Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 105 (2008) 155–163
Information Processing Letters
www.elsevier.com/locate/ipl

Compact Hilbert indices: Space-filling curves for domains with unequal side lengths ☆

Chris H. Hamilton, Andrew Rau-Chaplin *

Dalhousie University, 6050 University Avenue, Halifax, NS, Canada, B3H 1W5

Received 28 March 2007

Available online 10 October 2007

Communicated by F. Dehne

Abstract

In this paper we define a new *compact Hilbert index* which, while maintaining all of the advantages of the standard Hilbert curve, permits spaces with unequal dimension cardinalities. The compact Hilbert index can be used in any application that would have previously relied on Hilbert curves but, in the case of unequal side lengths, provides a more memory efficient representation. This advantage is particularly important in distributed applications (Parallel, P2P and Grid), in which not only is memory space saved but communication volume is significantly reduced.

Keywords: Hilbert curves; Space filling curves; Algorithms

1. Introduction

At the heart of many data intensive applications is the need to store, manipulate and analyze large repositories of multi-dimensional data. Such multi-dimensional data comes in many forms [1–13] including, for example, spectral elements in a parallel high resolution atmospherical global circulation model [14], tissue microarray data in a co-operative Grid-based oncology system [15] or business oriented OLAP data [16].

A common challenge in all of these applications is how best to group and order the multi-dimensional data to promote efficient processing. For one-dimensional data, sorting is an obvious approach as it groups data items that are close together in the key dimension. For example, if we have time-stamped transactional data items for a bank account, we may first sort them by time-stamp in order to then efficiently compute hourly, daily and monthly running balances.

With multi-dimensional data the appropriate grouping strategy is much less obvious. We may of course pick an ordering of the dimensions, say $dim_1, dim_2, \ldots, dim_d$, and sort by it, but such an approach favors some dimensions over others. Data items that are close in dim_1 will be closely grouped, while items that share values in $dim_2, \ldots, dim_d$, but not dim_1, may be very far apart. If our data items represent points in 3D space which ordering is better: x, y, z or z, x, y or one of the other four possible orderings? Note that none of these orderings captures the natural idea of locality, that is that

☆ Funded in part by the Natural Sciences and Engineering Research Council of Canada.

* Corresponding author.
E-mail addresses: chamilton@cs.dal.ca (C.H. Hamilton), arc@cs.dal.ca (A. Rau-Chaplin).

0020-0190/$ – see front matter © 2007 Published by Elsevier B.V.
doi:10.1016/j.ipl.2007.08.034

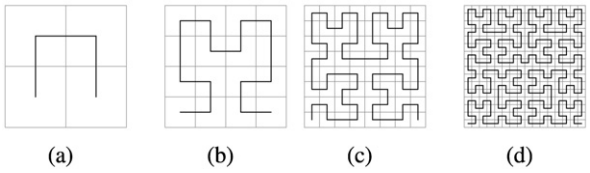

Fig. 1. First four iterations of the 2D Hilbert curve, standard view.

points that are close together in Euclidian space (and therefore more likely to interact in any physical simulation) are grouped closely together in the resulting linear ordering.

A powerful and widely used paradigm for ordering multi-dimensional data is the use of space-filling curves [17–22]. Space-filling curves are continuous self-similar functions that map between a one-dimensional interval and a multi-dimensional set. By convention, they are generally defined as continuous mappings from the unit interval to the unit n-dimensional hypercube. Originally formulated by Peano in 1890 [22] they have found applications in a variety of fields, including mathematics [23], image processing and compression [9,11], cryptology [24], algorithms [12], scientific computing [7], parallel computing [8], geographic information systems [2] and database systems [5,10]. Such curves tend to be good at preserving locality: points that are close together (with respect to Euclidean distance) in the multi-dimensional set tend to be close together in the linear ordering defined by the curve. Of the space-filling curves, the Hilbert curve (see Fig. 1) has been shown to have strong locality preserving properties [21] and, as such, has been the focus of considerable research, with numerous algorithms constructed to compute it [25,18, 20,26,27].

More formally, consider an n-dimensional lattice with 2^m points per dimensions,

$$\mathbb{P} = \underbrace{\mathbb{B}^m \times \cdots \times \mathbb{B}^m}_{n \text{ times}},$$

where $\mathbb{B}^m = \{0,1\}^m$. A standard Hilbert index is a function $H : \mathbb{P} \to \mathbb{B}^{nm}$, which maps each point to its index (interpreting $\mathbf{x} \in \mathbb{B}^{mn}$ as an integer in $\mathbb{Z}_{2^{mn}}$) on the Hilbert curve as it passes through the lattice.

Space-filling curves, particularly Hilbert curves, have been extensively used to maintain spatial groupings of multi-dimensional data in a wide variety of applications. In database systems they are used to map multi-dimensional data to linearly ordered external memory (i.e. disk drives) [28]. In data structures they are used to order multi-dimensional data to promote query efficiency [29]. And in distributed information systems they are used to *partition* multi-dimensional data in such a way that points that are close in Euclidian space are likely to be allocated to the same or neighboring processors. The idea of using space-filling curves for partitioning has been key to applications in parallel [16], P2P [30] and Grid Computing [15] settings.

