データベースシステム 106－17
（１９９６．１．２５）

共有可能な索引の二次記憶格納方式および推移的閉包計算への応用

井上 裕策　　岩井原 瑞穂
九州大学 大学院 総合理工学研究科 情報システム学専攻
〒816 春日市春日公園 6-1
Email:{yinoue, iwaihara}@is.kyushu-u.ac.jp

本稿では，関係に対する索引として，索引のうち同じ形の部分を共有により１つにまとめて表すデータ構造を提案し，この共有可能な索引を二次記憶上に格納する方法，およびその上での効率的な関係演算の処理方法について考察する．推移的閉包計算における中間関係表現サイズを，既存の二次記憶上でのデータ構造の格納方式と比較した結果，提案する格納方式は，中間関係が疎から密に大幅に変化する場合にも，表現サイズが著しく変化しないとの結果を得た．

Disk Storage Design of Shared Indices and Its Application to Transitive Closure Computation

Yusaku INOUE　　Mizuho IWAIHARA
Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyushu University
6-1 Kasuga-koen, Kasuga-shi, Fukuoka 816 Japan
Email:{yinoue, iwaihara}@is.kyushu-u.ac.jp

We have proposed an indexing method for a relation that includes sharing of the isomorphic particle of indices. In this paper, we propose a design of storing shared indices on disks, and a method of manipulating relational operations. We further compared our method with an existing storage design through transitive closure computation, and the result showed that our method is suitable in situations such that whether temporary relations are dense or sparse is hard to predict.

1 はじめに

本稿は，関係に対する索引として論理関数のコンパクトな表現方法である**二分決定グラフ** (Binary Decision Diagrams, 以下 BDD)[1] を用いることを提案する．関係を 2 値のビットマップとして表現し，それを結合および集合演算のための索引として用いる手法は研究されているが，本手法ではビットマップをさらに BDD を用いてコンパクトに表現することを特徴としている．BDD の索引としての特徴は，

- 索引のうち同じ形の部分が，共有により 1 つにまとめられることにより，記憶効率が向上する
- 1 つの関係に対して索引の形が一意に決まる (表現の一意性) ことから，等価性がポインタの比較のみで可能になる
- 表現の一意性を利用して，同じ演算を繰り返し処理するのを防ぐことができるため，処理の効率化が可能となる (演算キャッシュ)

ことにある [2]．このことは，

- 同じ部分関係が繰り返し現れる関係の重複の除去された表現
- 同じ定義域をもつ複数の関係を，重複を除去して保持する
- 同じ関係演算が繰り返し発行される場合の演算キャッシュによる効率化

などに有効と考えられる．以後，関係を表現する BDD を**共有可能な索引** (Shared Index) と呼ぶことにする．

共有可能な索引については従来，主記憶上に格納される場合のみを仮定してきた．しかし，主記憶だけでは容量におのずから限界があるため，実用的な規模のデータを扱うためには，二次記憶の利用を考慮する必要がある．一般に，データベースの処理では，二次記憶に対するデータ入出力に要する時間の割合が大きいことから，データ入出力コストの軽減が重要になる．

本稿では，

1. 共有可能な索引を二次記憶上に格納する方法
2. 二次記憶上の共有可能な索引に対する，効率的な関係演算の処理方法

について考察する．関係に対するビットマップを小片に分割し，1 を含まない小片を除去し，残りの小片で木構造を構成したものをビットマップ木とすると，共有可能な索引は，ビットマップ木に対し，

- 同じ形の部分木を共有させる
- 1 が含まれない部分木は削除する

の操作を施したものと見ることができる．ビットマップ上での 1 の密度が大きい (小さい) とき，関係が**密** (**疎**) であると呼ぶことにする．共有可能な索引の特徴として，関係が演算により密と疎の間を変化しても，共有により索引のサイズを一定に保つことが挙げられる．本稿では，上記の特徴が活かされる例として，**推移的閉包** (Transitive Closure) 計算を取り上げ，既存の二次記憶上でのデータ構造の格納方式との，推移的閉包計算における比較を行っている．その結果，提案する共有可能な索引の格納方式は，中間関係の組数によって表現サイズが著しく変化せず，関係が疎と密の間で大規模に変動する場合にも対処できることが示された．

