分割統治法を用いたヒルベルトソート

今村 安伸[†1]　樋口 直哉[†1]　篠原 武[†2]

概要：多次元空間上の点をヒルベルト曲線に沿って並べることをヒルベルトソートという．ヒルベルトソート順で近いもの同士は，比較的良いクラスタをもたらす．本論文では，素朴な手法ではデータの次元数やヒルベルト曲線の次数に対して指数時間を要するヒルベルトソートを線形時間で効果的に実装できることを示す．

キーワード：空間充填曲線，ヒルベルト曲線，フラクタル

Hilbert Sort Algorithm using Divide and Conquer Method

YASUNOBU IMAMURA[†1]　NAOYA HIGUCHI[†1]
TAKESHI SHINOHARA[†2]

Abstract: Hilbert curve is a kind of space filling curve. Sorting multi-dimensional data along a Hilbert curve is called *Hilbert sort*. Grouping data near in Hilbert sort order generates relatively good clustering. For example, Hilbert R-tree, which is one of the most efficient spatial indexes, is constructed by Hilbert sort. Naive implementation of Hilbert sort, which first generates Hilbert curve, requires exponential time with respect to the data dimension and the order of Hilbert curve. In this paper, we present a linear time implementation of Hilbert sort by reconsidering a fast Hilbert sort algorithm introduced by Tanaka in 2001, which does not generate Hilbert curve.

Keywords: Space filling curve, Hilbert curve, fractal

1. はじめに

ヒルベルト曲線は，空間充填曲線の一つである．2 次元のヒルベルト曲線の例を図 1 に示す．多次元空間のデータをヒルベルト曲線に沿って並べることをヒルベルトソートと呼ぶ．ヒルベルトソート順で近いもの同士は，比較的良いクラスタリングをもたらす．Hilbert R-tree[4]では，ヒルベルトソートを用いて空間索引を構築する．

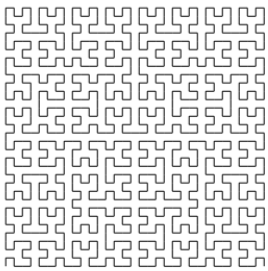

図 1　ヒルベルト曲線

素朴なヒルベルトソート実装において，ヒルベルト曲線を生成する際，次元数に対して指数オーダーの計算時間を要する．本論文では，2001 年の田中晶の論文[5]で発案された，ヒルベルト曲線を生成しない多項式時間ソート法を再考し，さらに高速化した実装を紹介する．

2. 原理

n 次元 1 次のヒルベルト曲線は，*n*-1 次元 1 次のヒルベルト曲線を新しい次元軸でつないだものとして定義できる．この定義において，*n* 次元 1 次のヒルベルト曲線は，図 2 の様に *n* ビットグレイコードとみなせる．

