アルゴリズム 22－8
（1991. 7. 22）

推移的閉包計算のための再分割を伴わない列ブロック分割法

大柳 俊夫　大内 東
北海道大学工学部

二次記憶装置とのデータI/Oを伴いながら推移的閉包計算を行うことは，大規模な関係データベースにおける再帰質問処理などで必要となる重要なことである．このような二次記憶環境下で推移的閉包計算を効率よく行う方法として，列ブロック分割法および行ブロック分割法が,Agrawal らにより提案されている．彼等の方法は,I/Oをできる限り行わないための動的なブロック分割の決定およびブロックの再分割方法に特徴がある．

本論文では,Agrawal らの列ブロック分割法における列分割方法を変更し，ブロックの再分割を完全に避けることが可能な，再分割を伴わない列ブロック分割法を提案する．

Non-repartitioning Blocked Column Transitive Closure Algorithm

Toshio OHYANAGI and Azuma OHUCHI
Faculty of Engineering,
Hokkaido University
Sapporo 060, JAPAN

A new transitive closure algorithm, called *non-repartitioning blocked column algorithm*,is proposed. It was obtained by modifying the column partitioning method of *blocked column algorithm* proposed by Agrawal et al. The new algorithm excludes a possibility of *dynamic column repartitioning* which is an essential feature of blocked column algorithm. A detailed comparison of the total number of I/O and the total amount of I/O of both algorithms is made. The result reveals that the proposed algorithm has better I/O features than that of blocked column algorithm.

1　はじめに

二次記憶装置とのデータ I/O を伴いながら推移的閉包計算を行うことは, 大規模な関係データベースにおける再帰質問処理などで必要となる重要なことである. このような二次記憶環境下で推移的閉包計算を効率よく行う方法として, データを主記憶に収るようにブロック分割し, そのデータを Warshall 法[1],Warren 法[2] の処理順序を変更して処理する, 列ブロック分割法および行ブロック分割法を Agrawal らは提案した[3),4)]. 彼等の方法は, I/O をできる限り行わないための動的なブロック分割の決定方法およびブロックの再分割方法に特徴がある.

本論文では,Agrawal らの列ブロック分割法の列分割方法を変更し, ブロックの再分割を完全に避けることが可能な新しい方法を提案する. この方法は,Agrawal らの方法に比べて (1) 列分割を大きくすることが可能で, 列分割数を小さくできる, (2) 無駄な I/O がまったく生じない, という特徴を持っている.

2　推移的閉包計算のための列ブロック分割法

2．1　グラフの推移的閉包

あるグラフの隣接行列 A が与えられたとき, その推移的閉包 A* とは,

$$a^{*}_{ij} = \begin{cases} 1: & \text{頂点 i から j へ道が存在するとき} \\ 0: & \text{頂点 i から j へ道が存在しないとき,} \end{cases}$$

×	10	19	28	37	46	55	64	73	82
1	×	20	29	38	47	56	65	74	83
2	11	×	30	39	48	57	66	75	84
3	12	21	×	40	49	58	67	76	85
4	13	22	31	×	50	59	68	77	86
5	14	23	32	41	×	60	69	78	87
6	15	24	33	42	51	×	70	79	88
7	16	25	34	43	52	61	×	80	89
8	17	26	35	44	53	62	71	×	90
9	18	27	36	45	54	63	72	81	×

図1　Warshall 法の処理順序
Fig. 1 Processing Order of Warshall's Algorithm.

×	46	47	48	49	50	51	52	53	54
1	×	55	56	57	58	59	60	61	62
2	3	×	63	64	65	66	67	68	69
4	5	6	×	70	71	72	73	74	75
7	8	9	10	×	76	77	78	79	80
11	12	13	14	15	×	81	82	83	84
16	17	18	19	20	21	×	85	86	87
22	23	24	25	26	27	28	×	88	89
29	30	31	32	33	34	35	36	×	90
37	38	39	40	41	42	43	44	45	×

図2　Warren 法の処理順序
Fig. 2 Processing Order of Warren's Algorithm.

を成分とする行列A* である. このA* を求めるには, A* = Aから出発して, A* のすべての要素 a^*_{ij} に対して, 次の処理を以下の先行条件のもとで行う.

[a^*_{ij} の処理]

a^*_{ij} の処理とは, 現在のその要素の値が1か0かを調べ, 1の場合に, 頂点jの後続者リストを頂点iの後続者リストへ重複がないように加えることである[1].

[先行条件]

a^*_{ij} の処理は, それ以前に,k < jであるすべてのkに対して, a^*_{ik} および a^*_{jk} の処理を終えてから行わなければならない[3),4)].

例として,10 × 10行列の場合の Warshall 法,Warren 法の処理順序をそれぞれ図1, 図2に示す. なお, 対角要素に対する処理は行なう必要がないので, 図中では×印とした. これらの方法が, 先行条件を満足していることは容易に確認できる.