本稿の構成は以下の通りである．2章では，準備として本稿で扱う共有可能な索引について述べる．3章では，共有可能な索引の二次記憶格納の方法について述べ，その索引に対する関係演算の処理方法を示す．4章では，本稿で述べる二次記憶格納方式に対する表現サイズの評価を実験によって行う．最後の 5 章では本稿のまとめを行う．

2 共有可能な索引

本稿で取り扱う共有可能な索引は，[2] で示した関係の BDD による表現に基づいている．BDD は，図 2に示すように非巡回有向グラフの形で表される．節点は変数節点と定数節点とからなり，各変数節点は 0 枝および 1 枝と呼ばれる 2 本の有向枝をもつ．

2.1 二分決定グラフによる関係の表現方法

関係の表現方法としては，**対数符号化** (Logarithmic Encoding) と**線形符号化** (Linear Encoding) の 2 通りを提案している．

対数符号化は，関係の特徴関数 [3] に対する BDD として表現する方法である．関係の特徴関数は以下のようにして構成される．

1. 関係の各属性の定義域の各要素に対してビットベクトルを割り当てる．
2. 関係の各組 t_j をビットベクトル $\vec{t_j}$ に変換し，その $\vec{t_j}$ を充足する積項 f_j を構成する．
3. 各積項 f_j の論理和が求める特徴関数である．

例えば，図 1(a) の関係 R に対しては，図 1(b) のようにビットベクトルを与えた場合，対数符号化は図 2(a) のようになる．

R

A	B
SAKAI	G8
SASAKI	G6
MIKAMI	H6
MISAKI	G6

(a)

MIKAMI → 10011　G6 → 010
MISAKI → 01101　G8 → 011
SAKAI → 11001　H6 → 110
SASAKI → 11101

(b)

図 1: 例：(a) 関係 R，(b) 属性値に対するビットベクトル割り当て

符号化する関係に組識別子 (TID) が与えられている場合は，TID をそのままビットベクトルとして用いることが可能である．

一方，線形符号化は，図 2(b) に示すように，1 つの属性値に対して 1 つの変数を用いて表現する方法である．ただし，図 2(b) のような表現方法では，必要となる論理変数の数が非常に大きくなるという問題がある．例えば，長さ n の英大文字列の集合を定義域にもつ属性に対する論理変数は，最大 26^n 個となる．

そこで，各属性値の文字列を何文字かごとに分解し，その各部分を論理変数として表現する．これを属性の**多段化**と呼び，分解された各部分を**段**と呼ぶ．多段化を用いると，図 1(a) の関係は，図 2(c) のように表現される．以後，線形符号化とは多段化を用いた線形符号化を指す．

例えば，上述の長さ n の英大文字列の集合を定義域にもつ属性に対しては，c 文字ごとに多段化した場合，必要な論理変数は，高々$26n/c$ 個である．

線形符号化は，関係が疎な場合に，対数符号化よりも節点数が少なくなるという性質がある [2]．

2.2 共有可能な索引の特徴

共有可能な索引の特徴としては，以下の点が挙げられる．

- TRIE 型の索引のように，根節点から 1 定数節点までのパスが，1 つの組に対応する．
- 索引の同じ部分が共有によって 1 つにまとめられることにより，記憶量が削減されるとともに，1 つの関係に対して索引の形が一意に決まるようになる．この性質を**表現の一意性**と呼ぶ．この性質により，1 つの関係は，1 つの節点 (索引の根節点) によって一意に表現される．

