平成 24 年度 論文

論文題目

推移閉包アルゴリズムを用いた Covert Channel 検出

神奈川大学 工学部 電気電子情報フロンティア学科

学籍番号　　200803007

中村　峻生

指 導 担 当 者　　木 下 宏 揚　　教授

目次

図目次

表目次

第 1 章

序論

近年，インターネットやイントラネットなどのネットワークの普及により，情報へのアクセス（読み・書き・実行）が容易になった．個人情報や企業機密を含んだデータをネットワークシステムに組み込むことで，より便利なシステムが実現されている．しかし，機密情報に誰でも自由にアクセスできる状況は好ましくない．そこで，データにアクセスできる者を制御するセキュリティ技術として，アクセス制御（Access Control）が発展してきた．これによりデータは，権限を持たない者によって，読まれること・改ざんされること・消去されることなどの脅威から保護される．

それでも，アクセス制御上の脆弱性によって情報漏洩が起きる可能性はある．情報漏洩を引き起こす脆弱性の一つに，covert channel と呼ばれる不正通信路がある [1, 2, 3]．covert channel はシステム上の欠陥によって発生するものではない．アクセス制御で許可されている権限によって連鎖的に情報が伝播することで，本来アクセスを禁止されている者が情報を入手できてしまうといった，潜在的な問題点によって発生する脆弱性である．

当然ながら，機密データを持つネットワーク上に covert channel が存在すると情報漏洩というリスクが顕在化する．セキュリティの観点から，covert channel の検出技術や，covert channel の制御技術の発展が望まれている．

本論文は，covert channel 検出問題の数学的な解析と効率的な covert channel 検出アルゴリズムについて記している．

以降本章では，問題の本質と範囲についての議論，研究目的の列挙，研究方法の解説，研究の主な結果の記載，アウトラインの提示を行う．

1.1　目的

1 番目の目的は，covert channel 検出が数学的にどのような問題であるか明かすことである．そこで，ACL(Access Control List[1, 2]) をグラフとして捉え，covert channel 検出をグラフ理論によって整理した．

2 番目の目的は，従来の方法より効率的な covert cahnnel 検出アルゴリズムを見つけることである．1 番目の目的を果たすことで，この目的に対して良い視座が得られる．

1.2　方法

実社会への実用化を考るとき，アルゴリズムの平均計算量は重要である．そのため，平均計算量を研究する方法が必要だった．しかし，平均計算量に関する一般的な数学的解析方法は存在せず，最悪計算量の数学的解析よりはるかに難しい．

そこで，コンピュータ・シミュレーションによって平均計算量を研究した．この選択の恩恵は，全ての入力グラフモデルと全ての性能指標に対して，同じ技術を使えるということである．新しい性能評価を導入するために，測定基準の数値を収集するコードを挿入するだけで済む．その他の利益は，数学的解析より数値的に正確な結果が得られるということであろう．シミュレーションを走らせ，シミュレーションの出力を分析する際に統計手法を用いて，結果の精度を確認した．

シミュレーションにおける弱点は，漸近的性能の情報を得られないことである．得られる情報は，空間計算量的にシミュレーション可能な入力に対して，アルゴリズムがどのように振舞うかということだけである．

1.3　提案

本論文の主な結果は次の通りである．

- covert channel 検出問題は推移閉包問題の応用ないし変形問題であることを示した．
- 一般的に推移閉包アルゴリズムを covert channel 検出アルゴリズムに応用できることを示した．
- 効率的な covert channel 検出アルゴリズムを示した．このアルゴリズムは Esko の STACK_TC[4] を用いる．
- 提案した covert channel 検出アルゴリズムが従来の covert channel 検出アルゴリズムより高速であることをシミュレーション実験により示した．

1.4　アウトライン

第 2 章では，covert channel の諸定義を行う．

第 3 章では，本論文で用いるグラフ理論の用語の定義を行う．また，グラフ理論を踏まえて covert channel 検出問題がどのような問題であるか明かす．その際，covert channel 検出問題をグラフ理論を用いて再定義する．

第 4 章では，先行研究の紹介として，従来のアルゴリズムを推移閉包アルゴリズムの観点から解説する．

第 5 章では，基本的な推移閉包アルゴリズムを変換して covert channel 検出アルゴリズムを示す．

第 6 章では，強連結成分分解に基づく推移閉包アルゴリズムを用いた covert channel 検出アルゴリズムを提案する．

第 7 章では，効率的に凝縮グラフの推移閉包を表すための方法を紹介する．トポロジカルソートとその符番に基づいた，区間表現について解説する．

第 8 章では，シミュレーション実験によって，第 6 章で提案したアルゴリズムの平均計算量を示し，従来の方法とそれらを比較する．

第 9 章では，論文の結論を示す．

第 2 章

covert channel 検出問題

セキュリティモデルは，アクセス制御システムを構築する上で，セキュリティポリシーを具体的な論理的形式で表現したものであり，そこには制御したいサービスや組織の構造が反映される．一般に，アクセス制御のための基本的な要素は，object(客体)，subject(主体) である [1, 2]．object はデータベースに格納する情報である．subject はデータベースに格納されている object にアクセスする行為者である．最も単純な型での要素間の関係は，R(Read:読み込み可), W(Write:書き込み可能),RW(Read+Write:読み書き可能), φ (Phi:読み書き不可) の 4 種類のアクセス権限 (permission) がある．これら permission をリストで表したものをアクセス制御リスト (ACL;Access Control List) と呼ぶ.

定義 2.1　集合 $O := \{x | x \text{ is object.}\}$ を object 集合と呼ぶ．集合 $S := \{x | x \text{ is subject.}\}$ を subject 集合と呼ぶ．ある object $x \in O$, とある subject$y \in S$ において，y が x を read 可の時 xRy と書き，y が x に write 可の時，yWx と書く．また，y が x を read 不可の時 xR^cy と書き，y が x に write 不可の時，yW^cx と書く．

ある ACL 上の object $o_1, o_2, \cdots, o_n \in O$ と, subject$s_1, s_2, \cdots, s_n \in S$ について, $o_1R^cs_n \wedge o_1Rs_1 \wedge s_1Wo_2 \wedge o_2Rs_2 \wedge \cdots \wedge o_nRs_n$ が成り立つとき，o_1, s_n 間に covert channel が存在すると言う．また，$o_1Rs_1 \wedge s_1Wo_2 \wedge o_2Rs_2 \wedge \cdots \wedge o_nRs_n$ を ACL 上の covert channel と呼ぶ.

例 2.1　object 集合，subject 集合がそれぞれ，$O = \{o_1, o_2, o_3\}$，$S = \{s_1, s_2s_3\}$ である時，permisson として，o_1Rs_1, o_1Rs_2, o_2Rs_2, o_3Rs_3, s_1Wo_1, s_2Wo_2, s_2Wo_3. が許可されているとする．このとき，$o_2R^cs_3$ であるが，o_2Rs_2, s_2Wo_3, o_3Rs_3 が成り立つ．したがって，o_2 , s_3 間には covert channel o_2Rs_2, s_2Wo_3, o_3Rs_3 が存在する．

具体的に言えば，s_3 が o_2 を read 不可であるのに，s_2 が o_2 を read し，その内容を o_3 に write し，s_3 が o_3 を read すると，s_3 は o_2 の内容を間接的に read できる．この矛盾した状態を covert channel が存在すると呼んでいる．また，この隠れた情報流出経路 (間接的な read 権限) を covert channel と呼んでいる．　□

covert channel にはいくつかの定義が存在し，文献 [3] に詳細にまとめられている．本論文の定義が一般的な covert channel の定義というわけではない．

定義 2.2　covert channel 検出問題とは，各 object，subject 間の covert channel の存在有無を判定する問題である．

例 2.2　object $x \in O$，subject $y \in S$ の間に covert channel が存在するとき，かつその時に限り $xR^{\#}y$ と書こう．すると，例 2.1 における covert channle 検出の解は $o_1R^{\#}s_3$，$o_2R^{\#}s_3$ である．□

2.1　アクセス制御

本論文において，アクセス制御は各 object と subject の組にそれぞれ permission を設定する単純なモデルを採用する．これは素朴なアクセス制御であり，一般性を持つ．一方で，高度なアクセス制御も存在する．その中のいくつかのアクセス制御モデルについて covert channel の観点から解説する．

2.1.1　Bell and LaPadula Model

Bell and LaPadula Model[5] は多階層セキュリティをコンピュータシステム上に実現するためのモデルである．アクセス行列に機密レベルの考えを加え，安全な状態を保つようにアクセスのルールを決めている．object と subject はそれぞれレベルを持つ．ある object x のレベルを $level(x)$，ある subject y のレベルを $level(y)$ とする．この時，$level(x) < level(y)$ ならば，y は x を read できるが，x に write できない．$level(x) > level(y)$ ならば，y は x に write できるが，x を read できない．$level(x) = level(y)$ ならば，y は x を read できるし，x に write できる．したがって，高位レベルの情報が下位レベルに流れることはない．逆に下位レベルの情報は上位レベルに流れる．

object x のレベルを $level(x) = k$ とする．subject y のレベルが $k \leq level(y)$ であるとき，xRy が成り立つ．すなわち，xR^cy となる y は存在せず，x, y 間に covert channel は存在しない．$xRs_1 \wedge s_1Wo_2 \wedge \cdots \wedge o_nRs_n$ の各レベルに着目すると，$level(x) \leq level(s_1) \leq level(o_2) \leq \cdots \leq level(o_n) \leq level(s_n)$ が成り立つ．ここで，subject z のレベルが $k > level_S(z)$ であるとき，任意の s_n に対して $z \neq s_n$ が成り立つ．ゆえに，x, z 間に covert channel は存在しない．したがって，このモデルにおいて covert channel は存在しない．

2.1.2　Chinese Wall

Brewer と Nash は，特定の業界内での情報アクセス保護の必要性を説き，そのためのモデルを定義した [6]．このモデルは，競合 (利害衝突) を情報保護の基本としている．

Chinese Wall では，任意の object はただ一つの企業グループに属する．任意の企業グループはただ一つの競合クラスタに分類される．ある subject y がある object x にアクセスしたとき，x を所有する企業と競合する企業の object に対して，y はアクセスできなくなる．この規則に従ってアクセス権限が付与される．

ある企業 A の所有する object の集合を $Data(A) \subset O$ で表し，A が分類される競合クラスタを $Conflict(A)$ で表そう．企業 A, B, C に対して，$B \in Conflict(A)$, $C \notin Conflict(A)$ が成り立つとする．ある subject s_a が object $o_a \in Data(A)$ にアクセスする．すると s_a は A と競合する B の object $o_b \in Data(B)$ にはアクセスできなくなる．ある subject s_b が object $o_b \in Data(B)$ にアクセスする．すると s_b は B と競合する A の object $o_a \in Data(A)$ にはアクセスできなくなる．しかし，s_a，s_b は $o_c \in Data(C)$ にアクセスできる．しかるに，$o_a R^c s_b \wedge o_a R s_a \wedge s_a W o_c \wedge o_c R s_b$ が成り立つ．したがって，o_a，s_b 間には covert channel $o_a R s_a \wedge s_a W o_c \wedge o_c R s_b$ が存在する．

2.1.3　Role Based Access Control

従来のアクセス制御では，user にアクセス権限の設定を行うことで制御を行ってきた．しかし，政府や企業の大規模システムにおける user 数 (subject の数) や file 数 (object の数) は数万〜数十万に及ぶ．各 user と file の対一つ一つにそれぞれアクセス権限を設定すると，ACL は膨大になり，permission の管理が煩雑になりやすい．そこで，Role Based Access Controld[7] は role(役割) を設定し，これを解決した．user は一つ以上の role が割り当てられ，user ではなく role に対して permission が設定される．これにより，permission の管理の煩雑さを回避している．

役割が n 個あって，役割 i に属する subject の集合を $role_i (i = 1, 2, \cdots, n)$ と書こう．subject x に割り当てられた役割の集合を $Role(x) := \{role_i | x \in role_i; i = 1, 2, \cdots, n\}$ で書く．ある subject $x, y \in S$ について同値関係 $x \sim y$ を $x \sim y \Leftrightarrow Role(x) = Role(y)$ で定義する．role に対して permission が設定されているので，$x \sim y$ であるとき x と y には同じ permission が設定されている．したがって，covert channel $o_1 R s_1 \wedge \cdots \wedge o_i R x \wedge x W o_{i+1} \wedge \cdots \wedge o_n R s_n$ が存在するとき．$o_1 R s_1 \wedge \cdots \wedge o_i R y \wedge y W o_{i+1} \wedge \cdots \wedge o_n R s_n$ も covert channel となる．つまり，$x \sim y$ であるとき，covert channel という観点から x と y を区別する必要がない．そこで，Role Based Access Control では，S の $\sim$ による商集合 $S/\sim$ に関してのみ covert channel を考えれば良い．

第 3 章

グラフ理論

本章では，covert channel 検出の研究に必要なグラフ理論の用語を定義する．本章で定義されないその他の概念は，関連性がより明確になる箇所で逐次，定義していく．

アルゴリズムの分析は，ランダムアクセスマシン (RAM) モデルを用いる [8]．なお漸近的記法，ビッグオー ($\mathcal{O}$)，ビッグオメガ (Ω)，ビッグシータ (Θ) は，プログラムの実行に必要な実行時間とメモリ容量の増加率の漸近的な評価をするために用いる [9]．これらは次式で定義される．

定義 3.1

$$f(n) = \mathcal{O}(g(n)) \Leftrightarrow (\exists c, n_0(c))(\forall n)(n \geq n_0 \Rightarrow f(n) \leq cg(n))$$
$$f(n) = \Omega(g(n)) \Leftrightarrow (\exists c, n_0(c))(\forall n)(n \geq n_0 \Rightarrow f(n) \geq cg(n))$$
$$f(n) = \Theta(g(n)) \Leftrightarrow (\exists c_1, c_2, n_0(c_1, c_2))(\forall n)(n \geq n_0 \Rightarrow c_1 g(n) \leq f(n) \leq c_2 g(n))$$

本論文ではある集合 A の濃度を $|A|$ で表すが，漸近的記法の中だけに限って記号 $|A|$ を記号 A で表す．たとえば，「このアルゴリズム は $\mathcal{O}(AB)$ で走る」と言うとき，このアルゴリズムが $\mathcal{O}(|A||B|)$ で走ることを意味する．この約束は実行時間の式を読みやすくし，曖昧さの危険性もない．

写像 $f : A \to B$ において，A の部分集合 $A' \subset A$ に対して，$f(A') := \{f(x)|x \in A'\} \subset B$ を f による A' の像と呼ぶ．全ての $x \in A'$ について，それぞれ $f(x)$ を求めることを単に，$f(A')$ を求めるとか，f による A' の像を求めると言うことにする．

3.1　グラフ理論の準備

グラフ理論はいくつかの文献によって定義が異なっているため，ここで基本概念を定義する．詳しい定義は参考文献 [10, 11] を参照されたい．

定義 3.2　有向グラフ G とは，集合 V と，集合 $E \subseteq V \times V$ の組 (V, E) である．V の要素を頂点と呼び，V を頂点集合と呼ぶ．E の要素を辺と呼び，E を辺集合と呼ぶ．辺 (x, y) において，先に来る x が辺の tail であり，後に来る y が辺の head である．有向グラフ $G = (V, E)$ の有向部分グラフ $G' = (V', E')$ とは，$V' \subseteq V$，と $E' \subseteq E$ を満たす有向グラフである．

本論文では，有向グラフのみを研究するため，以降，有向という修飾語を省略して有効グラフを単にグラフと言う．また，頂点集合，辺集合は有限集合であるとする．いくつかの論文がグラフで $(x,\ x)$ の形をした自己ループの辺を禁止している．自己ループは本論文において全く困難とならないので，自己ループを許容する．

定義 3.3　$(x,\ y)$ が G の辺であれば，x から y へ隣接していると言い，y へ x から隣接していると言う．y へ隣接している頂点の数は，y の入次数といい $Indeg(y)$ で表す．そして，x から隣接している頂点の数は，x の出次数といい，$Outdeg(x)$ で表す．

辺は tail と head を一つづつ持つので，各辺はそれぞれの tail の出次数に 1 だけ寄与し，それぞれの head の入次数に 1 だけ寄与する．しかるに，入次数と出次数，辺数は次式の関係で結ばれる．

$$\sum_{x \in V} Indeg(x) = \sum_{x \in V} Outdeg(x) = |E| \tag{3.1}$$

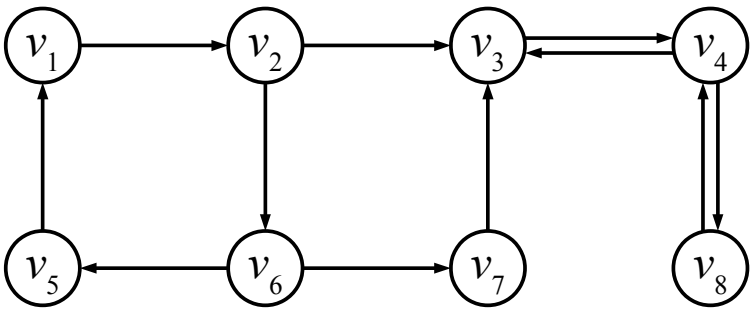
グラフを表現する標準的なデータ構造が 2 つある．1 つめの表現は隣接行列である．

定義 3.4　グラフ $G = (V, E)$ の隣接行列は $|V| \times |V|$ ブール行列 A である．各頂点にはある方法で $1, 2, \cdots, |V|$ と番号が振られていると仮定する．このとき，隣接行列の i, j 成分 $A[i, j]$ は G が辺 $(v_i, v_j) \in V \times V$ を持つ時，かつその時に限り **true** である．

隣接行列では，辺 (v_i, v_j) の存在を確認するときの計算量は $\mathcal{O}(1)$ である．v へ隣接している頂点の数，v から隣接している頂点の数にかかわらず，入次数や出次数を調べるのに $\Theta(V)$ 時間かかる．隣接行列の主な欠点はグラフが疎 (sparse) であるとき，言い換えれば辺が $|V|^2$ より非常に小さいときにさえ，$\Theta(V^2)$ の計算量を必要とすることである．例えば，隣接行列がディスクファイル上にあるなら，単に行列を読み込むだけで $\Omega(V^2)$ 時間かかる．疎グラフのより適した表現は隣接リスト表現である．

定義 3.5　頂点 x の隣接リスト $AdjFrom(x)$ は, x から隣接している頂点を含むリストである．グラフの隣接リスト表現は頂点の隣接リストから構成される．隣接リスト表現は $\mathcal{O}(V+E)$ の空間を必要とする．頂点 x から隣接している頂点を数え上げるには $\mathcal{O}(Outdeg(x))$ 時間かかる．また，グラフの全ての辺を列挙するには $\mathcal{O}(V+E)$ 時間かかる．辺 (x,y) の存在を確認するのに，$\mathcal{O}(Outdeg(x))$ 時間かかる．頂点 x へ隣接している頂点を数え上げるには，それぞれの隣接リストで x の存在を確認しなければならないので，$\mathcal{O}(V+E)$ 時間かかる．頂点 x へ隣接している頂点がしばしば必要であるなら，私たちはそれらを格納するもう 1 つのリスト $AdjTo(x)$ を使用できる．グラフが密 (dense) であるなら，すなわち $|V|^2$ 近く辺数があるならば，隣接行列表現は隣接リストより効率的である（隣接リストは一つの辺を格納するためのコストが隣接行列より高い）．

例 3.1　図 3.1 のグラフ $G=(V,E)$ を例に示す．頂点は円で表現され，辺は tail から head まで伸びる矢印として表現されている．G の下に，その隣接行列と隣接リスト表現を示している．□