One significant limitation in the standard definition of Hilbert curves is the requirement that the grid size (i.e. the cardinality) in each dimension be the same (i.e. 2^m). In many applications involving points in 3D space, this may be a relatively harmless assumption but in information systems applications where one dimension may represent product id (cardinality 1 000 000) while another represents gender (cardinality 2) it can be extremely wasteful. The approach of padding all dimensions to the cardinality of the largest dimension wastes memory and disk space and increases processing time and communication volume, when manipulating and communicating these "inflated" values.

In this paper we define a new *compact Hilbert index* which, while maintaining all of the advantages of the Hilbert curve, permits unequal dimension cardinalities. More formally, consider an n-dimensional data-set

$$\mathbb{P}' = \mathbb{B}^{m_0} \times \cdots \times \mathbb{B}^{m_{n-1}},$$

where $m_i \in \mathbb{Z}_+$ is the *precision* of the ith dimension (there is an obvious injection $U : \mathbb{P}' \to \mathbb{P}$ that prepends zeroes to each component until they have length m). Storing an element in $\mathbb{P}'$ requires $M = \sum_i m_i$ bits. However, a Hilbert index must be calculated with respect to a hypercube of precision $m = \max_i\{m_i\}$ and requires $nm \geqslant M$ bits of storage. Our compact Hilbert index preserves the ordering of H on $\mathbb{P}'$, but requires only M bits to represent. Formally, it is a mapping $H' : \mathbb{P}' \to \mathbb{B}^M$, such that for all $\mathbf{p}_1, \mathbf{p}_2 \in \mathbb{P}'$,

$$H\big(U(\mathbf{p}_1)\big) < H\big(U(\mathbf{p}_2)\big) \quad \Leftrightarrow \quad H'(\mathbf{p}_1) < H'(\mathbf{p}_2). \quad (1)$$

Note that the compact Hilbert index can be used in any application using Hilbert indices but, in the case of unequal side lengths, provides a more compact representation. This advantage is particularly important in distributed applications, in which not only is memory space saved but communication volume is significantly reduced.

To explore the performance of compact Hilbert indices we performed a series of experiments with both synthetic and real multi-dimensional data. In both cases, in addition to significant space savings, the use of compact Hilbert curves reduced the time required to order data in Hilbert order. For example, for a 4-dimensional data-set extracted from a large Apache web log, compact Hilbert indices achieved a data size reduction of 2.2 and sorting based on these indices was 4.3 times faster

than the dynamic comparison routine implemented in Moore's widely used library [27].

The remainder of this paper is organized as follows. In Section 2 we review the definition of and algorithms for computing Hilbert curves while emphasizing a geometric perspective. In Section 3 we define the notion of *compact Hilbert indices* and derive an algorithm for calculating the mapping. In Section 4 we explore the performance of compact Hilbert indices, in particular demonstrating significantly improved sorting times as compared to competing techniques.

2. A geometric approach to Hilbert curves

In this section we briefly describe the standard Hilbert curve from a geometric point of view and give an algorithm for finding the index on the Hilbert curve of a given point in the lattice. While motivated and derived geometrically, the resulting algorithm is a variant of the *de facto* standard method presented by Butz [18] and later implemented by Moore [27]. However, our geometric approach highlights the source of redundant data in standard Hilbert indices and facilitates the development of compact Hilbert indices in the following section.

Consider the traditional recursive geometric construction of the two-dimensional Hilbert curve. The curve is initially defined on a 2×2 lattice with a $\sqcap$ shape as shown in Figure 1. Given an order k curve defined on a $2^k \times 2^k$ lattice we define the curve on a $2^{k+1} \times 2^{k+1}$ lattice as follows:

(i) Place a copy of the curve, rotated 90° counter clockwise, in the lower right cell.
(ii) Place a copy of the curve, rotated 90° clockwise, in the lower left cell.
(iii) Place a copy of the curve in each of the upper cells.
(iv) Connect these four disjoint curves in the obvious manner.

The first four iterations of this construction are shown in Fig. 1.

The basic unit of the Hilbert curve is the familiar $\sqcap$ shape, which may be uniquely parameterized by considering the entry and exit points into the square lattice of points being walked through. Using the same approach as that taken in [25], Fig. 2 illustrates the Hilbert curve where the line segments of Fig. 1 have been replaced by arcs. As noted in [25] this presentation conveys more information as it indicates at some level the order in which points are visited in a given cell. The arcs show that the curve enters each cell at a given vertex, visits all the points in the cell and exits through another vertex before entering the next cell.

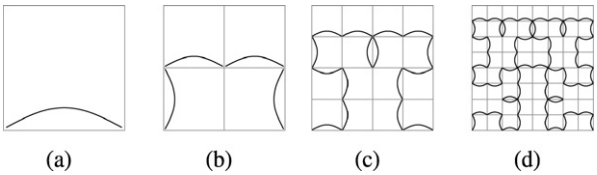

Fig. 2. First four iterations of the 2D Hilbert curve, arc view.

Many algorithms for calculating Hilbert indices are based on a geometric analysis of how the curve decomposes into appropriately transformed smaller versions of itself. Nulty [31] presents a generic algorithm describing this approach with his function *SpaceKey*:

1. Find the cell containing the point of interest.
2. Update the key (index) value appropriately.
3. Transform as necessary.
4. Continue until sufficient precision has been attained.

This generic framework motivates our algorithmic approach.

