倉庫番パズル

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

今回取り上げる問題は2000年筑波大会のProblem C「Push!!」である．図-1のような盤面上で，倉庫番（warehouseman）が荷物を一度に1マスずつ押すことができる．荷物をゴールまで移動させるのが目標である．

コンピュータゲームやパズルに詳しい人は知っていると思うが，これは1980年代はじめにThinking Rabbit社から発売され一世を風靡したゲーム「倉庫番」を元にした問題である．盤面の大きさを7×7以下に，荷物の数を1つに制限しているため問題としてはやさしくなっている．

求めるのは最短手数，すなわち荷物の移動回数の最小値であり，人の動きは手数に数えない．図-1の問題では，灰色の矢印で示された手順が最短で，答えは5となる．解のない初期配置に対しては−1を返すことが求められている．

図-1の問題に対応する入力は以下のようになる．

```
5 5
0 0 0 0 0
4 2 0 1 1
0 1 0 0 0
1 0 0 0 3
1 0 0 0 0
```

盤面の幅と高さのあとに，初期配置における各マスの状態（0: 空のマス，1: 壁，2: 荷物，3: ゴール，4: 人）が続く．ゲーム中は人はゴールと重なることがあるが，初期配置では重ならないことが保証されている．

今回も7月号と同様に，なるべく手を抜いてプログラムを記述する方法に重点を置いて，C++とSTL（Standard Template Library）を利用してプログラムを記述する．

■準備

パズルを解くアルゴリズムとしては定石的なものがいくつかある．どのアルゴリズムが適しているかは，パズルの性質に依存するところが大きい．倉庫番パズルでは，以下の性質を考慮に入れる必要がある．

- サイクルの存在
  倉庫番パズルでは数手後に元と同一の局面に戻るサイクルが存在する．図-2のような長さ2のサイクルだけでなく，さらに長手数のサイクルも存在するので，全探索ですべての局面を求めようとするときには，サイクルの存在をチェックし，検出したときは探索を打ち切るようにしないと探索が終了しない．
- 合流の存在
  倉庫番パズルでは図-3のように別の経路をたどって同一の局面に至ることが頻繁にある．合流を無視して，局面A–局面Bを通過して至る局面Dと局面A–局面Cを通過して至る局面Dとを別局面として別々に探索すると，指数オーダの計算がかかってしまう．

