パズルの解法

田中　哲朗

計算機の進歩により，人間が解けるパズルを計算機に解かせることは困難な問題ではなくなってきた．プログラミング初級者への練習問題としてパズルを解くプログラムを作成させることは多いが，それよりは一歩進んだ求解アルゴリズムをいくつか紹介する．パズルを解くのではなく，面白いパズルの問題を作ることは現在でも研究の対象になっている．これらについても紹介する．

キーワード：パズル，プログラム，最適化問題，バックトラック，A*

1.　はじめに

何をパズルと定義するかは人によって異なるが，本稿では，以下のような特徴を持つゲームをパズルと呼ぶことにする．

- 一人で遊ぶ．
- 状態の遷移が確定的かつ予想が容易であるか[1]，確率的（ただし分布は既知）である．

この定義では不完全情報パズルや，テトリスのようなリアルタイムパズルゲーム，あるいは詰将棋などもパズルに含まれる．本稿ではこの中でも，特に完全情報パズルに絞って，解法と問題作成に関して説明する．

2.　穴埋めパズルの解法

2.1　問題の定義

一番パズルらしいパズルが，穴埋めパズルの仲間である．ここで穴埋めパズルと呼んでいるのは，図1のような，紙の上で解くパズルの仲間を指す．

これらのパズルは，一般に制約充足問題と呼ばれる以下のような特徴を持つ問題のクラスに属する．

- 選択できる場所は最初から与えられている．
- それぞれの場所で選択できる要素が決まっている．
- 別々の場所での選択に関して制約条件が定まっていて，すべての制約条件を満たすことが解の条件になる．

これらのパズルでは，解が一意であるという条件も加わることが多い．紙の上で解くパズルの場合は，与えられた制約条件を満たすかどうかのチェックを「正解」と一致するかどうかで行うことが一般的だからであろう．

自明なパズルを除くと，これらの問題に解が存在するかどうかという決定問題はNP完全問題であることが多い．正解が示されれば，それを検証するのが容易である（問題サイズの多項式時間で検証可能であ

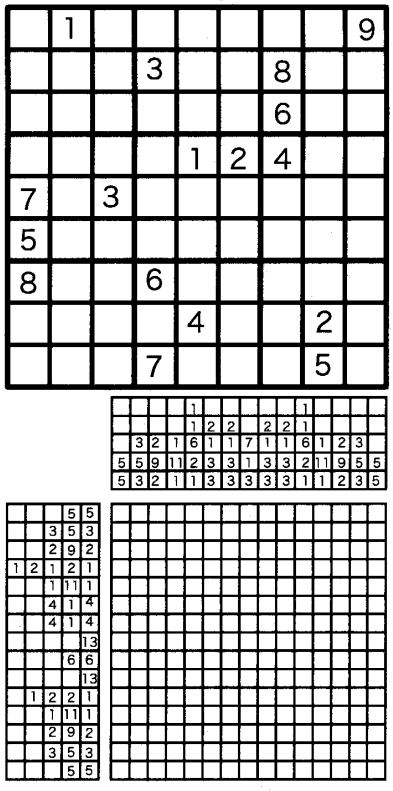

図1　穴埋めパズルの仲間（数独，お絵かきロジック）

[1] ビデオゲームのパックマンはモンスターの移動アルゴリズムが乱数を用いないので習熟プレイヤーにとっては予想可能である．このようなプレイヤーにとってはパックマンはパズルなのかもしれないが，一般プレイヤーを基準に考えるとパズルに含めないのが適当だろう．

たなか　てつろう
東京大学 情報基盤センター
〒113-8658 文京区弥生 2-11-16

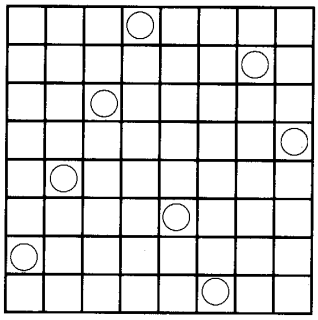

図2　8クイーンの解の一つ

る）ので，クラスNPに含まれていることは明らかである．

NP完全問題は計算機で扱うのが難しい問題であると考えられるが，実際には問題のサイズが小さい場合だけを扱えば良いことが多い．これは，一般にパズルが「人間が解ける問題である」という性質にもよるし，「覆面算の10進数をn進数にする」のようにサイズを大きくして一般化した問題が人間に出題されることがほとんど無いようなパズルも多い．