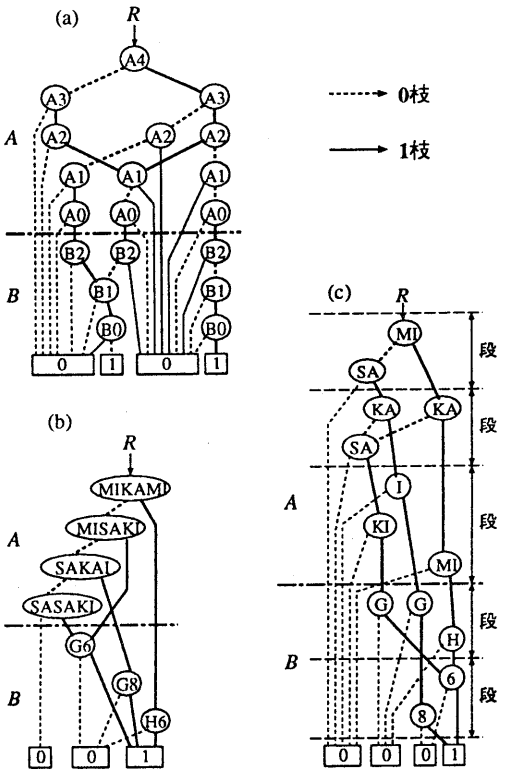

図 2: 図 1(a) の関係に対する共有可能な索引：(a) 対数符号化，(b) 線形符号化，(c) 多段化を用いた線形符号化

- 表現の一意性により，演算キャッシュ(同じ演算の繰り返しを避けるための技法) を用いることができる．演算キャッシュは，最近行われた演算のオペランドおよび結果を表す節点へのポインタを，ハッシュテーブルに登録することにより実現される．この技法を用いることにより，演算処理の高速化が可能になる．

2.3 対数符号化と線形符号化の比較

対数符号化に基づく共有可能な索引の二次記憶格納方式は，[4] の BDD に対する方法を応用して実現することができる．

しかし，[4] の方法では，論理変数の数と同じ高さの索引が作られ，特に疎な関係の場合に不必要に高い索引となるという問題点がある．例えば，定義域の

大きさがNの属性を対数符号化する場合，$\lceil \log_2 N \rceil$個の変数が必要となり，索引の高さも$\lceil \log_2 N \rceil$となる．このため，実際に表現する値の数がNより十分小さいときには，冗長となる．

これに対し，同じ定義域の大きさNの属性を，文字数aのアルファベットの文字列として表現し，1文字ごとの多段化で線形符号化した場合，段数は$\lceil \log_a N \rceil$段となる．これに以下で述べる節点ブロックを用いることにより，高さ$\lceil \log_a N \rceil$の索引を作ることができる．即ち，線形符号化と対数符号化の索引の高さの比は$\log_2 a$となる．例えば，$a = 5$とすると，索引の高さを1/5にすることができる．ただし，aを大きく取ることにより，関係が密になる場合の節点数が多くなるというトレードオフがある．

そこで本稿では以後，線形符号化に基づく共有可能な索引を，単に共有可能な索引と呼び，その二次記憶格納方式について考察する．

3 提案する索引の格納方式および関係演算処理

3.1 節点格納の方針

組の検索は，索引の各段における段値の検索を行うことによってなされる．段値の検索は，検索したい値の変数節点が見つかるまで，0枝をたどり続けることによってなされる．見つからない場合は最終的に0定数節点にたどり着くことになる．このとき，3(a)の枠囲みの部分のように，同一段内の0枝によって指される節点どうしの列を**層内パス**と呼ぶ．

特に層内パスに含まれる節点の数が多い場合には，層内パスをたどる際のページ入出力コストを考慮しなければならない．段値の検索コストを軽減するための方針として，

- 層内パスに含まれる節点をできるだけ同じページに格納する
- 層内パスに含まれる節点を連続する領域に格納する

方法が考えられる．

3.2 節点ブロック

本稿で提案する索引の格納方式では，層内パスを図3(b)のように仮想的な多分岐節点として表すことによって，上記の方針を実現する．この仮想的な多分岐節点を**節点ブロック** (Nodeblock) と呼ぶ．