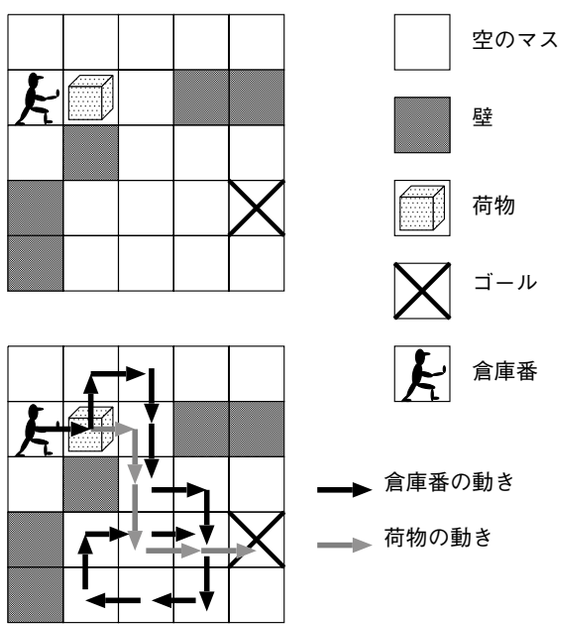

図-1　問題例

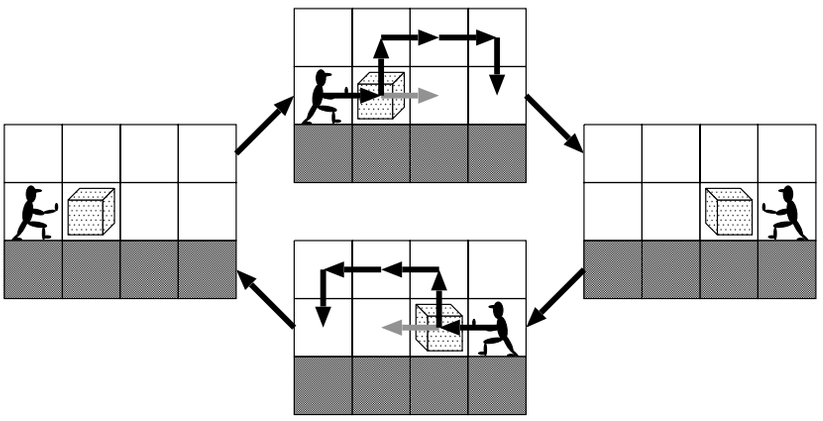

図 -2　サイクルの例

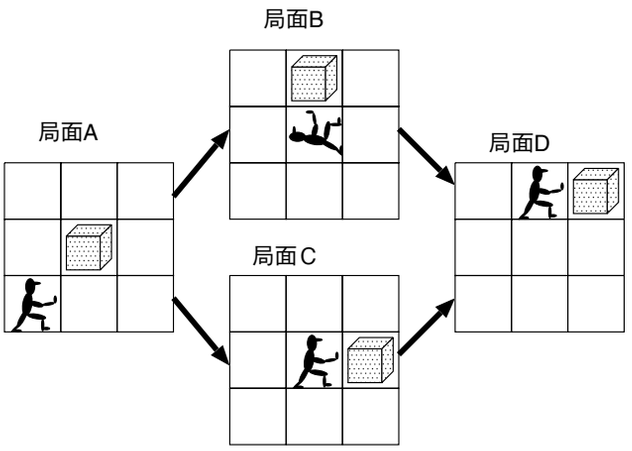

図 -3　合流の例

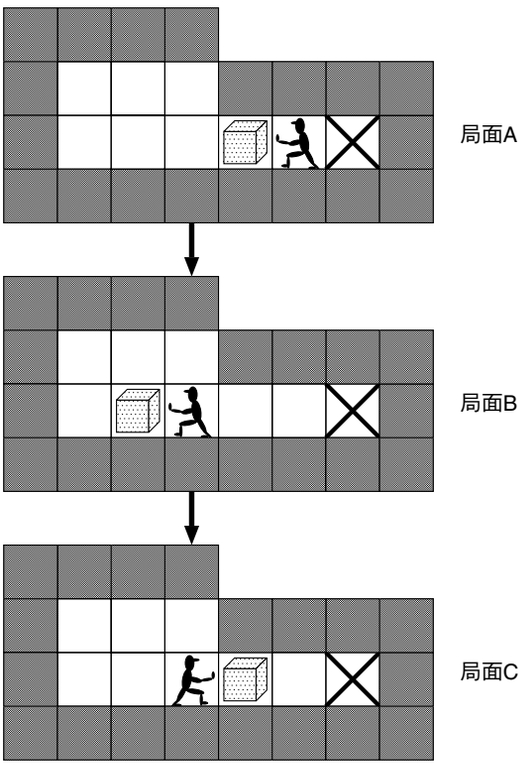

図 -4　人の位置の必要性

サイクルも合流もないパズルでは，局面を表現するデータを世界に 1 つだけ用意して，探索の途中で手を進めたり，戻したりするときに局面の更新を行うという単純な方法で効率よく解ける．それに対して，サイクルや合流があるパズルにおいて効率のよい探索を行うには，局面の同一性をチェックする必要があり，局面を表現するためのメモリのサイズをなるべく小さくすることが重要になる．

この問題では，初期配置から壁の位置，ゴールの位置は動かないので，ゲーム中の局面は荷物と人の位置だけを保持すればよい．

人の動きは手数に数えないので荷物の位置だけで局面を表現可能だと思うかもしれないが，荷物が人の動きを制限するのでそれでは不十分である．図 -4 の局面 A から解にたどりつくためには，4 手かけて局面 C を経由する必要があるので，局面 A と局面 C は荷物の位置が同じであっても，別局面と見なす必要がある．

位置を表す型 Point は 0 から（幅×高さ − 1）の整数値で表すことも可能だが，こんなところでメモリを節約しても，プログラムが見にくくなるだけなので，自然に x 座標と y 座標の対で扱うことにする．

ここでは，保守性などを考慮せずに簡潔にプログラムを作成することを目的として，algorithm ヘッダの中の pair を使って，

```
typedef pair<int,int> Point;
```

