

## 課題：8 パズル ( 8 p u z z l e )

( 1 ) 縦型探索法のプログラムを作成して解決しなさい。

まず初めに、縦型探索の考え方を軽く説明する。

縦型探索とは、読んで字のごとく縦に探索していくことである。優先順位をつけてどの方向に優先的に動かしていくかを決めておき、終了状態になったら終了する。あるいは、以前と同じ状態が出てきたらひとつ前の状態の戻り次の優先順位の方向に向かっていくというものである。この探索方法では横型探索よりも早く終了する可能性もあるが、最短の方法を見落とす可能性も高い。

次に、今回作った縦型探索のプログラムにおいて自分で設定した事を説明する。まず初めに優先順位をあらかじめ上・左・右・下と決めておき（これは自分で決めておいた）『 ブランクの位置、

その時に動ける位置、 今あるブランクの位置から何通りブランクを動かせるか』を `tate10.dat` に記しておき、`fscan` 文で読み込んでいく。もしファイルが見つからない場合は『FILE GA ARIMASENN ( ファイルがありません )』と出力されるようにしておいた。ここで注意する点は、`dat` の中身はあらかじめ優先順位の順番で書いたのをこれを入れ替えてしまうとちゃんとした処理がなされない可能性があると言う事である。

又、ブランク等の数字の置かれている位置について説明しておく。今回は  $3 \times 3$  の空間で数字の移動が行われるが、一番上の行の左から  $0 \cdot 1 \cdot 2$ 、二番目の行の左から  $3 \cdot 4 \cdot 5$ 、一番下の左から  $6 \cdot 7 \cdot 8$  として設定されており、最終状態が格納されている `a[j]` と初期状態等が格納されている `b[h][m[h]][j]`、それに次の状態が今までに出てきた配列と同じでないかを確認するための配列 `c[n][j]` において、`j` 部分にこれらの数字が入れている。

さて、ここでやっとプログラムの流れについての説明に入れる。

まず初めに `tate10.dat` から情報を取り込み、次に配列 `a[j]`、`b[0][0][j]` にそれぞれ最終状態と初期状態を格納し出力する。次に、ブランクを探し出し、深さが 1 以上なら以前に出てきた状態全てと比べる（深さが 0 の時は他に配列が無いので確かめる必要も無い）。そしてそこで以前に同じ状態が無い（新しい状態である）事が分かれば、次の状態を配列 `b` と配列 `c` に格納し、最終状態と見比べ、`hyou = 9` になった時点で終了となる。`hyou` には、9 つのマスの内、最終状態といくつ同じ状態があるかを示す値が入れる。

もちろん `hyou` が 9 になればまったく同じ = 終了となるわけである。

また、出力する時は『その状態になるために使ったルールナンバー』と『最終状態といくつ同じものがあるか』を表示した。

行き詰まって動かせなくなると 1 つ前の状態に戻り別方向に進む訳だが、各ブランクにはそこから動かせる数が決まっており、その数が配列 `pat` に格納されている。そこまで行くともう 1 つ前の状態に戻る事になる。これを繰り返して最終状態まで進んでいくのである。この様に、縦型探索の場合は同じ状態が見つかるまでは優先順位の通りに階層をどんどん突き進んでいき、同じ状態が見つかったら一つ前の状態に戻る。

また `dat` を用いてプログラムを作ったのは、それを用いることにより、ルールを変更したい時には

この dat 部分だけを変えればよく、プログラムの中身を変える必要が無いからである。これは define と似た感じである。

次に、プログラムを表示し、その中でも説明していく。

```
/* dat の中身。ルールを記述してある。 */
```

```
012
```

```
032
```

```
103
```

```
123
```

```
143
```

```
212
```

```
252
```

```
303
```

```
343
```

```
363
```

```
414
```

```
434
```

```
454
```

```
474
```

```
523
```

```
543
```

```
583
```

```
632
```

```
672
```

```
743
```

```
763
```

```
783
```

```
852
```

```
872
```

```
/* <プログラム> */
```

```
#include<stdio.h>
```

```
#define MAX 24
```

```
#define TATE 100
```

```
#define LINE "=====¥n"
```

```

int main()
{
    int e,f,g,h,i,j,k,l,n,num,x,y,hyou;
    int m[100],a[9],b[TATE][20][9],c[TATE][9],d[100];
    int blank[MAX],then[MAX],pat[MAX];
    FILE *ei;

    if(NULL == (ei=fopen("tate10.dat","r"))){                /*tate10.dat を開く。そして、ブランク
        の位置とその時の動く位置と*/
        printf("FILE GA ARIMASENN¥n");                    /*そこから動かせる数を配列に格納する。ま
        た、ファイルが*/

        return(0);                                        /*見つからない時はその旨を表示する。*/
    }
    num=0;
    while(EOF!=fscanf(ei,"%1d",&blank[num])){
        fscanf(ei,"%1d",&then[num]);
        fscanf(ei,"%1d¥n",&pat[num]);
        ++num;
    }
    fclose(ei);

    printf(LINE);
    printf("RULE BLANK THEN PAT¥n");                    /*プリント文でルール番号と共に tate10.dat
    の中身を出力する。『R U L E』がルールナンバー、『B L A N K』がブランクの位置、『T H E N』
    がその時に動く位置、『P A T』がそこから何通り動く事が可能かを表す数である。*/
    for(i=0;i<num;i++){
        printf(" %2d   %1d   %1d   %1d¥n",i,blank[i],then[i],pat[i]);
    }

    a[0]=1; a[1]=2; a[2]=3;
    a[3]=8; a[4]=0; a[5]=4;
    a[6]=7; a[7]=6; a[8]=5;                                /*終了状態を a[j]に、初期状態を
    b[0][0][j]に格納し、プリント文で出力する。*/
    b[0][0][0]=2; b[0][0][1]=8; b[0][0][2]=3;
    b[0][0][3]=1; b[0][0][4]=6; b[0][0][5]=4;

