Program Promenade

連載

組木細工

田中 哲朗（東京大学情報基盤センター）
ktanaka@tanaka.ecc.u-tokyo.ac.jp

■問題の定義

今回は，1998 年に行われたアジア予選東京大会の問題 B「Lattice Practices」を取り上げる（http://www.isse.kuis.kyoto-u.ac.jp/acm-japan/problems/98/html/node2.html 参照）．10 枚の長方形の板をぴったり組み合わせて，**図 -1** のような正方形の格子 4×4 個を作るという問題である．

それぞれの板には**図 -2** のように等間隔に 5 個の切り込みが入れられている．

切り込みの深さは 2 種類ある．格子は，縦方向の板 5 枚と横方向の板 5 枚を組み合わせて作るが，25 カ所の交叉点は，縦方向が浅い切り込みのときは横方向が深い切り込み，縦方向が深い切り込みのときは横方向が浅い切り込みという 2 種類の組合せしか許されない．

互いに異なる 10 枚の板が与えられたときに，格子を完成させる「本質的に」異なる組合せが何通りあるかを答えるプログラムを作成するのが今回の問題である．

入力は 1 問 1 行で，サンプル入力として提示された

```
10000 01000 00100 11000 01100 11111 01110
11100 10110 11110
10101 01000 00000 11001 01100 11101 01110
11100 10110 11010
END
```

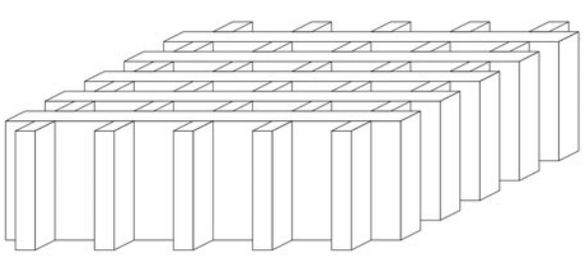

図 -1　格子の完成図

のように，10 枚の板が深い切り込みを 0, 浅い切り込みを 1 として表現される．出力として，これに対する本質的に異なる組合せの数（この場合は 40 と 6）を返すプログラムを作成することが求められている．入力の最後は END という文字列だけからなる行で終わる．