とする．比較演算子が自然な辞書式順序で定義されているで，set， map 等の連想コンテナもそのまま使える．

Point に必要な演算子と 4 方向のベクトルを定義しておく．

```
// デバッグ表示
ostream& operator<<(ostream& os,
                    const Point & p){
```

```
  return os << "(" << p.first
            << "," << p.second << ")";
}
// 加算
Point operator+(const Point& p1,
                const Point& p2){
  return Point(p1.first+p2.first,
               p1.second+p2.second);
}
// 減算
Point operator-(const Point& p1,
                const Point& p2){
  return Point(p1.first-p2.first,
               p1.second-p2.second);
}
// 4方向のベクトル
const Point dirs[]={
  Point(1,0), // 右方向
  Point(0,-1), // 上方向
  Point(-1,0), // 左方向
  Point(0,1), // 下方向
};
```

ゲーム中の局面を表す型 State も同様に，荷物の位置と人の位置からなる pair で定義する．

```
typedef pair<Point,Point> State;
```

各問題を保持するクラス Problem を定義する．盤面が 7 × 7 以下に制限されているので，壁の配置は 9 × 9 の固定サイズ（番兵として外側に 1 重の壁を定義する）の 2 次元配列で表すことも可能だが，ここでは自然に vector<int> の vector で表現することにする．

```
const int Nothing=0;
const int Pillar=1;
const int Cargo=2;
const int Goal=3;
const int Man=4;

// 壁の配置の1行分
typedef vector<int> Line;
// 壁の配置全体
typedef vector<Line> Mat;

// 問題
struct Problem{
  int width, height; // 幅と高さ
  Mat mat; // 壁の配置
  Point goal; // ゴールの位置
  State stat; // 初期局面
  // ストリームから Problem を読み込む
  Problem(istream& is){
    // 省略
  }
  // 以下では，アルゴリズムによって，
  // solve1, solve2 等の名前で表す
  int solve(){
    // この部分を定義する
  }
}
```

■簡単な解法

前章で定義したデータ構造を使って解を求めるには，以下のような単純なアルゴリズムで実現できる．未処理の局面の集合である next，出現済み（next に含まれるものも含む）の局面の集合 done を用いる．next には最初に初期局面だけ入れておく．

1．next から初期局面からの手数が最小である局面を 1 つ取り出す．next が空のときは，問題に解がないことが分かるので −1 を返す．
2．人を 4 方向に動かせるなら動かす．
   (a) 移動後の局面が done に登録済みなら，何もしない．
   (b) 荷物を押した場合は移動後の荷物がゴールに一致したら終了，一致しない場合は，その局面の初期局面からの手数と局面を対にして next に登録し，done にも登録する．
   (c) 空のマスに移動しただけの場合もその局面の初期局面からの手数と局面を対にして next に登録し，done にも登録する．
3．1 に戻る．

局面から手数（初期局面からの）への対応に別の map を定義してもよいが，ステップ 1 で最小のものを取り出す必要があるので，手数と State の対の priority_queue を作ることにする．この対は pair により定義し，IState と名前をつける[☆1]．

```
typedef pair<int,State> IState;
```

上のアルゴリズムをプログラムにすると以下のようになる．

```
int Problem::solve1() {
  // IState のヒープの定義
  // 対の第1要素の小さいものから取り出すので，
  // greater で比較する
  priority_queue<IState,
                 vector<IState>,
                 greater<IState> > next;
  // 初期手数(0)と初期配置
  next.push(IState(0,stat));
  // 出現済み集合
  set<State> done;
  // 出現済み集合に初期配置を加える
  done.insert(stat);
  while(!next.empty()){
    // 最小要素の取り出し
    IState nextPair=next.top();
```