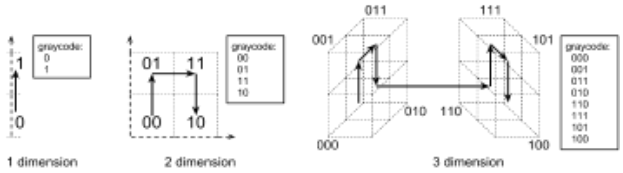

図 2　*n* 次元 1 次のヒルベルト曲線

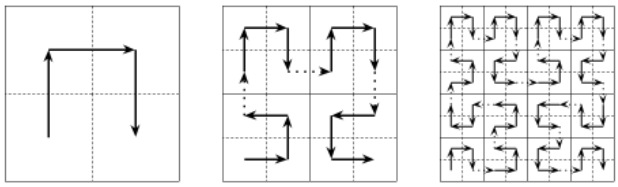

図 3　*m* 次のヒルベルト曲線

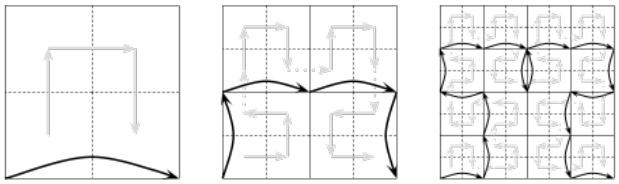

図 4　単位部分空間における始点と終点

このとき，各単位部分空間に対して再帰的に回転しながら同様の分割を行うことで，*n* 次元 *m* 次のヒルベルト曲線を定義できる．2 次元の場合の例を図 3 に示す．単位部分空間におけるヒルベルト曲線の回転については，図 4 に示す様に，始点と終点の移動に着目することで説明できる．また，これを 3 次元に拡張したものを図 5 に示す．このとき，各単位部分空間においての最初の切断軸は上位次数によって一意に決定するが，それ以降の切断軸については自由に決めることができる．

†1 九州工業大学情報工学府情報工学専攻
Graduate School of Computer Science and Systems Engineering,
Kyushu Institute of Technology

†2 九州工業大学情報工学研究院
Department of Artificial Intelligence,
Kyushu Institute of Technology

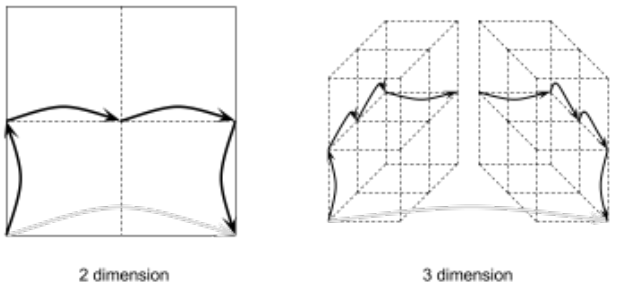

図 5　3 次元での始点と終点

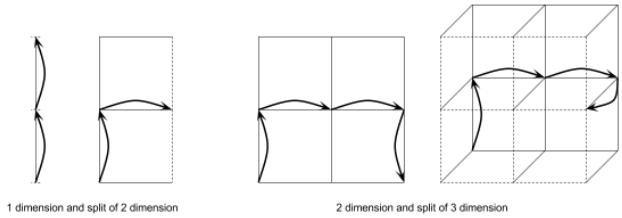

図 6　低次元と切断された高次元の相違

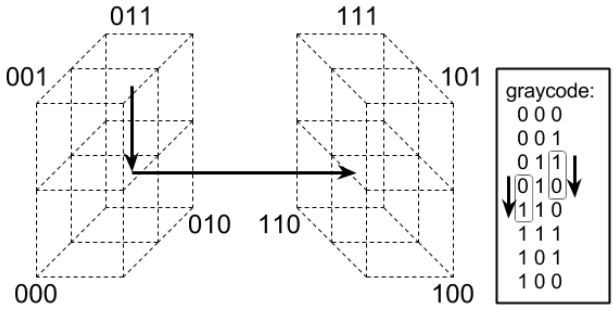

図 7　単位部分空間の遷移とグレイコードの遷移

ここで気を付けなければならないことは，1 次のヒルベルト曲線は対称の中心で切断してもそれぞれの部分空間もまた対称を保ち続けるが，2 次以降では完全な対称とはならない．切断面と行き来をする単位部分空間だけが，切断面方向を向くように回転する．このことが，非対称性をもたらす．2 次元と 3 次元の場合を図 6 に示す．

n 次元 1 次のヒルベルト曲線を *n* ビットグレイコードと見なすと，グレイコードにおいてのビット変化が，切断面に対応していることが分かる．全体の始点と終点を除いて切断面は各単位部分空間に 2 つずつ存在しているが，それは図 7 に示すように，グレイコードの前後に対応する．グレイコードの前後において変化したビット番号が，切断方向の次元番号となる．2 つの切断に隣接しているが，より切断順の早い方向が回転方向となる．これは，グレイコードにおいて，より高いビットが回転方向と一致する．そして，低い方のビットは常に最下位ビットであり，最後の切断次元軸である．始点と終点に限っては最下位ビットの変化方向しか持たず，この点でもグレイコードと *n* 次元 1 次のヒルベルト曲線とは対応付いている．グレイコードが訪問順を 1 ビット下位方向へビットシフトしたものと自身との排他的論理和から求められることを考えると，高い方のビットは，訪問順の 2 進数表現において最下位ビットから連続する同じ符号のビット数と対応付いていることが分かる．これを利用することで，回転方向についても容易に求めることができる．