この節点ブロックを導入することにより，共有可能な索引について以下のような性質が得られる．

- 第$L(>1)$段の節点ブロックの枝は，必ず第$L-1$段の節点ブロックを指す．
- 第1段 (最下位の段) の節点ブロックの枝は，必ず1定数節点を指す．

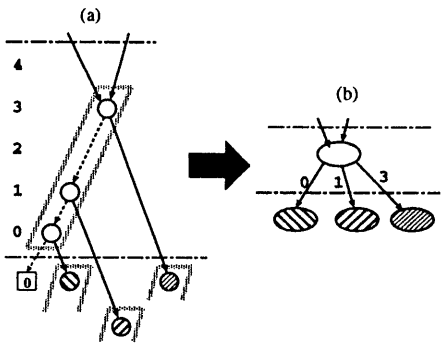

図 3: 節点ブロック

このようにして表現された共有可能な索引は，関係に対するビットマップを木構造で表したビットマップ木に対し，

- 同じ形の部分木を共有させる
- 1が含まれない部分木は削除する

の操作を施して得られたデータ構造と見ることができる．

3.3 関係演算処理の基本方針

3.3.1 幅優先方式

本稿で提案する関係演算の処理は，各段ごとに一括して行う幅優先方式に基づくものである．幅優先方式の処理の目的は，二次記憶に対するランダムアクセスを可能な限り軽減し，順アクセスを主体とすることである．

3.3.2 展開フェーズおよび正規化フェーズ

関係演算の処理は，**展開フェーズ** (Expansion Phase) と**正規化フェーズ** (Reduction Phase) の2フェーズ方式で行われる．

展開フェーズは，結果となる関係を表現するのに十分な節点ブロックが生成されるフェーズで，上位

の段から下位の段に向かって実行される．展開フェーズで生成される節点ブロックを一時節点ブロック (Temporary Nodeblock) と呼ぶ．

正規化フェーズは，一時節点ブロックのうち冗長なものを削除するフェーズで，下位から上位の段に向かって実行される．冗長な一時節点ブロックとは，以下のいずれかに該当する節点ブロックである．

1. 等価な節点ブロックがすでに存在している．等価な節点ブロックとは，
   (a) 枝の本数が等しい
   (b) 同じ段値の枝がそれぞれ同じ節点ブロックを指す

   をともに満たす節点ブロックである．
2. すべての枝がどの節点ブロックをも指さない．

冗長な節点の削除と同様，冗長な一時節点ブロックの削除も，表現の一意性の維持を可能にする．

3.4 データ構造

3.4.1 節点ブロック領域

節点ブロック領域とは，節点ブロック集合を格納する領域のことであり，段ごとに分けて構成される．展開フェーズにおいては，一時節点ブロックは節点ブロック領域の連続するエントリに格納される．

節点ブロックは，主に以下のメンバからなる構造体として表現することができる (図 4)．

参照カウンタ その節点ブロックを参照しているポインタの本数を示す．カウンタの値が 0 になったときは，その節点ブロックは不要になる．

節点ビットベクトル 節点ブロックにおける節点の分布を示す．各ビットは段の値の節点に対応し，節点があれば 1，なければ 0 が記される．

参照ポインタ 各節点が参照している (1 つ下位の段の) 節点ブロックへのポインタを示す．節点ブロックに含まれる節点の数だけ設けられるため，この領域は可変長である．

節点ブロック領域の実現方法としては，図 4に示すように，可変長となる参照ポインタの領域を，本体とは別に格納する．本体には，ある節点ブロックの参照ポインタが，参照ポインタ領域のどの部分に当たるかを示すポインタを付け加える．

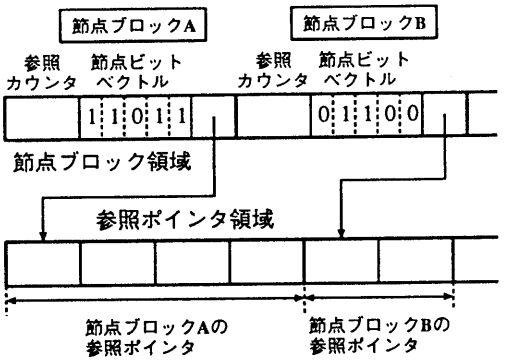

図 4: 節点ブロック領域の実現

3.4.2 ハッシュテーブル

表現の一意性の維持および演算の効率化のためには，節点ブロック領域のほか，**節点ブロックテーブル** (Nodeblock-table)，および**演算結果テーブル** (Operation-result-table) の 2 種類のハッシュテーブルが必要となる．これも段ごとに構成される．