$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\left(\begin{array}{cccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}\right)
\end{array}
$$

$$
\begin{aligned}
AdjFrom(v_1) &= \{v_2\} \\
AdjFrom(v_2) &= \{v_3, v_6\} \\
AdjFrom(v_3) &= \{v_4\} \\
AdjFrom(v_4) &= \{v_3, v_8\} \\
AdjFrom(v_5) &= \{v_1\} \\
AdjFrom(v_6) &= \{v_5, v_7\} \\
AdjFrom(v_7) &= \{v_3\} \\
AdjFrom(v_8) &= \{v_4\}
\end{aligned}
$$

図 3.1　An example graph $G=(V,E)$ and its representations

定義 3.6　G の頂点 v_0 から v_k へのパスとは c で表されるもので，$((v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k))$ の形をした辺の系列である．パスの長さはそれを構成する辺数に等しい．もし，長さが 0 より大きいなら，x から y までのパスは非ヌルであると言い，$x \xrightarrow{+} y$ で表す．最初の頂点と最後の頂点を除いてパス上の全ての頂点が異なる頂点であれば，そのパスは単純であるという．サイクルは最初と最後が同じ頂点である非ヌルな単純パスである．サイクルを含まないグラフを非巡回グラフと呼ぶ．グラフが非巡回であると明言しない場合，そのグラフはサイクルを含む巡回グラフであるかもしれない．

例 3.2　図 3.1 の $((v_1, v_2), (v_2, v_3), (v_3, v_4))$ は単純パスである．また，$((v_1, v_2), (v_2, v_6), (v_6, v_5), (v_5, v_1))$ はサイクルである．パス $((v_2, v_3), (v_3, v_4), (v_4, v_3))$ は単純でない．図 3.1 のグラフ G は巡回グラフである．　□

定義 3.7　グラフ $G = (V, E)$ の推移閉包はグラフ $G^+ = (V, E^+)$ で表す．ただし，G が非ヌルのパス $x \xrightarrow{+} y$ を含む時，かつその時に限り E^+ は辺 (x, y) を要素に含む．頂点 x の後者集合 (successor set) とは，集合 $Succ(x) := \{y | (x, y) \in E^+\}$ である．すなわち，非ヌルなパスを通って頂点 x から到達可能な全ての頂点からなる集合である．頂点 x の前者集合 (predecessor set) とは，集合 $Pred(x) := \{z | (z, x) \in E^+\}$ である．すなわち，非ヌルなパスを通って x へ到達可能な全ての頂点からなる集合である．頂点 x から隣接している頂点 $y \in AbjFrom(x)$ は x のすぐ次の後者である，そして，x へ隣接している頂点 $z \in AbjTo(x)$ は x のすぐ前の前者である．

例 3.3　図 3.1 のグラフ G の推移閉包を図 3.2 に示す．図を参照すると，頂点 v_1, v_2, v_5, v_6 の後者集合は V である．頂点 v_3, v_4, v_7, v_8 の後者集合は $\{v_3, v_4, v_8\}$ である．同様に，数個の頂点が共通の前者集合を持つ．　□

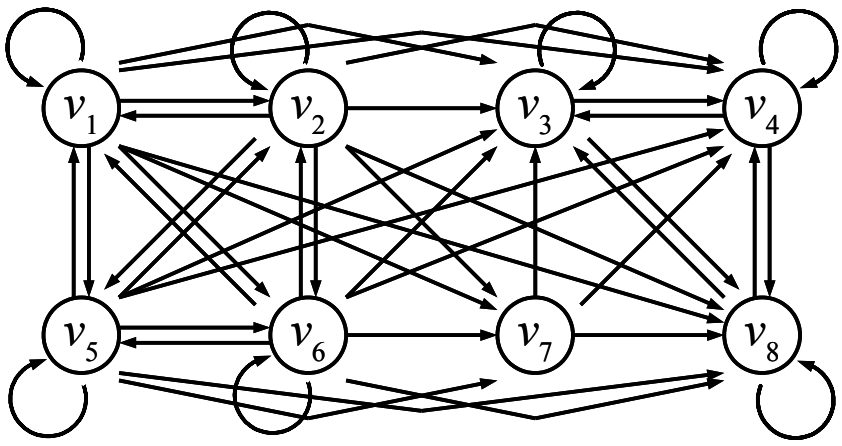


図 3.2　The transitive closure of graph G of Figure 3.1

定義 3.8　グラフ $G = (V, E)$ の反射推移閉包はグラフ $G^* = (V, E^*)$ であらわす．ただし，G がパス $x \xrightarrow{*} y$ を含む時，かつその時に限り E^* は辺 (z, y) を要素に含む．□

例 3.4　図 3.2 に提示されたグラフ G の推移閉包は，辺 (v_7, v_7) を挿入することによって，グラフ G の反射推移閉包になる．一般に $E^* = E^+ \cup I$ である．ただし，集合 I は $I := \{(x, x) | x \in V\}$ で定義され，反射閉包と呼ばれる．□

定義 3.9　G の 2 つの頂点 x と y について，G がパス $x \xrightarrow{*} y$ と，パス $y \xrightarrow{*} x$ を含む時，かつその時に限り x, y は強連結であると言う．極大で強連結な部分グラフは G の強連結成分と呼ばれる．頂点 x を含む強連結成分は $Comp(x)$ で表す．1 つの頂点だけから成る強連結成分は，自明な強連結成分と呼ばれる．グラフ G の強連結成分によって誘導されるグラフを凝縮グラフ $\bar{G} = (\bar{V}, \bar{E})$ で表す．$\bar{V}$ は，G の強連結成分の集合である．$\bar{E}$ は，E が辺 (x, y) を要素に含む時，かつその時に限り辺 (X, Y) を要素に含む．ただし，$X = Comp(x)$ ，$Y = Comp(y)$ である．

例 3.5　グラフ G の強連結成分は図 3.3(a) において点線で囲まれている．G の強連結成分によって誘導される凝縮グラフ $\bar{G}$ を，図 3.3(b) に示す．自己ループを取り除くなら凝縮グラフはいつも非巡回グラフであることに注意されたい．□

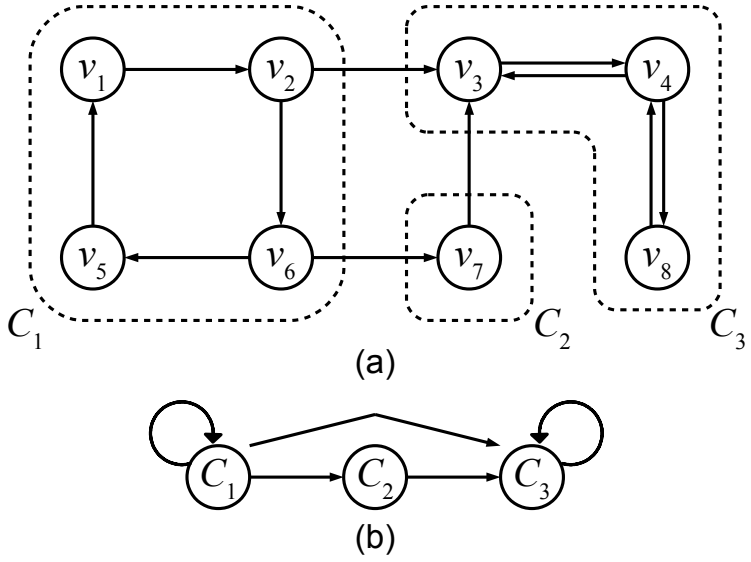


図 3.3　(a) The strong components of graph G of Figure 3.1 and (b) the condensation graph induced by the strong components of G.

定義 3.10　グラフ $G = (V, E)$ の頂点集合 V に関する位相順序とは，$(x, y) \in E$ であるとき，$x \leq y$ となる，V の任意の全順序 $\leq$ のことである．もし $x \leq y$ であれば，x が y より小さいと言い，y が x より大きいと言う．頂点集合 V の位相順序 $\leq$ の逆の関係は，V の反転位相順序と呼ばれる．頂点を位相順に並べ替えることをトポロジカルソートと言う．

任意の非巡回グラフには，少なくとも 1 つの位相順序が存在する．また，凝縮グラフ $\bar{G}$ が非巡回グラフに自己ループを増やしたグラフであることから，それぞれの凝縮グラフ $\bar{G}$ には位相順序が存在する．自己ループ以外のサイクルを持つどんなグラフも位相順序を持たない．

定義 3.11　グラフ $G = (V, E)$ の推移還元 $G_r = (V, E_r)$ は G と同じ推移閉包を持つが，できるだけ少ない辺で構成されたグラフである．推移還元の解に必ずしも一意性はない．

定義 3.12　(根つきの有向) 木 T は，以下の特性を満たす非巡回グラフである:

1. T には，辺の head にならないちょうど 1 つの頂点 r(根，root) をもつ．
2. 根以外の各頂点はちょうど 1 つの辺の head となる．
3. 根 r からそれぞれの頂点 x まで，唯一のパス $r \xrightarrow{*} x$ が存在する．

x が木の頂点であれば，部分木 T_x とは，$\{x\} \cup Succ(x)$ を頂点集合として持つ T の極大な部分グラフである．いくつかの木から成るグラフは森と呼ばれる．木 T(森 F) がグラフ G の全ての頂点を含むとき，木 T(森 F) を G のスパニング木と呼ぶ．

例 3.6　図 3.4 に図 3.1 のグラフ G のスパニング木の一つを示す．　□

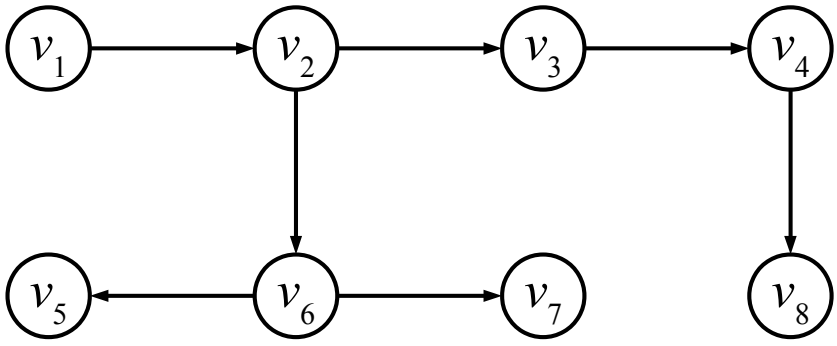


図 3.4　A spaning tree of graph G of Figure 3.1

定義 3.13　2 部グラフ $G = (V_1, V_2, E)$ とは，$V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$, $V_1 \times V_1 \cap E = \emptyset$, $V_2 \times V_2 \cap E = \emptyset$ を満たす，グラフ $G = (V, E)$ である．V_1, V_2 を独立集合，または独立頂点集合と呼ぶ.

例 3.7　図 3.1 のグラフ G は，$V_1 = \{v_1, v_3, v_6, v_8\}$，$V_2 = \{v_2, v_4, v_5, v_7\}$ を独立集合とした 2 部グラフである．□

辺集合 $E \subset V \times V$ は V 上の二項関係である．逆に，定義域 V のあらゆる二項関係 E は有向グラフとして定義できる．二項関係に関する諸条件でも推移閉包を定式化できる.

定義 3.14　二項関係 $E \subset V \times V$ の推移閉包 $E^+ \subset V \times V$ は，$E^+ = \bigcup_{n \geq 0} E^n$ で定義される．ただし，$E^0 = I$ で，$E^{n+1} = E^n \circ E = E \circ E^n$ である．ここで，I は反射閉包であり，$\circ$ は合成演算子である.

例 3.8　図 3.1 のグラフ $G = (V, E)$ を例に考える．単純パスを利用することによって，どんなグラフの推移閉包も計算できる．一般に，グラフで最も長い単純パスは高々 $|V|$ 個の辺を持つ．しかし，図 3.1 では，最も長い単純パスは 7 つの辺しか持たない．したがって，$E^+ = E \cup E^2 \cup E^3 \cup E^4 \cup E^5 \cup E^6 \cup E^7$ □

3.2　推移閉包問題

全頂点対推移閉包問題とは与えられた有向グラフ $G = (V, E)$ から推移閉包 $G^+ = (V, E^+)$ を計算する問題である.

単一始点推移閉包問題とは，与えられたグラフ $G = (V, E)$ から，V のある頂点 x の後者を計算する問題である．深さ優先探索や，幅優先探索などの単純なグラフ探索アルゴリズムでこの問題は解ける．頂点 x から検索を始めて，到達する各頂点を解集合 $Succ(x)$ に集めればよい.

複数始点推移閉包問題は，グラフ $G = (V, E)$ と始点集合 $X \in V$ が与えられ，X の各頂点の後者を計算する問題である．この問題は強い複数始点推移閉包問題と弱い複数始点推移閉包問題に分けられる．強い問題は，X の各頂点それぞれの後者集合を計算する問題である.

強い複数始点問題は，各開始頂点について単一始点問題を解けばよい．あるいは，最初に入力グラフの全頂点対推移閉包を計算し，次に適切な後者集合を選択すれば解ける.

弱い複数始点問題は X の各頂点の後者集合の和集合を計算する問題である．単一始点問題のように，単純なグラフ探索アルゴリズムで解ける.

3.3　グラフ理論的な covert channel 検出問題の定義

permission を object と subject の和集合 $V := O \cup S$ 上の二項関係と捉えれば，ACL からグラフが構成できる．

定義 3.15　ある ACL に対するアクセスグラフ $G_A = (V, E) = (O, S, E)$ を，頂点集合 $V := O \cup S$，辺集合 $E := \{(x, y) \in V \times V | xRw \vee xWy\}$ として定義する．明らかに $O \cup S = V$，$O \cap S = \emptyset$，$O \times O \cap E = \emptyset$，$S \times S \cap E = \emptyset$ であるから，アクセスグラフは object 集合と subject 集合を独立集合として持つ 2 部グラフである．

この定義に従えば，ACL で $xR^c y$ であることと，そのアクセスグラフ $G_A = (V, E)$ で，$(x, y) \notin E$ となることは同値である．ACL で $o_1 R s_1 \wedge s_1 W o_2 \wedge o_2 R s_2 \wedge \cdots \wedge o_n R s_n$ が成り立つことと，G_A 上にパス $((o_1, s_1), (s_1, o_2), (o_1, s_1), \cdots, (o_n, s_n),)$，すなわち $o_1 \xrightarrow{*} s_n$ が存在することは同値である．したがって，covert channel の定義もアクセスグラフを用いてグラフ理論的に定義できる．

定義 3.16　アクセスグラフ $G_A = (O, S, E)$ 上の covert channel とは, $(x, y) \in O \times S \cap E^+ - E$ を満たす x, y 間の任意のパス $x \xrightarrow{+} y$ である．本論文では区別する必要がないので，「ACL 上の covert channel」と「G_A 上の covert channel」を以降「covert channel」と呼ぶ．

アクセスグラフ $G_A = (V, E)$ の covert channel 検出 $G_A^{\#} = (V, E^{\#})$ とは，$E^{\#} := \{(x, y) \in O \times S | (x, y) \in E^+ - E\}$ となるグラフである．object x の隠れた後者 $CovertSucc(x)$ とは，集合 $CovertSucc(x) := \{y \in S | (x, y) \in E^{\#}\}$ である．ここで，$xR^{\#}y \Leftrightarrow (x, y) \in E^{\#} \Leftrightarrow y \in CovertSucc(x)$ が成り立つ．

また, covert channel 検出問題の定義もアクセスグラフを用いてグラフ理論的に定義できる．

定義 3.17　covert channel 検出問題とは，与えられた有向グラフ $G_A = (O, S, E)$ から covert channel 検出 $G_A^{\#} = (V, E^{\#})$ を計算する問題である．

covert channel 検出問題は $CovertSucc(O)$ を求める問題と言える．$CovertChannel(x)$ は，定義より $CovertChannel(x) = S \cap Succ(x) - AdjFrom(x)$ より求められる．アクセスグラフ $G_A = (O, S, E_A)$ が隣接リスト形式で与えられた場合，S と $AdjFrom(x)$ は入力時に得られる．ゆえに，計算すべきは後者集合 $Succ(x)$ だけである．すなわち，始点集合として O を選び，強い複数始点推移閉包問題を解くことで，covert channel 検出問題が解ける．これは，covert channel 検出問題が強い複数始点推移閉包問題の変形問題であることを意味する．

第 4 章

先行研究

4.1 先行研究

前節で触れた通り，covert channel 検出問題は推移閉包と密接に関連している．そこで，本節では，いくつかの先行研究を推移閉包の観点から再考察する．各先行研究の詳細に関しては，オリジナルの論文を参照されたい．

4.1.1 行列積による検出アルゴリズム

A を $G = (V, E)$ の $|V| \times |V|$ 隣接行列とする．A は長さ 1 のパスが存在するとき，かつその時に限り $A[i, j] = \mathbf{true}$ である．$A^2 = A \wedge A$ を計算すると，各成分は G が長さ 2 のパス $v_i \xrightarrow{+} v_j$ を含むとき時，かつその時に限り $A^2[i, j] = \mathbf{true}$ となる．より一般に，行列 A^k の各成分は，G が長さ k のパス $v_i \xrightarrow{*} v_j$ を含むとき時，かつその時に限り $A^k[i, j] = \mathbf{true}$ となる．必要なのは推移閉包を構成する単純パスのみであるから，推移閉包 G^+ の隣接行列 A^+ は A を用いて次式で書ける．

$$A^+ = \bigvee_{k=1}^{|V|} A^k \tag{4.1}$$

Furman は，簡単な帰納法を用いて $A^+ = \bigvee_{k=1}^{|V|} A^k = (I \vee A)^{\log_2(|V|-1)} A$ となることを示し，A^+ を $\mathcal{O}(\log_2 |V|)$ 回の行列積で計算できることを示した [14]．$|V| \times |V|$ の行列積が $\mathcal{O}(V^{\alpha})$ 時間かかると仮定すれば，Furman のアルゴリズムは $\mathcal{O}(V^{\alpha} \log V)$ 時間で推移閉包を計算する．Coppersmith と Winograd は，$\alpha \leq 2.376$ を示した [15].

