Program Promenade
連載

六角形の組合せ

寺田　実（電気通信大学情報通信工学科）
terada@ice.uec.ac.jp

今回取り上げるのは，2001 年 11 月にはこだて未来大学で行われた，アジア地区予選函館大会の問題の E, Beehives である．

正六角形をしきつめた平面を蜂の巣に見立て，そこに女王蜂が産卵して，あるパターンを作り出す．それを 2 人の学生が独立に記録し，その結果を照合するのが目的である．

産卵の際，女王蜂はひと筆書きの要領で移動し，その軌跡にはループや交差はない．しかし，学生が記録する時点ではもはや女王蜂の軌跡は分からず，学生の判断で六角形で構成される図形をアルファベットの並びで記述することになる．その記述には後述するような冗長性があり，同じ図形でも異なる記述となる場合があるので，照合が必要とされるのである（図 -1）．

具体的な記述の方式は，まず図形の中から始点となる六角形を 1 つ選び，そこからの軌跡を，次への隣接 6 方向（a から f までの英字で表現．図 -2）の並びで表現する．したがって，n 個の六角形からなる図形に対して，$n-1$ の長さの表現英字列が得られる．図 -1 の左端の図形を例とすると，左端の六角形を始点として，右（a），右上（b），右下（f）の経路をたどることによって abf の表現が得られる．

この記述にはいくつかの自由度があり，図形とは多対一の対応関係となる：

- 始点選択が一意ではない（もちろんひと筆書きのできない点は始点にはできない）．図 -1 の (2) は，(1) を逆順にたどったものである．
- 図形を記録する際の回転方向の基準が決まっていないので，図形を回転したものは同一視する．図 -1 の (3) がその例である．
- 途中経路の選び方に自由度がある．図 -1 の (4) がその例である．

なお，鏡像（うらがえし）については別の図形として扱う．

アルゴリズムとしては単純で，あまり工夫の余地もないが，六角形の回転など，注意を要するところもある問題である．

なお，問題のスケールとしては，記述の長さ ≤100 という規模の制限が与えられている（つまり構成要素の正六角形の個数 ≤101 である）．

■アプローチ

まず，この問題に対して，2 つのアプローチを考える．

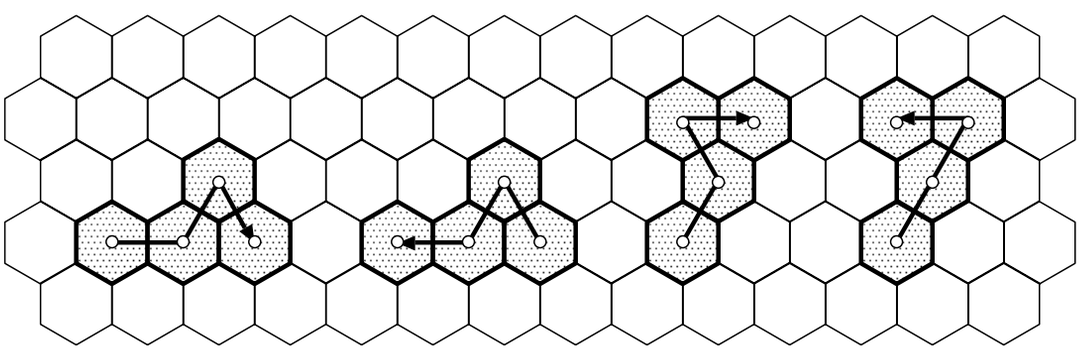

図 -1　同一図形に対するさまざまな表現

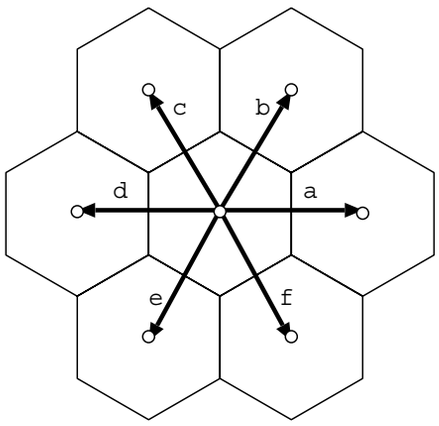

図 -2　6 方向の表現

アプローチ 1 は，入力文字列を直接記号的に操作して，合同性を判定しようとするもので，考えられる記号操作としては以下があろう：

・文字の置き換え

たとえば, a → b, b → c, ...f → a の置き換えによって，軌跡の方向を 60 度だけ反時計方向に回転できる．図 -1 の (1) は，この操作によって (3) と一致する．

