Program Promenade

連載

トランプの1人遊び

石畑　清（明治大学理工学部）
ishihata@cs.meiji.ac.jp

今回は，2003年会津大会の問題F「Gap」を取り上げる（http://www.u-aizu.ac.jp/conference/ACM/problems/参照）．カード（トランプ）の1人遊びを成功させるための手順の発見を要求する問題である．Gapという意味不明のタイトルは，このゲームの名前である．

■ゲームのルール

まずルールを説明しよう．4種類のスーツ（スペード，ハート，ダイヤモンド，クラブ）で，ランク（カードに表示されている数）が1～7のカードが与えられている．合計28枚ということになる．

スーツを文字で表すのは面倒なので，問題ではスーツも1～4の数字で示すことにしている．1枚のカードの表現は，スーツとランクの数字を並べた2桁の10進数のような格好になる．11～17，21～27，31～37，41～47の28枚である．

場として縦4×横8，合計32枚のカードを置けるスペースを用意する．最初に，ここに28枚のカードをランダムに並べる．このとき，各列の一番左のスペースは空けておく．たとえば，**図-1**のようになる．

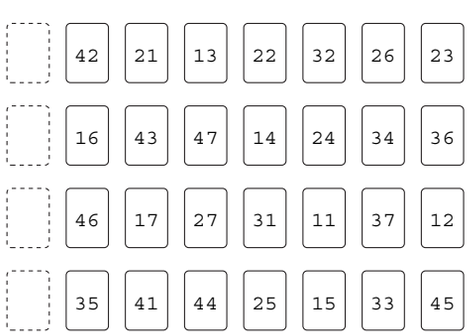

図-1　カードを配った直後の状態

次に，ランクが1のカードを所定の位置に移す．スーツx，ランク1のカードを上からx段目の列の左端に移すのが決まりである．たとえば，31は上から3段目の左端に移すことになる．図-1の配置は，**図-2**のように変わる．ここまでは，決まりに従うだけで，思考力を発揮する余地はない．ここからがゲーム開始である．

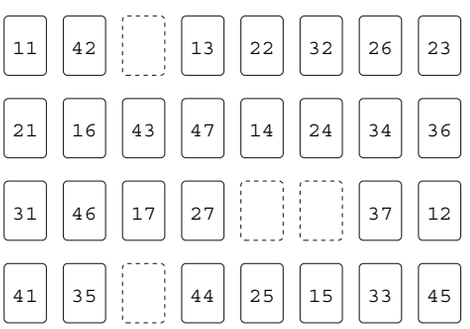

図-2：ゲーム開始の状態

ゲームは，カードの移動を繰り返すことによって進行する．カードの移動の仕方に関するルールは次のとおりである．場の中のギャップ（盤面中の空き，カードが置かれていない場所）に，そのすぐ左にあるカードと同じスーツで，ランクが1だけ大きいカードを移動させることができる．たとえば，図-2では，最下段にあるギャップのすぐ左のカードが35なので，このギャップに36を移動させることができる．今まで36があった位置が今度はギャップになる．この操作の結果を**図-3**に示す．

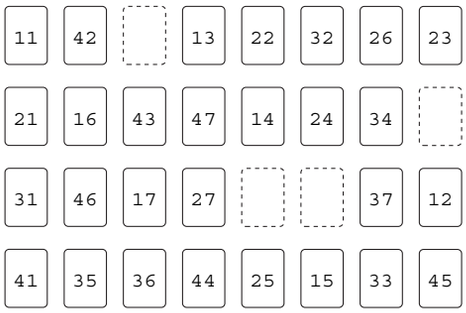

図-3　カードを1枚移動した後の状態

ギャップのすぐ左がまたギャップだった場合，右側のギャップにカードを移動させることはできない．また，ギャップのすぐ左のカードのランクが7だった場合も，そのギャップにカードを移動させることはできない．しかし，これらはいずれも，それで行き詰まってしまったというわけではない．該当のギャップの左側の状況が他のギャップに対する移動で変化するかもしれない．そうなれば，そのギャップへの移動が可能になる．