この手のパズルは，実行効率を考慮しないで書いても実用的なプログラムが作れることが多いので，しばしばプログラミングの練習問題として使われる．しかし，効率に関してはちょっとした工夫で改善されることが多いので，ここでいくつか取り上げることにする．

例としては，Nクイーン問題を扱う．この問題は図2のように，8×8の盤面に8個のクイーン（チェスの駒，将棋の飛車と角の両方の動きが出来る）を，それぞれが互いに利きを持たないように配置するというパズルである．この盤面のサイズを$n \times n$に一般化し，クイーンの数もnにしたものがNクイーン問題になる．Nクイーン問題は表1のように一般に解が複数あり，プログラミングの例題としてよく用いられる．

2.2　モデル化

この手のパズルは，変数とその変数間の制約の組として定義される制約充足問題としてモデル化するが，この際，何を変数にしてどのような制約で表現するかが重要である．

Nクイーン問題の場合は，$n \times n$のマスそれぞれについて，変数を割り当てて，

$x_{ij}=1 \Leftrightarrow i$行$j$列のマスにクイーンを置く

$x_{ij}=0 \Leftrightarrow i$行$j$列のマスにクイーンを置かない

のようにモデル化する方法も考えられるし，

$x_i=j \Leftrightarrow i$列にクイーンを置くのはj行

表1　Nクイーン問題の解の個数（文献[1]より）

N	解の個数
1	1
2	0
3	0
4	2
5	5
6	4
7	40
8	92
9	352
10	724
11	2,680
12	14,200
13	72,712
14	365,596
15	2,279,184
16	14,772,512
17	95,815,104
18	666,090,624
19	4,968,057,848
20	39,029,188,884
21	314,666,222,712
22	2,691,008,701,644
23	24,233,937,684,440
24	227,514,171,973,736

とモデル化する方法も考えられる．

前者のモデル化と比較すると，後者のモデル化は各列にクイーンを1つしか置かないという制約を含んだモデルになっているため，探索空間が小さくなるというメリットがある．実際，制約を入れない場合は，前者が2^{n^2}の解空間を持つのに対して，後者はn^nであり，$n \geq 1$の元ではつねに後者が小さくなる．以降では主に，このモデル化を前提に議論を進める．

後者のモデル化を用いた時の制約は，以下のように書ける．

$$\forall_{i,j,i \neq j} \quad x_i != x_j \wedge |x_i - x_j| \neq |i-j|$$

2.3　生成後検査法（Generate and Test）

最もナイーブな解法は，可能な変数の組み合わせを生成していき，制約をすべて満たしたものを解とする解法である．これは生成後検査法と呼ばれる解法である．Nクイーン問題の場合は，変数の組み合わせの生成の際に同じ数を2回含まないようにすると$n!$回のチェックを行うことになるが，nが大きくなると実用的でなくなる．

2.4 バックトラック法 (Backtracking)

可能な変数の組み合わせをすべて生成するプログラムは，変数を1つずつ決めていく再帰的なプログラムとして書くことができる．この過程は図3のように，木構造で表すことができる．

図3のAのように，変数1，変数2を具体化した時点で制約を満たさなくなることが判明した場合は，その組み合わせを打ち切ることができる．そして，次にBのように変数2を別の値に具体化することを試みる．この方法は，バックトラック法と呼ばれ，パズルを解く場合の基本となるアルゴリズムとなっている．

Prologのようにバックトラックの仕組みを取り入れたプログラミング言語もあるし，そうでないプログラミング言語でも多くの場合，再帰を使って自然に書くことができる．

例えば，Haskellのような遅延評価関数型言語の場合は，以下のように簡潔に書ける．

```
queens 0=[[]]
queens(n+1)=
    [q : b | b<-queens n, q<-[0..7], safe q b]
safe q b=and[not(checks q b i) | i<-[0..(length b-1)]]
checks q b i=q==b !!i | | abs(q-b !!i)==i+1
```

