任意のグラフノードを起点とする正規パス式の評価手法

只石　正輝†　　森嶋　厚行†　　田島　敬史††

† 筑波大学大学院図書館情報メディア研究科　〒 305–8550 茨城県つくば市春日 1–2
†† 京都大学大学院情報学研究科　〒 606–8501 京都市左京区吉田本町
E-mail: †{tada,mori}@slis.tsukuba.ac.jp, ††tajima@i.kyoto-u.ac.jp

あらまし　今日，XML や RDF データ等の普及により，エッジラベル付き有向グラフに対する効率的な問合せ処理が重要な問題となっている．グラフに対する問合せとしては様々な種類の問合せが存在するが，あるノードから指定された正規パス式にマッチするパスで到達可能なノード集合を計算する問合せは重要なクラスの一つである．正規パス式の重要な構成要素として，指定されたエッジラベルを通して取得可能な子供／子孫を求める演算がある．我々はこれまでの研究で，グラフを全域木と非木エッジに分割して格納し，その後子供／子孫演算を処理する手法を提案したが，どのような全域木を構築すれば効率がよいかに関しては議論していなかった．本稿では，この全域木の構築方法と，提案手法の改良について説明する．

キーワード　問合せ処理, グラフデータ

A Method to Evaluate Regular Path Expressions from Any Graph Node

Masateru TADAISHI†, Atsuyuki MORISHIMA†, and Keishi TAJIMA††

† Grad. Sch. of Library, Information and Media Studies, Univ. of Tsukuba.　1–2 Kasuga, Tsukuba, Ibaraki, 305–8550 Japan
†† Grad. Sch. of Informatics, Kyoto University.　Yoshida–Honmachi, Sakyo–ku, Kyoto 606–8501, Japan
E-mail: †{tada,mori}@slis.tsukuba.ac.jp, ††tajima@i.kyoto-u.ac.jp

Abstract　Today, we have a large amount of XML and RDF data, and efficient processing of queries against edge-labeled directed graphs is important. Regular path queries are one of the important queries against graphs. So far, we have developed a scheme for efficient processing of child/descendant queries that are important primitives of regular path queries. Our scheme constructs a spanning tree based on the given graph and uses it to process child/descendant queries. However, it is not clear which spanning tree is appropriate for efficient query processing. This paper discusses first how to construct spanning trees based on the given graphs, and then proposes an improvement of our proposed scheme.

Key words　Query Processing, Graph Data

1. はじめに

本稿では，エッジラベル付き大規模グラフデータに対する正規パス式 (Regular Path Expressions) を効率よく評価するためのディスク上でのデータ格納方式について議論する．XML や各種オントロジデータ [1] など，近年はエッジラベル付きの大規模グラフデータの扱いが重要となっている．また，正規パス式は半構造データ言語等の文脈で議論されてきたが，近年 XML データに対しても Regular XPath 式という形で再び注目されており [2]，その効率よい処理が求められている．

一般に，正規パス式は，子要素を求める演算と，閉包を求める演算に展開できる．本稿では閉包に関しては特に需要の大きいと考えられる「特定ラベルの繰返し」のみに焦点を当て，次の 2 つの演算とその連結 (·) を扱う．

(1) 子供演算：$a \xrightarrow{l} X$: ノード a から，ラベル l を持つエッジを 1 回辿ることで到達可能なノード集合．

(2) 子孫演算：$a \xrightarrow{l*} X$: ノード a から，ラベル l を持つエッジを 0 回以上辿ることで到達可能なノード集合．

本稿で扱う問題の応用範囲は広い．例えば，RDF スキーマにおいて，あるクラス a の派生クラスを求める問合せは，$a \overset{rdfs:subClassOf}{\rightarrow} X$ として記述可能である．また，SNS データベースに対する問合せとして下記のようなものが考えられる．

$prof[name ='' fernandez''] \overset{advises*}{\rightarrow} \cdot \overset{name}{\rightarrow} X$

上記の問合せは，名前が”fernandez”という教授

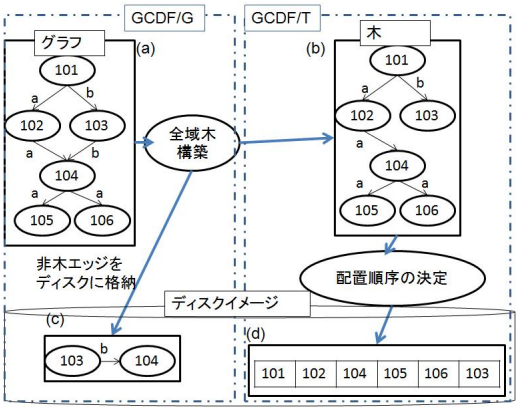

図 1　概　念　図

$(prof[name ='' fernandez''])$ が直接的／間接的に指導した $(advises*)$ 人物の名前 $(name)$ を求める．

論文 [3] の手法の概要: 我々はこれまで，グラフに対する子供／子孫問合せを効率的に処理するためのディスク格納手法を提案してきた [3]．論文 [3] で提案した手法の新規性として，次の 2 つが挙げられる．(1) 問合せ処理コストが対象のデータではなく，**解のサイズ**のオーダとなること．これにより，超大規模データへの対応が可能になる．(2) 木のルートからだけではなく，**任意のノードを起点とした問合せ**を対象としていること．

