Program Promenade

連載

計算機用ジグソーパズル

和田 英一（IIJ 技術研究所）
wada@u-tokyo.ac.jp

■計算機用はどこが違う

2001 年度国内予選の問題 C は，ジグソーパズルであった（http://www.fun.ac.jp/icpc/index-j.html）．ジグソーパズルと聞いても，まぁそう身を乗り出してこないでよい．通常のジグソーパズルは，辺が凹凸になっている 100 から 1000 オーダーのピースをうまく嵌め合わせて，一幅の名画を完成するものだ．それにはピースに描いてある図柄を頼りにする．まずピースをコーナー用のもの，周辺用のもの，その他に分類し，その他のものはさらに大体の色で再分類しておく．コーナー用と周辺用をつなぎ，大枠を作る．昔の「誉れの軍旗」みたいなものができる．その後順次内部に向かって接続していくのが大方の手順であろう．そこでその裏をかく嫌らしいジグソーパズルもあると聞く．たとえば両面に図柄を描く．コーナーがないように周囲を円形にするなど．

人工知能の研究として，計算機に（灰色）ジグソーパズルをやらせたという報告も以前に読んだ気がする．しかし，プログラミングコンテストは人工知能の研究ではないので劇的に簡単化し，今回の問題ではピースの辺上に図 -1 のように文字を書いておき，それを凹凸の代用にしている．

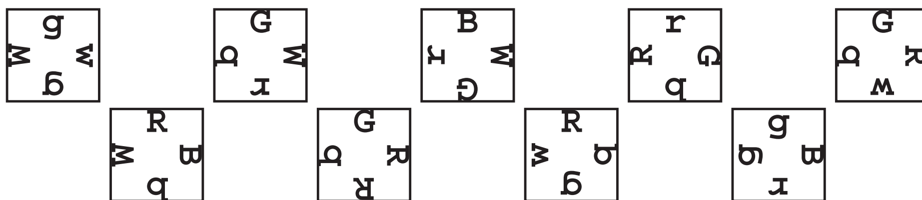

図-1

文字は R，G，B，W，r，g，b，w のいずれかで，図 -2 のように，同じ文字の大文字小文字同士が向き合うとき，ピースはつながるという約束である．どのピースにも各辺に文字があるから，コーナー用，周辺用という特別のものはない．

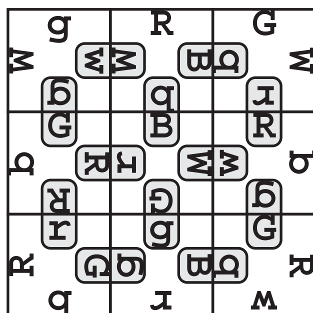

図-2

Program Promenade Program Promenade

そこで問題はあるピースの組（9 個）が，図 -1 の組なら（左から）gwgW RBbW GWrb GRRb BWGr Rbgw rGbR gBrg GRwb のように与えられたとき，3 × 3 の形で完成したパターンが何通りあるかを求めるものである．問題では，90 度，180 度，270 度回転して自分自身と同じになるピースはなく，同じピースが 2 つあることはない．図 -3 に示すようにある完成図は，90 度ずつ回転してもやはり完成図なので，完成パターンの合計は 4 の倍数になるはず．

