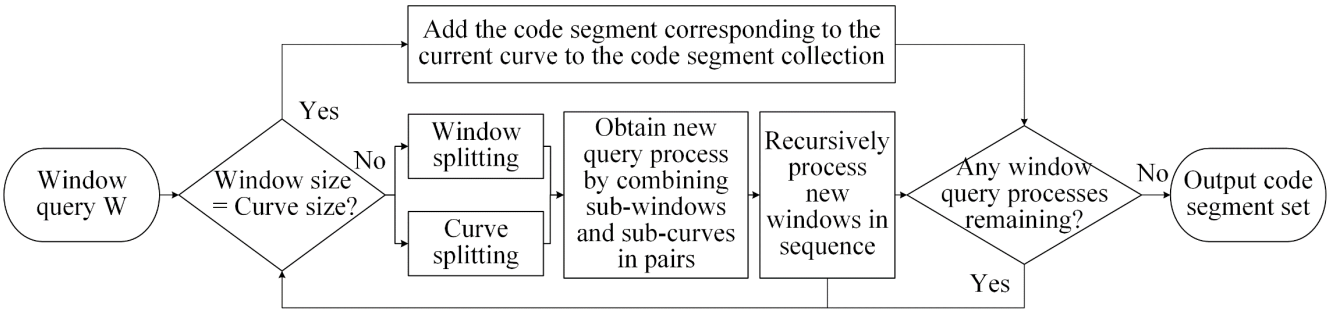

FIGURE 1. Flow of the MI-HCS algorithm presented in this paper.

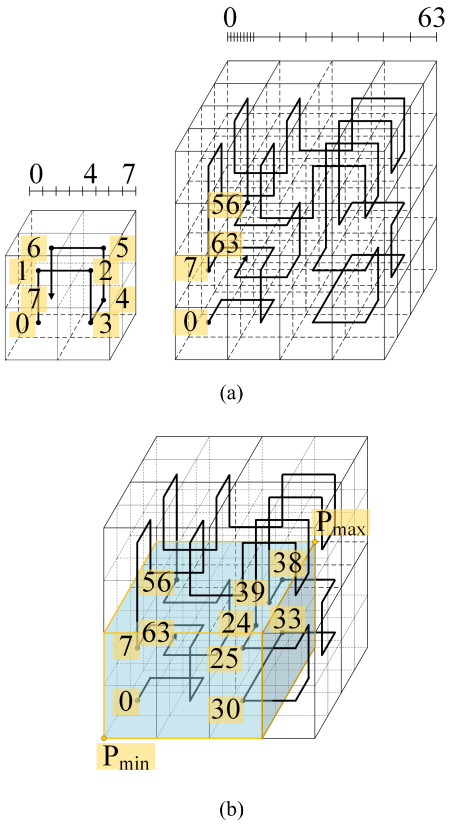

FIGURE 2. Hilbert code and Hilbert code segment corresponding to the query window. (a) Three-dimensional 1st and 2nd order Hilbert curves and Hilbert codes. (b) Hilbert code segment corresponding to the query window.

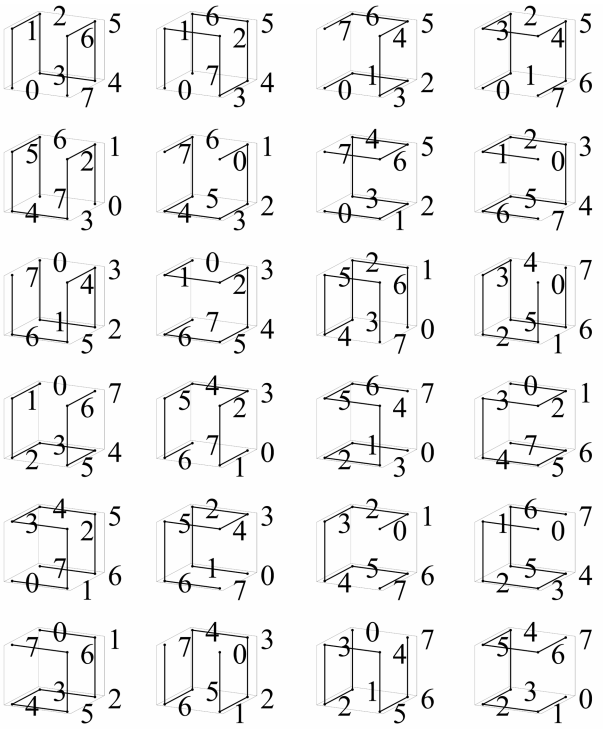

FIGURE 3. Sequence of Hilbert primitives.

$HR_n = (HStar_n, HEnd_n)$. Because the elements in *HRange* conform to the monotonically increasing nature, each code segment satisfies $HEnd_n < HStar_{n+1}$. As shown in Fig. 2 (b), the query window is a blue parallelepiped. The set of Hilbert code segments obtained are the following: $HRange = \{(0, 7), (24, 25), (30, 33), (38, 39), (56, 63)\}$.

B. HILBERT CURVE FILLING RULE

The Hilbert code is the arrangement order of m-level grid elements on the Hilbert curve. Therefore, mastering the filling rule of the curve between the grid elements is the premise of analyzing the Hilbert code. The self-similarity and self-replication characteristics of the Hilbert curve determine the order of the primitives of only 24 types in the three-dimensional Hilbert curve, which are shown in Fig. 3 [24], from left to right and top to bottom, respectively $\varphi_1, \varphi_2, \ldots, \varphi_{24}$.

After performing an octet division on the cube area containing the three-dimensional Hilbert curve, the eight sub-grid elements are numbered (0, 1, 2, 3, 4, 5, 6, 7), as shown in Fig. 4. The distinguishing criterion of primitive order is the filling state of the Hilbert curve in eight sub-grid elements.

To facilitate the description of the 24 primitive sequences and the calculation of the program, this paper introduces a state matrix to record these sequences. According to the sub-grid elements order of (0, 1, 2, 3, 4, 5, 6, 7), the number of sub-grid elements passed by k kinds of primitive sequences

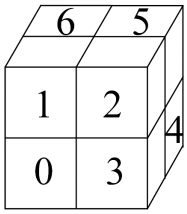

FIGURE 4. Numbered sub-grid elements.

before filled to the current sub-grid elements are recorded as state vector $\mathbf{s}_k$ ($k \in (1, 2, \ldots, 24)$). Then, the 24 row vectors are combined into a matrix $\mathbf{S}$, and the uth row of matrix $\mathbf{S}$ corresponds to state vector $\mathbf{s}_{u+1}$ and primitive sequence φ_{u+1}, as shown in (1).

$$\mathbf{S} = \begin{pmatrix} 0 & 1 & 6 & 7 & 4 & 5 & 2 & 3 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 7 & 4 & 3 & 2 & 5 & 6 & 1 \\ 0 & 3 & 4 & 7 & 6 & 5 & 2 & 1 \\ 4 & 5 & 2 & 3 & 0 & 1 & 6 & 7 \\ 4 & 7 & 0 & 3 & 2 & 1 & 6 & 5 \\ 0 & 7 & 6 & 1 & 2 & 5 & 4 & 3 \\ 6 & 1 & 0 & 7 & 4 & 3 & 2 & 5 \\ 6 & 7 & 4 & 5 & 2 & 3 & 0 & 1 \\ 6 & 1 & 2 & 5 & 4 & 3 & 0 & 7 \\ 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 \\ 2 & 3 & 0 & 1 & 6 & 7 & 4 & 5 \\ 2 & 1 & 6 & 5 & 4 & 7 & 0 & 3 \\ 6 & 5 & 2 & 1 & 0 & 3 & 4 & 7 \\ 2 & 5 & 4 & 3 & 0 & 7 & 6 & 1 \\ 4 & 3 & 2 & 5 & 6 & 1 & 0 & 7 \\ 0 & 3 & 2 & 1 & 6 & 5 & 4 & 7 \\ 6 & 5 & 4 & 7 & 0 & 3 & 2 & 1 \\ 4 & 3 & 0 & 7 & 6 & 1 & 2 & 5 \\ 2 & 1 & 0 & 3 & 4 & 7 & 6 & 5 \\ 4 & 7 & 6 & 5 & 2 & 1 & 0 & 3 \\ 6 & 7 & 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 & 7 & 0 & 1 \\ 2 & 5 & 6 & 1 & 0 & 7 & 4 & 3 \end{pmatrix} \tag{1}$$

For grid elements, the filling order of the Hilbert curve between its sub-grid elements is φ_k ($k \in (1, 2, \ldots, 24)$), and the filling order of the Hilbert curve between the grid elements after the sub-grid elements are divided again is determined by φ_k. This mapping relationship does not change with the level [25].