たとえば，図-2で，3段目に2つ並んだギャップには，いずれも（現時点では）移動できない．左側のギャップは，すぐ左のカードのランクが7だから移動不可である．また，右側のギャップは，すぐ左がギャップだから移動できない．

ゲームの各ステップでは，最大4とおりの操作が可能である．4つあるギャップのどれを選んで移動させてもよい．もちろん，どのギャップを選ぶかによって，結果は異なる．各ステップにおけるギャップの選び方に戦略の余地がある．

ゲームの目的は，**図-4**のように，各スーツのカードがランク順にきちんと並んだ状態を作ることである．正確に言えば，スーツx，ランクnのカードがx段目の左からn番目になるようにする．この状態に到達させることができれば成功，到達せずに，4つのギャップのどれにも移動ができない状態になれば失敗である．

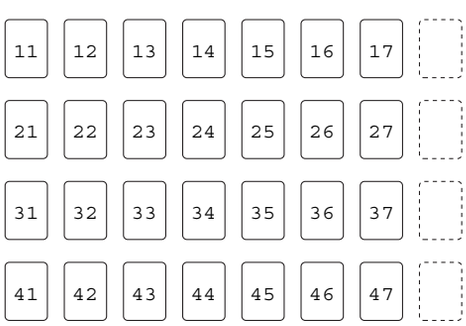

図-4　目標状態

問題では，与えられた初期状態から目標状態に到達するための最短手数を求めることを要求している．目標状態への到達が不可能な場合は，−1と答える．入力として与えられるのは，図-1のような状態である．図-1を図-2に変えるために4手必要だが，これは手数には含めない．

■ゲームの分析

解法を考えるには，ゲームがどんな性質を持っているかの分析が欠かせない．このゲームの場合，場の状態1つ1つを頂点（ノード）とする有向グラフで初期状態からどんな状態に到達可能であるかの全体像を表現することができる．これはすぐに分かるだろう．ある状態で移動できるカードは最大4枚なので，1つの頂点から出る辺（矢印）は最大4本である．

グラフの全体を図示することは，とても不可能である．一般の初期状態（図-2の状態ではない）から2手以内で移動できる状態の関係に限ったグラフを**図-5**に示す．これで，どんなグラフになるか，およそのことは把握できると思う．

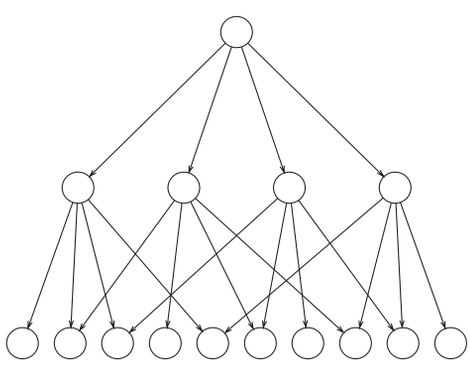

図-5　初期状態から2手以内の状態を表すグラフ

ゲームのグラフは木にはならない．ある頂点からの辺と別の頂点からの辺が同じ頂点で合流することがある．これは当然である．ギャップAとBがあったとして，先にAへ移動してから次にBへ移動しても，Bへ移動してからAへ移動しても，結局同じ状態になるケースがほとんどだからである．

グラフ中にループがないことも，できれば見つけ出してほしい性質である．辺の矢印に従った移動を繰り返していって元の頂点に戻ることは決してない．カードの移動が非可逆なので，一見明らかだが，証明にはテクニックを要する．

