

LOCALITY-PRESERVING PROPERTIES OF  
SPACE-FILLING CURVES

By

HUA LI

Bachelor of Physics  
Nanjing University  
Nanjing, China  
1989

Doctor of Philosophy  
Nanjing University  
Nanjing, China  
1995

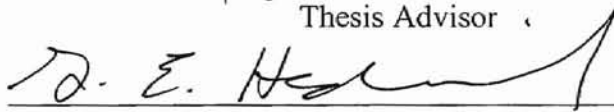
Submitted to the Faculty of the  
Graduate College of the  
Oklahoma State University  
in partial fulfillment of  
the requirement for  
the Degree of  
MASTER OF SCIENCE  
May, 2002

LOCALITY-PRESERVING PROPERTIES OF  
SPACE-FILLING CURVES

Thesis Approved:



Thesis Advisor





  
Dean of the Graduate College

## **ACKNOWLEDGMENTS**

I would like to extend my sincere gratitude to Dr. H.K. Dai, my advisor, for his guidance, ideas, and expertise in helping me to develop and complete this study. Sincerely thanks to Dr. Heisterkamp for his support in computing facilities. Sincerely thanks to Dr. Hedrick and Dr. Chandler for their patience, support and services as members of my advisory committee.

This thesis is dedicated to my husband Eric. It's his love, support, encouragement, and patience that have kept me motivated and escorted me through the years.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION .....	1
II. SPACE-FILLING CURVES.....	4
III. LOCALITY OF SPACE-FILLING CURVES: SURVEY AND RELATED WORK.....	7
IV. SIMULATIONS AND ANALYSES .....	12
V. APPLICATIONS IN IMAGE PROCESSING.....	48
VI. CONCLUSION.....	55
REFERENCES.....	57
APPENDIX.....	60
Source code <thesis.cpp>.....	60
<hilbert.h>.....	67

## LIST OF TABLES

Table	Page
1. Inter-cluster distance constants for three kinds of space-filling curves vs. three kinds of query shapes for two-dimensional grid spaces. The first column is the result in $2^7 \times 2^7$ grid space and the second is in $2^{15} \times 2^{15}$ grid space.....	32
2. Ratios of inter-cluster distance among three kinds of space-filling curves for three kinds of query shapes on three-dimensional grid spaces. The result is for $2^{10} \times 2^{10} \times 2^{10}$ grid. ....	33

## LIST OF FIGURES

Figure	Page
1 Illustration of two-dimensional space-filling curves: (a) Z-curve, (b) Gray-coded curve, (c) Hilbert curve.....	3
2 The first three steps of the two-dimensional Hilbert space-filling curve: (a) first step, (b) second step, (c) third step. ....	7
4-1 Average number of clusters for two-dimensional queries (exhaustive simulation on $2^7 \times 2^7$ grid): (a) square queries, (b) circular queries, and (c) rectangular queries.....	15
4-2. Average number of clusters for two-dimensional queries (sampling simulation on $2^{15} \times 2^{15}$ grid): (a) square queries, (b) circular queries, and (c) hyper-rectangular queries. ....	17
4-3. Average number of clusters for three-dimensional queries (exhaustive simulation on $2^6 \times 2^6 \times 2^6$ grid): (a) square queries, (b) circular queries, and (c) rectangular queries. ....	19
4-4. Average number of clusters for three-dimensional queries (sampling simulation on $2^{10} \times 2^{10} \times 2^{10}$ grid): (a) cubic queries, (b) spherical queries, and (c) hyper-rectangular queries. ....	21
4-5. Average query indexing ranges for two-dimensional queries (exhaustive simulation on $2^7 \times 2^7$ grid): (a) square queries, (b) circular queries, and (c) rectangular queries. ....	24
4-6. Average query indexing ranges for two-dimensional queries (sampling simulation on $2^{15} \times 2^{15}$ grid): (a) square queries, (b) circular queries, and (c) rectangular queries. ....	26
4-7. Average query indexing ranges for three-dimensional queries (exhaustive simulation on $2^6 \times 2^6 \times 2^6$ grid): (a) cubic queries and (b) spherical queries. ....	28
4-8. Average query indexing ranges for three-dimensional queries (sampling simulation on $2^{10} \times 2^{10} \times 2^{10}$ grid): (a) cubic queries and (b) spherical queries. ....	30

4-9. Average inter-cluster distances for two-dimensional queries (exhaustive simulation on $2^7 \times 2^7$ grid): (a) square queries, (b) circular queries, and (c) rectangular queries. ....	35
4-10. Average inter-cluster distances for two-dimensional queries (sampling simulation on $2^{15} \times 2^{15}$ grid): (a) square queries, (b) circular queries, and (c) rectangular queries. ....	37
4-11. Average inter-cluster distances for three-dimensional queries (exhaustive simulation on $2^6 \times 2^6 \times 2^6$ grid): (a) cubic queries, (b) spherical queries, and (c) hyper-rectangular queries. ....	39
4-12. Average inter-cluster distances for three-dimensional queries (sampling simulation on $2^{10} \times 2^{10} \times 2^{10}$ grid): (a) cubic queries, (b) spherical queries, and (c) hyper-rectangular queries. ....	41
4-13. Index-difference versus taxi/cab distance on two-dimensional grid (a) exhaustive simulation on $2^7 \times 2^7$ grid and (b) sampling simulation on $2^{15} \times 2^{15}$ grid. ....	45
4-14. Ratio of the index-difference versus taxi/cab distance on two-dimensional grid (a) exhaustive simulation on $2^7 \times 2^7$ grid and (b) sampling simulation on $2^{15} \times 2^{15}$ grid. ....	46
4-15. Index-difference versus taxi/cab distance on three-dimensional grid (a) exhaustive simulation on $2^6 \times 2^6 \times 2^6$ grid and (b) sampling simulation on $2^{10} \times 2^{10} \times 2^{10}$ grid. ....	47
4-16. Ratio of the index-difference versus taxi/cab distance on three-dimensional grid (a) exhaustive simulation on $2^6 \times 2^6 \times 2^6$ grid and (b) sampling simulation on $2^{10} \times 2^{10} \times 2^{10}$ grid. ....	48
5-1. (a) A deer image applied Hilbert space-filling curve scanning, (b) Auto-correlation of the three scanned images under three families of space-filling curves and row-by-row scanning. ....	53
5-2. Auto-correlation of the three scanned images under three families of space-filling curves and row-by-row scanning, in which the image is 200 randomly placed squares of size $8 \times 8$ on $2^{10} \times 2^{10}$ grid. ....	54

1905. The first formal representation of space-filling curve was given by Hilbert [Hil91].

Several other curves have been proposed in the literature. One based on

## CHAPTER ONE

### INTRODUCTION

Indexing of multi-dimensional data has been the focus of a considerable amount of research effort over many years but no generally agreed paradigm has emerged to compare with the impact of B-trees, for example, on the indexing of one-dimensional data. Gaede and Günther [GG98] gave an extensive review of multi-dimensional access methods. At the same time, the need for efficient multi-dimensional indexing methods is ever more important in an environment where databases become larger and more complex in their structures and aspirations for extracting valuable information become more sophisticated.

Mapping multi-dimensional data to one dimension, enabling simple and well-understood one-dimensional access methods to be exploited, has been suggested as a solution in the literature by Faloutsos [Fal86][FS89]. One way of implementing such a mapping is to utilize space-filling curves. For a historical account of classical space-filling curves, we could refer to Sagan's book [Sag94].

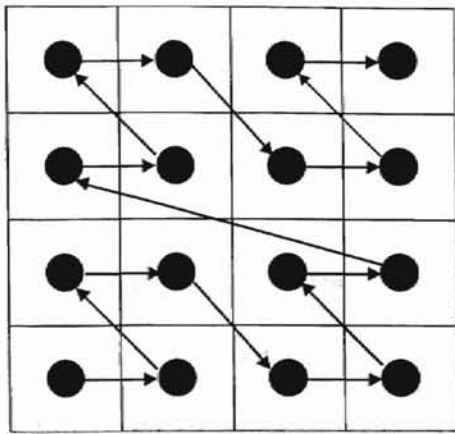
A space-filling curve is a linear traversal of a discrete finite multi-dimensional space, thus it passes through every point in a space once so giving a one-to-one correspondence between the coordinates of the points and the one-dimensional sequence numbers of the curve. Space-filling curves were a topic of interest for mathematicians in the late 19th



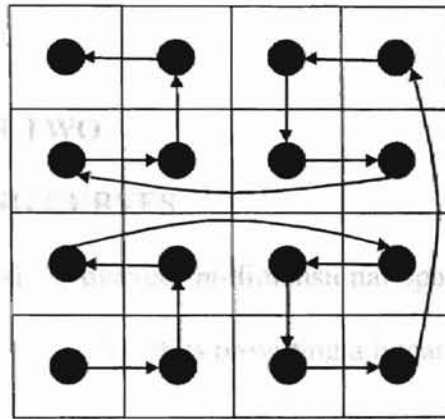
century. The first graphical representation of space-filling curve was given by Hilbert [Hil1891].

Several mapping functions have been proposed in the literature. One based on interleaving bits from the coordinates, which is called Z-ordering, was proposed by Orenstein [Ore86]. Its modification was suggested by Faloutsos [Fal86] using Gray coding on the interleaved bits. A third mapping method, based on Hilbert space-filling curve. In a mathematical context, these three mapping functions are based on different space-filling curves: Z-curves, the Gray-coded curves and the Hilbert curves. Figure 1 illustrates the linear ordering yielded by the space-filling curves for a 4x4 grid. More information on the will be presented in the next chapter.

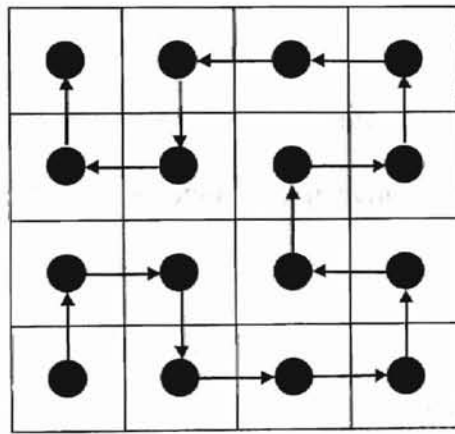




(a)



(b)



(c)

Figure 1. Illustration of two-dimensional second order space-filling curves: (a) Z-curve, (b) the Gray-coded curve, (c) the Hilbert curve.

## CHAPTER TWO

### SPACE-FILLING CURVES

For positive  $N$ , denote  $\{1, \dots, N\}$  by  $[N]$ . A discrete  $m$ -dimensional space-filling curve of length  $N^m$  is a bijective mapping  $C : [N^m] \rightarrow N^m$ , thus providing a linear traversal ordering of the grid points in  $[N^m]$ . An  $m$ -dimensional grid is said to be of order  $k$  if it has size  $N = 2^k$ ; a space-filling curve has order  $k$  if its codomain is a grid of order  $k$ . The generation of a sequence of multi-dimensional space-filling curves usually follows a recursive process.

Although it was G. Peano who discovered the first space-filling curve, it was Hilbert who made this phenomenon of surface-filling curves luminous to the geometric imagination. Hilbert in 1891, first recognized a general geometric procedure that allows the construction of an entire class of space-filling curves [Hil1891]. Thereupon he proceeded to promulgate the following heuristic principle: If the interval  $I$  can be mapped continuously onto the square  $S$ , then after dividing  $I$  into four congruent subintervals and  $S$  into four congruent subsquares, each subinterval can be mapped continuously onto one of the subsquares. Next, each subinterval is, in turn, partitioned into four congruent subintervals and each subsquare into four congruent subsquares, and the process is repeated. If this is carried on infinitely,  $I$  and  $S$  are partitioned into  $2^{2n}$  congruent replicas for  $n = 1, 2, 3, \dots, \infty$ . Hilbert proved that the subsquares can be arranged so that adjacent subintervals correspond to adjacent subsquares with an edge in common, and so that the inclusion relationships are preserved, that is, if a square corresponds to an interval, then

its subsquares correspond to the subintervals of the interval. Figure 2 illustrates how the process is to be actualized for the first three steps. The generalization of a three-dimensional Hilbert curve was described in [Jag90][Sag93]. A generalization of the Hilbert curve, in an analytic form, for higher-dimensional space was given in [But69].

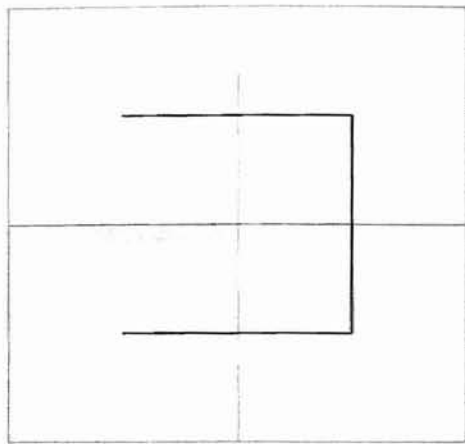
Sagan [Sag94] gave the definition of the Hilbert curve as follows: Every  $t \in I$  is uniquely determined by a sequence of nested closed intervals (that are generated by successive partitioning), the lengths of which shrink to 0. With this sequence corresponds a unique sequence of nested closed squares, the diagonals of which shrink into a point, and which define a unique point in the square  $S$ .

Z-curve is based on Z-ordering, proposed by Orenstein [Ore86], which is based on interleaving bits from the coordinates. For a  $m$ -dimensional Z-curve has order  $k$ , grid size is  $N = 2^k$ . So in binary code, we use the following notation  $c_l^{k-1}c_l^{k-2} \dots c_l^1c_l^0$  to represent the  $l$ th coordinate, where  $1 \leq l \leq m$  and each small  $c$  is 0 or 1. Then the index is constructed from interleaving bits from the coordinates

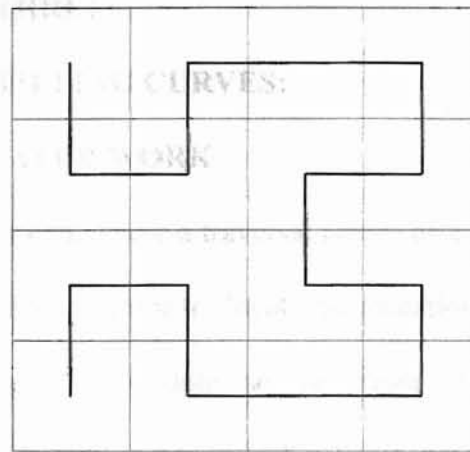
$$c_m^{k-1}c_{m-1}^{k-1} \dots c_1^{k-1} c_m^{k-2}c_{m-1}^{k-2} \dots c_1^{k-2} \dots \dots c_m^0c_{m-1}^0 \dots c_1^0.$$

In the Gray-coded curve, the index is generated from applying Gray coding on the index under Z-ordering. There is one simple way to implement Gray coding is performing XOR between the original index and its one-bit-right-shifted version.

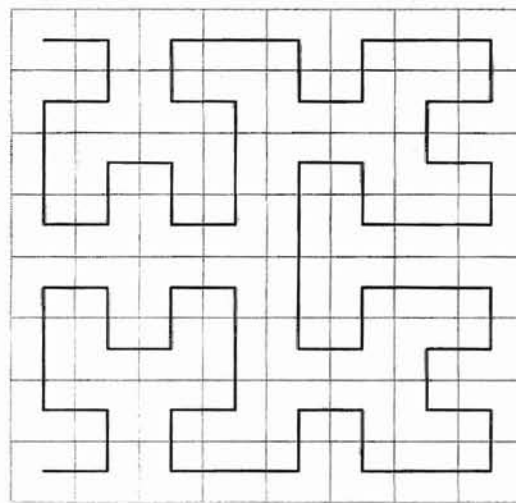
Here in thesis, we will focus on these three families of space-filling curves and try to do some work on the locality properties which we will discuss in the next chapter.



(a)



(b)



(c)

Figure 2. The first three steps of the two-dimensional Hilbert space-filling curve:  
 (a) first step, (b) second step, (c) third step.

CHAPTER THREE Empirical and analytical studies of  
**LOCALITY OF SPACE-FILLING CURVES:** the literature (see  
**SURVEY AND RELATED WORK**

Space-filling curves are useful in applications where a traversal (scan) of a multi-dimensional grid is needed. Some algorithms perform local computations on neighborhoods, or exploit spatial correlation present in data, so the preservation of “locality” is required. By “locality”, we mean that the traversal reflects proximity between the points of  $[N]^m$ , that is, points close in  $[N]^m$  are also close in the traversal order, and vice versa.

The locality-preserving properties of space-filling curves are used for efficient algorithms in various fields of computation (for examples, data organization in multi-dimensional or geographic databases, data compression, image manipulation and compression, computational geometry problems, and parallel computation). In Moon et al.’s work [MSF01], they listed several categories in which applications benefit from linear mapping that preserves locality and also some related references were given out.

Clustering, one of the most desired locality properties, means the locality between points in multi-dimensional space is also kept in the linear space. A cluster is a set of grid points that are consecutively connected by a mapping curve. In terms of multi-dimensional database indexing, all the points within the query range that have consecutive indices will be considered as a cluster.

Under linear mapping, each grid point in multi-dimensional space is given a new index in one-dimensional coordinate system. It is shown that under most situations, the linear mapping based on Hilbert space-filling curves perform better than the others in

preserving locality [Jag90][MJF01]. Much work on empirical and analytical studies of clustering effect of the space-filling curves have been reported in the literature (see [Jag90], [Jag97], [RF91], and [MJF01] for details.)

Jagadish [Jag90] compared the clustering properties of several space-filling curves by considering  $2 \times 2$  range queries. Rong and Faloutsos [RF91] derived a closed-form expression of the average number of the clusters for Z-curves. Jagadish [Jag97] derived closed-form expressions of the average number of clusters for the Hilbert curves in a two-dimensional grid, but only for  $2 \times 2$  and  $3 \times 3$  query regions.

Results are generalized in [MJF01]. Moon et al. analyzed the clustering property of Hilbert space-filling curves by:

- (1) Giving an asymptotic formula for the clustering property of Hilbert space-filling curves for general polyhedra in a multi-dimensional space.
- (2) Derive a closed-form, exact formula of the number of clusters within query region of size  $2^k \times 2^k$  in a two-dimensional grid space of order  $n$ .

Both the asymptotic solution for the general case and the exact solution for a special case generalize previous work in [Jag90]. They agreed with the empirical results that the number of clusters depends on the hyper-surface area of the query region and not its hyper-volume. This works shows that the Hilbert curves achieve better clustering than Z-curves and the Gray-coded curves.

Although the Hilbert curves are superior to Z-curves and also the Gray-coded curves on clustering properties, however clustering is only one of the metric of locality-preserving properties. Besides the clustering measure, there are a few locality measure have been proposed and analyzed for space-filling curves in the literature. Denote by  $d$ ,

$d_1$ , and  $d_\infty$  the Euclidean, taxi-cab, and maximum metrics, respectively. Let  $\mathcal{C}$  denote a family of  $m$ -dimensional curves of successive orders.

For quantifying the proximity preservation of close-by points in the  $m$ -dimensional space  $[n]^m$ , [PKK92] employ an average locality measure:

$$L_{PKK}(C) = \sum_{i,j \in [n^m] | i < j} \frac{|i-j|}{d(C(i), C(j))} \quad \text{for } C \in \mathcal{C}$$

Mitchison and Durbin [MD86] use a more restrictive locality measure parameterized by  $q$ :

$$L_{MD,q}(C) = \sum_{i,j \in [n^m] \text{ and } d(C(i), C(j))=1} |i-j|^q \quad \text{for } C \in \mathcal{C}$$

to study optimal 2-dimensional mappings for  $q \in [0,1]$ .

For measuring the proximity preservation of close-by points in the indexing space  $[n^m]$ , Gotsman and Lindenbaum [GL96] consider the following measures:

$$L_{GL,\min}(C) = \min_{i,j \in [n^m] | i < j} \frac{d(C(i), C(j))^m}{|i-j|}, \text{ and}$$

$$L_{GL,\max}(C) = \max_{i,j \in [n^m] | i < j} \frac{d(C(i), C(j))^m}{|i-j|} \text{ for } C \in \mathcal{C}$$

They show that for arbitrary  $m$ -dimensional curve  $C$ ,

$$L_{GL,\min}(C) = O(n^{1-m}), \text{ and}$$

$$L_{GL,\max}(C) > (2^m - 1) \left(1 - \frac{1}{n}\right)^m$$

For the  $m$ -dimensional Hilbert curve family  $\{H_k^m \mid k = 1, 2, \dots\}$ , they prove that

$$L_{GL,\max}(H_k^m) \leq 2^m (m+3)^{\frac{m}{2}}.$$

For the two-dimensional Hilbert curve family, they obtain tight bounds:



$$6(1 - O(2^{-k})) \leq L_{GL, \max}(H_k^2) \leq 6\frac{2}{3}.$$

Furthermore Dai et al. [DS02] present an analytical study of the locality properties of the  $m$ -dimensional  $k$ -order discrete Hilbert and Z-curve families,  $\{H_k^m \mid k = 1, 2, \dots\}$ , and  $\{Z_k^m \mid k = 1, 2, \dots\}$ , respectively, based on the following locality measure  $L_\delta$  that cumulates all the index-differences between point-pairs at a common taxi-cab distance  $\delta$ , which is similar to  $L_{MD, I}$  conditional on a taxi-cab distance of  $\delta$  between points in  $[n]^m$ .

$$L_\delta(C) = \sum_{i, j \in [n]^m \mid i < j \text{ and } d_{\text{taxi-cab}}(C(i), C(j)) = \delta} |i - j|$$

They derived the exact formulas for  $L_\delta(H_k^m)$  and  $L_\delta(Z_k^m)$  for  $m = 2$  and arbitrary  $\delta$  that is an integral power of 2, and  $m = 3$  and  $\delta = 1$ . The results yield a constant

asymptotic ratio  $\lim_{k \rightarrow \infty} \frac{L_\delta(H_k^m)}{L_\delta(Z_k^m)} > 1$ , which suggests that the Z-curves perform better

than the Hilbert curves.