Find the cell containing the point of interest. Finding the cell amounts to determining whether the point lies in the upper or lower half-plane with respect to each dimension. Assuming we are working on an order m curve, a point is represented by a point $\mathbf{p} = [p_0, p_1] \in \mathbb{B}^m \times \mathbb{B}^m$. Determining in which half-plane the point lies with respect to the ith coordinate is equivalent to determining the truth value of $p_i < 2^{m-1}$, which is equal to the $(m-1)$th bit of p_i, $\text{bit}(p_i, m-1)$.

Update the key. Given the orientation at the current resolution (uniquely defined by the entry e and exit f of the curve through the lattice), we determine the order in which each of the cells will be visited. Knowing that all points in a cell are visited before moving on to the next, the index of the cell of interest tells us whether the point of interest is visited in the first quarter of the curve, or the second and so on. In other words, we may determine two bits of the Hilbert index h.

Transform as necessary. Knowing the index i of the cell in which the point of interest lies, we may determine the entry and exit points of the Hilbert curve through this cell. In order to proceed, we zoom in on the cell containing the point and transform (rotate and reflect) it to the canonical orientation (entry in lower left, exit in lower right). This can be done by taking the composition

of the transforms associated with our current orientation and that of the block we are zooming in on.

Continue until sufficient precision has been attained. Zooming in on the cell containing our point of interest, we are now inspecting an order $m-1$ Hilbert curve through a sub-cell of our original space. We repeat this procedure for each of the remaining $m-1$ levels of precision, each time calculating a further 2 bits of the Hilbert index. At the end of the process, we have a $2m$ bit Hilbert index, isolating a single point on the curve of length 2^{2m} through the $\mathbb{B}^m \times \mathbb{B}^m$ lattice.

2.1. Generalizing to higher dimensions

The described approach yields a straightforward algorithm for the calculation of two-dimensional Hilbert indices. In order to generalize it to higher dimensions, we need to identify the properties of the Hilbert curve we wish to generalize. The first observation relates to the order of the curve through cells. In two dimensions, successive cells are immediate neighbors along exactly one dimension. Given a 2 bit labeling for each of the cells, this means that in labels of successive sub-cells, *exactly* one bit will change. This is simply a *Gray Code* [32] over 2 bit integers. In n dimensions, we have 2^n cells each labeled with an n-bit string and we may use the n-bit Gray Code to impose an ordering on the cells.

A Gray Code may be interpreted as a Hamiltonian circuit through the vertices of a hypercube in n dimensions. This implies that the first (entry) and last (exit) points are also immediate neighbors. Thus, we may uniquely determine the orientation of a given cell i by considering the entry $e(i)$ into the cell and the dimension $0 \leqslant d(i) < n$ along which the exit point is our neighbor. Having chosen the Gray Code order as the ordering through the cells, consistent orientations of each cell have to be determined such that the exit vertex of a cell is immediately adjacent to the entry vertex of the next cell. In [33], a closed form is derived for the quantities $e(i)$, $f(i)$ and $d(i)$. An order 2 three-dimensional Hilbert curve and the associated arc representation may be found in Fig. 3.

A full analysis of the rotations and reflections involved in the calculation of the Hilbert index shows that they may all be expressed very naturally in base 2 arithmetic. In fact, reflection may be viewed as the exclusive-or ($\veebar$) operation and rotation as a bitwise rotation ($\circlearrowleft$) operation. These simplifications lead directly to the formulation of Algorithm 1. For full details, including proofs and inverse algorithms, refer to [33].

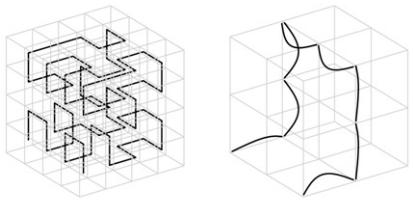

Fig. 3. Standard and arc views of the order 2 three-dimensional Hilbert curve.

Calculates the Hilbert index of a point.

Input: $n, m \in \mathbb{Z}_+$ and a point $\mathbf{p} \in \mathbb{P}$.
Output $h \in \mathbb{B}^{nm}$, the Hilbert index of the point $\mathbf{p} \in \mathbb{P}$.

1: $(h, e, d) \leftarrow (0, 0, 0)$
2: **for** $i = m-1$ to 0 **do**
3: $\quad l \leftarrow [\mathrm{bit}(p_{n-1}, i) \cdots \mathrm{bit}(p_0, i)]_{[2]}$ // Get cell label
4: $\quad t \leftarrow (l \veebar e) \circlearrowleft d$ // Transform to canonical orientation
5: $\quad w = \mathrm{gc}^{-1}(t)$ // Determine cell index in gc order
6: $\quad h \leftarrow (h \triangleleft n) \vee w$ // Add n bits to Hilbert index
7: $\quad e \leftarrow e \veebar (e(w) \circlearrowleft d)$ // Compose transforms
8: $\quad d \leftarrow d + d(w) + 1 \bmod n$
9: **end for**

Algorithm 1. HILBERTINDEX.

Algorithm 1 is clearly visible as falling under the *SpaceKey* framework of [31]. In contrast, Butz's algorithm merges the transformation (line 4) and composition (line 7) into a compound operation and moves the inverse Gray Code operation (line 5) outside of the loop, leading to a more terse implementation with less intermediate variables. However, having each of the *SpaceKey* steps visible facilitates the development of compact Hilbert indices.