図 1 に我々が論文 [3] で提案した手法の概要図を示す．論文 [3] で，我々は 2 つの手法を提案した．(1) 木を対象としたディスク格納手法 (本稿では **GCDF/T**(General Child-Depth-First / Tree) と呼ぶ)．(2) グラフを対象としたディスク格納手法 (本稿では **GCDF/G**(General Child-Depth-First / Graph) と呼ぶ)．次に，それぞれの手法について簡単に説明する．

(1)**GCDF/T** は，木を対象としたディスク格納手法である．その目的は，子供／子孫問合せを処理するために必要なディスクアクセス数(注1)を減らす事である．**GCDF/T** では，ディスクアクセス数を減らすために，全ての問合せの解が連続して配置されるような順序でノードをディスクに配置する．全ての問合せの解が連続して配置されるため，問合せ処理に必要なディスクアクセス数は定数回となる (表 1 の 1 行目)．具体的な例として，図 1(b) の木とその木の配置順序，図 1(d) を用いて説明する．図 1(b) における子供／子孫問合せの解はいずれも連続して配置されている．たとえば，104 $\stackrel{a}{\rightarrow} X$ の解 {105,106} や，101 $\stackrel{a*}{\rightarrow} X$ の解，{101,102,104,105,106} はいずれも連続して配置されている．

(2)**GCDF/G** は，グラフを対象としたディスク格納手法である (図 1)．**GCDF/G** では，グラフから全域木を構築し，その木に対して **GCDF/T** を適用する．さらに，全域木の作成において取り除かれたエッジをディスクに格納する．以降，このエッジを**非木エッジ**と呼ぶ．

本稿で扱う問題: **GCDF/G** の場合には，非木エッジでつながれたノードは連続して配置されないため，必ずしも全ての子供／子孫問合せの解が連続して配置されるわけではない．たとえば，図 1(a) のグラフにおいて，101 $\stackrel{b*}{\rightarrow} X$ の解は {101,103,104} だが，104 が非木エッジでつながれているため，その問合せの解は連続して配置されていない (図 1(d) のディスクイメージ)．したがって，101 $\stackrel{b*}{\rightarrow} X$ を処理するためには，その分のディスクアクセスが必要となる．

(注1)：本稿において，ディスクアクセス数とは，アクセスされるディスクのページ数ではなく，ディスクへのアクセス要求の数を指す．

表 1　問合せ q を処理するために必要なディスクアクセス

問合せ q		$a \stackrel{l}{\rightarrow} X$	$a \stackrel{l*}{\rightarrow} X$
論文 [3]	**GCDF/T**	$O(1)$	$O(1)$
	GCDF/G	$O(s_q)$	$O(t_q)$
本稿	**GCDF/G**	$O(s'_q)$	$O(t'_q)$ ただし $t'_q << t_q$
	GCDF/G$^+$	$O(1)$	$O(t'_q)$

以上をまとめると **GCDF/G** で，問合せ q を処理するために必要なディスクアクセス数は，表 1 の 2 行目のようになる．ここで，s_q, t_q は，問合せ q を処理する際に辿る必要のある非木エッジの数の期待値である．

したがって，問合せ処理時に辿る必要がある非木エッジの数が多いグラフの場合，問合せ処理では多くのディスクアクセスが発生し，問合せを効率的に処理できない可能性がある．

本稿では，対象がグラフデータであっても，論文 [3] で提案した単純な **GCDF/G** の適用と比較して，子供／子孫問合せを処理するために必要なディスクアクセス数が少なくなるようなディスク格納手法を提案する．具体的には次の 2 つの手法を提案する．

(1) **GCDF/G における全域木の選択手法**: 後述するように，**GCDF/G** においては全域木をどのように選ぶかによって，子孫問合せを処理するために必要なディスクアクセス数が大幅に異なる．そこで本稿では，少ない回数のディスクアクセスで子孫問合せが処理可能となるような全域木を選択するための手法を提案する．これにより，表 1 の 3 行目に示すディスクアクセス数で問合せを処理可能となる．ここで，s'_q, t'_q とは，本手法で構築した全域木で，問合せ q を処理する際に辿る必要のある非木エッジの数の期待値である．

(2) **子供問合せのディスクアクセスを減らした GCDF/G$^+$**: **GCDF/G** によって作成された全域木を加工することによって，いかなる子供問合せでも，問合せに依存せず定数回のディスクアクセス数で処理可能となる (表 1 の 4 行目) ような手法 **GCDF/G$^+$** を提案する．

関連研究: 我々がこれまでに示してきたノードの配置順序は，CAD/CAM アプリケーションを対象とした J. Banerjee らの研究 [4] の処理の一般化になっている．しかし，我々の知る限り，グラフデータに対する正規パス式の処理に関してこのようなアプローチで取り組んだ研究は存在しない．また，論文 [4] では，本稿が対象としているようなエッジラベル付きグラフは対象とされていない．子供／子孫問合せの高速化に関しては，(1)Reachability Queries [5]，(2)Structural Joins [6]，(3)XML を対象とした XPath 式などの問合せの高速化 [7] などが提案されている．このうち，(1)Reachability Queries や (2)Structural Join は，いずれも処理コストが解のサイズではなくデータのサ