Besides the clustering properties, we will have interest in other locality properties, such as inter-cluster distance, query indexing range, and index-difference we've mentioned above. By inter-cluster distance, we mean the index space between two neighboring clusters. Query indexing-range means the index range the region query covers. We find the largest index and smallest index inside the query region, then subtract these two indices, and the result is the query indexing range. Inter-cluster and query indexing range will be both collected via region queries. While index-difference can be got through point queries. In chapter 4, we will present the simulation results and analyses of all these locality properties. We will also apply different query shapes on

low-dimensional grid spaces. Our empirical study attempts to find out how these three families of space-filling curves will behave, together with plausible explanations.

Space-filling curves are attractive to many image-space algorithms that are based on the spatial coherence of nearby pixels. The resulting sequence of pixels is processed as required by the particular application. For instance, the sequence may be compressed using lossless or lossy compression, it may be processed for halftoning, analysis, pattern recognition or texture analysis, and it may be converted into an analog form and be transmitted through channels with limited bandwidth. To obtain the image after processing, the pixel-sequence is placed back in a frame along the same space-filling curve. In the above applications and others, it is important that the intraframe correlation in the image translates to a favorable auto-correlation  $\rho(r)$  within the pixel-sequence in which  $\rho(r) = E(f(x)f(x+r))$ ,  $E$  means the expectation over pixel index  $x$ ,  $r$  is the distance between two pixels and  $f(x)$  is the function of scanned image ( $f(x)$  is of value 0 for white pixel and 1 for black pixel in binary images).

As an example of application of space-filling curves on image processing, we will make comparisons between the one-dimensional representation of images that are scanned along the three families of space-filling curves. This will be presented in chapter 5.

approximately  $(N - \delta + 1)^2 (\delta + 1)$ . In a  $d$ -dimensional  $N \times N \times \dots \times N$  grid space, the total number of distinct positions of a  $d$ -dimensional  $k \times k \times \dots \times k$  hyper-cubic query is  $(N - k + 1)^d$ .

## CHAPTER FOUR

### SIMULATIONS AND ANALYSES

To obtain the average inter-cluster distance, it is required that we compute the mean inter-cluster distance within a query region at all possible positions in a given grid space. Exhaustive simulation runs allow us to approach the exact result of the average inter-cluster distance versus the side length of the query region. For circular query and spherical query, side length refers the radius. In a  $d$ -dimensional  $N \times N \times \dots \times N$  grid space, the total number of distinct positions of a  $d$ -dimensional  $k \times k \times \dots \times k$  hyper-cubic query is  $(N - k + 1)^d$ . For query shapes, we consider squares, rectangles, and circles for two-dimensional cases. Here we chose the two sides of the rectangle to 0.5 and 1.5 of the side length. Cubes, hyper-rectangles, and spheres are query shapes for three-dimensional cases. We chose the three sides of the hyper-rectangle to be 0.5, 1.667, and 1 of the side length.

To obtain the relationship between the Euclidean metric for one-dimensional space and taxi/cab (Manhattan) metric for multi-dimensional space, it is necessary that we average index-differences for all possible point-pair queries with a given taxi/cab distance. Such exhaustive simulation runs allow us to approach the exact result of the average index-difference versus taxi/cab distance.

In a two-dimensional  $N \times N$  grid space, the total number of distinct positions of point-pair queries with taxi distance  $\delta$  is approximately  $(N - \delta + 1)^2 (\delta + 1)$ . In a three-dimensional  $N \times N \times N$  grid space, the total number of distinct positions of point-pair

queries with taxi distance  $\delta$  is approximately  $(N - \delta + 1)^2 (\delta + 1) (\delta + 2)/2$ . In a  $d$ -dimensional  $N \times N \times \dots \times N$  grid space, the total number of distinct positions of a  $d$ -dimensional point-pair query is approximately  $(N - k + 1)^d \delta^{d-1}/(d-1)!$ .

Consequently, for a large grid space and a high dimensionality, each simulation run may require processing an excessively large number of queries, which in turn makes the simulation time-consuming. Thus, for large grid spaces, we carried out statistical simulations by random sampling of queries. For small grid spaces, we performed exhaustive simulations. Simulations are performed on two-dimensional and three-dimensional grid spaces.

The first set of experiments was carried out in two-dimensional grid spaces with size  $2^7 \times 2^7$  and  $2^{15} \times 2^{15}$ . For  $2^7 \times 2^7$  grid, we performed exhaustive simulations, and for  $2^{15} \times 2^{15}$  grid we carried out sampling simulations. The second set of experiments was carried out in three-dimensional grid spaces with size  $2^6 \times 2^6 \times 2^6$  and  $2^{10} \times 2^{10} \times 2^{10}$ . For  $2^6 \times 2^6 \times 2^6$  grid, we performed exhaustive simulations, and for  $2^{10} \times 2^{10} \times 2^{10}$  grid we carried out sampling simulations.

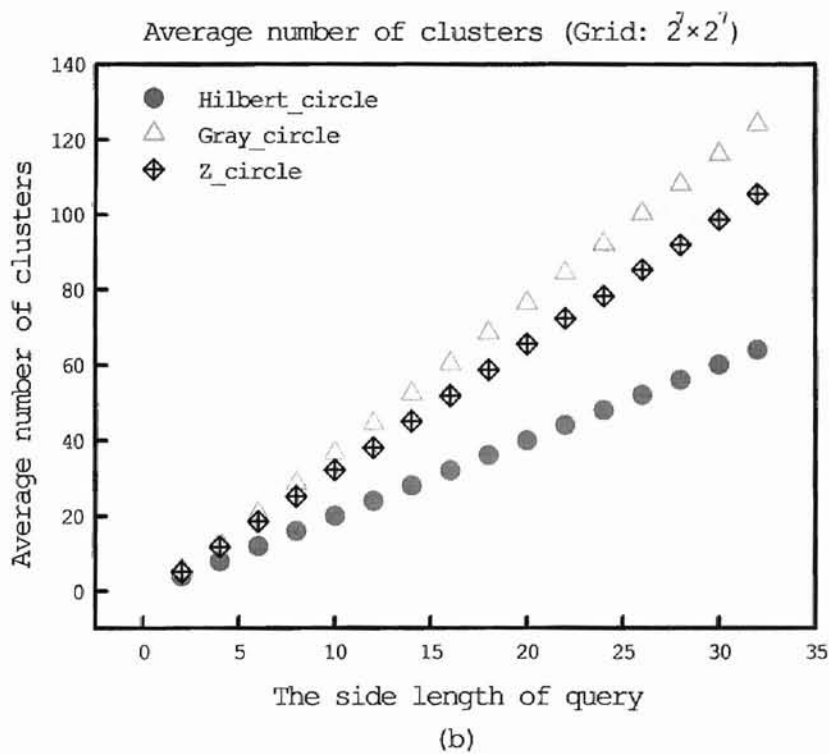
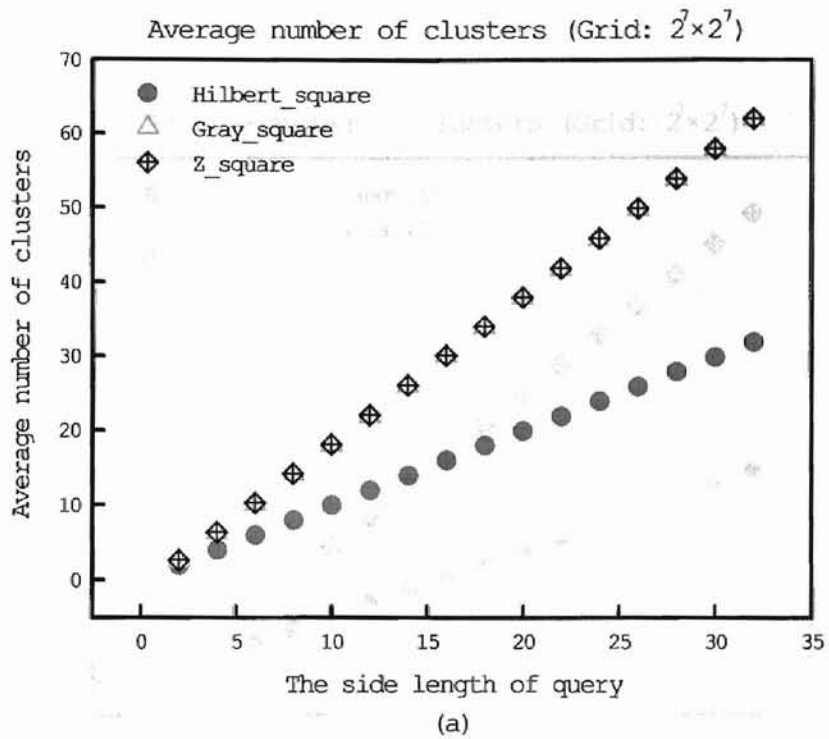
### Results and Analyses on Average Number of Clusters

Figures 4-1 to 4-4 present the average number of clusters for square, rectangular, and circular queries on two-dimensional grid spaces, and cubic, hyper-rectangular, and spherical queries on three-dimensional grid spaces.

The results of average number of clusters comply with those of Moon et al.'s work [MJF01]. The Hilbert curves perform the best on the clustering properties under all dimensionality and query shapes considered. In two-dimensional grid spaces, for square queries and rectangular queries, Z-curves and the Gray-coded curves perform almost the

same; for circular queries, Z-curves achieve better clustering properties than the Gray-coded curves. In three-dimensional grid spaces, the Gray-coded curves outperform Z-curves for all query shapes.

The average number of clusters depends on the hyper-surface area of the query region [MHF01]. For the two-dimensional case, the average number of clusters is a linear function of the side length, for the three-dimensional case, it is a quadratic function of the side length. The Hilbert curves is superior to the other two space-filling curves on the clustering properties.



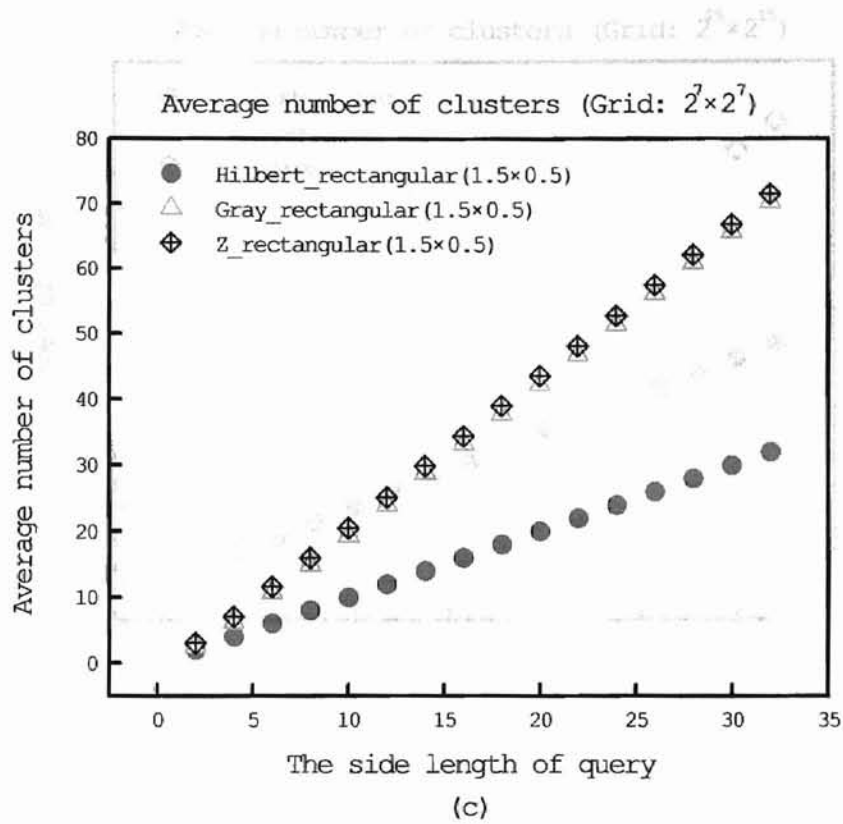
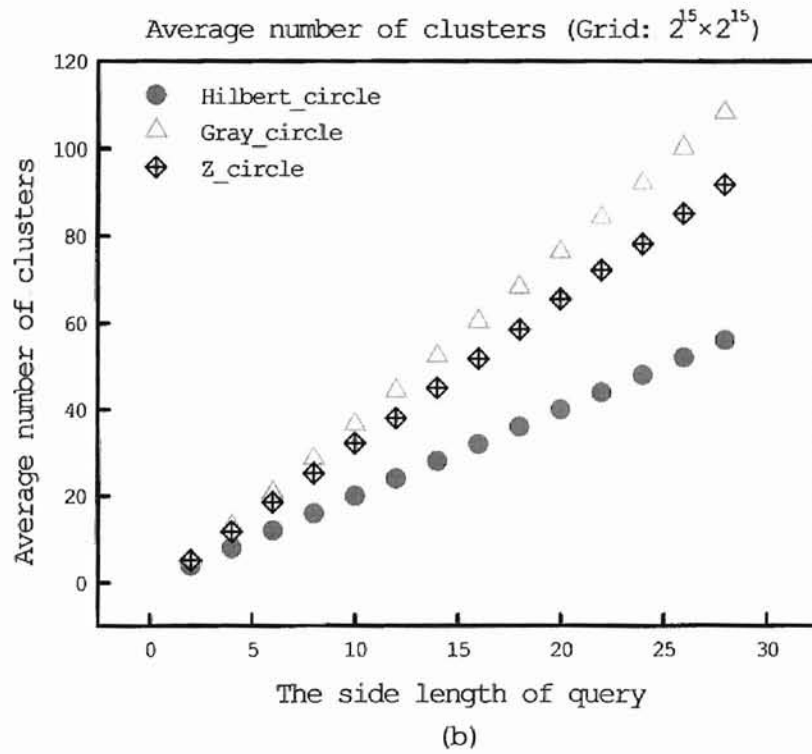
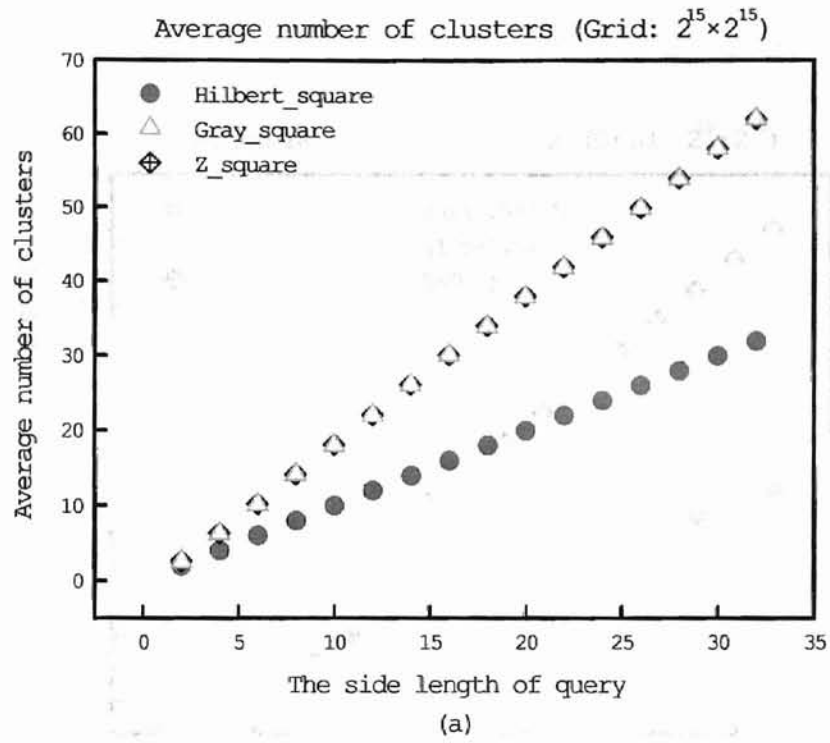


Figure 4-1. Average number of clusters for two-dimensional queries (exhaustive simulation on  $2^7 \times 2^7$  grid): (a) square queries, (b) circular queries, and (c) rectangular queries.





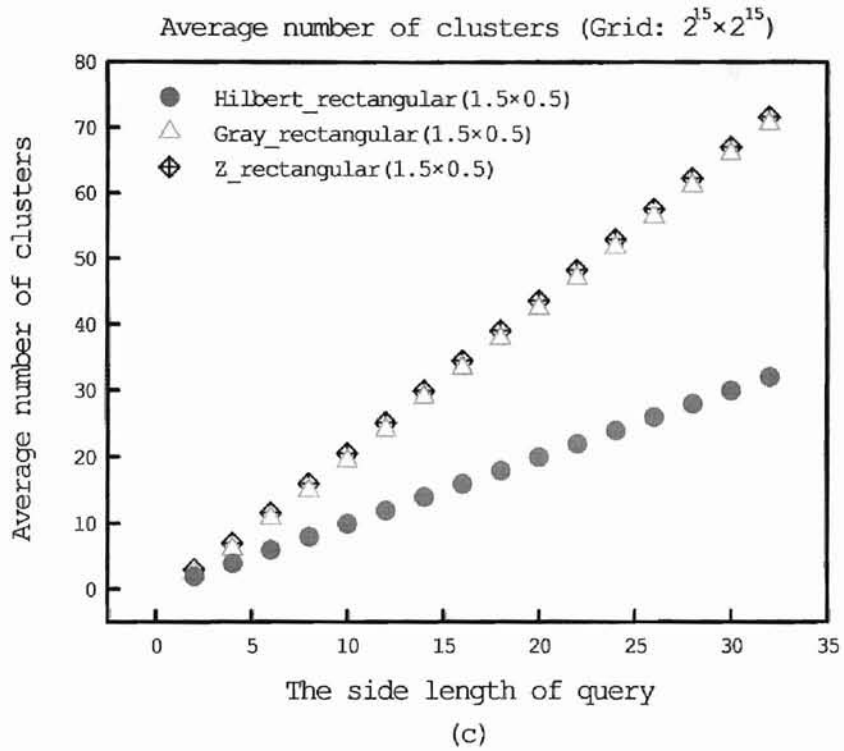
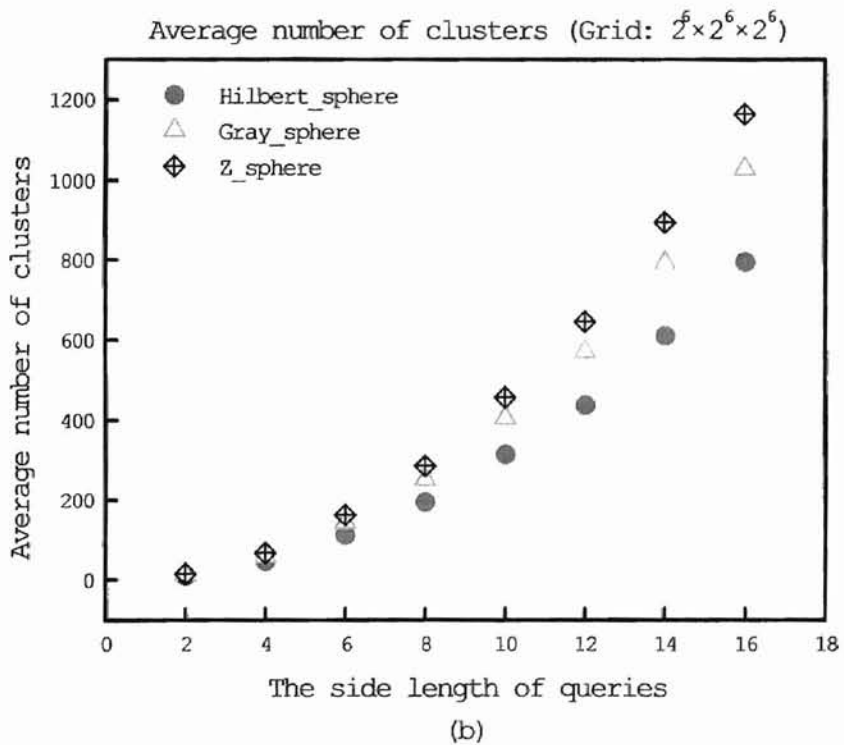
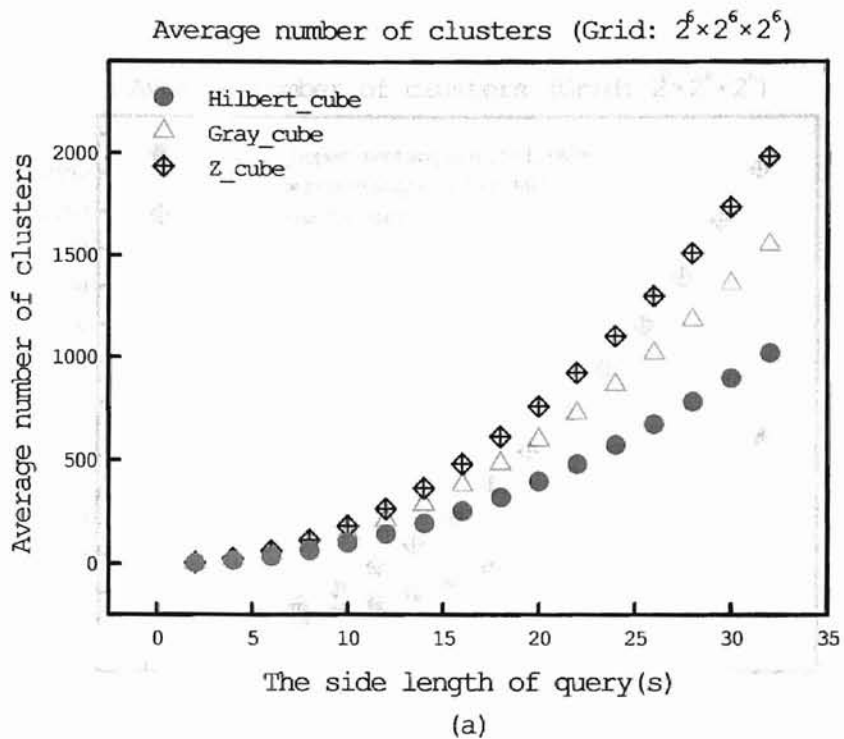


Figure 4-2. Average number of clusters for two-dimensional queries (sampling simulation on  $2^{15} \times 2^{15}$  grid): (a) square queries, (b) circular queries, and (c) rectangular queries.