3. Compact Hilbert indices

As discussed in Section 1 it is desirable to have a mapping that preserves the relative ordering of the Hilbert curve but does not require additional space to represent. A simple method to construct such a mapping is to walk through all the points in $\mathbb{P}'$, calculate their Hilbert indices and sort them based on these values. Then, assign to each point $\mathbf{p}$ its rank in this sorted list as an index. Trivially, this index has the same ordering as the Hilbert ordering over $\mathbb{P}'$ and it requires only $M = \sum_i m_i$ bits to represent. However, in order to generate the index in this manner we must first enumerate the entire space, a prohibitive cost. The key to calculating this index directly, referred to as the *compact Hilbert index*, lies in an observation of how bits from the point $\mathbf{p}$

travel through Algorithm 1 and contribute to the Hilbert index.

3.1. An Observation

We inspect line 3 of Algorithm 1 which calculates the location l of the point $\mathbf{p}$ as

$$l = \left[\text{bit}(p_{n-1}, i) \cdots \text{bit}(p_0, i)\right]_{[2]}.$$

Due to the varying precisions of each coordinate we know that for any point $\mathbf{p} \in \mathbb{P}'$, $\text{bit}(p_j, i) = 0$ when $i \geqslant m_j$. Thus at any given iteration i, some subset of the n bits of l may be fixed and known to be zero. These bits do not provide any information to the calculation, yet they are still used to calculate a full n bits of the index h. Following these redundant bits through lines 4–5 shows how we can remove them from the output while still preserving the total ordering of the points in $\mathbb{P}'$ as visited by the Hilbert curve over $\mathbb{P}$.

Let $\mathcal{A}_i = \{j\colon\ m_j > i,\ 0 \leqslant j < n\}$ be the set of "active" dimensions at iteration i. Consider the calculation of t, the transformed location, on line 4 of Algorithm 1. Since every inactive bit of l is zero valued, then the bits of $l \veebar e$ at these positions will simply take on the value of the corresponding bits of e. Thus the only bits of $l \veebar e$ whose values are "free" are precisely those in $\mathcal{A}_i$. The rotation operator serves only to shuffle the bits of $l \veebar e$ in a simple manner. Let $\mathcal{F}_i$ be the set of free bits of t at iteration i; that is, those bits whose values are affected by l and in turn $\mathbf{p}$. It is easy to see that precisely those bits j of t such that $m_{(j+d \bmod n)} > i$ will be free at iteration i. Since $|\mathcal{F}_i| = |\mathcal{A}_i|$, then we see that both l and t may only be one of $2^{|\mathcal{F}_i|}$ unique values. Additionally, we know that the Gray Code, and hence its inverse, is a bijective operator over $\mathbb{B}^n$ (see [33] for a proof of this simple fact). Thus the final value w may also only be one of $2^{|\mathcal{F}_i|}$ distinct values.

Let r be the rank of w with respect to all possible values w may take on at a given iteration i of the algorithm. Then r may be represented as an $|\mathcal{F}_i|$-bit integer and satisfies $r_1 < r_2 \Leftrightarrow w_1 < w_2$. Instead of appending w to our partially calculated index h, we may append the rank r. By the previous result we see that for any $\mathbf{p}_1, \mathbf{p}_2 \in \mathbb{P}'$, the following holds with respect to modified indices, h_1 and h_2, constructed from rank values:

$$h_1 < h_2 \quad \Leftrightarrow \quad H\big(U(\mathbf{p}_1)\big) < H\big(U(\mathbf{p}_2)\big).$$

Additionally, by counting the number of free bits $|\mathcal{F}_i|$ over all i, we see that the resulting compact index will be exactly M bits in length. Thus an index calculated in this manner satisfies Eq. (1), as desired. It remains only to show how to calculate the rank r of a value w given $\mathcal{F}$. We first consider an example.

Example 3.1. We consider the values of t, w and r for $n = 4$, $(e \circlearrowleft d) = [0100]_{[2]}$ and 2 free bits. In the following table, the free bits of t have been underlined, while the ranks r have been calculated by inspection over the set of all w values.

t	4	5	12	13
w	7	6	8	9
r	1	0	2	3

$[t]_{[2]}$	$0\underline{10}0$	$0\underline{10}1$	$1\underline{10}0$	$1\underline{10}1$
$[w]_{[2]}$	$0\underline{11}1$	$0\underline{11}0$	$1\underline{00}0$	$1\underline{00}1$
$[r]_{[2]}$	01	00	10	11

As can be seen, the rank r of w can be constructed by extracting the free bits $f \in \mathcal{F}$ from the Gray Code index w. We formalize this in Lemma 3.4 and Theorem 3.5, first stating without proof a few necessary lemmas.

Lemma 3.2. *(See Theorem 2.1 of [33].) Consider a non-negative integer $w \in \mathbb{B}^m$. Let $t = \text{gc}(w)$. Then it follows that $t = w \veebar (w \triangleright 1)$, or equivalently,* $\text{bit}(t, j) = \text{bit}(w, j) + \text{bit}(w, j+1) \bmod 2$.

Lemma 3.3. *(See Theorem 2.2 of [33].) Consider a non-negative integer $t \in \mathbb{B}^m$. Let $w = \text{gc}^{-1}(t)$. Then it follows that* $\text{bit}(w, j) = \sum_{k=j}^{m-1} \text{bit}(t, k)$.