ループの不可能性を厳密に証明するには，次のよう

にすればよい．盤面の中で，同じスーツ，連続するランクのカードが並んでいる個所に対して，ポイントと呼ぶ値を定義する．

x6 と x7 が並んでいれば 1 点
x5 と x6 が並んでいれば 2 点
...
x1 と x2 が並んでいれば 6 点

この値を盤面中の連続の個所すべてについて加え合わせる．これを盤面のポイントと呼ぶことにしよう．

こう定義すると，盤面のポイントは単調増加になる．たとえば，14 と 15 の連続がくずれるのは，14 が移動した場合だけだが，その場合は 13 と 14 の連続ができているはずだから，盤面全体のポイントは増えている．ポイントが単調増加だから，ループはあり得ない．

■グラフの探索

この問題は，数学的な考察によってスパッと解けるとは到底思えない．試行錯誤的にいろいろな移動を試してみて，解を探し回るしかないだろう．アルゴリズムの用語で表現すれば，グラフの探索ということになる．探索の途中で目標の状態に到達できれば，解が見つかったことになるし，可能な移動をすべて試した末に目標状態に到達できないのなら，そもそもその初期状態からは目標状態に到達不可能だったことが分かる．

グラフの探索の基本的な戦略には，幅優先探索と深さ優先探索がある．この問題は，このどちらの戦略を採用しても解ける．2 つの戦略のこの問題に対する適用性を簡単に考察しよう．

一般的に言って，深さ優先探索の長所はメモリ使用量が少なくて済むことである．特に，ゲームを表すグラフが木になる場合は，木の根から現在調べている頂点までの経路上の頂点だけ記憶していれば十分である．幅優先探索だと，それ以前に調べた頂点すべてを記憶することが必要なので，かなりの量のメモリが必要になる．

しかし，このゲームの場合は，事情が異なる．単純な深さ優先探索だと，2 つの辺が合流した状態からの探索を 2 回行ってしまうことになる．寸分違わない探索を 2 回行うだけで，何も新しい情報をもたらさない．これは時間の無駄である．合流はグラフのあちこちで起こり得るので，これによる計算時間の損は莫大なものになる．つまり，単純な深さ優先探索でも，原理的には Gap の問題を解くことができるのだが，現実的な時間で計算を終わらせることはできない．

深さ優先探索を使って，現実的な時間で Gap の問題を解くには，探索途中に現れた状態をすべて記憶することが必要である．探索の途中で，ある状態に到達したときは，まずその状態が今までに探索済みかどうかを調べる．探索済みであれば，その状態についての結果はすでに得られているはずなので，何もせずに引き返す．まだ探索していない新しい状態だったときに限って，探索を行い，結果を表（データベースと言ってもよい）に記録する．

この方法を採った場合，必要となるメモリ量は幅優先探索とほぼ同じになる．探索途中に現れた状態をすべて記録するという点で何ら変わりないからである．つまり，メモリ量が少なくて済むという深さ優先探索の長所は，発揮されない．

深さ優先探索の場合，次のようなケースがあり得ることにも注意が必要である．**図 -6** のような状態があったとする．最初に 12 を 11 の右に移動し，13，14，... と，この順で移動すれば，6 手で目標状態に到達できる．ところが，最初に 15 を移動して 14，15 と並んだ状態を作り，それからこの 2 枚をまとめて左にずらすような手順も可能である．この手順を採った場合，[15]，[14,15,16]，[13,14,15,16,17]，[12,13,14,15,16,17] の順で，15 手もかけて，やっと目標状態に到達できる．

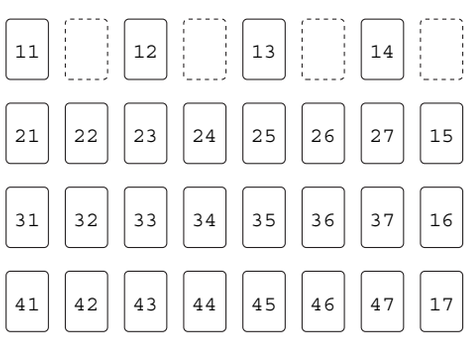

図 -6　手数の違う解が存在する例

深さ優先探索の場合，どちらの手順を先に試すかは分からない．最初に見つかった解が後の手順のような最短手数でないものである可能性がある．したがって，解を 1 つ見つけたとしても，それで探索を打ち切ってはいけない．どうしても，グラフ全体を調べ終わるまで，探索を続ける必要があるだろう．