```

```
b[0][0][6]=7; b[0][0][7]=0; b[0][0][8]=5;
```

```
printf(LINE);
```

```
printf("SYOKIJYOUTAI SYUURYOUJYOUTAI¥n");
```

```
printf(" |%d%d%d | |%d%d%d | ¥n",b[0][0][0],b[0][0][1],  
        b[0][0][2],a[0],a[1],a[2]);
```

```
printf(" |%d%d%d | |%d%d%d | ¥n",b[0][0][3],b[0][0][4],  
        b[0][0][5],a[3],a[4],a[5]);
```

```
printf(" |%d%d%d | |%d%d%d | ¥n",b[0][0][6],b[0][0][7],  
        b[0][0][8],a[6],a[7],a[8]);
```

```
printf(LINE);
```

```
hyou=0;
```

```
for(h=0;h<9;h++){
```

```
    c[0][h]=b[0][0][h];
```

```
    if(b[0][0][h] == a[h])
```

```
        ++hyou;
```

```
}
```

/\*初期状態と終了状態の見比べ、それ(hyou)が9、つまり、同じ状態だったらかう表示する。『IMINASI(意味無し)』と・・・ \*/

```
if(hyou==9){
```

```
    printf("IMINASI¥n");
```

```
    return(0);
```

```
}
```

```
for(h=0;h<100;h++)
```

/\*d[h]は次の深さに進む事になった時に使った  
ルールナンバーを、m[h]はhの深さの時にいくつ配列があるかを示す。\*/

```
    d[h]=m[h]=0
```

```
h=n=0;
```

```
for(i=0;;i++){
```

```
    x=l=0;
```

```
    for(j=0;j<num;j++){
```

```
        if(x != 1){
```

```
            f=0;
```

```
            if(b[h][m[h]][blank[j]] == 0){
```

/\*ルール番号の0から23までを調べていき、  
空白があったらその空白とその時に動くべき位置との交換を行う。\*/

```
                b[h][m[h]][blank[j]]=b[h][m[h]][then[j]];
```

```
                b[h][m[h]][then[j]]=0;
```

```

        if(h>=1){
            for(y=0;y<=n;y++){
                /*深さが 1 以上の時は今までに出てきた
                ものと全て見比べ、いくつ同じ状態があるかを g に入れる*/
                g=0;
                for(k=0;k<9;k++){
                    if(b[h][m[h]][k] == c[y][k])
                        ++g;
                }

                if(g == 9){
                    f=1;
                    /*もし g が 9、つまり、以前に出てきた
                    のと同じ状態があったら f を 1 とし、以下の流れに規制をかける。*/
                    ++l;
                    /*又、l を 1 足す。この l と、配列 pat に格納されたパターン数
                    が同じ数になったら一つ上の深さに戻ることとなる。それはまた後で説明する。*/
                }
            }
        }

        if(f != 1){
            x=1;
            hyou=0;
            /*もし新たに出てきた状態なら、配列 a と配列 b に格納し、新たに
            出てきたものと最終状態を見比べ、いくつ同じ状態があるかを hyou に入れる*/
            for(k=0;k<9;k++){
                b[h+1][m[h+1]][k]=b[h][m[h]][k];
                c[n+1][k]=b[h][m[h]][k];
                if(a[k] == b[h][m[h]][k])
                    ++hyou;
            }
            b[h][m[h]][then[j]]=b[h][m[h]][blank[j]];
            b[h][m[h]][blank[j]]=0;
            /*新しい配列「h+1」に格納したの
            で、深さ h は元の状態に戻し、*/
            ++h;
            /*配列 b と配列 c の深さを 1 足し、次の状態に進む。*/
            ++n;
            d[h]=j;
            /*d[h]に、今の状態になる時に使っ
            たルールナンバーを入れておく。*/

            if(hyou == 9){