Lemma 3.4. *Given e, d and i, let $\mathcal{F} = \{j\colon\ m_{(j+d \bmod n)} > i,\ 0 \leqslant j < n\}$. Let $\mathcal{T}$ be the set of 2^k distinct values that may differ from $(e \circlearrowleft d)$ only at the $k = |\mathcal{F}|$ bits $j \in \mathcal{F}$. Consider $a \neq b \in \mathcal{T}$. Let l be the index of the most significant bit of a and b that does not match; in other words, $l = \max\{k\colon\ \text{bit}(a, k) \neq \text{bit}(b, k)\}$. It follows that $l \in \mathcal{F}$.*

Proof. Define a mask μ as the n-bit integer such that $\text{bit}(\mu, j) = 1$ for $h \in \mathcal{F}$ and $\text{bit}(\mu, j) = 0$ otherwise. The mask μ is created such that only bits in free positions are one valued. Since $t \in \mathcal{T}$ may only differ from $(e \circlearrowleft d)$ at the bits $j \in \mathcal{F}$, we may rewrite

$$\mathcal{T} = \big\{t\colon\ t \wedge \mu = (e \circlearrowleft d) \wedge \mu, t \in \mathbb{B}^n\big\}.$$

By Lemma 3.3 it follows that

$$\text{bit}(a, l) = \sum_{l \leqslant k < n} \text{bit}(\text{gc}(a), k) \bmod 2.$$

Knowing $\text{bit}(a, k) = \text{bit}(b, k)$ for $k > l$, Lemma 3.2 implies $\text{bit}(\text{gc}(a), k) = \text{bit}(\text{gc}(b), k)$ for $j > l$. Thus:

$$\begin{aligned}
&\text{bit}(a, l) + \text{bit}(b, l) \\
&\quad = \sum_{l \leqslant k < n} \big(\text{bit}\big(\text{gc}(a), k\big) + \text{bit}\big(\text{gc}(b), k\big)\big) \bmod 2 \\
&\quad = \text{bit}\big(\text{gc}(a), l\big) + \text{bit}\big(\text{gc}(b), l\big).
\end{aligned}$$

Suppose $l \notin \mathcal{F}$. Then it follows that $\text{bit}(\text{gc}(a), l) = \text{bit}(\text{gc}(b), l) = \text{bit}(e \circlearrowleft d, l)$ and therefore $\text{bit}(a, l) = \text{bit}(b, l)$, a contradiction. Hence, $l \in \mathcal{F}$. □

Extracts a mask μ indicating which bits of the transformed location t are free at a given iteration i of the CompactHilbertIndex algorithm.

Input: $n, m_0, \ldots, m_{n-1} \in \mathbb{Z}_+$, $i \in \mathbb{Z}_m$ and the current rotation parameter $d \in \mathbb{Z}_n$.
Output: The mask μ of free bits of the transformed location t at iteration i.

1: $\mu \leftarrow 0$
2: **for** $j = n-1$ to 0 **do**
3: $\quad \mu \leftarrow \mu \lhd 1$
4: $\quad$ **if** $m_{(j+d \bmod n)} > i$ **then**
5: $\quad\quad \mu \leftarrow \mu \vee 1$
6: $\quad$ **end if**
7: **end for**

Algorithm 2. ExtractMask.

Returns the Gray Code rank of the given Gray Code index w, with respect to the given mask μ.

Input: $n \in \mathbb{Z}_+$, $\mu, w \in \mathbb{B}^n$.
Output: The Gray Code rank of w with respect to μ.

1: $r \leftarrow 0$
2: **for** $j = n-1$ to 0 **do**
3: $\quad$ **if** $\text{bit}(\mu, j) = 1$ **then**
4: $\quad\quad r \leftarrow (r \lhd 1) \vee \text{bit}(w, j)$
5: $\quad$ **end if**
6: **end for**

Algorithm 3. GrayCodeRank.

Theorem 3.5. *Let $\mathcal{F}, \mathcal{T}$ and μ be as in Lemma* 3.4. *Define the Gray Code Rank as*

$$\text{gcr}(w) = \left[\text{bit}(w, f_{k-1}), \ldots, \text{bit}(w, f_0)\right]_{[2]},$$

where $\mathcal{F} = \{f_0 < \cdots < f_{k-1}\}$ and $w = \text{gc}^{-1}(t)$ for some $t \in \mathcal{T}$. Then for all $t_1, t_2 \in \mathcal{T}$ it follows that $\text{gc}^{-1}(t_1) < \text{gc}^{-1}(t_2) \Leftrightarrow \text{gcr}(\text{gc}^{-1}(t_1)) < \text{gcr}(\text{gc}^{-1}(t_2))$.

Proof. Lemma 3.4 tells us that the most significant differing bit between $\text{gc}^{-1}(t_1)$ and $\text{gc}^{-1}(t_2)$ must be in a free bit position. In other words, the only bits necessary to compare the relative order of these two values are precisely the bits of index $f \in \mathcal{F}$. Thus, if we remove the constrained bits from $\text{gc}^{-1}(t_1)$ and $\text{gc}^{-1}(t_2)$ keeping only the free bits in the same relative order, we are left with two $|\mathcal{F}|$ bit values which preserve the ordering of $\text{gc}^{-1}(t_1)$ and $\text{gc}^{-1}(t_2)$. This corresponds exactly to the values $\text{gcr}(\text{gc}^{-1}(t_1))$ and $\text{gcr}(\text{gc}^{-1}(t_2))$. □

The results of this section and particularly Theorem 3.5 give us the last tools required to create an algorithm to compute compact Hilbert indices. Using Algorithm 1 as a starting point, Algorithms 2–4 allow the computation of the mapping H'.