節点ブロックテーブルは，正規化フェーズにおいて，ある一時節点ブロックと等価な節点ブロックがすでに存在している否かを調べるために用いる．

演算結果テーブルは，最近行われた演算の情報を記録しておくためのハッシュテーブルである．この技法は演算キャッシュと呼ばれ，同じ演算の繰り返しを防ぐことによる処理コストの削減を目的としたものである．

3.5 データ入出力

3.5.1 展開フェーズ

和集合演算など，Union Compatible な 2 つのオペランドに対する集合演算の場合には，展開フェーズにおいて，参照されるべき節点ブロック集合の段が，オペランドと結果とで常に等しい．段 L の展開フェーズは，

- 段 $L, L-1$ の節点ブロック領域
- 段 $L+1, L$ の参照ポインタ領域
- 段 L の演算結果テーブル

の情報が主記憶上にあれば実行できる．

一方，例えば自然結合の展開フェーズの場合には，オペランドと結果とで異なる段の節点ブロック集合を参照しなければならない．自然結合 $R_3 = R_1(A, B) \bowtie$

$R_2(B,C)$ の展開フェーズは，以下の 3 つのステップに分かれる (図 5)．ただし属性 A,B,C の定義域はすべて等しいと仮定する．

1. R_1 の属性 A の各節点ブロックオペランドとして，R_3 の属性 A の一時節点ブロックを生成する．
2. R_1 および R_2 の属性 B の部分をオペランドとして，R_3 の属性 B の一時節点ブロックを生成する．
3. R_2 の属性 C の最上位段の節点ブロックを共有する．

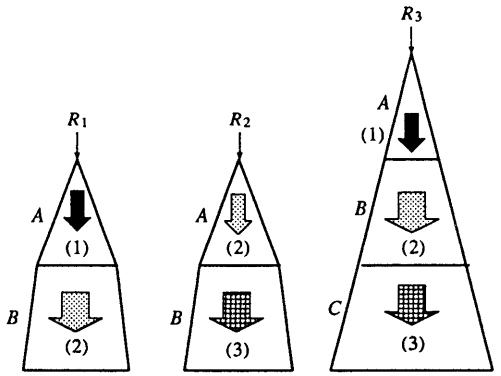

図 5: 自然結合の展開フェーズの実現

3.5.2 正規化フェーズ

正規化フェーズの処理は，どの関係演算の場合でも同じように行われる．段 L の正規化フェーズは，

- 段 $L, L-1$ の節点ブロック領域
- 段 L の参照ポインタ領域
- 段 L の節点ブロックテーブル

の情報が主記憶上にあれば実行できる．

3.5.3 ページ置き換え方針

ある段の節点ブロック領域およびハッシュテーブルを読み込む際，そのデータ構造全体が必要となるわけではないので，処理に必要なページのみ主記憶に読み込む．ページ置き換えの方針としては，将来参照の機会が多いページを主記憶に残す方針が考えられる．参照の機会が多いページとは，具体的には

- 参照カウンタの大きい節点ブロックを含むページ
- 一次節点ブロックを含むページ

などである．

4 実験および考察

本章では，3章で提案した格納方式と，既存の格納方式との比較を，実験を通して行う．実験としては，Semi-Naive 法 [5] による推移的閉包の計算を行う際の，各繰り返しにおける中間関係の表現に使用されるページ数 (以下，中間関係表現サイズと呼ぶ) を，両者で測定した．

4.1 推移的閉包計算におけるデータ構造

推移的閉包 (Transitive Closure) については，現在までに多くのアルゴリズムが提案されている [6][7][8]．それらのアルゴリズムで用いられている二次記憶上のデータ構造としては，

- Successor List[6][8]
- Successor Tree[7]

などがある．各節点 i に対する Successor List とは，i の子孫節点の集合を表したもので，Successor Tree は，Successor List にグラフ構造に関する情報を付加したものである．

本章の実験で用いる既存の格納方式としては，Successor List[8] を比較の対象とする．

Successor List の格納方法は以下に示す通りである．1 つのページを数個の等しいサイズのブロックに分割し，各節点の Successor List を，1 個または連続する数個のブロックに格納する．このとき，1 個のブロックには，異なる節点の Successor List の要素は格納しない．1 個のブロックに格納できる要素の数をブロック要素数 (Blocking Factor) と呼ぶ．