「本質的に」同一であるかどうかは以下のように定義される．

まず，板の表と裏には区別がないとする．**図 -3** のように左右対称な板は，裏返しても同一と見なされる．

図 -4 の 1 番上の完成図を例に考える．図の左側は縦方向の板の切り込みの深さを表し，右側は横方向の

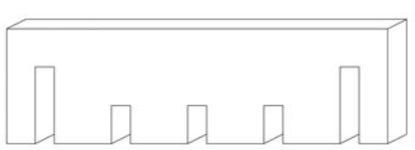

図 -2　板の例

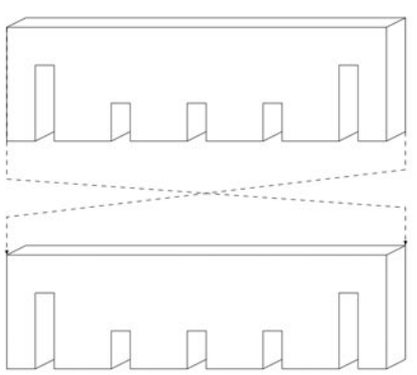

図 -3　板の対称性

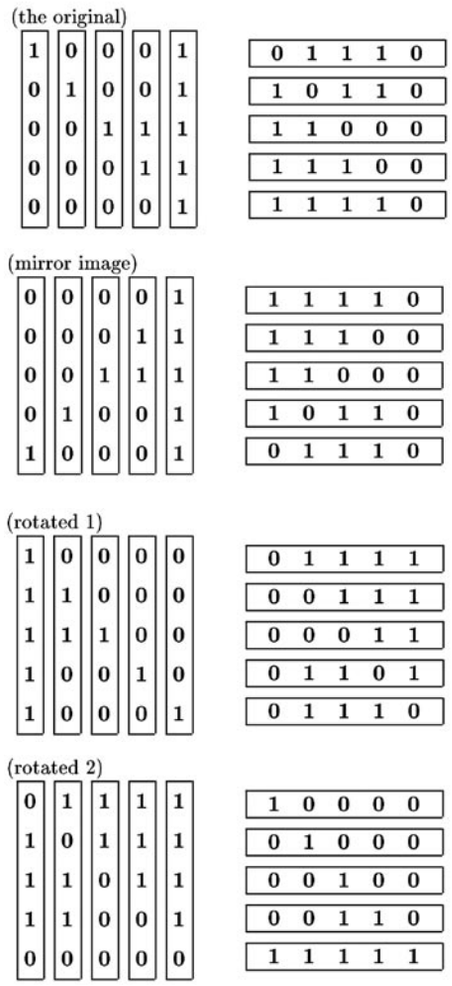

図-4　同一と見なされる組合せ

板の切り込みの深さを表す．図の左側と右側を重ね合わせると，必ず 0 と 1 が対応しているのが分かるだろう．

図 -4 の 2 番目のような鏡像を求めると，それも同じ 10 枚を使って作った解になっている．この操作は板の組み替えを必要とするが，ここでは元の組合せと「本質的に」同一であると定める．

また，図 -4 の 3 番目のように，元の完成図から平面上で 180 度回転したものも明らかに解である（組み直す必要がない），これも同一であるとする．図 -4 の 4 番目のように，元の完成図から上下反転して縦横を入れ替えたものも（組み直す必要がない），同一であるとする．

図 -4 のように組合せを表現するとき，ある組合せから，この 3 種類の置換の積によって，8 種類の異なる表現が得られることが分かる．

■準備

今回も解答プログラムは C++ 言語を用いて作成する．問題は 5 個の切り込みがある板を 10 枚組み合わせるとしているが，ここでは問題を一般化して n 個の切り込みがある板を $2n$ 枚組み合わせるプログラムを作成する．

板の集合を表す型 Boards を，ここでは以下のように STL (Standard Template Library) を使って定義する．

```
typedef vector<string> Boards;
```

入力のための演算子 >> は以下のように定義する．

```
istream& operator>>(istream& is,
                    Boards& boards){
  string s;
  is >> s;
  if(s != "END"){
    int n=s.length();
    boards.push_back(s);
    for(int i=1;i<n*2;i++){
      is >> s;
      boards.push_back(s);
    }
  }
  return is;
}
```

呼び出し側は boards を空にして呼び出す．終了を表す文字列「END」を読み込んだ際には，boards を変更せずに返している．呼び出し側は boards の大きさが 0 かどうかをチェックすれば，「END」を読み込んだかどうかを判断できる．

準備のために，裏返した板を生成する関数 reverse と板が対称（裏返して自分自身と重なる）かどうかを判定する関数 isSymmetric を以下のように定義しておく．

```
string reverse(string const& b){
  return string(b.rbegin(),b.rend());
}
bool isSymmetric(string const& b){
  return b == string(b.rbegin(),b.rend());
}
```

板を置ける位置は $2n$ カ所ある．横方向と縦方向を区別して，それぞれに 0 から $n-1$ の番号をつけるのが自然だが，データ構造を簡潔に保つために，**図 -5**

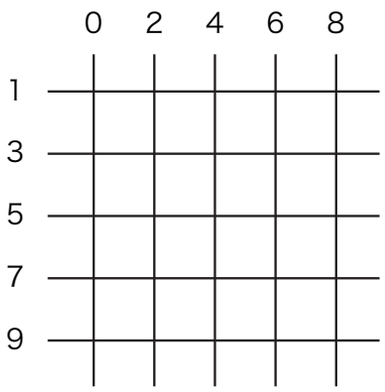

図-5　板の位置の指定

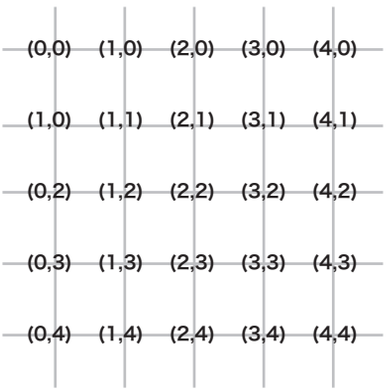

図-6　交叉点の座標の表現

のように偶数の番号は縦方向，奇数の番号は横方向と，縦横交互に番号をつけることにする．

板を途中まで置いた状態を表現するには，それぞれの位置にどの板が置かれているかを表現するだけで十分であるが，ある状態で板を空いている位置に置けるかどうかの判定を容易にするため，ここでは交叉点全体の状態を表す以下の Mat のようなデータ構造を用いる．

```
typedef vector<vector<char> > Mat;
```