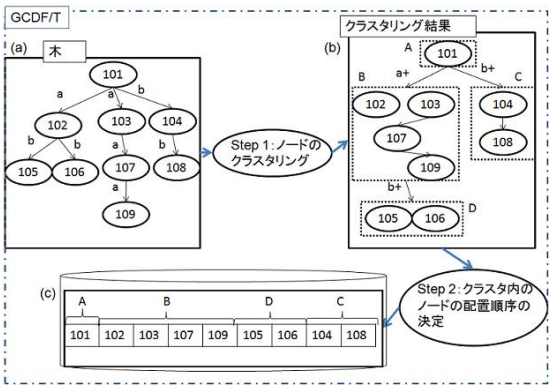

図 2　**GCDF/T** によるノードの配置順序のプロセス

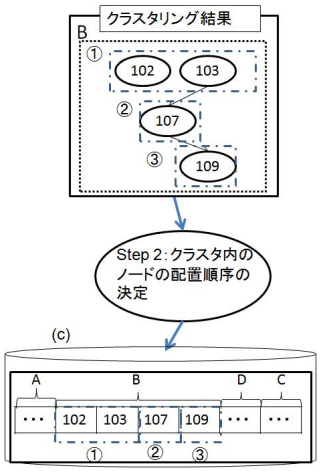

図 3　クラスタ B のノード配置順序

イズに依存する．一方，我々が論文 [3] で提案した手法の処理コストは**解のサイズ**に依存する．また，(3) に関しては，ルートからの問合せが対象とされており，我々が対象とする**任意のノードを起点とした問合せ**の高速化には対応していない．

本稿の構成は次の通りである．2 章では，我々が論文 [3] で提案した **GCDF/T** および **GCDF/G** について説明する．3 章では，本稿での提案手法について説明する．4 章では，実験について述べる．5 章はまとめと今後の課題である．

2.　GCDF/T と GCDF/G

2.1　GCDF/T

図 2 に **GCDF/T** によって木をディスクに格納するプロセスを示す．**GCDF/T** でのノードの配置順序は，ノードのクラスタリングとクラスタ内のノードの配置順序の決定という 2 つのステップで行われる．次に，各ステップの具体的な内容について説明する．

Step 1. ノードのクラスタリング

ここではエッジラベルごとにノードのクラスタリングを行う．たとえば，図 2(a) の木のクラスタリング結果は，図 2(b) となる (点線で囲まれたノード集合が一つのクラスタである)．具体的には，ノードのクラスタリングは次の 3 つのプロセスを適用することによって行う．

1.　ルートノードだけからなるクラスタを 1 つ構築する．

2.　ルートノードから同じエッジラベルのみで到達可能なノード集合をそれぞれ 1 つのクラスタとしてまとめる．

3.　**2.** でクラスタを構築する際に，別のエッジラベルが出現したとき，**2.** で得られたクラスタに含まれるノード集合をルートノードと見なし，再び **2.** を適用する．

例えば，図 2(a) の木をクラスタリングすると，図 2(b) となる．まず，**1.** に従い，ルートノードである 101 だけからなる 1 つのクラスタ A を構築する．次に，**2.** に従い，ルートノードから同じエッジラベル a のみで到達可能なノード集合，$\{102, 103, 107, 109\}$ と $\{104, 108\}$ をそれぞれクラスタ，B，C としてまとめる．最後に，**3.** に従い，クラスタ B を構築する際に，別のエッジラベル b が出現しているので，クラスタ B をルートノードとして，クラスタ B に含まれるノード集合からラベル b で到達可能な $\{105, 106\}$ を 1 つのクラスタ D としてまとめる．

ノードをクラスタリング後，各クラスタを深さ優先順に配置する．

Step 2. クラスタ内のノード配置順序の決定

ここでは，**Step 1** で作成した各クラスタ内のノードの配置順序を決定し，ディスクに配置する．図 2(c) に，図 2(b) をディスクに格納した際の順序を示す．クラスタ内のノードの配置順序は次の 3 つのプロセスによって決定する．

1. クラスタ内のノードのうち，元の木において同じ親を持つ兄弟ノードをそれぞれ 1 つのグループとしてまとめる．以下では，ここでグループ化したノード集合を**兄弟グループ**と呼ぶ．

2. 各兄弟グループに対して，深さ優先順に番号を振る．

3. 番号順に兄弟グループを配置する．グループ内の兄弟ノードの順序は，元の木での順序と同じとする．

具体的な例として図 2(b) におけるクラスタ B のノードの配置順を決定するプロセスを説明する．図 3 に、クラスタ B のノード配置順序を決定するプロセスを示す．まず，**1.** に従い，クラスタ B における兄弟ノードを一つのグループとしてまとめる．クラスタ B の場合，{102,103}，{107}，{108} がそれぞれ兄弟であるため，一つのグループとしてまとめられる (図 3(上))．その後，**2.** に従い，各兄弟グループを深さ優先順に番号を振る．図 3 の場合，それぞれを深さ優先順に番号を振ると，図 3(上) の①，②，③となる．最後に，**3.** に従い，番号順に兄弟グループを配置する．したがって，102,103,107,109 という順序で配置する．

2.2　GCDF/G

GCDG/G による，グラフの格納手法は次の 3 つのステップからなる．

Step 1. グラフから全域木を構築する．

Step 2. 全域木に対して **GCDF/T** を適用する．

Step 3. **Step 1.** で取り除かれたエッジをディスクに格納する．

例えば，図 1(a) のグラフに対して **GCDF/G** を適用する場合を考える．まず，**Step 1.** に従い，図 1(a) に示したグラフを図 1(b) に示した全域木と，103 から 104 への非木エッジの 2

