数理解析研究所講究録
第359巻 1979年 197-215

バックトラック法の実現について

電通大・計算機科学科
一松 宏　野下 浩平

§1. はじめに

バックトラック法（系統的行きつ戻りつ法，backtracking）は，様々な組合せ的問題を解くのに用いられているプログラミング技法のひとつであり，古くからいろいろな見方から研究されている[10]．バックトラック法をプログラムとして実現する場合，探索の対象となる“候補の集合を計算中に絞ること（preclusion）”は，計算時間に重要な影響を与えるものであり，S．Golomb等の初期の論文[3]以来，基本的な問題として，そのための良い解決が求められてきている．本稿では，探索の型が“順列型”とも称すべきものに対して，ひとつのpreclusionの方法を示す（[4]も参照）．この方法は，D.E.Knuth [7]によってもっと一般的な型の探索にも応用できるように拡張されたが，ここでは，具体的な例題を用いて，従来行なわれてきた普通の方法，ここでの新方法，

よびKnuthの拡張した方法を比較評価しょう．

バックトラック法は，その使い方が多見て，狭い意味での‘計算量の理論’的評価（問題のサイズnに対して手間T(n)の漸近的な振舞を調べること）のみならず，特定のn_0に対して（特定の計算機上で）何時間（何日ということもある）かかるか，あるいはそのプログラムが別のものより何倍速いかという極めて具体的な評価が必要になることも多い．例えば単一の問題に対して解を列挙するとか，解が存在しないことを証明するとかいうような場合がそうである．このような場合，少ない手間で全計算時間を予め見積る方法が重要になる（[6], [8]）．本稿では，バックトラック法による探索で割合よく現われる型のものに対して，簡単な確率モデルを設定しそこで要する計算時間を推測するとともに，前記のprecluisionの効果も評価しょう．

§2. バックトラック法の実現

いま，J_iを有限の順序集合とする（$0 \leq i \leq n-1$）．ある性質P_kにおいて

$P_{k+1}(j_0, \cdots, j_k)$ ならば $P_k(j_0, \cdots, j_{k-1})$ $(0 \leq k \leq n-1)$ が成り立つものとする．ここで$(j_0, \cdots, j_k) \in J_0 \times \cdots \times J_k$である．この時，性質$P_n(j_0, \cdots j_{n-1})$を満たすような解$j_0, \cdots, j_{n-1}$をすべ

て求めるという問題が与えられた時，その解を系統的に教え上げる方法として，バックトラック法という探索手順が，よく知られている．'順列型'バックトラック法による探索は，ALGOL風の言語で一般的な形を書くと次のようになろう[3]．

```
procedure Search; element j;
 begin for each j in S do ---------- select
       if test(j) then
         begin S←S-{j} ---------- remove
               状況の更新; l←l+1;
               if l=N then 解[j0, j1, ···, jN-1]
                      else Search;
               l←l-1; 状況の復元;
               S←S∪{j} ---------- recover
         end
 end;
{主プログラム}
begin S←{0, 1, ···, N-1} ---------- initialize
      状況の初期設定; l←0;
      Search
end
```

さて，ここで $(j_0, \cdots j_{l-1})$ という部分解が，得られていて，$P_{l+1}(j_0, \cdots j_l)$ が成立するか否かをテストする段階で，候補(j_l)をみつける操作とその手間に着目して，上記アルゴリズムをもっと具体的に実現していこう．まず，従来行なわれてきた方法によれば，次のようになる．（なお，'状況の更新・復元'の具体例については，§4参照）

(方法A)[1]

```
initialize  : for j←0 until N-1 do FREE[j]← true;
select      : for j←0 until N-1 do
                if FREE[j] then
remove      : FREE[j] ← false
recover     : FREE[j] ← true
```

この方法Aは，1つのlの値について（$0 \leq l \leq N-1$），Sの要素のselectにNに比例する手間がかかる．一方，次の方法Bによれば，その手間が$(N-l)$に比例するようになる．

(方法B)[4]

```
initialize  : for j←0 until N-1 do
                begin F[j-1] ← j; B[j] ← j-1 end;
```

```
                F[N-1] ← -1 ; B[-1] ← N-1
select      ; j ← -1
                for j ← F[j] while j ≥ 0 do
remove      ; F[B[j]] ← F[j] ;  B[F[j]] ← B[j]
recover     ; B[F[j]] ← j    ;  F[B[j]] ← j
```