```

```

for(k=0;k<=h;k++){
    printf(" | %d%d%d | ¥n",b[k][m[k]][0],b[k][m[k]][1],

        b[k][m[k]][2]);
    printf(" | %d%d%d | ¥n",b[k][m[k]][3],b[k][m[k]][4],

        b[k][m[k]][5]);
    printf(" | %d%d%d | ¥n",b[k][m[k]][6],b[k][m[k]][7],          /*hyou が 9 になった
ら、終わりなので、深さ 0 から最終状態に到達するまでの過程も表示する。*/
        b[k][m[k]][8]);
    hyou=0;          /*また、その状態が最終状態といくつ同じかも調べる。*/
    for(y=0;y<9;y++){          /*そして、使ったルールナンバーといく
つ同じ状態があるかをを表示する。ちなみに、ルールナンバーは先程使った配列、d[h]に格納され
ている。*/

        if(b[k][m[k]][y] == a[y])
            ++hyou;
    }
    if(k != 0)          /*深さ 0 は初期状態であり、ルールは使わないので、もちろんルー
ルナンバーは出力しない。*/
        printf("RULE=%2d¥n",d[k]);
        printf("hyou=%d¥n¥n",hyou);
    }
    printf("SYUURYOU¥n");          /* 『SYUURYOU ( 終了 )』 と表示
して終わりである。*/
    printf(LINE);  /* ちなみに、見やすくするために所々に線を引いている。*/
    return(0);
}
}
if(f == 1){
    b[h][m[h]][then[j]]=b[h][m[h]][blank[j]];          /*もし、以前に出てきたの
と同じ状態なら、深さ h は元の状態に戻す。*/
    b[h][m[h]][blank[j]]=0;
    }
    e=pat[j];
}
}          /*e にルール j の時のパターン数を代入する。*/
}          /*先程調べた l と e が同じなら、つまり、もうこの深さでは全て

```

のルールを\*/

```
    if(l == e){                                     /*動かしてしまい、もう動かせる場所が無い時は、*/
        ++m[h];                                     /*深さ h の配列の数を 1 つ足し、深さ自体は 1 つ減らし、*/
        --h;                                         /*配列 c を使い、一つ前の深さの状態に戻す作業をする。*/
        for(k=0;k<9;k++)
            b[h][m[h]][k]=c[n-1][k];
    }
}
```

/\* <実行結果> \*/

=====

RULE BLANK THEN PAT

0	0	1	2
1	0	3	2
2	1	0	3
3	1	2	3
4	1	4	3
5	2	1	2
6	2	5	2
7	3	0	3
8	3	4	3
9	3	6	3
10	4	1	4
11	4	3	4
12	4	5	4
13	4	7	4
14	5	2	3
15	5	4	3
16	5	8	3
17	6	3	2
18	6	7	2
19	7	4	3
20	7	6	3
21	7	8	3
22	8	5	2

23      8      7      2

=====

SYOKIJYOUTAI   SYUURYOUJYOUTAI

| 283 |                      | 123 |

| 164 |                      | 804 |

| 705 |                      | 765 |

=====

| 283 |

| 164 |

| 705 |

hyou=4

| 283 |

| 104 |

| 765 |

RULE=19

hyou=6

| 203 |

| 184 |

| 765 |

RULE=10

hyou=5

| 023 |

| 184 |

| 765 |

RULE= 2

hyou =6

| 123 |

| 084 |

| 765 |

RULE= 1

hyou=7

| 123 |



| 804 |

| 765 |

RULE= 8

hyou=9

SYUURYOU

=====

( 2 ) 横型探索法のプログラムを作成して解決しなさい。

横型探索の考え方を軽く説明する。

横型探索の場合、同じ深さを全て探索して行き、次の深さに進んでいく探索法である。

まず、初期状態からブランクを動かせる所に全て動かす。次に、今出てきた全ての状態を一つずつ見ていき、更に同じように動かせるところ全てに動かしていく。この様にして終了状態と同じ配置になるまで操作を行い、なったら終了するというものである。

この探索方法では最短の距離で目的まで行けるが、その過程が長い場合、非常に面倒なことになる。

次に、今回作った横型探索のプログラムにおいて自分で設定した事を説明する。最初に、『 ブランクの位置、 その時に動ける位置』を eight.dat に記しておき、fscan 文で読み込んでいく。もしファイルが見つからない場合は『FILE GA ARIMASENN ( ファイルがありません )』と出力されるようにしておいた。

ブランク等の数字の置かれている位置についても説明しておく。今回も  $3 \times 3$  の空間で数字の移動が行われるが、一番上の行の左から  $0 \cdot 1 \cdot 2$ 、二番目の行の左から  $3 \cdot 4 \cdot 5$ 、一番下の左から  $6 \cdot 7 \cdot 8$  として設定されており、最終状態が格納されている  $a[j]$  と初期状態等が格納されている  $b[h][i][j]$  において、 $j$  部分にこれらの数字が入れられている。

最後にプログラムの流れについての説明に入れる。

まず初めに eight.dat から情報を取り込み、次に配列  $a[j]$ 、 $b[0][0][j]$  にそれぞれ最終状態と初期状態を格納し出力する。

次に、ブランクを探し出し、深さが 1 以上なら以前に出てきた状態全てと比べる ( 深さが 0 の時は他に配列が無いので確かめる必要も無い )。そしてそこで同じ状態が無い事が分かれば、次の状態を配列  $b$  に格納し、その後、終了状態と見比べる。もし同じ状態がある事が分かれば、その配置は考えない事とする。