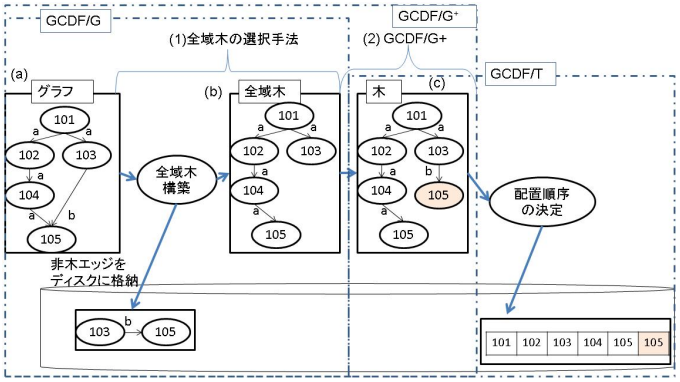

図 4　提案手法の概念図

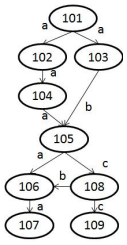

図 5　グラフ G_1

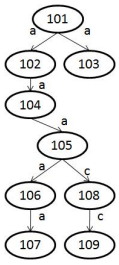

図 6　図 5 から構築可能な全域木 spt_1

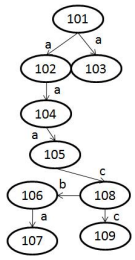

図 7　図 5 から構築可能な全域木 spt_2

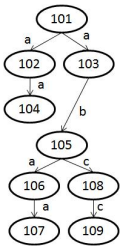

図 8　図 5 から構築可能な全域木 spt_3

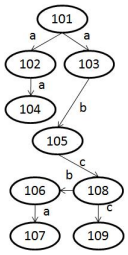

図 9　図 5 から構築可能な全域木 spt_4

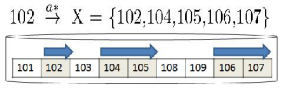

図 10　図 9 のディスクイメージ

つに分割する．次に，**Step 2.** に従い，図 1(b) の全域木に対して **GCDF/T** を適用する．最後に，**Step 3.** に従い，非木エッジの情報を図 1(c) に示すようにディスクに格納する．

3. 提 案 手 法

本章では，本稿で提案する手法について説明する．本稿での提案手法と，論文 [3] で提案した手法の関係を図 4 に示す．本稿で提案する部分は，**GCDF/G** における全域木の選択手法 (図 4(1)) と **GCDF/G**$^+$(図 4(2)) である．

3 章の構成は以下の通りである．まず，3.1 節で提案手法を説明するためにいくつかの用語を定義する．その後，3.2 節で全域木の選択手法について説明する．最後に，3.3 節で，**GCDF/G**$^+$ について説明する．

3.1 用語の定義

グラフを G=(V, E) とする．また，グラフに存在するエッジ $e \in E$ に対して，e の始点となるノードを $source(e)$，e の終点となるノードを $dest(e)$，e のラベルを $label(e)$ とする．例えば，図 5 における 101 から 102 へのエッジを e とした場合，$source(e)$=101，$dest(e)$=102，$label(e)$=”a” となる．

あるグラフ G に **GCDF/G** を適用し，全域木として T が用いられた時，問合せ q を処理するために必要なディスクアクセス数を $diskAccess(G, q, T)$ と表記する．具体的な例として，図 5 のグラフ G_1 に **GCDF/G** を適用し，全域木として spt_4 (図 9) を選択した場合のディスク格納イメージを図 10 に示す．このとき，G_1 に対する子孫問合せ $102 \overset{a*}{\rightarrow} X$ の解，{102,104,105,106,107} は，3 つの領域に分かれて記録されていることが分かる．よって，$diskAccess(G_1, 102 \overset{a*}{\rightarrow} X, spt_4) =$ 3 となる．

3.2 全域木選択手法

本節では，まず **GCDF/G** における全域木の選び方とディスクアクセス数との関係について説明し，続いてディスクアクセスが少なくなるよう全域木を選択する手法について説明する．

3.2.1 全域木の選択とディスクアクセスの関係

まず，グラフから適切な全域木を選ぶことによって，ディスクアクセスの数が変わる例を示す．図 6〜図 9 に示す木 (spt_1〜spt_4) は，図 5 のグラフ G_1 から構築可能な全域木である．

このとき，$G_1 = (V, E)$ に対して，解が存在する全ての子孫問合せを処理するために必要なディスクアクセスの総数と平均を次のように計算する (表 2)．まず，解が存在する全ての子孫問合せの集合 Q_{G_1} は次の通りである．

$$Q_{G_1} = \bigcup_{e \in E} \{source(e) \overset{label(e)*}{\rightarrow} X\}$$

Q_{G_1} に含まれる子孫問合せの数 ($|Q_{G_1}|$) は 9 である．このとき，必要なディスクアクセスの総数は次のようになる．平均はこれを $|Q_{G_1}|$ で割ったものである．

$$\sum_{q \in Q_{G_1}} diskAccess(G_1, q, spt).$$

表 2 を見ればわかるとおり，格納する全域木によってディスクアクセスの平均に 1.5 倍程度の差が生じる（表 2 の 3 列目）．したがって，どの全域木を選ぶかは重要な問題であることがわかる．