ところで，アクセスグラフ $G_A = (O, S, E)$ は二部グラフである．$1 \leq i \leq |O| \Rightarrow v_i \in O \subset V$，$|O| + 1 \leq j \leq |O| + |S| \Rightarrow v_j \in S \subset V$ のように頂点を添え字付ける．すると，i, j が $1 \leq i, j \leq |O|$ か，もしくは $|O| + 1 \leq i, j \leq |O| + |S|$ であるとき，$A[i, j] = \mathbf{false}$ が成り立

つ．しからば，G_A に対応する隣接行列 A は次式の形で表せる．

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} 0 & A_{1,2} \\ A_{2,1} & 0 \end{pmatrix} \tag{4.2}$$

$A_{1,2}$ は $|O| \times |S|$ 区分行列である．アクセス権限を意識して言えば，$v_{|O|+j} \in S$ が $v_i \in O$ を read 可であるとき，かつその時に限り $A_{1,2}[i,j] = \mathbf{true}$ である．

$A_{2,1}$ は $|S| \times |O|$ 区分行列である．アクセス権限を意識して言えば，$v_{|O|+i} \in S$ が $v_j \in O$ を write 可であるとき，かつその時に限り $A_{2,1}[i,j] = \mathbf{true}$ である．

また，隣接行列 A のべき乗は偶数乗，奇数乗のとき，それぞれ次式となる．

$$A^{2k} = \begin{pmatrix} (A_{1,2}A_{2,1})^k & 0 \\ 0 & (A_{2,1}A_{1,2})^k \end{pmatrix} \tag{4.3}$$

$$A^{2k+1} = \begin{pmatrix} 0 & (A_{1,2}A_{2,1})^k A_{1,2} \\ A_{2,1}(A_{1,2}A_{2,1})^k & 0 \end{pmatrix} \tag{4.4}$$

これより，アクセスグラフの推移閉包 G_A^+ の隣接行列 A^+ を A と同じように区分けすれば，$A_{1,2}^+$ は次式で求まる．

$$\begin{aligned} A_{1,2}^+ &= A_{1,2} \vee (A_{1,2}A_{2,1})A_{1,2} \vee (A_{1,2}A_{2,1})^2 A_{1,2} \vee \cdots \vee (A_{1,2}A_{2,1})^{min(|O|,|S|)} A_{1,2} \\ &= (I \vee (A_{1,2}A_{2,1}) \vee (A_{1,2}A_{2,1})^2 \vee \cdots \vee (A_{1,2}A_{2,1})^{min(|O|,|S|)}) A_{1,2} \\ &= (I \vee (A_{1,2}A_{2,1}))^{\log_2(min(|O|,|S|)-1)} A_{1,2} \end{aligned} \tag{4.5}$$

$A_{1,2}^+$ は $S \cap Succ(O)$ を表現したものである．ここで，$|O| \times |S|$ 行列 $A_{1,2}^{\#}$ の各成分を次式で定義する．

$$A_{1,2}^{\#}[i,j] := \neg A_{2,1}[i,j] \wedge A_{2,1}^+[i,j] \tag{4.6}$$

これは $CovertSucc(O)$ を表現したものであるから，$A_{1,2}^{\#}$ は covert channel 検出問題の解となる．

単位行列 I との論理和 $I \vee A_{2,1}$ は，A の対角成分を全て **true** にするだけなので $\Theta(O)$ 時間で求まる．$l \times m$ 行列と $m \times n$ 行列の積が $\mathcal{O}(M(l,m,n))$ で計算できると仮定すれば，$R_{1,2}^+$ は $\mathcal{O}(O^\alpha \log min(O,S) + M(O,S,O) + M(O,O,S))$ 時間で求まる．各要素ごとに式 (4.6) を求めればよいので，$A_{1,2}^+$ から $A_{1,2}^{\#}$ は $\mathcal{O}(OS)$ 時間で求まる．結局，この covert channel 検出アルゴリズムの最悪計算量は $\mathcal{O}(O^\alpha \log min(O,S) + M(O,S,O) + M(O,O,S))$ である．次式も言えるので，同様にして最悪計算量は $\mathcal{O}(S^\alpha \log min(O,S) + M(S,O,S) + M(S,S,O))$ でもある．

$$A_{1,2}^+ = A_{1,2}(I \vee (A_{2,1}A_{1,2}) \vee (A_{2,1}A_{1,2})^2 \vee \cdots \vee (A_{2,1}A_{1,2})^{min(|O|,|S|)}) \tag{4.7}$$

$|O| = |S|$ と仮定すれば，最悪計算量は $\mathcal{O}(O^\alpha \log O)$ である．

この covert channel 検出アルゴリズムは文献 [1] で示されている．

4.1.2　幅優先探索を用いた検出アルゴリズム

幅優先探索 (Breadth first search;BFS[12, 13]) はグラフに関するアルゴリズムの中で最も基本的なアルゴリズムの 1 つである．幅優先探索は根 r から到達可能な全ての頂点を探索できるので，単一始点推移閉包問題を解ける．具体的には図 4.1 のようにしてグラフ $G = (V, E)$ と始点 x から $Succ(x)$ を計算する．

図 4.1 では，探索の順序をキュー **Q** によって管理している．VISITED(y) は y が探索済みか否かを表す．したがって，x 以外の任意の頂点 y で，VISITED(y) の初期値は **false** である．これによって同じ頂点を 2 度探索することを防いでいる．4 行目の実行時点で未探索の頂点でなければ，7 行目で後者集合 $Succ(x)$ に加わることはできない．同じ頂点を 2 度探索することはないので，後者集合に同じ頂点が 2 度加わることはない．x は最初から探索済みであるから，x が後者集合の要素になることはない．すなわち，G 上に非ヌルなパス $x \xrightarrow{+} x$ が存在しても，図 4.1 のアルゴリズムにおいて，$x \notin Succ(x)$ となる．

幅優先探索の最悪時間計算量は $\mathcal{O}(V + E)$ である．7 行目の後者集合 $Succ(s)$ の計算では，同じ頂点が 2 度加わることがないので，要素同士の重複の有無を計算する必要がない．$Succ(s)$ の要素をソートせずに配列で保持する場合，5 行目の計算は $\mathcal{O}(1)$ で済む．このとき，図 4.1 のアルゴリズムの最悪時間計算量は $\mathcal{O}(V + E)$ である．

```
1  Succ(x) := ∅; VISITED(x) := true; Q:= {x}
2  while Q≠ ∅ do
3      y :=DEQUEUE(Q)
4      for each vertex z ∈ AbjFrom(y) such that VISITED(z) = false do
5          VISITED(z) := true
6          ENQUEUE(Q, y)
7          Succ(x) := Succ(x) ∪ {z}
8      end for
9  end while
```

図 4.1　A computation of $Succ(x)$ using BFS.

幅優先探索を用いて $Succ(O)$ を求めることで，covert channel 検出問題が解ける．そのアルゴリズムは図 4.2 のようになる．

$CovertSucc(x) \cap AbjFrom(x) = \emptyset$ となるように，3 行目で $AbjFrom(x)$ の要素である頂点を探索済みにしている．10 行目では $CovertSucc(x) \subset S$ を満たすように IF 文を用いている．図 4.2 は，図 4.1 のアルゴリズムを $|O|$ 回繰り返しているにすぎないので，その最悪時間計算量は $\mathcal{O}(OV + OE) = \mathcal{O}(O^2 + OS + OE)$ である．

```
1   for each object x ∈ O do
2       for each vertex y ∈ V do VISITED(y) := false
3       for each vertex y ∈ AbjFrom(x) do VISITED(y) := true
4       CovertSucc(x) := ∅; VISITED(x) := true; Q:= AbjFrom(x)
5       while Q≠ ∅
6           y :=DEQUEUE(Q)
7           for each vertex z ∈ AbjFrom(y) such that VISITED(y) = false do
8               VISITED(z) := true
9               ENQUEUE(Q, z)
10              if z ∈ S then CovertSucc(x) := CovertSucc(x) ∪ {z}
11          end for
12      end while
13  end for
```

図 4.2　A covert channel detection using BFS.

グラフ $G = (V, E)$ で，任意の始点 x から幅優先探索した場合，探索時に部分木が構成される．これを幅優先探索木 $T_{BFS}(x) = (V'_{BFS}(x), E'_{BFS}(x))$ と呼ぶ [12]．部分木であるから，$V'_{BFS} = \{x\} \cup Succ(x) \subset V$ が成り立つ．$T_{BFS}(O)$ を求めるアルゴリズムは図 4.3 のようになる．最悪時間計算量は $\mathcal{O}(OV + OE) = \mathcal{O}(O^2 + OS + OE)$ である．

```
1   for each object x ∈ O do
2       for each vertex y ∈ V do VISITED(y) := false
3       V'_BFS(x) := {x}; E'_BFS(x) := ∅; VISITED(x) := true; Q:= {x}
4       while Q≠ ∅
5           y :=DEQUEUE(Q)
6           for each vertex z ∈ AbjFrom(y) such that VISITED(z) = false do
7               VISITED(z) := true
8               ENQUEUE(Q, z)
9               V'_BFS(x) := V'_BFS(x) ∪ {z}
10              E'_BFS(x) := E'_BFS(x) ∪ {(y, z)}
11          end for
12      end while
13  end for
```

図 4.3　A computation of breadth-first tree.

$T_{BFS}(x)$ は木であるから，任意の $y \in V'_{BFS}(x) - \{x\}$ に対して，非ヌルなパス $x \xrightarrow{+} y$ を一つだけ持っている．このとき，パスの長さはグラフ $G = (V, E)$ 上で最短のものである．ゆえに，幅優先探索木は始点から各後者へのある最短パスの集まりと考えられる．ここで，$T_{BFS}(x)$ から個々の最短 covert channel を全て出力するアルゴリズムを図 4.4 に示す．

このアルゴリズムは深さ優先探索 (Depth first search;DFS) を用いている [12, 13].

アクセスグラフの任意のパスは object と subject が交互に並ぶ．また，covert channel は object を始点として subject を終点としたパスである．ゆえに covert channel は奇数長のパスである．ただし，長さ 1 のパスは covert channel でない．**STACK** には始点から最後に Pushu された頂点 TOP(**STACK**) までのパスが入っている．そこで **STACK** の長さをもとに 5 行目で covert channel であるか否かを判定し，covert channel である場合，**STACK** の中身を出力している．

図 4.4 において，$T_{BFS}(x)$ は隣接リスト表現で表される．8 行目で隣接リスト $AbjFrom(y)$ をスタックとして扱っている．これは，同じ辺を 2 度辿ることを防ぐためである．

深さ優先探索のアルゴリズムの最悪時間計算量は $\mathcal{O}(V'_{BFS}(x) + E'_{BFS}(x))$ である．木の辺数は頂点数を越えない．すなわち，$|E'_{BFS}(x)| < |V'_{BFS}(x)| = 1 + |Succ(x)| \leq |V|$ が成り立つ．最短路を構成する頂点は最大でも $2\min(O, S)$ 個であるから，5 行目の print(**STACK**) は $\mathcal{O}(\min(O, S))$ の計算量で実行される．図 4.4 の最悪時間計算量は $\mathcal{O}(V \min(O, S))$ である．$|V| = |O| + |S| \leq 2\max(|O|, |S|)$ より，$|V|\min(|O|, |S|) < 2|O||S|$ となるから，$\mathcal{O}(V \min(O, S)) = \mathcal{O}(OS)$.

図 4.4 を各 object $x \in O$ の幅優先探索木 $T_{BFS}(x)$ に行う場合，最悪時間計算量は $\mathcal{O}(O^2S)$ となる．

```
1   STACK:= {x}
2   while STACK≠ ∅
3       y :=TOP(STACK)
4       if AbjFrom(y) = ∅ then
5           if HEIGHT(STACK) is not 1 and is odd then print(STACK)
6           POP(STACK)
7       else
8           PUSHU(STACK, POP(AbjFrom(y)))
9       end if
10  end while
```

図 4.4　An output algorithm of covert channle.

ある object$x \in O$，subject $y \in S$ 間に最短 covert channel が 2 つ以上存在しても，図 4.3 と図 4.4 を組み合わせたアルゴリズムでは，1 つだけしか最短 covert channel が出力されな

い．図 4.5 のアルゴリズムを用いて $G'_{BFS}(x) = (V'_{BFS}(x), E'_{BFS}(x))$ を求め，$G'_{BFS}(x)$ に対して図 4.4 を実行すれば全ての最短 covert channel が出力される．このアルゴリズムは文献 [16] に示されている．ただし，このアルゴリズムの最悪計算量は現実的でない．このことを図 4.6 を用いて示す．

図 4.6 のグラフはある非負整数 m に対して，$|O| = |S| = m^2$ を満たすアクセスグラフである．このグラフでは，m 個の object からなる列と，m 個の subject からなる列が交互に $2m$ 列並んでいる．列 k の各 object から列 k の各 subject まで辺が伸びている．また，列 k の各 subject から列 $k+1$ の各 object まで辺が伸びている．さらに，列 m の各 subject から列 1 の各 object まで辺が伸びている．したがって，全ての頂点が，全ての頂点を後者に持つ．これは，Jakobsou の論文 [17] で例示されたグラフを変形したものである．

図 4.5 のアルゴリズムによって $G'_{BFS}(o_1)$ を計算すると図 4.7 となる．o_1 から s_{m^2} への最短 covert channel は $\Omega(m^{2(m-1)}) = \Omega\left(O^{\sqrt{O}}\right)$ 本存在する．しかるに，図 4.6 のグラフに対して，o_1 から s_{m^2} への最短 covert channel を全て出力しようとするならば，どのようなアルゴリズムを用いても，$\Omega\left(O^{\sqrt{O}}\right)$ を計算量の下界として持つ．

```
1   for each object x ∈ O do
2       V'(x) := {x}; E'(x) := ∅; Q:= {x}
3       for each vertex y ∈ V do Depth(y) := ∞
4       Depth(x) := 0
5       while Q≠ ∅
6           y :=DEQUEUE(Q)
7           for each vertex z ∈ AbjFrom(y) such that Depth(z) = ∞ do
8               Depth(z) := Depth(y) + 1
9               ENQUEUE(Q, z)
10              V'(x) := V'(x) ∪ {y}
11          end for
12      end while
13      for each edge (y, z) ∈ E such that Depth(y) + 1 = Depth(z) do
14          E'(x) := E'(x) ∪ {(y, z)}
15      end for
16  end for
```

図 4.5

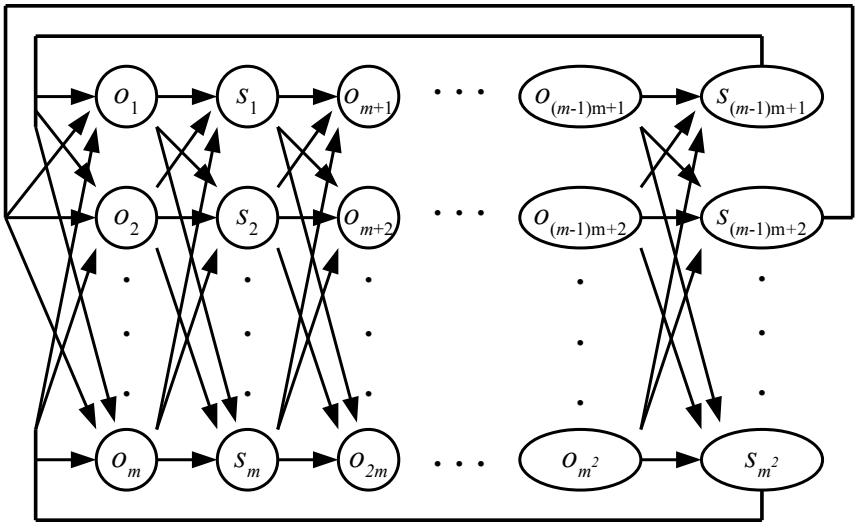


図 4.6　An example graph based on Jakobson's paper[17].

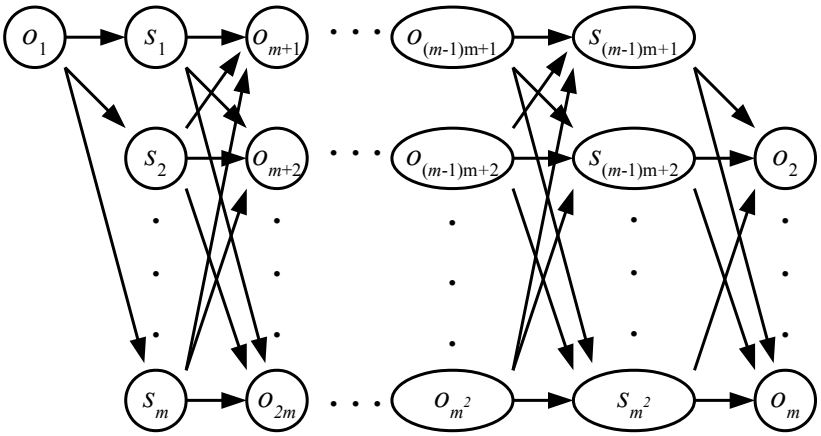


図 4.7　$G'(o_1)$ of a graph of fig.4.7.

第 5 章

推移閉包の応用

5.1 Warshall のアルゴリズム

最もよく知られている推移閉包アルゴリズムは Warshall のアルゴリズムである [18]．このアルゴリズムは多くの教科書に記載されている．

Warshall のアルゴリズムによる考えは以下の通りである．ある集合 W の元を中間頂点にもつパス $x \xrightarrow{*} y$ と $y \xrightarrow{*} z$ をグラフが持つならば，$W \cup \{y\}$ の元を中間頂点にもつパス $x \xrightarrow{+} z$ を同様に持つ．Warshall のアルゴリズムは，k を 1 から $|V|$ まで動かし，集合 $\{v_1, \cdots, v_k\}$ の元を中間頂点にもつパスを繰返し走査する．

アルゴリズムを図 5.1 に示す．アルゴリズムの入力は，グラフ G を表す隣接行列 A である．アルゴリズムは A をグラフ G^+ を表す隣接行列 A^+ に変換する．Warshall のアルゴリズムの最悪計算時間は $\mathcal{O}(V^3)$ である．また，最良計算時間は $\Omega(V^2)$ である．これは，疎な入力グラフに対して，Warshall のアルゴリズムが効果的でないことを意味する．

```
1  for k := 1 to |V| do
2    for i := 1 to |V| do
3      if i ≠ k ∧ A[i,k] then
4        for j := 1 to |V| do A[i,j] := A[i,j] ∨ A[k,j]
```

図 5.1　Warshall's algorithm

5.2　強連結成分分解に基づく推移閉包アルゴリズム

任意のグラフ $G = (V, E)$ のある強連結成分 C に属する 2 頂点 $x, y \in C$ について，$Succ(x) = Succ(y)$ が成り立つ．なぜならば，$x, y \in C$ より，グラフには $x \xrightarrow{*} y$，$y \xrightarrow{*} x$ となるパスが存在する．x の任意の後者 $a \in Succ(x)$ に対して，グラフ上にパス $x \xrightarrow{*} a$ が存在する．x を中間頂点とすれば，明らかに $y \xrightarrow{*} a$ なるパスが存在する．ゆえに $Succ(x) \subset Succ(y)$ である．同様にして，$Succ(y) \subset Succ(x)$ である．しかるに $Succ(x) = Succ(y)$ で，強連結成分内のすべての頂点は同じ後者集合を持つ．Purdom はこの事実に基づいて 4 つのステップからなる推移閉包アルゴリズムを示した [19].

1. 入力グラフ G を強連結成分分解し，凝縮グラフ $\bar{G}$ を構築する．
2. $\bar{G}$ の頂点，すなわち G の強連結成分をトポロジカルソートする．
3. $\bar{G}$ の推移閉包を計算する．$\bar{G}$ の頂点 C の後者集合を形成するには，C から隣接している頂点とそれらの後者集合の和集合を求めればよい．後者集合の計算は，$\bar{G}$ の位相的に最大の頂点から始め，最小の頂点まで行う．トポロジカルソートは，C から隣接している頂点の後者集合が C の後者集合の前に計算し終わっていることを保証する．
4. $\bar{G}$ の推移閉包を G 上の推移閉包に展開する．もし，頂点 x が成分 X に，頂点 y が成分 Y に含まれているならば，$\bar{G}$ の推移閉包で Y が $Succ(X)$ に含まれている時，かつその時に限り y を $Succ(x)$ に挿入する．

アルゴリズムの詳細は，7 ページにおよぶほど複雑である．

Purdom は，最初のステップ 1-3 を結合できると主張している．実際，Eve と Kurki-Suonio[20]，Eber[21]，Schmitz[22] 等は，強連結成分分解を使用するが，凝縮グラフは発生させずに推移閉包を計算するアルゴリズムを示した．これら全てのアルゴリズムが Tarjan の強連結成分分解アルゴリズムに基づいている [23].