これを繰り返して最終状態と一致するまで進んでいくのである。

また `dat` を用いてプログラムを作ったのは、それを用いることにより、ルールを変更したい時にはこの `dat` 部分だけを変えればよく、プログラムの中身を変える必要が無いからである。これは `define` と似た感じである。

次に、プログラムを表示し、その中でも説明していく。

```
/*  dat の中身。ルールを記述してある。  */
```

```
01
```

```
03
```

```
10
```

```
12
```

```
14
```

```
21
```

```
25
```

```
30
```

```
34
```

```
36
```

```
41
```

```
43
```

```
45
```

```
47
```

```
52
```

```
54
```

```
58
```

```
63
```

```
67
```

```
74
```

```
76
```

```
78
```

```
85
```

```
87
```

```
/*  <プログラム>  */
```

```
#include<stdio.h>
```

```
#define MAX 24
```

```

#define LINE "=====¥n"
int main()
{
    int f,g,h,i,j,k,l,num,x,y,z,hyou;
    int m[100],a[9],b[100][100][9];
    int blank[MAX],then[MAX];
    FILE *ei;

    if(NULL == (ei=fopen("eight.dat","r"))){ /* e i g h t . d a tを開く。その時NULL、つまり、ファイルが存在しない時はこのように出力し、プログラムを終了させる。*/
        printf("FILE GA ARIMASENN¥n");
        return(0); /*FILE GA ARIMASENN ( ファイルがありません ) と。*/
    }
    num=0;
    while(EOF!=fscanf(ei,"%1d",&blank[num])){ /* 0 ( ブランク ) の位置と、その時ブランクが動く位置をファイルが終わるまで e i g h t . d a tから読み込む。*/
        fscanf(ei,"%1d¥n",&then[num]);
        ++num;
    }
    fclose(ei);

    printf(LINE);
    printf("RULE BLANK THEN¥n"); /* e i g h t . d a tから読み込んだ内容をルールナンバーと共に出力する。『RULE』がルールナンバー、『BLANK』がブランクの位置、『THEN』がその時に動く位置である。*/
    for(i=0;i<num;i++)
        printf(" %2d %1d %1d¥n",i,blank[i],then[i]);

    a[0]=1; a[1]=2; a[2]=3;
    a[3]=8; a[4]=0; a[5]=4; /* 配列 a[z]に終了状態を格納し、配列 b[0][0][z]には初期状態を格納する。そしてこれ等を出力する。*/
    a[6]=7; a[7]=6; a[8]=5;

```

```

b[0][0][0]=2; b[0][0][1]=8; b[0][0][2]=3;
b[0][0][3]=1; b[0][0][4]=6; b[0][0][5]=4;
b[0][0][6]=7; b[0][0][7]=0; b[0][0][8]=5;

```

```

printf(LINE);
printf("SYOKIJYOUTAI  SYUURYOUJYOUTAI¥n");
printf("  | %d%d%d |      | %d%d%d | ¥n",b[0][0][0],b[0][0][1],
        b[0][0][2],a[0],a[1],a[2]);
printf("  | %d%d%d |      | %d%d%d | ¥n",b[0][0][3],b[0][0][4],
        b[0][0][5],a[3],a[4],a[5]);
printf("  | %d%d%d |      | %d%d%d | ¥n",b[0][0][6],b[0][0][7],
        b[0][0][8],a[6],a[7],a[8]);
printf(LINE);

```

```

hyou=0;
for(h=0;h<9;h++){
    if(b[0][0][h] == a[h])
        ++hyou;
}
/*初期状態と終了状態の見比べ、それ(hyou)が9、つ
まり、同じ状態だったらこう表示する。『IMINASI(意味無し)』と・・・*/
if(hyou==9){
    printf("IMINASI¥n");
    return(0);
}

```

```

x=1;
for(h=0;h<100;h++){
    m[h]=x;
    /*m[0]、つまり、深さ0の時は、初期状態の1つしか状態が*/
    x=0;
    /*ないので、1としておく。そして、それ以降はその前の深さの*/
    for(i=0;i<m[h];i++){
        /*状況 x により変わっていく。x については以降説明する。*/
        for(j=0;j<num;j++){