・文字列の順序の逆転

始点と終点の交換が解決できる．正確には，文字列を逆順にしたのち，前項の置換を利用して 180 度回転させる必要がある．図 -1 の (1) は，この操作によって (2) に一致する（abf → fba → ced）．

しかし，図 -1 の (4) のような経路選択の差を記号的に解決するのは困難であり，このアプローチは残念ながら放棄せざるを得ないであろう．

アプローチ 2 は，文字列による記述が表現する図形を，もっと処理しやすい形式で表す方法である．ここでの「処理しやすさ」とは，個々の構成要素（六角形）の位置が分かるということで，具体的には 2 次元平面の点（正六角形の中心）の集合が適当である．文字列表現を点集合で表現できれば，あとは回転と平行移動を適用して一致を確かめるだけでよい．こちらを採用することにしよう．

残っている検討事項は以下のとおりである：

(1) 点の表現方法

正六角形をそのまま直交座標系で扱うと，座標値に $\sqrt{3}$ が出現することになる．計算機でこれを扱うには浮動小数点とせざるを得ず，誤差の問題が発生して，比較などが面倒になる．

座標値を整数で表現できれば誤差の心配がないうえ，（今回は直接の関係はないが）2 次元配列の添字として使える，という利点もある．

座標値を整数にするための変換は，1 次変換であることが望ましい．そうでないと，移動操作の関係が保存されなくなるからである（たとえば，a の移動に続いて c の移動を行ったものが b と同じ点に移るかどうか）．

ここでは，具体的な 1 次変換の例として，

$$T_1 = \begin{pmatrix} 2 & 0 \\ 0 & 2\sqrt{3}/3 \end{pmatrix}, T_2 = \begin{pmatrix} 1 & \sqrt{3}/3 \\ 0 & 2\sqrt{3}/3 \end{pmatrix}$$

などが考えられる（変換結果は**図 -3**）．

(2) 平行移動

1 次変換を採用したおかげで，ベクトルの加算で簡単に済む．

(3) 回転操作

図形を回転させるには，どの段階で行うかについて 2 通りの選択肢がある：

- 記号列の段階で回転させる（前述の，文字の置き換え）
- 座標表現で回転させる．前述の T_1, T_2 のもとでの反時計まわり 60 度の回転行列は以下の通りである．

$$R_1 = \begin{pmatrix} 1/2 & -3/2 \\ 1/2 & 1/2 \end{pmatrix}, R_2 = \begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix}$$

(4) 図形の比較

この問題では一対比較が要求されているが，それにも 2 通りの方法が可能である：

- 片方を変換して，もう片方に到達するかどうかを判断する．
- 両方を「標準形式」に変換して比較する．

「標準形式」は，(1) 合同な図形は同じ標準形式になる，(2) 合同でない図形は異なる標準形式になる，の 2 点を満たす必要がある．

まず，点の間に大小関係を定義し，それをもとにして点の列の間に辞書順（lexicographic order）の大小関係を定義する．これは上記の要請を満たす関係である．

点の大小関係は，座標値を用いれば簡単である．

また，点集合から一意的な点の列を求めるには，点の大小関係を用いて点集合をソートし，さらにそのうちの最小の点が原点になるように平行移動する．これで，ある特定の回転角に対する基本形が求まるので，

