クラウド環境における
効率的なグラフクエリ処理に関する研究

提出先　　大阪大学大学院情報科学研究科

提出年月　2019 年 1 月

新 井　淳 也

関連発表論文

1. 学会論文誌発表論文

- 新井淳也, 塩川浩昭, 山室健, 鬼塚真, 岩村相哲. 効率的なグラフ分析のための実行時並列リオーダリング. 電子情報通信学会和文論文誌 *D*, Vol. J102-D, No. 4, 2019 (to appear).
- 新井淳也, 鬼塚真, 塩川浩昭. クラスタリングと空間分割の併用による効率的な k-匿名化. 日本データベース学会和文論文誌, Vol. 13-J, No. 1, pp. 72–77, 2014.

2. 国際会議発表論文

- Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium*, pp. 22–31, 2016.

3. 国内会議発表論文 (査読なし)

- 新井淳也, 鬼塚真, 藤原靖宏, 岩村相哲. 探索失敗履歴を用いた高速サブグラフマッチング. 第 10 回データ工学と情報マネジメントに関するフォーラム論文集, 2018.
- 伊藤竜一, 新井淳也, 佐々木勇和, 鬼塚真. 並列グラフ処理エンジンに対す

るグラフ圧縮と頂点順序最適化の適用. 第 9 回データ工学と情報マネジメントに関するフォーラム論文集, 2017. [優秀インタラクティブ賞]

- 新井淳也, 藤原靖宏, 鬼塚真, 岩村相哲. データグラフ上の並列枝刈りによるサブグラフマッチング. 第 9 回データ工学と情報マネジメントに関するフォーラム論文集, 2017.
- 新井淳也, 塩川浩昭, 山室健, 鬼塚真. 頂点順序の最適化によるスケーラブルなグラフ並列処理. 第 7 回データ工学と情報マネジメントに関するフォーラム論文集, 2015. [優秀インタラクティブ賞]
- 新井淳也, 鬼塚真, 塩川浩昭. クラスタリングと空間分割の併用による効率的な k-匿名化. 第 6 回データ工学と情報マネジメントに関するフォーラム論文集, 2014.

内容梗概

情報化の進展に伴い，実世界に設置されたセンサや人々の電子的な活動はますます多くのデータを生み出すようになった．このようなデータはビッグデータと呼ばれ，その管理と活用は学術的にも産業的にも重要視されている．ビッグデータは多種多様な情報を含むことから，高い柔軟性を持つグラフ表現がしばしば利用される．グラフに対するクエリ処理ではグラフデータに対して不規則なランダムアクセスが発生するため，大規模なグラフを利用する際は大容量のメモリを搭載した計算機の使用が望ましい．そのような計算機の導入は高コストであったが，近年はパブリッククラウドのサービスが充実し大容量メモリの計算環境を安価に利用できるようになった．ますます大規模化していくグラフデータに対応するためにクラウド環境の利用は重要である．以上のような背景から，本研究ではクラウド環境におけるグラフクエリ処理に焦点を当てる．

クラウド環境においてはパーソナルデータに関するリスクと規制に配慮する必要がある．ビッグデータはしばしば個人に関する情報を含む．そのような情報をオンラインのクラウド環境に保存することは情報流出のリスクを高める．また法的にも，特定の条件を満たさない限りクラウドへのパーソナルデータのアップロードは禁止されている．そこで本研究では次のように匿名化を通じたクラウド利用の枠組みを考える．データ所有者は自身の計算環境においてパーソナルデータを匿名化し，それをクラウドへアップロードする．クラウドには高速なクエリ処理エンジンを配備し，匿名化されたデータに基づいてクエリを処理する．

この枠組みを大規模なデータに対して適用するためには高速なクエリ処理エンジンと匿名化プログラムが必要である．しかし高速なクエリ処理エンジンの実現

には次の 2 つの課題がある．1 つ目に，グラフは事物間の複雑なつながりを表現可能である反面，そのつながりを辿るために局所性の低い不規則なメモリアクセスを生じる．これにより CPU のキャッシュヒット率が低下し，メモリのレイテンシによって処理効率が低下する．2 つ目に，最も典型的なグラフクエリであるグラフパターンの検索，即ちサブグラフマッチングは，組み合わせ爆発によって途方もなく長い計算時間を要する場合がある．さらに匿名化にも課題がある．匿名化においては入力データと処理後のデータの差分，即ち情報損失を小さく抑えることが重要である．また同時に，ますます大規模化するデータに対応するためには高速に匿名化可能であることが求められる．しかしながら，情報損失を低減するためには全体最適となるように差分を加える必要があり，高速な処理との両立が難しい．