☆1 手数 n と手数 n+1 の局面の集合を別々に管理すれば priority_queue を使う必要はないが，後での使い回しを考えてこうしている．

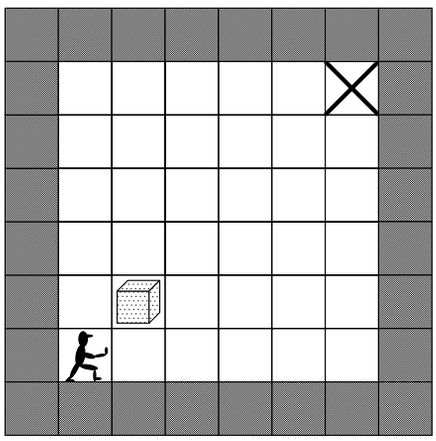

図 -5　最も探索局の多かった問題

```
      next.pop();
      // 初期局面からの手数の取り出し
      int step=nextPair.first;
      State stat=nextPair.second;
      Point cargo=stat.first;
      Point man=stat.second;
      // 4 方向についてチェック
      for(int d=0;d<4;d++){
        Point to=man+dirs[d];
        // 行先に荷物がある
        if(to == cargo){
          Point nCargo=to+dirs[d];
          // 荷物移動先が空なら運べる
          if(mat[nCargo.first][nCargo.second]
             ==Nothing){
            // 手数 step+1 の解を見つけた
            if(nCargo==step){
              return step+1;
            }
            // 新たな局面の作成
            State nStat(nCargo,to);
            // done のメンバでなければ登録
            if(done.find(nStat)==done.end()){
              done.insert(nStat);
              next.push(IState(step+1,nStat));
            }
          }
        }
       // 荷物を押さない移動
        else if(mat[to.first][to.second]
                ==Nothing){
          State nStat(cargo,to);
          // done のメンバでなければ登録
          if(done.find(nStat)==done.end()){
            done.insert(nStat);
            next.push(IState(step,nStat));
          }
        }
      }
    }
    return -1;
}
```

この簡単なプログラムで，筑波大会で使われたテストデータは余裕で解ける．チェックした局面数は最も難しいデータ（図 -5）に対してでも 1156 だった．図 -5 は人間にとってはきわめてやさしい問題だけに意外に思えるが，このアルゴリズムは自由度が高い問題ほど時間がかかることから当然の結果と言える．

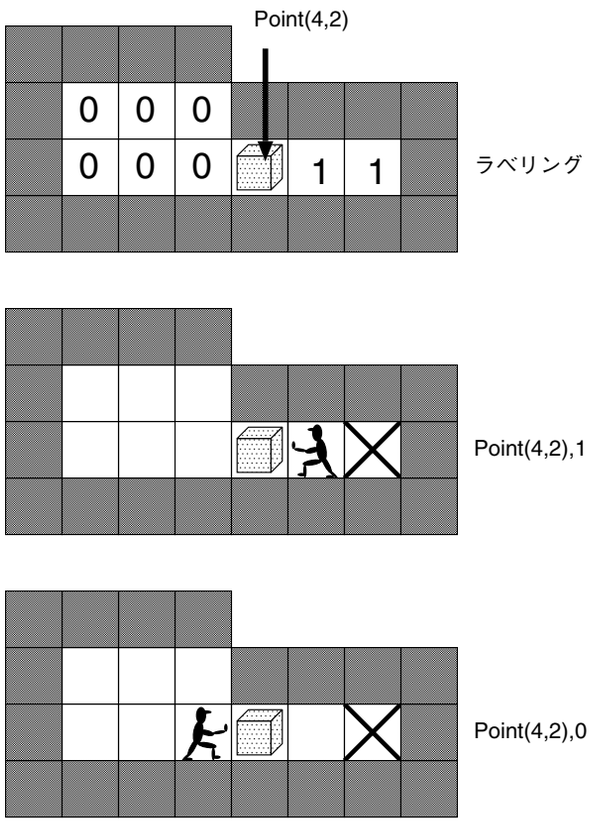

図 -6　ラベリング

■ラベリング

前章では局面を荷物の座標と人の座標の対で定義した．人が動き回るのは手数に数えないので，荷物を動かさずに人が動き回った状態を同一局面で表現すると，探索する局面数を減らすことができる．

そこで，壁と荷物で区切られた閉領域にラベルをつけ，荷物の位置と，人がどの閉領域にいるかの対で局面を表現することを試みる．図 -6 の上のように，閉領域に 0 から始まるラベルをつける．図 -6 中と下は荷物の位置は同じ Point(4,2) だが，局面が異なることは人がいる閉領域のラベルで区別できる．

ラベリングは以下のようなアルゴリズムで容易に実現できる．

1．領域を左上から右下まで順に走査していく．

2．ラベルがついていない空白のマス（空のマス，ゴール，人）を見つけたら，新たなラベルを割り当て，そこから閉領域のそのラベルでの塗り潰しを開始する．

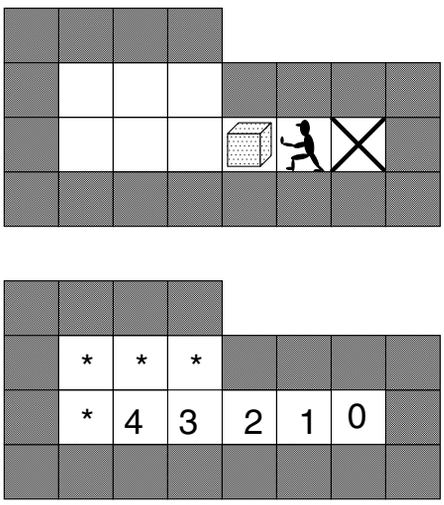

図 -7　push 距離

これまで Point の対で表していた State を荷物と人の位置のラベルとの対で，

```
// 荷物と人の位置のラベルとの対
typedef pair<Point,int> LState;
// 初期状態からの手数と LState の対
typedef pair<int,LState> ILState;
```