5.3　行列積に基づく推移閉包アルゴリズム

A を $G = (V, E)$ の $|V| \times |V|$ 隣接行列とする．A は長さ 1 のパスが存在するとき，かつその時に限り $A[i, j] = \mathbf{true}$ である．$A^2 = A \wedge A$ を計算すると，各成分は G が長さ 2 のパス $v_i \xrightarrow{+} v_j$ を含むとき時，かつその時に限り $A^2[i, j] = \mathbf{true}$ となる．より一般に，行列 A^k の各成分は，G が長さ k のパス $v_i \xrightarrow{*} v_j$ を含むとき時，かつその時に限り $A^k[i, j] = \mathbf{true}$ となる．必要なのは推移閉包を構成する単純パスのみであるから，推移閉包 G^+ の隣接行列 A^+ は

A を用いて次式で書ける．

$$A^+ = \bigvee_{k=1}^{|V|} A^k \tag{5.1}$$

Furman は，簡単な帰納法を用いて $A^+ = \bigvee_{k=1}^{|V|} A^k = (I \vee A)^{|V|-1} A$ となることを示し，A^+ を $\mathcal{O}(\log_2 |V|)$ 回の行列積で計算できることを示した [14]．$|V| \times |V|$ の行列積が $\mathcal{O}(V^{\alpha})$ 時間かかると仮定すれば，Furman のアルゴリズムは $\mathcal{O}(V^{\alpha} \log V)$ 時間で推移閉包を計算する．Coppersmith と Winograd は，$\alpha \leq 2.376$ を示した [15]．

このように行列積と推移閉包計算は密接な関係があるが，その厳密な関係は Munro[24], Fischer と Meyer[25] によって発見された．彼らは行列積と推移閉包計算が計算上同等であるという以下の結果を証明した．

定理 5.1　$M(2n) \geq 4M(n), M(3n) \leq 27M(n)$ を満たす関数を $M(n)$ と置く．このとき，任意の 2 つの $n \times n$ 行列の積を $\mathcal{O}(M(n))$ 時間で計算可能な時，かつその時に限り推移閉包は $\mathcal{O}(M(n))$ 時間で計算可能である．

したがって，反射推移閉包は $\mathcal{O}(n^{\alpha})$ で計算できる．ただし，$\alpha \leq 2.376$.

Munro は上記の定理を明白にする次のアルゴリズムを示した [24]．

1. 入力グラフ $G = (V, E)$ から凝縮グラフ $\bar{G} = (\bar{V}, \bar{E})$ を構築する．
2. 強連結成分をトポロジカルソートする．
3. $\bar{G}$ に対応する隣接行列 $\bar{A}$ を構築する．このとき $\bar{G}$ のトポロジカルソートによって，行列は上三角行列となる．
4. 以下の恒等式を再帰的に使用し，$\bar{A}$ の推移閉包を計算する．ただし，$\bar{A}$(および $\bar{A}_{11}$ と , $\bar{A}_{22}$) は上三角行列である．

$$\bar{A}^* = \begin{pmatrix} \bar{A}_{1,1} & \bar{A}_{1,2} \\ 0 & \bar{A}_{2,2} \end{pmatrix}^* = \begin{pmatrix} \bar{A}^*_{1,1} & \bar{A}^*_{1,1}\bar{A}_{1,2}\bar{A}^*_{2,2} \\ 0 & \bar{A}^*_{2,2} \end{pmatrix} \tag{5.2}$$

5. $\bar{G}$ の推移閉包を G 上の推移閉包に展開する

step 4 では，$\bar{A}_{1,1}, \bar{A}_{2,2}, \bar{A}_{1,2}$ の次数が 2^i であると仮定する．これは，$\bar{A}$ の次数が 2^{i+1} であることも意味している．ゼロパディングすれば次数 2^i の行列を構築できるため，任意の行列に対して適用できる．$\bar{A}_{1,1}$ と $\bar{A}_{2,2}$ は，$\bar{G}$ の部分グラフ G'_1 と G'_2 を表す．また，$\bar{A}_{1,2}$ はこれら部分グラフ間の接続部を表す．上記 $\bar{A}^*_{1,1}\bar{A}_{1,2}\bar{A}^*_{2,2}$ の項は，次のように解釈できる．v を部分グラフ S_1 の頂点，u を部分グラフ S_2 の頂点とする．このとき，パス $v \xrightarrow{*} x$ は，頂点が全て S_1 内のパス $v \xrightarrow{*} x$ と，頂点が全て S_2 内のパス $y \xrightarrow{*} v$ を辺 (x, y) で連結することで構築できる．

ステップ 1-3 は $\mathcal{O}(V^2)$ 時間かかる．Munro は，ステップ 4 が $\mathcal{O}(\bar{V}^{\alpha})$ の操作を必要とする

ことを示した．ステップ 5 は $\mathcal{O}(V^2)$ 時間かかる．$|\bar{V}^\alpha| \leq |V^\alpha|$ より，総実行における最悪計算量は $\mathcal{O}(\bar{V}^\alpha) = \mathcal{O}(V^\alpha)$ となる．

ステップ 5 の代わりに図**??**を用いれば，covert channel 検出アルゴリズムに変換できる．この変換を行なっても，アルゴリズムの最悪計算量は変わらない．

行列積に基づく推移閉包アルゴリズムは計算時間の漸近に関して良い上限を持っている．しかし，恒常的要因が多く，疎グラフに対して非効率的である隣接行列表現を使用するため，実用的ではない．

5.4　表現と動的更新

多くの推移閉包アルゴリズムが，閉包を表すためにリストや行列を用いる．しかしながら，いくつかのアルゴリズムは，計算を高速化するために特別な表現を用いる．非巡回グラフに関する Simon の推移閉包アルゴリズムは，最悪計算時間を小さくするため，結合演算を高速化する鎖分解を用いている [29]．Dar と Jagadish[30] および Jakobson[31, 32] によるハイブリッドアルゴリズムは，不要な後者集合演算を避けることを目的とし，後者木表現を用いる．covert channel 検出問題は推移閉包問題の応用なので，これらの表現は covert channel 検出においても重要である．本節では，推移閉包の他の特別な表現を紹介する．

メモリ上の動的更新

茨木と加藤は，入力グラフに辺を挿入する際，もしくは削除する際の推移閉包を動的に維持するアルゴリズムに関して，影響力の高い論文を示した [28]．アルゴリズムは特別なデータ構造を用いない．推移閉包は隣接行列で表される．辺の挿入に対する閉包更新アルゴリズムにおいて，1 つの辺の挿入を処理する際の最悪計算量は $\mathcal{O}(V^2)$ を必要とする．ただし，辺 E の挿入を処理する最悪計算量は $\mathcal{O}(VE^+)$ である．辺 1 本の挿入か，もしくは削除の過程における最悪計算量は，数本の挿入と削除のならし計算量を超える．これは推移閉包を動的に維持するアルゴリズムの典型的な特性である．茨木と加藤の辺の削除に対する閉包更新アルゴリズムは，E 本の辺を削除，処理するために最悪計算量 $\mathcal{O}((V+E)E^+)$ を必要とする．

Italiano は，専門的なデータ構造が使用されるならば，辺の挿入に対する更新をより速くできることを示した [**?**]．系統的な後者集合を含むデータ構造には，木を表すポインターの $|V| \times |V|$ 行列とパスの木がある．このデータ構造であれば，連続した辺 e の挿入に対する閉包更新が $\mathcal{O}(VE)$ の最悪計算量で可能となる．また，パスの長さに比例した時間で 2 つの頂点をパスで接続することも可能である．Buchsbaum 他 [35] は，グラフの辺をラベル付けすることで Italiano の結果を拡張した．別紙で Italiano は，使用するデータ構造を変更したバージョンの，辺 e の削除に対する非巡回グラフの閉包を最悪計算量 $\mathcal{O}(VE)$ で更新できるアルゴリズムを示した [34].

La Peutré と van Leeuwen は，辺の挿入か削除に対するの推移還元とグラフの推移閉包の両方の動的な維持を研究した [36]．彼らは Italiano のアルゴリズムに類似したアルゴリズムとデータ構造を示した．辺の挿入を処理するアルゴリズムは，Italiano の挿入アルゴリズムと同じ最悪計算量 $\mathcal{O}(VE)$ を持つ．入力が非巡回であるとき，辺の削除を処理するアルゴリズムは，Italiano の削除アルゴリズムと同じ上界の最悪計算量をもつ．また，Italiano のアルゴリズムと異なって，La Peutré と van Leeuwen のアルゴリズムは巡回グラフでの辺の削除を処理できる．ただし，$|E|$ 本の削除処理の最悪計算量は $\mathcal{O}(VE)$ よりわずかに大きい．

Yellin は，Italiano および La Peutré と van Leeuwen と同様のアルゴリズムとデータ構造を示した [37]．Yellin のアルゴリズムも，最悪計算量 $\mathcal{O}(VE)$ の上界を持つ．しかしながら Yellin は，グラフの最大出次数 d によって，アルゴリズムが別の最悪計算量 $\mathcal{O}(dE^+)$ を上界に持つことを示した．これはグラフの出次数による最適な上界である．

データベース上のグラフ探索を補助する特別な表現

入力グラフがディスクにあるとき，入力の走査中に発生するページングは計算のボトルネックとなる．ディスクに格納されるグラフの順序は性能に影響する．ランダムな順序が原因で，ディスクの読み書きヘッドはランダムなスキップを行う．グラフがメモリに収まらないなら，各ページはディスクからメモリまで何回も動かされるかもしれない．

Larson と Deshpande は，非巡回グラフの反復走査を補助するための簡単なファイル構造を示した [38]．頂点と辺は，パス計算を可能にするためラベルを持っているかもしれない．メインファイルとインデックスファイル，2 個のファイルから表現は構成される．メインファイルは各頂点あたり 1 件分のデータが入っている．1 件分のデータには，頂点の頂点のラベル，後者，および前者のデータが入っている．後者と前者は頂点ラベルの組として表される．各頂点のデータは，トポロジカルソートされ，B 木に保存される．トポロジカルソートによって，メインファイルを一回スキャンするだけで全ての後者集合 (もしくは前者集合) を発見することができる．インデックスファイルは頂点名から位相番号への写像を行う．キーは B 木で頂点データの記録箇所を特定するために用いる Larson と Deshpande は頂点か辺が挿入されるか，削除されるか，または変更されるとき，ファイル構造を更新するためのアルゴリズムを示した．辺を加えることはトポロジカルソートを無効にし，メインファイルにおけるいくつかの記録の配置換えを強制する可能性のある唯一の操作である．他の保守作業は簡単で，なおかつ高速である．

Larson と Deshpande が示したグラフ走査アルゴリズムは，頂点の訪問順序を管理するため優先度付きキューを用いる．頂点の出次数が大きいならば，優先度付きキューはメモリに収まりきらないほど大きくなるかもしれない．また，優先度付きキューは多くのディスク I/O を引き起こすかもしれない．Hua 他 [56] は，ファイル上を一度スキャンするだけでグラフ走査を可能にする，接続インデックスと呼ばれる同様のファイル構造を示した．接続インデックス

の走査は，ディスク I/O において優先度付きキューよりはるかに効率的な，FIFO キューを用いて制御される．

Agrawal と Kiernan は，入力に巡回グラフを許容することを除いて，Larson と Deshpande，および Hua 他のものに類似した，走査インデックスと呼ばれるファイル構造を示した [39]．走査インデックスは，入力が非巡回グラフであるならトポロジカルソートで維持され，入力が巡回グラフであるなら緩和トポロジカルソートで維持される．Agrawal と Kiernan は，走査インデックスを作成するためのアルゴリズムと，入力グラフが変化した際にインデックスを維持するためのアルゴリズムを示した．このとき，入力グラフが変化した際，走査インデックス上をただ一度スキャンするだけで，どのように入力グラフを走査できるかも示している．

データベース上で推移閉包を具体化する特別な表現

Jagadish は，推移閉包を具体化する方法として鎖分解を研究した [40]．Simon の表現からの違いは，Jagadish が後者集合 $Succ$ を組 (C_i, v_i) のリストで表現したことである．ただし，C_i は鎖であり，v_i は $Succ \cap C_i$ の位相的に最小な要素である．Simon は k 個の要素からなるベクトル $[v_1, v_2, \cdots, v_k]$ として $Succ$ を表現した．したがって，Jagadish の表現は二つの後者集合の和集合を取るのに $\Omega(k)$ の計算量で済む．しかし，後者集合への頂点の挿入と，ある頂点が後者集合に存在するかの確認における最悪計算量も $\Omega(k)$ である．

Jagadish は，非巡回グラフの最適な鎖分解を見つける問題が，最小フロー問題と一致することを示した．彼は，分解計算に関するいくつかのヒューリスティックを示し，それらを実験によって比較した．実験はトポロジカルソートに基づいた分解法が，最も少ない鎖で分解できることを示した．これはまさに Simon が提案した方法 [29] であるが，Jagadish は Simon の仕事について直接言及しなかった．また，Jagadish は入力グラフが変更された際，どのように鎖分解と後者集合を維持するか示した．

Agrawal 等は，非巡回グラフの推移閉包を具体化する，コンパクトな別の表現を示した [26]．この表現は整数の区間に基づいている．後者集合は区間 $\{I_1, I_2, \cdots, I_m\}$ を蓄積している．このとき区間 $I = (i, j)$ は整数 $i, i+1, \cdots, j$ を示す．通常，ほんのいくつかの区間を用いることで後者集合 $Succ$ を表せる．したがって，2 つの後者集合の和集合，頂点の挿入，ある頂点が含むか否かの確認は，通常後者集合リスト表現よりはるかに高速である．しかしながら，これらの操作の最悪計算量の上界はリスト表現と同じである．いくつかの方法で頂点に添字付けられるが，異なった添字付けには，後者集合を表す異なった区間が必要となる．Agrawal 等は，一定の条件の下で最適な付番を計算するためのアルゴリズムを示した．また，入力グラフが変更される際，どのように表現を維持するかを示した．

鎖分解や区間表現にパス情報を関連付けることはできない．グラフの頂点数に対して，存在しうるパスの数が数指数関数的に増大するため，パス情報の完全な具体化は実現不可能である．

いくつかの論文では, パス情報の部分的な具体化が研究されている．Agrawal と Jagadish は 2 頂点間のパス情報を部分的に具体化する表現を示した [27]．表現は，組 (y, z, L) の形をした集合を各頂点 x に関連づける．ここで y は，x から到達可能な頂点である．$z \in AbjFrom(x)$ は y へ通じるパスに始めに現れる頂点である．そして，L はパスに関連づけられたラベルである．Agrawal と Jagadish は，入力グラフが変更される際，どのように表現を作成し，維持するか，そして，表現を使用することでどうすれば効率的にパスを計算できるか示した．

Guh 等は, 非巡回グラフで 2 頂点間の経路を部分的に具体化する別の表現を示した [41, 42]．その表現は $(x, y, G_{x,y}, g_{x,y})$ の形をした組に基づいている．ここで，x と y はパス $v \xrightarrow{*} w$ によって接続された頂点である．$G_{x,y}$ は x, y 間の異なるパスの本数を表す．そして $g_{x,y}$ は，(x, y) が入力グラフの辺である時，かつその時に限り **ture** となる．各頂点 v に関連づけられているものは，すべて v を (パスの始点か終点として) 含む組である．Guh 等は，この表現を作成し，管理するための逐次アルゴリズムと並列アルゴリズムを示した．

Agrawal と Jagadish の表現 [27] と Guh 等による表現 [41, 42] の両方の欠点は，その巨大なデータサイズである．Agrawal と Jagadish による表現は，最悪の場合 $\Omega(dn^2)$ の組を必要とする．ここで，d は全頂点中最大の出次数である．このような最悪空間計算量は，この表現における例外と言うより標準であろう．Guh 等による表現は，最悪の場合 $\Omega(n^2)$ の組を必要とするが，n が大きいとき，これはあまりにも多すぎる．したがって頂点数が大きいときは，パス情報の部分的な具体化さえ実用的ではない．

第 6 章

提案アルゴリズム

推移閉包アルゴリズムを用いた次のような covert channel 検出アルゴリズムを提案する．

1. ACL からアクセスグラフ G_A を構築する．
2. G_A から凝縮グラフ $\bar{G}_A$ の推移閉包 $\bar{G}_A^+$ を計算する．
3. $\bar{G}_A^+$ と G_A から covert channel 検出 $G_A^{\#}$ を計算する．

$G_A = (O, S, E) = (V, E)$ とする．隣接リスト形式でグラフを表現する場合，step1 の計算量は $\Theta(E)$ である．Esko の STACK_TC[4] を用いた場合，step2 の最悪計算量は $\mathcal{O}(V\bar{V} + VE_r + \mu)$ である．step3 は $\mathcal{O}(OS)$ で求まる．したがって，提案アルゴリズムの最悪計算量は $\mathcal{O}(OS + VE_r + V\bar{V} + \mu)$ である．ただし，$\mu \leq V\bar{V}$ である．

以下，本章では各ステップの詳細を述べる．

Step.1　G_A の構築

ACL からアクセスグラフ G_A を構築する方法は単純である．ACL から permission を読み込み，xRy であれば，$AbjFrom(x) := AbjFrom(x) \cup \{y\}$ を実行し，xWy であれば，$AbjFrom(x) := AbjFrom(x) \cup \{y\}$ を実行すれば良い．アクセスグラフ $G_A = (V, E)$ の辺数は read 権限，write 権限の総数と同数であるから，ステップ 1 の計算量は $\Theta(E)$ である．

Step.2　STACK_TC

STACK_TC[4] は Tarjan のアルゴリズム [23] を用いた入力グラフの強連結成分分解に基づく推移閉包アルゴリズムである．

STACK_TC は強連結成分 C が検出されるまで C の後者集合 $Succ(C)$ を構築しない．すなわち，任意のある強連結成分に対して生成される後者集合はただ 1 つであり，構成途中の後者集合は高々 1 つである．$Succ(C)$ を構築する際に必要となる，C から隣接した強連結成分を

収集する．これにより強連結成分 C の検出時，入力グラフを 2 回スキャンすることを避ける．この目的のために使用される補助スタックは，Tarjan のアルゴリズムの頂点用のスタックに類似している．強連結成分 C の検出時，いくつかの強連結成分がスタックに格納されている．強連結成分 C の後者集合 $Succ(C)$ は，スタックの強連結成分とその後者集合の和集合を取ることによって計算され，スタックから削除される．後継集合を計算する前に，位相順序でスタック上の強連結成分をソートすることで，和集合を取る操作数を必要最小限に抑えている．

Tarjan の強連結成分分解アルゴリズム

Tarjan は文献 [23] で $\Theta(V+E)$ 時間で実行される見事な強連結成分分解アルゴリズムを示した．このアルゴリズムは，グラフに対し 2 つの探索を交互に行っている．1 つ目，深さ優先探索はすべての辺を探索し，深さ優先スパニング木を構成する．2 つ目，強連結成分のいわゆる根を見つけると，以前に発見した強連結成分の要素でないすべての子孫がこの強連結成分の要素としてマークされる．この 2 番目の探索は，スタックを用いて実行される．ただし，深さ優先順序で各頂点に訪問する．根を離脱する前に，根までのすべての頂点がスタックから取り出され，どの強連結成分に属するか問われる．

Tarjan のアルゴリズムを図 6.1 に示す．再帰的手続き VISIT と，未探索の各頂点に VISIT を当てはめるメインプログラムから成る．VISIT は深さ優先順でグラフの頂点にアクセスする．それぞれの強連結成分 C において，VISIT がアクセスする C の最初の頂点は強連結成分 C の根 (root) と呼ばれる．アルゴリズムの重要な仕事は，強連結成分の根を見つけることである．そこで，各頂点 x の根を $Root(x)$ で表す．VISIT が頂点 x を処理しているとき，$Root(x)$ は x を含む強連結成分の根の候補を保持している．

3 行目，頂点 x は自身が根の候補として初期化される．VISIT が 5 から 8 行目で，頂点 x から伸びる辺を処理するとき，x と同じ強連結成分に属す子から新しい根の候補を得る．7 行目での MIN 操作は，VISIT が頂点を訪問した順番を用いて頂点同士を比較する．すなわち，VISIT が頂点 y を訪問する前に頂点 x を訪問したならば，$\mathrm{MIN}(x,y)=x$．そうでなければ，$\mathrm{MIN}(x,y)=y$ となる．これを実行する簡単な方法は，配列とカウンタを利用して，深さ優先順序を各頂点に固有に与えることである．VISIT が x から伸びる全ての辺を処理したとき，x が x を含む強連結成分の根である時，かつその時に限り $Root(x)=x$ である (9 行目)．しかしながら，x が強連結成分の根でないのであれば，$Root(x)$ が，x を含む強連結成分の正しい根であるかどうかは知ることができない．

頂点 x と同じ強連結成分に属する頂点と他の強連結成分に属する頂点を見分けるために，各頂点 y の属する強連結成分を $Comp(y)$ で表す．初期値は Nil である．強連結成分 C が検出されたとき，VISIT は C に属する各頂点を $Comp(y)$ に設定する (1 3 行目)．補助スタックをこのために用いる．各頂点は初めに VISIT のスタックに格納される．強連結成分が検出されるとき，その強連結成分に属す頂点がスタック上にある．VISIT はスタックからそれらの

```
1   procedure VISIT(x)
2       begin
3           Root(x) := x; Comp(x) := Nil
4           PUSH(x, stack)
5           for each vertex y ∈ AbjFrom(x) do
6               if y is not aleady visited then VISIT(y)
7               if Comp(y) = Nil then Root(v) :=MIN(Root(x), Root(w))
8           end for
9           if Root(x) = x then begin
10              create a new component C
11              repeat
12                  y := POP(stack)
13                  insert w into component C
14                  Comp(y) := C
15              until y = x
16          end if
17      end
18      begin /* Main program */
19          stack := ∅
20          for each vertex x ∈ V
21              if x is not already visited then VISIT(x)
22      end
```

図 6.1　Tarjan's algorithm detects the strongly connected components of graph $G = (V, E)$.