```

```

f=0;
if(b[h][i][blank[j]] == 0){          /*ルール番号 0 から 2 3 まで調べていき、ブ
ンクがあったらそのblankとその時に動くべき位置との交換を行う。*/
    b[h][i][blank[j]]=b[h][i][then[j]];
    b[h][i][then[j]]=0;
    if(h>=1){
        for(y=0;y<h;y++){
            for(z=0;z<=m[y];z++){
                g=0;          /*深さ 1 以上の時は今までに出てきたものと全て見比べ、いくつ同じ
状態があるかを g に入れる*/
                for(k=0;k<9;k++){
                    if(b[h][i][k] == b[y][z][k])
                        ++g;
                }
                if(g == 9)
                    f=1;          /*もし g が 9、つまり、以前に出てきたのと
同じ状態が 1 つでもあったら f を 1 とし、以下の流れに規制をかける。*/
            }
        }
    }
    if(f!=1){
        hyou=0;
        for(k=0;k<9;k++){      /*もし新たに出てきた状態なら、配列 b に格納し、新たに出て
きたものと最終状態を比べ、いくつ同じ状態があるかを調べ、hyou に入れる。*/
            b[h+1][x][k]=b[h][i][k];
            if(b[h][i][k] == a[k])
                ++hyou;
        }
        ++x;          /*ここで x を足していく。これをするにより、新しい配置
が出来る度に x が足されていくので、いくつ出てこようと配列 b の違う場所に格納され、後での操
作が可能となるのである。又、深さ h+1 での取り扱うべき数も分かる。*/

        if(hyou == 9){
            b[h][i][then[j]]=b[h][i][blank[j]];
            b[h][i][blank[j]]=0;          /*hyou が 9 になったら、ひとまず深さ h の状態を元に
戻し、初期状態から終了状態になるまでの全ての深さ（階層）を出力させる。*/

```

```
m[h+1]=x;
for(k=0;k<=h+1;k++){
    for(y=0;y<m[k];y++)
        printf("
%d%d%d | ",b[k][y][0],b[k][y][1],b[k][y][2]);
        printf("¥n");
        for(y=0;y<m[k];y++)
            printf("
%d%d%d | ",b[k][y][3],b[k][y][4],b[k][y][5]);
            printf("¥n");
            for(y=0;y<m[k];y++)
                printf("
%d%d%d | ",b[k][y][6],b[k][y][7],b[k][y][8]);
                printf("¥n¥n");
    }
    printf("SYUURYOU¥n");
}
printf(LINE);
return(0);
}
}
b[h][i][then[j]]=b[h][i][blank[j]];
/*もし、以前に出てきたのと同じ状態なら、深さ h は元の状態に戻す。つまり、同じ状態だったらそれは考えずに無視して次の操作に移るのである。*/
b[h][i][blank[j]]=0;
}
}
}
}
```

1	0	3
2	1	0
3	1	2
4	1	4
5	2	1
6	2	5
7	3	0
8	3	4
9	3	6
10	4	1
11	4	3
12	4	5
13	4	7
14	5	2
15	5	4
16	5	8
17	6	3
18	6	7
19	7	4
20	7	6
21	7	8
22	8	5
23	8	7

=====

SYOKIJYOUTAI   SYUURYOUJYOUTAI

283	123
164	804
705	765

=====

| 283 |  
| 164 |  
| 705 |

283		283		283
104		164		164
765		075		750

203		283		283		283		283
184		014		140		064		160
765		765		765		175		754

023		230		083		283		280		283		083		283		280		283
184		184		214		714		143		145		264		604		163		106
765		765		765		065		765		760		175		175		754		754

123		234		803		283		208		283		803		203		283		283		208		203		283		283
084		180		214		714		143		145		264		684		640		674		163		186		016		156
765		765		765		605		765		706		175		175		175		105		754		754		754		704

| 123 |  
| 804 |  
| 765 |

SYUURYOU

=====

( 3 ) 知識ベース形式のプログラムを作成して解決しなさい。

知識ベース形式のプログラムについて軽く説明する。

この場合、まずは同じ深さを全て探索して行き、同時にそれ等の評価値を調べておく。

出てきた中から表価値の最も高いものの一つを次の深さ(階層)にし、最終段階までいくというものである。

次に、今回作った知識ベース形式のプログラムにおいて自分で設定した事を説明する。

このプログラムでも縦型・横型と同様、知識ベースと推論機構を分離した。知識ベースとして、

『 ・ ブランクの位置、 ・ その時に動ける位置、 次の深さ(階層)に進むにあたり、選んではいけないルール』を eight.dat に記しておき、fscan 文で読み込んでいく。最後の は一つ前の状態から現在の状態になったのに、その逆の移動をしてまた戻ってしまっは意味がないので、それを防ぐための決まりである。又、もしファイルが見つからない場合は『FILE GA ARIMASENN (ファイルがありません)』と出力されるようにしておいた。

ブランク等の数字の置かれている位置についても説明しておく。今回も  $3 \times 3$  の空間で数字の移動が行われるが、一番上の行の左から  $(0,0) \cdot (0,1) \cdot (0,2)$ 、二番目の行の左から  $(1,0) \cdot (1,1) \cdot (1,2)$ 、一番下の左から  $(2,0) \cdot (2,1) \cdot (2,2)$  として設定されており、終了状態が格納



されている  $a[i][j]$  と初期状態等が格納されている  $b[h][i][j]$  において、 $i$  および  $j$  部分にこれらの数字が入れられている。