交叉点の座標は，**図-6** のように x 座標，y 座標の対で扱う．交叉点の状態は，char 型で表す．以下のような意味を持たせることにする．

'0' 縦方向の板がこの交叉点を通っていて，その切り込みが '0'，あるいは横方向の板がこの交叉点を通っていて，その切り込みが '1'，あるいはその両方．

'1' 縦方向の板がこの交叉点を通っていて，その切り込みが '1'，あるいは横方向の板がこの交叉点を通っていて，その切り込みが '0'，あるいはその両方．

' ' 縦方向の板も横方向の板もこの交叉点を通っていない．

準備のために，'0' と '1' を反転するための関数 alt を定義しておく．' ' のときは ' ' を返すことにしておく．

```
char alt(char c){
  if(c=='0') return '1';
  if(c=='1') return '0';
  return ' ';
}
```

ある位置 p に板 s を置けるかどうかをチェックする関数 canSetPos を以下のように定義する．

```
bool canSetPos(Mat const& mat,int p,
               string const& s) {
  int n=mat.size();
  if((p%2)==0){ // 縦方向の板
    int x=p/2;
    for(int i=0;i<n;i++){
      if(mat[x][i]==' ') continue;
      if(mat[x][i]!=s[i])
        return false;
    }
  }
  else{
    int y=p/2;
    for(int i=0;i<n;i++){
      if(mat[i][y]==' ') continue;
      if(mat[i][y]!=alt(s[i]))
        return false;
    }
  }
  return true;
}
```

ある位置 p に板 s を置いたときに mat を書き換える関数 setPos も同様に書ける．ここではプログラムの掲載は省略する．

板を途中まで置いた状態を表現するには，交叉点の状態以外にも，どの板が残っているか，どこの位置が空いているかも持っておく必要がある．これを表すためのクラス State を定義する．板を置く順番を固定すれば，どこの位置が空いているかを持つ必要はないが，後での拡張を考えてこの情報を保持することにする．

bool 型の vector の boardUsed で，どの板が

使われたかを管理し，posUsed でどの位置が使われたかを管理する．また，Mat に対する canSetPos, setPos のラッパーメソッドも定義してある．これらは，板そのものではなく，板の番号 i と裏返して使うかどうかを表す isReverse を引数として持つ．

```
struct State{
  int n;
  Boards const& boards;
  Mat mat;
  vector<bool> boardUsed;
  vector<bool> posUsed;
  State(int n,Boards const& boards)
    :n(n),boards(boards),
     mat(Mat(n,vector<char>(n,' '))),
     boardUsed(n*2,false),
     posUsed(n*2,false){
  }
  bool canSetPos(int p,int i,
                 bool isReverse) const{
    if(isReverse)
      return
        ::canSetPos(mat,p,
                    reverse(boards[i]));
    else
      return ::canSetPos(mat,p,boards[i]);
  }
  void setPos(int p,int i,bool isReverse){
    if(isReverse)
      ::setPos(mat,p,reverse(boards[i]));
    else
      ::setPos(mat,p,boards[i]);
    boardUsed[i]=true;
    posUsed[p]=true;
  }
};
```

■簡単な解法

この問題の簡単な解法としては，以下のアルゴリズムが考えられる．

1. 板を置く位置の順番はあらかじめ決めておく．
2. 途中まで板を置いた状態で，次の位置に置ける板を置いて 3 に進む操作を繰り返す．全部の板を試し終わったらバックトラックする．
3. 置けた板が最後の板のときは，解の個数を 1 増やしてバックトラックする．
4. 置いた状態で 2 に進む．

この解法でほぼ正解なのだが，同一と見なされる組合せを重複して数えてしまう点が問題となる．

同一な解を重複して数えないためには以下のような方法が考えられる．

1. 重複して数えたものから，同一解の排除を行った場合の数が計算できるなら同一解の排除を行わなくてもよい．
2. 板の置き方に関して制約（ただしその制約によって一般性は失わないようなもの）を与えて同一の局面が現れないようにする．
3. ある解が求まったら，その同一解の集合を解集合に加える（集合として）．
4. 板を置いた状態に関して正規形（canonical form）を求めて，正規形のみを解の個数として数える．