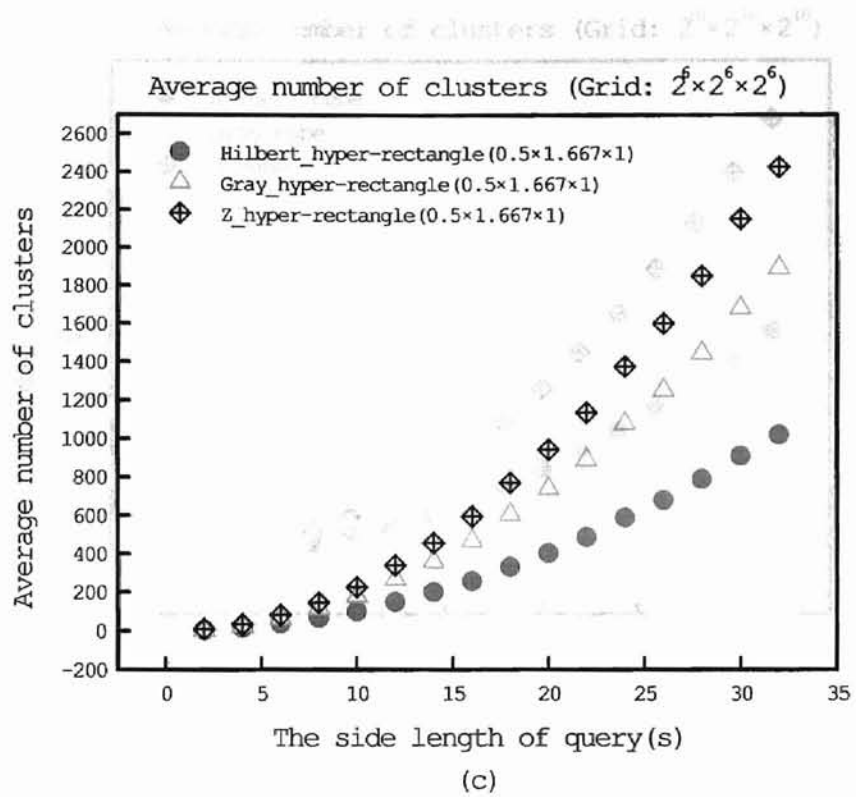
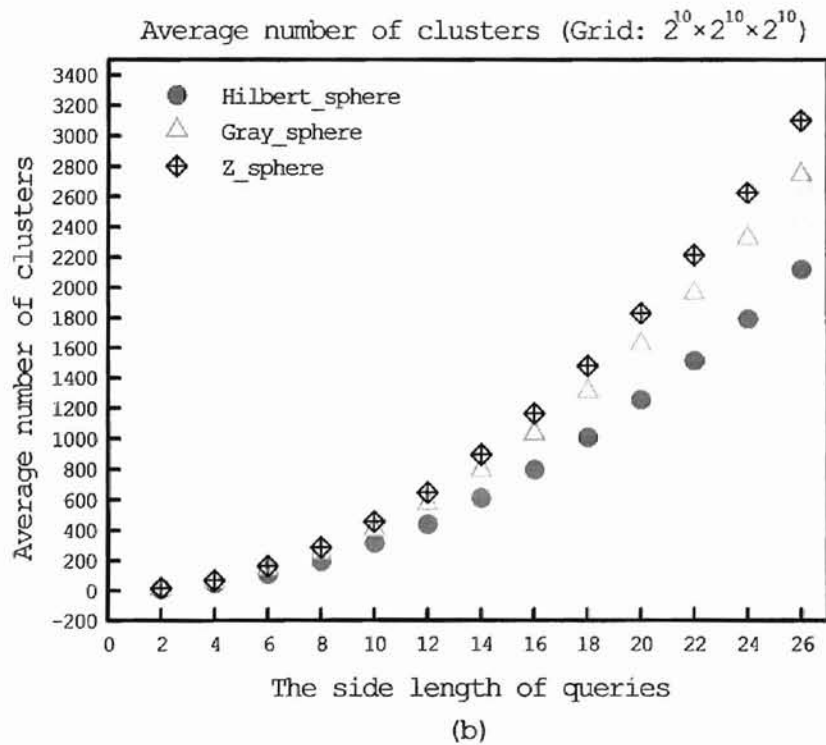
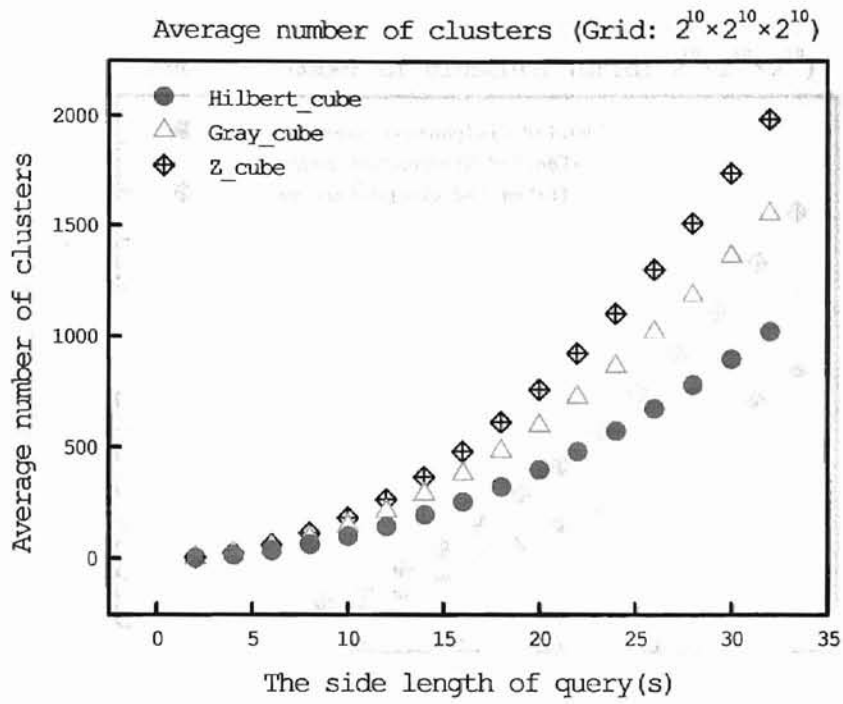
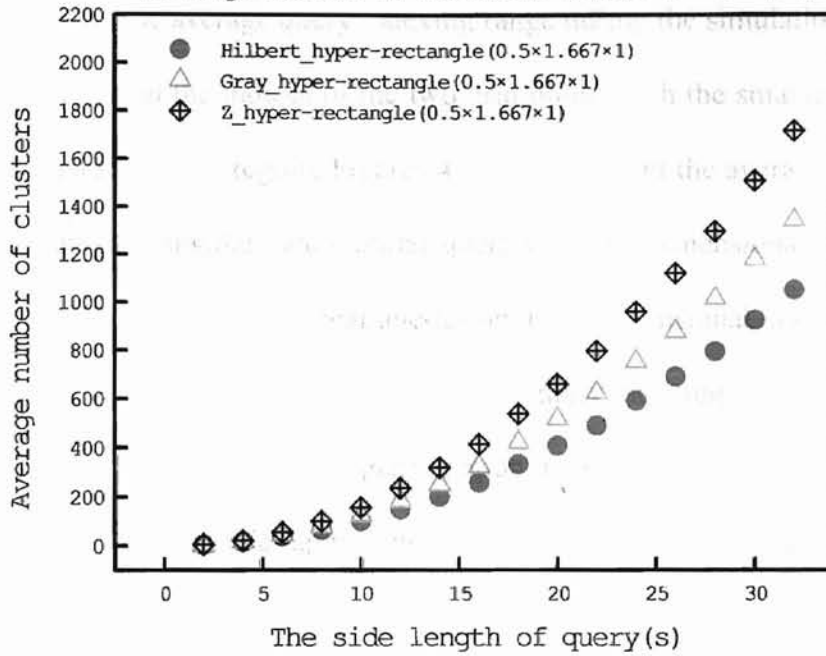


Figure 4-3. Average number of clusters for three-dimensional queries (exhaustive simulation on  $2^6 \times 2^6 \times 2^6$  grid): (a) cubic queries, (b) spherical queries, and (c) hyper-rectangular queries.



Results and Analyses on Query Indexing Range  
 Average number of clusters (Grid:  $2^{10} \times 2^{10} \times 2^{10}$ )



(c)

Figure 4-4. Average number of clusters for three-dimensional queries (sampling simulation on  $2^{10} \times 2^{10} \times 2^{10}$  grid): (a) cubic queries, (b) spherical queries, and (c) hyper-rectangular queries.

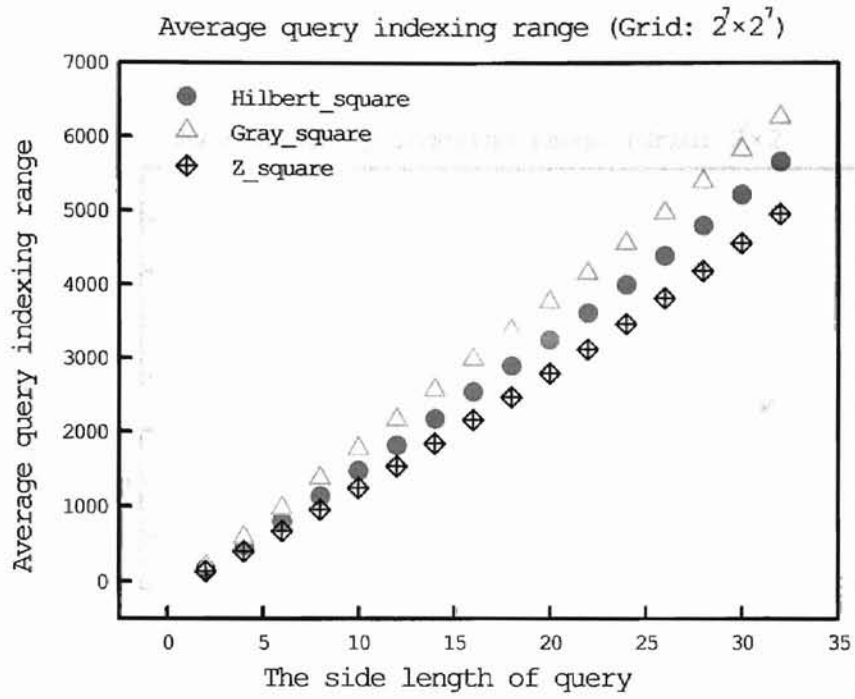
### Results and Analyses on Query Indexing Range

We also collect the average query indexing range during the simulation, which means the difference between the indices of the two grid points with the smallest index and the largest index inside a query region. Figures 4-5 to 4-8 present the average query indexing ranges for square, rectangular, and circular queries on two-dimensional grid spaces, and cubic, hyper-rectangular, and spherical queries on three-dimensional grid spaces.

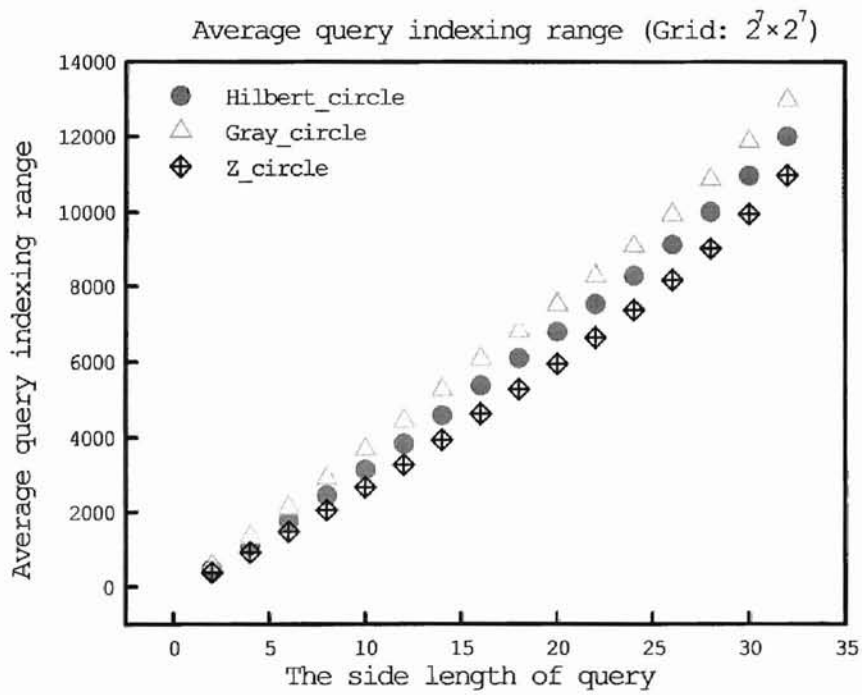
We can see from those figures that the average query indexing range is nearly a linear function of the side length of the query region in both two-dimensional and three-dimensional grid spaces. We apply the linear fit, which fits quite well with the experimental results especially in larger grids with size  $2^{15} \times 2^{15}$  and  $2^{10} \times 2^{10} \times 2^{10}$ . We can also see that the query indexing range depends not only on the side length of the query region, but also on the size of the grid space. The query indexing range increases with the growing size of the grid space. Besides the query indexing range is related to the query shape.

The Gray-coded curves have the largest average query indexing range for all the query shapes. Except two-dimensional rectangular queries, Z-curves have the smallest average query indexing range, and the Hilbert curves are in between.

Generally speaking, Z-curves are the best on the query indexing range property.



(a)



(b)

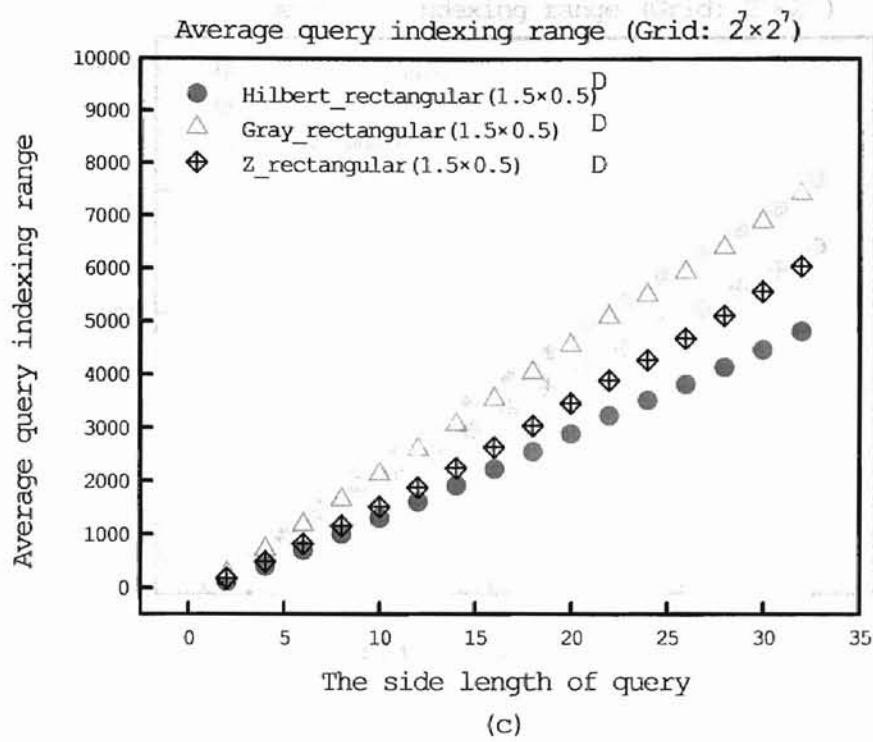
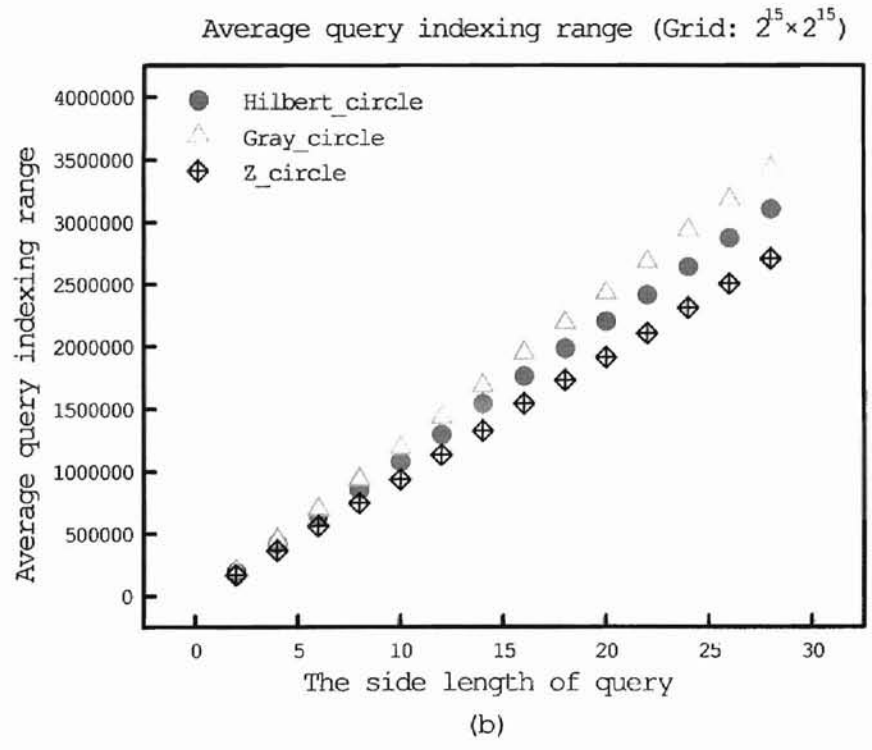
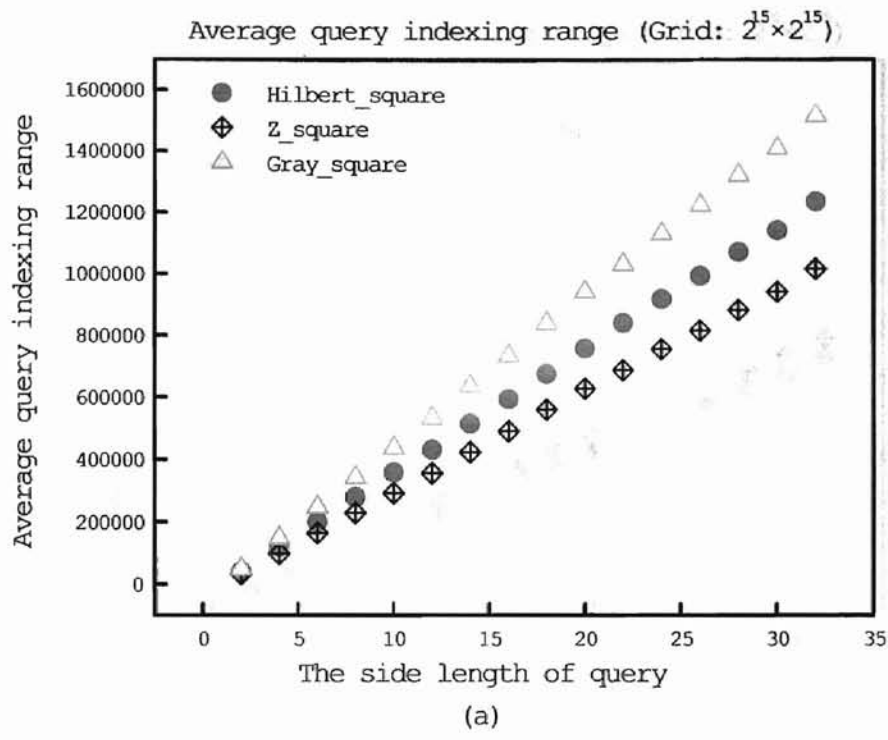


Figure 4-5. Average query indexing range for two-dimensional queries (exhaustive simulation on  $2^7 \times 2^7$  grid): (a) square queries, (b) circular queries, and (c) rectangular queries.