この方法では，集合Sが両方向リストで実現される．ここで，配列Fで前向き，Bで後向きのポインタを表わすことにする．またSの要素はこの配列の添字で表わす．

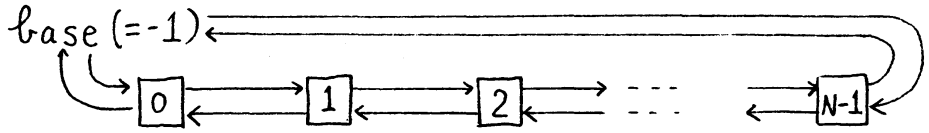

図1. 集合Sの両方向リストによる表現

要素jがリストからremoveされる時点での様子を描いたのが図2である．ここで注意すべき点は，リストからはずされたjに対しても，そのポインタをそのまま生かしておくという点である．

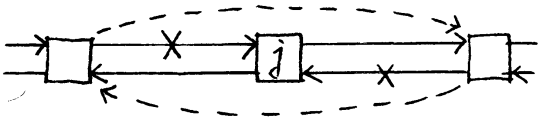

図2. 要素j∈Sのremove（点線が新しいポインタ）

この方法によれば，要素 j を S に recover するのに，一定の手間でできることになる．ここで，remove と recover の操作が正しく働くことは、バックトラック法での探索によれば，最後に remove される要素が最初に recover されるということより納得されよう．

§3. 手間の評価

3.1 大雑把な評価

バックトラック法によるプログラムの全計算時間については，select の手間によって大雑把な評価ができる．既に述べたように，方法Aによる select の手間は、部分解が $[j_0, j_1, \cdots j_{l_0-1}]$ の場合，$l=l_0$ に対して全体で N に比例する．一方，方法Bについては，この手間が $(N-l_0)$ に比例するにすぎない．従って全体の計算時間については、通常方法Bの方が，方法Aより優れているものと判断されよう．

3.2 やや詳しい解析

方法AとBの能率をもう少し詳しく比較しよう．

[定義1] 部分解 $[j_0, j_1, \cdots, j_{l-1}]$ が定まっている時，オ l レベルの test(j) $(j \in S)$ が，true になる確率を $P_N(l)$ とする．

[定義2] オ l レベルの test(j) が true になる回数（の期待値）

を success(ℓ) とする.

Note : $success(\ell) = success(\ell-1) \cdot P_N(\ell) \cdot (N-\ell)$,

$success(\ell) = P_N(0) \cdot P_N(1) \cdots P_N(\ell) \cdot N \cdot (N-1) \cdots (N-\ell)$

が成立する. 便宜上, $success(-1) = 1$ とおく.

[定義3] 方法Aにおいて, 操作selectの対象になる要素数全体を $C_A(N)$ とする. また方法Bにおいて, 同様のものを $C_B(N)$ とする.

Note : $C_B(N)$ は, test(j) の回数と一致する.

さて, $\ell=0$ レベルでの select の対象となる要素数は、ともに N, ℓ レベルで test が true になるごとに, $(\ell+1)$ レベルでの select の対象となる要素数は, 方法Aでは N, 方法Bでは $(N-\ell-1)$ である. これより次の性質が成立つ.

[性質1]

$$C_A(N) = \sum_{\ell=-1}^{N-2} success(\ell) \times N = N + N^2 P_N(0) + N^2(N-1) P_N(0) P_N(1) + \cdots + N \cdot N! P_N(0) \cdots P_N(N-2),$$

$$C_B(N) = \sum_{\ell=-1}^{N-2} success(\ell) \times (N-\ell-1) = N + N(N-1) P_N(0) + N(N-1)(N-2) P_N(0) P_N(1) + \cdots + N! P_N(0) \cdots P_N(N-2)$$

全計算時間を詳しく評価するためには, select 以外の手間も考慮する必要がある. 方法AとBについて, 調べると両者とも全体の手間をほぼ次のように表わすことができる.