最初の方法は，この問題に関してはたまたま有効である．前述のように，ある板の組合せ方に対して同一だがクラス Mat としての表現が異なるものは一般に 8 通りできる．同じ板が 2 回以上使われることがないという条件があるので，これら 8 通りの中に同一のものは現れない．したがって，同一局面の排除を行わずにプログラムを書いて，後で 8 で割ればよいことが分かる．コンテスト本番でこの問題に出会ったら，この方法でさっさと解くことをお勧めするが，ここでは，より実行時間の少ないプログラムに改良することを考えて，他の方法も検討することにする．

2 番目の方法は，許される置換の数が少ないときは有効な場合が多い．たとえば，この問題でも縦横入れ替えしかなかったら，「板 0 は必ず縦で使う」という制約条件を入れるだけで重複解を除くことができる．しかし，この問題のように置換の種類が多い場合は，正しい制約条件を過不足なく書くのは難しそうなので，ここでは考慮しない．

3 番目の方法は，ある解が求まったときに，それを置換して得られる 8 通りの組合せからなる集合を解として登録するというものである．この方針でも，プログラムは簡単に書けるが，解集合を管理する点が面倒である．集合の代表元としての正規形を定義しておけば，解にたどり着いた際にそれが正規形になっている場合だけ数えればよい．これが 4 番目の方法になっている．

正規形というと，論理式の積和標準形や，多項式の展開などのように，「項の書き換え規則が定義されていて，それ以上の書き換えが行えなくなったかたち」

であり，「停止性，合流性の議論が必要」なことが多いが，この手の問題では「ある状態と同一な状態の数は有限でかつ列挙は簡単」という性質があるので，状態の間での全順序関係が定義できれば，集合中の順序が最小のものを正規形と定義できる．以降は，この4番目の方法に従って，プログラムを作成することにしよう．

この方針に従って，Mat の要素がすべて埋まった際に，正規形かどうかを判定する関数 isCanonical を定義する．y 方向にスキャンしていった結果得られる1次元文字列の文字列としての辞書式順序を順序関係に用いる．準備のために，鏡像変換を行う mirror，平面上の180度回転を行う rotate1，上下反転して縦横の入れ替えを行う rotate2，および，辞書順で Mat を比較する lessThan も定義してある．

```
Mat mirror(Mat const& mat){
  int n=mat.n;
  Mat newMat(mat);
  for(int j=0;j<n;j++)
    for(int i=0;i<n;i++)
      newMat[i][j]=mat[i][n-1-j];
  return newMat;
}
Mat rotate1(Mat const& mat){
  int n=mat.n;
  Mat newMat(mat);
  for(int j=0;j<n;j++)
    for(int i=0;i<n;i++)
      newMat[i][j]=mat[n-1-i][n-1-j];
  return newMat;
}
Mat rotate2(Mat const& mat){
  int n=mat.n;
  Mat newMat(mat);
  for(int j=0;j<n;j++)
    for(int i=0;i<n;i++)
      newMat[i][j]=alt(mat[j][i]);
  return newMat;
}
bool lessThan(Mat const& s1,
                    Mat const& s2){
  int n=s1.size();
  for(int j=0;j<n;j++)
    for(int i=0;i<n;i++)
      if(s1[i][j]=='0' && s2[i][j]=='1')
        return true;
      else if(s1[i][j]=='1' &&
              s2[i][j]=='0')
        return false;
  return false;
}
bool isCanonical(Mat const& mat){
  for(int i=1;i<8;i++){
    Mat cmpState= mat;
    if((i&1)!=0)
      cmpState=mirror(cmpState);
    if((i&2)!=0)
      cmpState=rotate1(cmpState);
    if((i&4)!=0)
      cmpState=rotate2(cmpState);
    if(lessThan(cmpState, mat))
      return false;
  }
  return true;
}
```

ここまで準備ができたところで，図-5の番号順に置くプログラムを書くことにする．図-5のように，交互に番号をつけているので，2枚目の板を置く時点から置ける板の種類に制約がかかり，探索局面の減少につながっている．0から4までを縦方向，5から9までを横方向の番号としてしまうと，6枚目の板を置くまではまったく制約なく置けるので，非対称の板だけからなる場合は，最低でも ${}_{10}P_5\times 2^5=967680$ 局面のチェックが必要になる☆1．