Inspection shows that each of ExtractMask and GrayCodeRank have $O(n)$ time complexity. Similarly, Lemma 3.3 shows that calculating $\text{gc}^{-1}(t)$ is $O(n)$ (in fact, it may be implemented with $O(\log n)$ complexity). Given that each of $e(w)$, $d(w)$ and $\circlearrowleft$ may be implemented in less than $O(n)$ complexity, we see that both Algorithms 1 and 4 have a net complexity of $O(nm)$. Specifically, this shows us that compact Hilbert indices are only at most a constant factor more expensive to compute than regular Hilbert indices.

Calculates the compact Hilbert index of a point.

Input: $n, m_0, \ldots, m_{n-1} \in \mathbb{Z}_+$ and a point $\mathbf{p} \in \mathbb{P}'$.
Output: $h_c \in \mathbb{B}^M$, the compact Hilbert index of the point $\mathbf{p} \in \mathbb{P}'$.

1: $(h_c, e, d) \leftarrow (0, 0, 0)$
2: $m \leftarrow \max_i\{m_i\}$
3: **for** $i = m-1$ to 0 **do**
4: $\quad \mu \leftarrow \text{ExtractMask}(n, m_0, \ldots, m_{n-1}, i, d)$
5: $\quad l \leftarrow [\text{bit}(p_{n-1}, i) \cdots \text{bit}(p_0, i)]_{[2]}$
6: $\quad t \leftarrow (l \veebar e) \circlearrowleft d$
7: $\quad w \leftarrow \text{gc}^{-1}(t)$
8: $\quad r \leftarrow \text{GrayCodeRank}(n, \mu, w)$
9: $\quad h_c \leftarrow (h_c \lhd \|\mu\|) \vee r$
10: $\quad e \leftarrow e \veebar (e(w) \circlearrowleft d)$
11: $\quad d \leftarrow d + d(w) + 1 \bmod n$
12: **end for**

Algorithm 4. CompactHilbertIndex.

4. Experimental results

To quantify the performance of our algorithms we implemented routines for mapping to and from both regular and compact Hilbert indices. The algorithms are written in C++ and seamlessly handle arbitrary precision.[1] Our Hilbert curve algorithms were then compared to Moore's [27] implementation of Butz's [18] algorithms for various precisions and dimensions (up to $nm \leqslant 64$, the maximum supported by Moore's code) on both artificial and real data. The running times of our compact Hilbert indices were then compared to those of regular Hilbert indices over these and other data-sets. Finally, we examine the effect of using compact Hilbert indices in applications where regular Hilbert indices are currently used. All experiments were performed on a commodity Dual Intel Xeon 3.06 GHz based computer with 2 GB of main memory. All quoted times are wall time.

The WEBLOG data-set consists of the log files of an Apache web server, taken over a 139 day period from August to September of 2004. A 4-dimensional data-set of ~ 7.7 million points was extracted from the over 154 million rows of log data. The four dimensions recorded the IP address, day of access, hour of access and HTTP

[1] See http://www.cs.dal.ca/~chamilto/hilbert/.

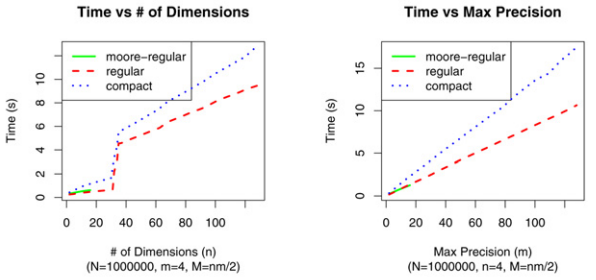

Fig. 4. Comparing performance over random data-sets. (a) Time to calculate N indices with $m = 4$ as n varies. (b) Time to calculate N indices with $n = 4$ as m varies.

return code for each log entry. They had cardinalities of 834 406, 139, 24 and 16, respectively, with bit sizes of 20, 8, 5 and 4. A regular Hilbert index requires 80 bits to represent while a compact Hilbert index requires only 37, a savings of over 2.2 times.

4.1. Performance

In order to characterize the performance of our algorithms, we compared them against Moore's code over randomly generated data-sets and varying parameters for N, m, n and M. For the purposes of compact Hilbert indices, precisions m_i were chosen in a monotonically decreasing fashion such that $M = nm/2$. Fig. 4 shows the basic results. The jump visible at $n = 32$ in Fig. 4(a) arises from the code switching to multiple precision representations of n-bit intermediate variables. In general, our regular Hilbert curve implementation slightly outperforms Moore's implementation. When $n \leqslant 32$ the overhead associated with compact Hilbert indices is as much as 2.5 times or 150%. However, as both n and m increase this reduces to a more reasonable ratio of 1.4, or 40%. Although the compact Hilbert indices take slightly longer to compute, they are smaller than full Hilbert indices allowing data points to be replaced with compact Hilbert indices in-place.