頂点 y を移し，それら y の変数 $Comp(y)$ を設定し，強連結成分 C に y を挿入する．

Tarjan のアルゴリズム以外の別の線形時間強連結成分分解アルゴリズムが多くの教科書に示されている．アルゴリズムは R.Kosaraju の論文 [43] の結果であると考えられる．このアルゴリズムは，入力グラフの深さ優先探索と，辺を逆にして得られるグラフの深さ優先探索の二度の探索を必要とするため，Tarjan のアルゴリズムに劣っている．

深さ優先スパニング木

$G = (V,\ E)$ をグラフとする．Tarjan のアルゴリズムを実行して生成された G の深さ優先スパニング木とは，G のスパニング木 $F = (V, E')$ である．ただし E' は，Tarjan のアルゴリズム 6 行目で，手順 VISIT が辺 (x, y) を通って y に探索する時，かつその時に限り

$(x, y) \in E'$ を満たす辺集合である．

この定義によって，パス $x \xrightarrow{*} y$ が深さ優先スパニング木に存在する時，かつその時に限り VISIT(x) の実行は VISIT(y) の実行を包含する．Tarjan のアルゴリズムの２０行目で頂点をスキャンする順序と，５行目で辺をスキャンする順序が固定されていないため，通常，グラフには多くの深さ優先スパニング木が存在する．

定義 6.1　$\leq_\tau$ で，Tarjn のアルゴリズムの実行におけるグラフ $G = (V, E)$ の深さ優先順序を示す．$\leq_\tau$ は，Tarjn のアルゴリズムの実行において手順 VISIT が y の前に x にを探索する時，かつその時に限り，$x <_\tau y$ となり，頂点集合 V に関する全順序となる．

深さ優先スパニング木と同様に，グラフには，多くの深さ優先探索順序がある．もし，深さ優先スパニング木 F がパス $x \xrightarrow{*} y$ を含むならば，$x \leq_\tau y$ であることに注意されたい．逆は必ずしも真ではない．

定義 6.2　Tarjan のアルゴリズムの実行によって生成された，深さ優先スパニング木 $F = (V, E')$ と，G の深さ優先順序 $\leq_\tau$ に伴って，グラフ $G = (V, E)$ の辺集合 E は以下の 4 つのグループに分けられる．(x, y) を E の要素とし，T_y を y を根とする F の部分木であるとする．

1. $(x, y) \in E'$ ならば，(x, y) は木辺 (tree edge) である．
2. $x <_\tau y$ かつ $(x, y) \notin E'$ ならば，(x, y) は前進辺 (forward edge) である．
3. $y \leq_\tau x$ かつ $x \in T_y$ ならば，(x, y) は後退辺 (back edge) である．
4. $y <_\tau x$ かつ $x \notin T_y$ ならば，(x, y) は交差辺 (cross edge) である．

後退辺がグラフにおいて１つのサイクルを表すことに注意されたい．

STACK_TC

STACK_TC を，図 6.2 に示す．STACK_TC は頂点を格納する代わりに後者集合に強連結成分を格納する．後者集合 $Succ(C)$ には強連結成分 C 自身が挿入される．STACK_TC は辺 (v, w) 間の木辺か，または交差辺である場合を除いて，頂点 v を残し辺 (v, w) を処理する．$Succ(Root(v))$ に $Comp(w)$ と $Succ(Comp(w))$ を追加する代わりに，STACK_TC では補助スタック $cstack$ に $Comp(w)$ を格納する．これは，13 行目で行われる．強連結成分 C が検出された時，新しい後者集合 $Suce(C)$ を作成する．C が自明であるか，$C = r$ で，入力グラフに自己ループ (r, r) が含まれている場合，$Succ(C)$ に C を含める．20-21 行目で，C の検出時に $cstack$ に格納されている強連結成分を位相順序でソートし，重複を排除している．強連結成分 C の検出時，$cstack$ に格納されている強連結成分の数を知るため，5 行目でローカル変数 $SavdedHeight(V)$ に $cstack$ の現在の高さを格納している．位相順に強連結成分を選択したのち，22-25 行目で $cstack$ から強連結成分を削除する．各強連結成分 X について，X が

$Succ(C)$ にすでに含まれている場合，X は $cstack$ から削除される．また，X が $Succ(C)$ に含まれない場合，$Succ(C)$ に X と $Succ(X)$ を加える．

ソートの詳細は，図 6.2 に示されていない．それは，次の方法で効率的に行うことができる．r を強連結成分 C の根とする．TOP と $SavedHeight(r)$ の間にある $cstack$ 上の強連結成分をスキャンし，ビットベクトルを使用して，現在重複している強連結成分を削除する．TOP と $SavedHeight(r)$ の間にある $cstack$ 上の固有の強連結成分の数 x が少ない，すなわち，$x \log x$ が $|V|$ より小さい場合，いくつかの一般的なソートアルゴリズムを使用して，位相順序にそれらを並べ替えることができる．それ以外の場合，ヒットベクトルを用いる．

アクセスグラフは 2 部グラフであり，自己ループを持たない．したがって，図 6.2 のアルゴリズムから自己ループの判定に関する操作は除外できる．また，再帰的な関数の定義と操作は入力グラフが大きくなったときに，スタック領域が飽和するエラー，スタックオーバーフローを引き起こす原因になる．そこで，実装にあたっては，再帰的な書き方を避ける．

このアルゴリズムの最悪計算量は後者集合を bit ベクトルで表すとき，$\mathcal{O}(V\bar{V} + VE_r + \mu)$ である．

```
1   procedure STACK_TC(x)
2       begin
3           Root(x) := x; Comp(x) := Nil
4           PUSH(x, vstack)
5           SavedHeight(x) :=HEIGHT(cstack)
6           SelfLoop(x) :=false
7           for each vertex y ∈ AbjFrom(x) do
8               if y = x then SelfLoop(x) :=true
9               else
10                  if y is not aleady visited then TACK_TC(y)
11                  if Comp(y) = Nil then Root(v) :=MIN(Root(x), Root(w))
12                  else if (x, y) is not a foward edge then
13                      PUSH(Comp(y), cstack)
14              end if
15          end for
16          if Root(x) = x then begin
17              create a new component C
18              if TOP(vstack)≠ x or SelfLoop(x) then Succ(C) := {C}
19              elseSucc(C) := ∅
20              sort the components in cstack between SavedHeight(v) and
21              HEIGHT(cstack) into a topological order and elimintal duplicates.
22              while HEIGHT(cstack)≠ SavedHeight(x) do begin
23                  X :=POP(cstack)
24                  if X ∈ Succ(C) then Succ(C) := ∪{X} ∪ Succ(X)
25              end while
26              repeat
27                  y := POP(vstack)
28                  Comp(y) := C
29                  insert w into component C
30              until y = x
31          end if
32      end
33      begin /* Main program */
34          vstack := ∅; cstack := ∅;
35          for each vertex x ∈ V
36              if x is not already visited then TACK_TC(x)
37      end
```

図 6.2　Esko's algorithm STACK_TC.

Step.3　$\bar{G}_A^{\#}$ の計算

5.2 節でも述べたが，Step.3 は図 6.3 に示すアルゴリズムで実行できる．ただし，強連結成分 C は，自身に属する頂点を配列で保持しているものとする．また，$NumOfComp(y)$ は頂点 y が属する強連結成分 $Comp(y)$ の何番目の要素に y が格納されているかを表す．これにより，$Comp(y)[NumOfComp(y)] := Nil$ の操作で，$Comp(y) - \{y\}$ の計算が $\mathcal{O}(1)$ で行える．

$S \cap C$ と $O \cap C$ は別々に構築できる．covert channel 検出のみを行うのであれば，$O \cap C$ は必要ない．

図**??**の最悪計算量は $\mathcal{O}(OS) = \mathcal{O}(V^2)$ である．

```
1  for each object x ∈ O do
2      for each vertex y ∈ AbjFrom(x) do Comp(y)[NumOfComp(y)] := Nil
3      for each strong compornent C ∈ Succ(Comp(x)) do
4          Succ(x) := Succ(x) ∪ (S ∩ C)
5      for each vertex y ∈ AbjFrom(x) do Comp(y)[NumOfComp(y)] := y
```

図 6.3　A calculation $G_A^{\#}$ from G_A and $\bar{G}_A^{+}$

第 7 章

区間表現

強連結成分分解を利用した推移閉包計算では，実行時間の大部分が後者集合を構成することに費やされる．後者集合の構成時間は，集合演算 (すなわち，構成頂点の探索，挿入，和集合の計算) の実行時間によって決定される．STACK_TC は集合演算数を減らすことに注力し，それによって良い計算量を得ている．

本章では，後者集合の適切な表現を選ぶことで，集合演算に必要な時間がどのように減少するかを紹介する．bit 配列，リスト，または二分ヒープなどの集合表現より必要とするメモリスペースが少なく，より速い集合演算を可能にする区間表現を示す．これは強連結成分の反転位相順序による区間に基づいている．この表現は非巡回グラフのコンパクトな推移閉包表現のために開発された方法に基づいている [26].

7.1 一般的な集合表現の特性

表現を示す前に，bit ベクトル，リスト，または二分ヒープなどの一般的な集合表現のいくつかの重要な特性を調べる．

一般的な共通集合表現が使用されるとき，後者集合 $Succ$ は，$\Omega(Succ)$ のメモリ空間を占有する．２つの集合 $Succ_1, Succ_2$ の和集合のサイズは $\Omega(\max(Succ_1, Succ_2))$ である．グラフ G の推移閉包 G^+ には，少なくとも G と同じくらい多くの辺がある．更に言えば，G^+ の辺 E^+ の数は G の辺 E の数よりかなり大きいかもしれない．

例えば，G が $|V|$ 個の頂点と辺からなる一つのサイクルであるとき，$|E| = |V|$ に対して $|E^+| = |V|^2$ である．このとき，一般的な集合表現を用いると，後者集合は $\Omega(E^+)$ のメモリ空間を占有する，そして，最良時間計算量は $\Omega(E^+)$ である．

一般的な集合表現における弱点は，その一般性のために推移閉包に特化していないことである．

7.2　区間表現

区間表現は，推移閉包アルゴリズムに使用できる後者集合のコンパクトな表現である．上記で議論した集合データ構造とは逆に，この表現はしばしば集合 A を $\Omega(A)$ より小さいメモリ空間に保存できる．この表現が使用されているとき，2 つの集合 A, B の和集合は引数のどちらかよりさらに小さいかもしれない．通常，$\mathcal{O}(E^+)$ よりはるかに小さい空間計算量と，$\Omega(E^+)$ によって制限されない最良時間計算量によって推移閉包を格納できる．この表現は非巡回グラフの推移閉包を圧縮するための方法に基づいている [26].

整数集合をコンパクトに保存するためのメソッドと，強連結成分を整数に写像するためのメソッドの 2 つの部品から表現は構成される．

整数集合をコンパクトに格納するための方法を最初に解説する．集合 $Succ \subseteq \{1, 2, \cdots, n\}$ を考える．もし，$Succ$ が連続整数 $i, i+1, \cdots, i+m$ $(m > 2)$ の 1 つ以上の数列から成るならば，$Succ$ をコンパクトに表す方法は，それぞれの極大な連続数列の始点と終点だけを格納することである．そのような表現を区間表現と呼ぶ．区間 $[i, j]$ は集合 $\{i, i+1, \cdots, j\}$ を表す．また，区間の集合 $\{I_1, I_2, \cdots, I_r\}$ は区間が表す集合の和集合を表現する．新しい区間が集合に追加されるとき，重なっている区間は結合される．同様に，もし 2 つの区間 $[i, j]$ と $[j+1, k]$ が存在すれば，それらは区間 $[i, k]$ に統合される．

例 7.1　集合 $Succ_1 = \{0, 1, 2, 4, 5, 6, 8\}$ は 3 つの区間 $[0, 2], [4, 6], [8, 8]$ によって格納される．また，集合 $Succ_2 = \{0, 2, 3, 4, 5, 9, 10\}$ は 3 つの区間 $[0, 0], [2, 5], [9.10]$ によって格納される．7 を $Succ_1$ に挿入した場合，集合 $Succ'_1 = \{0, 1, 2, 4, 5, 6, 7, 8\}$ を得る．この集合は 2 つの区間 $[0, 2], [4, 8]$ によって格納される．$Succ_1$ と $Succ_2$ の和集合を取ると，集合 $\{0, 1, 2, \cdots, 10\}$ が得られる．この集合は 1 つの区間 $[0, 10]$ によって格納される．　□

$Succ = \{1, 3, 5, \cdots, 2\lfloor (n-1)/2 \rfloor + 1\}$ であるとき，必要な区間数は上界に達する．したがって，必要な区間の最大数は $\lceil n/2 \rceil$ であり，$Succ$ を格納するには $\Omega(n)$ のスペースが必要となる．

必要な区間数の期待値は計算が難しい．それは $Succ$ での整数の分布による．もし集合 $Succ$ に含まれる整数が何らかの値 x の近くに偏っているならば，必要な区間の数が高い確実性で少なくなることに注意すべきである．

次に，区間表現の 2 つ目の部品，強連結成分を整数に写像するための方法を調べる．使用する写像は単純である．区間が含む整数は，強連結成分の反転位相順序の番号である．強連結成分は順序にしたがって次第に付番され，検出されるということである．通常，成分の無作為な付番より，この付番の方が，後者集合を表すのに必要な区間がはるかに少なくて済む．理由は二つある．まず最初に，成分 C の後者となり得る全ての成分は，C より小さい番号しか持た

ないためである．これにより，$Succ(C)$ の要素は $\{1, 2, \cdots, n\}$ からではなく，$\{1, 2, \cdots, k\}$ から得られる．ここで，k は C の反転位相順序における番号である．次に，深さ優先探索で生成されたスパニング木の部分木は単一の区間として表されるためである．付番は成分が検出される順番によって，多くの順序が有り得る事に注意が必要である．番号の付け方によっては，後者集合を表すために必要な区間の数が異なる．

例 7.2　図 7.1 に，非巡回グラフと，その頂点の後者集合を，グラフのある走査 $(g, f, i, h, c, d, e, a, b)$ によって生成された区間の集合で示す．各頂点に関しては，図に反転位相順序番号を示す (”rtn:”で示される)．また，後者を区間の集合として示してある (”s:”で示される)．この走査では，後者集合を表すのに必要な区間の最大数は 2 である，そして，必要な区間の総数は 7 である．　□

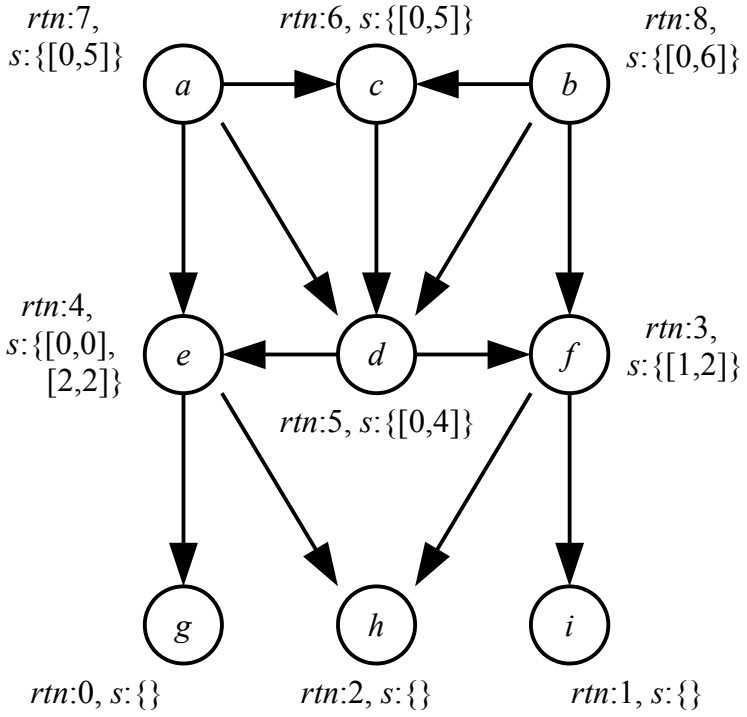


図 7.1　A DAG and the successor sets of its vertices represented as intervals.

次の例は，この付番メソッドでも，後者集合を表すのに $\Omega(n^2)$ の空間計算量が必要になることを示している．

例 7.3　図 7.2 では，$n = 2m + 1$ の頂点を持つグラフを例に示す．頂点 0 から $1, \cdots, m$ まで辺がある．頂点 $m + 1, \cdots, 2m$ から，$1, \cdots, m$ まで辺がある．ここで，m は偶数であると仮定する．Tarjan のアルゴリズムに基づく推移閉包アルゴリズムによってこのグラフの推移閉

包を計算し，区間表現を用いるときに何が起こるか調べる．頂点 0 から実行され，0 の隣接リストが $1, 2, \cdots, m$ の順にあると仮定する．各頂点 $i(1 \leq i \leq m)$ は自明な強連結成分であり，反転位相順序番号は i である．0 の後者集合は単一の区間 $[1, m]$ から成る．その後，アルゴリズムが頂点 $j(m+1 \leq j \leq 2m)$ の後者集合を構成するとき，後者集合は $(m+1)/2$ 個の区間から成る．したがって，$\Omega(m^2) = \Omega(n^2)$ の区間が必要となる．　□

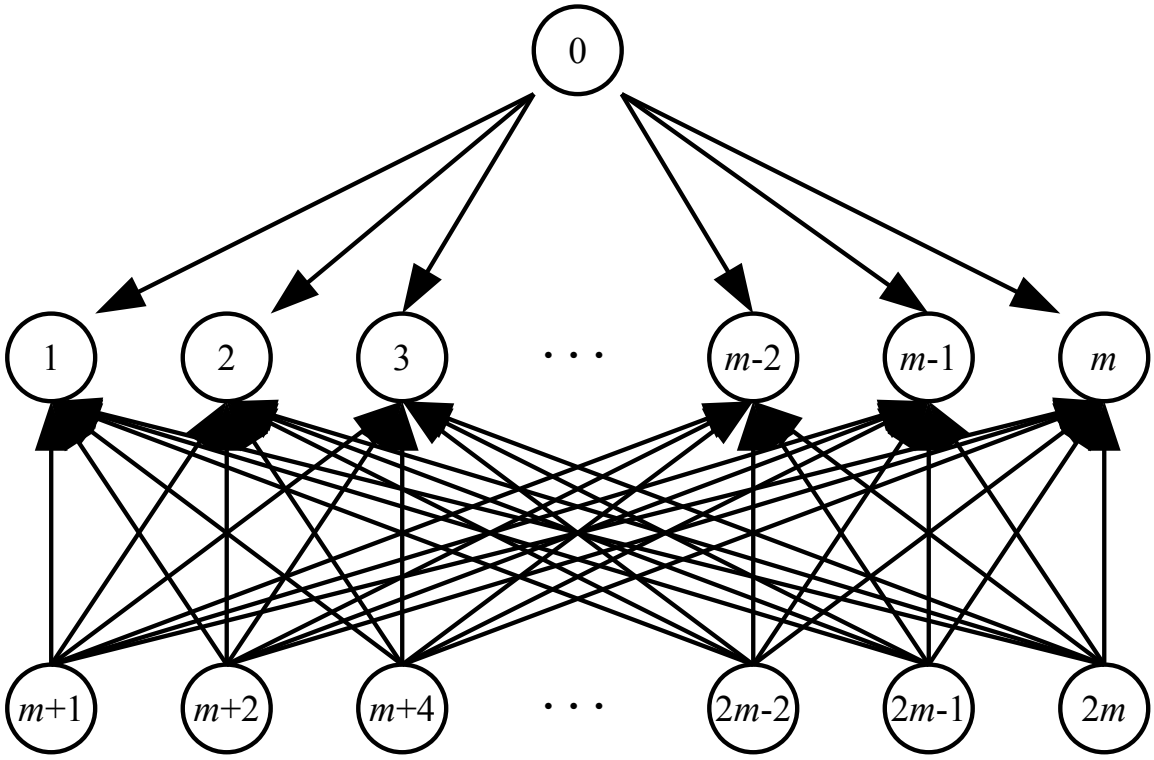


図 7.2　A graph the successor sets of which may require $\Omega(n^2)$ space.