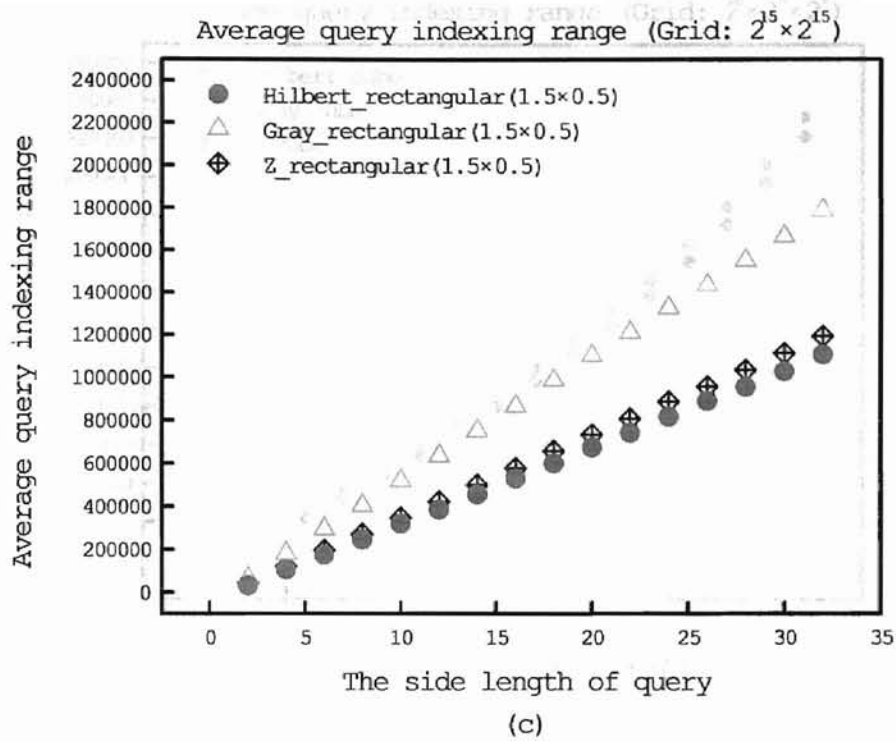
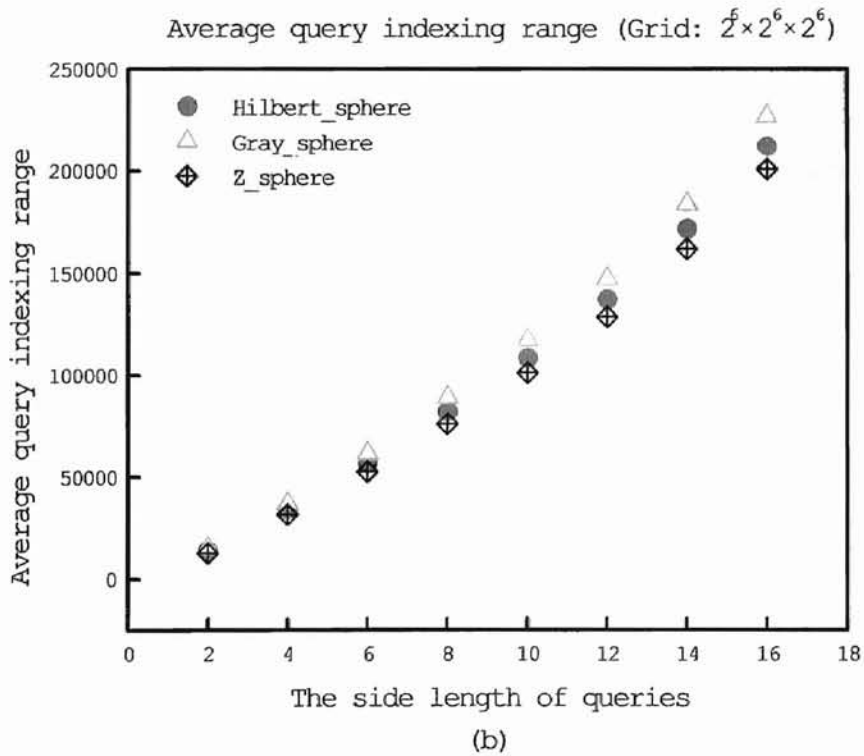
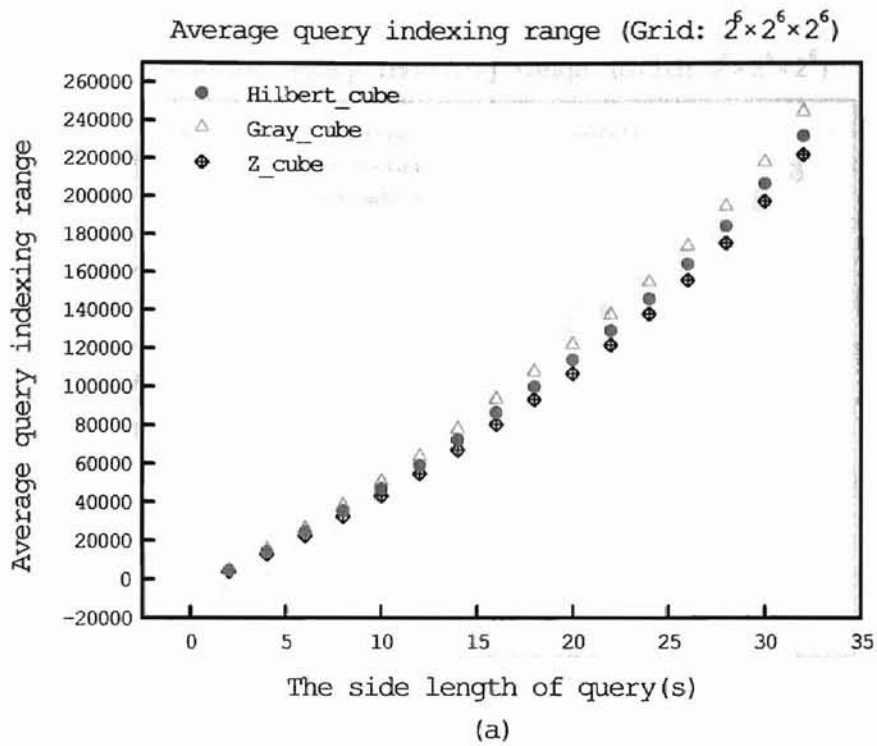


Figure 4-6. Average query indexing range for two-dimensional queries (sampling simulation on  $2^{15} \times 2^{15}$  grid): (a) square queries, (b) circular queries, and (c) rectangular queries.



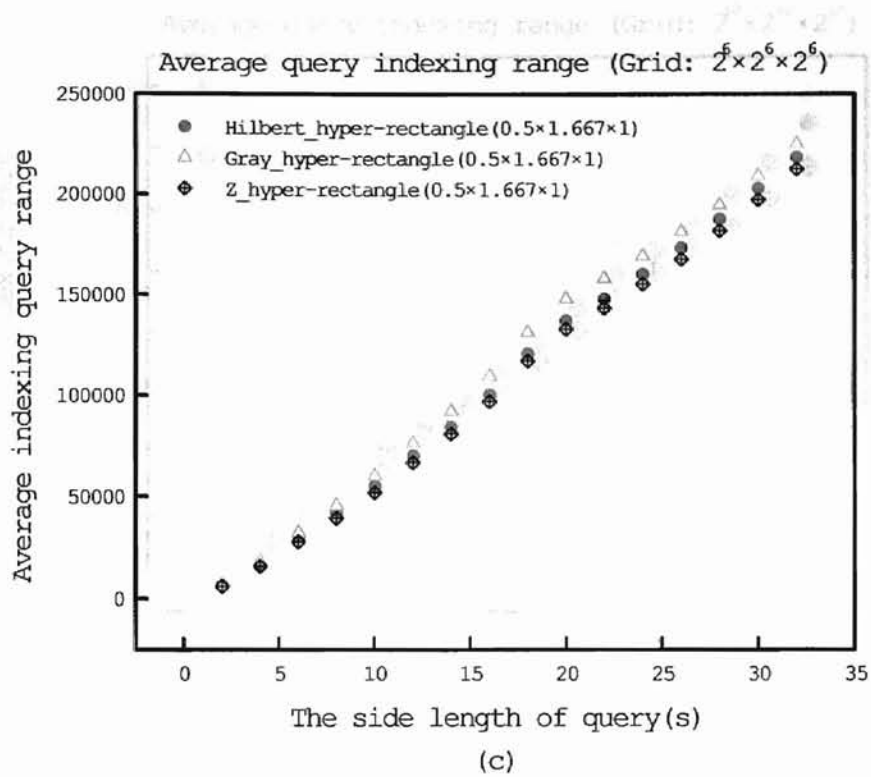
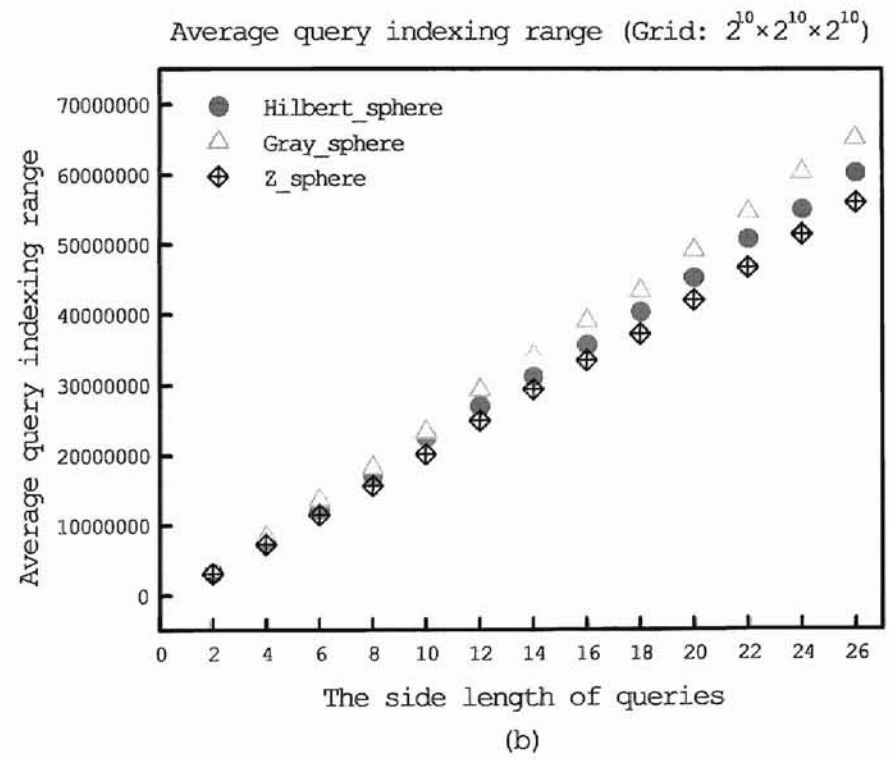
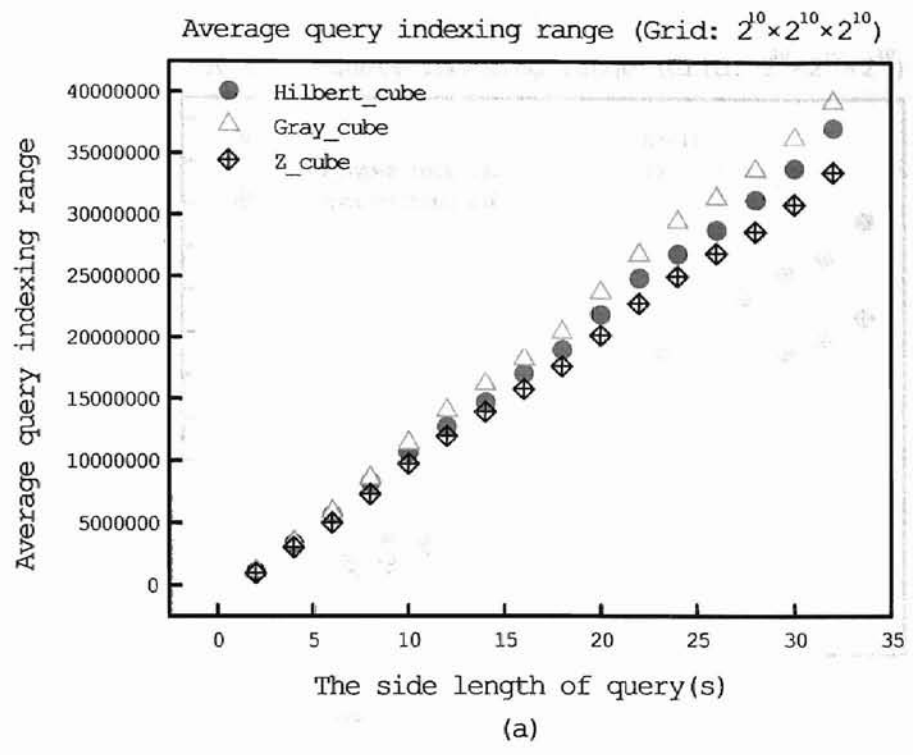


Figure 4-7. Average query indexing range for three-dimensional queries (exhaustive simulation on  $2^6 \times 2^6 \times 2^6$  grid): (a) cubic queries, (b) spherical queries, and (c) hyper-rectangular queries.



Results and Analyses on Average Inter-cluster Distance

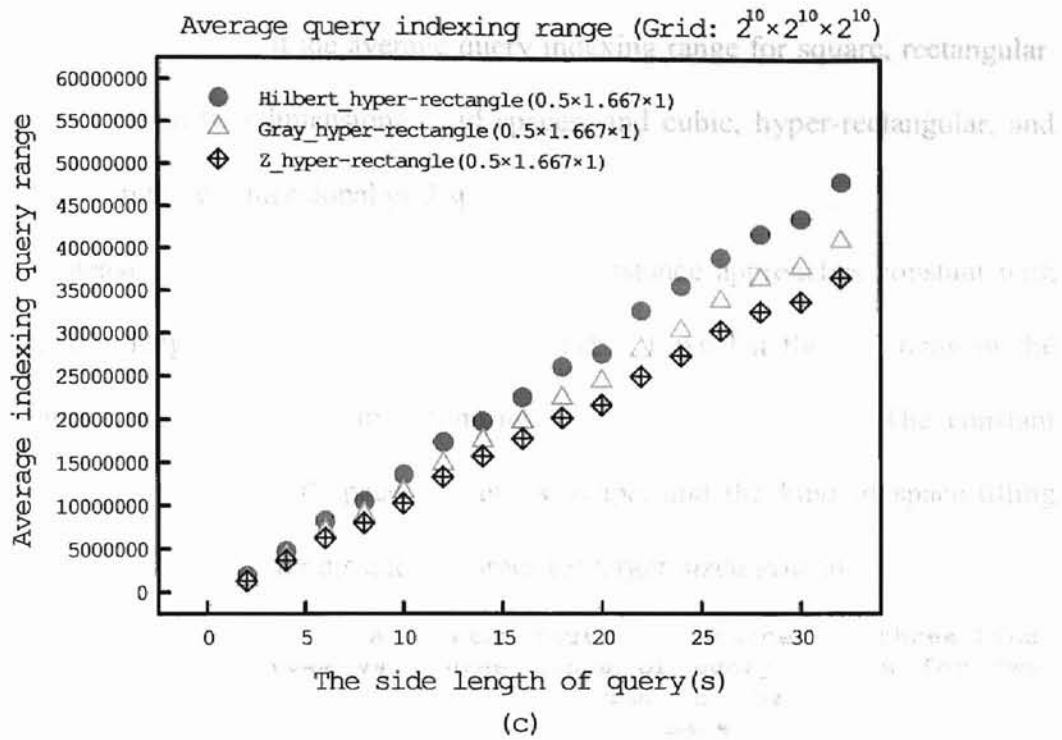


Figure 4-8. Average query indexing range for three-dimensional queries (sampling simulation on  $2^{10} \times 2^{10} \times 2^{10}$  grid): (a) cubic queries, (b) spherical queries, and (c) hyper-rectangular queries.

## Results and Analyses on Average Inter-cluster Distance

Figures 4-9 to 4-12 present the average query indexing range for square, rectangular, and circular queries in two-dimensional grid spaces, and cubic, hyper-rectangular, and spherical queries in three-dimensional grid spaces.

In two-dimensional grid spaces, the inter-cluster distance approaches constant with the growing side length of the query region. In Table 1 we list the constants of the average inter-cluster distance for three families of space-filling curves. The constant depends on the size of the grid space, the query shape, and the kind of space-filling curves. The average inter-cluster distance is larger for larger-sized grid space.

Table 1. Constants of average inter-cluster distance for three kinds of space-filling curves vs. three kinds of query shapes for two-dimensional grid spaces. The first column is the result in  $2^7 \times 2^7$  grid space and the second is in  $2^{15} \times 2^{15}$  grid space.

	Hilbert curve		Gray-coded curve		Z-curve	
	$2^7 \times 2^7$	$2^{15} \times 2^{15}$	$2^7 \times 2^7$	$2^{15} \times 2^{15}$	$2^7 \times 2^7$	$2^{15} \times 2^{15}$
Square	150	40,000	85	25,000	64	16,800
Circle	140	56,000	80	32,000	73	30,000
Rectangle	130	35,000	80	25,000	70	17,000

In three-dimensional grid spaces, the average inter-cluster distance decreases with the growing side length of query region. In Table 2 we list the ratios of the average inter-cluster distance among three families of space-filling curves. From these figures and Table 2, it seems that the ratios approach a constant with the growing side length of the query region.

Table 2. Ratios of the average inter-cluster distance among three kinds of space-filling curves for three kinds of query shapes on three-dimensional grid spaces. The result is for  $2^{10} \times 2^{10} \times 2^{10}$  grid.

	Hilbert over Gray-coded	Gray-coded over Z
Cube	1.4	1.42
Sphere	1.2	1.3
Hyper-rectangle	1.3	1.3

So on inter-cluster properties in both two-dimensional and three-dimensional grid spaces, Z-curves perform the best, the Hilbert curves are the worst, and the Gray-coded curves in between.

Let  $N$  denote the number of all possible  $d$ -dimensional query regions with side length  $L$ . Suppose for the  $i$ th query, the query indexing range is  $R_i$ , and the number of clusters is

$C_i$ , in which  $i \in \{1, 2, \dots, N\}$ . Then the average inter-cluster distance is  $\frac{\sum_{i=1}^N (R_i - L^d)}{\sum_{i=1}^N (C_i - 1)}$ , in

which  $L^d$  is the number of grid point within the query region, and  $C_i - 1$  is the number of inter-cluster segments for the  $i$ th query region. Compared with query range  $R_i$ ,  $L^d$  is so small for large grid space, and the average inter-cluster distance is approximated by

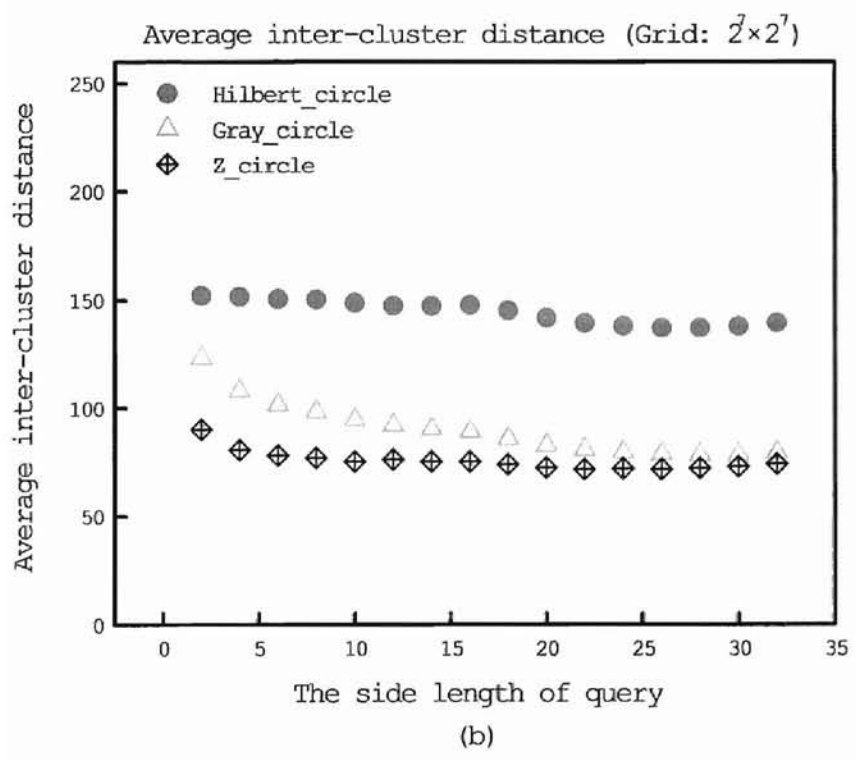
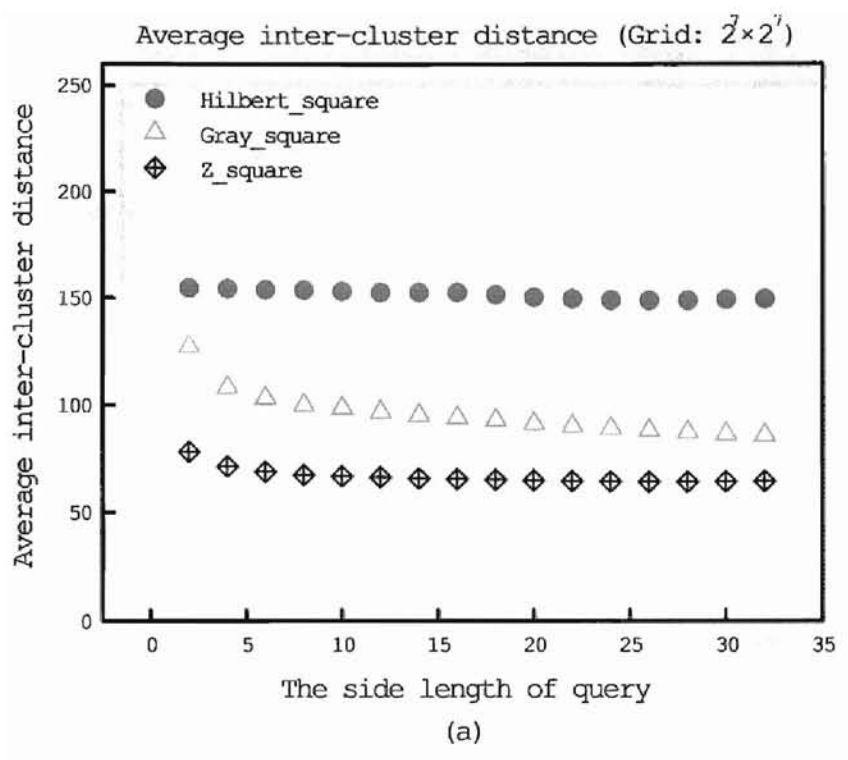
$$\frac{\sum_{i=1}^N R_i}{\sum_{i=1}^N C_i} = \frac{N \bar{R}}{N \bar{C}} = \bar{R} / \bar{C},$$

in which  $\bar{R}$  and  $\bar{C}$  are the average query indexing range and average number of clusters respectively.