結局，メモリ量にしろ，計算時間にしろ，深さ優先探索と幅優先探索のどちらを採用しても，大差ないと予想される．

なお，深さ優先探索で状態の記憶が必要なケースは，

ほかにもある．Gap とは逆に，状態のループが起こり得るが，状態の合流は起こらないという性質を持った問題が存在する．このような問題なら，探索開始位置から現在位置までの途中の状態を記憶するだけで足りる．調べ終わった状態に関する記録は，そのつど消して差し支えない．これなら，大したメモリ量にはならないので，深さ優先探索のメモリ量に関する優位が保てるのだが，残念ながら Gap の問題の場合，そうはいかないのだ．

以下では，幅優先探索，深さ優先探索の両方でのプログラミングを試みる．

■共通の道具立て

初めに，幅優先探索でも深さ優先探索でも必要になる共通の道具立てから用意しよう．

まず，次のような #define 定数を用意する．

```
#define M            7
#define N           (M+1)
#define Hash_Size    293129
```

M はカードのランクの最大値，N は横 1 列に並ぶカードの枚数である．N は M より 1 だけ大きい．

幅優先探索にしろ，深さ優先探索にしろ，状態を記録する表を作って，探索を行うことが必要になるのは間違いない．ここでは，そのアルゴリズムとしてハッシュ法（チェイン方式）を採用する．C++ の STL や Java のクラスライブラリを上手に使えば，探索アルゴリズムまで書かずに済むのだが，C だとこんなところから苦労しなければならない．Hash_Size は，ハッシュ表として使う配列の長さ（要素数）である．ハッシュ法の効率面での安全性を考えて，素数を選んである．

個々の盤面（場の状態）は，次のような構造体型で表すことにする．

```
typedef struct state {
  char a[4*N];
  struct state *link;
  .....
} state;
```

メンバ a は，長さ 32 の配列である．これに 32 個の位置それぞれにあるカードの値（11 ～ 47）を入れる．カードのない位置（ギャップ）の場合は，0 を入れると約束する．

a の要素の型は char にしてある．整数だから int にするのが自然だが，この問題ではメモリの量を節約することが大切である．大量のメモリを使うプログラムになるので，いろいろな意味でメモリ量がネックになる．メモリの量を節約するだけで，キャッシュなどの効果が大きくなって，スピードアップにつながるという可能性だってある．0 ～ 47 の値しか入れないのだから，8 ビットの整数，すなわち char 型で十分で，この型を使うべきであろう．

link は，ハッシュ法で使うリンク（次の要素を指すポインタ）である．この 2 つのメンバのほかに，幅優先探索と深さ優先探索のそれぞれで，型 state に独自のメンバが追加される．

初期状態，目標状態とハッシュ表を表す変数の宣言は次のとおりである．

```
state init;
state final;
state *hash_table[Hash_Size];
```

これらの変数の初期化（初期状態の場合は値の読み込み）は次のようになる．

```
for (j = 0; j < 4; j++) {
  init.a[N*j] = 10*j+11;
  for (i = 1; i <= M; i++) {
    scanf("%d", &x);
    if (x%10 == 1)
      x = 0;
    init.a[N*j+i] = x;
  }
}
for (j = 0; j < 4; j++) {
  final.a[N*(j+1)-1] = 0;
  for (i = 0; i < M; i++)
    final.a[N*j+i] = 10*j+i+11;
}
for (i = 0; i < Hash_Size; i++)
  hash_table[i] = NULL;
```