3.2.2 提案する全域木選択手法のアイデア

ディスクアクセスが少なくなるような全域木を選択するための基本となるアイデアは，グラフの各エッジ e に対して，**エッジポイント**と呼ぶ値 $point(e)$ を付与し，これを手がかりに全域木を選択する事である．$point(e)$ は，そのグラフから全域木を

表 2　9 種類の子孫問合せ処理に必要なディスクアクセスの総数と平均

	総数	平均
全域木 spt_1 (図 6)	18	2.00
全域木 spt_2 (図 7)	23	2.56
全域木 spt_3 (図 8)	25	2.78
全域木 spt_4 (図 9)	27	3.00

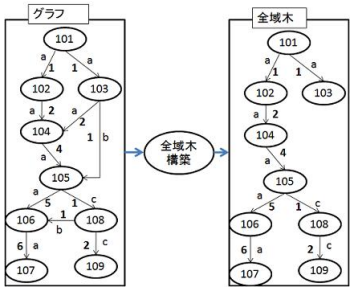

図 11　タイプ A のエッジポイント

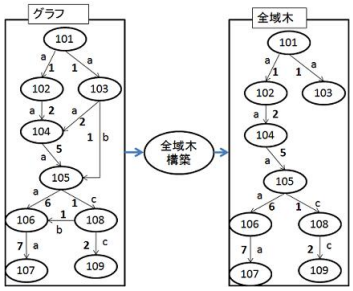

図 12　タイプ B のエッジポイント

作成するために e を削除すると，問合せ処理時にどの程度ディスクアクセスが必要となるかを表す評価値である．したがって，エッジポイントの少ないエッジを削除して全域木を作れば，問合せ処理時にディスクアクセスが少ない全域木を選択することができる．

3. 2. 3　エッジポイントの計算

本節では，2 種類のエッジポイントを比較する．

(タイプ A): e に到達可能な祖先の数.　$point(e)$ として，$a \overset{label(e)*}{\rightarrow} X$ で e を経由するような全てのノード a の個数を用いる．すなわち，エッジの終点 $dest(e)$ から，e を通り，かつ e と同一のエッジラベルを持つエッジを逆方向に辿ることで得られるノードの個数である．タイプ A のエッジポイントを付与した例を図 11 に示す．この図において，108 から 109 へのエッジのエッジポイントを考える．この場合，エッジの終点は 109 であり，e を通り，かつ e と同一のエッジラベル c のを持つエッジを逆方向にたどっていくと，該当するノードは {105,108} の 2 つとなる．よって，108 から 109 へのエッジのポイントは 2 となる．定義より，タイプ A のポイントを割り当て，もっとも小さな値のエッジを削除しながら全域木を選択すれば，$\sum_{q \in Q_G} diskAccess(G_1, q, spt)$ は最小になる．

(タイプ B)　e を経由するパスの総数.　$point(e)$ として，$a \overset{label(e)*}{\rightarrow} X$ で e を経由するような全ての (a から $dest(e)$ への) パスの数を用いる．タイプ B のエッジポイントを付与した例を図 12 に示す．図 12(左) における 104 から 105 からのエッジのポイントについて考える．エッジ e を経由する問合せは，{ 101 $\overset{a*}{\rightarrow}$ X,102 $\overset{a*}{\rightarrow}$ X,103 $\overset{a*}{\rightarrow}$ X,104 $\overset{a*}{\rightarrow}$ X} の 4 種類であり，これらの問合せで e を経由するパスの総数は { 101→102→104→105，101→103→104→105，102→104→105，103→104→105，104→105} の合計 5 つである．よって，そのエッジのポイントは，5 となる．

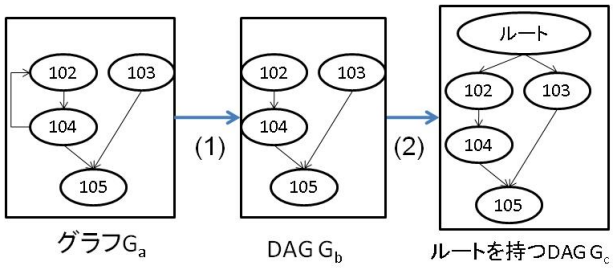

図 13　前　処　理

以上の 2 つを比較したとき，タイプ A のポイントの利用は最善の結果をもたらすが，ポイントの計算が困難である．一方，タイプ B のポイントを利用して，もっとも小さな値のエッジを削除しながら全域木を作成しても，$\sum_{q \in Q_G} diskAccess(G_1, q, spt)$ は必ずしも最小になるとは限らないが，タイプ B のポイントは次の利点を持つ．

(1) エッジポイントの計算が簡単である．タイプ B のエッジポイントは，親を終点とするエッジのエッジポイントから計算可能である．例えば，図 12 において 105 から 106 へのエッジのポイントを計算する場合，親，105 を終点としラベル”a”を持つエッジのエッジポイント 5+1 で計算可能である．

(2) タイプ A のエッジポイントの近似である．グラフが木の場合，タイプ A とタイプ B のエッジポイントの値は同じになる．それ以外では値が異なるが，後の実験で示すように，タイプ B のエッジポイントを用いても実用上効率の良い全域木を選択できている．