In the earlier section we've noticed that  $\bar{R}$  is approximately a linear function of the side length of the query region from the simulation runs. For the two-dimensional case,



$\bar{C}$  is a linear function of side length of the query region [MJF01]; the average inter-cluster distance approaches constant in two-dimensional grid spaces. In three-dimensional grid spaces,  $\bar{C}$  is a quadratic function of side length of the query region [MJF01]; the inter-cluster distance decreases with the growing side length of the query region.



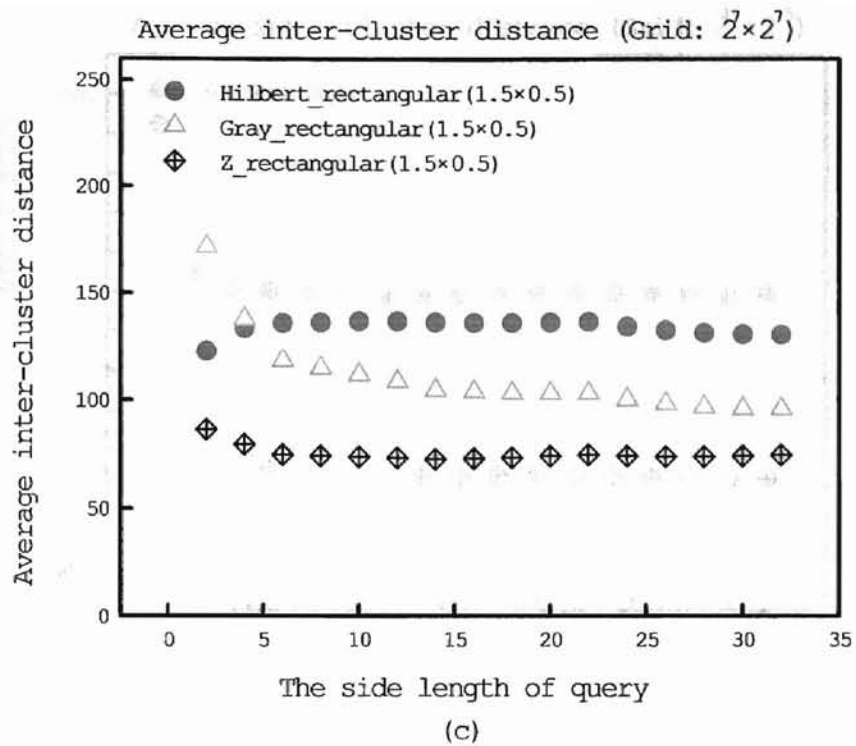
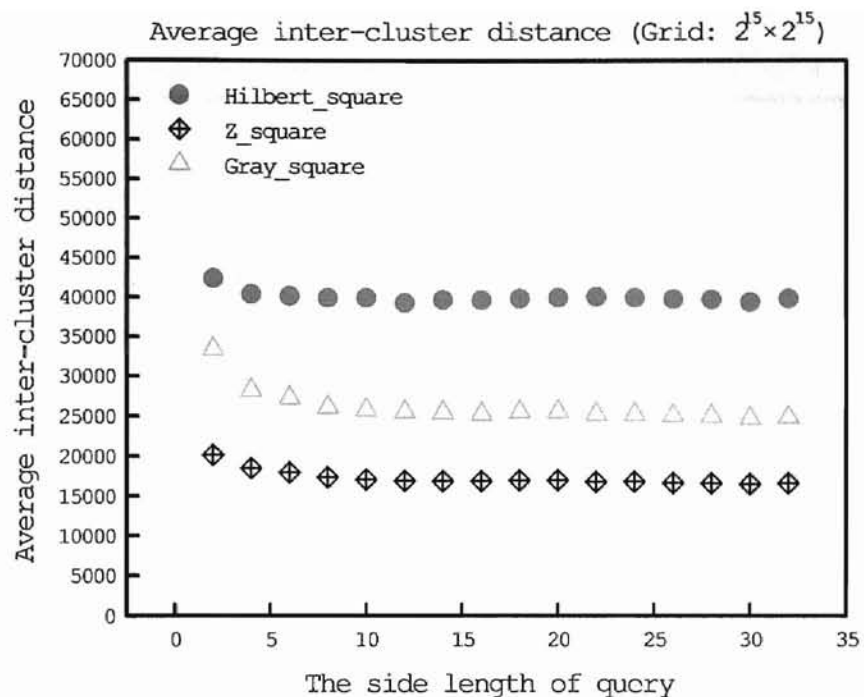
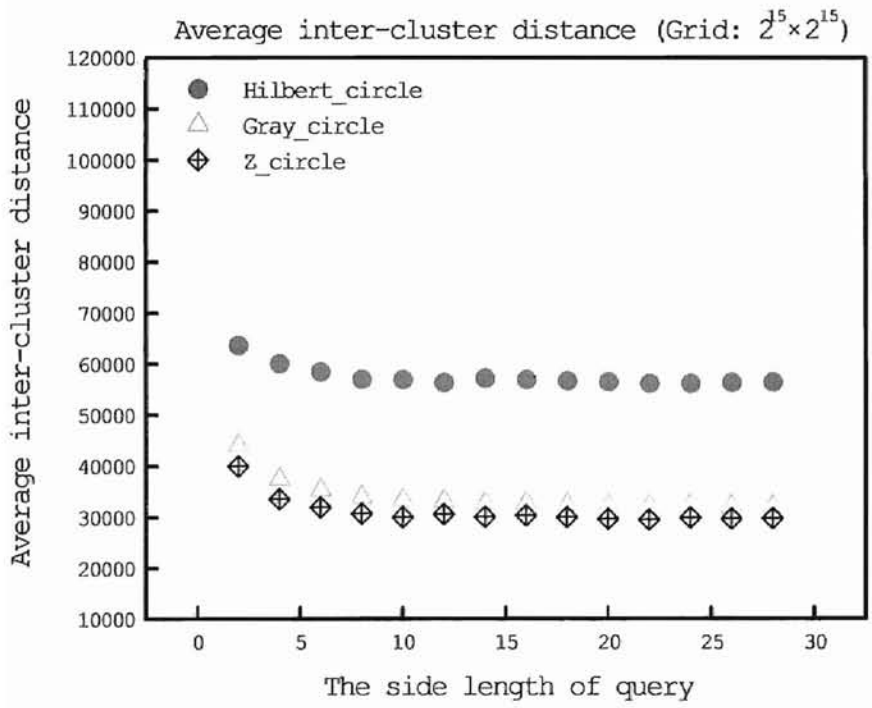


Figure 4-9. Average inter-cluster distance for two-dimensional queries (exhaustive simulation on  $2^7 \times 2^7$  grid): (a) square queries, (b) circular queries, and (c) rectangular queries.



(a)



(b)

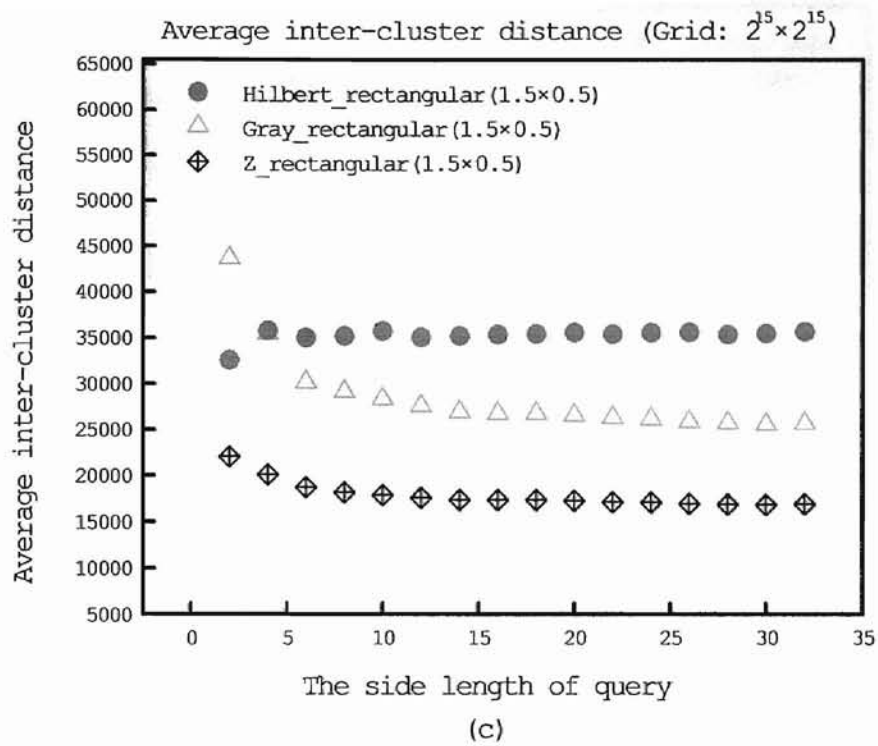
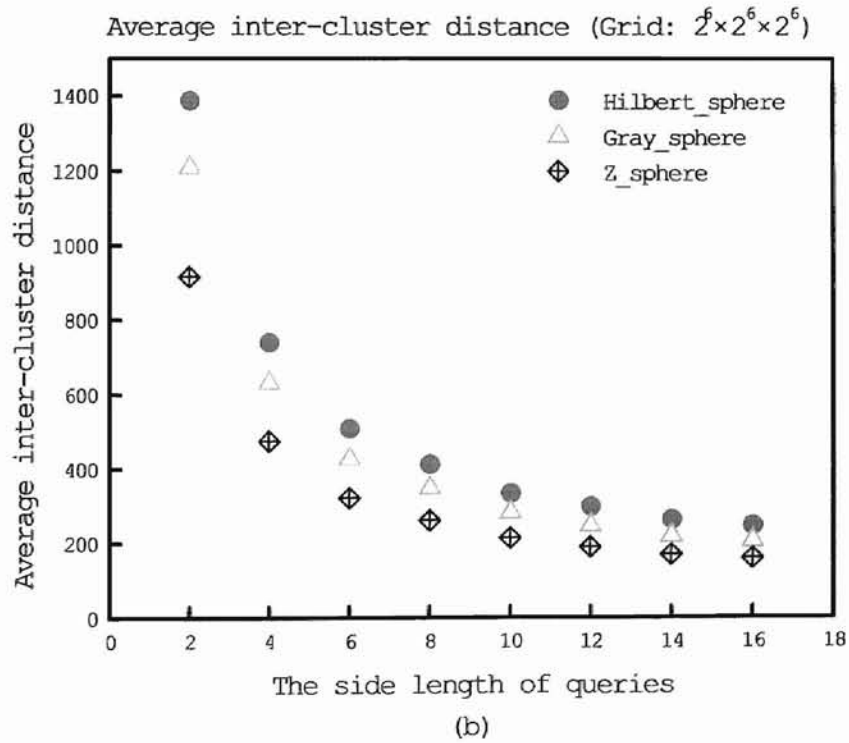
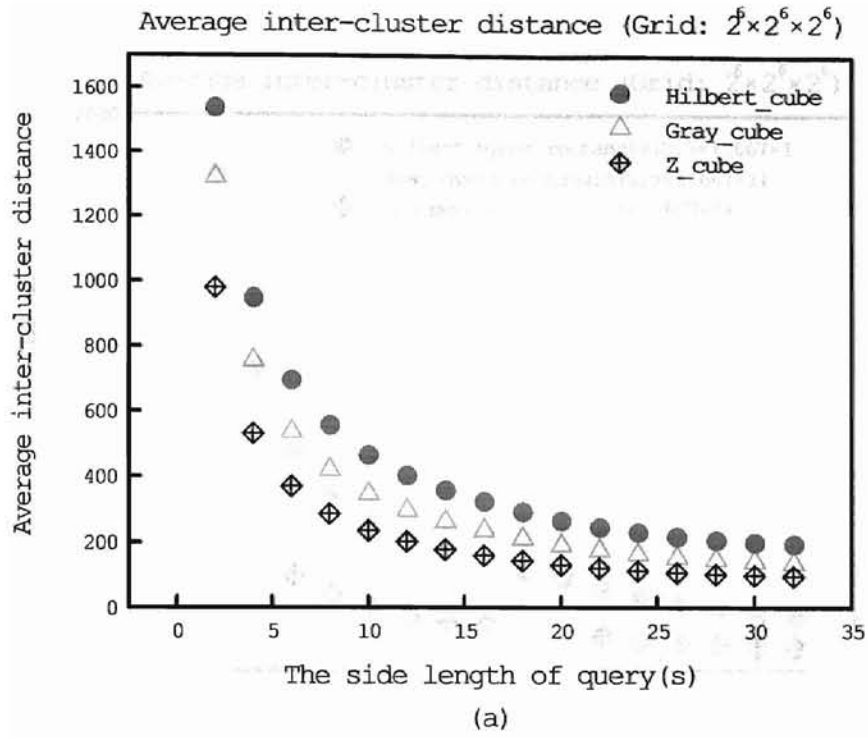


Figure 4-10. Average inter-cluster distance for two-dimensional queries (sampling simulation on  $2^{15} \times 2^{15}$  grid): (a) square queries, (b) circular queries, and (c) rectangular queries.

Iranian Journal of Statistics and Probability



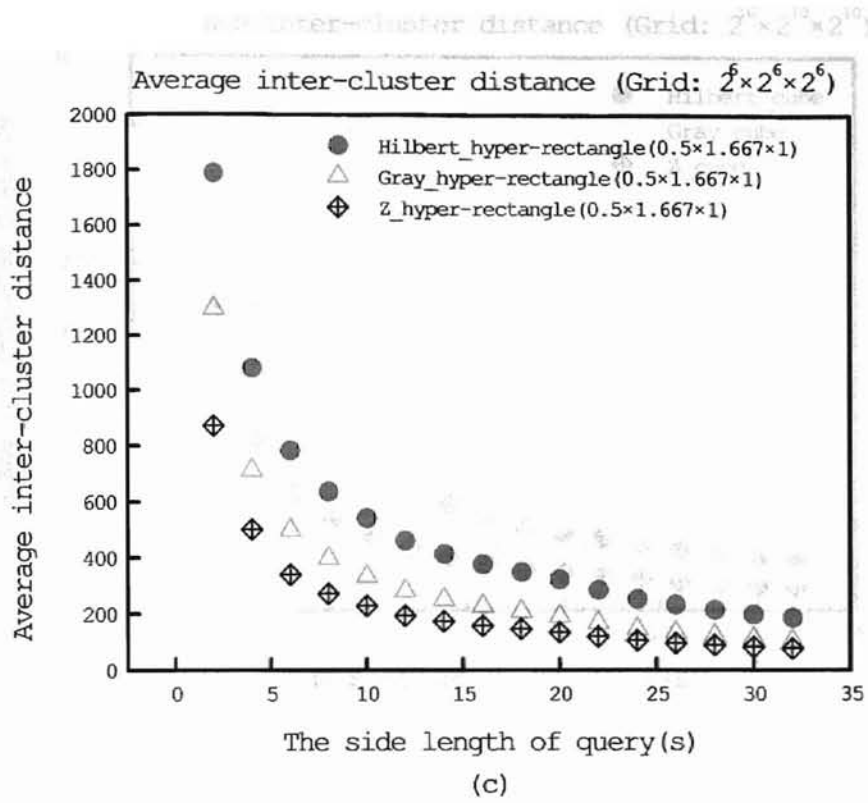
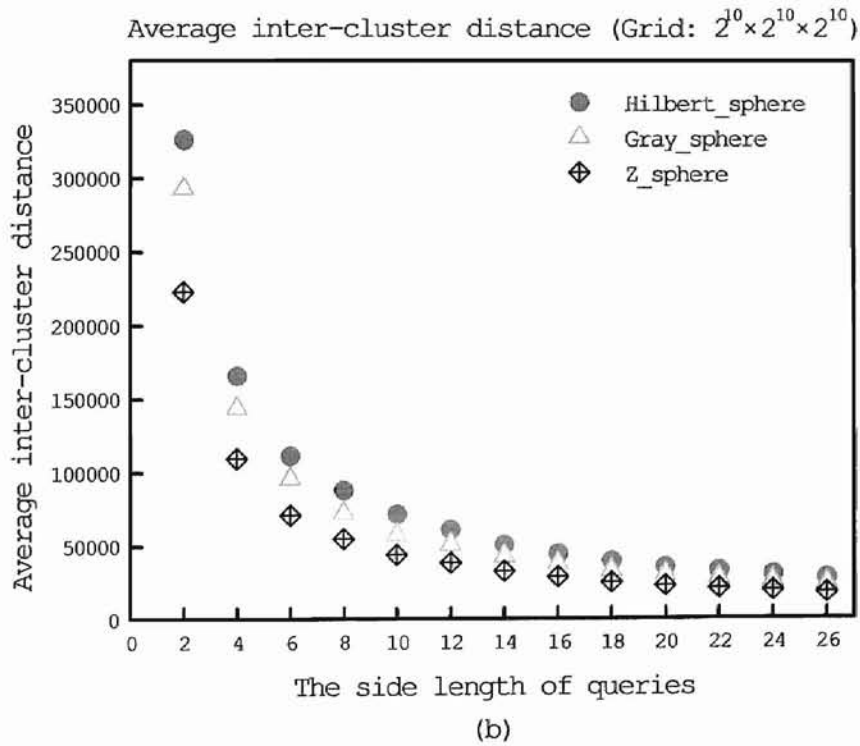
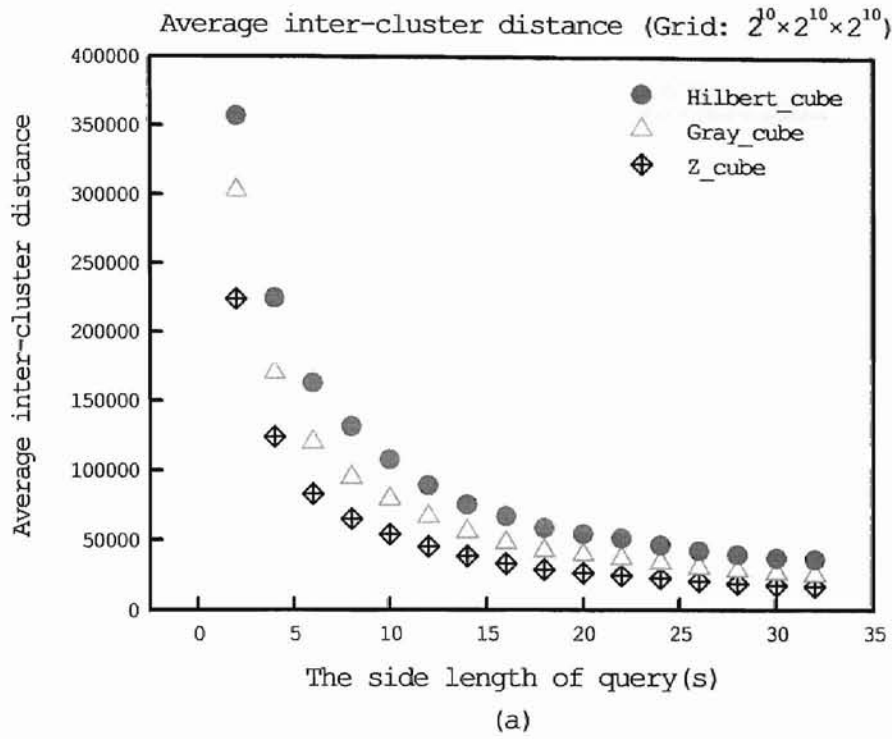


Figure 4-11. Average inter-cluster distance for three-dimensional queries (exhaustive simulation on  $2^6 \times 2^6 \times 2^6$  grid): (a) cubic queries, (b) spherical queries, and (c) hyper-rectangular queries.





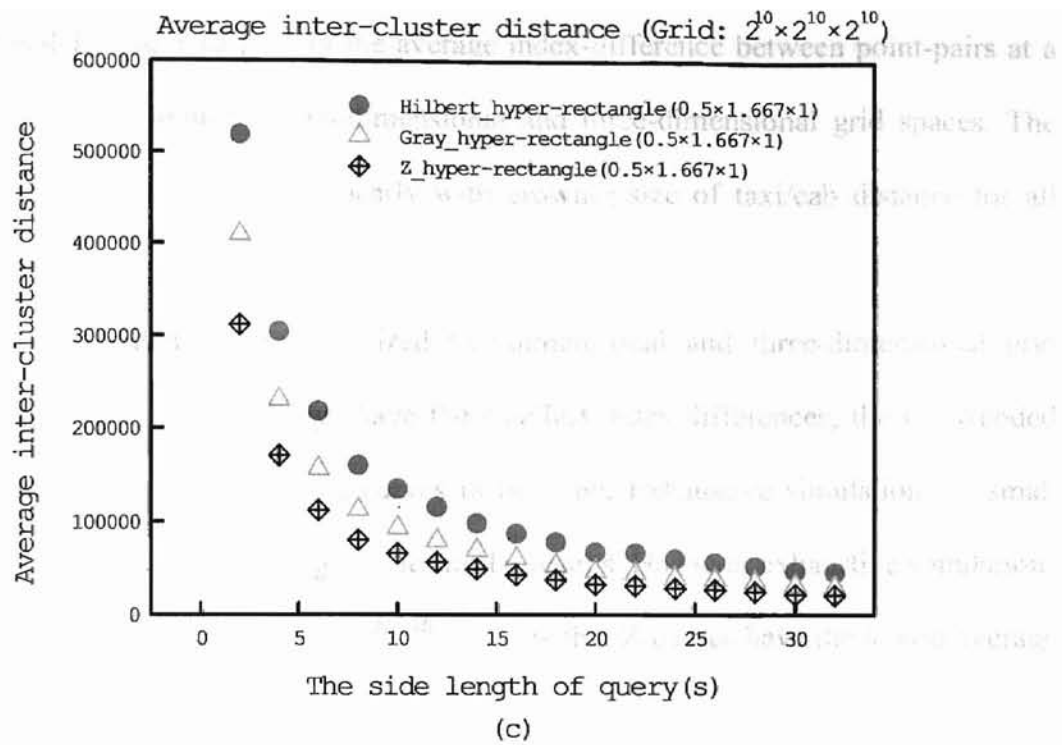


Figure 4-12. Average inter-cluster distance for three-dimensional queries (sampling simulation on  $2^{10} \times 2^{10} \times 2^{10}$  grid): (a) cubic queries, (b) spherical queries, and (c) hyper-rectangular queries.