上のプログラムは一見するとすべての解を求めるプログラムに見えるが，解を一つしか必要としない場合は，遅延評価の仕組みによって余計な計算が行われないようになっている．

2.5 動的変数順序付け (Dynamic Variable Ordering)

バックトラック法をプログラミングする場合，一番自然な方法は変数を決まった順序（例えば，Nクイーン問題の場合は列1,2,3の順）で具体化していく方法である．

しかし，なるべく選択肢が少なく，かつ他の具体化していない変数との制約の数の多い変数から先に具体化した方が一般的に探索ノード数が少なくなる．例えば，図4は3つの列にクイーンを置いた状態を示す．残りの列のクイーンを置けない場所を灰色，置ける場所を白で示すが，各列ごとに白のマスを数えると，右から3番目の列が最も少ないことが分かる．この場合は，この列から具体化する（クイーンを置く）．

これが，動的変数順序付けと呼ばれる方法である．この方法は，どの変数から具体化するかを決定するのにコストがかかるが，探索ノード数は大幅に減少することが多い．また，選択肢が0である変数がある場合は，ここで枝刈りを行うことができる．

図5の例では，4つのクイーンを置いた時点で，右から2つめの列にクイーンを置ける場所が残っていないことがわかるので，探索を打ち切ることができる．このようなチェックは動的変数順序付けとは独立に行うこともでき，その場合は前向きチェック (Forward Checking) と呼ばれる．

2.6 その他の解法

Nクイーン問題の解の個数は，文献[1]にあるように今のところ24クイーンまで求まっている．この際に使われたプログラムはDynamic Variable Ordering等のテクニックは使っていないものの，ビット演算を利用した高速なプログラムをPCクラスタを使って並列に動かして実現されている．

すべての解を見つけるのではなく，解を1つ見つけるのは，また別な問題となる．Nクイーン問題に関して，いくつかの制約を満たさない不完全な解から局

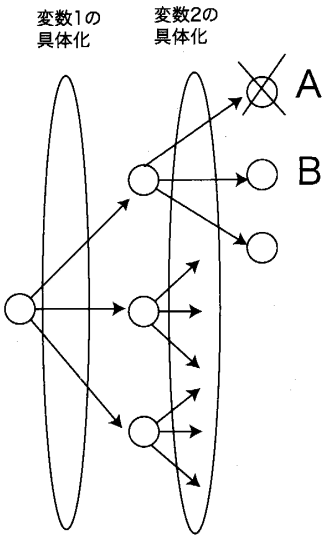

図3 バックトラック法

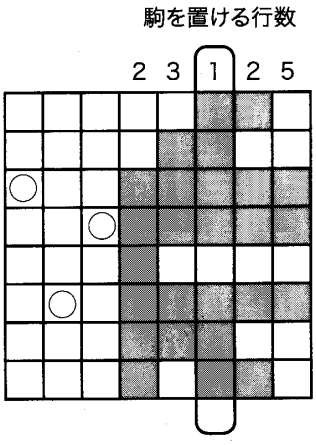

図4 列ごとにクイーンを置けるマスを数える

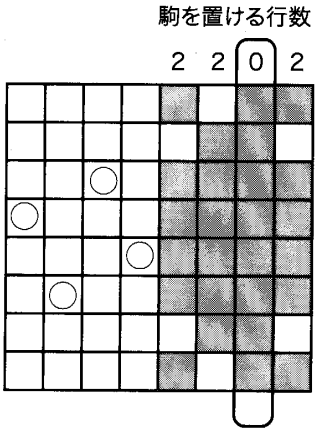

図5　置けない列の発見

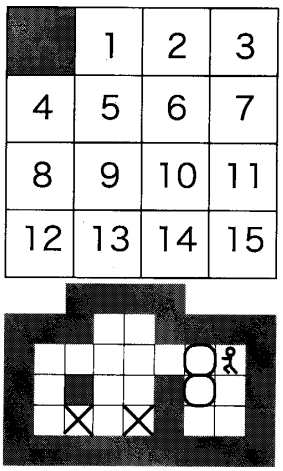

図6　移動型パズルの仲間（15パズル，倉庫番）

所探索で改善するという方法で1,000万を超えるようなNについて解を求めた例がある[2].