ブロック要素数を大きくとった場合，長い Successor List を少ないブロックで格納できるが，Successor List が短いときは，未使用部分が多くなりオーバヘッドとなるというトレードオフがある．

4.2 ビットマップ

本章の実験では，さらにビットマップ (Bitmap) と呼ばれるデータ構造との比較も行う．ビットマップとは，N 個の属性値からなる 2 項関係を，$N \times N$ ビットのビット列で表すデータ構造である．各ビットに

は，対応する組が関係に含まれるときは 1，含まれないときは 0 が記される．ビットマップによって中間関係を表現する場合，計算途中での表現サイズは N^2 ビットのまま不変である．関係が密である (1 のビットが多い) ときは記憶効率がよいが，疎であるときにはオーバヘッドとなる．

4.3 インスタンスおよびパラメータ

インスタンスの有向グラフとしては，節点数 N，有向辺数 E のランダムグラフを与える．初期有向グラフの N と E の比を変えることにより，推移的閉包グラフの大きさを制御することができる．初期有向グラフの各有向辺に対する節点番号差はすべて $N/10$ 以下という局所性を与えた．

Successor List のブロック要素数は，$B = 8$ に固定した．これは，本章の実験のインスタンスでは，これ以上 B の値を大きくすると未使用部分が増えるからである．共有可能な索引に対しては，1 つの段は 10 進整数の 1 つの桁を表すものとする．ページサイズは 4096 バイトと仮定する．

実験は，Cray Superserver 6400 上で行った．

4.4 実験結果

図 6〜8は，それぞれ，

- $N = 4000, E = 4000$
- $N = 2000, E = 3000$
- $N = 1000, E = 2000$

に対して，中間関係表現サイズを，各繰り返しごとに示したものである．サンプルは同じ E, N に対して 3 個ずつ取り上げた．横軸は各サンプルにおける繰り返し回数を表している．

4.5 考察

$E = 1.5N$ および $E = 2N$ のランダムグラフの場合には，図 6および図 7に示す実験結果より，以下の点を読み取ることができる．

1. ランダムグラフの密度を高くすると，密な推移的閉包が得られる．このとき，計算が進むにつれて中間関係が密になり，それに比例して Successor List による使用ページ数が著しく増大する．
2. 一方，共有可能な索引においては，中間関係が密になるにつれて共有が起きやすくなる．これは，等しい子孫節点集合が多く現れること