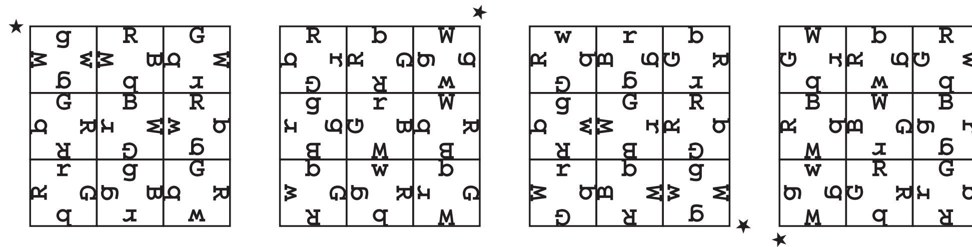

図 -3

これはペントミノなどのプログラムと同じで，しらみつぶしで片端から探していくより方法はない．枝刈りもうまくいきそうもない．

しらみつぶしの一般的なアルゴリズムは次のように書ける．

```
準備 すべてのピースに番号をつけ，未使用としてバッファに入れておく．
test(n) 手続き (n はすでに使用済のピースの数 )
    現時点で未使用のピースについて番号順にとり，
    目的にかなっているか調べ，目的にかなっていれば：
        そのピースを既使用にする．
        すべてのピースを使っていたら，
            適切な処理を行う．
            そうでなければ test(n+1) を実行する．
        そのピースを未使用にする．
test(0) を実行する．
```

ここで番号順にとるとは，ピースにまた方向性などサブ構造があれば，それも含めて番号順にとるということである．イメージとしてはテストする枝に順に入っていき，袋小路に達すると戻る．この戻り動作をバックトラックとかバックアップという．

プログラムの技としては，合わさり具合をどうチェックするか，ピースの回転をどう実現するかなどがある．昨今ではバックトラックは学習必須の基本アルゴリズムになっているから，工夫の余地は周辺にしかないらしい．

R と r，G と g，B と b，W と w が嚙み合うかどうかをみるには，経験や勘から思いつくことかもしれないが，

```
R, G, B, W, r, g, b, w に
0, 1, 2, 3, 7, 6, 5, 4 を対応させる．
```

そうすると，大文字，小文字が合うときは，このコーディングでは向かい合う文字の組の和が 7 になる．ピースを置く位置とその各辺に図 -4 のように番号を与えると，ピースを 0 から番号順に置こうとするときの条件は表 -1 のようになる．プログラムでは，表 -1 通りに忠実にコーディングしたが，プロムナード常連の石畑君の意見は，個々にコーディングするより，実行時間は多少かかっても規則として書いた方が虫が入りにくいという．そうかも知れぬ．また田中君の説は，図 -4 のようにピースを 1 行目，2 行目，3 行目と順に置くより，0, 1, 3, 4, 2, 5, 6, 7, 8 の順に置く，つまり 2 辺で隣接するピースを早めに置くと早めに枝の数が減り，結果として test を呼ぶ回数が 4117 回から 2569 回になるという．なるほど．

辺上の文字は図から分かるように上→右→下→左の順に並んでいる．上辺の文字を知るには配列の第 0 要素をとる．このピースを左に 90 度回転すると，上辺にくる文字は配列の第 1 要素，180 度回転では，

第2要素になるから辺の番号に回転回数を足した要素でその文字が分かる．あとは和が4以上になったときの対策で，以下のプログラムでは，自動バックトラックの方は4で割った剰余をとることで，手動バックトラックの方では要素を2回繰り返し並べることで解決している．

0 3 0 1 2	0 3 1 1 2	0 3 2 1 2
0 3 3 1 2	0 3 4 1 2	0 3 5 1 2
0 3 6 1 2	0 3 7 1 2	0 3 8 1 2

図-4

ピース	条件
0	無条件
1	0の1 + 1の3 = 7
2	1の1 + 2の3 = 7
3	0の2 + 3の0 = 7
4	1の2 + 4の0 = 7 かつ 3の1 + 4の3 = 7
5	2の2 + 5の0 = 7 かつ 4の1 + 5の3 = 7
6	3の2 + 6の0 = 7
7	4の2 + 7の0 = 7 かつ 6の1 + 7の3 = 7
8	5の2 + 8の0 = 7 かつ 7の1 + 8の3 = 7