最後にプログラムの流れについての説明に入れる。

まず初めに知識ベースから情報を取り込み、次に配列  $a[j]$ 、 $b[0][0][j]$  にそれぞれ最終状態と初期状態を格納し出力する。

次に、ブランクを探し出し（ここでは先程言ったように、次の深さ（階層）に進むにあたり、選んではいけないルールと見比べ、これに当てはまらないという条件もついている）、前に出てきた状態全てと比べる。そしてそこで同じ状態が無い事が分かれば評価値を調べる。表価値が高いか同じならばそれを次の深さ（階層）の候補とし、低いならそれは無視する。もし、一番高い表価値が2つ出てきてしまったら乱数でどちらかを決定し、先に進む。ちなみに、評価値とは次の段階に進むにあたり、どの状態を次の段階にさせるかを決めるための値であり、そこには、最終状態とおなじ状態がいくつあるかが入ることになる。

このようにして操作を行い、最終状態まで進んで行き、終了となるのである。

知識ベース形式では、知識ベースと推論機構を分離して考えてある。これは、そう考えることにより、ルール（知識ベース）を変更したい時にはこの `data` 部分だけを変えればよく、プログラムの中身を変える必要が無いからである。

最後に、知識ベース、推論機構、作業記憶とプログラムの対応であるが、プログラム上の15行目に出てくる `eight10.dat` が知識ベースであり、その中身は下に書いておく。

推論機構は実作業部分であり、プログラム上では73行目の『`for(h=0;h<50;h++){`』から最後までである。

また、縦型・横型探索と同じく、次の深さ（階層）にいく時に、それを新しい配列に記憶させなくてはならぬ場面が現れる。プログラム上では147行目の『`b[h+1][i][j]=b[h][i][j];`』でその作業が行われており、『`b[h+1][i][j]`』が作業記憶である。

次に、プログラムを表示し、その中でも説明していく。

```
/* 知識ベース */
```

```
000102
```

```
001007
```

```
010000
```

```
010205
```

```
011110
```

```
020103
```

```
021214
```

```
100001
```

101111  
102017  
110104  
111008  
111215  
112119  
120206  
121112  
122222  
201009  
202120  
211113  
212018  
212223  
221216  
222121

/\* < プログラム > \*/

#include<stdio.h>

#define MAX 24

#define KEI "=====¥n" /\*今回はより\*/

#define SEN "-----¥n" /\*見やすくするために2種類の線を用意した。\*/

void uniran();

int main()

{

int h,i,j,k,l,o,num,t,u,v,w,x,y,z,hyou;

int a[3][3],b[100][3][3],r[50];

int blank\_g[MAX],blank\_r[MAX],then\_g[MAX],then\_r[MAX],numb[MAX];

float ransu,ran[100];

long s,shoki;

FILE \*ei;

if(NULL == (ei=fopen("eight10.dat","r"))){

/\* 『eight10.dat』を開く。もしそのフ

ァイルが存在しなかったら\*/

printf("FILE GA ARIMASENN¥n");

/\*その旨を表示して終了する。\*/

```

    return(0);
}
num=0;
while(EOF!=fscanf(ei,"%1d",&blank_g[num])){
    fscanf(ei,"%1d",&blank_r[num]);          /* 0 ( ブランク ) の位置と、その時ブラ
ンクが動く位置、それに*/
    fscanf(ei,"%1d",&then_g[num]);          /* 次の深さ ( 階層 ) に進むにあたり、選
んではいけないルールを*/
    fscanf(ei,"%1d",&then_r[num]);          /* ファイルが終わるまで e i g h t . d a t から
読み込む。*/
    fscanf(ei,"%2d",&numb[num]);
    ++num;
}
fclose(ei);
printf(KEI);
printf("RULE BLANK THEN N.RULE¥n");          /* e i g h t . d a t から読み
込んだ内容をルールナンバーと共に*/
for(i=0;i<num;i++){          /* 出力する 『RULE』 がルールナンバー、『BLANK』 がブランク*/
    printf(" %2d  %1d%1d  %1d%1d  %2d¥n",i,blank_g[i],          /* の位置、『THEN』 がその
とき動く位置、『N.RULE』 が次の*/
        blank_r[i],then_g[i],then_r[i],numb[i]);          /* 深さ ( 階層 ) に進むにあた
り選んではいけないルールである。*/
    }
printf(KEI);
a[0][0]=1; a[0][1]=2; a[0][2]=3;
a[1][0]=8; a[1][1]=0; a[1][2]=4;
a[2][0]=7; a[2][1]=6; a[2][2]=5;          /* 配列 a[i][j]に終了状態を格納し、配列
b[0][i][j]には初期状態*/
                                          /* を格納する。*/

b[0][0][0]=2; b[0][0][1]=8; b[0][0][2]=3;
b[0][1][0]=1; b[0][1][1]=6; b[0][1][2]=4;
b[0][2][0]=7; b[0][2][1]=0; b[0][2][2]=5;

hyou=0;
for(i=0;i<=2;i++){
    for(j=0;j<=2;j++){