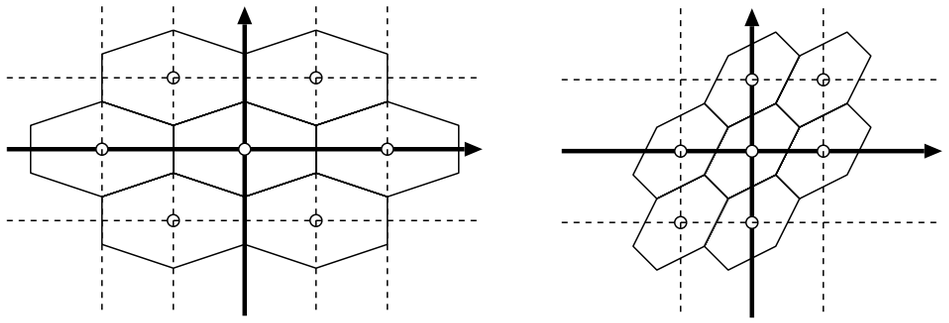

図 -3　T_1（左），T_2（右）による変換結果

さまざまな回転角での基本形を辞書順で比較して，そのうちで最小のものを選べば，それをもってその図形の標準形式とできる．

以上をまとめると，選択肢として残っているのは以下の 3 点である：

- 座標の変換方式
- 回転操作をどの段階で行うか
- 一対比較の方法

■実装方法の設計

記述言語としては，C 言語を用いることとする．まず，データの具体的な実装方法を設計する必要がある．

(1) 図形の記号列による記述

C の通常の文字列を利用する．

(2) 点（x, y 座標値の組）の表現

- 構造体の利用（メンバとして x, y 座標を持つ）．
- 1 つの整数に（$x+\alpha y$）の形でコーディング．

などいろいろ方法があるが，ここでは後者を用いることとする．

- x, y の値の範囲は限られている．
- 整数は 1 ワードに納まるのでメモリ割当を必要としない．
- 比較，平行移動などの操作が高速に（2 つの座標を同時に）実行できる．

などがその根拠である．

ただ，1 つ問題点があって，それは座標値 x, y が負になった場合のデコードである．基本的には商と剰余の計算によるのだが，剰余が 正，負の両方にわたるようにするには条件判断が必要になる．

ここでは，あらかじめ x, y にある値を加算しておき，正となるようにしてからエンコードすることにした．デコードの際には，剰余計算ののちその値を減じてやればよい．

このエンコーディングでのもう 1 つの利点として，エンコード結果が必ず正になるので，0 を特殊目的に利用することが可能になる．

```
#define MAXPT 102
#define X0 1000
#define Y0 1000
#define OFF 10000
#define P(x,y) (((y)+Y0)*OFF+(x)+X0)
#define X(p) ((p)%OFF-X0)
#define Y(p) ((p)/OFF-Y0)
typedef int point;
```

(3) 座標平面での回転

座標の 1 次変換方式としては前述の T_1 を用いることにする．まず，6 つの隣接正六角形への移動のベクトルを，配列として持つ．

```
point dir[] = {
  P(2,0)-P(0,0), P(1,1)-P(0,0),
  P(-1,1)-P(0,0), P(-2,0)-P(0,0),
  P(-1,-1)-P(0,0), P(1,-1)-P(0,0)
};
```

さらに，原点を中心とした 60 度回転（反時計回り）を行う関数（R_1 に対応）を用意した．

```
point rot_pt(point p)
{
  int x = X(p), y = Y(p);
  return P((x - y*3)/2, (x + y)/2);
}
```

(4) 図形（点の集合）の表現

点（のエンコード結果である整数）の配列とする．可変長なので，終端として特別の値 0 を使う．その長さ

は（終端も含めて）最大長 MAXPT＝100＋1＋1 なので，領域のサイズとしてはこれで固定した．

```
typedef point pattern[MAXPT];
```

次に，以下にあげるような各種操作関数を定義した．プログラムは一部省略する．

(1) 記号列の回転

```
void rot_layout(char layout[]);
```

(2) 記号列から図形の生成

```
int gen_pat(char layout[],
            pattern pat);
```

(3) 図形の回転

```
void rot_pat(pattern pat);
```

(4) 図形の平行移動

```
void reposition_pat(pattern pat)
{
  int i;
  point p0;

  qsort((void *)pat, length_pat(pat),
      sizeof *pat, &cmp_pt);
  p0 = pat[0] - P(0, 0);
  for(i=0; pat[i] != 0; i++){
    pat[i] -= p0;
  }
}
```

図形の左下隅を原点に移動する．ここで，qsort は C の標準ライブラリ関数で，同一サイズのデータの並びを，こちらの用意した大小比較関数（第 4 引数）に従ってソートする，C ではめずらしい高階関数である．

(5) 図形の辞書順比較

```
int cmp_pat(pattern p1, pattern p2)
{
  int i, d;
  for(i=0; p1[i] != 0; i++){
    d = p1[i] - p2[i];
    if(d != 0) return d;
  }
  return p2[i];
}
```