Considering the example shown in Fig. 5, the filling order of the level 2 curve is φ_2, and the corresponding state vector is $\mathbf{s}_2$. The filling order of the Hilbert curve between the sub-grid elements is $\varphi_3, \varphi_1, \varphi_2, \varphi_{10}, \varphi_3, \varphi_2, \varphi_5, \varphi_{10}$; the state vectors correspond to $\mathbf{s}_3, \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_{10}, \mathbf{s}_3, \mathbf{s}_2, \mathbf{s}_5, \mathbf{s}_{10}$, and there is a state vector level mapping of $T(\mathbf{s}_2) \rightarrow \mathbf{s}_3, \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_{10}, \mathbf{s}_3, \mathbf{s}_2, \mathbf{s}_5, \mathbf{s}_{10}$. This state vector mapping relationship does not change with the level, meaning that regardless of the grid element level, if the state vector is $\mathbf{s}_2$, the state vectors of its sub-grid elements are $\mathbf{s}_3, \mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_{10}, \mathbf{s}_3, \mathbf{s}_2, \mathbf{s}_5, \mathbf{s}_{10}$.

$$\mathbf{E} = \begin{pmatrix} 7 & 2 & 1 & 8 & 7 & 1 & 11 & 8 \\ 3 & 1 & 2 & 10 & 3 & 2 & 5 & 10 \\ 2 & 4 & 3 & 9 & 2 & 3 & 6 & 9 \\ 17 & 3 & 4 & 18 & 17 & 4 & 19 & 18 \\ 15 & 11 & 5 & 16 & 15 & 5 & 2 & 16 \\ 20 & 19 & 6 & 21 & 20 & 6 & 3 & 21 \\ 1 & 17 & 7 & 22 & 1 & 7 & 21 & 22 \\ 22 & 20 & 8 & 1 & 22 & 8 & 18 & 1 \\ 10 & 23 & 9 & 3 & 10 & 9 & 22 & 3 \\ 9 & 13 & 10 & 2 & 9 & 10 & 14 & 2 \\ 24 & 5 & 11 & 19 & 24 & 11 & 1 & 19 \\ 19 & 22 & 12 & 24 & 19 & 12 & 23 & 24 \\ 21 & 10 & 13 & 20 & 21 & 13 & 24 & 20 \\ 18 & 24 & 14 & 17 & 18 & 14 & 10 & 17 \\ 5 & 18 & 15 & 23 & 5 & 15 & 20 & 23 \\ 23 & 21 & 16 & 5 & 23 & 16 & 17 & 5 \\ 4 & 7 & 17 & 14 & 4 & 17 & 16 & 14 \\ 14 & 15 & 18 & 4 & 14 & 18 & 8 & 4 \\ 12 & 6 & 19 & 11 & 12 & 19 & 4 & 11 \\ 6 & 8 & 20 & 13 & 6 & 20 & 15 & 13 \\ 13 & 16 & 21 & 6 & 13 & 21 & 7 & 6 \\ 8 & 12 & 22 & 7 & 8 & 22 & 9 & 7 \\ 16 & 9 & 23 & 15 & 16 & 23 & 12 & 15 \\ 11 & 14 & 24 & 12 & 11 & 24 & 13 & 12 \end{pmatrix} \tag{2}$$

An evolution matrix $\mathbf{E}$ is introduced to record the mapping relationship between the 24 state vectors and the next-level state vectors, as shown in (2). To explain the meaning of row u of state evolution matrix $\mathbf{E}$, row u ($u \in (0, 1, \cdots, 23)$) corresponds to the mapping relationship $T(\mathbf{s}_{u+1})$ in the following manner:

$$T(\mathbf{s}_{u+1}) \rightarrow \mathbf{s}_{\mathbf{E}[u][0]}, \mathbf{s}_{\mathbf{E}[u][1]}, \mathbf{s}_{\mathbf{E}[u][2]}, \mathbf{s}_{\mathbf{E}[u][3]}, \mathbf{s}_{\mathbf{E}[u][4]}, \mathbf{s}_{\mathbf{E}[u][5]}, \mathbf{s}_{\mathbf{E}[u][6]}, \mathbf{s}_{\mathbf{E}[u][7]}, \mathbf{s}_{\mathbf{E}[u][8]} \tag{3}$$

C. HILBERT CODE OF CURVE STARTING GRID CELL

For a Hilbert curve of order m and size $T \times T \times T$, the Hilbert code is a series of integers that are continuously increasing and have an equal difference of 1. The value range is $(0, T \times T \times T - 1)$. The state vector in the previous section records the filling order of the Hilbert curve between the grid elements. Because the Hilbert code is the order of m-level grid elements in the curve, the Hilbert code of the starting grid element can be directly calculated according to the sub-grid element containing the sub-curve.

The element $\mathbf{S}[k-1][i]$ in matrix $\mathbf{S}$ indicates that the curve with state vector $\mathbf{s}_k$ (size $T \times T \times T$, $T = 2^m$) passes through $\mathbf{S}[k-1][i]$ sub-grid elements before filling the ith sub-grid elements, and there are $T^3/8$ m-level grid elements in each sub-grid element. If the starting sub-grid element of the curve is $HStar$, then because the Hilbert code is a series of consecutively increasing, equal-difference numbers, the starting grid cell element of the sub-curve (size $T/2 \times T/2 \times T/2$) in the ith sub-grid element conforms to (4):

$$HStar_i = HStar + \mathbf{S}[k-1][i] \times T^3/8 \tag{4}$$

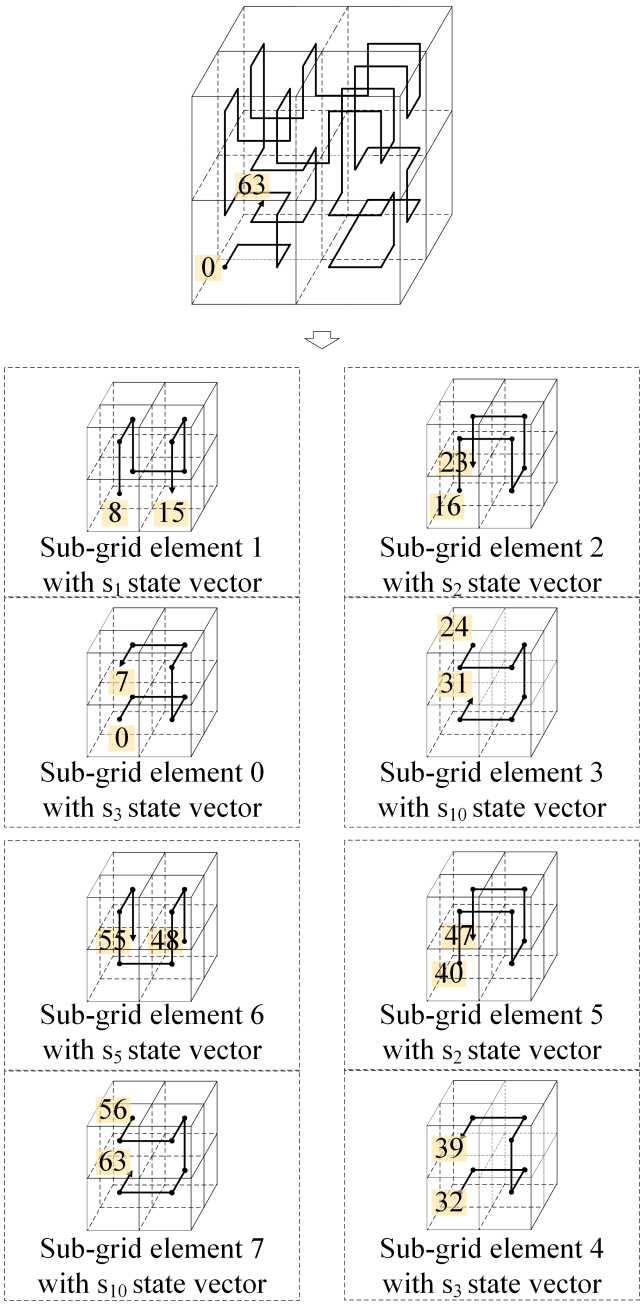

FIGURE 5. State vector mapping between grid element and its 8 sub-grid elements.

III. MONOTONICALLY INCREASING HILBERT CODE SEGMENT DIRECT GENERATION ALGORITHM

A. QUERY WINDOW SPATIAL RELATIONSHIP

The spatial relationship between the eight sub-grid elements obtained by octagonal division for the cubic region (side length T) of the order m Hilbert curve and query window W can be regarded as a query window parallelepiped that is parallel to the eight grid elements showing the topological relationship of a hexahedron in three-dimensional space. The reasoning for this conclusion and a qualitative description of the positional relationship in three-dimensional space is given in [26]. According to the reasoning given, the topological relationship between the query window parallelepiped and the grid elements parallelepiped can be divided into four categories as follows:

1) The query window intersects all eight sub-grid elements, as shown in Fig. 6 (a).
2) The query window intersects four of the eight sub-grid elements, as shown in Fig. 6 (b).
3) The query window intersects two of the eight sub-grid elements, as shown in Fig. 6 (c).
4) The query window intersects one of the eight sub-grid elements, as shown in Fig. 6 (d).

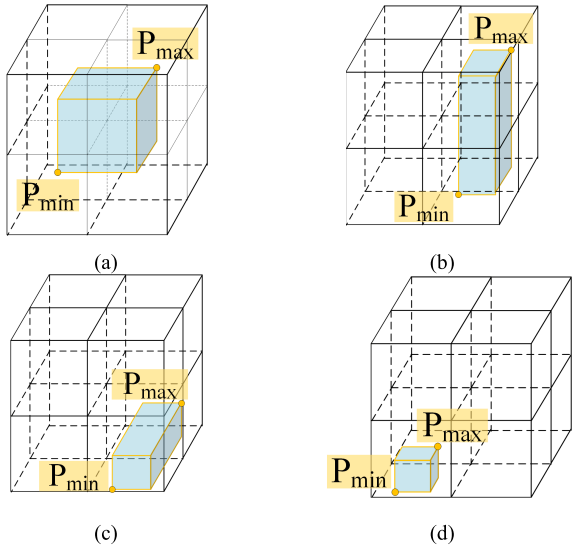

FIGURE 6. Query window spatial relationships. (a) Query window intersecting all eight sub-grid elements. (b) Query window intersecting four of the eight sub-grid elements. (c) Query window intersecting two of the eight sub-grid elements. (d) Query window intersecting one of the eight sub-grid elements.

Among the 27 total cases for the topological relationship in space, there is one case in space type 1 (Fig. 6 (a)), six cases in space type 2 (Fig. 6 (b)), twelve cases in space type 3 (Fig. 6 (c)), and eight cases in space type 4 (Fig. 6 (d)). Table 1 shows the quantitative relationship between the coordinates of the start corner P_{min}, the end corner P_{max} of the query window, and the side length T of the cube area for all the 27 spatial topological relations.

B. CODE SEGMENTATION GENERATION

In the MI-HCS algorithm, the Hilbert curve (with size $T \times T \times T$, $T = 2^m$) of order m, whose state vector is $\mathbf{s}_k$ and whose starting point code is $HStar$, is denoted as $H(\mathbf{s}_k, T, HStar)$, and the query process located on $H(\mathbf{s}_k, T, HStar)$ with query window $W(x, y, z, l, w, h)$ is denoted as $MIHCS(\mathbf{s}_k, T, HStar, x, y, z, l, w, h)$.