一度，変数とその間の制約という形にモデル化してしまえば，CPLEX等の数理計画ソルバを用いて問題を解くことも可能である．文献[3]のようにスリザーリンクと呼ばれるパズルを整数計画法によって定式化し，ソルバを使って解いた研究もある．

今のところは，

- 人間が解けるパズルを解かせる場合は，効率が重要になることが少ない．
- 商用の高性能のソルバは高価である．
- プログラミングそのものを楽しんでいる人が多い．

などの理由で，この手のアプローチは一般的ではないが，GNU Linear Programming Kit[4]のようなフリーのソルバの質も向上してきているので，今後は盛んになっていくものと考えられる．

3. 移動型パズルの解法

3.1 問題の定義

穴埋めパズルと並んでパズルの主流を占めるのが図6のような移動型パズルである．移動型パズルは以下のような性質を持つ．

- 状態と，その状態からの遷移（手）によって問題が定義される．
- 解の一意性は保証されない．解の手順は重視されない．
- 手数が自明には予想できない．

移動型パズルの計算量を考える場合は，注意が必要になる．15パズル（図6の上）を一般化したNパズルは，ある配置が解を持つかどうかを判定する問題の計算量はクラスPに属し（偶奇性をチェックするだけ），実際に解の手順を構成することも多項式時間で可能であるが，最短手数がある数より小さいかどうかを判定する問題の計算量はNP-completeとなる．

一方，倉庫番（図6の下）問題に解が存在するかどうかを判定する問題の計算量はPSPACE完全であることが知られている[5]．倉庫番問題は最短手数が多項式オーダに収まらないような問題もあり，パズルの中では難しい部類に属している．

3.2 全探索

移動型パズルで規模の小さい問題は，全探索ですべての解を求めることができる．探索は多くの場合，木として表現される．単純に全探索を実現するには，深さ優先探索が多くの場合，用いられる．

別の経路を経て同じ局面に至ることが頻繁にあり，再計算のコストが多い場合は局面表（transposition table）を持ち，一度計算した結果を再利用する方法が一般的である．また，ループがあるパズルでは局面表はループを検出するためにも用いられる．

初期配置の数が少ないパズルに関しては，解けた状態からスタートして後退解析（retrograde analysis）により，可能な解すべてのテーブルを作成しておく方法もある．

3.3 幅優先探索と深さ優先探索

解ける場合に，解に至る手順が一意であるパズルよりも，途中で合流があり，複数の解があるパズルの方が多い．このようなパズルで，解を一つ見つけることを目的とする場合は，探索順序が重要になってくる．

深さ優先探索（depth first search）ある経路で探索して手詰りに陥った際に，まだ探索していない枝分かれのうちの，もっとも深い枝を先に探索するというもの．プログラミングは楽．また，局面表を

使わなければ，深さ分のメモリを消費するだけで済む．

幅優先探索（bredth first search）まだ探索していない枝分かれのうちの，もっとも浅い枝を先に探索するというもの．この順序で探索を行うと，自然にもっとも浅い解をみつけることができる．

実装では，探索していない局面を深さ順に優先度キューに入れて，最も浅い局面を取って探索する方法が自然だが，局面を表現するデータ構造が大きい場合は，メモリ効率の点で劣る．

深さを閾値にして，深さ優先探索を行い，閾値を繰り返し大きくしていくことで，幅優先探索と同じ順序での探索を行うことも可能である．

ゲームの性質により，どちらが向いているかが決まる．深さ d までは幅優先探索，それより深い部分は深さ優先探索に切り替えて組み合わせることも可能である．

3.4 最良優先探索

ある局面が解にどれだけ近いかを近似するヒューリスティック関数を用いて，未探索の局面のうち，最も解に近そうな局面を探索するのが最良優先探索である．中でも，A* はよく用いられる．

ある局面からゴールへの距離を見積もる関数 $h(X)$ が，真の距離以下の値を返すことが保証されているとする．

$$cost(X)=distance(root, X)+h(X)$$

とした時に，未探索ノードのうち，$cost(X)$ が最小となるノードから探索していくと，ゴールを1つ見つけると，それが最短経路であることが言える．距離を問題とせずに解を1つだけ見つければ良いという場合も，$distance(root, X)=0$ として A* で探索すると，$h(X)$ が良い見積もりの場合は，早く解にたどり着くことが期待される．例えば，15-puzzle では $h(X)$ として各ピースのゴールでの位置と現在の位置のマンハッタン距離を求め，それを足し合わせたものを用いることが多い．