$$\sum_{\ell=-1}^{N-2} success(\ell) \times \left[\left\{ \begin{array}{l} remove, \\ procedure\ 呼出し, \\ recover, \\ 状況の更新・復元等 \\ それぞれ1回 \\ あたりの手間 \end{array} \right\} + \left\{ \begin{array}{l} (\ell+1)レベルでの \\ Selectの手間 \\ の合計 \end{array} \right\} + \left\{ \begin{array}{l} (\ell+1)レベルでの \\ testの手間 \\ の合計 \end{array} \right\} \right]$$

[定義4]

R_Aは，方法Aにおいて remove，procedure 呼出し，recover，
状況の更新・復元等それぞれ1回あたりの手間．

R_Bは，方法Bにおいて remove，procedure 呼出し，recover，
状況の更新・復元等それぞれ1回あたりの手間．

S_Aは，方法Aにおいて select 1回あたりの手間．

S_Bは，方法Bにおいて select 1回あたりの手間．

Tは，test 1回あたりの手間とする．

以上のことより，次の性質が成立つ．

[性質2]

Aの手間は全体でほぼ，

$$\sum_{\ell=-1}^{N-2} success(\ell) \times \{ R_A + N \cdot S_A + (N-\ell-1) \cdot T \}$$

Bについては，

$$\sum_{\ell=-1}^{N-2} success(\ell) \times \{ R_B + (N-\ell-1) S_B + (N-\ell-1) T \}$$ となる．

[定義5] $P_N(\ell) = \alpha[1-\beta(\ell/N)]^{\gamma}$ $(\alpha, \beta, \gamma \geq 0)$ の形で、書ける探索の型を $[\alpha, \beta, \gamma]$ uniform とよぶ.

Note1; この型は、いくつかの問題で現われる1つの典型的なものである(文献[5]参照).

Note2; $[1, \beta, 0]$ uniform の場合は、辞書式順序による順列の生成に対応する.

Note3; 与えられた問題に対して, 実際に $[\alpha, \beta, \gamma]$ uniform の形であるかどうかを調べるには, 文献[6]によるモンテカルロ法を適用する手がある.

$P_N(\ell)$ が残った候補の割合の関数であるような場合は、$\beta=1$ となるが, その場合には, 次のような性質が成立つ.

[性質3]

success(ℓ) が最大となる ℓ の値 $(0 \leq \ell < N)$ を ℓ_{max} とおくと, $\ell_{max} = \lceil N - (N^{\gamma}/\alpha)^{\frac{1}{\gamma+1}} - 1 \rceil$ となる. また ℓ_{max} レベルの test が, true になるごとに, $(\ell_{max}+1)$ レベルでの select の行なわれる回数は, A では N 回であるが, B では, $\lfloor (N^{\gamma}/\alpha)^{\frac{1}{\gamma+1}} \rfloor$ 回ですむ.

Note ; N の増加につれて, 最も頻繁に実行するレベルでの select の行なわれる回数について, 方法Bは、方法Aよりますます有利になることがわかる.

(証明略)

[小性質4]

[α,β,0] uniform(すなわち $P_N(\ell)=\alpha$)の時には、AとB各々全体の手間は、ほぼ、$\alpha^N N!\cdot e^{1/\alpha}(R_A+S_A\cdot N+T/\alpha)$と$\alpha^N N!\cdot e^{1/\alpha}(R_B+S_B/\alpha+T/\alpha)$となる

証明.

$$\sum_{\ell=-1}^{N-2} success(\ell) = 1+\sum_{\ell=0}^{N-2} success(\ell)$$
$$= 1+\sum_{\ell=0}^{N-2}\alpha^{\ell+1}N!/(N-\ell-1)!$$
$$= 1+\alpha^N N!\left(\frac{1/\alpha}{1!}+\cdots+\frac{1/\alpha^{N-1}}{(N-1)!}\right)$$
$$\simeq 1+\alpha^N N!\,(e^{1/\alpha}-1)\simeq\alpha^N N!\,e^{1/\alpha}$$

$$\sum_{\ell=-1}^{N-2} success(\ell)\cdot(N-\ell-1) = N+\sum_{\ell=0}^{N-2} success(\ell)\cdot(N-\ell-1)$$
$$= N+\sum_{\ell=0}^{N-2}\alpha^{\ell+1}N!\cdot(N-\ell-1)/(N-\ell-1)!$$
$$= N+\alpha^{N-1}N!\left(1+\frac{1/\alpha}{1}+\cdots+\frac{1/\alpha^{N-2}}{(N-2)!}\right)$$
$$\simeq N+\alpha^{N-1}N!\,e^{1/\alpha}\simeq\alpha^{N-1}N!\cdot e^{1/\alpha}$$

および 性質2を用いる. □

Note: R_A, R_B, S_A, S_Bはすべて$\Theta(1)$なので、Tすなわちtest1回あたりの手間が$\Theta(1)$である問題については、方法Bの方が確かに有利であることがわかる. なお、R_A, R_Bに含まれる「状況の更新・復元」については、普通$\Theta(1)$であるプログラムが多い.