In this paper, recursive subdivision of query process $MIHCS(\mathbf{s}_k, T, HStar, x, y, z, l, w, h)$ is performed to generate a Hilbert code segment. The process is shown in Algorithm 1:

TABLE 1. Quantitative relationship of query window space.

Number	Intersecting sub-grid elements	P_{min}	P_{max}
0	(0,1,2,3, 4,5,6,7)	$x < 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
1	(0,3,4,7)	$x < 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h \leq 0.5T$
2	(1,2,5,6)	$x < 0.5T$ $y < 0.5T$ $z \geq 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
3	(0,1,6,7)	$x < 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l \leq 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
4	(2,3,4,5)	$x \geq 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
5	(0,1,2,3)	$x < 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w \leq 0.5T$ $z + h > 0.5T$
6	(4,5,6,7)	$x < 0.5T$ $y \geq 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
7	(0,3)	$x < 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w \leq 0.5T$ $z + h \leq 0.5T$
8	(1,2)	$x < 0.5T$ $y < 0.5T$ $z \geq 0.5T$	$x + l > 0.5T$ $y + w \leq 0.5T$ $z + h > 0.5T$
9	(5,6)	$x < 0.5T$ $y \geq 0.5T$ $z \geq 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
10	(4,7)	$x < 0.5T$ $y \geq 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h \leq 0.5T$
11	(0,1)	$x < 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l \leq 0.5T$ $y + w \leq 0.5T$ $z + h > 0.5T$
12	(6,7)	$x < 0.5T$ $y \geq 0.5T$ $z < 0.5T$	$x + l \leq 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
13	(2,3)	$x \geq 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w \leq 0.5T$ $z + h > 0.5T$
14	(4,5)	$x \geq 0.5T$ $y \geq 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
15	(0,7)	$x < 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l \leq 0.5T$ $y + w > 0.5T$ $z + h \leq 0.5T$
16	(3,4)	$x \geq 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h \leq 0.5T$
17	(1,6)	$x < 0.5T$ $y < 0.5T$ $z \geq 0.5T$	$x + l \leq 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
18	(2,5)	$x \geq 0.5T$ $Y < 0.5T$ $z \geq 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
19	(0)	$x < 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l \leq 0.5T$ $y + w \leq 0.5T$ $z + h \leq 0.5T$
20	(1)	$x < 0.5T$ $y < 0.5T$ $z \geq 0.5T$	$x + l \leq 0.5T$ $y + w \leq 0.5T$ $z + h \geq 0.5T$
21	(2)	$x \geq 0.5T$ $y < 0.5T$ $z \geq 0.5T$	$x + l > 0.5T$ $y + w \leq 0.5T$ $z + h > 0.5T$
22	(3)	$x \geq 0.5T$ $y < 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w \leq 0.5T$ $z + h \leq 0.5T$
23	(4)	$x \geq 0.5T$ $y \geq 0.5T$ $z < 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h \leq 0.5T$
24	(5)	$x \geq 0.5T$ $y \geq 0.5T$ $z \geq 0.5T$	$x + l > 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
25	(6)	$x < 0.5T$ $y \geq 0.5T$ $z \geq 0.5T$	$x + l \leq 0.5T$ $y + w > 0.5T$ $z + h > 0.5T$
26	(7)	$x < 0.5T$ $y \geq 0.5T$ $z < 0.5T$	$x + l \leq 0.5T$ $y + w > 0.5T$ $z + h \leq 0.5T$

In Algorithm 1, step 1 is to determine whether the current query window size is equal to the Hilbert curve size. If the size of the query window is equal to the size of the Hilbert curve, then the entire Hilbert curve fills the current query window exactly, and code segments ($HStar$, $HStar+T^3-1$) are added to $HRange$. If the size of the query window is not equal to the size of the Hilbert curve, then the algorithm proceeds

Algorithm 1 *MIHCS* ($\mathbf{s}_k, T, HStar, x, y, z, l, w, h$)

```
if (l = w = h = T) then
    HRange.push(HStar, HStar + T^3 − 1);
else
    switch s_k do
        case
            s_k (k ∈ (1, 2, ··· , 24)) do
        s_k : MIHCS − s_k (T, HStar, x, y, z, l, w, h) ;
    end switch;
end if;
Merge(HRange);
```

to steps 2–8 and recursively octets the query process. The recursive method is as follows:

First, the Hilbert curve is divided into octets to obtain eight sub-curves. According to the number of sub-grid elements containing the sub-curves, each sub-curve is recorded as $NextH_i$ ($i \in (0, 1, \ldots, 7)$). The content combined with II.B and II.C shows the following:

$$NextH_i = H\left(\mathbf{E}[k-1][i], T/2, HStar + \mathbf{S}[k-1][i] \times T^3/8\right) \tag{5}$$

As shown in Fig. 7, each sub-curve is $NextH_0 = H(\mathbf{s}_3, 2, 0)$, $NextH_1 = H(\mathbf{s}_1, 2, 8)$, $NextH_2 = H(\mathbf{s}_2, 2, 16)$, $NextH_3 = H(\mathbf{s}_2, 2, 24)$, $NextH_4 = H(\mathbf{s}_2, 2, 32)$, $NextH_5 = H(\mathbf{s}_2, 2, 40)$, $NextH_6 = H(\mathbf{s}_2, 2, 48)$, and $NextH_7 = H(\mathbf{s}_2, 2, 56)$.

Second, the query window W is divided into octets to obtain eight sub-query windows. Each sub-query window is denoted as $NextW_i (i \in (0, 1, \ldots, 7))$ according to the number of sub-grid elements containing the sub-query window. There are 27 different cases of spatial topological relationships between the query window and the eight sub-grid elements. If the query window does not intersect the ith sub-grid element, there is no sub-query window $NextH_i$. The sub-query window for each spatial relationship is shown in Table 2. Taking Fig. 6 as an example, the query window $W(0, 0, 0, 3, 4, 2)$ intersects the sub-grid elements (0,3,4,7) in the eight sub-grid elements, and the corresponding spatial relationship is numbered 1. The resulting sub-query windows obtained after subdividing the query window $W(0, 0, 0, 3, 4, 2)$ are as follows: $NextW_0 = W(0, 0, 0, 2, 2, 2)$, $NextW_3 = W(0, 0, 0, 1, 2, 2)$, $NextW_4 = W(0, 0, 0, 1, 2, 2)$, and $NextW_7 = W(0, 0, 0, 2, 2, 2)$.

Finally, the sub-curves and the sub-query windows are combined in pairs to form a new query process. Taking Fig. 7 as an example, after subdivision, only sub-curves $NextH_0$, $NextH_3$, $NextH_4$, and $NextH_7$ exist on the sub-query window. Therefore, each new query process is as follows:

MIHCS($\mathbf{s}_3$, 2, 0, 0, 0, 0, 2, 2, 2), *MIHCS*($\mathbf{s}_{10}$, 2, 24, 0, 0, 0, 1, 2, 2), *MIHCS*($\mathbf{s}_3$, 2, 32, 0, 0, 0, 1, 2, 2), and *MIHCS*($\mathbf{s}_{10}$, 2, 56, 0, 0, 0, 2, 2, 2). The new query process is recursively substituted into Algorithm 1 to solve. After obtaining the several octets mentioned above, the required Hilbert code segment set *HRange* can finally be obtained.

TABLE 2. Sub-query windows for different spatial relationships.

Number	Sub-query windows
0	$NextW_0 = W(x, y, z, 0.5T - x, 0.5T - y, 0.5T - z)$; $NextW_1 = W(x, y, 0, 0.5T - x, 0.5T - y, h + z - 0.5T)$; $NextW_2 = W(0, y, 0, x + l - 0.5T, 0.5T - y, h + z - 0.5T)$; $NextW_3 = W(0, y, z, x + l - 0.5T, 0.5T - y, 0.5T - z)$; $NextW_4 = W(0, 0, z, x + l - 0.5T, y + w - 0.5T, 0.5T - z)$; $NextW_5 = W(0, 0, 0, x + l - 0.5T, y + w - 0.5T, h + z - 0.5T)$; $NextW_6 = W(x, 0, 0, 0.5T - x, y + w - 0.5T, h + z - 0.5T)$; $NextW_7 = W(x, 0, z, 0.5T - x, y + w - 0.5T, 0.5T - z)$;
1	$NextW_0 = W(x, y, z, 0.5T - x, 0.5T - y, h)$; $NextW_3 = W(0, y, z, x + l - 0.5T, 0.5T - y, h)$; $NextW_4 = W(0, 0, z, x + l - 0.5T, y + w - 0.5T, h)$; $NextW_7 = W(x, 0, z, 0.5T - x, y + w - 0.5T, h)$;
2	$NextW_1 = W(x, y, z - 0.5T, 0.5T - x, 0.5T - y, h)$; $NextW_2 = W(0, y, z - 0.5T, x + l - 0.5T, 0.5T - y, h)$; $NextW_5 = W(0, 0, z - 0.5T, x + l - 0.5T, y + w - 0.5T, h)$; $NextW_6 = W(x, 0, z - 0.5T, 0.5T - x, y + w - 0.5T, h)$;
3	$NextW_0 = W(x, y, z, l, 0.5T - y, 0.5T - z)$; $NextW_1 = W(x, y, 0, l, 0.5T - y, h + z - 0.5T)$; $NextW_6 = W(x, 0, 0, l, y + w - 0.5T, h + z - 0.5T)$; $NextW_7 = W(x, 0, z, l, y + w - 0.5T, 0.5T - z)$;
4	$NextW_2 = W(x - 0.5T, y, 0, l, 0.5T - y, h + z - 0.5T)$; $NextW_3 = W(x - 0.5T, y, z, l, 0.5T - y, 0.5T - z)$; $NextW_4 = W(x - 0.5T, 0, z, l, y + w - 0.5T, 0.5T - z)$; $NextW_5 = W(x - 0.5T, 0, 0, l, y + w - 0.5T, h + z - 0.5T)$;

TABLE 2. *(Continued.)* **Sub-query windows for different spatial relationships.**