(6) 図形の標準化

```
void norm_pat(pattern pat)
{
  int i;
  pattern work;
  reposition_pat(pat);
  copy_pat(pat, work);
  for(i=1; i<6; i++){
    rot_pat(work);
    reposition_pat(work);
    if(cmp_pat(pat, work) < 0){
      copy_pat(work, pat);
    }
  }
}
```

(7) 図形のコピー

```
void copy_pat(pattern p1, pattern p2);
```

(8) 点の図形への包含判定

```
int member(point p, pattern pat);
```

前章末で述べた選択肢に基づき，2 つのプログラムを示すことにする．

(1) プログラム 1

記号列の段階で回転操作して一対比較をするもの

```
int cmp_layout3(char l1[], char l2[])
{
  int i;
  pattern p1, p2;
  int len1 = gen_pat(l1, p1);
  int len2 = gen_pat(l2, p2);
  if(len1 != len2) return FALSE;
  reposition_pat(p1);
  for(i=0; i<6; i++){
    reposition_pat(p2);
    if(cmp_pat(p1, p2) == 0)
      return TRUE;
    rot_layout(l2);
    gen_pat(l2, p2);
  }
  return FALSE;
}
```

(2) プログラム 2

記述から図形を作り，双方を標準形式に変換して比較するもの

```
int cmp_layout2(char l1[], char l2[])
{
  pattern p1, p2;
  int len1 = gen_pat(l1, p1);
  int len2 = gen_pat(l2, p2);
```

```
  if(len1 != len2) return FALSE;
  norm_pat(p1);
  norm_pat(p2);
  return cmp_pat(p1, p2) == 0;
}
```

■発展－箱詰めパズルのピース生成

プログラミングコンテストの問題は以上ですべてであるが，ここからは一歩進んだ問題にチャレンジしてみよう．

前章までで基本図形を並べてできる図形の合同判定ができるようになったので，その応用として，箱詰めパズルのピースの生成を考えてみることにする．

たとえば本連載第5回のペントミノのピースは正方形5個からなる図形であり，全部で12種類が存在する（ピースは裏返しの同一視も含まれる点がこれまでとは異なる）．第5回のプログラムでは，ピースの形状はあらかじめデータとして与えていた．

ここでは，比較的少数の正六角形からなるピースの生成を試みる．

前章までの合同判定では，ひと筆書きできる図形だけを対象としてきたが，これからはそれだけでは不十分である．つまり，記号列による表現は使えないので，座標表現だけを使っていくことにする．

基本的な生成のアイディアは，要素図形の個数の少ないものを出発点とし，それに要素図形を1つ付加したものを生成してはそれまでとの重複をチェックするという，generate and test のアプローチをとる．

■再帰による深さ優先の生成

まず，単純な再帰による深さ優先の生成を考えよう．引数は以下のとおり：

- 現在の要素図形の個数 (n)
- 目標とする要素図形の個数 (to)
- 現在生成中の図形 (current)

処理内容は，以下の2つの場合に分かれる：

(1) 目標とする個数の図形が生成できていれば，その標準形を求め，それが未登録のものであれば登録する（今回の標準形では鏡像も考慮する）．

(2) さもなければ，current を構成する個々の点に対して，その隣接点のそれぞれを（current に含まれていなければ）current に追加しては再帰呼出を行う．

```
#define MAXPIECE 1000
pattern pieces[MAXPIECE];
int np = 0;
void piece(int n, int to, pattern cur)
{
  if(n == to){
    pattern work;
    int i;
    copy_pat(cur, work);
    norm_pat_flip(work);
    for(i=0; i<np; i++){
      if(cmp_pat(work, pieces[i]) == 0)
        return;
    }
    copy_pat(work, pieces[np]);
    np++;
  } else {
    int i, j, k;
    point p;
    for(i=0; i<n; i++){
      for(j=0; j < ndir; j++){
        p = cur[i] + dir[j];
        for(k=0; k<n; k++){
          if(cur[k] == p)
            goto used;
        }
        cur[n] = p;
        cur[n+1] = 0;
        piece(n+1, to, cur);
      used:
      }
    }
  }
}
```