グラフの後者集合を表すのに必要な区間の平均値は解析が難しい．なぜなら，区間の数が頂点の数とグラフの辺だけでなく，グラフの位相によっても変わるからである．さらに，同じグラフであっても異なった走査によって，異なった数の区間がもたらされる．別の問題は入力の適切なモデルを選ばなければならないことである．

いくつかのパスに後者集合を表す区間の集合を格納できる．区間が異なっているため，二分ヒープに格納できる．二分ヒープは平衡をとることができるが，区間の木は小さいために通常は必要としない．別の可能性として, ソート済み配列に区間を格納できる．$Succ, Succ_1, Succ_2$ を区間 I, I_1, I_2 の集合として表される後者集合であるとする．$Succ_1$ と $Succ_2$ の和集合の計算は $\mathcal{O}(I_1 + I_2)$ 時間かかる．区間が平衡な二分ヒープかソート済み配列に格納されるなら

ば，集合 $Succ$ における要素 x の存在を確かめることは，最悪 $\mathcal{O}(\log(I))$ の計算量が必要となる．区間が非平衡な二分ヒープに格納されている場合，最悪 $\mathcal{O}(I)$ の計算量が必要となる．区間が平衡な二分ヒープに格納されるならば，要素 x を集合 $Succ$ に挿入することは，最悪 $\mathcal{O}(\log(I))$ の計算量が必要となる．区間が非平衡な二分ヒープかソート済み配列に格納されている場合，最悪 $\mathcal{O}(I)$ の計算量が必要となる．二分ヒープ表現はソート済み配列表現の 2 倍のメモリ空間を必要とする．配列表現は，メモリ割り当てがより簡単である．後者集合を表すのに必要な区間の数が通常小さいのであれば，区間をソート済み配列としてすべての後者集合を格納することは，恐らく賢明である．

例 7.4　図 7.1 に示された非巡回グラフを考える．図 7.3 では，再びこのグラフとその推移閉包を示すが，今度は Agrawal 他の方法を用いて後者集合が計算されている．図の各頂点に関して，スパニング木によって頂点を含む区間を示す (”i:”で示される)，後者は区間の集合として与えた (”s:”で示される)．アルゴリズムによって生成されるスパニング木の辺は実線で描かれ，他の辺 (交差辺) を破線で描いている．　□

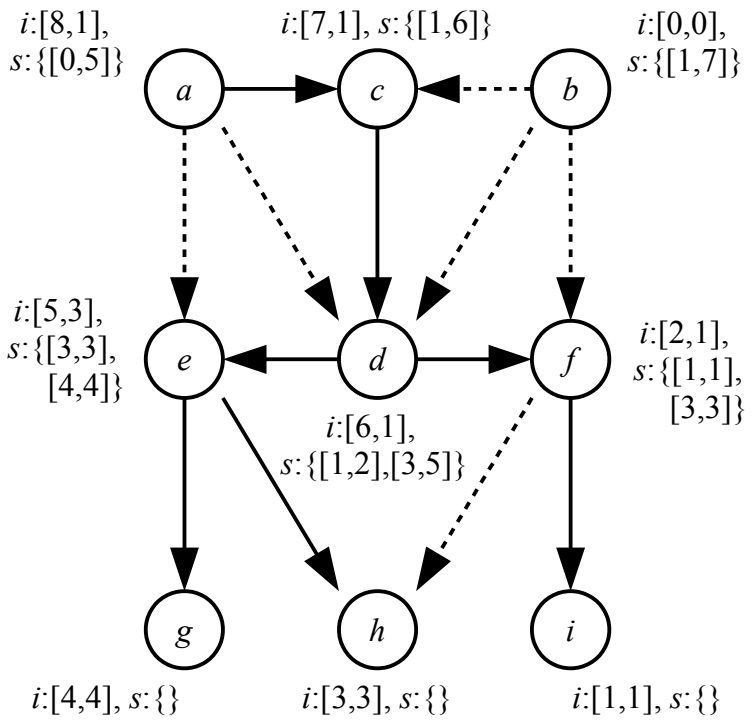


図 7.3　The interval representaition of the graph in Figure 7.1 when the method of [26] is used.

第 8 章

シミュレーション

提案アルゴリズムと，先行研究の幅優先探索を用いたアルゴリズム [16] を Java で実装し，そのソースコードを提示する．また，それぞれの実行時間を測定し比較する．実行時間を評価する際，実際のデータをテストデータに用いることは良い方法である．しかし，残念なことに実際の ACL を入手することはできなかった．そこで，ランダムな有向 2 部グラフを生成し，これを入力とすることでその性能を評価した．したがって本章での性能評価は現実の ACL に即したものではないことに注意されたい．

定義 8.1　ランダム有向 2 部グラフ $G(n, m, p)$ は，n 個の要素からなる object 集合 $O := \{o_1, o_2, \cdots, o_n\}$ と，m 個の要素からなる subject 集合 $S := \{s_1, s_2, \cdots, s_m\}$ を独立頂点集合に持つ 2 部グラフである．任意の順序対 $(x, y) \in O \times S$ は確率 $p \in [0, 1]$ で $G(n, m, p)$ の辺となる．同様に，任意の順序対 $(x, y) \in S \times O$ は確率 $p \in [0, 1]$ で $G(n, m, p)$ の辺となる．

ランダム有向 2 部グラフ $G(n, m, p)$ に関する簡単な期待値の計算を示しておく．

$G(n, m, p)$ の辺数の期待値は $|E| = \sum_{(x,y)\in O\times S} p + \sum_{(x,y)\in S\times O} p = 2|O||S|p = 2nmp$ である．

ある object x の出次数の期待値は $Outdeg(x) = \sum_{y\in S} p = mp$ である．したがって，object の平均出次数の期待値は $d(O) = (\sum_{x\in O} Outdeg(x))/n = nmp/n = mp$ である．同様にして，subject の平均出次数の期待値は $d(S) = np$ である．

また, 全ての頂点の平均出次数の期待値は $d(V) = (nd(O)+md(S))/(n+m) = 2nmp/(n+m)$ である．

シミュレーションを行う前に，次節で，アルゴリズムの $G(n, m, p)$ に対する計算量の期待値 (平均計算量) を評価する．非常に荒い評価であるが，ある閾値によって $G(n, m, p)$ の構造が大きく変化することがわかる．

8.1　計算量の期待値

ランダム有向 2 部グラフ $G(n,m,p)$ について，濃度が k の object の部分集合を $O' \subset O$ とする．また，濃度が k の subject の部分集合 $S' \subset S$ とする．ただし，$k = \min(n,m)$ である．

$G(n,m,p)$ の部分グラフ $G' = (O', S', E')$ について，G' の全ての頂点をちょうど 1 度だけ通るサイクルの個数の期待値を K と書こう．このようなサイクルは O' と S' の要素を交互に $2k$ 個円形に並べ，それぞれの頂点が右隣の頂点へ辺が伸びるようなグラフとして考えられる．O' と S' の要素を交互に $2k$ 個円形に並べる場合の数は，$(k-1)! \times (k-1)! = (k!/k)^2$ である．ただし，$k!$ は k の階乗を表す．O' と S' の要素を交互に $2k$ 個円形に並べたとき，全ての頂点が右隣の頂点へ辺を持つ確率は $p^{2k} = (p^k)^2$ である．しからば，求めるサイクルの個数の期待値は $K = (k!/k)^2 \times (p^k)^2 = (k!p^k/k)^2$ となる．

π を円周率，e をネイピア数とする．スターリングの近似公式として，次の不等式が知られている．$\sqrt{2\pi k}(k/e)^k e^{1/(12k+1)} < k! < \sqrt{2\pi k}(k/e)^k e^{1/(12k)}$．これより，$K > (\sqrt{2\pi k}(k/e)^k p^k/k)^2 = (\sqrt[2k]{2\pi/k}(k/e)p)^{2k}$ が成り立つ．

ここで，$f(k) = \sqrt[2k]{2\pi/k}$ が極値を取る k を求める．

$$
\begin{aligned}
f'(k) &= 0 \\
-\sqrt[2k]{\frac{2\pi}{k}}\frac{\log_e(2\pi/k)+1}{2k^2} &= 0 \\
k &= 2\pi e
\end{aligned}
$$

$f(1) = \sqrt{2\pi} > 2.5$，$\lim_{k\to\infty} f(k) = 1$ である．$f(2\pi e) = \sqrt[4\pi e]{1/e} < 1$ であるから，$f(2\pi e) = \sqrt[4\pi e]{1/e}$ は極小値であり，最小値である．これより，$K > (\sqrt[2k]{2\pi/k}(k/e)p)^{2k} = (f(k)(k/e)p)^{2k} > (k/e\sqrt[4\pi e]{e})p$ が成り立つ．更に，$e\sqrt[4\pi e]{e} < 3$ より，$K > (k/e\sqrt[4\pi e]{e})p > (k/3)p$ が成り立つ．

しかるに，$3 \le pk = p\min(n,m)$ であれば，$1 < K$ である．つまり，$3 \le p\min(n,m)$ であれば，G' の全ての頂点をちょうど 1 度だけ通るサイクルの個数の期待値は，1 より大きくなる．すると，G' の任意の頂点は，そのサイクルにそってパスをたどることで任意の頂点に到達できる．すなわち，G' は 1 つの強連結成分となる．この強連結成分を C' とする．

このとき，後者集合の濃度の期待値 $|Succ|$ の下界はどうなるだろうか．話を簡単にするため，$k = n \le m$ と仮定する．$|O'| = k = n = |O|$ であるから $O' = O$ が成り立つ．subject の平均出次数の期待値 $d(S) = np > 3 > 1$ より，$S - S'$ の任意の頂点は，それぞれ G' の $O' = O$ の頂点に対して辺を持つ．一方，$S - S'$ のある頂点が，O' の全ての頂点から辺を貰わない確率は $(1-p)^k$ である．

Shannon は，シャノンの補助定理の証明において不等式 $\log_e(x) \leq x-1$ を用いた．この不等式は，$0 \leq x \leq 1$ と仮定すれば次のように変形できる．

$$\begin{aligned}\log_e(x) \leq x-1 &\Rightarrow \log_e(1-x) \leq -x \\ &\Rightarrow \log_e(1-x)^{1/x} \leq -1 \\ &\Rightarrow (1-x)^{1/x} \leq \exp(-1)\end{aligned}$$

これより，$(1-p)^k = ((1-p)^{1/p})^{kp} \leq \exp(-kp) \leq= \exp(-3) < 0.05$ が成り立つ．したがって，G' の object から辺を全く貰わない $S-S'$ の頂点の個数の期待値は $0.05|n-m|$ 個以下である．すなわち，$S-S'$ の頂点のうち 95% 以上の頂点は G' の頂点と強連結である．つまり，$G(n,m,p)$ の 95% 以上 $(n+m-0.05|n-m| > 0.95(n+m))$ の頂点が一つの強連結成分 C' となる．ゆえに，$|Succ| = \Omega(0.95V) = \Omega(V)$．また，明らかに後者集合は全頂点数を超えないので，$|Succ| = \mathcal{O}(V)$．しかるに，$3/\min(n,m) \leq p \Rightarrow |Succ| = \Theta(V)$ となる．

また，このとき，$G(n,m,p)$ の凝縮グラフ $\bar{G}$ の頂点数 $|\bar{V}|$ と，辺数 $|\bar{E}|$ の上界はどうなるだろうか．上の議論で $G(n,m,p)$ の 95% 以上の頂点が一つの強連結成分 C' となることが示された．一方で C' でない (自明な) 強連結成分 C'' は $0.05|n-m|$ 個以下しか存在しない．したがって，$|\bar{V}| \leq 0.05|n-m|+1$．また，$(C'',C') \in \bar{E}$ と $(C',C'') \in \bar{E}$ が同時に成り立つことはありえないし，(自明な) 強連結成分は S の頂点であるから，互いに辺を持つことがない．したがって，$|\bar{E}| \leq 0.05|n-m|+1$．しかるに，$3/\min(n,m) \leq p \Rightarrow |\bar{V}| = \mathcal{O}(|n-m|+1)$, $|\bar{E}| = \mathcal{O}(|n-m|+1)$ となる．

以上の議論を踏まえると，$3 \leq p\min(n,m)$ のとき，提案アルゴリズムの $G(n,m,p)$ に対する計算量の期待値は，$\mathcal{O}(V\bar{E}_r + V\bar{V} + OS) = \mathcal{O}((n+m)|n-m| + n + m + nm) = \mathcal{O}(|n^2-m^2| + n + m + nm) = \mathcal{O}(|n^2-m^2| + nm)$ となる．これは，$n \approx m$ のとき，object 数と subject 数の積に対して線形である．先行研究の幅優先探索を用いたアルゴリズムの $G(n,m,p)$ に対する計算量の期待値は，$\mathcal{O}(O(V+E)) = \mathcal{O}(n(n+m+nmp))$ となる．

$3 > p\min(n,m)$ のとき，辺の期待値が $|E| = \mathcal{O}(\max(n,m))$ であるから，提案アルゴリズムの $G(n,m,p)$ に対する計算量の期待値は，$\mathcal{O}(V\bar{E}_r + V\bar{V} + \mu + E) = \mathcal{O}((\max(n,m))^2)$ である．先行研究の幅優先探索を用いたアルゴリズムの $G(n,m,p)$ に対する計算量の期待値は，$\mathcal{O}(O(V+E)) = \mathcal{O}(n\max(n,m))$ となる．

余談ではあるが，$\log_e(\max(n,m)) \leq \min(n,m)p$ ならば次の不等式が成り立つ．

$$\begin{aligned}\lim_{n,m\to\infty} |n-m|e^{-kp} &= \lim_{n,m\to\infty} \max(n,m)e^{-kp} - \lim_{n,m\to\infty} \min(n,m)e^{-\min(n,m)p} \\ &= \lim_{n,m\to\infty} \max(n,m)e^{-\min(n,m)p} \\ &\leq 1\end{aligned}$$

これは，$\log_e(\max(n,m))/\min(n,m) \leq p$ ならば，漸近的には $G(n,m,p)$ の全ての頂点が互いに強連結になることを意味する．このとき，提案アルゴリズムの $G(n,m,p)$ に対する計算量の期待値は，グラフのサイズに対して線形である．

8.2　平均計算量の有意差

平均計算量に有意差があるかシミュレーション実験により調べた．シミュレーションの流れは以下の通りである．実験に用いたプログラムのソースコードは付録 A に添付する．

1. パラメーターを $p = 0.001, 0.009, \cdots, 0.0001$．$n = 1000, 2000, \cdots, 10000$．$m = 1000, 2000, \cdots, 10000$．として，$G_A = G(n,m,p)$ を生成する．
2. 生成した G_A をそれぞれのアルゴリズムで解析したときの実行時間を計測する．ただし，ACL からアクセスグラフを構成する実行時間 (G_A を生成する時間) は実行時間の計測に含めない．
3. 測定結果を，2 標本 t 検定 [44] で，実行時間の平均値に有意差があるか検定する．ただし, 有意水準は $\alpha = 0.01$ とする．また,t 検定は R 言語を用いて行う．

検定の結果を表 8.1，8.2，8.3，8.4，8.5，8.6，8.7，8.8，8.9，8.10 に示す．

表の読み方は, 例えば表 8.1 であれば，表のタイトルにあるように $p = 0.0002$ である．$n = 4000, m = 7000$ の欄を見ると，1 であるが，これは $G(4000, 7000, 0.0002)$ をそれぞれのアルゴリズムで解析したとき，検定の結果, それぞれの平均実行時間に有意差があると判断できたことを意味する．また，いずれの結果においても，有意差がある場合は提案アルゴリズムのほうが平均実行時間が短かった．すなわち，パタメータが $n = 4000, m = 7000, p = 0.0002$ であるならば，提案アルゴリズムが優れていると結論できる．

逆に，$n = 2000, m = 5000$ の欄は 0 である．これは $G(2000, 5000, 0.0002)$ をそれぞれのアルゴリズムで解析したとき，検定の結果, それぞれの平均実行時間の有意差については判断できないことを意味する．すなわち，パタメータが $n = 2000, m = 5000, p = 0.0002$ であるならば，提案アルゴリズムと，先行研究のアルゴリズムの平均実行時間における優劣は決められない．

シミュレーション結果への評価

表 8.1，8.2，8.3，8.4，8.5，8.6，8.7，8.8，8.9，8.10 を参照すると，p の値が増加にともなって，有意差のある欄の数が増加している．また，同じ p の値に対しては，n，m がそれぞれ大きいほうが有意差が出やすい．したがって，定性的な評価としては p, n, m が大きいほど有意差が出やすいことがわかる．では，有意差がある場合とない場合の境界 (表の 0,1 の境界) はどのように決まっているのだろうか．それぞれの欄で，$p\sqrt{nm}$ の値を計算すると，その値がちょうど 1 付近の欄で境界があることがわかった．つまり，「$p\sqrt{nm} < 1 + \epsilon$ ならば有意差がない，$p\sqrt{nm} > 1 - \epsilon$ ならば有意差がある」ということである．この結論は，実験で扱ったパラメータの範囲でしか言えないことではあるが，しかし重要な結論であろう．

実際，$p\sqrt{nm}$ を意味ある閾値関数として導出することは可能である．以下，本節ではこのことを示す．

閾値関数 $p\sqrt{nm}$

4.1.1 節で述べたとおり，2 部グラフであるアクセスグラフ G_A に対応する隣接行列 A は次式の形で表せる．

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} 0 & A_{1,2} \\ A_{2,1} & 0 \end{pmatrix} \tag{8.1}$$

これに習って，$G(n, m, p)$ に対する $(n + m) \times (n + m)$ 隣接行列 $A(n, m, p)$ を次式で定義する．

$$\begin{aligned} A(n, m, p) &= \begin{pmatrix} A_{1,1}(n, m, p) & A_{1,2}(n, m, p) \\ A_{2,1}(n, m, p) & A_{2,2}(n, m, p) \end{pmatrix} \\ &= \begin{pmatrix} 0 & A_{1,2}(n, m, p) \\ A_{2,1}(n, m, p) & 0 \end{pmatrix} \end{aligned} \tag{8.2}$$

$$A_{1,2}(n, m, p)[i, j] = A_{2,1}(n, m, p)[i, j] = p \tag{8.3}$$

なお，$A(n, m, p)$ は実行列として扱い，その行列積も実行列と同じ定義を用いるものとする．すると，$A^k(n, m, p)$ の i, j 成分は v_i から v_s への長さ k の (ことによると単純でない) パスの本数の期待値を表す．