表-1　ピースが置ける条件

■自動バックトラック

バックトラックで全解探索するには，バックトラック機構を備えている言語を使えば楽勝である．Prologもそういう言語だったが，第五世代コンピュータのプロジェクトが終わってからははやらないので，Sussmanたちの教科書「計算機プログラムの構造と解釈」[1)] にあるAmb評価器を使ってみよう．Amb評価器にはそのための手続きambとrequireがあり，式(amb e0 e1 ...)では上から実行してきたときは，e0, e1, ...から任意のものを選んで，下に実行を進めていく．下からバックアップしてきたときは，e0, e1, ...でまだ選ばれずに残っているものがあれば，その中から任意のものを選んで，再び下に実行を進めていく．残っているものがなければ，上にバックアップしていく．(amb)は無条件にバックアップする．

式(require p)はpが真なら下へ実行を進め，偽ならバックアップする．たとえば3, 5, 7で割った剰余がそれぞれ2, 3, 2なる整数（答は23）を探すには，

```
(let ((n (an-integer-from 1)))
  (require (and (= (remainder n 3) 2)
                (= (remainder n 5) 3)
                (= (remainder n 7) 2)))
  n)
```

と書く．(an-integer-from n)はnから順に整数をambを使って発生するもので，requireとともに次のように定義する．

```
(define (require p)
  (if (not p) (amb)))

(define (an-integer-from n)
  (amb n (an-integer-from (+ n 1))))
```

似たようなものに次の定義の(an-integer-between low high)があり，ここではそれを使う．

```
(define (an-integer-between low high)
  (require (not (> low high)))
  (amb low (an-integer-between (+ low 1) high)))
```

各ピースに0から8の番号をつけ，0から3のすべての方向について，組み合うかどうかテストする．ピース番号はiで，方向はjで制御する．ピースを置く場所も0から8と番号づけ，それぞれにどのピ

Program Promenade Program Promenade

ースがどの方向で置いてあるかをi-stack, j-stackが記憶する．ピースの情報はすでに文字を0から7の整数に変換し，リストのリストで記憶してある．

```
(define pieces
'((6 4 6 3) (0 2 5 3) (1 3 7 5) (1 0 0 5) (2 3 1 7)
;; g w g W   R B b W   G W r b   G R R b   B W G r
  (0 5 6 4) (7 1 5 0) (6 2 7 6) (1 0 4 5)))
;; R b g w   r G b R   g B r g   G R w b

(define i-stack '())
(define j-stack '())
(define count 0)

(define (test n)
 (define (fit)              ;fit(表-1の条件チェック)の定義
  (define (m a b c d)       ;fitの下請け手続き
   (define (l e f)          ;mの下請け手続き
    (list-ref (list-ref pieces (list-ref i-stack (- n e)))
          (remainder (+ f (list-ref j-stack (- n e))) 4)))
   (= (+ (l a b) (l c d)) 7))
  (cond ((= n 0) true)
        ((= n 1) (m 0 1 1 3))
        ... 表-1の通りなので省略
        ((= n 8) (and (m 5 2 8 0) (m 7 1 8 3)))))
 (let ((i (an-integer-between 0 8)))
  (require (not (member i i-stack)))
  (set! i-stack (cons i i-stack))
  (let ((j (an-integer-between 0 3)))
   (set! j-stack (cons j j-stack))
   (require (fit))
   (if (= n 8) (begin (permanent-set! count (+ count 1)) (amb))
       (test (+ n 1))))))
```

実行してみると

```
;;; Amb-Eval input:
(if-fail (test 0)
         count)

;;; Amb-Eval value:
12
```

少し説明が必要であろう．i-stack，j-stackはset!で設定されるが，バックトラックのとき，自動的に元の値に戻される．countは戻されると意味ないので，permanent-set!を使う．permanent-set!の後の(amb)は選ぶべき式が1つもないので，バックトラックを開始する．(if-fail (test 0) count)は(test 0)を実行し，バックトラックしてきたらcountを返すということ．

これは簡単に書け，快適に動くけれども，ものすごく遅い．全解探索に10分位かかり，コンテスト(Schemeは公用言語ではないし）では使えない．それで積極的にバックトラックするCのプログラムを書いてみる．