3. 2. 4　タイプ B のエッジポイントを求めるアルゴリズム

本アルゴリズムは，3 つのステップからなる．**Step 1.** まず，前処理としてグラフからただ 1 つのルートからなる DAG を構築する．**Step 2.** 次に，**Step 1.** で構築したグラフの各ノードをトポロジカルソートする．**Step 3.** 最後に，ソートされた順序に従って，各エッジポイントを計算する．

Step 1. 前処理

まず前処理として，エッジポイントを計算するグラフ G_a から，ルートを持つ DAG である G_c を構築する．図 13 に前処理の例を示す．説明の簡略化のため，エッジラベルは省略している．

ルートを持つ DAG を構築するために，元々のグラフに対して次の 2 つの処理を行う．(1) グラフ G_a からエッジを取り除き，DAG を構築する．この結果を G_b とする．(2) G_b に対して，ノードを 1 つ追加し，追加したノードから，G_b において終点とならないノードにエッジを張る．この処理で出来たグラフを G_c とする．これらの詳細は次に説明する．

(1) まず，グラフ G_a からエッジを取り除き，DAG を構築す

```
Input: グラフ G_a におけるノード集合 V
 1:for each v ∈ V{
 2:  v.isVisited = v.currentVisited = false
 3:}
 4:for each v ∈ V{
 5:  if (!v.isVisited) deleteEdge(v)
 6:}
 7:deleteEdge(Node v){
 8:  v.isVisited = v.currentVisited = true
 9:  for each c ∈ v's children{
10:    if (c.currentVisited) delete(v -> c)
11:    else if (!c.isVisited) deleteEdge(c)
12:  }
13:  v.currentVisited = false
14:}
```

図 14　グラフから DAG である G_b を構築するための擬似コード

る．図 13 のグラフ G_a の場合，102 から 103 へのエッジを取り除くことで，グラフから DAG，G_b を構築することが可能となる．

図 14 にエッジを取り除き，DAG，G_b を構築するための擬似コードを示す．擬似コードでは，エッジを取り除くために，各ノードに対して，2 つのフラグを立てている．1 つは既に訪問したことを表す $isVisited$ フラグ，もう 1 つは，現在訪問中である事を表す $currentVisited$ フラグである．1〜3 行目では，2 つのフラグの値を初期化する．4〜6 行目では，グラフの各ノード v に対して，$deleteEdge()$ 関数を呼び出している．7〜14 行目は，DAG を構築するためにエッジを取り除くための関数，$deleteEdge()$ である．$deleteEdge()$ 関数では，グラフの各ノードを深さ優先順に訪れ (8 行目〜12 行目)，現在訪問中のノードを再び訪れた場合，エッジを取り除く (10 行目)．

(1) における手法において，G_a から DAG を構築する際，どのエッジを削除するかは特に限定していない．例えば，図 13 の G_a において，104 から 102 へのエッジを削除する (図 13 の G_b) のではなく，102 から 104 へのエッジを削除しても良い．

(2) その後，G_b に対して，ダミールートを作成し，ダミールートから G_b におけるルートノードへエッジを張る．図 13 の G_b の場合，まずダミールートを作成し，G_b におけるルートノード，102 と 103 に対してエッジをエッジを張る．

Step 1. では，グラフ G_a から，G_a とは異なるグラフである G_c を構築するため，G_a におけるエッジポイントの値と G_c におけるエッジポイントの値は異なる可能性がある．しかし，ディスクアクセスの総数が最も小さくなる全域木を選択するタイプ A とは違い，タイプ B は，近似であるため，それほどの問題にはならないと考えられる．

Step 2. ノードのトポロジカルソート

次に，グラフノードのトポロジカルソートを行う．図 15(下) は，図 15(上) のグラフをトポロジカルソートした結果である．説明の簡略化のためエッジラベルは省略している．

Step 3. エッジポイントを計算

最後に，トポロジカルソートされた順に各ノードを見ていき，それぞれのノードを終点とするエッジ e のポイント $point(e)$ を計算していく．3.2.5 節で述べたように，各エッジのポイントは親へのエッジのポイントから計算可能である．

エッジのポイントを計算するための擬似コードを図 16 に示す．2 行目から 5 行目では，ノード v を終点とする各エッジのポイントの計算を行う．まず，3 行目で親 p から v へのエッジ

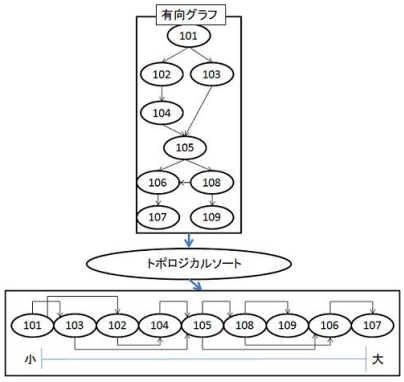

図 15　(例) トポロジカルソート

```
Input: トポロジカルソートされたノードのリスト V
 1:for each v ∈ V{
 2:  for each p ∈ v's parent{
 3:     edgeLabel = label(p->v)
 4:     point(p->v) = p.pathes(edgeLabel) + 1
 5:  }
 6:}
```