Figures 4-13 and 4-15 present the average index-difference between point-pairs at a common taxi/cab distance in two-dimensional and three-dimensional grid spaces. The average index-difference grows linearly with growing size of taxi/cab distance for all three families of space-filling curves.

Sampling simulations in large-sized two-dimensional and three-dimensional grid spaces show that Z-curves always have the smallest index-differences, the Gray-coded curves the largest, and the Hilbert curves in between. Exhaustive simulations in small size two-dimensional grid  $2^7 \times 2^7$  generate similar results. However exhaustive simulations in small size three-dimensional grid  $2^6 \times 2^6 \times 2^6$  show that Z-curves have the lowest average index-difference at small taxi/cab distance, and as the taxi/cab distance grows, Z-curves exceed both the Gray-coded curves and the Hilbert curves. Over the considered range of taxi/cab distance, the Gray-coded curves always have larger average index-difference than the Hilbert curves.

Figures 4-14 and 4-16 present the ratios of the index-differences in two-dimensional and three-dimensional grid spaces. Each figure consists of two curves, one is that between the Gray-coded curves and the Hilbert curves, and the other is that between the Hilbert curves and Z-curves.

Figure 4-14(b) shows that in larger-sized  $2^{15} \times 2^{15}$  grid, the ratio of the Hilbert curves versus Z-curves fits quite well with 1.21. This result complies with the analytical result of Dai and Su in [DS02], in which the ratio approaches 1.21 as the order of space-filling curves grows to infinity. However Figure 4-14(a) shows in smaller-sized grid  $2^7 \times 2^7$ , this result only holds for small taxi/cab distances.

Figure 4-14(b) shows that the ratio of the Gray-coded curves versus the Hilbert curves is about 1.21 in larger-sized grid  $2^{15} \times 2^{15}$ . As switched to the smaller-sized grid  $2^7 \times 2^7$  in Figure 4-14(a), the value decreases slightly over the range of large taxi/cab distance. However the change is not so large compared with ratio between the Hilbert curves and Z-curves.

Figure 4-16(b) shows that in larger-sized grid  $2^{10} \times 2^{10} \times 2^{10}$ , the ratio of the Hilbert curves versus Z-curves keeps constant at approximately 1.08 over the considered range of taxi/cab distance. This result again complies with Dai and Su's result in [DS02], where they give the asymptotic ratio only for taxi/cab distance  $\delta = 1$  in three-dimensional grid spaces as the order of space-filling curve goes to infinity. While in our simulation, we found that this ratio also holds for all  $\delta \geq 1$ . Again Figure 4-16(a) shows that in smaller-sized grid  $2^6 \times 2^6 \times 2^6$ , the result is valid only over the range of small taxi/cab distance.

Figure 4-16(b) also shows that the ratio of the Gray-coded curves versus the Hilbert curves is about 1.08 in larger-sized grid  $2^{10} \times 2^{10} \times 2^{10}$ . In smaller-sized grid  $2^6 \times 2^6 \times 2^6$  as figure 4-16(a) shows, the value decreases slightly over the range of large taxi/cab distance. Still just like the two-dimensional case, the change is not so large compared with ratio between the Hilbert curves and Z-curves.

We can conclude that based on locality measure which is the index-difference between point-pairs at a common taxi/cab distance, Z-curves perform better than the Hilbert curves over the considered ranges of dimension, grid-order and taxi/cab distance, and the Hilbert curves outperform the Gray-coded curves. As the dimensionality grows, the differences between the index-difference of three families of space-filling curves diminish.

Figure 4-13. Index-difference versus taxi/cab distance on two-dimensional grid (a) exhaustive simulation on  $2^7 \times 2^7$  grid and (b) sampling simulation on  $2^{15} \times 2^{15}$  grid.

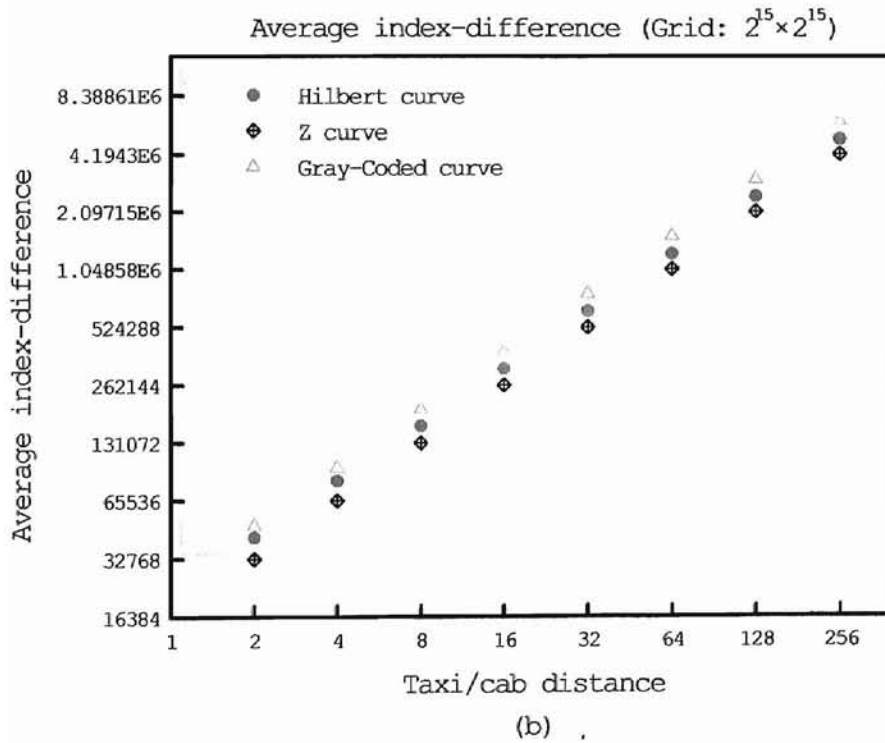
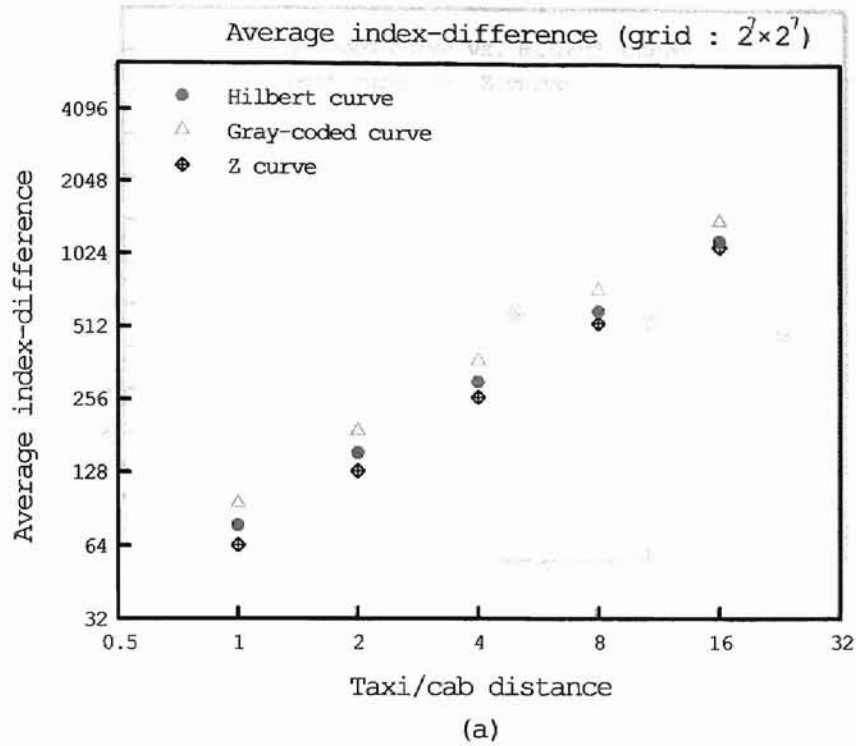
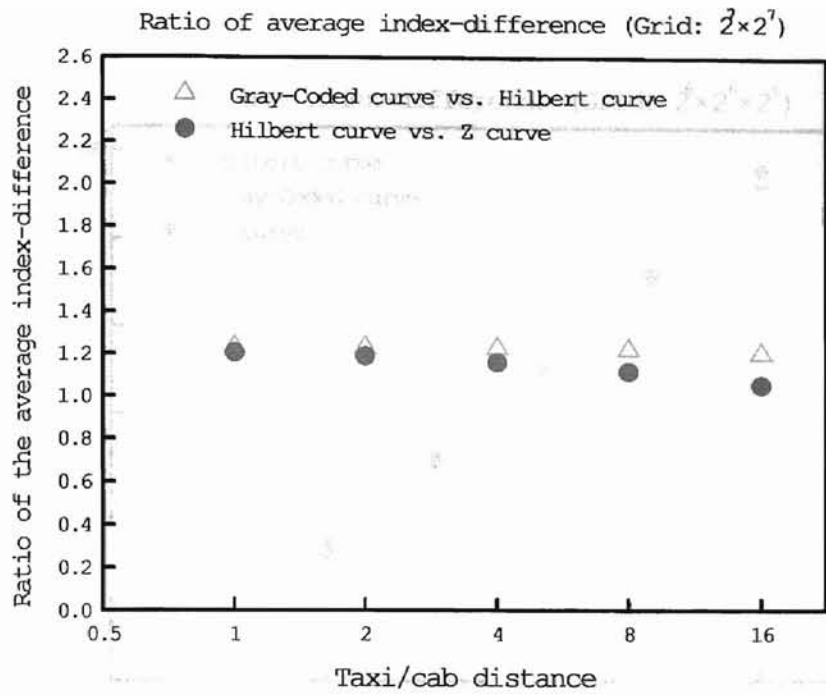
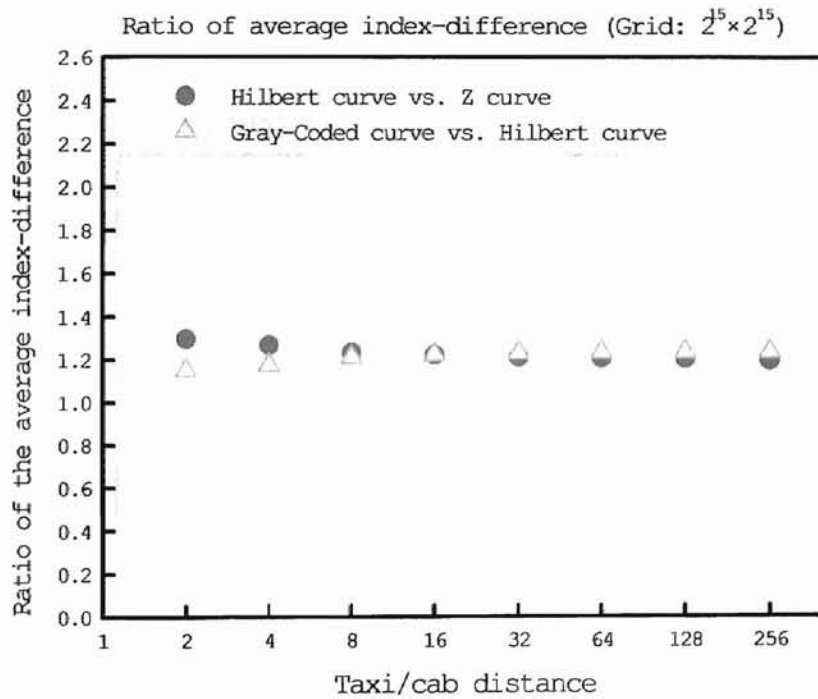


Figure 4-13. Index-difference versus taxi/cab distance on two-dimensional grid (a) exhaustive simulation on  $2^7 \times 2^7$  grid and (b) sampling simulation on  $2^{15} \times 2^{15}$  grid.



(a)



(b)

Figure 4-14. Ratio of the index-difference versus taxi/cab distance on two-dimensional grid (a) exhaustive simulation on  $2^7 \times 2^7$  grid and (b) sampling simulation on  $2^{15} \times 2^{15}$  grid.

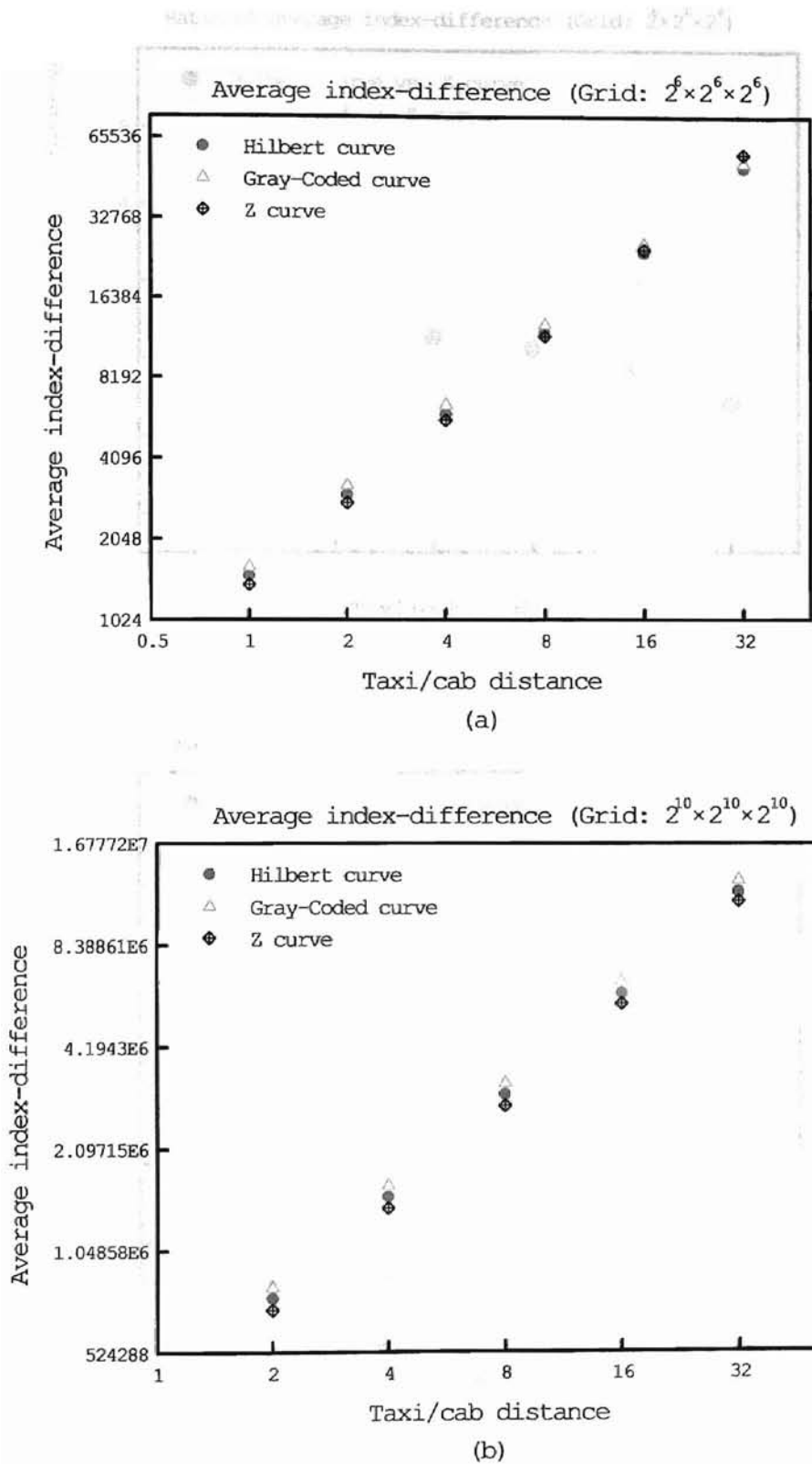


Figure 4-15. Index-difference versus taxi/cab distance on three-dimensional grid (a) exhaustive simulation on  $2^6 \times 2^6 \times 2^6$  grid and (b) sampling simulation on  $2^{10} \times 2^{10} \times 2^{10}$  grid.

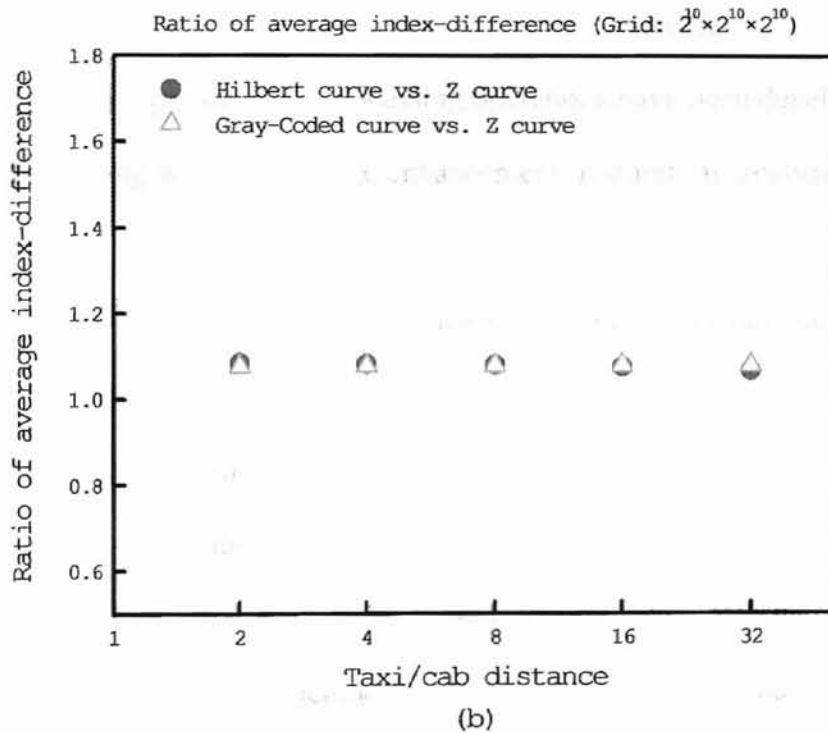
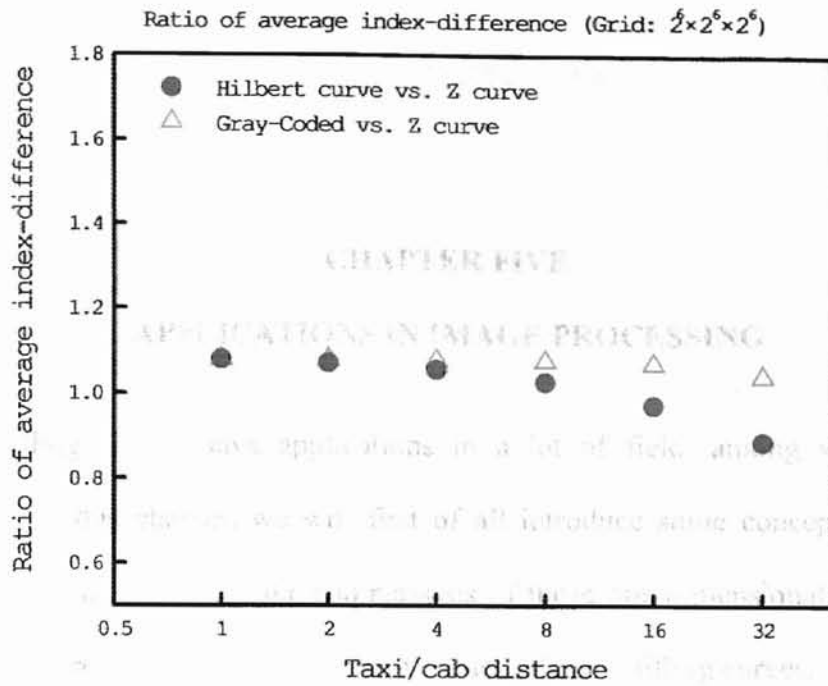


Figure 4-16. Ratio of the index-difference versus taxi/cab distance on three-dimensional grid (a) exhaustive simulation on  $2^6 \times 2^6 \times 2^6$  grid and (b) sampling simulation on  $2^{10} \times 2^{10} \times 2^{10}$  grid.

University of Illinois at Urbana-Champaign

## **CHAPTER FIVE**

### **APPLICATIONS IN IMAGE PROCESSING**

Space-filling curves have applications in a lot of field, among which is image processing.. In this chapter, we will first of all introduce some concepts about image processing, and then try to make comparisons of those one-dimensional representations of images that are scanned long the three families of space-filling curves.

Image processing has many interesting applications in medicine, biological research, photography, space exploration, astronomy, and law enforcement forensics. Numerous and widely varying types of image processing operations have been developed during the past 20 years, among which are image enhancement, restoration, analysis, compression, and synthesis. With the advent of the Internet and the WWW, image compression is now more important than ever. To achieve acceptable throughout over the Internet, images are usually compressed to minimize their size.