■手動バックトラック

バックトラックのプログラムは，自動バックトラックのプログラムを高水準言語で書き，それを展開するという手もある．参考文献2）にあるペントミノのプログラムはマクロを使う．Floydもそういうプロ

グラムを発表している[3]．だが直接書けないこともない．Dijkstra のエイトクィーンのプログラム[4] など，構造的プログラミングの例題として書かれているが，変数の代入とその戻しが，たとえば

```
begin SET QUEEN ON SQUARE[n,h]; n:= n + 1;
      if n = 8 then PRINT CONFIGURATION
         else generate;
      n:= n−1; REMOVE QUEEN FROM SQUARE[n,h];
end
```

のように忠実に対称的になっていて，いかにも機械的に展開したという気分にできている．こういう精神はぜひ受け継ぎたい．

```
#include<stdio.h>
int pieces[9][8]=
   {{6,4,6,3,6,4,6,3},{0,2,5,3,0,2,5,3},{1,3,7,5,1,3,7,5}, /* ピースの情報は */
    {1,0,0,5,1,0,0,5},{2,3,1,7,2,3,1,7},{0,5,6,4,0,5,6,4}, /*2 度繰り返して */
    {7,1,5,0,7,1,5,0},{6,2,7,6,6,2,7,6},{1,0,4,5,1,0,4,5}};/* 書いてある */
int j_stack[9],i_stack[9],count,p=0x1ff;

l(i,k)
{return pieces[i_stack[i]][j_stack[i]+k];}

m(a,i,b,j)
{return l(a,i)+l(b,j)==7;}

fit(n)                       /* 表 -1 の条件チェック */
{switch(n)
   {case 0:{return 1;}
    case 1:{return (m(1,3,0,1));}
        ... 表 -1 の通りなので省略
    case 8:{return (m(8,0,5,2)&& m(8,3,7,1));}}}

test(n)
{int i,j;
 for(i=0;i<9;i++)
 {if(p&(1<<i))
  {p^=(1<<i);i_stack[n]=i;
   for(j=0;j<4;j++)
   {j_stack[n]=j;
    if(fit(n)){if(n==8){/* 全部置けたら出力 */printout();count++;}else test(n+1);}}
   p|=1<<i;}}}

main ()
{count=0;test(0);printf("count= %d¥n",count);}
```

グローバル変数 p はピースの使用状況をビットで表している．(n==8) で使っている printout はコンテストのプログラムとしては要求されていないが，揃ったところを印字してみたくて書いた．

```
printout()
{printf(" %d   %d   %d¥n",l(0,0),l(1,0),l(2,0));
 printf("%d %d %d %d %d %d¥n",l(0,3),l(0,1),l(1,3),l(1,1),l(2,3),l(2,1));
 printf(" %d %d %d %d %d %d¥n",l(0,2),l(3,0),l(1,2),l(4,0),l(2,2),l(5,0));
 printf("  %d %d %d %d %d %d¥n",l(3,3),l(3,1),l(4,3),l(4,1),l(5,3),l(5,1));
 printf(" %d %d %d %d %d %d¥n",l(6,0),l(3,2),l(7,0),l(4,2),l(8,0),l(5,2));
 printf("%d %d %d %d %d %d¥n",l(6,3),l(6,1),l(7,3),l(7,1),l(8,3),l(8,1));
 printf(" %d   %d   %d¥n¥n",l(6,2),l(7,2),l(8,2));}
```

置けた結果は図-2なら

```
 6   0   1
3 4 3 2 5 3
 6 1 5 2 7 0
  5 0 7 3 4 5
 7 0 6 1 1 6
0 1 6 2 5 0
 5   7   4
```