初期状態に関しては，データを読み込むと同時に，ランク 1 のカードを左端に移す操作まで済ませている．なお，ここでは標準入力から入力しているが，コンテストの場合は指定されたファイルから入力するようにプログラムを書かなければならない．

ハッシュ法による探索は，次のような関数で実現できる．この関数は，引数として与えられた状態 x が表の中にあるかどうか調べ，なければ新しく作って表に入れる．つまり，search_table という名前であるが，search だけでなく，insert の処理も含んでいる．

```
state *search_table(state x, int *new)
{
  int i, f;
  unsigned int h;
  state *p, *q;
```

```
  h = 0;
  f = 1;
  for (i = 0; i < 4*N; i++) {
    h += f*x.a[i];
    f += i+1;
  }
  h %= Hash_Size;
  p = hash_table[h];
  while (p != NULL) {
    for (i = 0; i < 4*N; i++)
      if (x.a[i] != p->a[i])
        break;
    if (i >= 4*N) {
      *new = 0;
      return (p);
    }
    p = p->link;
  }
  q = (state *)malloc(sizeof(state));
  assert(q != NULL);
  *q = x;
  q->link = hash_table[h];
  hash_table[h] = q;
  *new = 1;
  return (q);
}
```

結果の値として返すのは，表の中で見つかった（または，新たに挿入された）状態へのポインタである．引数 new は，状態が見つかったのか，それとも新しく作られたのかの区別を返す．状態が新しく作られた場合は，この引数の指す整数値が 1 になる．既存の状態だった場合は，0 になる．

アルゴリズムは，ごく素直なハッシュ法（チェイン方式）だから，説明の要はなかろう．ハッシュ関数は，32 個の値に 1，2，4，7，11，... という値をそれぞれ掛けて，足し合わせた上で，Hash_Size で割ったときの余りを採用している．掛け算の定数の列は，差が 1, 2, 3, 4, ... となるように選んだものである．本当は，こんないい加減な定数列を使うよりも，素数の列を使うなどの工夫をした方がよいのかもしれないが，さほどの差はなさそうなので，簡単な方法を選んだ．なお，最初は定数列を 1，2，3，4，5，... としたプログラムを書いたのだが，これは明らかに性能が悪かった．これではサボりすぎのようだ．

状態を記録する記憶域は，ライブラリ関数 malloc を使って 1 回ごとに動的に確保している．普通の探索なら，大きな配列を宣言して，そこに順に入れていくようなプログラムの書き方でもよいのだが，Gap の場合は記録するデータの個数の見積もりができない．配列の長さをいくつに決めたらよいか分からないので，動的記憶域割当てを使わざるを得ないであろう．

ハッシュ関数の計算や状態の比較で，char 型の配列要素を 1 つずつ順に調べている．C プログラミングのベテランになると，技を使って速くする誘惑にかられるところだろう．char の配列と int の配列の union（共用体型）を用意して，int の配列の方でハッシュなどの計算を行うようにすれば，少し速くなるはずである．しかし，int の要素 1 個が char の要素何個に相当するかは，計算機の機種によって異なる．ポータビリティの問題を無視してまで，union を使うべきだということもなかろう．

■幅優先探索による解法

幅優先探索の概略を箇条書きの形で示すと，次のようになる．

(1) 初期状態を表に入れる．これがステップ数 0 の状態である．
(2) ステップ数を表す変数 step の値を 0 にする．
(3) ステップ数 step の状態の中に目標状態と同じものがあれば処理を打ち切る（解が見つかった）．また，ステップ数 step の状態が 1 つもない場合も処理を打ち切る（目標状態に到達できないことが分かった）．
(4) 変数 step の値を 1 だけ増やす．
(5) ステップ数 step-1 の状態から 1 手で移動できる状態を順次生成し，それらが表になければ，表に入れる．これらがステップ数 step の状態である．
(6) (3) に戻る．