そこで，本研究は 3 つのアルゴリズムの提案を通じ，高速なクエリ処理と匿名化による効率的なグラフクエリ処理を可能にする．本論文は全 5 章から構成される．第 1 章では研究の目的と提案手法の全体像を説明する．

第 2 章ではグラフ処理に共通の問題であるメモリアクセス局所性の低さを改善するため，新しいリオーダリングアルゴリズムを提案する．リオーダリングとはグラフデータに含まれる頂点の順序 (頂点 ID 番号) を最適化することでグラフ処理の局所性を向上させる前処理である．従来，リオーダリングには長い時間を要し，リオーダリングとグラフ処理を合わせた全体の処理時間はむしろ増加してしまうケースが多く存在した．これに対し，提案手法はグラフのクラスタ構造を効率的な並列アルゴリズムで捉えることにより，局所性の高い頂点順序を短時間で生成する．実験では提案手法がリオーダリングを含めた全体の処理時間の短縮に効果があることを確認した．

第 3 章では不要な探索を枝刈りする効率的なサブグラフマッチングアルゴリズムを提案する．既存のアルゴリズムはグラフ構造を分析する前処理によって枝刈り可能な探索範囲を見極め，クエリ処理を高速化してきた．しかし探索開始前に実施可能な枝刈りでは削減できない探索範囲が広く存在する．提案手法では探索開始後に得られる情報を用いた枝刈りによってクエリ処理を高速化する．アイデアは「失敗から学ぶ」ことである．提案手法は探索によって検索条件にヒットし

ないことが判明 (探索失敗) した探索範囲を枝刈り対象として順次追加する．これにより失敗の繰り返しを避ける．さらに実験で提案手法の有効性を確認した．

第 4 章では代表的な匿名化手法である k-匿名化を行うためのアルゴリズムを提案する．情報損失の小さい k-匿名化のためには，k 個以上の互いに似通ったデータ点から成るグループへデータ全体を分割する処理が必要となる．提案手法は最近傍グラフを用いることで近傍に存在するデータ点を効率的に発見しグループ化することで情報損失を抑える．さらに空間分割による粗粒度のグループ化を組み合わせることによって小さい情報損失を保ちつつ処理を高速化する．実世界のデータを用いた性能評価によって提案手法は既存の k-匿名化手法よりも低情報損失かつ高速であることを確認した．

最後に第 5 章では本研究の成果を要約し全体のまとめを行うとともに，今後の研究課題について述べる．

目次

第 1 章

序章

1.1　研究の背景と目的

情報処理技術の発達は実世界のセンサ情報や人々の電子的な活動の記録から成る大規模データ，即ちいわゆるビッグデータの収集を可能にした．ビッグデータに含まれる情報は多様である．その中には関係データベースが扱うような表形式の構造化データとして表現することが適さない情報も多く含まれる．そのような情報を表現するための代表的なデータ構造としてグラフがある．グラフは頂点および頂点間を接続するエッジという 2 つの要素から成る．頂点によって事物を表現し，エッジによって事物間の関係を表現することによって，グラフは様々な情報を事物間のつながりという観点から柔軟に表現することができる．さらに頂点とエッジにそれらが表現する事物や関係の種類，値などの付加情報 (プロパティ) を与える場合がある．そのようなグラフは**プロパティグラフ**と呼ばれ，高い表現力と柔軟性を持つことから幅広く利用されている [31, 45, 76]．図 1.1 にソーシャルネットワーキングサービス (SNS) 上の情報をイメージしたプロパティグラフの例を示す．他にグラフ表現が適している情報の例としては，Web ページ間のハイパーリンク，金融取引，道路網などが挙げられる．これらのグラフは様々な用途で利用されている．例えばハイパーリンクのつながりを加味することによる Web 検索の品質向上 [75]，金融取引の分析による株価操作の発見 [65]，道路網グラフに基づいた道案内などがグラフの利用によって可能になる．このようにグ