まったく工夫がないので，同じ図形を何度も生成し，指数オーダの時間を要するのが欠点である．候補の重複を排除して，処理の高速化を図ろう．

■改良した幅優先の生成

上記の単純再帰バージョンでも，基本図形の個数（再帰呼出のレベル）ごとに「発見図形リスト」を管理しておけば，重複は防止できる．

また，単純再帰バージョンは木の深さ優先探索になっているが，これを幅優先にすると，呼出の深さを節約できる．一般に幅優先の探索では，候補の集合を明示的に管理する必要があるが，ここでは，上述のレベルごとの「発見図形リスト」がまさにそれに相当するので余計なコストがかかるわけではない．

ここでは，幅優先探索でレベルごとの「発見図形リ

スト」を 1 本の配列で済ませる巧妙な方法を紹介する．これはリスト構造のごみ集めの一種であるコピー GC（ルートと呼ぶセルから指されているセルのみをある領域にまとめる）で用いられる，Cheney のアルゴリズム [1] の応用である．

主な変数は以下のとおり：

- 発見図形リスト

```
pattern pieces[MAXPIECE];
```

- 要素個数ごとのリスト中の開始位置

```
int phead[MAXN];
```

- リスト内の位置を保持するポインタ

```
int from, to;
```

準備として，発見図形リストの先頭に，「たね」となる図形を格納しておく．

その後の処理は，リストの from の位置から図形を取り出して，それを 1 つだけ拡張した図形候補（複数）を生成し，未登録のものであればリストの to の位置に格納していく（**図 -4**）．

本来のごみ集めであれば，この処理を繰り返すうちに未到達のセルがなくなり，from が to に追いついて全体の処理を終了するのだが，このプログラムではいくらでも大きな図形を作り続けるので，目標となるサイズの図形の生成が完了したら処理を終了する．

```
void piece(int nlimit)
{
  int n;
  int from, to, to0;
  int j, k, l;
  point c;

  pieces[0][0] = P(0,0);
  pieces[0][1] = 0;
  n = 1;
  from = 0;
  to = 1;

  phead[1] = 0;
  phead[2] = 1;
  for(n=1; n<nlimit; n++){
    to0 = to;
    while(from < to0){
      for(j=0; j<n; j++){
        for(k=0; k<6; k++){
          c = pieces[from][j] + dir[k];
          if(member(c, pieces[from]))
            continue;
          /* 番兵 */
          copy_pat(pieces[from],
```

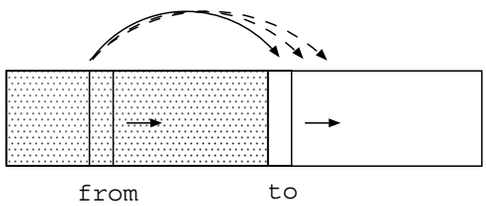

図 -4　**Cheney** のアルゴリズムによる生成

```
                   pieces[to]);
          pieces[to][n] = c;
          pieces[to][n+1] = 0;
          norm_pat_flip(pieces[to]);
          for(l = to0;
              cmp_pat(pieces[l],
                      pieces[to])!=0;
              l++);
          if(l == to) to++;
        }
      }
      from++;
    }
    phead[n+2] = to;
  }
}
```

正六角形だけでなく他の正多角形から構成されるピースも同様にして作ることができる．

そのためのプログラムの修正点は，隣接点の位置の配列，単位角度の回転の関数，裏返しの関数などの定義をプログラム本体とは別にしておけば，それらを差し替えて生成アルゴリズム自身は共用することが可能である．

ただ，正三角形の場合にはやや状況が複雑である．正方形，正六角形と違って正三角形で平面を埋めた場合には，上向き／下向きの 2 種があるためである．「向き」によって，隣接する点の位置が 6 近傍のうちの 3 近傍となる．

この「向き」の判定がポイントだが，原点における向きをたとえば上向きと固定しておけば，個々の向きの判定は，位置（座標値）によって簡単に求めることができる．生成アルゴリズムでは，それをもとに，隣接点の配列を取り替えて使えばいいだけである．

参考文献

1）Cheney, C.J.: A Nonrecursive List Compacting Algorithm, CACM, Vol.13, No.11, pp.677-678 (Nov. 1970).

（平成 14 年 9 月 6 日受付）