There are a host of known applications for image compression techniques and systems, and new applications are being proposed each day, creating a demand for modifications of existing compression methods or entirely new approaches. Existing applications of compression methods include facsimile, personal communications systems, still-image archival, videoconferencing, and video and movie distribution.

The goal of image compression is the reduction of the amount of data required to represent a digital image. The idea of image compression is to remove redundant data



from the images (i.e., data which do not affect image quality significantly). Image compression is very important for image storage and image transmission.

Some applications of image storage are: educational and business documents, medical images, weather maps and fingerprints. Some applications of image transmission are: remote sensing via satellite, military communications via aircraft, radar, and sonar, teleconferencing, and facsimile transmission.

Image compression techniques are divided into two general groups: loseless and lossy. Loseless image compression preserves the exact data content of the original image. Lossy image compression preserves some specified level of image quality, but not the absolute data content of the original image.

Loseless coding techniques provide for exact recovery of the original image. Huffman coding, arithmetic decomposition, Lempel Ziv algorithms, and run length coding are all loseless coding techniques. Lossy coding techniques include predictive coding, frequency-oriented coding, importance-oriented coding, and hybrid coding.

Usually loseless coding techniques yield lower compression ratio, while lossy coding techniques have higher compression ratio. So there is a tradeoff between image quality and compression ratio. By compression ratio, we mean quotient of the number of units used to represent the original image and number of units used to represent the compressed image.

Before applying the image compression encoding, the two-dimensional image data should first be scanned into a bit sequence, or a bit string using some scanning function. For simplicity, here we only consider the binary image in which 0 stands for white pixels, and 1 for black pixels.

www.civildatas.com

We assume an image to be a  $N \times N$  grid of pixels. The pixel at point  $(i, j)$  in the image has intensity value  $P_{ij}$ . Definition: A scan of the image is a bijection  $f$  from the closed interval of integers  $[1, \dots, N^2]$  to the set of ordered pairs  $\{(i, j): 1 \leq i, j \leq N\}$ , where the latter set denotes the points in the image. Equivalently, the image is encoded using the scan  $f$  by encoding the pixel intensities in the order  $P_{f(1)}, P_{f(2)}, \dots, P_{f(N^2)}$ .

The level of compression using run length encoding is entirely dependent upon the repetitive nature of the source image. An image with large runs of the same pixel intensity will compress well. Images without this property will compress poorly. However for the same image, different scanning functions yields different compression ratios.

There is a lot of scanning methods applied in practice. Some of them are row-by-row scan, column-by-column scan, zig-zag scan. Besides these ordinary scanning methods, space-filling curves are also very popular scanning methods. If the image does not have long runs, better compression may be produced by a method that scans the bitmap area by area instead of row-by-row scanning or column-by-column scanning. A space-filling curve completely fills up a part of space by passing through every point in that part.

To investigate how the three space-filling curves we have interested in this thesis will behave in image compression, we applied the image scan on an image of deer first, and then on a  $2^{10} \times 2^{10}$  image filled with 200 randomly placed squares of size  $8 \times 8$ .

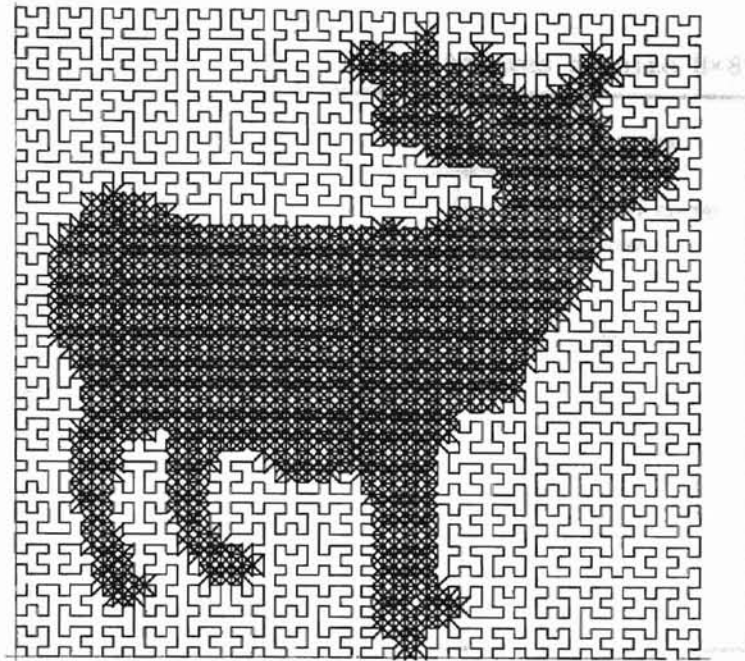
In Figure 5-1(a) we presented the image of deer [DCM00], and in Figure 5-1(b) the auto-correlation function based on scanned images which are scanned along three families of space-filling curves and also row-by-row scanning. We can see that Hilbert space-filling curves have the highest auto-correlation over the whole range of index

difference, while Z-curves and the Gray-coded curves perform better than row-by-row scanning at relatively large index difference.

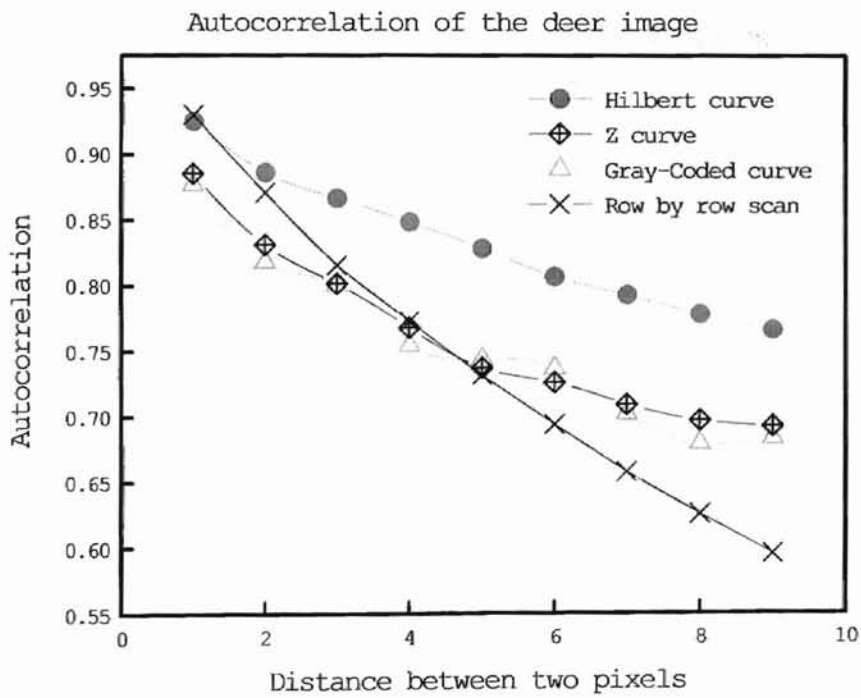
Still we also want to compare all these space-filling curves when image of small objects is scanned. In Figure 5-2, we presented the auto-correlation results obtained through one-dimensional images scanned along three families of space-filling curves and row-by-row scanning. We can see from the figure that space-filling curves produce really good auto-correlation even when the auto-correlation reaches almost zero under row-by-row scanning.

It is very clear that space-filling curves are really good at scanning images with scattered small objects. Among these three families of space-filling curves, Hilbert space-filling curves are superior to the other two. Z-curves and the Gray-coded curves have almost the same behavior when applied on image scanning. This is easy to understand if we can recall that Hilbert space-filling curves have the best clustering properties, while Z-curves and the Gray-coded curves have almost the same clustering properties in two-dimensional cases as shown in Figure 4-1 and 4-2.

Finally we can conclude that when applied on image compression, Hilbert space-filling curves is superior to Z-curves and the Gray-coded curves, and Z-curves and the Gray-coded curves behave almost the same.



(a)



(b)

Figure 5-1. (a) A deer image applied Hilbert space-filling curve scanning, (b) Auto-correlation of the three scanned images under three families of space-filling curves and row-by-row scanning.

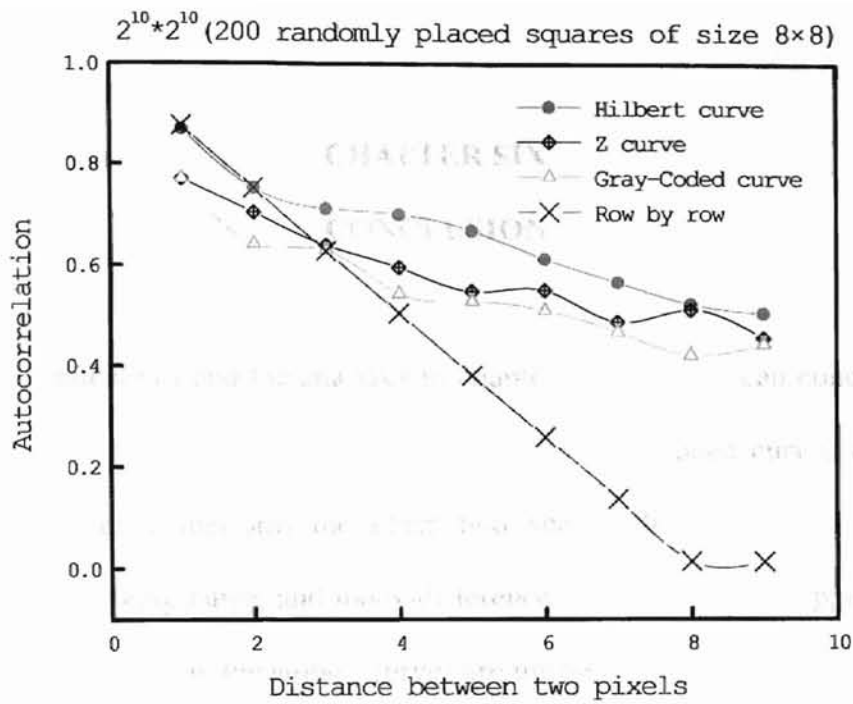


Figure 5-2. Auto-correlation of the scanned images under three families of space-filling curves and row-by-row scanning, in which the image is 200 randomly placed squares of size  $8 \times 8$  on  $2^{10} \times 2^{10}$  grid.

It is the best clustering method in that field, and also we should consider the space-filling Hilbert metric.

## CHAPTER SIX

### CONCLUSION

From the simulations and the analyses in chapter 4 and 5, We can conclude that the Hilbert curves are superior to Z-curves and also the Gray-coded curves on clustering properties, Z-curves outperform the other two space-filling curves on inter-cluster distance, query indexing range, and index-difference properties. When applied on image scan in image compression, the Hilbert curves are the best.

Even though it seems that the Hilbert space-filling curves are the best on clustering locality measure, Gotsman and Lindenbaum [GL96] posed the question as to “whether there exists families of space-filling curves with locality properties better than those of the Hilbert curves for all sizes”. In their work [GL96], classic Hilbert space-filling curves come close to achieving optimal locality based on locality metric of  $L_{GL,max}(C)$ . The answer from R. Niedermeier et al.’s work [NRS97] is affirmative. In their work, they proposed a new two-dimensional indexing scheme what they called *H-indexing*, which is related to two-dimensional Sierpinski curve [Sag94]. They proved that H-indexings are optimally locality-preserving with respect to the Euclidean metric, maximum metric, and Manhattan or taxi/cab metric.

Here we can say that no matter which space-filling curve, it can’t be always superior to all the other space-filling curves on all those locality property measures. To chose which space-filling curve in a specific application filed depends on the locality measure

which is the most important to that field, and also we should consider the space-filling curve that performs the best in that locality metric.

## REFERENCES

1. Comparative Analysis of Space-Filling Curves, *Journal of Applied Probability*, vol. 2, pp. 11, 1966.

2. Space-Filling Curves and Their Application to Data Structures, *Journal of Applied Probability*, vol. 2, pp. 11, 1966.

3. H. S. H. H.

4. H. S. H. H.

5. H. S. H. H.

6. H. S. H. H.

7. H. S. H. H.

8. H. S. H. H.

9. H. S. H. H.

10. H. S. H. H.

## REFERENCES

- [AM90] D.J. Abel and D.M. Mark, "A Comparative Analysis of Some Two-Dimensional Orderings," *Int'l Journal of Geographical Information Systems*, 4(1): 21-31, 1990.
- [Bia69] T. Bially, "Space-Filling Curves: Their Generation and Their Application to Bandwidth Reduction," *IEEE Transactions on Information Theory*, 15(6): 658-664, 1969.
- [BP82] J. J. Bartholdi and L.K. Platsman, "An  $O(n \lg n)$  Travelling Salesman Heuristic Based on Spacefilling Curves," *Operation Research Letters*, 1(4): 121-125, 1982.
- [But69] A.R. Butz, "Convergence with Hilbert's Space Filling Curve," *Journal of Computer and System Sciences*, 3: 128-146, 1969.
- [DCM00] Revital Dafner, Daniel Cohen-Or and Yossi Matias, "Context-based Space Filling Curves," *EUROGRAPHICS*, 19:1-9, 2000.
- [DS02] H.K. Dai and H.C. Su, "On the Locality Properties of Space-Filling Curves," draft, 2002.
- [Duf84] I.S. Duff, "Design Features of a Frontal Code for Solving Sparse Unsymmetric Linear Systems Out-of-Core," *SIAM Journal of Scientific Statistical Computing*, 5(2): 270-280, 1984.
- [Fal86] C. Faloutsos, "Multiattribute Hashing Using Gray Codes," *Proceedings of 1986 ACM SIGMOD Conference*, pages 227-238, May 1986.
- [FS89] C. Faloutsos and S. Roseman. "Fractals for Secondary Key Retrieval," *Proceeding of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 247-252, 1989.
- [GG98] V. Gaede and O. Günther. "Multi-dimensional Access Methods," *ACM Computing Surveys*, 30(2): 170-231, June 1998.
- [GL96] C. Gotsman and M. Lindenbaum. "On the Metric Properties of Discrete Space-Filling Curves," *IEEE Transactions on Image Processing*, 5(5), 1996.
- [Hil1891] D. Hilbert. "Über stetige Abbildung einer Linie auf ein Flächenstück," *Mathematische Annalen*, 38: 459-460, 1891



- [Jag90] H.V. Jagadish, "Linear Clustering of Objects with Multiple Attributes," *Proceedings of ACM SIMOD Conference*, pages 332-342, May 1990.
- [Jag97] H.V. Jagadish, "Analysis of the Hilbert Curves for Representing Two-dimensional Space," *Information Processing Letters*, 62(1): 17-22, 1997.
- [KOR95] M. Kaddoura, C.W. Ou, and S. Ranka, "Partitioning Unstructured Computational Graphs for Nonuniform and Adaptive Environments," *IEEE Parallel and Distributed Technology*, 3(3): 63-69, 1995.
- [LZ84] A. Lempel and J. Ziv, "Compression of Two-Dimensional Images," *NATO ASI Series. F12*: 141-154, 1984.
- [Lau85] R. Laurini. "Graphical Databases Built on Peano Space-Filling curves," In *Proceedings of Eurographics '85*, pages 327-338. North-Holland, 1985.
- [LZ86] A. Lempel and J. Ziv, "Compression of Two-Dimensional Data," *IEEE Transactions on Information Theory*, 32(1): 2-8, 1986.
- [MD86] G. Mitchison and R. Durbin. "Optimal Numberings of an  $N \times N$  Array," *SIAM Journal on Algebraic and Discrete Methods*, 7(4): 571-582, 1986.
- [MJF01] B. Moon, H.V. Jagadish, C. Faloutsos. "Analysis of the Clustering Properties of the Hilbert Space-Filling Curve," *IEEE Transactions on Knowledge and Data Engineering*, 13(1): 124-141, 2001.
- [NRS97] R. Niedermeier, K. Reinhardt, and P. Sanders. "Towards Optimal Locality in Mesh-Indexings," *Lecture Notes in Computer Science* 1279, 1997.
- [Ore86] J. Orenstein, "Spatial Query processing in an Object-Oriented Database System," *Proceedings of 1986 ACM SIGMOD Conference*, pages 326-336, May 1986.
- [PAR68] E.A. Patrick, D.R. Anderson and F.K. Bechtel, "Mapping Multi-Dimensional Space to One Dimension for Computer Output Display," *IEEE Transactions on Computers*, 17(19): 949-953, 1968.
- [Pea1890] G. Peano, "Sur une Courbe qui Remplit Toute une Aire Plane," *Math. Annual* 36:157-160, 1890.
- [PB89] L.K. Platsman and J. J. Bartholdi, "Spacefilling Curves and the Planar Travelling Saleman Problem," *Journal of the Association for Computing Machinery*, 36(4): 719-737, 1989.
- [PKK92] A. Perez, S. Kamata, and E. Kawaguchi. "Peano Scanning of Arbitrary Size Images," *Proceedings of the International Conference on Pattern Recognition*, pages 565-568. IEEE Computer Society, 1992.

- [RF91] Y. Rong and C. Faloutsos, "Analysis of the Clustering Property of Peano Curves," Technical Report CS-TR-2792, UMIACS-TR-91-151, University of Maryland, Dec. 1991.
- [Riv76] R.L. Rivest, "Partial Match Retrieval Algorithms," *SIAM Journal on Computing*, 5(1): 19-50, 1976.
- [Sag93] H. Sagan, "A Three-Dimensional Hilbert Curve," *International Journal of Math. Ed. Science Technology*, 24: 541-545, 1993.
- [Sag94] H. Sagan. *Space-Filling Curves*. Springer Verlag, 1994.
- [Sim63] G.F. Simmons, *Introduction to Topology and Modern Analysis*. New York: McGraw-Hill Book Company, Inc., 1963.
- [ZW93] Y. Zhang and R.E. Webber, "Space Diffusion: An Improved Parallel Halftoning Technique Using Space-Filling Curves," *Computer Graphics Proceedings, Annual Conference Series*, pages 305-312. ACM, 1993.

## APPENDIX

Source code of simulation <thesis.cpp>

```
/*
 * Author      : Hua LI
 * Date       : Oct., 2001 - April 2002
 * Purpose    : This algorithm is used to perform simulation on locality*
 *              preserving properties of three kinds of space-filling *
 *              curves, Hilbert curves, Z-curves, and Gray-coded curves.*
 ****
 * Function   : zOrder_c2i( int nDims, int nBits, bitmask_t coord[]) *
 * Purpose    : Mapping coordinates (x,y,z) into the index under Z-curve.*
 ****
 * Function   : gray_c2i( int nDims, int nBits, bitmask_t coord[] ) *
 * Purpose    : Mapping coordinates (x,y,z) into the index under *
 *              Gray-coded curve.
 ****
 * Function   :
 * numCluster( int x,int y,int z,int querySize,double &maxLength ) *
 * Purpose    : Given the querySize and also the randomly chosen locate *
 *              (x,y,z), return the number of clusters, and also collect *
 *              the number of inter-cluster distance.
 ****
 * Function   : interCluster()
 * Purpose    : This function will collect the average number of cluster,*
 *              the inter-cluster distance, and the average query range. *
 ****
 * Function   : random( int& izeed )
 * Purpose    : With input izeed, return a random value between 0 and 1. *
 ****
 * Function   : indexDiff( int taxiDistance )
 * Purpose    : With input taxiDistance, give an average index-difference*
 ****
 */
```

```
#include "hilbert.h"
#include <iostream>
#include <string>
#include <iomanip>
#include <math.h>
#include <fstream>
#include <stack>
#include <utility>
#include <vector>
#include <list>
#include <numeric>
#include <algorithm>
using namespace std;
```

```
/*
 * G - Gray-coded curve
 */
```

```

* H - Hilbert curve
* Z - Z-curve
*****/lengthSample; i ++ )
const char curve_flag = 'G';
//output file name for index-difference
const char * distribution = "dis_3d_gray_10";
ofstream disfile(distribution);

//output file name for inter-cluster metrics
//const char *statFile = "stat_3d_poly_gray_10";
//ofstream statOut(statFile);

//constants
const int nDims = 3; //dimensionality
const int nBits = 10; // k in 2^(k)
const int length = pow(2, nBits); // the grid size
// the number of sampling for inter-cluster distance
const int numSample = 100000;
// the number of sampling for index-difference
const int lengthSample = 20000000;

//global variants
double nInterDis = 0;
double totInterDis = 0;

//functions and subroutines
int zOrder_c2i( int nDims, int nBits, void const* coord1 );
int gray_c2i( int nDims, int nBits, void const* coord1 );
int numCluster( int x, int y, int z, int querySize, double
&maxLength );
void interCluster();
float random( int& izeed );
void indexDiff( int taxiDistance );

int main()
{
    for( int i = 2; i <= 256; i *= 2 )
        indexDiff(i);

    return 0;
}

/*****
* Function : indexDiff( int taxiDistance )
* Purpose  : With input taxiDistance, give an average index-difference.
*****/
void indexDiff( int taxiDistance )
{
    static int xseed = 2834;
    static int yseed = 28479;
    static int zseed = 1484;

    double totGrid = 0;
    double testLength = 0;

```

```

double testTot = 0; testTot << endl;

for( int i = 0; i < lengthSample; i ++ )
{
    //coordinates of the beginning point
    int x1 = random(xseed) * length;
    int y1 = random(yseed) * length;
    int z1 = random(zseed) * length;

    for( int dx = 0; dx <= taxiDistance; dx++)
    for( int dy = 0; dy <= taxiDistance - dx; dy++)
    {
        //taxi distance
        int dz = taxiDistance - dx - dy;

        //coordinates of the ending point
        int x2 = x1 + dx;
        int y2 = y1 + dy;
        int z2 = z1 + dz;

        if( x2 < length && y2 < length && z2 < length)
        {
            double linearDis;
            unsigned long c1[3] = {x1,y1,z1};
            unsigned long c2[3] = {x2,y2,z2};

            switch(curve_flag)
            {
                case 'H':
                    linearDis =
                        abs(hilbert_c2i(3,nBits,c1)
                            -hilbert_c2i(3,nBits,c2));
                    break;
                case 'Z':
                    linearDis =
                        abs(zOrder_c2i(nDims,nBits,c1) -
                            zOrder_c2i(nDims,nBits,c2));
                    break;
                case 'G':
                    linearDis =
                        abs(gray_c2i(nDims,nBits,c1) -
                            gray_c2i(nDims,nBits,c2));
                    break;
            }

            testLength += linearDis;
            testTot ++;
        }
    }
    if( i % 100000 == 0)
        cout << taxiDistance << " " << i << endl;
}

//output the results
disfile << taxiDistance << " ";
disfile << testLength/testTot << endl;
cout << taxiDistance << " ";