プログラムは以下のようになる．

```
int solveRec(State const&,int);
int tryRec(State const& state,
           int d,
           int b,bool isReverse){
  State copyState(state);
  copyState.setPos(d,b,isReverse);
  return solveRec(copyState,d+1);
}
int solveRec(State const& state,int d){
  int n=state.n;
  if(d==n*2){
    if(state.isCanonical()) return 1;
    return 0;
  }
  int count=0;
  for(int i=0;i<n*2;i++)
    if(!state.boardUsed[i]){
      if(state.canSetPos(d,i,true))
        count+=tryRec(state,d,i,true);
      if(!isSymmetric(state.boards[i]) &&
         state.canSetPos(d,i,false))
        count+=tryRec(state,d,i,false);
    }
  return count;
}
```

☆1 もちろん，これでもコンテストはクリアできる程度の実行時間だろう．

これで，サンプル入力に対しては，61793 局面の探索が必要だった．

■効率化

探索中の正規形チェック

正規形かどうかのチェックは板を 10 枚置き終わる前でもできる．途中でも完成時に正規形とは成り得ないことが分かれば，それ以上その局面を探索する必要はない．

たとえば，1 枚目を置いて，

```
1 ? ? ? ?
0 ? ? ? ?
0 ? ? ? ?
0 ? ? ? ?
1 ? ? ? ?
```

となった形を考える．これに対して rotate2 を実行した結果は，

```
0 1 1 1 0
? ? ? ? ?
? ? ? ? ?
? ? ? ? ?
? ? ? ? ?
```

となるが，「?」に何が入っても rotate2 の結果の方が元の配置よりも辞書式順序で小さくなる．したがって，元の形は正規形とは成り得ないことが分かる．

solveRec の最初に正規形になる可能性が残っているかどうかのチェックを入れると，枝刈りの効果が期待できる．そのためには，Mat クラス中に，「?」になっている部分を埋めていって，正規形になる可能性があるかどうかを判定する関数を用意する必要がある．

実は lessThan の定義中の

```
else if(s1[i][j]=='1' && s2[i][j]=='0')
```

の行を

```
else if(s1[i][j]!='0' && s2[i][j]!='1')
```

と変更すれば，lessThan は空白の集合が '0', '1' の組合せのどれに確定しても辞書式順序で小さくなるときに true を返すという定義になり，isCanonical が空白を含む Mat に対しても，上の定義に従う答えを返すようになる．

したがって，solveRec の先頭で，

```
if(!state.isCanonical())
  return 1;
```

と入れておけばよい（ついでに d==n*2 の場合の正規形のチェックは外す）．

この改良により，サンプル入力に対する探索局面数は 16708 局面と減り，かなりの改善がみられた．

置く順番の工夫

探索問題を解く場合，探索木の浅いところで分岐数を減らすことが総探索局面数を減らすのに有効な場合が多い．

置いていく順番を位置によって固定する場合，縦方向，横方向と交互に置くのは良い置き方だが，途中まで置いたとき，最も枝分かれの少ない位置から決めていくという方法も可能である．また，この問題はすべての板を使用するという制約があるので，逆に置く位置の限られている板を置くことにするのも可能である．

このような方針でプログラムを作成してみると，サンプル入力に対しては，探索局面数は 2381 局面とさらに減少した．手元の環境では探索中の正規形チェックを入れたプログラムと比較して実行時間は半分程度に減った．

この枝刈りの効果は切り込みの数 n を増やしていったときに効果があると期待されるので，n=7 の問題を 1 問作ってみて解かせたところ，元のプログラムと比較して，探索局面数は 200 分の 1 程度に減り，実行時間も 10 分の 1 程度に減った．期待通りの結果といえる．

探索プログラムでは，このように枝刈りを効率的に行うことにより，実行時間が桁違いに小さくなることは結構ある．しかし，プログラミングコンテストの問題を作成する際に，このような工夫をしないと正解にならない（実行時間制限にひっかかる）ように問題を設定するのは，なかなか難しい．

コンテストの際には，計算量のオーダの違う場合を除けば，探索プログラムは枝刈り等の工夫はそこそこにして，正しい簡潔なプログラムを書くことに集中した方がよいだろう．効率を良くするための工夫は，コンテスト終了後にゆっくり楽しめばよい．

（平成 16 年 7 月 15 日受付）