```

```

        if(a[i][j] == b[0][i][j])
            hyou++;
    }
}
/*初期状態と終了状態の見比べ、それ(hyou)が9、つまり、同じ状態
だったらかう表示する。『IMINASI(意味無し)』と・・・*/
if(hyou == 9){
    printf("IMINASI¥n");
    return(0);
}

printf("SYOKIJYOUTAI SYUURYOUJYOUTAI¥n");
printf("  |%1d%1d%1d|      |%1d%1d%1d| ¥n",b[0][0][0],b[0][0][1],    /*初期状態と終了
状態を出力する。その際、*/
        b[0][0][2],a[0][0],a[0][1],a[0][2]);    /*初期状態には、最終状態と
いくつ同じのがあるかも合わせて表示する。*/
printf("  |%1d%1d%1d|      |%1d%1d%1d| ¥n",b[0][1][0],b[0][1][1],
        b[0][1][2],a[1][0],a[1][1],a[1][2]);
printf("  |%1d%1d%1d|      |%1d%1d%1d| ¥n",b[0][2][0],b[0][2][1],
        b[0][2][2],a[2][0],a[2][1],a[2][2]);
printf("  hyou=%1d¥n",hyou);
printf(KEI);

shoki=3548276;
for(h=0;h<50;h++){    /*乱数のプログラム。ran[0]から ran[49]に乱数を格納していき、*/
    uniran(&shoki,&ransu); /*もし、最高表価値が同じものが出てきたらこれを使い判断する。*/
    ran[h]=ransu;
}

l=24;    /* 1 には最後に使われたルールナンバーが入れられる。最初は何も使われてない*/
for(h=0;h<50;h++){    /*ので、初期状態と関係のないルールナンバーを入れておいた。*/
    x=0;
    v=0;
    for(k=0;k<num;k++){
        if((b[h][blank_g[k]][blank_r[k]] == 0) && (l != numb[k])){    /*ルール番号の0 から
2 3 まで調べていき、空白があったら*/
            b[h][blank_g[k]][blank_r[k]]=b[h][then_g[k]][then_r[k]];    /*その空白とその時

```

に動くべき位置との交換を行う。\*/

```
b[h][then_g[k]][then_r[k]]=0;
```

/\* その時、ブランクが

あると同時に、最後に使われたルールナンバーが num[k] と同じでない事が条件になる。\*/

```
w=0;
```

```
for(o=0;o<h;o++){
```

```
  r[o]=0;
```

```
  for(i=0;i<=2;i++){
```

```
    for(j=0;j<=2;j++){
```

/\* 深さ 1 以上の時は今までに出てきたものと全て見比べ、  
いくつ同じ状態があるかを r[o] に入れる。\*/

```
      if(b[o][i][j] == b[h][i][j])
```

```
        ++r[o];
```

```
    }
```

```
  } /*そして、同じ状態が以前にもあったと分かったら w に 1 を足す。*/
```

```
if(r[o] == 9)
```

```
  ++w;
```

```
}
```

```
if(w == 0){
```

```
  hyou=0;
```

/\* w が 0、つまり、以前に同じ状態が無かったら表価値を調べる。\*/

```
  for(i=0;i<=2;i++){
```

```
    for(j=0;j<=2;j++){
```

```
      if(a[i][j] == b[h][i][j])
```

```
        ++hyou;
```

```
    }
```

```
  }
```

```
if(hyou>x){
```

```
  x=hyou;
```

/\* 表価値が x より大きければその値を x に入れ、y にその時の\*/

```
  y=k;
```

/\* ルールナンバーを入れる。\*/

```
  v=0;
```

/\* v は後で乱数を使用するかどうかを判断する値で、これが 0 の時は\*/

```
}
```

/\* 使用しない、0 以外の時は使用するという風にしておいた。\*/

/\* ここではもちろん新しい最高表価値がでたので v は 0 にしておくのである。\*/

```
else if(hyou == x){
```

```
  ++v;
```

/\* 表価値が同じ場合は後で乱数を使用するので v に 1 足しておく。\*/

```

        z=k;                                /* z には y と同様、ルールナンバーを入れておく。*/
    }

    printf(" | %d%d%d | ¥n",b[h][0][0],b[h][0][1],b[h][0][2]);
    printf(" | %d%d%d | ¥n",b[h][1][0],b[h][1][1],b[h][1][2]);    /*今の状態を一応出力し
    ておく。この深さでの全ての*/
    printf(" | %d%d%d | ¥n",b[h][2][0],b[h][2][1],b[h][2][2]);    /*状態を調べた終わった後、
    最終的にどれを次の深さと*/
    printf("RULE=%d¥nHYOUKATI=%1d¥n¥n",k,hyou);    /*してもっていくかを決め
    て、改めて出力する。その作業はまた後で・・・*/
    if(hyou == 9){
        printf("SYUURYOU¥n");    /*表価値が9だったら SYUURYOU (終了)*/
        printf(KEI);    /*と表示して作業を終了する。*/
        return(0);
    }
}

b[h][then_g[k]][then_r[k]]=b[h][blank_g[k]][blank_r[k]];    /*一度深さ h の状態を元に
戻しておく。*/
b[h][blank_g[k]][blank_r[k]]=0;
}
}

if(v == 1){
    printf("RANSUU¥n");    /*v が 1、つまり、最高表価値が 2 つ出た時は乱数を
    使用して決定する。*/

    s=0;    /*その旨を表示する。*/
    t=0;    /*先程求めた ran[h]を整数にし、2 で割る。*/
    u=0;
    s=(long)ran[h];
    t=s*1;
    u=t%2;
}

if(u == 1)    /*もし割り算で余りが出てきたら y に z を入れておく。*/
    y=z;

    b[h][blank_g[y]][blank_r[y]]=b[h][then_g[y]][then_r[y]];    /*ここで、又、ブランクとその時
    動くべき位置との入れ替えを行う。*/
    b[h][then_g[y]][then_r[y]]=0;    /*もし、割り算が割り切れれば最初