のように表すようにプログラムを書き換えてみる．アルゴリズムとしては，前章のものに加えて，

- 局面を next から取り出したり，荷物を動かして登録する前に，ラベリングを行う．
- ある局面から次の手の生成の際は，荷物を基点にして，4 近傍のラベルが局面中の人のラベルと一致するかをチェックした上で，そこに人が来たときに荷物の移動が可能かをチェックする．

という変更を施したものになっている．

前章では異なる局面とされたものが同一と見なされるようになり，展開する局面数は劇的に減少する．前章のプログラムでは展開する局面は最大 1156 だったが，こちらでは 34 まで減っている．

■ push 距離を使った A*

前章までのプログラムは，「初期局面からの手数が小さい局面」から順に調べていった．そのため，ゴールとは反対方向に動かす方向に自由度があると，そちらを動かす手を調べて，ゴールの方になかなか向かっていかない．良い「局面からゴールまでの手数の下限」を求めれば，「初期局面からの手数と局面からゴールまでの手数の下限の和」が小さい有力な局面から調べていくことができる．これは，A* と呼ばれる最適化アルゴリズムにあたる．

「局面からゴールまでの手数の下限」として，たとえばマンハッタン距離（x 座標の差の絶対値と y 座標の差の絶対値の和）も使えるが，ここでは，「人は非連結な領域間でも自由に行き来できる」という条件での手数（push 距離）とする．これは，ゴールからスタートして 1 手ずつ逆に引っ張ることを考えると計算できる．

図 -7 に例を示すが，そこに引っ張ろうとすると人が壁と重なった状態でないと引っ張れない場所の距離は「*」としてある．そこに荷物を押すと，ゴールには辿り着かない．プログラム中では，BIGVAL という大きな値を入れておく．「初期局面からの手数と局面からゴールまでの手数の下限の和」が BIGVAL を超えている場合は，next に登録する必要がない．

プログラムは，前章のプログラムに以下の変更を施したものになる．

- あらかじめ，ゴールからすべての位置への push 距離を計算し，Mat 型の変数 dists に入れておく．
- 最初に next に入れる初期配置の ILState の第 1 要素を 0 から，初期配置の荷物の位置 dists[cargo.first][cargo.second] にする．なお，この値が BIGVAL のときは，解は存在しないことが分かる．
- 荷物を押したときは，ILState の第 1 要素から，押す前の位置のゴールからの push 距離を引いて手数を求め，それに 1 を足した後で，押した後の位置のゴールからの push 距離を足して LState と合わせて next に入れる．

これで，ほとんど迷いがなく解に一直線に向かっていくようになる．最大局面数は 16 になる．

■一般の問題

前章のプログラムを元に，荷物（およびゴール）の数を複数にした一般の問題を扱うプログラムを考えてみる．

```
typedef pair<Point,int> LState;
```
を
```
typedef pair<vector<Point>,int> LState;
```

として（同一性のチェックのため vector<Point> はソートしてから保持する）．「局面からゴールまでの手数の下限」を，「すべての荷物について，最も近くのゴールまでの距離の和」とすると，荷物 1 つ用のプログラムをほとんど変更せずに，一般の倉庫番パズルのプ