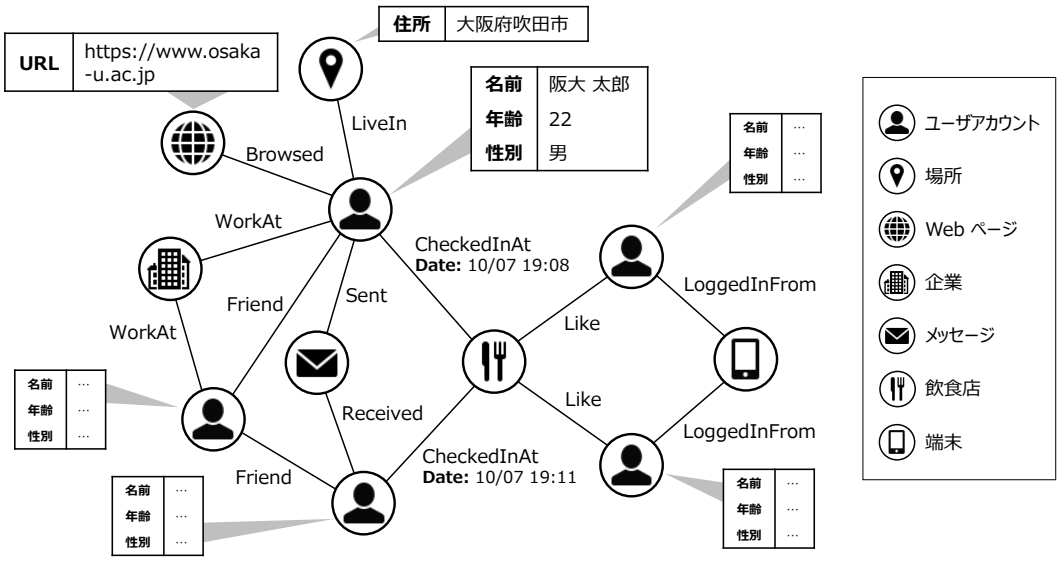
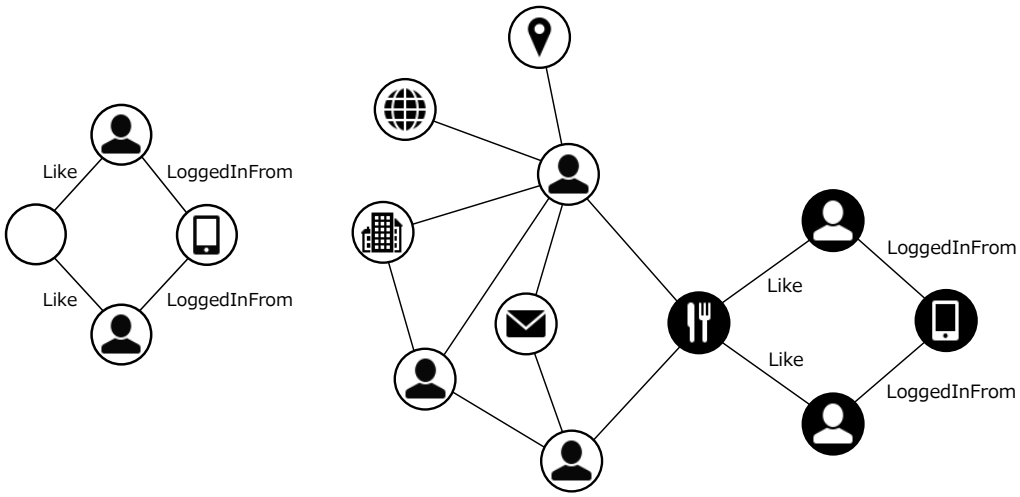


図 1.1: プロパティグラフの例．ユーザアカウントに名前，年齢，性別などのプロパティが付与されている．同様に CheckedInAt を表すエッジは日付 (Date) のプロパティを持つ．