5	$NextW_0 = W(x, y, z, 0.5T - x, w, 0.5T - z);$ $NextW_1 = W(x, y, 0, 0.5T - x, w, h + z - 0.5T);$ $NextW_2 = W(0, y, 0, x + l - 0.5T, w, h + z - 0.5T);$ $NextW_3 = W(0, y, z, x + l - 0.5T, w, 0.5T - z);$
6	$NextW_4 = W(0, y - 0.5T, z, x + l - 0.5T, w, 0.5T - z);$ $NextW_5 = W(0, y - 0.5T, 0, x + l - 0.5T, w, h + z - 0.5T);$ $NextW_6 = W(x, y - 0.5T, 0, 0.5T - x, w, h + z - 0.5T);$ $NextW_7 = W(x, y - 0.5T, z, 0.5T - x, w, 0.5T - z);$
7	$NextW_0 = W(x, y, z, 0.5T - x, w, h);$ $NextW_3 = W(0, y, z, x + l - 0.5T, w, h);$
8	$NextW_1 = W(x, y, z - 0.5T, 0.5T - x, w, h);$ $NextW_2 = W(0, y, z - 0.5T, x + l - 0.5T, w, h);$
9	$NextW_5 = W(x, y - 0.5T, z - 0.5T, 0.5T - x, w, h);$ $NextW_6 = W(0, y - 0.5T, z - 0.5T, x + l - 0.5T, w, h);$
10	$NextW_4 = W(x, y - 0.5T, z, 0.5T - x, w, h);$ $NextW_7 = W(0, y - 0.5T, z, x + l - 0.5T, w, h);$
11	$NextW_0 = W(x, y, z, l, w, 0.5T - z);$ $NextW_1 = W(x, y, 0, l, w, h + z - 0.5T);$
12	$NextW_6 = W(x, y - 0.5T, z, l, w, 0.5T - z);$ $NextW_7 = W(x, y - 0.5T, 0, l, w, h + z - 0.5T);$
13	$NextW_2 = W(x - 0.5T, y, z, l, w, 0.5T - z);$ $NextW_3 = W(x - 0.5T, y, 0, l, w, h + z - 0.5T);$
14	$NextW_4 = W(x - 0.5T, y - 0.5T, z, l, w, 0.5T - z);$ $NextW_5 = W(x - 0.5T, y - 0.5T, 0, l, w, h + z - 0.5T);$
15	$NextW_0 = W(x, y, z, l, 0.5T - y, h);$ $NextW_7 = W(x, 0, z, l, y + w - 0.5T, h);$
16	$NextW_3 = W(x - 0.5T, y, z, l, 0.5T - y, h);$ $NextW_4 = W(x - 0.5T, 0, z, l, y + w - 0.5T, h);$
17	$NextW_1 = W(x, y, z - 0.5T, l, 0.5T - y, h);$ $NextW_6 = W(x, 0, z - 0.5T, l, y + w - 0.5T, h);$
18	$NextW_2 = W(x - 0.5T, y, z - 0.5T, l, 0.5T - y, h);$ $NextW_5 = W(x - 0.5T, 0, z - 0.5T, l, y + w - 0.5T, h);$

Unlike the traversal algorithm in [16], the proposed algorithm does not need to calculate the Hilbert code of all grid elements. When the recursive termination condition is reached, the corresponding code segment of the curve is output, and the number of calculations is reduced, which improves the efficiency.

TABLE 2. *(Continued.)* **Sub-query windows for different spatial relationships.**

19	$NextW_0 = W(x, y, z, l, w, h);$
20	$NextW_1 = W(x, y, z - 0.5T, l, w, h);$
21	$NextW_2 = W(x - 0.5T, y, z - 0.5T, l, w, h);$
22	$NextW_3 = W(x - 0.5T, y, z, l, w, h);$
23	$NextW_4 = W(x - 0.5T, y - 0.5T, z, l, w, h);$
24	$NextW_5 = W(x - 0.5T, y - 0.5T, z - 0.5T, l, w, h);$
25	$NextW_6 = W(x, y - 0.5T, z - 0.5T, l, w, h);$
26	$NextW_7 = W(x, y - 0.5T, z, l, w, h);$

C. ALGORITHM RECURSIVE ORDER

The recursive splitting of the query process in the MI-HCS algorithm is similar to the multidimensional tree search process in [23], but because [23] does not specify the search order when searching for tree nodes, the resulting final set of Hilbert code segments may not be monotonically increasing, requiring all code segments to be sorted before the continuous code segment merging step is performed.

To make the code segment set *HRange* conform to the requirement of a monotonically increasing nature, this paper specifies the recursive order of the new query process in Steps 3–5 of Algorithm 1. The smaller the starting point code of the Hilbert curve in the query process, the earlier the query process is recursively called. It can be known from II.B and II.C that the starting point code of the sub-curve can be calculated by (4), which is related to the state vector of the curve, and filling the sub-grid element earlier results in a smaller sub-curve starting code.

Taking the first octet split in Fig. 7 as an example, four new query processes are located in the sub-grid elements (0,3,4,7). Because the state vector of the 2-level curve in Fig. 7 is $\mathbf{s}_2$, the curve first fills sub-grid element 0, followed sequentially by sub-grid elements 3, 4, and 7. In other words, sub-curve $NextH_0$ has the smallest starting code, followed sequentially by the starting code of sub-curve $NextH_3$, $NextH_4$, and $NextH_7$, which is the largest.

In accordance with the principle that smaller Hilbert starting codes should begin earlier in the recursive call of the query processes, the first query process $MIHCS(\mathbf{s}_3, 2, 0, 0, 0, 0, 2, 2, 2)$ in sub-grid element 0, followed by $MIHCS(\mathbf{s}_{10}, 2, 24, 0, 0, 0, 1, 2, 2)$ in sub-grid element 3, then $MIHCS(\mathbf{s}_3, 2, 32, 0, 0, 0, 1, 2, 2)$ in sub-grid element 4, and finally $MIHCS(\mathbf{s}_{10}, 2, 56, 0, 0, 0, 2, 2, 2)$ in sub-grid element 7.

In summary, different state vectors specify different recursive calling sequences of the query processes. Steps 3–5 in Algorithm 1 are extended as per Algorithm 2:

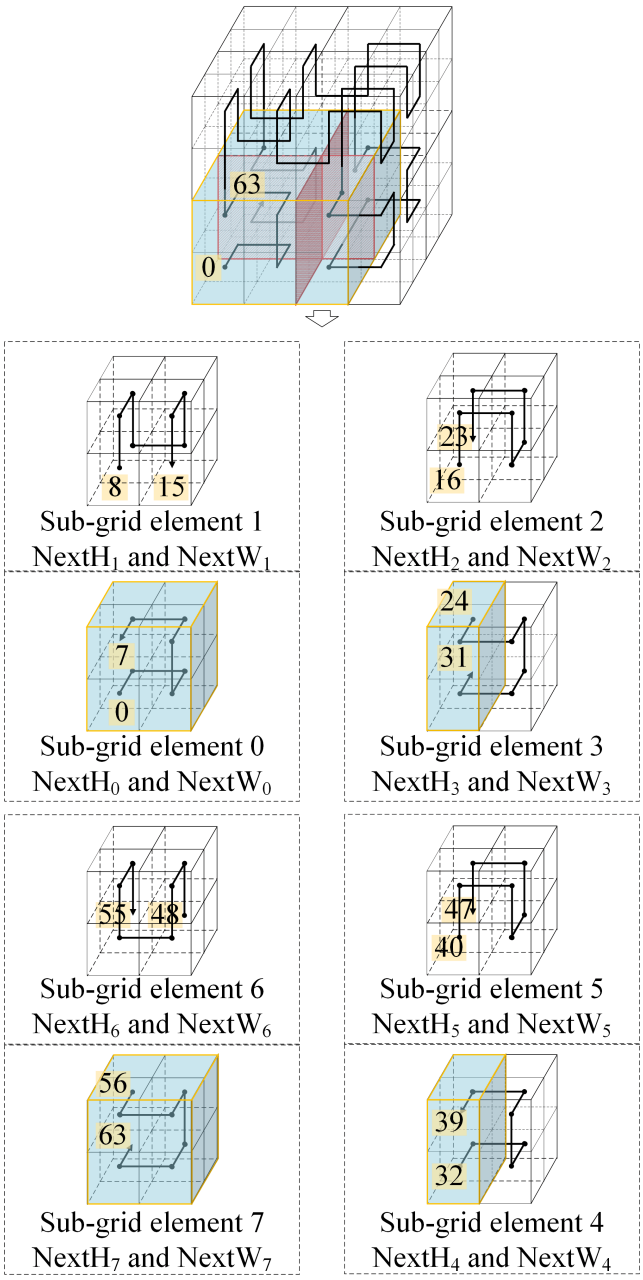

FIGURE 7. The window and the curve are divided to obtain sub-windows and sub-curves.

In Algorithm 2, Step 3 is to calculate the spatial topology of the current query window. The array $a[\ ]$ stores the number of sub-grid elements that intersect with the query window. Steps 4–8 start to call each sub-query process in the prescribed order. Step 5 is to obtain the *order*th sub-grid element number i of the Hilbert curve filled with the state vector $\mathbf{s}_k$. Step 6 determines whether the sub-grid element i intersect with the query window; that is, whether i exists in the array $a[\ ]$. Step 7 is a recursive call of the query process corresponding to the ith sub-grid elements.