ここで，$A_{1,2}^{2k-1}(n, m, p)[i, j] = n^{k-1}m^{k-1}p^{2k-1} = (nmp^2)^k/(nmp)$ が成り立つ．$|nmp^2| < 1$ ならば，$A_{1,2}^{2k-1}(n, m, p)[i, j] = 0$ $(k \to \infty)$ であるが，$|nmp^2| > 1$ ならば，$A_{1,2}^{2k-1}(n, m, p)[i, j] = \infty$ $(k \to \infty)$ である．$A_{1,1}^{2k}(n, m, p) = A_{1,2}^{2k-1}(n, m, p)A_{1,2}(n, m, p)$ であるから，同様に，$|nmp^2| < 1$ ならば，$A_{1,1}^{2k}(n, m, p)[i, j] = 0$ $(k \to \infty)$ であるが，$|nmp^2| > 1$ ならば，$A_{1,1}^{2k}(n, m, p)[i, j] = \infty$ $(k \to \infty)$ である．

したがって，$|nmp^2| > 1 \Rightarrow p\sqrt{nm} > 1$ ならば，任意の object から任意の頂点へ到達可能である．同様にして，$p\sqrt{nm} > 1$ ならば，任意の subject から任意の頂点へ到達可能である．

結局，$p\sqrt{nm} > 1$ のとき，全ての頂点が同じ強連結成分に属する．

$p\sqrt{nm} > 1$ のとき，$G(n, m, p)$ の辺数の期待値は $|E| = 2nmp > 2\sqrt{nm}$ である．object の平均出次数の期待値は $d(O) = mp > \sqrt{m/n}$ である．subject の平均出次数の期待値は $d(S) = np > \sqrt{n/m}$ である．

8.3　平均計算量の比

前節のシミュレーション実験で $1 < p\sqrt{nm}$ で計算量に有意差が現れることがわかった．そこで，提案アルゴリズムに比べて先行研究のアルゴリズムが何倍ほど遅いのか，シミュレーション実験により比を調べた．

1. パラメーターを $n = 1000, m = 10000.$　$n = 2000, m = 5000.$　$n = 3162, m = 3162.$　$n = 5000, m = 2000.$　$n = 10000, m = 1000.$ とし，変数を $p = 1000, 2000, 5000, 10000, 20000, 50000, 100000$ として，$G_A = G(n, m, p)$ を生成する．
2. 生成した G_A をそれぞれのアルゴリズムで解析したときの実行時間を計測する．ただし，ACL からアクセスグラフを構成する実行時間 (G_A を生成する時間) は実行時間の計測に含めない．

提案アルゴリズムの実行時間を t_1, 先行研究のアルゴリズムの実行時間を t_2 とする．nmp を横軸，t_2/t_1 を縦軸に取った散布図を図 8.1 に示す．

図 8.1 を参照すると定性的には，$1 < p\sqrt{nm}$ の範囲で，nmp の増加に伴って t_2/t_1 も増加している．しかし，各 p に対して t_2/t_1 の値を 10 回計測しており，ばらつきが大きいので散布図が見辛い．そこで，t_2/t_1 の平均を縦軸に取ってプロットしたものを図 8.2 に示す．

図 8.2 を参照すると，よりはっきりと nmp の増加に伴って t_2/t_1 が増加していることがわかる．これより，nmp が大きいほど，STACK_CC は ObjectsBFS に比べて効率的であると結論できる．

また，n, m のパラメータに注目して，図 8.2 を参照すれば，n の値が大きい時，すなわち Object 数が多い時ほど比が大きくなっている．

n \ m	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	0	0	0	0	0	0	0	0	0	0
2000	0	0	0	0	0	0	0	0	0	0
3000	0	0	0	0	0	0	0	0	0	0
4000	0	0	0	0	0	0	0	0	0	0
5000	0	0	0	0	0	0	0	0	0	0
6000	0	0	0	0	0	0	0	0	0	0
7000	0	0	0	0	0	0	0	0	0	0
8000	0	0	0	0	0	0	0	0	0	0
9000	0	0	0	0	0	0	0	0	0	0
10000	0	0	0	0	0	0	0	0	0	0

表 8.1　The experimental result; $G(n, m, p = 0.0001)$

n \ m	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	0	0	0	0	0	0	0	0	0	0
2000	0	0	0	0	0	0	0	0	0	0
3000	0	0	0	0	0	0	0	0	0	1
4000	0	0	0	0	0	0	1	1	1	1
5000	0	0	0	0	0	1	1	1	1	1
6000	0	0	0	0	1	1	1	1	1	1
7000	0	0	0	1	1	1	1	1	1	1
8000	0	0	0	1	1	1	1	1	1	1
9000	0	0	0	1	1	1	1	1	1	1
10000	0	0	1	1	1	1	1	1	1	1

表 8.2　The experimental result; $G(n, m, p = 0.0002)$

n \ m	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	0	0	0	0	0	0	0	0	0	0
2000	0	0	0	0	0	0	1	1	1	1
3000	0	0	0	0	1	1	1	1	1	1
4000	0	0	0	1	1	1	1	1	1	1
5000	0	0	1	1	1	1	1	1	1	1
6000	0	0	1	1	1	1	1	1	1	1
7000	0	1	1	1	1	1	1	1	1	1
8000	0	1	1	1	1	1	1	1	1	1
9000	0	1	1	1	1	1	1	1	1	1
10000	0	1	1	1	1	1	1	1	1	1

表 8.3　The experimental result; $G(n, m, p = 0.0003)$

n \ m	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	0	0	0	0	0	1	1	1	1	1
2000	0	0	0	1	1	1	1	1	1	1
3000	0	0	1	1	1	1	1	1	1	1
4000	0	1	1	1	1	1	1	1	1	1
5000	0	1	1	1	1	1	1	1	1	1
6000	1	1	1	1	1	1	1	1	1	1
7000	1	1	1	1	1	1	1	1	1	1
8000	1	1	1	1	1	1	1	1	1	1
9000	1	1	1	1	1	1	1	1	1	1
10000	1	1	1	1	1	1	1	1	1	1

表 8.4　The experimental result; $G(n, m, p = 0.0004)$

n \ m	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	0	0	0	0	1	1	1	1	1	1
2000	0	0	1	1	1	1	1	1	1	1
3000	0	1	1	1	1	1	1	1	1	1
4000	0	1	1	1	1	1	1	1	1	1
5000	1	1	1	1	1	1	1	1	1	1
6000	1	1	1	1	1	1	1	1	1	1
7000	1	1	1	1	1	1	1	1	1	1
8000	1	1	1	1	1	1	1	1	1	1
9000	1	1	1	1	1	1	1	1	1	1
10000	1	1	1	1	1	1	1	1	1	1

表 8.5　The experimental result; $G(n, m, p = 0.0005)$

n \ m	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	0	0	0	1	1	1	1	1	1	1
2000	0	1	1	1	1	1	1	1	1	1
3000	0	1	1	1	1	1	1	1	1	1
4000	1	1	1	1	1	1	1	1	1	1
5000	1	1	1	1	1	1	1	1	1	1
6000	1	1	1	1	1	1	1	1	1	1
7000	1	1	1	1	1	1	1	1	1	1
8000	1	1	1	1	1	1	1	1	1	1
9000	1	1	1	1	1	1	1	1	1	1
10000	1	1	1	1	1	1	1	1	1	1

表 8.6　The experimental result; $G(n, m, p = 0.0006)$

n \ m	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	0	0	1	1	1	1	1	1	1	1
2000	0	1	1	1	1	1	1	1	1	1
3000	1	1	1	1	1	1	1	1	1	1
4000	1	1	1	1	1	1	1	1	1	1
5000	1	1	1	1	1	1	1	1	1	1
6000	1	1	1	1	1	1	1	1	1	1
7000	1	1	1	1	1	1	1	1	1	1
8000	1	1	1	1	1	1	1	1	1	1
9000	1	1	1	1	1	1	1	1	1	1
10000	1	1	1	1	1	1	1	1	1	1

表 8.7　The experimental result; $G(n, m, p = 0.0007)$

n \ m	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	0	0	1	1	1	1	1	1	1	1
2000	0	1	1	1	1	1	1	1	1	1
3000	1	1	1	1	1	1	1	1	1	1
4000	1	1	1	1	1	1	1	1	1	1
5000	1	1	1	1	1	1	1	1	1	1
6000	1	1	1	1	1	1	1	1	1	1
7000	1	1	1	1	1	1	1	1	1	1
8000	1	1	1	1	1	1	1	1	1	1
9000	1	1	1	1	1	1	1	1	1	1
10000	1	1	1	1	1	1	1	1	1	1

表 8.8　The experimental result; $G(n, m, p = 0.0008)$

n \ m	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	0	1	1	1	1	1	1	1	1	1
2000	1	1	1	1	1	1	1	1	1	1
3000	1	1	1	1	1	1	1	1	1	1
4000	1	1	1	1	1	1	1	1	1	1
5000	1	1	1	1	1	1	1	1	1	1
6000	1	1	1	1	1	1	1	1	1	1
7000	1	1	1	1	1	1	1	1	1	1
8000	1	1	1	1	1	1	1	1	1	1
9000	1	1	1	1	1	1	1	1	1	1
10000	1	1	1	1	1	1	1	1	1	1

表 8.9　The experimental result; $G(n, m, p = 0.0009)$

n \ m	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1000	1	1	1	1	1	1	1	1	1	1
2000	1	1	1	1	1	1	1	1	1	1
3000	1	1	1	1	1	1	1	1	1	1
4000	1	1	1	1	1	1	1	1	1	1
5000	1	1	1	1	1	1	1	1	1	1
6000	1	1	1	1	1	1	1	1	1	1
7000	1	1	1	1	1	1	1	1	1	1
8000	1	1	1	1	1	1	1	1	1	1
9000	1	1	1	1	1	1	1	1	1	1
10000	1	1	1	1	1	1	1	1	1	1

表 8.10　The experimental result; $G(n, m, p = 0.0010)$

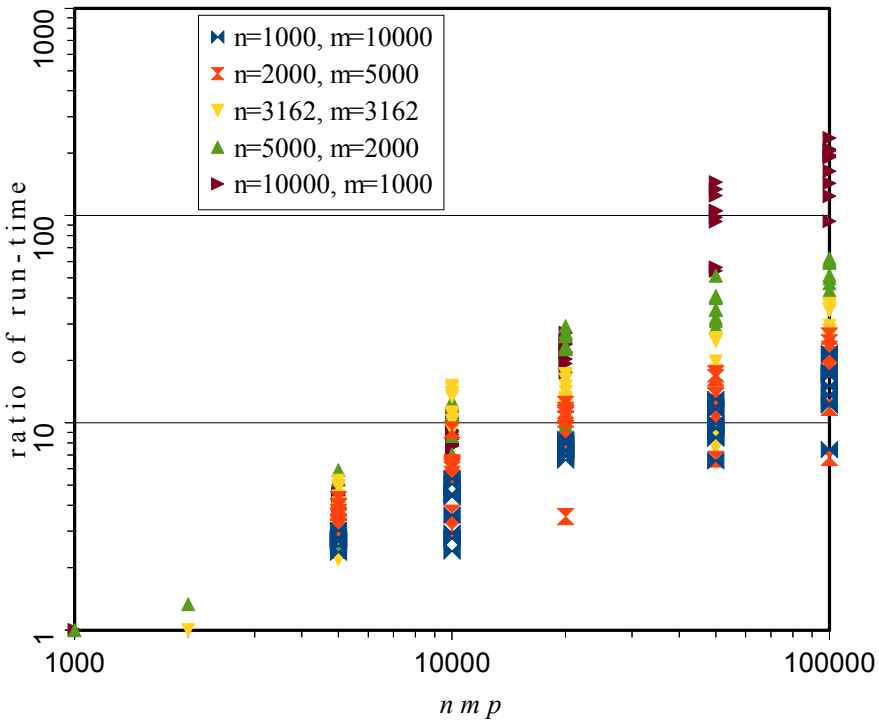


図 8.1　The scatter plot of nmp — t_2/t_1

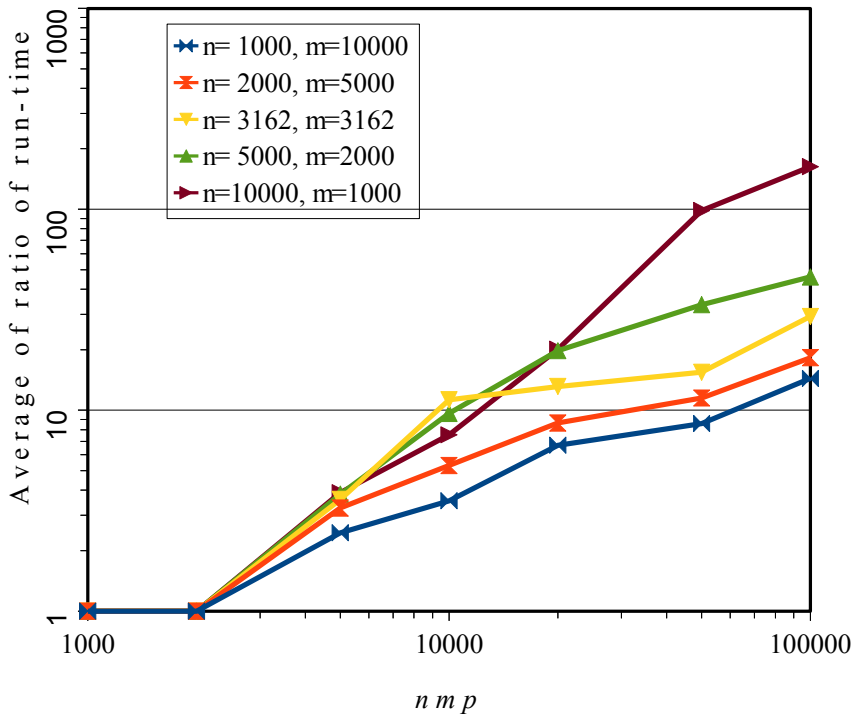


図 8.2　The scatter plot of nmp — average of t_2/t_1

第 9 章

結論

第 3 章では，ACL がグラフとして表現できることを示した．ACL のグラフによる表現では，object を始点とし，subject を終点とする，辺でないパスとして covert channel が表現されることを示した．また，covrt channel 検出問題が object 集合を始点集合とする，強い複数始点推移閉包問題の応用問題ないし変形問題であることを示した．ACL を表すグラフの良い性質として 2 部グラフであることを示した．この性質は，いくつかの covert channel 検出アルゴリズムを効率化する．ただし，本論文で提案したアルゴリズムはこの性質を使用しない．

第 4 章では，先行研究の covert channel 検出アルゴリズムが推移閉包問アルゴリズムの応用ないし変形であることを示した．したがって，covert channel 検出問題にとって推移閉包問題は有用な概念である．少し強く言えば，効率的な covert channel 検出アルゴリズムを考える上で，グラフ理論と推移閉包アルゴリズムは必須である．

第 5 章では，いくつかの推移閉包アルゴリズムを covert channel 検出に変形した．一般的に，推移閉包問アルゴリズムは covert channel 検出アルゴリズムに変形できる．このとき，推移閉包アルゴリズムの計算量が covert channel 検出アルゴリズムの計算量に大きく影響する．

第 6 章で，効率的な推移閉包アルゴリズムを用いた，covert channel 検出アルゴリズムを提案した．効率的な推移閉包アルゴリズムとして，Esko の STACK_TC[4] を用いた．この，STACK_TC は第 7 章で述べた区間表現を用いることで，より効率的に動作する．

第 8 章では，従来のアルゴリズムと提案アルゴリズムの実行速度をランダム 2 部グラフ $G(n, m, p)$ を用いて比較した．$p\sqrt{nm}$ が 1 を超えたあたりで，提案アルゴリズムは従来のアルゴリズムより有意に速かった．ただし，有意水準は 0.01 である．また，$1 < p\sqrt{nm}$ を満たすならば，nmp が大きいほど，すなわち辺数が大きい時，従来のアルゴリズムに比べて提案アルゴリズムはより効率的であった．更に，n が大きい時，すなわち object 数が大きい時，従来のアルゴリズムに比べて提案アルゴリズムはより効率的であった．したがって，辺数が大きいとき，特に object 数が大きい時は covert channel 検出に提案アルゴリズムを使用すべきである．

付録 A

ソースコード

ソースコードを添付する．プログラムは java で実装している．

ソースコード A.1　Vertex.java

```
1  import java.util.*;
2
3  public class Vertex {
4
5          public int id;
6          public List<Vertex> AbjFrom = new ArrayList<Vertex>();
7          public List<Vertex> CovertSucc = new ArrayList<Vertex>();
8
9          public Vertex(int id) { this.id = id; }
10
11 }
```

ソースコード A.2　Component.java

```
1  import java.util.*;
2
3  public class Component {
4
5          public int id;
6          public List<Vertex> capO = new ArrayList<Vertex>();
7          public List<Vertex> capS = new ArrayList<Vertex>();
8          public IntervalSet Succ = new IntervalSet();
9
10         public Component(int id) { this.id = id; }
11
12 }
```

ソースコード A.3　G_A.java

```
1  import java.util.*;
2
3  public class G_A {
```

```
4
5           public ArrayList<Vertex> O = null;
6           public ArrayList<Vertex> S = null;
7           public int E;
8
9           public G_A(int n, int m, double p) {
10                  if(n<1 || m<1) throw(new NullPointerException());
11
12                  O = new ArrayList<Vertex>(n);
13          for(int i=0; i<n; i++) O.add(new Vertex(i));
14                  S = new ArrayList<Vertex>(m);
15          for(int i=0; i<m; i++) S.add(new Vertex(n+i));
16
17          for(Vertex x : O){
18                  for(Vertex y : S){
19                          if(Math.random()<p) {x.AbjFrom.add(y); E++;}
20                              if(Math.random()<p) {y.AbjFrom.add(x); E++;}
21                  }
22          }
23          }
24  }
```