実装では，探索していない局面を深さ順に優先度キューに入れて，最も浅い局面を取って探索する方法が自然だが，局面を表現するデータ構造が大きい場合は，メモリ効率の点で劣る．

値を閾値にして，深さ優先探索を行い，閾値を繰り返し大きくしていくことによって，A* と同じ順序での探索を行うことも可能である．これを IDA*（Iterative Deepening A*）[6]と呼ぶ．

3.5 乱数を用いた探索

解に至る経路が複数あり，かつその数が十分である場合，行き止まりになるまで，乱数によって手を生成して選んでいく方法も有力である．例えば，

- ある程度以上の手数にならないと解がなく，それを超えるとかなりの確率で解がある．幅優先探索ではそれ以下の手数のノードをすべて探索する必要があるが，メモリや時間の関係でできない．
- ある程度以上の手数でも解のあるなしに関して，局所性があり，ある枝を選ぶとその下には多くのノードがあるのに，解が全くなく，別の枝は解ばかりということがある．この場合，深さ優先探索で，最初に解のない枝を選んでしまうと，その下のノードすべてをチェックしてからでないと，解にたどり着かない．
- 局面を評価する良いヒューリスティック関数がない．そのため，最良優先探索が行えない．

という条件では，この方法が有力になる．この方法は，プログラムとしても簡単になるというメリットはある．この方法の代表的なものとしては，Iterative Sampling[7]がある．

なお，この方法も深さ優先探索や幅優先探索と組み合わせることは可能である．

4. パズルの問題作成

計算機で出題するパズルの場合，初期配置を乱数で与える場合が多いが，完全情報ゲームの場合は倉庫番のように人間が作成した問題セット[2]の場合もある．

乱数で初期配置を与える場合は，最善のプレイ（不完全情報ゲームの場合は，完全情報化して，本来得られない情報も与えたうえで）をしても解けない初期配置を除くことも考えられる．このためには，以下のような方法が考えられる．

1. ゴール局面から出発して，最終手から逆向きにランダムに逆向きの手を生成していく．この作業を適当な回数繰り返した結果，得られる局面を初期局面とする．
2. 乱数で初期局面を生成して，簡単なソルバで解いて，解を見つけることができた問題のみを問題として提示する．このソルバはすべての解のある初期局面を解ける必要はない．解のある初

[2] [8]で計算機による問題の作成の研究もある．

期局面を解けなければ，次の初期局面を乱数生成するだけである．

どちらの方法でも，作成された問題がランダムではなく「癖」が出てしまう可能性はある．1の方は，逆向きの手の生成の回数を超えるような正解手順を持つ問題は作成できないし，そうでなくても複数の経路で正解に至る（やさしい）問題は，唯一の経路でしか正解に至ることのない問題よりも高頻度で生成されることになる．

2はソルバの能力が低い場合は，やさしい問題ばかりが生成されることになってしまうし，ある程度能力が高くても，解ける問題に偏りがあり，特定のテクニックを使った解法が全く解けないということもある．

参考文献

[1] Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba: Solving the 24-queens Problem using MPI on a PC Cluster, Technical Report UEC-IS-2004-6, Graduate School of Information Systems, The University of Electro-Communications (2004).

[2] 久保幹雄，松井知己：組合せ最適化「短編集」，朝倉書店（1999）．

[3] 杉村由花：整数計画法を用いたスリザーリンクの解法，東京大学工学部計数工学科卒業論文（2004）．

[4] GLPK (GNU Linear Programming Kit), http://www.gnu.org/software/glpk/.

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

[6] R. E. Korf: Depth-first iterative-deepening: An optimal admissible tree search, Art. Intell., vol. 27, pp. 97-109 (1985).

[7] Harvey, W. D. and Ginsberg, M. L.: Limited discrepancy search, Proceedings of IJICAI, Vol. 1, pp. 607-615 (1995).

[8] 村瀬芳生，松原仁，平賀譲：「倉庫番」の問題の自動作成，第38回プログラミングシンポジウム資料集（1997）．