のように印字する．3行目と5行目は上段の下と中段の上，中段の下と下段の上が同じ行にあるので少し見にくいが，なるべく正方形に近く出力したくてこうなった．

遅いプログラムも速いプログラムも私にはそれぞれ面白く，十分楽しんだ．

二本(ふたもと)の梅に遅速を愛す哉　　　蕪村

■ペントミノの場合

ついでだから，ペントミノもやってみよう．ペントミノは情報科学標準問題といわれるエイトクィーンやハノイの塔ほどはポピュラーでないが，一度は挑戦したくなる問題である．1辺1単位の正方形5個を辺のところで接続して作れる異なる形（ピース）は**図-5**の左のように12種類あり，それぞれ図のように名前がついている．

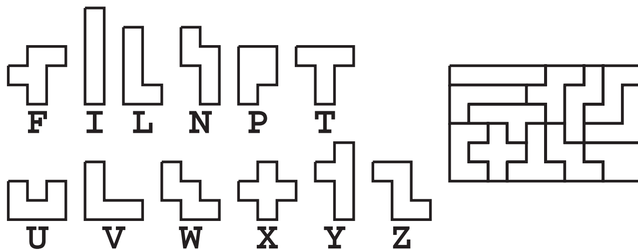

図-5

これを全部使ってたとえば縦6単位，横10単位の箱に同図右のように詰めることができる．それが何通りあるか全解探索する問題である．これについては昔Communications of the ACM誌のアルゴリズムの欄にプログラムが載った[2)]．またほとんど同じ頃，筆者のところで卒論をまとめた東山君が，当時日本橋にあったIBM計算センターにあったIBM7090のLispを使いペントミノのプログラムを書くと意気込んで始めたが，結局たとえば5個の正方形でできる形はこの12通りというようなところまでしか進まなかった．しかし日本でのほとんど最初のLispプログラムであろう．ずっと後だが，寺田君はPostScriptでプログラムを書き，プリンタでペントミノの全解探索を実行したそうだ．

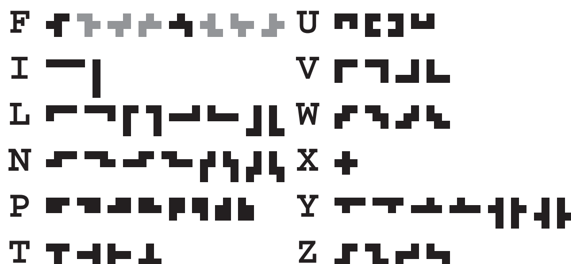

図-6

さて，各ピースは裏返したり，回転したりできるので，使い方には**図-6**のような可能性があり，そのそれぞれについてテストしなければならない．しかしそうするとジグソーパズルの問題で同じ解が4通りあったように，ここでも4倍の答えが得られてしまう．私の持っていたパズルの解答記録用のノートには，ピースXの中心を右上の1/4の領域に置くように記入せよと注意書きがあった．以下のプログラムではFのピースを8通りのうち，（灰色のものを除き）縦置きと横置きの2種類に制限して，解の冗長性を除去している．

私はペントミノのプログラムをLispでは何度も書いている．でもCでは実は初めてで，一応書いておいたのを寺田君が見ていられなくなったらしく，少し手を入れてくれたのが以下のプログラムである．

ピースのそれぞれの形は格子上の5つの点の位置で表現できる．一番上の列の一番左を起点とし，各点の起点からの相対位置で示す（**図-7**左）．先頭の0は不要だが，美的センスで残してあるようなもの．並べる箱は6×10なのだが，箱の外の検出には，すでに置いたピースとの衝突の検出と兼ねるいわゆる番兵方式をとりたいので，右と下に番兵領域をとり，10×16のサイズにした（**図-7**右）．