```

```

        cout << testLength/testTot << endl;
    }

/*****
* Function : interCluster()
* Purpose  : This function will collect the average number of cluster,
*           the inter-cluster distance, and the average query range.
*****/
void interCluster()
{
    static int xseed = 876749;
    static int yseed = 1343;
    static int zseed = 34589;

    for( int querySize = 2; querySize <= 32; querySize +=2 )
    {
        nInterDis = 0;
        totInterDis = 0;
        double totCluster = 0;
        double maxRange = 0;
        int maxCluster = 0;

        for( int i = 0; i < numSample; i++ )
        {
            double maxLength = 0;

            //Cube queries
            int x = random(xseed) * (length - querySize);
            int y = random(yseed) * (length - querySize);
            int z = random(zseed) * (length - querySize);

            //Polyhedra queries
            int x = random(xseed) * (length - querySize);
            int y = random(yseed) * (length - querySize);
            int z = random(yseed) * (length - 2*querySize);

            //sphere queries
            int x = random(xseed) * (length - 2*querySize)
            + querySize;
            int y = random(yseed) * (length - 2*querySize)
            + querySize;
            int z = random(zseed) * (length - 2*querySize)
            + querySize;

            int nc = numCluster(x,y,z,querySize,maxLength);
            totCluster += nc;
            if( nc > maxCluster) maxCluster = nc;
            maxRange += maxLength;

            if( i % 10000 == 0 )
                cout << querySize << " " << i << endl;
        }

        float avgInterDis = totInterDis/float(nInterDis);
        float avgCluster = totCluster /= double(numSample);
    }
}

```

```

maxRange /= double(numSample);

/*
cout << querySize << " " << avgCluster
<< " " << maxCluster << " " << maxRange;
cout << " " << avgInterDis << endl;
statOut << querySize << " " << avgCluster
<< " " << maxCluster << " " << maxRange;
statOut << " " << avgInterDis << endl;
*/
}
}

int numCluster( int x, int y, int z, int querySize, double & maxLength
)
{
    int nc = 1;//number of clusters, default value is set to 1.

    list<int> lnc;

    //square query
    for( int i = x; i < x + querySize; i++ )
    for( int j = y; j < y + querySize; j++ )
    for( int k = z; k < z + querySize; k++ )

    //polyhedra query
    for( int i = x; i < x + 0.5 * querySize; i++ )
    for( int j = y; j < y + 1.6667 * querySize; j++ )
    for( int k = z; k < z + querySize; k++ )

    //sphere query
    for( int i = x - querySize; i < x + querySize; i++ )
    for( int j = y - querySize; j < y + querySize; j++ )
    for( int k = z - querySize; k < z + querySize; k++ )
    {
        //sphere query check
        /*float distance = (i-x)*(i-x);
distance += (j-y)*(j-y);
distance += (k-z)*(k-z);
if( distance <= pow(querySize,2))
*/
        {
            int index = 0;
            bitmask_t c1[nDims] = {i,j,k};
            switch(curve_flag)
            {
            case 'Z':
                index = zOrder_c2i(nDims,nBits,c1);
                break;
            case 'G':
                index = gray_c2i(nDims,nBits,c1);
                break;
            case 'H':
                index = hilbert_c2i(nDims,nBits,c1);
                break;
            }
            lnc.push_back(index);
        }
    }
}

```

```

    }
}

lnc.sort();

list<int>::iterator p = lnc.begin();
list<int>::iterator q = lnc.begin();

maxLength = -1*( *q );
q++;
for( ; q != lnc.end(); q++ )
{
    // discontinuous interger means two clusters
    if( *q > *p + 1 )
    {
        nc ++;
        //inter-cluster distance between two neighbouring clusters.
        nInterDis ++;
        totInterDis += *q - *p;
    }
    p++;
}
maxLength += *p;// the Maximum range of the query

return nc;
}

/*****
* Function : zOrder_c2i( int nDims, int nBits, bitmask_t coord[] )
* Purpose  : Mapping coordinates (x,y,z) into the index under Z-curve.
*****/
int zOrder_c2i( int nDims, int nBits, bitmask_t coord[] )
{
    int zcode = 0, n = 1;

    for( int i = 0; i < nBits; i++ )
    {
        /*
        int x1 = x%2;
        int y1 = y%2;
        int z1 = z%2;
        zcode += x1*4*n + y1*2*n + z1*n;//three-dimensional case
        x >>= 1;
        y >>= 1;
        z >>= 1;
        n <<= 3;
        */
        for( int j = 0; j < nDims; j++ )
        {
            int k = coord[j] % 2;
            zcode += k * n;
            n <<= 1;
            coord[j] >>= 1;
        }
    }
    return zcode;
}
}

```



```

/*****
* Function : gray_c2i( int nDims, int nBits, bitmask_t coord[] )      *
* Purpose  : Mapping coordinates (x,y,z) into the index under      *
*           Gray-coded                                             curve.
*
*****/
int gray_c2i( int nDims, int nBits, bitmask_t coord[] )
{
    int zcode = zOrder_c2i(nDims, nBits, coord);
    return zcode^(zcode >>1 );//convert binary code to gray code
}

/*****
* Function : random( int& izeed )      *
* Purpose  : With input izeed, return a random value between 0 and 1. *
*****/
float random( int& izeed )
{
    const int m = 1048583;
    const int a = 1997;

    izeed = ( a* izeed )%m;
    return float(izeed)/float(m);
}

```

<hilbert.h>

```
/* C header file for Hilbert curve functions */
#ifndef _hilbert_h_
#define _hilbert_h_

#ifdef __cplusplus
extern "C" {
#endif

/* define the bitmask_t type as an integer of sufficient size */
typedef unsigned long int bitmask_t;
/* define the halfmask_t type as an integer of 1/2 the size of
bitmask_t */
typedef unsigned long halfmask_t;

/*****
 * hilbert_i2c
 *
 * Convert an index into a Hilbert curve to a set of coordinates.
 * Inputs:
 *   nDims:      Number of coordinate axes.
 *   nBits:      Number of bits per axis.
 *   index:      The index, contains nDims*nBits bits (so nDims*nBits
must be <= 8*sizeof(bitmask_t)).
 * Outputs:
 *   coord:      The list of nDims coordinates, each with nBits bits.
 * Assumptions:
 *   nDims*nBits <= (sizeof index) * (bits_per_byte)
 */

void hilbert_i2c(unsigned nDims, unsigned nBits, bitmask_t index,
bitmask_t coord[]);

/*****
 * hilbert_c2i
 *
 * Convert coordinates of a point on a Hilbert curve to its index.
 * Inputs:
 *   nDims:      Number of coordinates.
 *   nBits:      Number of bits/coordinate.
 *   coord:      Array of n nBits-bit coordinates.
 * Outputs:
 *   index:      Output index value.  nDims*nBits bits.
 * Assumptions:
 *   nDims*nBits <= (sizeof bitmask_t) * (bits_per_byte)
 */

/*****
 * hilbert_cmp, hilbert_ieee_cmp
 *
 * Determine which of two points lies further along the Hilbert curve
 * Inputs:
 *   nDims:      Number of coordinates.

```

```

* nBytes:      Number of bytes of storage/coordinate (hilbert_cmp
only)
* nBits:      Number of bits/coordinate. (hilbert_cmp only)
* coord1:     Array of nDims nBytes-byte coordinates (or doubles for
ieee_cmp).
* coord2:     Array of nDims nBytes-byte coordinates (or doubles for
ieee_cmp).
* Return value:
*      -1, 0, or 1 according to whether
      coord1<coord2, coord1==coord2, coord1>coord2
* Assumptions:
*      nBits <= (sizeof bitmask_t) * (bits_per_byte)
*/
int hilbert_cmp(unsigned nDims, unsigned nBytes, unsigned nBits, void
const* coord1, void const* coord2);
int hilbert_ieee_cmp(unsigned nDims, double const* coord1, double
const* coord2);

/*****
* hilbert_box_vtx
*
* Determine the first or last vertex of a box to lie on a Hilbert
curve
* Inputs:
* nDims:      Number of coordinates.
* nBytes:     Number of bytes/coordinate.
* nBits:     Number of bits/coordinate. (hilbert_cmp only)
* findMin:   Is it the least vertex sought?
* coord1:    Array of nDims nBytes-byte coordinates - one corner of
box
* coord2:    Array of nDims nBytes-byte coordinates - opposite
corner
* Output:
*      c1 and c2 modified to refer to selected corner
*      value returned is log2 of size of largest power-of-two-aligned
box that
*      contains the selected corner and no other corners
* Assumptions:
*      nBits <= (sizeof bitmask_t) * (bits_per_byte)
*/

unsigned
hilbert_box_vtx(unsigned nDims, unsigned nBytes, unsigned nBits,
int findMin, void* c1, void* c2);

unsigned
hilbert_ieee_box_vtx(unsigned nDims,
int findMin, double* c1, double* c2);

/*****
* hilbert_box_vtx
*
* Determine the first or last vertex of a box to lie on a Hilbert
curve
* Inputs:
* nDims:     Number of coordinates.
* nBytes:    Number of bytes/coordinate.

```

```

* nBits:      Number of bits/coordinate (hilbert_cmp only)
* findMin:    Is it the least vertex sought?
* coord1:     Array of nDims nBytes-byte coordinates - one corner of
box
* coord2:     Array of nDims nBytes-byte coordinates - opposite
corner
* Output:
*   c1 and c2 modified to refer to selected corner
*   value returned is log2 of size of largest power-of-two-aligned
box that
*   contains the selected corner and no other corners
* Assumptions:
*   nBits <= (sizeof bitmask_t) * (bits_per_byte)
*/

```

```

unsigned
hilbert_box_vtx(unsigned nDims, unsigned nBytes, unsigned nBits,
               int findMin, void* c1, void* c2);

```

```

unsigned
hilbert_ieee_box_vtx(unsigned nDims,
                    int findMin, double* c1, double* c2);

```

```

/*****
* hilbert_box_pt
*
* Determine the first or last point of a box to lie on a Hilbert curve
* Inputs:
* nDims:      Number of coordinates.
* nBytes:     Number of bytes/coordinate.
* nBits:      Number of bits/coordinate.
* findMin:    Is it the least vertex sought?
* coord1:     Array of nDims nBytes-byte coordinates - one corner of
box
* coord2:     Array of nDims nBytes-byte coordinates - opposite
corner
* Output:
*   c1 and c2 modified to refer to least point
* Assumptions:
*   nBits <= (sizeof bitmask_t) * (bits_per_byte)
*/

```

```

unsigned
hilbert_box_pt(unsigned nDims, unsigned nBytes, unsigned nBits,
               int findMin, void* coord1, void* coord2);

```

```

unsigned
hilbert_ieee_box_pt(unsigned nDims,
                   int findMin, double* c1, double* c2);

```

```

/*****
* hilbert_nextinbox
*
* Determine the first point of a box after a given point to lie on a
Hilbert curve
* Inputs:
* nDims:      Number of coordinates.
* nBytes:     Number of bytes/coordinate.
* nBits:      Number of bits/coordinate.

```

```

* findPrev: Is the previous point sought?
* coord1: Array of nDims nBytes-byte coordinates - one corner of
box
* coord2: Array of nDims nBytes-byte coordinates - opposite
corner
* point: Array of nDims nBytes-byte coordinates - lower bound on
point returned
*
* Output:
if returns 1:
* c1 and c2 modified to refer to least point after "point" in box
else returns 0:
arguments unchanged; "point" is beyond the last point of the
box
* Assumptions:
* nBits <= (sizeof bitmask_t) * (bits_per_byte)
*/

int
hilbert_nextinbox(unsigned nDims, unsigned nBytes, unsigned nBits,
int findPrev, void* coord1, void* coord2,
void const* point);

/*****
* hilbert_incr
*
* Advance from one point to its successor on a Hilbert curve
* Inputs:
* nDims: Number of coordinates.
* nBits: Number of bits/coordinate.
* coord: Array of nDims nBits-bit coordinates.
* Output:
* coord: Next point on Hilbert curve
* Assumptions:
* nBits <= (sizeof bitmask_t) * (bits_per_byte)
*/

void
hilbert_incr(unsigned nDims, unsigned nBits, bitmask_t coord[]);

/* See LICENSE below for information on rights to use, modify and
distribute
this code. */

/*
* hilbert.c - Computes Hilbert space-filling curve coordinates,
without
* recursion, from integer index, and vice versa, and other Hilbert-
related
* calculations. Also known as Pi-order or Peano scan.
*
* Author: Doug Moore
* Dept. of Computational and Applied Math
* Rice University
* http://www.caam.rice.edu/~dougmo
* Date: Sun Feb 20 2000

```

```

* Copyright (c) 1998-2000, Rice University
*
* Acknowledgement:
* This implementation is based on the work of A. R. Butz ("Alternative
* Algorithm for Hilbert's Space-Filling Curve", IEEE Trans. Comp.,
April,
* 1971, pp 424-426) and its interpretation by Spencer W. Thomas,
University
* of Michigan (http://www-personal.umich.edu/~spencer/Home.html) in
his widely
* available C software. While the implementation here differs
considerably
* from his, the first two interfaces and the style of some comments
are very
* much derived from his work. */

#include "hilbert.h"

/* implementation of the hilbert functions */

#define adjust_rotation(rotation,nDims,bits)
\
do {
\
    /* rotation = (rotation + 1 + ffs(bits)) % nDims; */
\
    bits &= -bits & nd1Ones;
\
    while (bits)
\
        bits >>= 1, ++rotation;
\
    if ( ++rotation >= nDims )
\
        rotation -= nDims;
\
} while (0)

#define ones(T,k) (((T)2) << (k-1)) - 1)

#define rdbit(w,k) (((w) >> (k)) & 1)

#define rotateRight(arg, nRots, nDims)
\
(((arg) >> (nRots)) | ((arg) << ((nDims)-(nRots)))) &
ones(bitmask_t,nDims)

#define rotateLeft(arg, nRots, nDims)
\
(((arg) << (nRots)) | ((arg) >> ((nDims)-(nRots)))) &
ones(bitmask_t,nDims)

#define DLOGB_BIT_TRANSPOSE
static bitmask_t
bitTranspose(unsigned nDims, unsigned nBits, bitmask_t inCoords)
#if defined(DLOGB_BIT_TRANSPOSE)
{

```

```

unsigned const nDims1 = nDims-1;
unsigned inB = nBits;
unsigned utB;
bitmask_t inFieldEnds = 1;
bitmask_t inMask = ones(bitmask_t, inB);
bitmask_t coords = 0;

while ((utB = inB / 2))
{
    unsigned const shiftAmt = nDims1 * utB;
    bitmask_t const utFieldEnds =
        inFieldEnds | (inFieldEnds << (shiftAmt+utB));
    bitmask_t const utMask =
        (utFieldEnds << utB) - utFieldEnds;
    bitmask_t utCoords = 0;
    unsigned d;
    if (inB & 1)
    {
        bitmask_t const inFieldStarts = inFieldEnds << (inB-1);
        unsigned oddShift = 2*shiftAmt;
        for (d = 0; d < nDims; ++d)
        {
            bitmask_t in = inCoords & inMask;
            inCoords >>= inB;
            coords |= (in & inFieldStarts) << oddShift++;
            in &= ~inFieldStarts;
            in = (in | (in << shiftAmt)) & utMask;
            utCoords |= in << (d*utB);
        }
    }
    else
    {
        for (d = 0; d < nDims; ++d)
        {
            bitmask_t in = inCoords & inMask;
            inCoords >>= inB;
            in = (in | (in << shiftAmt)) & utMask;
            utCoords |= in << (d*utB);
        }
    }
    inCoords = utCoords;
    inB = utB;
    inFieldEnds = utFieldEnds;
    inMask = utMask;
}
coords |= inCoords;
return coords;
}
#else
{
    bitmask_t coords = 0;
    unsigned d;
    for (d = 0; d < nDims; ++d)
    {
        unsigned b;
        bitmask_t in = inCoords & ones(bitmask_t, nBits);
        bitmask_t out = 0;
    }
}

```

```

        inCoords >>= nBits;
        for (b = nBits; b--;)
        {
            out <<= nDims;
            out |= rdbit(in, b);
        }
        coords |= out << d;
    }
    return coords;
}
#endif

/*****
 * hilbert_i2c
 *
 * Convert an index into a Hilbert curve to a set of coordinates.
 * Inputs:
 *   nDims:      Number of coordinate axes.
 *   nBits:      Number of bits per axis.
 *   index:      The index, contains nDims*nBits bits
 *               (so nDims*nBits must be <= 8*sizeof(bitmask_t)).
 * Outputs:
 *   coord:      The list of nDims coordinates, each with nBits bits.
 * Assumptions:
 *   nDims*nBits <= (sizeof index) * (bits_per_byte)
 */
void
//hilbert_i2c(unsigned nDims, unsigned nBits, bitmask_t index,
bitmask_t coord[])
hilbert_i2c(unsigned nDims, unsigned nBits, bitmask_t index, bitmask_t
coord[])
{
    if (nDims > 1)
    {
        bitmask_t coords;
        halfmask_t const nbOnes = ones(halfmask_t, nBits);
        unsigned d;

        if (nBits > 1)
        {
            unsigned const nDimsBits = nDims*nBits;
            halfmask_t const ndOnes = ones(halfmask_t, nDims);
            halfmask_t const nd1Ones = ndOnes >> 1; /* for adjust_rotation
*/
            unsigned b = nDimsBits;
            unsigned rotation = 0;
            halfmask_t flipBit = 0;
            bitmask_t const nthbits = ones(bitmask_t, nDimsBits) / ndOnes;
            index ^= (index ^ nthbits) >> 1;
            coords = 0;
            do
            {
                halfmask_t bits = (index >> (b--nDims)) & ndOnes;
                coords <<= nDims;
                coords |= rotateLeft(bits, rotation, nDims) ^ flipBit;
                flipBit = (halfmask_t)1 << rotation;
                adjust_rotation(rotation, nDims, bits);
            }

```



```

        } while (b);
    for (b = nDims; b < nDimsBits; b *= 2)
        coords ^= coords >> b;
    coords = bitTranspose(nBits, nDims, coords);
}
else
    coords = index ^ (index >> 1);

    for (d = 0; d < nDims; ++d)
    {
        coord[d] = coords & nbOnes;
        coords >>= nBits;
    }
}
else
    coord[0] = index;
}

/*****
 * hilbert_c2i
 *
 * Convert coordinates of a point on a Hilbert curve to its index.
 * Inputs:
 * nDims:      Number of coordinates.
 * nBits:      Number of bits/coordinate.
 * coord:      Array of n nBits-bit coordinates.
 * Outputs:
 * index:      Output index value.  nDims*nBits bits.
 * Assumptions:
 * nDims*nBits <= (sizeof bitmask_t) * (bits_per_byte)
 */
bitmask_t
//hilbert_c2i(unsigned nDims, unsigned nBits, bitmask_t const coord[])
hilbert_c2i(unsigned nDims, unsigned nBits, bitmask_t coord[])
{
    if (nDims > 1)
    {
        unsigned const nDimsBits = nDims*nBits;
        bitmask_t index;
        unsigned d;
        bitmask_t coords = 0;
        for (d = nDims; d--;)
        {
            coords <<= nBits;
            coords |= coord[d];
        }

        if (nBits > 1)
        {
            halfmask_t const ndOnes = ones(halfmask_t, nDims);
            halfmask_t const nd1Ones = ndOnes >> 1; /* for adjust_rotation
*/
            unsigned b = nDimsBits;
            unsigned rotation = 0;
            halfmask_t flipBit = 0;
            bitmask_t const nthbits = ones(bitmask_t, nDimsBits) / ndOnes;
            coords = bitTranspose(nDims, nBits, coords);

```

```

coords ^= coords >> nDims;
index = 0;
do
{
    halfmask_t bits = (coords >> (b-=nDims)) & ndOnes;
    bits = rotateRight(flipBit ^ bits, rotation, nDims);
    index <<= nDims;
    index |= bits;
    flipBit = (halfmask_t)1 << rotation;
    adjust_rotation(rotation, nDims, bits);
} while (b);
index ^= nthbits >> 1;
}
else
index = coords;
for (d = 1; d < nDimsBits; d *= 2)
index ^= index >> d;
return index;
}
else
return coord[0];
}

#ifdef __cplusplus
}
#endif

#endif /* _hilbert_h_ */

```

γ

VITA

Hua LI

Candidate for the Degree of  
Master of Science

Thesis: LOCALITY PRESERVING PROPERTIES OF SPACE-FILLING CURVES

Major Field: Computer Science

Biographical:

Education: Graduated from Nanjing University, Nanjing, China, with a Bachelor in Physics in June, 1989; received a Doctor of Philosophy in Physics at Nanjing University, Nanjing, China in June, 1995; completed the requirements for the Master of Science degree in Computer Science at Oklahoma State University in May, 2002.

Experience: Visiting Computational Physicist March 1998 - September 1999 Department of Physics, HKUST Hong Kong. Computational Physicist and faculty September 1995 - September 1999 National Lab of Solid State Microstructures, Nanjing University, Nanjing China.