初期状態からのステップ数が同じである状態を 1 つに束ねて扱う必要がある．この目的のため，ここでは状態の線形リスト（リンクリスト）を使う．型 state の宣言を次のように変えよう．

```
typedef struct state {
  char a[4*N];
  struct state *link;
  struct state *next;
} state;
```

メンバ a と link は，前に述べたとおり．next が状態の線形リストを作るためのメンバである．リスト上の次の要素を指すポインタを入れる．

変数としては，次のようなものを用意する．

```
int step;
int solved;
state *state_chain;
state *solved_state;
```

step は，概略の説明にあったとおり，初期状態からのステップ数を表す．solved は，探索の途中で目

標状態に等しいものが見つかった場合に1ずつ増やしていく変数である（初期値は0）．state_chainは，同一世代に属する（初期状態からのステップ数が同じ）状態のリストの先頭を指す．solved_stateは，表の中にある目標状態（最初に表に入れておく）へのポインタである．

幅優先探索を行うプログラムは次のようになる．

```
int find_solution(void)
{
  state *p;

  solved = 0;
  solved_state = NULL;
  state_chain = NULL;
  insert_table(final);
  solved_state = state_chain;
  state_chain = NULL;
  insert_table(init);
  step = 0;
  for (;;) {
    if (solved > 0)
      return (step);
    if (state_chain == NULL)
      return (-1);
    step++;
    p = state_chain;
    state_chain = NULL;
    while (p != NULL) {
      generate_moves(*p);
      p = p->next;
    }
  }
}
```

初めに目標状態と初期状態を表に入れてから探索を始めている．あるステップの状態生成が終わった時点で，そのステップ数を持つ状態がすべてstate_chainの指すリストに入っている．次のステップでは，state_chainの値を変数pに代入してから，state_chainにNULLを代入する．これで，次のステップの状態がまだ1つもできていない状況を設定できた．この時点で，pが指すリストに1つ前のステップの状態が1列になって入っているから，そのそれぞれについて最大4とおりの移動でできる新しい状態を作って，それらを表に入れていけばよい．

関数insert_tableは，前に示したsearch_tableに若干の処理を追加しただけのものである．次のようなプログラムになる．

```
void insert_table(state x)
{
  state *p;
  int new;

  p = search_table(x, &new);
  if (new) {
    p->next = state_chain;
    state_chain = p;
  }
  else
    if (p == solved_state)
      solved++;
}
```

新しく表に入れた場合は，現世代の状態だったということになるので，state_chainのリストの中に挿入する（リストの先頭に入れている）．そうでなければ，既存の状態なので，目標状態かもしれない．目標状態かどうか調べて，目標状態であれば，変数solvedの値を1増やす．

本当は，目標状態に到達したことが分かった瞬間に，insert_tableもfind_solutionも終わりにしてよい．幅優先探索だから，最初に見つかった解が最短手数の解である．しかし，このような途中打切りのプログラムをCで書くには，ライブラリ関数setjmpとlongjmpの組合せを使う必要がある．これはちょっとトリッキーなので，このプログラムでは該当ステップ数の状態すべてを生成するまで実行を続けるようにしている．

最後に，関数generate_movesは次のようなものである．この関数の中で，ある状態から1手で移動できる最大4種類の状態を順に生成し，それぞれについてinsert_tableを呼び出す．

```
void generate_moves(state x)
{
  state y;
  int i, j, v;

  for (i = 0; i < 4*N; i++) {
    if (x.a[i] != 0)
      continue;
    assert(i%N != 0);
    if (x.a[i-1] == 0 ||
        x.a[i-1]%10 == M)
      continue;
    v = x.a[i-1]+1;
    for (j = 0; j < 4*N; j++)
      if (x.a[j] == v)
        break;
    assert(j < 4*N);
    y = x;
    y.a[i] = v;
    y.a[j] = 0;
    insert_table(y);
  }
}
```