Note; 例えば，$P_N(l)=1/(l+1)$ の場合，$success(l)=\binom{N}{l+1}$ となるが，この時，l_{max}/N が一定となる．結局，AとB各々の全体の手間は，ほぼ
$2^N\cdot(R_A+S_A\cdot N+T\cdot N/2)$ と $2^N\cdot(R_B+S_B\cdot N/2+T\cdot N/2)$ となるので、両者の手間については，定数倍の違いしかない．

（証明略）

§4. 応用例：Nクイーン問題

簡単な応用例として，有名なNクイーン問題を取上げよう．

問題：NxNの大きさの（拡張された）チェス盤に，N個のクィーンをどの2つも互いに利かないように置く．置き方（解）の総数を数え上げる．
（ただし解の対称性は考慮しない）．

文献[9]で，方法AとBによる解法とその能率の比較が易しく解説されているので，本稿では，その要点をまとめるだけにとどめる．ここでは，オ3の解法を詳しく紹介することにしょう．

4.1 3つのプログラム

・方法Aによるプログラムは，数多くの教科書，論文に載せられている．（例えば，文献[1]，[2]．ここでは，あと

での対照のため文献[1]のプログラムを用いる）.

・方法Bによるプログラムを次に載せる.

ここで，配列upとdownについては，各々.

Boolean array up[-(N-1):(N-1)], down[0:2*(N-1)]

という具合に宣言されているものとする.

```
procedure Search; integer j;
 begin j←-1;
   for  j←F[j] while j≥0 do
     if up[ℓ-j] and down[ℓ+j] then
       begin  X[ℓ]←j;
              F[B[j]]←F[j] ; B[F[j]]←B[j];
              up[ℓ-j]←false ; down[ℓ+j]←false
              ℓ←ℓ+1 ;
              if ℓ=N then 解[X[0],…,X[N-1]]
                     else Search;
              ℓ←ℓ-1 ;
              down[ℓ+j]←true; up[ℓ-j]←true;
              B[F[j]]←j ; F[B[j]]←j
       end
 end;
```

```
begin
  for k←0 until N-1 do begin F[k-1]←k; B[k]←k-1 end;
  F[N-1]←-1 ; B[-1] ← N-1 ;
  for k←0 until 2*(N-1) do
    begin up[k-N+1] ← true ; down[k]←true end;
  l←0 ;
  Search
end
```

・方法Bを応用して，配列upとdownについての検査を省き，selectの手間をさらに減らすアルゴリズムを次に示す．これをC版とよぶ（このアイデアおよびその具体的なアルゴリズムはすべてKnuthによる[7]）．