4.2. Sorting by Hilbert index

As discussed in Section 1, Hilbert curves are often used to order or partition multi-dimensional data. Thus it becomes necessary to sort points by their Hilbert indices. The simplest approach is to simply calculate the Hilbert index for each point and use this value in sorting. However, often these indices are larger than the points they represent resulting in an increased storage cost. Given the large nature of the data-sets being sorted, it is often critical that the sort be in-place. Moore's solution to this problem was to create a dynamic comparison routine which simultaneously calculates the Hilbert index of both points being compared. It calculates the indices only to the precision required to determine the relative order of the two points. This approach has the benefit that the Hilbert indices are never explicitly stored, but suffers from the problem that they are recalculated at every comparison.

Dynamic Hilbert sorting. Suppose a comparison requires examining the first b bits of the Hilbert indices of two points in order to distinguish them. Since each bit costs O(1) to calculate, this incurs a cost of O(b) for the comparison. Fill et al. [34] explored the average bit-cost per comparison assuming a quick-sort algorithm is being used. They derive an expected O($\log N$) bits per comparison, which implies a total bit complexity of O($N \log^2 N$). Thus, using a quick-sort based algorithm we can expect a total dynamic Hilbert index sorting runtime on the order of O($N \log^2 N$). Although this particular analysis is valid only for the quick-sort algorithm, it is thought this bound holds for the general problem of sorting.[2]

Compact Hilbert sorting. As a competing approach we consider sorting using compact Hilbert indices. Since compact Hilbert indices are the same size as the data from which they are calculated, we first replace the data points with their associated compact Hilbert indices at a net cost of O(Nnm). We then sort these elements before converting back to the original data points. The net cost of this sort is O($N(\log N + nm)$). As long as $nm < \log^2 N$, such an approach will be asymptotically faster than dynamic Hilbert sorting.

Fig. 5(a) shows the results of sorting the `WEBLOG` data-set using both dynamic Hilbert indices and compact Hilbert indices. As predicted, for this and all other data sets tested, the compact Hilbert sorting proved to be much more efficient. As shown in Fig. 5(b), for as little as 100 K data items a speedup of 2 was observed. By 1 M data items that speedup had grown to a factor of 3.4. Speedup continued to increase beyond this point until it reached a factor of over 4.3 on the whole data set. Note that in distributed applications that order and partition data using Hilbert curves, such as [16,30, 15], the benefits of using compact Hilbert curves would be even more pronounced. The use of compact Hilbert

[2] Under the constraint that in order to compare a bit, we must first have compared all bits more significant than it; if we have random bit access a radix sort can generally do better. Hilbert indices are calculated incrementally precluding random bit access.

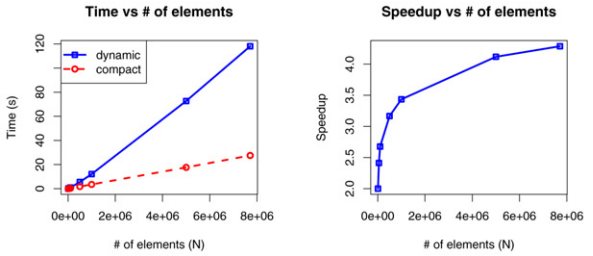

Fig. 5. A comparison of dynamic Hilbert sorting and compact Hilbert sorting using the WEBLOG data-set. The compact curve includes the cost of converting both to and from compact Hilbert indices. (a) Wall times. (b) Relative speed-up.

curves would result in the memory and time savings illustrated in Fig. 5 as well as a corresponding reduction in the overall communication volume and time.

5. Conclusion

Due their wide variety of uses and simplicity, space-filling curves have been often researched since their discovery, finding many applications. Motivated by the lack of intuition in the ubiquitous Butz [18] algorithms for Hilbert curves as implemented by Moore [27], we have reconstructed them from a geometric point of view. Based on this formulation we have then described a compact Hilbert curve which captures the ordering properties of the regular Hilbert curve but without the inefficiency in representation for spaces with unequal side lengths. Finally, we developed algorithms for computing compact Hilbert indices, and demonstrated their performance and utility in real-world applications. Although these compact indices are somewhat more computationally expensive to derive, they result in significant space savings and, in the critical operation of sorting by Hilbert indices, they result in a considerable time saving. For example, on a typical 4-dimensional web log data set, compact Hilbert indices achieved a data size reduction of 2.2 and a speedup of 4.3 over the widely used dynamic Hilbert sort method. It is our hope that the compact Hilbert indices introduced in this paper will find uses in information systems and other applications where multi-dimensional spaces have dimensions of unequal cardinalities.