The above description describes the process of generating the Hilbert code segment set using the algorithm in this article. Taking Fig. 7 as an example, the steps of recursively generating the Hilbert code segment set are as follows:

Algorithm 2 $MIHCS - \mathbf{s}_k\ (T, HStar, x, y, z, l, w, h)$

```
1 T1 ← T/2;
2 T2 ← T/2 × T/2 × T/2;
3 a[] ← GetSpatialRelationship();
4 for (order = 0; order < 8; order + +)
5     i ← getsequence (order) ;
6     if (i in a[]) then
7         MIHCS(E[k − 1][i], T1, HStar + order × T2,
          NextWi.x, NextWi.y, NextW.z, NextW.l,
          NextW.w, NextW.h);
8     end if;
9 end for;
```

First, $MIHCS$ ($\mathbf{s}_1$, 4, 0, 0, 0, 0, 3, 4, 2) is substituted into Algorithm 1, and a value comparison of $3 \neq 4 \neq 2 \neq 4$ is made. Following this outcome, the first octet split is performed, resulting in four new query processes comprising $MIHCS(\mathbf{s}_3, 2, 0, 0, 0, 0, 2, 2, 2)$, $MIHCS(\mathbf{s}_{10}, 2, 24, 0, 0, 0, 1, 2, 2)$, $MIHCS(\mathbf{s}_3, 2, 32, 0, 0, 0, 1, 2, 2)$, and $MIHCS(\mathbf{s}_{10}, 2, 56, 0, 0, 0, 2, 2, 2)$.

Next, $MIHCS(\mathbf{s}_3, 2, 0, 0, 0, 0, 2, 2, 2)$ is called recursively, and the value comparison result is $2 = 2 = 2 = 2$. Following this outcome, (0,7) is added to *HRange*, and is returned recursively. $MIHCS(\mathbf{s}_{10}, 2, 24, 0, 0, 0, 1, 2, 2)$ is then called recursively and determined as $1 \neq 2 = 2 = 2$, resulting in the second octet split, thereby generating four new query processes. According to the S_{10} state vectors, the corresponding order of the four new query processes is $MIHCS(\mathbf{s}_{14}, 1, 24, 0, 0, 0, 1, 1, 1)$, $MIHCS(\mathbf{s}_{13}, 1, 25, 0, 0, 0, 1, 1, 1)$, $MIHCS(\mathbf{s}_9, 1, 30, 0, 0, 0, 1, 1, 1)$, and $MIHCS(\mathbf{s}_2, 1, 31, 0, 0, 0, 1, 1, 1)$. These four new queries are substituted into Algorithm 1 again to get (24,24), (25,25), (30,30), (31,31), and return recursively. Algorithm 1 is continuously called recursively until there are no unprocessed queries.

The final complete code segment set generation process is shown in Fig. 8, and each obtained code segment corresponds to a leaf node in Fig. 8. Algorithm 1 finally merges consecutive code segments in all leaf nodes to obtain the final code segment set *HRange*.

D. ANALYSIS OF COMPLEXITY

The recursive termination condition of the proposed algorithm is the size of the query window being equal to the curve size. If the recursive termination condition is not met, the proposed algorithm divides the current query window into eight sub-query windows. The implementation method of recursive splitting of the octree is the same as the recursive splitting method of the proposed algorithm.

Lemma 1: In the recursive process of the MI-HCS algorithm, if the size of a query window is the same as the size of the corresponding curve, the query window is equivalent to a child node in the octree.

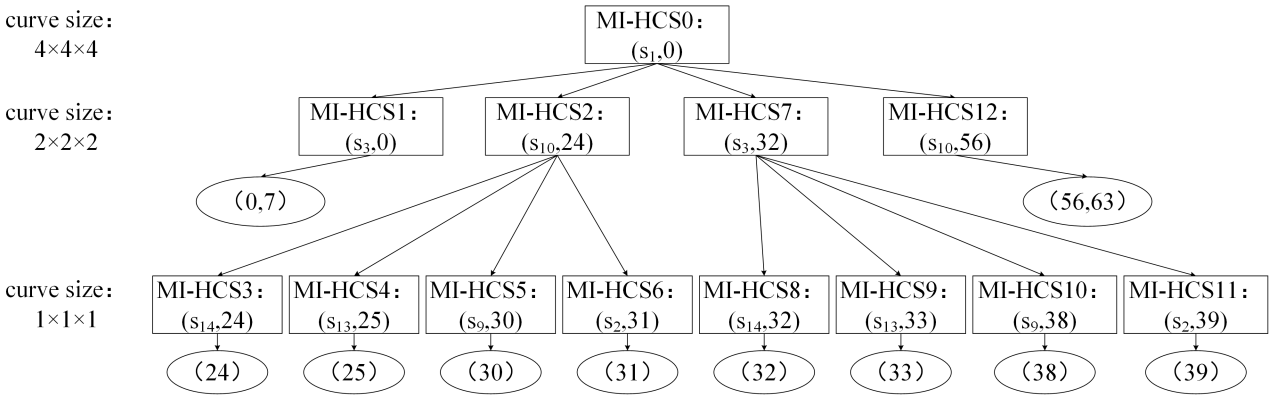

FIGURE 8. Code segment generation process.

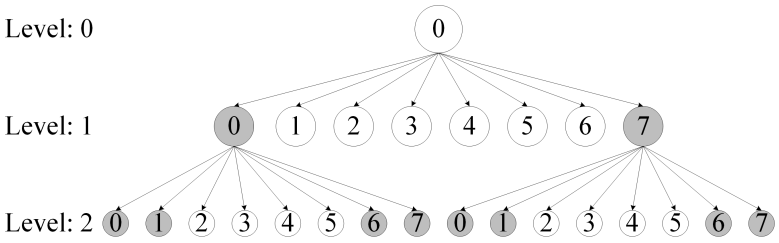

FIGURE 9. The query process corresponds to the octree structure.

Fig. 9 shows the octree structure of the query process illustrated in Fig. 7. The gray-filled child nodes in Fig. 9 correspond to the leaf nodes in Fig. 8. For example, the No.0 child node of level 1 in Fig. 9 corresponds to the (0, 7) code segment in Fig. 8. According to Lemma 1, the query window that meets the recursive termination condition is equivalent to the child nodes in the octree. Therefore, for a query window W, the time required for the algorithm to generate its corresponding code segment is related to the number of octree child node in the window. For example, the generation time required for the query process in Fig. 7 is the sum of the time required for the algorithm to recursively divide the query window into the 10 gray-filled child nodes in Fig. 9.

For a Hilbert curve of size $T \times T \times T$, if the size of the query window is $T \times T \times T$, no octet split is needed, and the proposed algorithm is only called once. If the size of the query window is $T/2 \times T/2 \times T/2$, one octet split is performed, and the algorithm is recursively called twice; if the query window size is $T/2^f \times T/2^f \times T/2^f$, octet splitting is performed f times and the algorithm is recursively called $f + 1$ times. Therefore, for a recursive termination condition, the size is $2^e \times 2^e \times 2^e \left(= T/2^{(\log_2 T - e)} \times T/2^{(\log_2 T - e)} \times T/2^{(\log_2 T - e)}\right)$, and the proposed algorithm needs to be called $log_2 T - e + 1$ times recursively.

Lemma 2: For the Hilbert curve of size $T \times T \times T$, the MI-HCS algorithm recursively splits into a query window with a size of $2^e \times 2^e \times 2^e$. A total of $log_2 T - e + 1$ recursive calls are performed.

Given a query window $W(x, y, z, l, w, h)$, [27] analyzes the average number of octree children in window W, which states that if l, w, h are odd numbers, and $\min(l, w, h) = 2^k - 1$, $\text{median}(l, w, h) = 2^{k+d_2} - 1$, and $\max(l, w, h) = 2^{k+d_3} - 1$, the average number $\overline{b}$ of octree children in window W conforms to (6) as follows:

$$\begin{aligned}\overline{b}(l, w, h) &= \overline{b}(2^k - 1, 2^{k+d_2} - 1, 2^{k+d_3} - 1) \\ &= \frac{4}{3}\left(2^{2k} - 1\right)\left(2^{d_2} + 2^{d_3} + 2^{d_2+d_3}\right) \\ &\quad - 6\left(2^k - 1\right)\left(1 + 2^{d_2} + 2^{d_3}\right) + 7k \end{aligned} \tag{6}$$

Additionally, the relationship between $\overline{b}(l + 1, w, h)$ and $\overline{b}(l, w, h)$ conforms to (7) as follows:

$$\begin{aligned}&\overline{b}(l + 1, w, h) \\ &= \overline{b}(l, w, h) + 2^{2k+d_2+d_3-3\lambda} + 8 - \frac{7}{3}\left(2^{\lambda+1} + 2^{-\lambda}\right) \\ &\quad - 2^k\left(2^{d_2} + 2^{d_3}\right)\left(\frac{7}{9} \times 2^{-2\lambda} - \frac{7}{6} \times \lambda\right) + \frac{2}{9}\end{aligned} \tag{7}$$

where $\lambda = \lfloor \log(\min(l + 1, w, h)) \rfloor$. Analysis of (6) and (7) shows that the highest order term in the calculation formula of $\overline{b}$ is $2^{2k+d_2+d_3}$; that is, its complexity is:

$$\begin{aligned} O\left(2^{2k+d_2+d_3}\right) &= O\left(2^{k+d_2} \times 2^{k+d_3}\right) \\ &= O\left(median(l, w, h) \times \max(l, w, h)\right) \end{aligned} \tag{8}$$

Lemma 3: Given a query window $W(x, y, z, l, w, h)$, the complexity of the average number of existing octree child

nodes $\bar{b}$ is $O(\alpha_1 \times \alpha_2)$, where $\alpha_1 = \text{median}(l, w, h)$ and $\alpha_2 = \max(l, w, h)$.

Combining Lemmas 1–3, the maximum complexity of the algorithm can be analyzed. The query window that needs to recursively call the algorithm of this paper the most times is one with a size of $1 \times 1 \times 1$, requiring $log_2 T + 1$ calls. Therefore, the longest calculation time required by the proposed algorithm occurs when all the child nodes in the query window are the smallest child nodes of $1 \times 1 \times 1$, requiring a total calculation time of $O\left(\alpha_1 \times \alpha_2 \times \left(\log_2 T + 1\right)\right)$. Through analysis, the following conclusion is reached:

Conclusion 1: Given a query window $W(x, y, z, l, w, h)$ and a Hilbert curve of size $T \times T \times T$, the complexity of the proposed algorithm is $O\left(\alpha_1 \times \alpha_2 \times \left(\log_2 T + 1\right)\right)$, where $\alpha_1 = \text{median}(l, w, h)$ and $\alpha_2 = \max(l, w, h)$.