N×Nチェス盤に対して，行ごと（変数lに対応）に両方向リストを用意する（合計N個）．例えば，N＝4の場合，initializeによって図3のようになろう．

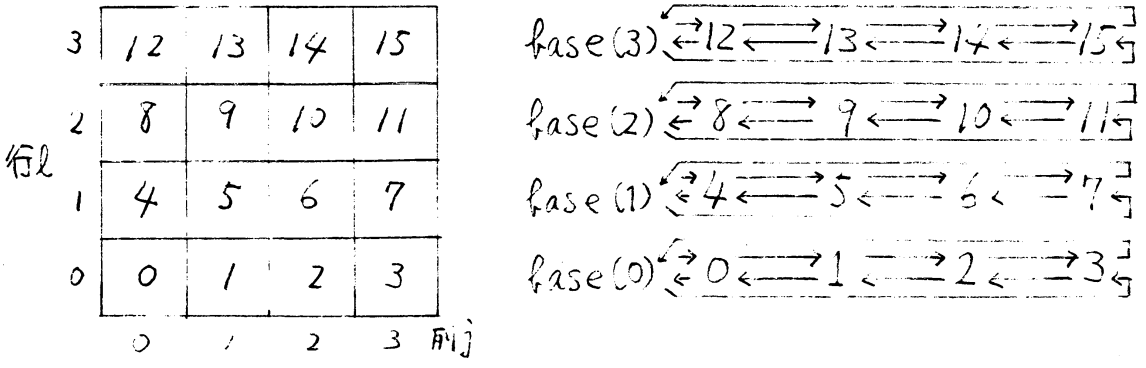

図3. 4×4チェス盤の表現と4つの両方向リスト

次に各マスに対して，すでに置かれているクイーンによって利いているか否かを表わすBoolean型のN×Nの配列を用意する．removeではずしたマスの番号は，stackに入れておき，recoverで取出す．

Note；lレベルのSearchが呼ばれた時，base(l)から参照できるマスは，(l-1)レベルまでに置かれたどのクィーンによっても利いていないので，testの必要がなくなる．手続Searchは，次のようになる．

```
procedure Search; element j;
  for each j in base(l)が指すリストで表わされる集合 do
    begin
      jのマスにクィーンを置くことによって，初めて利くマスをリストbase(l)~base(N-1)からはずす; --- remove
      l←l+1;
      if l=N then 解[j0,…jN-1]
             else Search;
      l←l-1;
      lレベルでremoveされたマスをすべて各々対応するリストに戻す                --- recover
    end
```

4.2 手間の比較

(1) 実行時間 (HITAC 8800 TSS の PASCAL を用いて計った、
HITAC 8350 の ALGOL 60 による結果もほぼ同様の傾向である.)

N	8	9	10	11	12
A	0.13	0.57	2.63	13.4	74.1
B	0.09	0.38	1.65	8.0	42.1
C	0.24	1.03	4.46	21.6	——

(Second)

方法Bが最も速く，従来の方法Aと比べるとN=8で69%，N=12で57%になっている.

(2) A版とB版における Select の回数とクイーンの置ける回数 (Success)

N=8

ℓ	0	1	2	3	4	5	6	7	合計
Aselect	8	64	336	1120	2752	4544	4400	2496	15720
Bselect	8	56	252	700	1376	1704	1100	312	5508
success	8	42	140	344	568	550	312	92	2056

N=12

ℓ	0	1	2	3	4	5	6	7
Aselect	12	144	1320	9072	48960	202224	634272	1441248
Bselect	12	132	1100	6804	32640	117964	317136	600520
success	12	110	756	4080	16852	52856	120104	195270

	8	9	10	11	合計
Aselect	2343240	2572640	1731568	817168	10103808
Bselect	781080	668160	321928	68264	2715740
success	222720	169764	68264	14200	856188

(3) クィーンの置ける回数が最大となるレベル(ℓmax)

N	8	9	10	11	12
ℓmax	4	5	6	7	8

容易にわかることであるが，一般に，

{ℓレベルでクィーンの置けた回数} × N =

{(ℓ+1)レベルでA版のselectの回数}，

{ℓレベルでクィーンの置けた回数} × (N-ℓ-1)=

{(ℓ+1)レベルでB版のselectの回数}が成立する．

この実験より N-ℓmax = 4 であることが読み取れる．以上のことより，(ℓmax+1)レベルについては，A版のselectの回数が最大になるが，B版の回数は，A版の(N-ℓmax-1)/N倍，すなわち 3/N 倍にすぎない．合計を見ると，N=8の時，B版のselectの回数は，A版の35%，また N=12の時，この値は30%になる．

この問題については，定義5において例えば[0.78, 0.5, 2] uniformが適用できよう．この型が与えているsuccess(ℓ)の値

について，実験結果と対照したのが図4である．方法Bにおいて Select される要素数の全体 $C_B(N)$ は，$N=8$ の時4726，$N=12$ の時 3.099×10^6 と推定できるが，本当の値は，$N=8$ の時5508，$N=12$ の時 2.916×10^6 となっている．同様に，$N=13$ では，1.882×10^7 $N=14$ では，1.219×10^8 と推定できる．（これにより実際の計算時間も推定できよう）