References

[1] J. Mellor-Crummey, D. Whalley, K. Kennedy, Improving memory hierarchy performance for irregular applications using data and computation reorderings, Int. Journal of Parallel Programming 29 (3) (2001) 217–247.
[2] D.J. Abel, D.M. Mark, A comparative analysis of some two-dimensional orderings, Int. J. of Geographic Information Systems 4 (1) (Jan 1990) 21–31.
[3] C. Alpert, A. Kahng, Multi-way partitioning via spacefilling curves and dynamic programming, in: 31st Conf. on Design Automation, 1994, pp. 652–657.
[4] J.J. Bartholdi III, A routing system based on spacefilling curves, http://www2.isye.gatech.edu/~jjb/mow/mow.pdf, April 1995.
[5] C. Böhm, S. Berchtold, D.A. Keim, Searching in high-dimensional spaces, ACM Computing Surveys 33 (3) (September 2001) 322–373.
[6] S. Chatterjee, A. R. Lebeck, P. K. Patnala, M. Thottethodi, Recursive array layouts and fast parallel matrix multiplication, in: 11th ACM Symp. on Parallel Algorithms and Architectures, 1999.
[7] G. Jin, J.M. Mellor-Crummey, R.J. Fowler, Increasing temporal locality with skewing and recursive blocking, in: 2001 ACM/IEEE Conf. on Supercomputing.
[8] M. Kaddoura, C.-W. Ou, S. Ranka, Partitioning unstructured computational graphs for nonuniform and adaptive environments, IEEE Parallel and Distributed Technology: Systems and Technology 3 (3) (September 1995) 63–69.
[9] C.-H. Lamaque, F. Robert, Image analysis using space-filling curves and 1d wavelet bases, Pattern Recognition 29 (8) (August 1996) 1309–1322.
[10] J.K. Lawder, P.J.H. King, Querying multi-dimensional data indexed using the Hilbert space-filling curve, SIGMOD Record 30 (1) (March 2001) 19–24.
[11] B. Moghaddam, K. Hintz, C. Stewart, Space-filling curves for image compression, in: 1st SPIE Conf. on Automatic Object Recognition, vol. 1471, pp. 414–421.
[12] L.K. Platzman, J.J. Bartholdi III, Spacefilling curves and the planar travelling salesman problem, J. ACM 36 (4) (October 1989) 719–737.
[13] Y. Zhang, R.E. Webber, Space diffusion: An improved parallel halftoning technique using space-filling curves, in: SIGGRAPH 93, 1993, pp. 305–312.
[14] J.M. Dennis, H.M. Tufo, R.D. Loft, Partitioning the cubed-sphere for Bluegene/l, in: PP04: 11th SIAM Conference on Parallel Processing for Scientific Computing, 2004.
[15] C. Schmidt, M. Parashar, W. Chen, D.J. Foran, Engineering a peer-to-peer collaboratory for tissue microarray research, in: CLADE'04: 2nd Int. Workshop on Challenges of Large Applications in Distributed Environments, 2004.
[16] F. Dehne, T. Eavis, A. Rau-Chaplin, Parallel multi-dimensional ROLAP indexing, in: CCGrid 2003: IEEE Int. Symp. on Cluster Computing and the Grid, 2003, pp. 86–93.
[17] G. Breinholt, C. Schierz, Algorithm 781: Generating Hilberts space-filling curve by recursion, ACM Trans. on Mathematical Software 24 (2) (June 1998) 184–189.
[18] A.R. Butz, Alternative algorithm for Hilbert's space-filling curve, IEEE Trans. on Computers (April 1971) 424–426.
[19] D. Hilbert, Über die stetige Abbildung einer Linie auf ein Flächenstück, Mathematische Annalen 38 (1891) 459–460.
[20] G. Jin, J.M. Mellor-Crummey, SFCGen: A framework for efficient generation of multi-dimensional space-filling curves by recursion, ACM Trans. on Math. Software 31 (1) (March 2005) 120–148.
[21] B. Moon, H. Jagadish, C. Faloutsos, J.H. Saltz, Analysis of the clustering properties of the Hilbert space-filling curve, Knowledge and Data Eng. 13 (1) (January 2001) 124–141.

[22] G. Peano, Sur une courbe, qui remplit toute une aire plane, Mathematische Annalen 36 (1890) 157–160.

[23] A.R. Butz, Convergence with Hilbert's space-filling curve, J. of Computer and System Sciences 3 (2) (May 1969) 128–146.

[24] Y. Matia and A. Shamir, A video scrambling technique based on space filling curves, in: Advances in Cryptology—CRYPTO'87, pp. 398–417.

[25] J.J. Bartholdi III, P. Goldsman, Vertex-labeling algorithms for the Hilbert spacefilling curve, Software—Practice and Experience 31 (5) (May 2001) 395–408.

[26] J.K. Lawder, Calculations of mappings between one and n-dimensional values using the Hilbert space-filling curve, Birkbeck College, University of London, Technical Report JL1/00, August 2000.

[27] D. Moore, Fast Hilbert curve generation, sorting, and range queries, http://www.caam.rice.edu/~dougm/twiddle/Hilbert/, 1999.

[28] H.V. Jagadish, Linear clustering of objects with multiple attributes, in: SIGMOD'90: ACM Int. Conf. on Management of Data, 1990, pp. 332–342.

[29] I. Kamel, C. Faloutsos, Hilbert R-tree: An improved R-tree using fractals, in: VLDB 94: Int. Conf. on Very Large Databases.

[30] C. Schmidt, M. Parashar, Enabling flexible queries with guarantees in P2P systems, IEEE Internet Computing 8 (3) (2004) 19–26.

[31] W.G. Nulty, Geometric searching with spacefilling curves, PhD dissertation, Georgia Institute of Technology, Atlanta, GA, 1993.

[32] F. Gray, Pulse code communication, US Patent Number 2,632,058, March 17, 1953.

[33] C. Hamilton, Compact Hilbert indices, Dalhousie University, Faculty of Computer Science, Technical Report CS-2006-07, July 2006.

[34] J.A. Fill, S. Janson, The number of bit comparisons used by quick sort: an average-case analysis, in: 15th ACM–SIAM Symp. on Discrete Algorithms, 2004, pp. 300–307.