IV. EXPERIMENT AND ANALYSIS

This section experimentally verifies the complexity of the proposed algorithm, and then compares the proposed algorithm with the algorithms in [16] and [23] to analyze the computational efficiency of the algorithm in generating Hilbert code segments in the query window. Here we give the experimental design principles. Two complexity verification experiments were performed: fixed window size and fixed curve size. When comparing and analyzing the efficiency of the different algorithms, we first used query windows with different sizes and volumes with a fixed curve size. Then, we used a fixed query window size, and set different curve sizes to perform a comparative analysis of the algorithms. The experimental hardware environment used in this study includes an Intel Core i7-7700K CPU (dual-core 4.2 GHz) and 64 GB RAM. The software environment was Visual Studio 2015, Release version, x64, C++.

A. ALGORITHM COMPLEXITY VERIFICATION

First, experiments were performed to verify the complexity of the MI-HCS algorithm with the size of the curve. The size of the query window W was set to $1 \times 1 \times 1$ or $5 \times 5 \times 5$, where the size $1 \times 1 \times 1$ corresponds to the case where the algorithm needs the most recursion. During the experiment, the size of the Hilbert curve was set to $4^t \times 4^t \times 4^t$ ($t \in (2, 3, \ldots, 14)$). The starting points of the query windows were randomly selected within the scope of the Hilbert curve, and the proposed algorithm was used to generate the Hilbert code segments corresponding to the query window. Each size curve was calculated 100,000 times, and the total generation time required by the algorithm in this size was recorded. The results are given in Table 3.

Fig. 10 is a visual representation of the data in Table 3. The solid line in Fig. 10 is the generation time, and the red underline is the fitting curve. Based on the analysis of the data in Table 3 and Fig. 10, it can be seen that when the size of the query window W is $1 \times 1 \times 1$, the fitting curve in Figure 10 (a) has a linear growth trend, indicating that the generation time of the algorithm in this case grows linearly with $\log_2 T + 1$. It is proved that when the query window size is fixed at $1 \times 1 \times 1$, the maximum complexity of the proposed algorithm is $O\left(\log_2 T + 1\right)$, which is consistent with Conclusion 1. When the size of the query window is fixed at $5 \times 5 \times 5$, the fitted curve in Fig. 10 (b) does not fully comply with the linear growth trend, but the overall trend is upward. The reason for this is that the starting corners of the query window were randomly selected. The octree child node in the query window is not necessarily always $1 \times 1 \times 1$ in size, so the required calculation time of each child node may differ, resulting in generation time that does not increase linearly with $\log_2 T + 1$.

Second, experiments were performed to verify the change in algorithm efficiency with window size. To verify the relationship between the complexity of the algorithm and the size of the window, the size of the Hilbert curve was set to $1024 \times 1024 \times 1024$. The maximum value of l, w, h was set to $\alpha_2 = \max(l, w, h) = 30 + (r - 1) \times 30$, and the median value of l, w, h was set to $\alpha_1 = \text{median}(l, w, h) = 30 + (r - 1) \times 20$, where $r \in (1, 2, \ldots 30)$. The starting

TABLE 3. Statistics of generation time with curve size.

t	T	$\log_2 T+1$	Generation time (s)	
			1×1×1	5×5×5
2	4^2	5	0.223	1.355
3	4^3	7	0.268	1.556
4	4^4	9	0.36	1.682
5	4^5	11	0.432	1.814
6	4^6	13	0.54	2.01
7	4^7	15	0.603	1.992
8	4^8	17	0.739	2.184
9	4^9	19	0.808	3.29
10	4^{10}	21	0.898	2.546
11	4^{11}	23	0.967	2.474
12	4^{12}	25	1.048	2.573
13	4^{13}	27	1.161	2.627
14	4^{14}	29	1.263	2.738

TABLE 4. Statistics of corresponding generation time of corresponding code segments of different sized windows.

r	α_1	α_2	$\alpha_1 \times \alpha_2$	Generation time (s)
1	30	30	900	0.71
2	50	60	3000	1.95
3	70	90	6300	2.48
4	90	120	10800	4.22
15	310	450	139500	46.25
16	330	480	158400	52.49
17	350	510	178500	65.9
28	570	840	478800	162.64
29	590	870	513300	169.82
30	610	900	549000	178.22

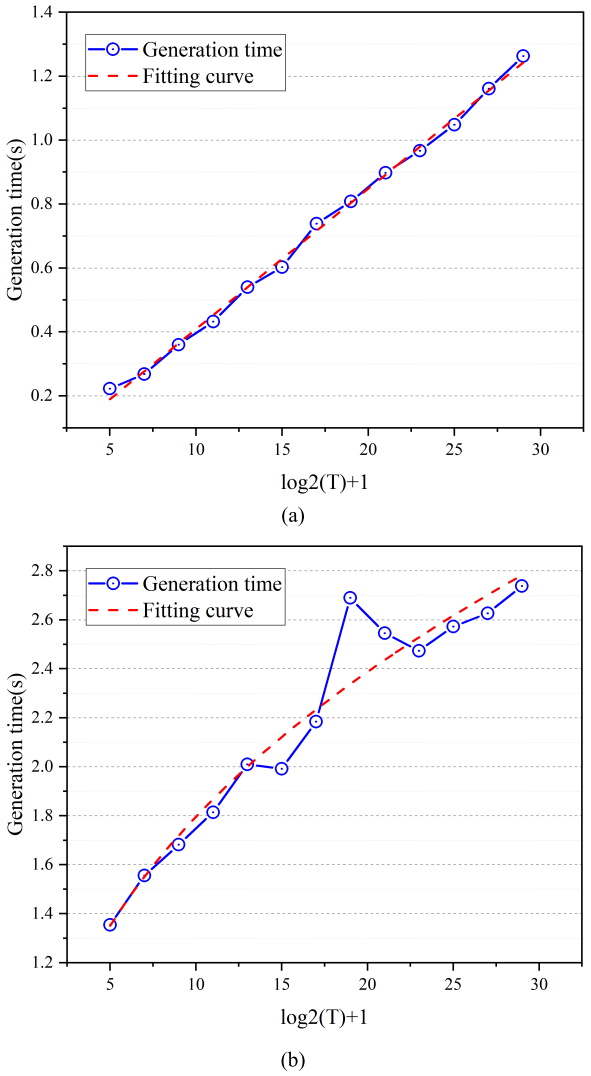

FIGURE 10. Comparison of generation time with different curve size. (a) Time required to generate a code segment corresponding to a 1 × 1 × 1 query window. (b) Time required to generate a code segment corresponding to a 5 × 5 × 5 query window.

points of the query windows were randomly selected within the Hilbert curve, and the proposed algorithm was used to generate Hilbert code segments corresponding to the query window. Each window size was calculated 1000 times, and the total generation time required by the algorithm for each window size was recorded. The generation time required for some sizes are shown in Table 4.

Fig. 11 is a visual representation of the data in Table 4. In Fig. 11, the solid line is the generation time, and the red underline is the fitting curve. Combining Table 4 with the data analysis in Fig. 11 shows that the fitting curve in Fig. 11 shows a clear linear growth trend, indicating that when the size of the Hilbert curve is fixed, the generation time of the proposed algorithm increases linearly with the increase of $\alpha_1 \times \alpha_2$. It is proved that the complexity of the proposed algorithm is $O(\alpha_1 \times \alpha_2)$ when the size of the Hilbert curve is fixed, which is consistent with Conclusion 1.

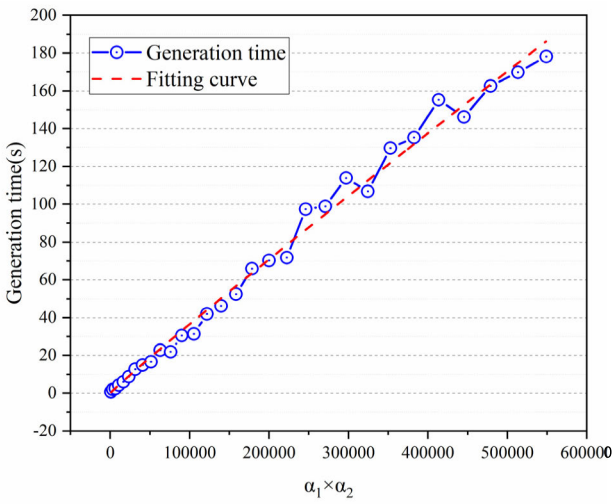

FIGURE 11. Generation time as a function of size.

B. ALGORITHM EFFICIENCY COMPARISON

First, query windows with different sizes were set for the comparison of efficiency with a Hilbert curve of size $1024 \times 1024 \times 1024$. The size of the query window was set to $l = w = h = r$, and the range of r was $r \in (30, 50, \cdots, 210)$. The starting corners of the query windows were randomly selected within the range of the Hilbert curve, and the Hilbert code segments corresponding to the query window were generated using the algorithm of this paper along with those in [16] and [23]. Each window size was calculated 1000 times and recorded. The total generation time required by the three algorithms for each window size is shown in Table 5. Fig. 12 is a comparison of the three algorithms.

TABLE 5. Statistics of query window generation time.