これで訪問順と回転方向は分かったが，部分空間における始点の座標解決が未だである．2001 年の田中晶の論文では，部分空間の座標区間および始点と終点の座標を逐一シミュレートする方法が用いられていた．しかしながら始点と終点は必ず角にあるため，最小値であるか最大値であるかのいずれかである．そして異なる部分空間同士はそこにいたるまでの空間切断時に順番の比較が完了してしまうため，比較対象となるオブジェクト同士は常に現在までの切断において同じ区間に所属しており，次の切断時に用いる参照値は，該当軸のうちで未比較の最も高い 1 ビットだけとなる．この 1 ビットがどちらの値であるかによって，ヒルベルト順の前後が決まるが，0 が先で 1 が後なのか，それとも 1 が先で 0 が後なのかは，そこに至るまでの反転によって決まる．この反転状態を保持すればよく，状態としては 1 つの次元軸ごとに 1 ビットだけを保持すれば良い．

では，どの様に軸に対しての反転が起こるのだろうか．切断後，切断面と行き来をする単位部分空間だけが非対称になることは既に述べた．それ以外の単位部分空間に関しては，始点も終点も対称性があるが，隣り合う 2 つの単位部分空間の始点と終点は同一であるため，始点が切断面と接している場合は始点だけが，終点が切断面と接している場合は終点だけが，非対称となっている．これは切断ごとに再帰的に起こるが，1 つの始点が複数の切断面と接することは無いため，どの単位部分空間に対しても常に非対称性は 1 つの切断からのみ発生する．更に，この時のもう一方は，常に最後の切断面に接している．つまり，この最後の切断面に接する点については最後の切断が起こるまでの間，例外なく常に対称性が保たれている状態にある．そこでこの点を基準に切断時の反転を考えると，切断面の次元軸の反転がまず起こっており，次に順番も反転していることが分かる．対称性を仮定しているため，更に順番を反転すると，今度は次に切断する次元軸に対して座標を反転させることになる．このことから，現在の切断面と次の切断面の 2 つの次元軸に対して反転処理を行えばよい．

次に，最後の切断を考える．最後の切断は，切断面を挟んで端から端へと横切るのではなく，常に切断面に面した位置を保持し続けることになる．これはその他の次元軸が常に外側にあり，端から端へと切断面を通り過ぎるのと対称的である．故に，他と正反対に，前者で反転し，後者では反転しない．

さて，ここまで最後の切断面の接点についてシミュレートしてきたため，最後の切断面の後であれば始点にあたるためそのままでよいが，最後の切断面の前であれば終点を指しているため，辿り着いた単位部分空間の方向，つまりこの後の最初の切断次元軸について反転を行う必要がある．

これにて，全ての反転処理が完了し，ヒルベルト曲線を実際に描くことなく，データ点の存在する部分空間のみ掘り下げてシミュレート可能となる．

3. 実装

各切断面をクイックソートのピボットになぞらえて実装を行った．データ件数 N に対してクイックソートの最悪計算量は $O(N\log N)$ではなく $O(N^2)$となるが，本論文のヒルベルトソートの場合は座標系のビット数が有限であり，データ件数はともかく領域の座標域としてはそれぞれが常に完全に半分になり続けるため，次元数 n，次数 m に対して $O(Nnm)$となり，データ件数に対しての二乗オーダーを回避できていることが分かる．

また，再起処理の際，次元数に比例する各次元の反転情報をコピーすると，$O(Nn^2m)$となってしまうため，再起呼び出し前に 0〜2 ビットの状態書き換えを行い，再起呼び出しから戻った後に undo 処理を行う様に実装した．状態書き換えは常に 2 ビット以内であるため，書き換え処理も undo 情報もどちらも次元数に比例することなく一定であることが保証される．