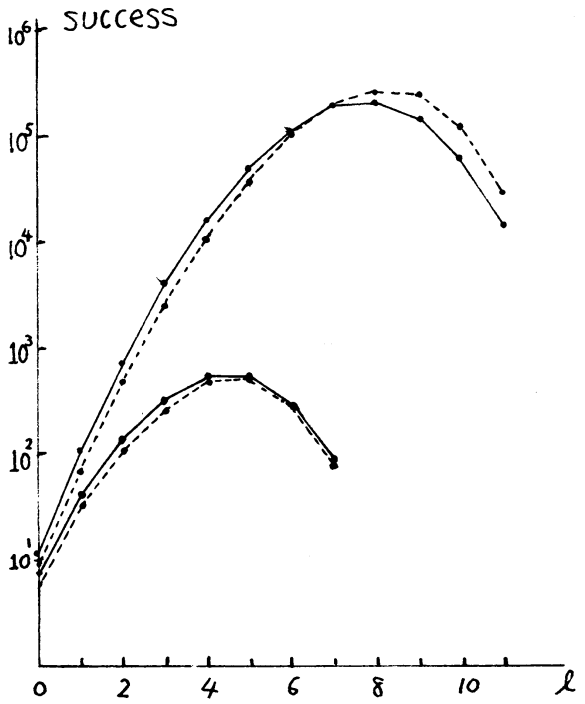

図4. 実線が実験値，
点線が[0.78, 0.5, 2]uniform

さらに詳しく推定するには，文献[6]，[8]で示されているような予備計算による方法を併用するのが有効と思われる．

(4) C版の評価

オℓレベルでクィーンを置いた時，調べる必要のあるマスは，クィーンの斜左上，真上，斜右上の3方向でよいので，その個数は高々 $c(N-\ell-1)$ である（ここで c は $2\leq c\leq 3$ なる定数）したがって，remove と recover の手間は，$\Theta(N-\ell-1)$ となる．C版では，B版と比べて，remove と recover に多くの手間をかけているが，既述の通り，N に近いレベルで最も頻繁に

実行されるので，全体の手間はあまり増加していないと思われる．B版とC版における各基本操作の手間は，次のようになる．

	remove と recover	select	test
B版	$\Theta(1)$	$\Theta(N-\ell-1)$	$\Theta(N-\ell-1)$
C版	$\Theta(N-\ell-1)$	$\Theta(N-\ell-1)$以下	0

結局，大雑把な評価では，C版もB版同様に，全体の手間が，$\sum_{\ell=1}^{N-2} success(\ell) \times \Theta(N-\ell-1)$ となる．ここで，$success(\ell)$ は，ℓ レベルでクィーンの置ける回数である．実際に，実行時間を計った前出の表をみると，C版は予想程の効果は見られない．それは，次の理由によるものであろう．操作selectでは，クィーンが置けるものしか取出さないという意味で，全く無駄がないが，操作removeでは，既に置かれたクィーンによって利いているマスについても調べるという無駄をしている．

Note: C版は，B版の方法の応用であると考えられるが，他の問題において，testの手間とremove-recoverの手間の関係によっては，C版の方向での実現法が有効になることもあろう．

参考文献

[1] E. W. Dijkstra, "Notes on Structured Programming," in Structured Programming, Academic Press (1972), 1-82.

[2] R. W. Floyd, "Nondeterministic Algorithms," JACM, 14, 4 (1967), 636-644.

[3] S. W. Golomb and L. D. Baumert, "Backtrack Programming," JACM, 12, 4 (1965), 516-524.

[4] H. Hitotumatu and K. Noshita, "A Technique for Implementing Backtrack Algorithms and Its Application," Information Processing Letters (1979, to appear).

[5] S. Kawai, K. Noshita and I. Takeuchi, "On Backtrack Programming and Some Results on Combinatorial Puzzles," Second USA-JAPAN Computer Conference Proceedings (1975), 300-305.

[6] D. E. Knuth, "Estimating the Efficiency of Backtrack Programs," Math. Comp., 29 (1975), 121-136.

[7] D. E. Knuth, personal communication (1979).

[8] P. W. Purdom, "Tree Size by Partial Backtracking," SIAM J. Comput., 7, 4 (1978), 481-491.

[9] TSINKY, "エイトクイーン問題再考," bit, 11, 4 (1979), 30-35.

[10] M. Wells, Elements of Combinatorial Computing, Pergamon (1971).