ソースコード A.4　IntervalSet.java

```
1  import java.util.*;
2
3  public class IntervalSet {
4
5          public List<Integer> set;
6
7          IntervalSet() {
8                  set = new ArrayList<Integer>();
9          }
10
11         public boolean contains(int i) {
12                 if(set.size() == 0) return false;
13                 for(int k=0; k+1<set.size(); k+=2){
14                         if(i < set.get(k)) return false;
15                         if(i <= set.get(k+1)) return true;
16                 }
17                 return false;
18         }
19
20         public void cupInt(int i) {
21                 if(set.size() == 0){
22                         set.add(i);
23                         set.add(i);
24                         return;
25                 }
26                 if(i < set.get(0)) {
27                         if(set.get(0)−1 == i) set.set(0, i);
28                         else{
29                                 set.add(0, i);
30                                 set.add(1, i);
31                         }
32                         return;
33                 }
34                 if(i <= set.get(1)) return;
35                 for(int k=2; k<set.size(); k+=2){
36                         if(i < set.get(k)) {
37                                 if(set.get(k−1)+1 == i){
38                                                 if(set.get(k)−1 == i){
39                                                         set.remove(k−1);
40                                                         set.remove(k−1);
41                                                 }
42                                                 else set.set(k−1,i);
43                                                 return;
44                                 }
45                                 else if(set.get(k)−1 == i) set.set(k,i);
46                                 else {
47                                         set.add(k,i);
48                                         set.add(k,i);
49                                 }
50                                 return;
```

```
51                      }
52                      if(i <= set.get(k+1)) return;
53              }
54              if(set.get(set.size()-1)+1 == i) set.set(set.size()-1,i);
55              else {
56                      set.add(i);
57                      set.add(i);
58              }
59              return;
60      }
61
62      public void cupList(List<Integer> S2) {
63              if(S2.size()==0) return;
64              if(set.size()==0){
65                      for(int k=0; k<S2.size(); k++) set.add(S2.get(k));
66                      return;
67              }
68
69              List<Integer> S1 = set;
70              set = new ArrayList<Integer>();
71              int k1=0, k2=0, k=0;
72              if(S1.get(0)<S2.get(0)) {
73                      set.add(S1.get(0));
74                      set.add(S1.get(1));
75                      k1+=2;
76              }
77              else {
78                      set.add(S2.get(0));
79                      set.add(S2.get(1));
80                      k2+=2;
81              }
82
83              while(k1+1<S1.size() && k2+1<S2.size()) {
84                      if(S1.get(k1)<S2.get(k2)) {
85                              if (set.get(k+1)+1 < S1.get(k1)) {
86                                      set.add(S1.get(k1));
87                                      set.add(S1.get(k1+1));
88                                      k+=2;
89                              }
90                              else if(set.get(k+1) < S1.get(k1+1)) {
91                                      set.set(k+1, S1.get(k1+1));
92                              }
93                              k1+=2;
94                      }
95                      else{
96                              if (set.get(k+1)+1 < S2.get(k2)) {
97                                      set.add(S2.get(k2));
98                                      set.add(S2.get(k2+1));
99                                      k+=2;
100
101                             }
```

```
102                                     else if(set.get(k+1) < S2.get(k2+1)) {
103                                             set.set(k+1, S2.get(k2+1));
104                                     }
105                                     k2+=2;
106                             }
107                     }
108                     if (!(k2+1<S2.size())){
109                             while(k1+1<S1.size()) {
110                                     if (set.get(k+1)+1 < S1.get(k1)) {
111                                             set.add(S1.get(k1));
112                                             set.add(S1.get(k1+1));
113                                             k+=2;
114                                     }
115                                     else if(set.get(k+1) < S1.get(k1+1)) {
116                                             set.set(k+1, S1.get(k1+1));
117                                     }
118                                     k1+=2;
119                             }
120                     }
121                     else if (!(k1+1<S1.size())){
122                             while(k2+1<S2.size()) {
123                                     if (set.get(k+1)+1 < S2.get(k2)) {
124                                             set.add(S2.get(k2));
125                                             set.add(S2.get(k2+1));
126                                             k+=2;
127                                     }
128                                     else if(set.get(k+1) < S2.get(k2+1)) {
129                                             set.set(k+1, S2.get(k2+1));
130                                     }
131                                     k2+=2;
132                             }
133                     }
134             }
135     }
```

ソースコード A.5　ObjectsBFS.java

```
1  import java.util.*;
2
3  public class ObjectsBFS {
4
5          private G_A g;
6          private List<Vertex> Q = new LinkedList<Vertex>();
7          protected BitSet VISITED;
8
9          public ObjectsBFS(G_A g) {
10                 this.g = g;
11                 this.VISITED = new BitSet(g.O.size()+g.S.size());
12         }
13
14     public void BFS(Vertex x) {
15         VISITED.clear();
16         VISITED.set(x.id);
17         for(Vertex y : x.AbjFrom) VISITED.set(y.id);
18
19         Q.addAll(x.AbjFrom);
20         while(Q.size() != 0){
21                 Vertex y = Q.remove(0);
22                 for(Vertex z : y.AbjFrom){
23                         if(!VISITED.get(z.id)){
24                                 VISITED.set(z.id);
25                                 Q.add(z);
26                                 if(g.O.size() <= z.id) x.CovertSucc.add(z);
27                         }
28                 }
29         }
30     }
31
32     public void analysis(){
33                 for(Vertex x : g.O) BFS(x);
34     }
35
36 }
```

ソースコード A.6　STACK_CC.java

```
1  import java.util.*;
2
3  public class STACK_CC {
4
5          protected int DFO = 0;
6          protected int N_c = 0;
7          protected List<Vertex> vstack;
8          protected List<Component> cstack;
9          protected G_A g;
10
11         public ArrayList<Component> component= new ArrayList<Component>();
12
13         protected Vertex[] Root;
14         protected Vertex[] BeFrom;
15         protected Component[] Comp;
16         protected int[] NumOfComp;
17         protected int[] SavedHight;
18         protected int[] DFOof;
19         protected int[] SearchIndex;
20         protected BitSet VISITED;
21
22
23         public STACK_CC(G_A g) {
24                 this.g = g;
25                 this.vstack = new LinkedList<Vertex>();
26                 this.cstack = new LinkedList<Component>();
27                 this.Root = new Vertex[g.O.size()+g.S.size()];
28                 this.BeFrom = new Vertex[g.O.size()+g.S.size()];
29                 this.Comp = new Component[g.O.size()+g.S.size()];
30                 this.NumOfComp = new int[g.S.size()];
31                 this.SavedHight = new int[g.O.size()+g.S.size()];
32                 this.DFOof = new int[g.O.size()+g.S.size()];
33                 this.SearchIndex = new int[g.O.size()+g.S.size()];
34                 this.VISITED = new BitSet(g.O.size()+g.S.size());
35         }
36
37     public void STACK_TC(Vertex x) {
38         List<Vertex> DFStack = new ArrayList<Vertex>();
39                 DFStack.add(x);
40         vstack.add(x);
41         Root[x.id] = x;
42         SavedHight[x.id] = cstack.size();
43                 DFOof[x.id] = DFO++;
44         SearchIndex[x.id] = 0;
45         VISITED.set(x.id);
46
47                 while(DFStack.size() != 0){
48                         Vertex y = DFStack.get(DFStack.size()−1);
49
50                         if(SearchIndex[y.id] < y.AbjFrom.size()){
```

```
51                          Vertex z = y.AbjFrom.get(SearchIndex[y.id]++);
52                          if(!VISITED.get(z.id)){
53                                  DFStack.add(z);
54                          vstack.add(z);
55                          Root[z.id] = z;
56                                  BeFrom[z.id] = y;
57                          SavedHight[z.id] = cstack.size();
58                                  DFOof[z.id] = DFO++;
59                          SearchIndex[z.id]= 0;
60                          VISITED.set(z.id);
61                          }
62                  }
63                  else{
64                          DFStack.remove(DFStack.size()-1);
65                          for(Vertex z : y.AbjFrom) {
66                                  if(Comp[z.id] == null){
67                                          if(DFOof[Root[z.id].id] < DFOof[Root[y
                                                .id].id]) Root[y.id] = Root[z.id];
68                                  }
69                          else if(DFOof[z.id] <= DFOof[y.id] || BeFrom[z.id] ==
                                y) {
70                                  cstack.add(Comp[z.id]);
71                          }
72                          }
73                          if(Root[y.id] == y) {
74                          Component C = new Component(N_c);
75                          component.add(N_c++, C);
76                          if(vstack.get(vstack.size()-1) != y) C.Succ.cupInt(C.id);
77
78                          while(cstack.size() != SavedHight[y.id]) {
79                                  Component X = cstack.remove(cstack.size()-1);
80                                  if(!C.Succ.contains(X.id)) {
81                                          C.Succ.cupInt(X.id);
82                                          C.Succ.cupList(X.Succ.set);
83                                  }
84                          }
85                          Vertex z = y;
86                          do{
87                                  z = vstack.remove(vstack.size()-1);
88                                  Comp[z.id] = C;
89                                  if(z.id < g.O.size()){
90                                          C.capO.add(z);
91                                  }
92                                  else{
93                                          NumOfComp[z.id-g.O.size()] = C.capS.
                                                size();
94                                          C.capS.add(z);
95                                  }
96                          } while(z != y);
97                  }
98                  }
```

```
99                  }
100     }
101
102     public void analysis(){
103                 for(Vertex x : g.O) if(!VISITED.get(x.id)) STACK_TC(x);
104                 for(Vertex x : g.S) if(!VISITED.get(x.id)) STACK_TC(x);
105
106                 for(Vertex x : g.O) {
107                         for(Vertex y : x.AbjFrom) Comp[y.id].capS.set(NumOfComp[y.
                                id-g.O.size()], null);
108                         List<Integer> X = Comp[x.id].Succ.set;
109                         if(X.size() != 0){
110                                 for(int i=0; i < X.size(); i+=2)
111                                         for(int j=X.get(i); j<=X.get(i+1); j++)
112                                                 for(Vertex y : component.get(j).capS)
113                                                         if(y != null)
114                                                                 x.CovertSucc.add(y);
115                         }
116                         for(Vertex y : x.AbjFrom) Comp[y.id].capS.set(NumOfComp[y.
                                id-g.O.size()], y);
117                 }
118     }
119 }
```

ソースコード A.7　JIKKEN138.java

```
1  import java.io.BufferedWriter;
2  import java.io.FileNotFoundException;
3  import java.io.FileWriter;
4  import java.io.IOException;
5  import java.util.*;
6
7  public class JIKKEN138 {
8          List<Integer>DATA = new ArrayList<Integer>();
9          long t1=0, t2=0;
10         int n = 0;
11         int m = 0;
12         double p = 0;
13         String folder = null;
14         G_A g;
15
16         JIKKEN138(String[] args){
17                 this.n = Integer.valueOf(args[0]);
18                 this.m = Integer.valueOf(args[1]);
19                 this.p = Double.valueOf(args[2]);
20                 this.folder = args[3];
21         }
22
23         public void create() {
24                 t1 = System.currentTimeMillis();
25                 g = new G_A(n, m, p);
26                 DATA.add(g.E);
27                 t2 = System.currentTimeMillis();
28                 DATA.add((int)(t2-t1));
29         }
30
31         public void analysis1() throws IOException {
32                 t1 = System.currentTimeMillis();
33                 STACK_CC ana1 = new STACK_CC(g);
34                 ana1.analysis();
35                 t2 = System.currentTimeMillis();
36                 DATA.add((int)(t2-t1));
37         }
38
39         public void analysis2() throws IOException {
40                 t1 = System.currentTimeMillis();
41                 ObjectsBFS ana1 = new ObjectsBFS(g);
42                 ana1.analysis();
43                 t2 = System.currentTimeMillis();
44                 DATA.add((int)(t2-t1));
45         }
46
47         public static void main(String[] args) {
48                 JIKKEN138 J = new JIKKEN138(args);
49                 int start = Integer.valueOf(args[4]);
50                 int stop = Integer.valueOf(args[5]);
```

```
51                  try{
52                          for(int i=start; i<=stop; i++) {
53                                  J.create();
54                                  J.analysis1();
55                                  for(Vertex x : J.g.O) x.CovertSucc = new ArrayList<
                                        Vertex>();
56                                  J.analysis2();
57                          }
58                          System.out.println("stop\txx:xx");
59                          BufferedWriter bw = new BufferedWriter(new FileWriter(args
                                [3] + "DATA.txt"));
60                          bw.write("引数" + "\tN1="+args[0] + "\tN2="+args[1] + "\tp=
                                "+args[2] + "\tfolder="+args[3] + "\tstart="+args[4] +
                                 "\tstop="+args[5] + "\n");
61                          bw.write("実辺数\t 生成時間[ms]\t 解析時間(STACK)[ms]\
                                t 解析時間(BFS)[ms]\n");
62                          for(int i=0; i<J.DATA.size(); i++){
63                                  bw.write(J.DATA.get(i) + "");
64                                  if((i+1)%4 == 0) bw.write("\n");
65                                  else bw.write("\t");
66                          }
67                          bw.close();
68                  } catch(FileNotFoundException e1) {
69                          e1.printStackTrace();
70                  } catch (IOException e1) {
71                          e1.printStackTrace();
72                  }
73          }
74  }
```

謝辞

本研究を進めるにあたって，御指導して下さった木下宏揚教授に感謝いたします．また，貴重な多くの助言，及び，御指導して下さったネッツエスアイ東洋株式会社の森住哲也氏，木下研究室特別助手鈴木一弘氏に感謝いたします．

最後に，神奈川大学木下研究室の諸兄にも感謝いたします．

2012 月 2 月

中村　峻生

参考文献

[1] 酒井剛典, 森住哲也, 畔上昭司, 小松充史, 稲積泰宏, 木下宏揚：“ Covert Channel 分析メカニズムと EJB による情報フィルタの構築 ”,2006 年暗号と情報セキュリティシンポジウム, (2006).

[2] 森住哲也, 木下宏揚：“ 社会システムの中の Covert Channel について ”, 技術と社会・倫理研究会, (2005).

[3] NCSC-TG-030, Covert Channel Analysis of Trusted Systems (Light Pink Book) from the United States Department of Defense (DoD) Rainbow Series publications,http://www.fas.org/irp/nsa/rainbow/tg030.htm

[4] Esko Nuutila, ”Efficient Transitive Closure Computation in Large Digraphs.”,ISBN 951-666-4512, ISSN 1237-2404, (1995).

[5] Bell D, LaPadula L, “ Secure Computer System: Mathematical Foundations and Model ”, MITRE report MTR 2547, (1973).

[6] David F.C. and Michael J. Nash, The Chinese Wall Security Policy, The Symposium on research in security and privacy, (1989), OAKLAND, CALIFORNIA, pp. 206-214.

[7] Sun Microsystems, INC., White paper “ RBAC in the Solaris Operating System ”, (2001).

[8] A.V.Aho, J.E. Hopcroft, and J.D.Ullman , ”The Design and Analysis of Computer Algorithms”, Addison-Wesley, Reading, Mass.(1974).

[9] D.E. Knuth. The Art of Computer Programming, Volume 1: ”Fundamental Algorithms”, Third Edition. Addison-Wesley, (1997). ISBN 0-201-89683-4. Section 1.2.11: Asymptotic Representations, pp.107-123.

[10] Reinhard Diestel：”Graph Theory, Springer-Verlag”,（2000）（根上生也、太田克弘訳、グラフ理論、シュプリンガー・フェアラーク東京（2000））

[11] Robin.J.Wilson, Introduction to Graph Theory, Longman Scientific & Technical, (1985).

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, MIT Press, (2001).

[13] Sara Baase, ”Computer Algorithms: Introduction to Design and Analysis” (アジソン・ウェスレイ・パブリッシャーズ・ジャパン株式会社発行，星雲者発売) ISBN:1998 (ISBN 4-7952-9720-7).

[14] M. E. Furman: Applications of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. Dokl. Akad. Nauk SSSR (in Russian), 194:524, (1970).

[15] D. Coppersmith and S. Winograd, “ Matrix multiplication via arithmetic progressions, ” Proceedings of the 19th ACM Symposium on Theory of Computing, (1987), pp. 1–6.

[16] 戸田　瑛人, ”クラウドの情報漏洩解析を高速に行うための MapReduce の適用”, 平成 21 年度修士論文

[17] H. Jakobson. Mixed-approach algorithms for transitive closure. In Proceedings of the 10th ACM Symposium on Principles of Database Systems, pages 199-205, (1991).

[18] Warshall, S . A theorem on Boolean matrices. J. ACM, 9, (1962), 11-12.

[19] P. Purdom, ”A Transitive Closure Algorithm”, BIT 10:76- 94, (1970).

[20] J. Eve and R. Kurki-Suomo, ”On Computing the Transitive Closure of a Relation”, Acta Informatica 8, (1977), 303-314. 10.

[21] J. Ebert. A sensitive transitive closure algorithm. Information Processing Let-. ters, 12:255-258, (1981).

[22] L. Schmitz. An improved transitive closure algorithm. Computing, 30:359-371,. (1983).

[23] R. E. Tarjan, Depth First Search and Linear Graph Algorithms, SIAM Journal on Computing, Vol. 1, No. 2, pp. 146-160, (1972).

[24] I. Munro, ”Efficient Determination of the Transitive Closure of a Directed Graph,” Information Processing Lett., Vol. 1, 56-58 (1971).

[25] M. J. Fischer, and A. R. Meyer, ”Boolean matrix multiplication and transitive closure”,. Conference Record IEEE 12th Annual Symposium on Switching and Automata Theory. (1971), 129-131.

[26] R. Agrawal, A. Borgida and H. V. Jagadish, ”Efficient Management of Transitive Relationships in Large Data and Knowledge Bases”, AT&T Bell Laboratories Technical Memorandum, Murray Hill, New Jersey, 1989. 5.

[27] R. Agrawal and H.V. Jagadish. Materialization and incremental update of path information. In Proceedings of the 5th International Conference on Data Engineering, LosAngeles, CA, February 1989.

[28] T. Ibaraki and N. Katoh, "On-Line Computation of Transitive Closures of Graphs", -presented at Inf. Process. Lett., 1983, pp.95-97.

[29] K. Simon. "An improved algorithm for transitive closure on acyclic digraphs". Theoretical Computer Science, 58:325-346, 1988.

[30] S. Dar and H.V. Jagadish, "A Spanning Tree Transitive Closure Algorithm," Proc. Eighth Int"l Conf. Data Eng., pp. 2-11,Phoenix, Ariz., 1992.

[31] H. Jakobson. Mixed-approach algorithms for transitive closure. In Proceedings of the 10th ACM Symposium on Principles of Database Systems, pages 199-205, 1991.

[32] H. Jakobson. On tree-based techniques for query evaluation. In Proceedings of the 11th ACM Symposium on Principles of Database Systems, pages 380-392, San Diego, California, 1992.

[33] G. Italiano. Amortized e ciency of a path retrieval data structure. Theoretical Computer Science, 48:273-281, 1986.

[34] G. Italiano. Finding paths and deleting edges in directed acyclic graphs. Information Processing Letters, 28(1):5-11, 1988.

[35] A.L. Buchsbaum, P.C. Kannelakis, and J.S. Vitter. A data structure for arc insertionand regular path nding. In Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, pages 22-31, 1990.

[36] J.A. La Peutré and J. van Leeuwen. Maintenance of transitive closures and transitivereductions of graphs. In H. Gottler and H.J. Schneider, editors, Graph-Theoretic Concepts in Computer Science, volume 314 of Lecture Notes in Computer Science, pages 106-120, Berlin, 1988. Springer-Verlag.

[37] D.M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. Acta Informatica, 30:369-384, 1993.

[38] P.-A. Larson and V. Deshpande. A le structure supporting traversal recursion. In Proceedings of the ACM-SIGMOD 1989 Conference on Management of Data, pages 243-252, 1989.

[39] R. Agrawal and J. Kiernan. An access structure for generalized transitive closure queries.In Proceedings of the IEEE 9th International Conference on Data Engineering, pages429-438, Vienna, Austria, April 1993.

[40] H.V. Jagadish. A compression technique to materialize transitive closure. ACM Transactions on Database Systems, 15(4), December 1990.

[41] K-H. Guh, C. Sun, and C. Yu. Real time retrieval and update of materialized transitiveclosure. In Proceedings of the IEEE 7th International Conference on Data Engineering,pages 690-697, Kobe, Japan, April 1991.

[42] K-H. Guh and C. Yu. E cient management of materialized generalized transitive closurein centralized and parallel environments. IEEE Transactions on Knowledge and Data Engineering, 4(4):371-381, August 1992.

[43] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. Data Structures and Algorithms. Addison-Wesley, Reading, Mass., 1983.

[44] William H.; Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery (1992). Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press. pp. p. 616

質疑応答

Q1:**松澤教授**　現実の ACL は完全なランダムではないと思われるが，ランダムグラフによる実験は意味があるのか．

A1:　確かに現実の ACL は完全なランダムではないので，ランダムグラフによる ACL の表現は完全とは言えない．今回は現実の ACL が手に入らなかったので一般性の高いランダムグラフでシミュレーション実験を行った．現実の ACL がランダムではなく何らかの特徴的な良い性質を持っているのであれば，その性質を生かしたヒューリセティックなアルゴリズムの提案は今後の課題である．

Q2:**豊嶋教授**　良い結果が得られたことはわかるが，提案がわかりにくい．

A2:　はい．