4. 疑似コード

疑似コードを図 8 に示す．

HilbertSort クラスが存在し，その中に外部から呼び出す main 関数が存在する．main から内部の sub 関数を呼び出し，sub 関数内では separate 関数の呼び出しと，sub 関数の再起呼び出しを行う．

HilbertSort クラスは，次元数を示す dim，現在の次数での最初の切断軸番号を示す baseAxis，現在の次数を示す currentBit，現在の始点の反転状況を示す bits，の 4 つのメンバ変数を持つ．このうち，bits のみが $O(n)$のサイズを持ち，それ以外は $O(\log n)$または $O(\log m)$のサイズを持つ．現実的な運用では，$O(\log n)$および $O(\log m)$は現代のマシンでの一般的な計算の最小単位である 32bit や 64bit に収まってしまうため，n や m が小さすぎることでの恩恵は得られない．（約 6 億次元を超える場合や，約 20 億次を超える場合には 32bit で収まらなくなる可能性が出てくる）

これに関連して，全体の計算量が $O(Nn^2m)$ではなく $O(Nn^2m(\log n+\log m))$であるという議論があると思うが，ポインタ表現自体のデータサイズが $O(\log N)$に依存することを考えれば多くのインメモリソートアルゴリズムのオーダー表記の説明が付かないため，それらに倣うこととする．（n および m は，データサイズに比例するため，この議論において N と条件は同じであるとみなせる）

main 関数ではメンバ変数の初期化を行っている．オーダーの関係上, begin および end はポインタのポインタとして実装すべきで，並び替え時の swap 処理が次元数や次数に比例することを避ける必要がある．

```
Class HilbertSort:
  Var dim : int
  Var baseAxis : int
  Var currentBit : int
  Var bits : bitset
  Function separate(begin, end, axis, flag):
    If flag が真:
      [begin, end)区間に対し，各要素の axis 次元目の第 currentBit ビットが(true 区間/false 区間)になる様に並び替え, [begin, mid)が true 区間, [mid, end)が false 区間になる様に定めた mid を戻り値として返す
    Else:
      [begin, end)区間に対し，各要素の axis 次元目の第 currentBit ビットが(false 区間/true 区間)になる様に並び替え，[begin, mid)が false 区間，[mid, end)が true 区間になる様に定めた mid を戻り値として返す
  Function sub(begin, end, axis, bf, cnt):
    If [begin, end)の区間の要素数が 1 以下:
      Return
    Var mid = separate(begin, end, axis, bits[axis])
    Var nxt := (axis + 1) % dim
    If nxt=baseAxis:
      If currentBit が 0 なら:
        Return
      currentBit を 1 つ減らす
      Var old := baseAxis
      baseAxis := (old+dim+dim-(bf ? 2 : cnt + 2)) % dim
      bits[baseAxis]をフリップ
      bits[axis]をフリップ
      sub(begin, mid, baseAxis, false, 0)
      bits[axis]をフリップ
      bits[baseAxis]をフリップ
      baseAxis := (old+dim+dim-(bf ? cnt+2 : 2)) % dim
      sub(mid, end, baseAxis, false, 0)
      baseAxis := old
      currentBit を 1 つ増やす
    Else:
      sub(begin, mid, nxt, false, bf ? 1 : cnt + 1)
      bits[axis]をフリップ
      bits[nxt]をフリップ
      sub(mid, end, nxt, true, bf ? cnt + 1 : 1)
      bits[nxt]をフリップ
      bits[axis]をフリップ
  Function main(begin, end, dim_, order_):
    dim := dim_
    currentBit := order_ -1
    bits のサイズを dim とし，要素を全て false にする
    baseAxis := 0
    sub(begin, end, 0, false, 0)
```