2.2 列ブロック分割法

Agrawal らの列ブロック分割法では, 後続者リストを分割の始点となる頂点から順に主記憶へ読み込み, 読み込みが可能であった最後の頂点に対応する列を列分割の終点として列分割を決定している. この方法による列分割および処理順序の例を図3に示す. この例では, 行列は1列～3列,4列～7列,7列～10列の3つの列ブロックに分割されている. 行列中の数字が処理順序を示す. ここで, 列分割中ですべての対角要素（図中の×印）を含む正方部分行列を対角ブロック, それ以外の部分を非対角ブロックと呼ぶ.

×	25	26	46	47	58	49	67	68	69
1	×	27	50	51	52	53	70	71	72
2	3	×	54	55	56	57	73	74	75
4	5	6	×	58	59	60	76	77	78
7	8	9	28	×	61	62	79	80	81
10	11	12	29	30	×	63	82	83	84
13	14	15	31	32	33	×	85	86	87
16	17	18	34	35	36	37	×	88	89
19	20	21	38	39	40	41	64	×	90
22	23	24	42	43	44	45	65	66	×

図3 列ブロック分割の例
Fig.3 An Example of Blocked Column Partitioning.

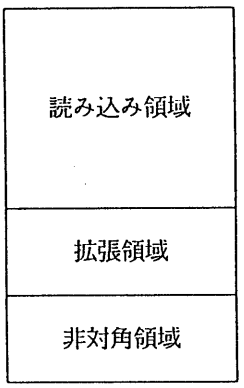

図4 Agrawal らの主記憶レイアウト
Fig.4 Memory Layout by Agrawal et al.

[1]ある頂点の後続者リストとは, その頂点から到達可能な頂点集合のことである. 本論文では, 隣接行列Aとその推移的閉包A* は, 後続者リストの集合として保持されているものとする.

また, 列ブロック分割法で採用している主記憶のレイアウトを図4に示す. 図中の読み込み領域は, 列分割決定の際に二次記憶装置から後続者リストを読み込むための領域, 拡張領域は, 処理した結果を保存する領域, そして非対角領域は, 主に非対角ブロックを処理する際に使用する領域である[2].

```
 1:  J_b := 1;
 2:  repeat
       (* 列分割の決定 *)
 3:    頂点J_bの後続者リストを読み込む;
 4:    i := J_b;
 5:    while (読み込み可能) and (i < n) do begin
 6:        i := i + 1;
 7:        頂点 i の後続者リストを読み込む
 8:    end;
 9:    J_e := i;
       (* 対角ブロックの下三角部分の処理 *)
10:    for i := J_b to J_e do
11:        for j := J_b to i-1 do begin
12:            a*_ij の処理を行う;
13:            if (拡張領域に余裕がない) then
                   列を再分割する
14:        end;
       (* 非対角ブロックの処理 *)
15:    for i := J_e+1 to n do begin
16:        頂点 i の後続者リストを読み込む;
17:        for j := J_b to J_e do
18:            a*_ij の処理を行う;
19:        頂点 i の後続者リストを書き戻す
20:    end;
21:    for i := 1 to J_b-1 do begin
22:        頂点 i の後続者リストを読み込む;
23:        for j := J_b to J_e do
24:            a*_ij の処理を行う;
25:        頂点 i の後続者リストを書き戻す
26:    end;
       (* 対角ブロックの上三角部分の処理 *)
27:    for i := J_b to J_e-1 do begin
28:        for j := i+1 to J_e do
29:            a*_ij の処理を行う;
30:        頂点 i の後続者リストを書き戻す;
31:    end;
32:    頂点J_eの後続者リストを書き戻す;
       (* 次の列分割への準備 *)
33:    J_b := J_e + 1
34:  until (J_e = n);
```

図5　列ブロック分割法
Fig.5 Blocked Column Algorithm.

[2]Agrawal らは, 読み込み領域を主記憶の 80 % 程度の記憶容量, 非対角領域をすべての頂点を後続者リストとして持つ場合に必要な最低限の記憶容量, そして拡張領域を残りの部分の記憶容量としている.

Agrawal らが提案した列ブロック分割法を図5に示す. 図に示すようにこの方法では, 読み込み可能な後続者リストをまとめて読み込み, これにより列分割を決定する (3-8 行). 図中の J_b, J_e がそれぞれ列分割の始めと終りを示す. そして, 列分割決定後の処理は,

(1) 対角ブロックの下三角部分 (10-14 行),

(2) 非対角ブロック (15-26 行),

(3) 対角ブロックの上三角部分 (27-31 行),

の順で進められる. ここで問題となることは,(1) の処理中に拡張領域に余裕がなくなる場合が起こることである (13 行). これに対して Agrawal らは, 列を再分割して読み込み領域の一部を解放し, 解放された領域を拡張領域として用いる方法を提案した. この再分割が, 彼等の列ブロック分割法の特徴的なところである. しかし, この再分割により放棄された後続者リストは, 無駄な読み込みであったことになり,I/O コストを考えるとできる限り避けたいことである.