外側のfor文でギャップの位置を探している．ギャップが見つかったら，内側のfor文で，ギャップの左側のカードの次のカードを探す．ゲームのルールどお

りにやっているだけで，特に難しいことはない．

筆者などは，ギャップの位置を探すだけのためにループを使うことに抵抗を感じる．状態の中に，ギャップの位置を記録する変数（メンバ）を4個用意しておけば，ループなしでギャップの位置を決められるはずである．しかし，これは労多くして効果の少ないプログラムの書き方らしい．実際にプログラムを書いて比べてみたところ，最適化なしなら確かに速くなるのだが，最適化を指定してコンパイルしたところ，逆に上に示した単純なプログラムの方が速くなってしまった．最近の計算機は速いので，この種の細かな工夫はあまり意味がないと言って差し支えないようである．

■深さ優先探索による解法

深さ優先探索の場合は，探索済みの状態に，その状態からの探索の結果を記憶させておく必要がある．したがって，状態を表す型は次のようにしなければならない．

```
typedef struct state {
  char a[4*N];
  struct state *link;
  int to_goal;
} state;
```

メンバ a と link は，すでに述べたとおり．to_goal は，その状態から目標状態に到達するまでに要するステップ数である．目標状態に到達できない場合は，無限大を表す次の値（またはこれより大きな値）を入れると約束する．

```
#define Infinity    100000
```

深さ優先探索を駆動するプログラムは次のようになる．

```
int find_solution(void)
{
  state *p;
  int dummy, result;

  p = search_table(final, &dummy);
  p->to_goal = 0;
  result = backtrack(init);
  if (result >= Infinity)
    result = -1;
  return (result);
}
```

最初に目標状態を表に挿入し，その to_goal を0にする．そして，初期状態 init を引数として，深さ優先探索を行う関数 backtrack を呼び出している．

関数 backtrack は，次のものである．ごく普通のバックトラック法のプログラムである．

```
int backtrack(state x)
{
  state y, *p;
  int new, i, j, v, min, steps;

  p = search_table(x, &new);
  if (!new)
    return (p->to_goal);
  p->to_goal = Infinity;
  min = Infinity;
  for (i = 0; i < 4*N; i++) {
    if (x.a[i] != 0)
      continue;
    assert(i%N != 0);
    if (x.a[i-1] == 0 ||
        x.a[i-1]%10 == M)
      continue;
    v = x.a[i-1]+1;
    for (j = 0; j < 4*N; j++)
      if (x.a[j] == v)
        break;
    assert(j < 4*N);
    y = x;
    y.a[i] = v;
    y.a[j] = 0;
    steps = backtrack(y);
    if (steps < min)
      min = steps;
  }
  p->to_goal = min+1;
  return (p->to_goal);
}
```

最初に，search_table を呼び出して，状態 x が表の中にあるかどうかを調べる．表の中にあれば，その状態は探索済みなので，前回の探索の結果得られた to_goal の値を取り出して返すだけでよい．

表の中になかった場合は，幅優先探索の generate_moves と同様にして4種類の状態を生成し，それぞれについて backtrack を再帰的に呼び出す．こうして得られた4種類のステップ数の最小値に1を加えたものが元の状態 x から目標状態までのステップ数になる．この値を to_goal に記録した上で，関数の値として返す．

すでに述べたとおり，Gap には状態のループがないが，上のプログラムはループがあっても正しく動くように書いてある．4とおりの移動を調べる前に行っている p->to_goal = Infinity; という代入がそのための仕掛けである．再帰呼出しによって，元の状態 x に到達することがあったとしたら，無限大の値が返る．ループに遭遇した場合は，目標状態に到達できないと考えてよいので，これがこの場合の答として正しい値である．