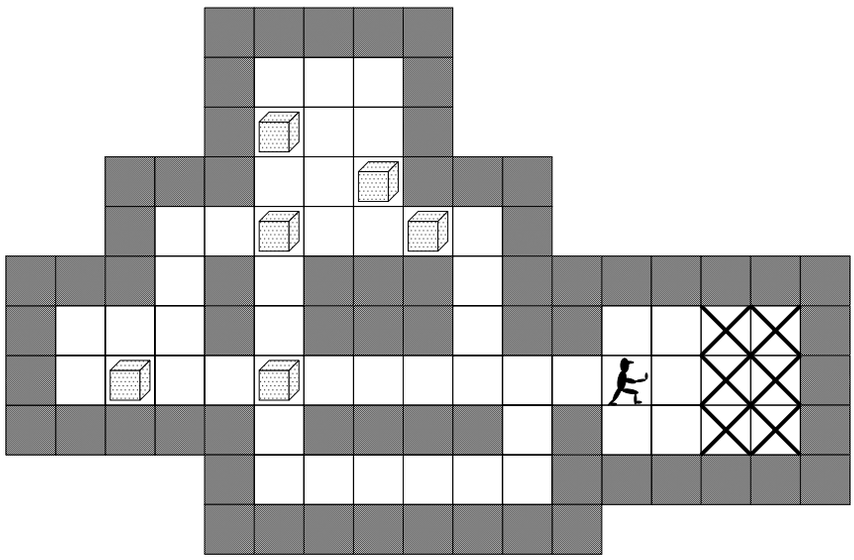

図 -8　一般の問題

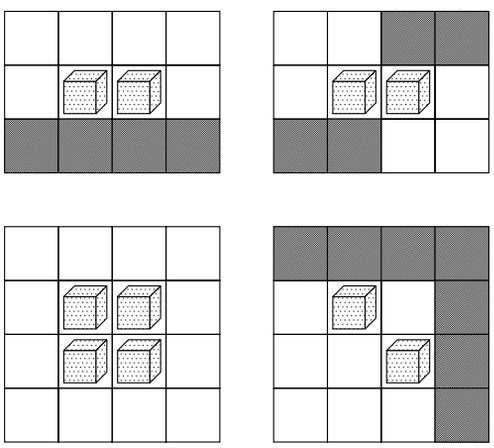

図 -9　死に手の例

ログラムを作成できる．

これでプログラムは動くのだが，実行時間がかかってとても使い物にならない．たとえば，オリジナルゲームの第 1 問である**図 -8** を解かせてみても，100 万以上の局面を探索して，遅い計算機では分単位の時間をかけて，やっと答を出すという状態で，それ以上難しい問題になると解く気がしない☆2．

実は，一般の倉庫番パズルは PSPACE 完全問題であることが知られている [1]．PSPACE 完全とは問題のサイズに対して多項式オーダのメモリを使用して解ける問題の中で最も難しい問題のクラスに属するということである．NP 完全問題も PSPACE 問題に含まれるので，PSPACE 完全問題は NP 完全問題よりも難しいと考えられている☆3．

倉庫番パズルが PSPACE 完全問題であることは，解の手順が問題のサイズに対して指数オーダになるような問題が容易に作れることから，直感的に理解できるだろう☆4．これに対して，NP 完全問題，たとえばハミルトン巡回路問題の場合は，解が存在するときに，ノード数に比例するステップ数で正解手順をトレースできる．

以上のことから，一般の倉庫番パズルを解く効率的なアルゴリズムが存在しないことが分かるが，少しでも多くの問題を解けるようなプログラムの作成が，多くの人により試みられている．

そのテクニックの 1 つとして，解が存在しない局面をパターンにより判定するというものがある．一般の倉庫番パズルでは，荷物同士が邪魔をして荷物をそれ以上動かせなくなる（あるいはゴールに近づかないサイクルの動きしかできない）ことがある．これを，文献 2）では死に手と呼んでいる．死に手が局面に含まれたら，探索を打ち切ると効率が大きく改善されるが，**図 -9** のような単純なパターンだけでなく，さまざまなパターンが出現するので，これらを効率よく判定することがプログラムの優劣を左右する．

現在のところ，最も優れたプログラムでも，人間の解ける問題をすべて解くことはできないという状態である．腕に覚えのある人は挑戦してみるとよいかもしれない．

☆2 コンピュータに解かせる場合の難易度は人間が解く場合の難易度とは比例はしないので，人間にとってより難しい問題が短い時間で解けることはある．

☆3 *P≠NP* が未証明なのと同様に *NP≠PSPACE* もまだ証明されていないので，「より難しい」と言い切ることはできない．

☆4 PSPACE 問題に属することの方は自明．

参考文献

1）Culberson, J.: Sokoban is PSPACEcomplete, Technical Report 97-02, Department of Computing Science, University of Alberta. http://www.cs.ualberta.ca/~joe/Preprints/Sokoban (1997).

2）上野　篤, 中山　康, 疋田輝雄 : 倉庫番を解くプログラム（上, 下）, bit, Vol.26, No.12 and Vol.27, No. 1, 共立出版 (1994-1995).

（平成 14 年 10 月 8 日受付）