3 再分割を伴わない列ブロック分割法

Agrawal らの列ブロック分割法の列分割方法を変更して, 無駄な読み込みを完全に避けることを可能にした新しい列ブロック分割法を提案する（図6）. また, この方法で用いた主記憶レイアウトを図7に示す. この方法と Agrawal らの方法との本質的な違いは, 後続者リストの読み込みごとにその後続者リストに対する処理を行うことである[3]. つまり提案する方法では, 主記憶中の読み込み・拡張領域に余裕がある間, 後続者リストの読み込みとその処理を交互に繰返し行う（図6の 7-9 行）. そして列分割は,

(1) 後続者リスト読み込み中に読み込み・拡張領域に余裕がなくなる,

(2) 後続者リスト処理中に読み込み・拡張領域に余裕がなくなる,

(3) 未処理の後続者リストが存在しない,

のいずれか1つが生じて決められる. いずれの場合も, すでに主記憶中に存在して処理が完了した後続者リストで列分割が決定される. このように列分割を決めると, 列分割決定時点で対角ブロックの下三角部分の処理が完了していることになる. したがって, (1) または (2) が生じた場合には, 既に非対角ブロックの処理に進んでいると考え, 非対角領域を用いて読み込みおよび処理を続ければよい. このようにすると, 後続者リストの読み込みは無駄にならないことになる.

Agrawal らの列ブロック分割法では, 対角ブロックの下三角部分の処理でのみ再分割が必要な場合が生じたことから, 今回提案する方法は, 再分割を完全に避けることが可能な列ブロック分割法であることになる.

---

[3]頂点 i に対応する行の, 列分割中の $a^*_{iJ_b}$ から $a^*_{iJ_e}$ までのすべての要素の処理をまとめて, 頂点 i の後続者リストの処理と呼ぶ.

```
 1:  J_b := 1;
 2:  repeat
       (* 対角ブロックの読み込みと下三角部分の処理 *)
 3:      頂点J_bの後続者リストを読み込む;
 4:      i := J_b;
 5:      while (読み込み可能) and (i < n) do begin
 6:          i := i + 1;
 7:          頂点 i の後続者リストを読み込む;
 8:          for j := J_b to i-1 do
 9:              a*_ij の処理を行う;
10:          if (読み込み・拡張領域に余裕がない) then
                 i := i - 1
11:      end;
12:      J_e := i;
       (* 非対角ブロックの処理 *)
13:      if (J_e ≠ n) then
             頂点J_e+1 の後続者リストを書き戻す;
14:      for i := J_e+2 to n do begin
15:          頂点 i の後続者リストを読み込む;
16:          for j := J_b to J_e do
17:              a*_ij の処理を行う;
18:          頂点 i の後続者リストを書き戻す
19:      end;
20:      for i := 1 to J_b-1 do begin
21:          頂点 i の後続者リストを読み込む;
22:          for j := J_b to J_e do
23:              a*_ij の処理を行う;
24:          頂点 i の後続者リストを書き戻す
25:      end;
       (* 対角ブロックの上三角部分の処理 *)
26:      for i := J_b to J_e-1 do begin
27:          for j := i+1 to J_e do
28:              a*_ij の処理を行う;
29:          頂点 i の後続者リストを書き戻す
30:      end;
31:      頂点J_eの後続者リストを書き戻す;
       (* 次の列分割への準備 *)
32:      J_b := J_e + 1
33:  until (J_e = n);
```

図6　再分割を伴わない列ブロック分割法
Fig.6 Non-Repartitioning Blocked Column Algorithm.

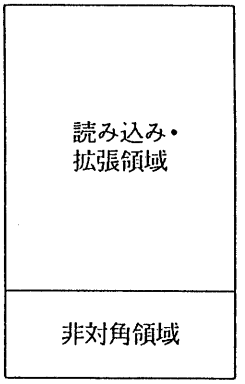

図7 提案する主記憶レイアウト
Fig.7 Proposed Memory Layout.

4　Agrawal らの方法との比較検討

二次記憶環境下での推移的閉包計算の効率は, 主記憶と二次記憶装置の間の I/O の回数と量に影響される. そして, これらに影響を与える要因として, 2 つの方法での列分割数の違いと Agrawal らの方法における再分割の発生が考えられる.