■2つの解法の比較

2つの解法の性能は，ほとんど変わらない．メモリの量も，計算時間も，ほとんど同じである．

強いて言えば，生成される状態の数は，幅優先探索の方が少しだけ少ない．幅優先探索の場合は，あるステップ数で解が見つかったら，それより先の探索はしない．これに対して，深さ優先探索は，すでに述べた事情から，どこかで解が見つかっても，ゲームのグラフすべてを調べ終わるまで探索をやめない．この差が生成される状態の数の差となって現れる．しかし，この差はごくわずかである．Gapのゲームの場合，2つの解法の状態数が大きく違う例はなさそうである．

一般に，最短手数を求める問題の場合，最短手数が判明するまでの計算が一直線になることから，幅優先探索の方が明快である．深さ優先探索だと，ループがあったらどうするか，合流があったらどうするかなど，考えるべきことが多い．これに対して，幅優先探索のプログラムだと，シナリオどおりに書けばよく，面倒な配慮が要らない．

しかし，Gapの場合，幅優先探索にそれほどの優位性はないようだ．状態の記憶が必要なことさえ突き止められれば，深さ優先探索でも難しいプログラミングは必要ない．幅優先探索と深さ優先探索の優劣は，ほとんどないと言ってよさそうである．どちらのプログラミングが得意かを考えて解法を選ぶとよいかもしれない．プログラミングのしやすさは，深さ優先探索がわずかに上のように思われる．

審判データを入力して実行させたときの計算時間は，幅優先探索，深さ優先探索とも8秒だった．筆者のやや古い計算機を使って，最適化なしでコンパイルした結果である．

■問題自体の評価

選手の立場に立ってこの問題を見ると，計算時間にしても，必要メモリ量にしても，どの程度になるか見当がつかない点に不安を感じると思う．自分たちの書いたプログラムで制限時間内に解けないことを恐れて，この問題を敬遠したくなる気持ちはよく分かる．実際，この問題に挑戦したチームは少なかった．34チーム中，挑戦したのは4チームだけであった（うち3チームが正解）．

しかし，この問題は，高級な最適化の技法を要求する性格のものではない．うまい技法によって，計算時間の大幅削減が可能になるとは到底思えない．素直に（強引に）グラフの探索を行うしか手がないのである．そういう問題だと見抜くことは十分可能だと思う．それができるのなら，審判団が計算時間やメモリ量の見通しなしに出題することはないと信用してほしかった．

本来のGapは，ランクが1～13の52枚すべてのカードを使って行うゲームである．それをランク7までに限ったのは，十分に短い計算時間で，かつ過大にならない程度のメモリ量で解けるように配慮した結果なのである．

メモリ量について言えば，今まで知られている中で，探索中に生成される状態の数が最も多くなるのは，**図-7**のような配置である．この状態から探索を始めると，83521個の状態が生成される．現在の計算機の持つメモリ量から見れば，どうということのない数である．これより状態数の多い配置がないと断言はできないが，あったとしても，大差はないと思われる．

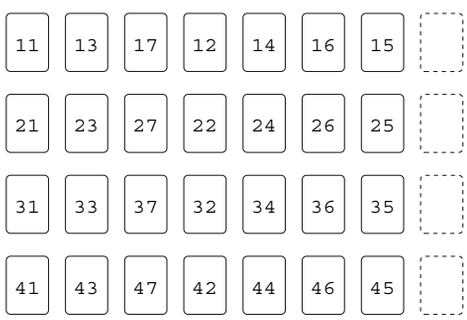

図-7　状態数が最も多くなる配置

とは言え，どの程度の量のメモリが必要か，問題文から判断できないのは，この問題の大きな欠点である．実際にやってみれば，さほどのメモリは必要ないと分かるのだが，それを言い訳にするのは，少し無理があるだろう．

結局，この問題は，幅優先探索ないし状態記憶ありの深さ優先探索さえできれば，決して難しい問題ではない．むしろ，やさしい部類に属する問題ではないかと思う．見かけの難しさにだまされてはならない．

（平成16年4月15日受付）