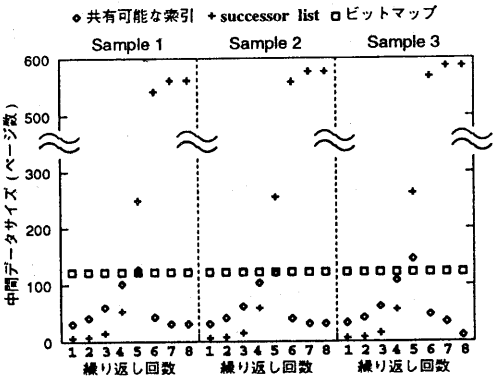

図 6: $N = 2000, E = 3000$ に対する中間関係表現サイズの変化

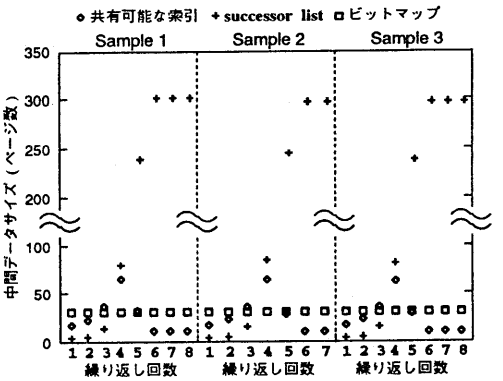

図 7: $N = 1000, E = 2000$ に対する中間関係表現サイズの変化

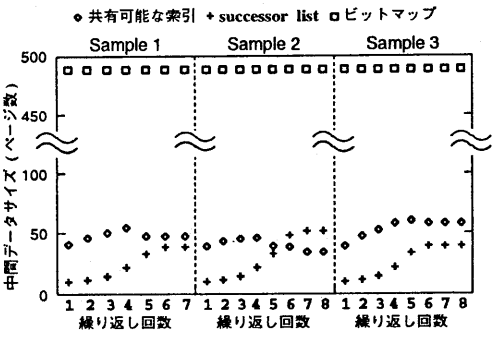

図 8: $N = 4000, E = 4000$ に対する中間関係表現サイズの変化

に起因している．そのため，途中で使用ページ数がSuccessor Listでの使用ページ数を大きく下回る．

3. ビットマップによる表現サイズと比較した場合，途中で共有可能な索引の使用ページ数がビットマップの使用ページ数を上回る．しかし，最終的には共有される部分が多くなるため，再び下回る．

一方，$E = N$とした場合に関しては，図8に示す実験結果より，以下の点を読み取ることができる．

1. $E = N$のランダムグラフに対しては，中間関係および最終的に得られる推移的閉包が疎になる．このとき，中間関係をビットマップで表現した場合，1の部分がきわめて少ないためオーバヘッドとなる．一方，共有可能な索引では，節点ビットベクトルに1が含まれない節点ブロックは削除される．そのため，ビットマップと比較すると，中間関係表現サイズは小さくなっている．
2. 中間関係が疎であるとき，共有可能な索引においては，共有が生じにくくなる．そのような場合，Successor Listに比べて表現効率が悪くなる．
3. しかし，最終的に得られる推移的閉包のサイズが大きくなる場合には，$E = 2N$あるいは$E = 1.5N$とした場合と同様，途中で共有可能な索引とSuccessor Listとが表現効率において逆転する現象も見られる．

ビットマップは，中間関係が疎な場合には無駄が多くなる．一方，Successor Listは中間関係が密になったときの表現サイズの増大の割合がきわめて大きい．それに対し，本稿で提案した共有可能な索引の格納方法では，中間関係の疎密によって表現効率が大きく左右されることはなく，安定性が高いといえる．

5 おわりに

本稿では，共有可能な関係の索引に対する二次記憶格納方式について提案し，その上での関係演算の処理方法を示した．さらに，推移的閉包計算の過程で得られる中間関係の表現サイズを，既存の二次記憶上でのデータ構造の格納方式とで比較した．

4章で示した実験結果より，本稿で提案する共有可能な索引の格納方法は，

- 計算途中での中間関係の組数が大規模に変わる場合
- 計算途中での中間関係の組数の予測が困難な場合

に適しているといえる．

謝辞

熱心に御討論頂いた九州大学大学院総合理工学研究科の安浦寛人教授，村上和彰助教授をはじめとする安浦研究室の諸氏に深く感謝致します．

参考文献

[1] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In *IEEE Trans. Comput. ,Vol.C-35 ,No.8*, pp. 677–691, 1986.

[2] M. Iwaihara and Y. Inoue. "Bottom-Up Evaluation of Logic Programs Using Binary Decision Diagrams". In *Proc. 11th Int. Conf. Data Engineering*, pp. 467–484, Mar. 1995.

[3] 石浦菜岐佐. "BDDとは". 情報処理, Vol.34, No.5, pp. 585–592, 1993.

[4] H. Ochi, K. Yasuoka and S. Yajima. "Breadth-First Manipulation of Very Large Binary Decision Diagrams". In *Proc. ACM/IEEE ICCAD93*, pp. 48–55, 1993.

[5] Jeffery D. Ullman. *"Principles of Database and Knowledge Base Systems"*, Vol. I. Computer Science Press, 1988.

[6] R. Agrawal and H. V. Jagadish. "Hybrid Transitive Closure Algorithms". In *Proc. 16th Int. Conf. Very Large Data Bases*, pp. 326–334, Aug. 1990.

[7] S. Dar and H. V. Jagadish. "A Spanning Tree Transitive Closure Algorithm". In *Proc. 8th Int. Conf. Data Engineering*, pp. 2–11, Feb. 1992.

[8] Y. E. Ioannidis, R. Ramakrishnan and L. Winger. "Transitive Closure Algorithms Based on Graph Traversal". In *ACM Trans. Database Syst.*, pp. 512–576, Sept. 1993.