図 16　エッジのポイントを計算するための擬似コード

のラベルを取得する．その後，4 行目で親ノード p へ張られたエッジのうち，エッジラベルが 3 行目で示した変数 edgeLabel となっているものを取得し，その値 $+1$ をノード p から v へのエッジのポイントする．

3.2.5　計　算　量

まず，**Step 1** における計算量について考える．**Step 1** では，グラフを深さ優先順に訪れるだけであるため，その計算量は $O(|V|+|E|)$ となる．次に，**Step 2** におけるトポロジカルソートの計算量は $O(|V|+|E|)$ であることが知られている．最後に，**Step 3** でエッジポイントを計算するために必要な計算量は，図 16 のアルゴリズムにおいて，ノードおよびエッジをたかだか 1 度しか読まないことから $O(|V|+|E|)$ である．以上から，本手法で示したアルゴリズム全体の計算量は $O(|V|+|E|)$ となり，グラフのサイズに対して線形のオーダとなる．

3.3　GCDF/G$^+$

GCDF/G$^+$ は基本的には **GCDF/G** と同じであるが，子供問合せ処理に必要なディスクアクセスの回数を定数回とする (表 1 の 4 行目) ために，全域木を加工する点が異なる．具体的には，ディスクアクセスを定数回にするために，全域木 T から非木エッジでつながれた先のノードをコピーした木を構築する．

具体的な例として図 4(a) のグラフの場合を考える．このグラフから図 4(b) の全域木が作成されたとする．**GCDF/G$^+$** においては，図 4(b) の全域木から非木エッジでつながれた先のノードをコピーした木，図 4(c) を構築する．

その後は，**GCDF/G** と同様に **GCDF/T** を適用し，また，非木エッジの情報をディスクに格納する．

1,2 章で述べたように，**GCDF/T** では，任意のノードからの子供は連続して配置される．したがって，子供情報をコピーした木，図 4(c) の木をディスクに格納することで，ディスク上で子供となるノードは連続して配置される．よって，子供問合せに必要なディスクアクセス数は，問合せにかかわらず定数回となる．

表 3　実験データにおけるエッジラベルごとの子孫問合せの総数

子孫問合せの種類	子孫問合せの総数
$a \xrightarrow{Contains*} X$	35,970
$a \xrightarrow{cp*} X$	38,096
$a \xrightarrow{path*} X$	555,322
$a \xrightarrow{refersTo*} X$	22,258
$a \xrightarrow{size*} X$	518,915
$a \xrightarrow{updateTime*} X$	518,915
合計	1,689,476

表 4　実験 1. 実験 1 で用いたグラフの詳細

	ノード数	エッジ数	構築可能な全域木の総数
G_1	114	120	54
G_2	148	158	864
G_3	186	203	110,592

4. 実　　験

提案手法を用いた場合のディスクアクセス数の削減効果の検証をするための実験を行った．

実験環境は，次の通りである．OS:Windows Vista Business. CPU:Intel Core2 DUO T7300 2.00GHz．メモリ:2.5GB．プログラムの実装は，Java で行った．

4.1　実験データ

実験で用いたデータは，著者が所属する研究室のファイルサーバのファイルの情報が格納されたグラフである．このグラフには，555,322 ファイルの情報が格納されている．グラフは，ファイル／ディレクトリを表す File/Dir ノードとそのメタデータの値を保持するノードで構成されている．メタデータとして，ファイルの path，size，updateTime を保持している．すなわち，全ての File ノードは path，size，updateTime というエッジラベルでつながれたノードを子供として持ち，全ての Dir ノードは path というエッジラベルでつながれたノードを子供として持つ．また，File/Dir ノード間には，ディレクトリの階層関係を表す Contains，ファイルの参照を表す refersTo，ファイルのコピーを表す cp の 3 種類のエッジが張られている．

このグラフに関して，解が存在する子孫問合せの数を表 3 に示す．

4.2　実験 1. 提案手法の全域木と全ての全域木との比較

実験 1 では，提案手法で構築する全域木とそれ以外の構築可能な全ての全域木について，ディスクアクセス数の比較を行った．一般に，グラフから構築可能な全域木の数は膨大であるため，ここでは比較的小さなグラフを用いて実験を行った．実験に用いたデータは，4.1 節で説明したグラフの部分グラフである．具体的には，4.1 節で示したデータのうち，31 ファイル，41 ファイル，51 ファイルで構成された 3 つのグラフ (G_1，G_2，G_3) を用いた．各グラフのノード数，エッジ数，構築可能な全域木の数を表 4 に示す．また，各グラフに関して，解が存在する子孫問合せの数を表 5〜表 7 に示す．

G_1，G_2，G_3 のそれぞれのグラフにおいて，提案手法におけるタイプ B のエッジポイントを用いて選択した全域木と，それ以外の全ての全域木で，表 5〜表 7 に示した各子孫問合せを処理し，必要なディスクアクセスの総数を計算した．

表 5　G_1 におけるエッジラベルごとの子孫問合せの総数

子孫問合せの種類	子孫問合せの総数
$a \xrightarrow{Contains*} X$	5
$a \xrightarrow{path*} X$	31
$a \xrightarrow{refersTo*} X$	2
$a \xrightarrow{size*} X$	26
$a \xrightarrow{updateTime*} X$	26
合計	90