```

に見つけた最高表価値の状態にし、1 余れば、次に見つけた同じ最高表価値を持つ状態にする\*/

```
l=y; /* いう事である。 */
for(i=0;i<=2;i++){
    for(j=0;j<=2;j++){ /* ここで、はれて次の状態が決定した訳である。 */
        b[h+1][i][j]=b[h][i][j];
    }
}
printf(SEN);
printf("TUGINODANNKAIHA¥n");
printf(" | %1d%1d%1d | ¥n",b[h+1][0][0],b[h+1][0][1],b[h+1][0][2]); /* 『次の状態は』 とい
うことで、今決まった状態と、その*/
printf(" | %1d%1d%1d | ¥n",b[h+1][1][0],b[h+1][1][1],b[h+1][1][2]); /* 表価値が出力され
る。 */
printf(" | %1d%1d%1d | ¥n",b[h+1][2][0],b[h+1][2][1],b[h+1][2][2]);
printf("HYOUKATI=%1d¥n",x);
printf(SEN);
}
}
```

void uniran(id,x) /\*これが乱数のプログラムである。\*/

```
long *id;
float *x;
{
    *id=*id+48828125;
    if(*id<0)
        *id=(*id+2147483647)+1;
    *x=(float)*id*0.4656613e-9;
}
```

/\* <実行結果> \*/

=====

RULE BLANK THEN N.RULE

0	00	01	2
1	00	10	7
2	01	00	0

3	01	02	5
4	01	11	10
5	02	01	3
6	02	12	14
7	10	00	1
8	10	11	11
9	10	20	17
10	11	01	4
11	11	10	8
12	11	12	15
13	11	21	19
14	12	02	6
15	12	11	12
16	12	22	22
17	20	10	9
18	20	21	20
19	21	11	13
20	21	20	18
21	21	22	23
22	22	12	16
23	22	21	21

=====

SYOKIJYOUTAI SYUURYOUJYOUTAI

| 283 | | 123 |

| 164 | | 804 |

| 705 | | 765 |

hyou=4

=====

| 283 |

| 104 |

| 765 |

RULE=19

HYOUKATI=6

| 283 |

| 164 |



| 075 |  
RULE=20  
HYOUKATI=3

| 283 |  
| 164 |  
| 750 |  
RULE=21  
HYOUKATI=3

-----  
TUGINODANNKAIHA  
| 283 |  
| 104 |  
| 765 |  
HYOUKATI=6

-----  
| 203 |  
| 184 |  
| 765 |  
RULE=10  
HYOUKATI=5

| 283 |  
| 014 |  
| 765 |  
RULE=11  
HYOUKATI=5

| 283 |  
| 140 |  
| 765 |  
RULE=12  
HYOUKATI=4

RANSUU  
-----

TUGINODANNKAIHA

| 203 |

| 184 |

| 765 |

HYOUKATI=5

-----

| 023 |

| 184 |

| 765 |

RULE=2

HYOUKATI=6

| 230 |

| 184 |

| 765 |

RULE=3

HYOUKATI=4

-----

TUGINODANNKAIHA

| 023 |

| 184 |

| 765 |

HYOUKATI=6

-----

| 123 |

| 084 |

| 765 |

RULE=1

HYOUKATI=7

-----

TUGINODANNKAIHA

| 123 |

| 084 |

| 765 |

HYOUKATI=7

-----

| 123 |

| 804 |

| 765 |

RULE=8

HYOUKATI=9

SYUURYOU

=====

考察：横型探索はゴール（目的状態）に行くまで最短経路が分かるだけに、有効な手段だと考えられる。縦型探索は下手をしたら永遠に終わらない可能性もある。今回は優先方向の設定と初期状態がうまくされていたおかげですんなり行った。知識ベース形式のプログラムでは乱数が問題である。乱数の選択のおかげで、遠回りする羽目になるとも考えられる。また、今の状態の中では一番高い表価値を選び次の状態に行ったが、結局その次の深さでは表価値が下がる恐れもある。そういった意味で、いかに先の状態まで読めるかがこの探索方法の発展のカギを握っているのではと考えられる。また、上で言った事を改善できれば、この3つの探索方法の中では知識ベース形式のプログラムが最も有効な手段になり、行き当たりばったりの探索方法だったものから抜け出す事ができると考えられる。