4．1　列分割数の比較

再分割を伴わない列ブロック分割法では, 後続者リストの読み込みとその処理を連続して行うことから, 新たに加える後続者のデータを読み込んだ後続者リストの後に続けて保存することができる. このため, 更新された後続者リストは主記憶中で連続して保持することが可能となる. これに対して Agrawal らの方法では, ある後続者リストとその処理結果を連続して保持することは困難である. したがって, 主記憶中で後続者リストを表現する時に必要な記憶容量は, 今回提案する方法が Agrawal らの方法よりも一般的に少ないと考えられる. また Agrawal らの方法では, 列ブロック分割の処理終了後, 拡張領域に余裕があり次の後続者リストの読み込みと処理を行うことが可能であってもそのような操作は行わない. 以上のことから列分割の大きさは, 今回提案する方法が Agrawal らの方法よりも大きくなると考えられる. したがって列分割数は, 今回提案する方法が Agrawal らの方法よりも多くなることはないと考えられる.

4．2　Agrawal らの方法における再分割の発生

Agrawal らの方法における再分割の発生は, 読み込み領域を小さくし, 拡張領域を大きくすることにより, ある程度避けることが可能である. しかし,

(1) 列分割が小さくなり, 列分割数が多くなる,

(2) 主記憶中に大きな未使用領域ができる,

といった問題が生じる. 結局, 再分割の発生と列分割数にはトレードオフの関係が存在し, 再分割を起こしたくなければ列分割数は大きくなり, 再分割を起こしてもよいのならば列分数を小さくできることになる.

4．3　I/O に関する比較検討

4．1 および 4．2 の考察と, 今回提案した方法が後続者リストを 1 つずつ読み込む方法であることから, I/O 回数に関しては, 次のように考えることができる.

隣接行列の大きさを n, 列ブロック分割法による列分割数を r, 再分割を伴わない列ブロック分割法による分割数を s とする. また, 列ブロック分割法による r 個の列分割それぞれの大きさを $r_i (i=1,\ldots, r)$ とする. すると, 列ブロック分割法による 1 つの列分割の処理に要する I/O 回数は, 入力が $1+(n-r_i)$ 回, 出力が n 回であるから[4], 総 I/O 回数 T_r は,

$$T_r = \sum_{i=1}^{r}(1 + n - r_i) = r(1 + n) - n + n r = n(2r - 1) + r,$$

[4]対角ブロックの r_i 個の後続者リストの入力はまとめて行なわれるので 1 回の入力と考えた.

である. 一方, 再分割を伴わない列ブロック分割法による1つの列分割の処理に要するI/O回数は, 入出力それぞれn回であるから, 総I/O回数T_sは,

$T_s = 2ns$,

である. よって, T_rとT_sの間には,

$T_r < T_s (r \leq s)$, $T_r > T_s (r > s)$,

という関係が成り立つことになる. ここでrとsの大小は, 4.1で述べたように一般にr≧sであるが, 主記憶容量に対して扱うデータが大規模であれば, r>sとなる可能性が大きくなると考えられる. したがってI/O回数に関しては, 今回提案する方法がAgrawalらの方法よりも少なくなると考えられる. なおrとsの大小関係を厳密に比較するには,Agrawalらの方法における読み込み領域の大きさ, 主記憶中での後続者リストの表現方法, 閉包計算を行うデータなどに関する詳細な検討が必要と考えられる.

また今回提案した方法は, 再分割を避けることが可能で無駄なI/Oが生じないことと, 前述の通りI/O回数が一般にAgrawalらの方法よりも少なくなることから, I/O量に関してもAgrawalらの方法よりも少なくなると考えられる.

5 おわりに

本論文では,Agrawalらの列ブロック分割法の列分割方法を変更して, ブロックの再分割を完全に避けることが可能な再分割を伴わない列ブロック分割法を提案した. 今回提案した方法は,Agrawalらの方法に比べて,

(1) 列分割を大きくすることが可能で, 列分割数を減らすことができる,

(2) 無駄なI/Oを完全に避けることが可能である,

という特徴がある. そして, I/O回数と量に関する比較検討を行い,I/O回数とI/O量ともに, 今回提案した方法がAgrawalらの方法よりも少なくなることを明らかにした.

参考文献

1)Warshall,S.:A Theorem on Boolean Matricies, J.ACM No.9,Vol.1,pp11-12(1962)

2)Warren,H.S.:A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations, Comm.ACM Vol.18,No.4 pp218-220(1975)

3)Agrawal,R. and Jagadish,H.V.:Direct Algorithms for Computing the Transitive Closure of Database Relations, in Proc. 13th International Conference of Very Large Data Bases,pp.255-266 (1987)

4)Agrawal,R., Dar,S. and Jagadish,H.V.:Direct Transitive Closure Algorithms:Design and Performance Evaluation,ACM Trans. Database Syst., Vol.15,No.3,pp.427-458(1990)