表 6　G_2 におけるエッジラベルごとの子孫問合せの総数

子孫問合せの種類	子孫問合せの総数
$a \xrightarrow{Contains*} X$	8
$a \xrightarrow{path*} X$	41
$a \xrightarrow{refersTo*} X$	3
$a \xrightarrow{size*} X$	33
$a \xrightarrow{updateTime*} X$	33
合計	118

表 7　G_3 におけるエッジラベルごとの子孫問合せの総数

子孫問合せの種類	子孫問合せの総数
$a \xrightarrow{Contains*} X$	9
$a \xrightarrow{path*} X$	51
$a \xrightarrow{refersTo*} X$	5
$a \xrightarrow{size*} X$	42
$a \xrightarrow{updateTime*} X$	42
合計	149

表 8　実験 1. 問合せを処理するために必要なディスクアクセスの総数

グラフ	提案手法	最小	最大	平均
G_1	180	180	184	181.45
G_2	235	235	242	239.41
G_3	295	295	305	302.44

実験 1 の結果を，表 8 に示す．表 8 の 2 列目は提案手法で選択した全域木に対して，各子孫問合せを処理した際に必要なディスクアクセスの総数である．3 列目，4 列目，5 列目はそれぞれグラフ，G_1，G_2，G_3 から構築可能なそれぞれの全域木に対して，各子孫問合せを処理する際に必要な最小，最大，平均のディスクアクセスの総数である．

表 8 より，少なくとも今回選んだデータに関しては提案手法によって，最もディスクアクセスの総数が少なく全域木を選択出来た事が分かった．表 8 では，ディスクアクセスの総数にそれほど大きな差が発生していないが，これは対象となるデータが非常に小さいからである．実際，次節で示す巨大データでの実験ではディスクアクセスの総数に 2 倍程度の差が発生した．

4.3　実験 2. 巨大データでのディスクアクセス数

実験 2 では，4.1 節で示した実験データのグラフから構築可能な全域木からランダムに選択した 10 個の全域木と，提案手

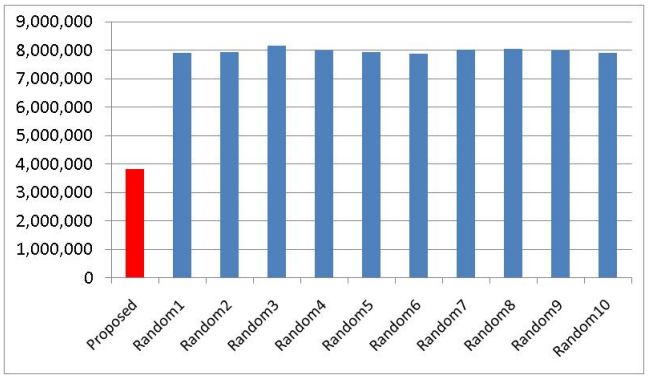

図 17　実験 2. 実験結果： ディスクアクセスの平均値

法におけるタイプ B のエッジポイントを用いて選択した全域木の合計 11 の場合において，表 3 に示した各子孫問合せを処理する際に必要なディスクアクセスの平均値を計算した．その結果を図 17 に示す．図中での Prposed は本手法で選択した全域木であり，$\texttt{Random}_i$ は，ランダムに 10 個選んだ全域木である．結果から，少なくとも今回対象としたデータにおいては，提案手法で全域木を選択した場合に，ディスクアクセス数が大幅に削減できたことがわかる．

5. おわりに

本稿では，任意のグラフノードを起点とする正規パス式の効率的な評価を実現するための，ディスク格納手法を提案した．具体的には，問合せを処理する際に必要なディスクアクセスの回数が少なくなるようにグラフから全域木を作成することで，問合せを効率的に処理する手法を提案した．今後の課題としては，**GCDF/G**$^+$ における性能向上と必要なディスクサイズとのトレードオフの検証，およびグラフの更新への対応などがあげられる．

謝　　辞

ゼミなどでコメントいただきました筑波大学大学院図書館情報メディア研究科の杉本重雄教授，阪口哲男准教授，永森光晴講師に感謝いたします．本研究の一部は科学研究費補助金若手研究 (B)(#20800076) による．

文　　献

[1] Ashburner, M. et al. Gene Ontology: tool for the unification of biology. Nature Genetics 2000: Vol. 25, pp. 25-29.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. L. Wiener: The Lorel Query Language for Semistructured Data. Int. J. on Digital Libraries 1(1): 68-88 (1997)

[3] 只石正輝，森嶋厚行，田島敬史．効率のよい子供／子孫問合せ処理のためのグラフデータ格納手法の提案．日本データベース学会 Letters, Vol. 7, No. 3, pp. 55-60, 2008 年 12 月, 日本データベース学会.

[4] Jay Banerjee, Won Kim, Sung-Jo Kim, Jorge F. Garza: Clustering a DAG for CAD Databases. IEEE Trans. Software Eng. 14(11): 1684-1699 (1988)

[5] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, Jeffrey Xu Yu: Dual Labeling: Answering Graph Reachability Queries in Constant Time. ICDE 2006: 75-86.

[6] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, Divesh Srivastava: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. ICDE 2002: 141-152

[7] Wei Wang and Haifeng Jiang and Hongzhi Wang and Xuemin Lin and Hongjun Lu and Jianzhong Li, “Efficient processing of XML path queries using the disk-based F&B Index,” VLDB 2005, 145-156, 2005.