グローバルな配列avvecは，各要素に対応するピースがまだ手元に残っているときは1，使ったときは0となる．

testの引数nはすでに置いたピースの個数，sは前回置いたピースの起点の場所（したがって，空き地の探索の開始位置）を示す．

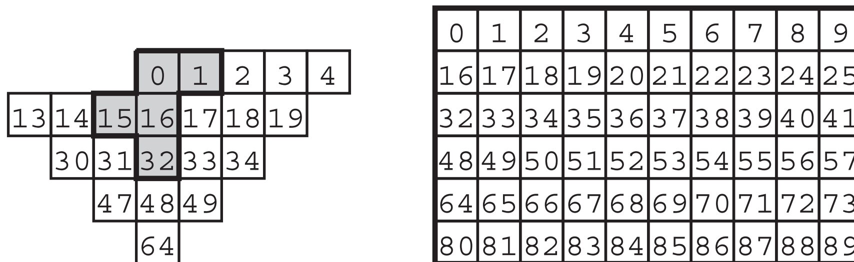

図-7

```
#include<stdio.h>
#define EMPTY '.'
#define DUMMY '@'
typedef int ss[5];
typedef int *ssp;
typedef ss *sssp;
ss a[12][8]={
{{0,1,15,16,32},{0,15,16,17,33}}, /*F*/
{{0,1,2,3,4},{0,16,32,48,64}},    /*I*/
... L,N,P,T,U,V,W,X,Y のデータは省略 ...
{{0,1,17,33,34},{0,1,16,31,32},{0,14,15,16,30},{0,16,17,18,34}}}; /*Z*/
int dirs[]={2,2,8,8,8,4,4,4,4,1,8,4};
int names[]={'f','i','l','n','p','t','u','v','w','x','y','z'};
char board[160];
int count=0,avvec[12];
printboard()
{int i,j;
 for(i=0;i<6;i++)
 {for(j=0;j<10;j++)printf("%c ",board[i*16+j]);
  printf("¥n");}}

test(n,s)
{int i,j,l,d,s1,s2,s3,s4;
 sssp p;
```

```
 ssp pp;
 while(board[s]!=EMPTY)s++;
 for(i=0;i<12;i++)if(avvec[i])
 {avvec[i]=0;l=names[i],d=dirs[i];p=a[i];
  for(j=0;j<d;j++)
  {pp=p[j];
   s1=s+pp[1];s2=s+pp[2];s3=s+pp[3];s4=s+pp[4];
   if((board[s1]==EMPTY)&&(board[s2]==EMPTY) &&
      (board[s3]==EMPTY)&&(board[s4]==EMPTY))
   {board[s]=l;board[s1]=l;board[s2]=l;board[s3]=l;board[s4]=l;
    if(n==11){count++;printf("count= %d¥n",count);printboard();}
    else test(n+1,s);
    board[s]=EMPTY;board[s1]=EMPTY;board[s2]=EMPTY;
    board[s3]=EMPTY;board[s4]=EMPTY;}}
  avvec[i]=1;}}

main()
{int i,j;
 for(i=0;i<12;i++)avvec[i]=1;
 for(i=0;i<160;i++)board[i]=DUMMY;
 for(i=0;i<6;i++)for(j=0;j<10;j++)board[i*16+j]=EMPTY;
 test(0,0);}
```

printboardはboardを

```
count= 1
i i i i i f f n z z
l l l l f f n n z v
l y y y y f n z z v
u u x y t w n v v v
u x x x t w w p p p
u u x t t t w w p p
```