Window size	Generation time (s)			Efficiency
	Proposed	[23]	[16]	increase (%)
30	0.507	1.752	36.89	345.6%
50	1.358	4.568	98.75	336.4%
70	2.516	9.933	194.04	394.8%
90	4.083	16.349	329.31	400.4%
110	6.307	24.613	478.83	390.2%
130	8.782	36.04	673.16	410.4%
150	11.937	49.624	906.43	415.7%
170	15.143	63.059	1161.7	416.4%
190	19.002	79.607	1415.35	418.9%
210	23.098	97.915	1760.06	423.9%

Subsequently, query windows with different volumes were set for efficiency comparison with a Hilbert curve of size $1024 \times 1024 \times 1024$. During the experiment, the size of the

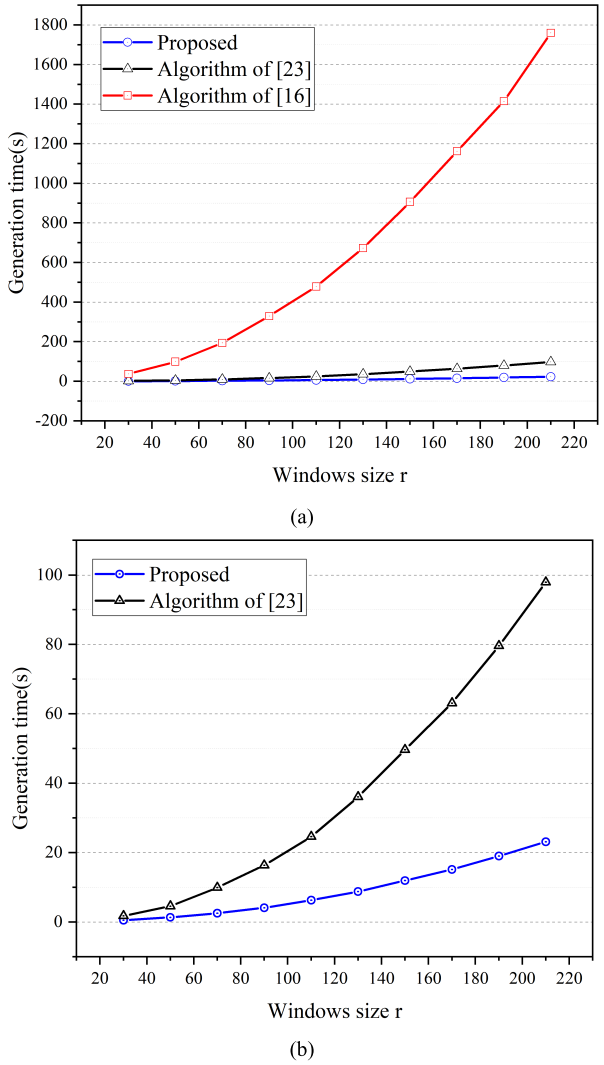

FIGURE 12. Comparison of generation time of query windows of different sizes. (a) Comparison of the generation time of the three algorithms. (b) Comparison of the generation time for the proposed algorithm and that in [23].

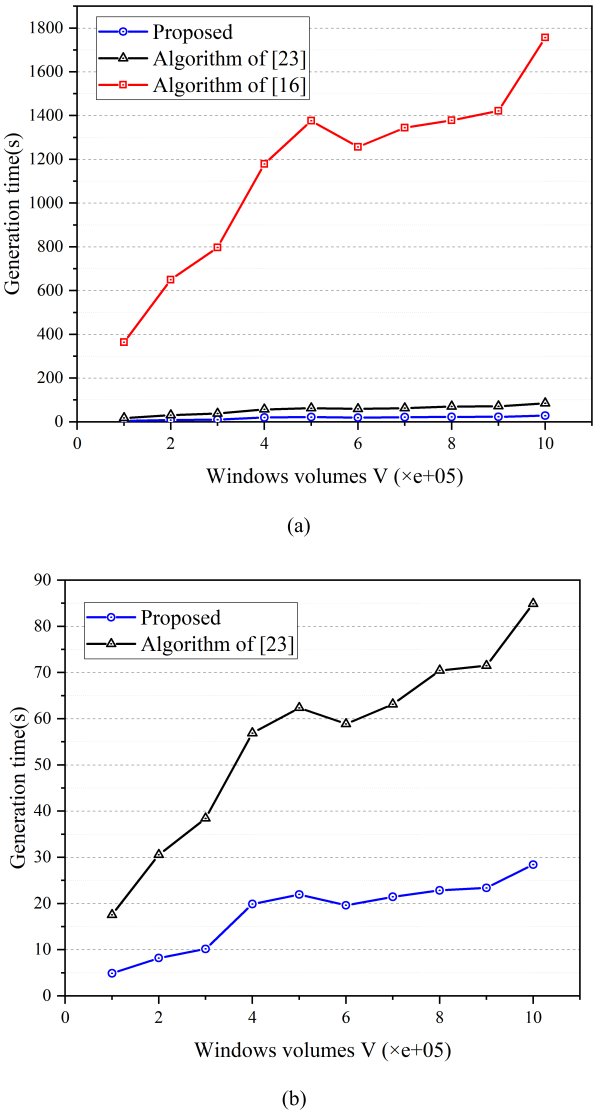

FIGURE 13. Comparison of the generation time of different volume query windows. (a) Comparison of the generation time of the three algorithms. (b) Comparison of the generation time of the proposed algorithm and that in [23].

query window was set as $l \times w \times h = V$, and the value range of V was $V \in (1 \times 10^5, 2 \times 10^5, \cdots, 10 \times 10^5)$. The starting points of the query windows were randomly selected within the range of the Hilbert curve, and the proposed algorithm was used along with those in [16] and [23] to generate Hilbert code segments corresponding to the query window. Each volume window was calculated 1,000 times and recorded. The total generation time required by the three algorithms for each window volume is shown in Table 6. Fig. 13 is a comparison of the three algorithms.

It can be seen from Figs. 12 and 13 as well as Tables 5 and 6 that, as the query window size and volume increase, the time required for the three algorithms to generate Hilbert code segments will gradually increase. Comparing query windows of the same size or volume, both the algorithm in [23] and the proposed algorithm take less time to complete than the algorithm in [16]. This is because the algorithm in [16] needs to calculate the Hilbert code by traversing several curves, which is inefficient. Compared with the time required for the algorithm in paper [23], the proposed algorithm takes less time to calculate the same size or volume of query window, and improves the efficiency by 284.1% to 423.9%. This is because the proposed algorithm specifies the recursive calling order of the algorithm and directly outputs a set of code segments that conform to the monotonically increasing nature. However, the output code segment of the algorithm in [23] is random and unordered, and therefore requires an additional step for sorting. When the query window is large, the sorting step increases time consumption, which reduces the efficiency of the algorithm in [23], making the advantages of this algorithm more apparent.

Finally, when the size of query windows is $8 \times 8 \times 8$, Hilbert curves of different sizes were set for efficiency comparison.

TABLE 6. Time statistics of query windows of different volumes.

Window volume V	Generation time (s)			Efficiency increase (%)
	Proposed	[23]	[16]	
1	4.878	17.558	364.085	359.9%
2	8.199	30.566	649.824	372.8%
3	10.171	38.429	797.504	377.8%
4	19.882	56.897	1178.92	286.2%
5	21.954	62.38	1376.48	284.1%
6	19.617	58.874	1257	300.1%
7	21.437	63.141	1344.51	294.5%
8	22.836	70.46	1378.29	308.5%
9	23.366	71.493	1421.28	305.9%
10	28.427	84.898	1757.02	298.7%

TABLE 7. Statistics of query window generation time.

t	T	Generation time (s)			Efficiency increase (%)
		Proposed	[23]	[16]	
2	4^2	3.269	11.984	98.9	366.6%
3	4^3	3.858	12.461	141.8	322.9%
4	4^4	3.892	12.341	185.7	317.1%
5	4^5	3.981	12.399	229.4	311.5%
6	4^6	4.123	12.648	274.7	306.8%
7	4^7	4.156	12.613	329.2	303.5%
8	4^8	4.469	12.745	392.8	285.2%
9	4^9	4.882	12.719	457.5	260.5%
10	4^{10}	4.804	13.759	543.5	286.4%
11	4^{11}	4.81	13.001	580.5	270.3%
12	4^{12}	4.828	13.476	582.6	279.1%
13	4^{13}	4.868	13.687	618	281.2%
14	4^{14}	4.911	13.542	664.7	275.7%

During the experiment, the size of the Hilbert curve was set to $4^t \times 4^t \times 4^t$ ($t \in (2, 3, \ldots, 14)$). Different query window starting points were randomly selected within the scope of the Hilbert curve, and the proposed algorithm was used to generate the Hilbert code segments corresponding to the query window. Each size of curve was calculated 1,000 times. The total generation time required by the three algorithms under each size of window is shown in Table 7. Fig. 14 is a comparison of the three algorithms.

From Fig. 14 and Table 7, we can see that the time required for the three algorithms to generate Hilbert code segments gradually increases with the size of the Hilbert curve. Comparing Hilbert curves of the same size, the algorithm in [16] requires significantly more time than the one

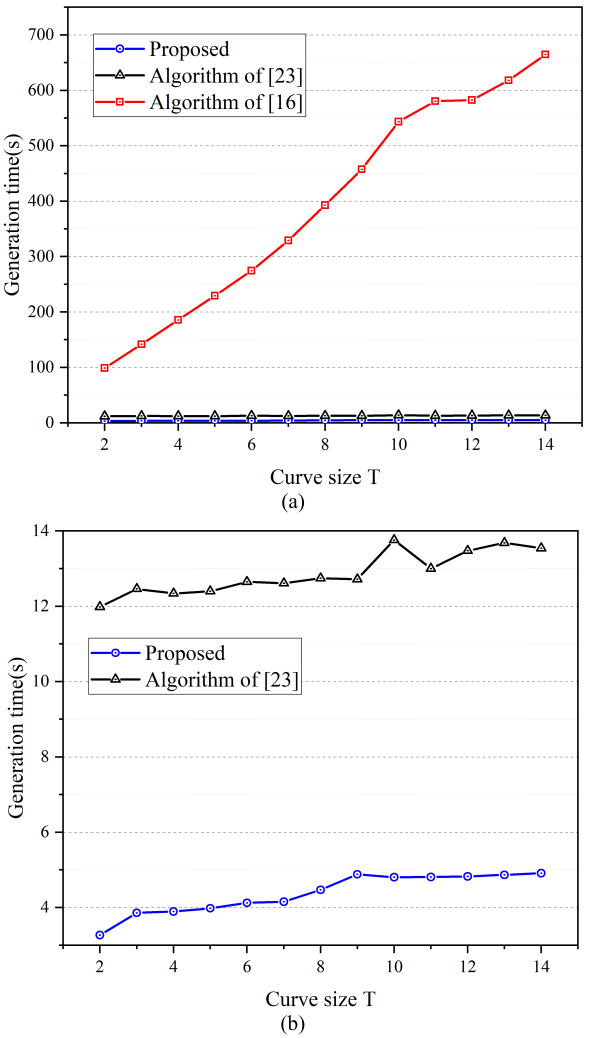

FIGURE 14. Comparison of generation time of Hilbert curve of different sizes. (a) Comparison of the generation time of the three algorithms. (b) Comparison of the generation time for the proposed algorithm and that in [23].

described in [23] and the proposed algorithm. This is because the algorithm in [16] calculates the Hilbert code by traversing several curves, which is inefficient. By contrast, the proposed algorithm reduces the number of calculations by outputting the code segment corresponding to the curve when the window size is equal to the curve size, eliminating the need to check all the grid elements in the window. Thus, the proposed method requires fewer calculations than that of the traversal method and provides improved efficiency. The proposed algorithm also specifies the recursive calling order of the algorithm and directly outputs a set of monotonically increasing code segments. However, the output code segment of the algorithm in [23] is random and unordered; therefore, it requires an additional step for sorting. As a result, comparing the time required for the proposed algorithm against the algorithm in paper [23] under the same size Hilbert curve, the proposed algorithm takes less time and improves the efficiency by 260.5% to 366.6%.