図 8　ヒルベルトソートの疑似コード

main で初期化されたメンバ変数は sub 内において変化することがあるが，その場合には必ず，変更と undo は対になっていて，木構造の呼び出し構造に対して，親の処理には依存するが，兄弟の処理には依存しない書き方になっていることが分かる．

sub 関数および separate 関数の引数および戻り値は全て，ポインタのポインタ，またはブール値または軸番号や軸数しか示しておらず，次元数に対しても次数（座標値のビット数）に対しても log にしか依存せず，現実的な使用においては通常用いられる 32bit や 64bit で充分に間に合うことが分かる．（メンバ変数同様，約 6 億次元や約 20 億次を超える場合に再考の余地あり）

bits は次元数に依存したサイズを持つが，再起呼び出し時のコピーは行わず，再起呼び出し前後の書き換え処理と undo 処理によって変化し，その変化を行うフリップ処理は次元数に依存しない．

なお，「axis 次元目」や「bits[axis]」の様な表記においての添え字は 0 オリジンであり，「第 currentBit ビット」の様な表記においての「第 0 ビット」は最下位ビットを示している．

5. 実験

8 次元の符号なし 3bit 整数（つまり，各次元ごとに 0〜7 の範囲）の全領域に 1 つずつデータ点を配置した 16777216 件のデータについて，ソート実験を行った．ソートされた後の全ての隣り合う 2 点のマンハッタン距離を測った所，全ての距離が 1 となった．このことをもって，本論文をもとにした実装は正しく動いていると推定される．

6. まとめ

ヒルベルトソートの高速な実装を実現できた．

7. 今後の課題

今回，主記憶上で高速に動作するクイックソートをベースとした実装を行った．これは 2 つの要素でのソートを可能とするため，マージソートも可能であり，その場合の計算量もオーダー表記上は変わらないが，マージソートにおいてはマージ前の各 2 つの配列それぞれの各連続する 2 要素において，何ビット目まで同一であったのかと，その時の反転状況等を残しておくことが可能である．これを使って，オーダーはそのままになるが，より高速に実装可能な余地がある．AB…と CD…をマージする時，A と B の比較で何ビット目までが同じだったか，C と D の比較で何ビット目までが同じだったかを残してあるものとする．この後，A と C を比較し，その際に何ビット目までが同じだったかが分かることになるが，この 3 つの比較においての同じだったビット数のうちの最も少ないビット数までは，同じであることが保証されることが分かる．その時の状態も記憶しておけば，その状態から比較を開始することが可能である．また，ソートの性質上，この時に同ビットとなる平均的な長さはそれなりに長くなり，そこからスタートして異なるビットが現れるまでの平均的な長さはそれなりに短くなる．この性質によって，比較処理の大部分を削減できる余地がある．また，この時にデータ量 $O(nm)$に対して $O(n)$ のデータを残せば良いことになるため，トレードオフ上も問題ない可能性は高い．

また，2011 年の田島圭の論文において，Hilbert R-tree に対して，少量の追加データをマージすることで，全体を作り直すよりも若干の高速化に成功した内容が発表されているが，これについても上記マージソートの比較途中データを保持しておくことで，議論の余地はまだ残されている様に思う．

参考文献

[1] Hilbert D.Uber die stetige Abbildung einer Linie auf ein Flachenstuck, Math.Ann., 38, 459-460 (1891)
[2] A.R. Butz, Alternative algorithm for Hilbert's space-filling curve, IEEE Trans. on Computers (April 1971) 424-426.
[3] S.Kamata, A.Perez and E.Kawaguchi, A computation of Hilbert's curves in N dimensional space, IEICE, J76-D-II, 797-801, (1993)
[4] I.Kamel, C.Faloutsos, Hilbert R-tree: An Improved R-tree Using Fractals, the 20th International Conference on Very Large Data Bases(VLDB), September 1994.
[5] A.Tanaka, Study on a fast ordering of high dimensional data to spatial index, Master Thesis, Kyushu Institute of Technology, 2001 (in Japanese).
[6] K.Tashima, Study on efficient method of insertion for spatial index structure by using Hilbert sort, Master Thesis, Kyushu Institute of Technology, 2011 (in Japanese).