のように出力する．これは図-5右と同じで，上のプログラムで最初に出てくる解である．このプログラムは，私のノートパソコンで全解を出力するのに6分50秒かかった．

■ピース位置のトライ構造データ

前述のCommunications of the ACM誌のプログラムは，上に説明したのとはかなり様子が違うので，簡単にそのやり方を説明したい．各ピースの形状の最上列の再左端からの相対位置の表し方は，上の場合と同じだが，データを次のように配置する．

```
( 0 (16 (32 (48 (64 .  1) ( 1 .  2) (47 .  2) (49 .  2) (15 . 10) (17 . 10)
                (33 . 10) (31 . 10))
            ( 1 (15 .  0) (17 .  4) (33 .  6) ( 2 .  7) (31 . 11))
            (31 (47 .  3) (15 .  4) (33 .  5) (30 .  7))
            (33 (49 .  3) (17 .  4) (34 .  7))
            (15 (14 .  5) (17 .  9))
            (17 (18 .  5)))
        (17 (18 (19 .  2) ( 1 .  4) ( 2 .  6) (15 . 10) (34 . 11))
            ( 1 (33 .  4) ( 2 .  4) (15 .  4))
            (15 (33 .  0) (14 . 10))
            (33 (49 .  3) (34 .  8)))
        (15 ( 1 (14 .  3) ( 2 .  3) (31 .  8))
            (14 (13 .  2) (30 . 11))
```

```
            (31 (47 .  3) (30 .  8)))
        ( 1 ( 2 ( 3 .  2) (18 .  6))))
    ( 1 ( 2 ( 3 ( 4 .  1) (19 .  2) (17 . 10) (18 . 10))
            (18 (19 .  3) (17 .  4) (34 .  7))
            (17 (33 .  5)))
        (17 (33 (49 .  2) (32 .  6) (34 . 11))
            (18 (19 .  3) (34 .  8))))))
```

この読み方だが，先頭の“(0”は「0を要素に持つピースが以下に並んでいる」の意味．次の“(16”は「16を要素に持つピースが並んでいる」ことを示す．下の方に“(16”と同じインデントで“(1”があるが，これは「16は要素に持たないが，1を要素に持つピースが以下にある」ということである．

また上に戻ると，“(0 (16 (32”があるが，これは当然「0, 16, 32を要素に持つ」ので，ピースの形が下に伸びていくことが分かる．このようにして“(64 . 1)”までくるが，「0, 16, 32, 48, 64を要素に持つピースは1という名前，つまりiである」といっている．その隣の“(1 . 2)”は，「48までは前と共通であり，最後が1の要素で，それは2，つまりlである」．

このように，前方の共通部分を括り出した木構造はトライ(trie)といい，このデータはそう作ってある(Knuthによると，trieはinformation re*trie*valが語源だそうだ[5])．

さて，この使い方だが，ピースを置こうとしたとき，アドレス順に探して，最初の空の位置をsとすると，s+16の場所が塞がっていたら，“(16”で始まるピース群は，もう調べる必要がない．s+1の場所が塞がっていたら，“(1”で始まるピース群も，調べないでよい．しかも探索の効率をあげるように，各分岐では，上の方ほど手下のピースが多くなるように並んでいる．

16の方が1より先にあるのは，16を含むピースは44個，1を含むピースは28個あったので，多い方を先に持ってきたのだが，最初に見つけた空き地の右の塞がっている方が下の塞がっているより多いのではないかという石畑君の意見に従った方が多少効率がよいかも知れぬ．

これだけ説明すれば，プログラムは書けるであろう．ペントミノのプログラムより，このデータ構造を構成するのが大変で，Lispのプログラムをいくつも書く羽目になった．

前述の文献では，マクロはアセンブリプログラムを入れ子処理のために，似たようなプログラム部分を展開するのに利用している．

最後に：リカーションを「頭山（あたまやま）」（この理解には落語の素養が必須）というように，水谷先生の計算機用語は超ユニークだが，水谷用語ではバックトラックを「後戻り出直し」という．

参考文献

1）Sussman, G.J., Abelson, H. and Sussman, J.（和田英一訳）: 計算機プログラムの構造と解釈 第二版，ピアソン・エデュケーション(2000).

2）Fletcher, J.G.: A Program to Solve the Pentomino Problem by the Recursive Use of Macros, Comm. ACM, Vol.8, No.10, pp.621-623 (Oct. 1965).

3）Floyd, R.W.: Nondeterministic Algorithms, J. ACM, Vol.14, No.4, pp.636-644 (Oct. 1967).

4）Dijkstra, E.W.: Notes on Structured Programming, in Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A.R.: Structured Programming, Academic Press (1972).

5）Knuth, D.E.: The Art of Computer Programming, Vol.3, 2nd Ed., Addison Wesley (1998).

（平成14年6月25日受付）