Through the experimental comparison of the different situations above, it can be concluded that the proposed algorithm is better than the other two algorithms in generating a query window corresponding to a Hilbert code segment.

V. CONCLUSION

This paper proposes an algorithm MI-HCS to quickly transform the query window into a set of corresponding monotonically increasing Hilbert code segments and elaborates on the algorithm process in detail. Design experiments were performed to verify the algorithm complexity, and the efficiency was compared against existing algorithms. The experimental results show that the proposed algorithm is better than the existing algorithms in terms of efficiency of query window Hilbert code segment generation and increasing the speed by 260.5% to 423.9%. Therefore, the MI-HCS algorithm can be better applied to window query based on Hilbert code.

Reviewing the algorithm in this paper, we summarize the following disadvantages and shortcomings:

- The recursive splitting process in the algorithm in this paper can be disassembled into several independent splitting processes. However, at present, this paper only uses a serial method for calculation, and the calculation efficiency has a certain impact.
- Based on the state vector, the algorithm in this paper realizes the conversion from three-dimensional query range to Hilbert code. However, because the state vectors of Hilbert curves in different dimensions are different, the algorithm in this paper cannot be directly applied to the query range transformation of higher dimensions.
- The algorithm in this paper can realize the conversion from query range to fixed-resolution Hilbert code, but it cannot be applied to the conversion of multi-resolution Hilbert codes.

In view of the shortcomings in the above summary, further work on this subject will include the following:

- Investigate how to combine parallel computing technology to allocate each independent segmentation process to each parallel computing thread and improve computing efficiency.
- Further study the filling properties of Hilbert curves in higher dimensions, calculate the state vectors of Hilbert curves in each dimension, and summarize the general algorithms applicable to Hilbert curves in various dimensions.
- Improve the termination condition of recursive splitting of the algorithm in this paper, and realize the adaptive conversion of query range to multi-resolution Hilbert curves.

REFERENCES

[1] D. J. Abel and D. M. Mark, "A comparative analysis of some two-dimensional orderings," *Int. J. Geographical Inf. Syst.*, vol. 4, no. 1, pp. 21–31, Jan. 1990.
[2] F. Lu and C. H. Zhou, "An Algorithm for hilbert ordering code based on spatial hierarchical decomposition," *J. Image Graph.*, vol. 6, no. 5, pp. 465–469, 2001.
[3] M. F. Mokbel, W. G. Aref, and I. Kamel, "Analysis of multi-dimensional space-filling curves," *Geoinformatica*, vol. 7, no. 3, pp. 179–209, 2003.
[4] A. Kumar, "Mean-variance analysis of the performance of spatial ordering methods," *Int. J. Geographical Inf. Sci.*, vol. 12, no. 3, pp. 269–289, May 1998.
[5] H. V. Jagadish, "Analysis of the Hilbert curve for representing two-dimensional space," *Inf. Process. Lett.*, vol. 62, no. 1, pp. 17–22, Apr. 1997.
[6] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the Hilbert space-filling curve," *IEEE Trans. Knowl. Data Eng.*, vol. 13, no. 1, pp. 124–141, Jan./Feb. 2001.
[7] H. Samet, "Foundations of multidimensional and metric data structures," in *Computer Graphics and Geometric Modeling* San Mateo, CA, USA: Morgan and Kaufmann 2005.
[8] Q. Li, Y. Lu, X. Gong, and J. Zhang, "Optimizational method of HBase multi-dimensional data query based on Hilbert space-filling curve," in *Proc. 9th Int. Conf. P2P, Parallel, Grid, Cloud Internet Comput.*, Nov. 2014, pp. 469–474.
[9] D. T. Cintra, R. B. Willmersdorf, P. R. M. Lyra, and W. W. M. Lira, "A parallel DEM approach with memory access optimization using HSFC," *Eng. Comput.*, vol. 33, no. 8, pp. 2463–2488, Nov. 2016.
[10] E. Nardelli and G. Proietti, "Time and space efficient secondary memory representation of quadtrees," *Inf. Syst.*, vol. 22, no. 1, pp. 25–37, Mar. 1997.
[11] E. Nardelli and G. Proietti, "Efficient secondary memory processing of window queries on spatial data," *Inf. Sci.*, vol. 84, nos. 1–2, pp. 67–83, May 1995.
[12] M. Meijers and P. van Oosterom, "Clustering and indexing historic vessel movement data with space filling curves," *ISPRS-Int. Arch. Photogram., Remote Sens. Spatial Inf. Sci.*, vol. 4, pp. 417–424, Sep. 2018.
[13] P. van Oosterom, O. Martinez-Rubi, M. Ivanova, M. Horhammer, D. Geringer, S. Ravada, T. Tijssen, M. Kodde, and R. Gonçalves, "Massive point cloud data management: Design, implementation and execution of a point cloud benchmark," *Comput. Graph.*, vol. 49, pp. 92–125, Jun. 2015.
[14] P. J. Couch, B. D. Daniel, and T. H. McNicholl, "Computing space-filling curves," *Theory Comput. Syst.*, vol. 50, no. 2, pp. 370–386, 2012.
[15] J. Zhang and S.-I. Kamata, "A generalized 3-D Hilbert scan using look-up tables," *J. Vis. Commun. Image Represent.*, vol. 23, no. 3, pp. 418–425, Apr. 2012.
[16] J. K. Lawder and P. J. H. King, "Querying multi-dimensional data indexed using the Hilbert space-filling curve," *ACM SIGMOD Rec.*, vol. 30, no. 1, pp. 19–24, Mar. 2001.
[17] J. K. Lawder and P. J. H. King, "Using space-filling curves for multi-dimensional indexing," in *Proc. Brit. Nat. Conf. Databases*. Berlin, Germany: Springer, 2000, pp. 20–35.
[18] F.-C. Hu, Y.-H. Tsai, and K.-L. Chung, "Space-filling approach for fast window query on compressed images," *IEEE Trans. Image Process.*, vol. 9, no. 12, pp. 2109–2116, Dec. 2000.
[19] W. G. Aref and H. Samet, "Decomposing a window into maximal quadtree blocks," *Acta Inf.*, vol. 30, no. 5, pp. 425–439, May 1993.
[20] C.-C. Chang, J.-Y. Hsiao, and J.-C. Yeh, "A novel lossy image compression scheme based on Hilbert curve and VQ suitable for fast window query," in *Proc. 2nd Int. Symp. Intell. Inf. Technol. Appl.*, Dec. 2008, pp. 341–345.
[21] K.-L. Chung, Y.-L. Huang, and Y.-W. Liu, "Efficient algorithms for coding Hilbert curve of arbitrary-sized image and application to window query," *Inf. Sci.*, vol. 177, no. 10, pp. 2130–2151, May 2007.
[22] C.-C. Wu and Y.-I. Chang, "Quad-splitting algorithm for a window query on a Hilbert curve," *IET Image Process.*, vol. 3, no. 5, pp. 299–311, Oct. 2009.
[23] X. Guan, P. van Oosterom, and B. Cheng, "A parallel N-Dimensional space-filling curve library and its application in massive point cloud management," *ISPRS Int. J. Geo-Inf.*, vol. 7, no. 8, p. 327, 2018.
[24] X. Liu and G. F. Schrack, "An algorithm for encoding and decoding the 3-D Hilbert order," *IEEE Trans. Image Process.*, vol. 6, no. 9, pp. 1333–1337, Sep. 1997.
[25] P. M. Campbell, K. D. Devine, and J. E. Flaherty, "Dynamic octree load balancing using space-filling curves," Dept. Comput. Sci., Williams College, Williamstown, MA, USA, Tech. Rep. CS-03-01, 2003.
[26] A. U. Frank, "Qualitative spatial reasoning about distances and directions in geographic space," *J. Vis. Lang. Comput.*, vol. 3, no. 4, pp. 343–371, Dec. 1992.
[27] C. Faloutsos, H. V. Jagadish, and Y. Manolopoulos, "Analysis of the n-dimensional quadtree decomposition for arbitrary hyperrectangles," *IEEE Trans. Knowl. Data Eng.*, vol. 9, no. 3, pp. 373–383, May/Jun. 1997.

YUHAO WU received the B.S. degree in environmental engineering from Information Engineering University, Zhengzhou, China, in 2017, where he is currently pursuing the M.S. degree.

He has published several articles on discrete global grid systems. His current research interest includes discrete global grid systems.

XUEFENG CAO received the Ph.D. degree in cartography and geographic information engineering from Information Engineering University, Zhengzhou, China, in 2012.

He is currently an Associate Professor with Information Engineering University. He has published several articles on discrete global grid systems and unmanned air vehicle technique. His current researches mainly focus on discrete global grid systems and unmanned air vehicle technique.

WANZHONG SUN received the Ph.D. degree in cryptography from Information Engineering University, Zhengzhou, China, in 2009.

He is currently a Senior Lecturer with Information Engineering University. He has published several articles on unmanned air vehicle technique. His current research interests include unmanned air vehicle and geospatial information security technique.