ラフは多様な情報を活用していくために重要な表現形式となっている．

情報を活用するためにはクエリ処理によってグラフに格納された情報を取り出す必要がある．表形式データに対して SQL の SELECT 文と WHERE 句を用いるのと同様に，グラフに対する最も典型的なクエリの一つとして特定のサブグラフ構造の検索が用いられる．具体的には，グラフクエリ処理では検索条件を表現する**クエリグラフ**を受け取り，大きいグラフ (**データグラフ**) 中に存在する同型なサブグラフへの対応関係 (**埋め込み**) を列挙する．アルゴリズムの観点ではこのような列挙を行う問題は**サブグラフマッチング**とも呼ばれる．サブグラフマッチングに基づくクエリはグラフ向けデータベースにおいて代表的なクエリ言語である SPARQL, Cypher, Gremlin などによって基本的なクエリとしてサポートされている [90]．さらにデータ分析の分野においても，人間関係の分析 [31]，営業活動の支援 [45]，マルウェアの検出 [76] など様々なアプリケーションにサブグラフマッチングは利用されている．アプリケーションの具体例として図 1.1 のグラフからサクラ行為を発見するためのクエリを図 1.2(a) に示す．同一端末か



(a) クエリグラフ (b) データグラフ

図 1.2: クエリグラフおよび同型なサブグラフを含むデータグラフの例．データグラフ中の色が反転されている部分はクエリグラフと同型なサブグラフを示す．

らログインするユーザアカウントは同一人物であると考えられる．そのため，端末を共有する複数アカウントが同じ飲食店などに高評価 (Like) を与えている場合はサクラである可能性が高い．そのような行為を図 1.2(a) のクエリグラフによって発見することができる．図 1.2(b) のデータグラフでは，このクエリグラフと同型なサブグラフを色の反転によって示している．このようにグラフに対するクエリは様々に利用され，現代の計算システムにおいて最も重要な処理の一つとなっている．

クエリ処理ではスムーズな対話的操作のため速やかに結果を返却することが求められる．効率的なグラフクエリ処理のためにはグラフ全体のデータをメモリ上にロードしておくことが望ましい．なぜならば，グラフは頂点間の複雑なつながりを表現可能である一方で，そのようなつながりを計算機上で辿る際にグラフデータに対する複雑なランダムアクセスを生じるためである [70]．高速なストレージとして使用されるソリッドステートドライブ (SSD) であってもメモリと比較するとレイテンシは約 1000 倍である [40] ことから，メモリ上で処理可能か

どうかは性能に致命的な影響を与える．しかしながらグラフデータの大規模化に伴い，一般的な計算機でグラフデータをメモリに格納することは困難になりつつある．例えば Web グラフはそれぞれ Web ページとハイパーリンクを表現する 500 億の頂点と 1 兆のエッジを含む [62]．また Facebook のソーシャルグラフはそれぞれユーザと友人関係を表現する 10 億の頂点と 2,000 億のエッジを含む [62]．これらの頂点とエッジにプロパティ情報が付加されることでグラフのデータサイズはますます増大する．複数の計算機のメモリにグラフを分割して格納する分散処理フレームワークも提案されている [18, 39, 69] が，通信や並行性制御のオーバーヘッドにより単一の計算機よりも性能は低下しがちであることが知られている [72]．そのため，大規模グラフの処理は大容量のメモリを搭載した計算機を必要とする．

そのような計算機の購入には高いコストがかかることから，近年ではパブリッククラウドが広く利用されている．クラウドでは必要に応じて様々な性能の計算機を安価に利用することができる．例えば最大手クラウドベンダーの一つである Amazon Web Services (AWS) は 2019 年 1 月時点で 12TB のメモリを搭載したインスタンス (仮想マシン) を提供している[*1]．このようなクラウド上の計算資源を使用することで大規模なグラフもインメモリで処理可能である．

以上のような背景から，本研究ではクラウド環境における効率的なグラフクエリ処理の実現を目指し，主要な技術的課題を解決する要素技術の確立に取り組む．

1.2　クラウド環境におけるグラフクエリ処理の枠組み

クラウド環境の利用はメモリ容量に関する問題の解決につながる．しかし同時に，クラウド上で個人に関する情報，いわゆる**パーソナルデータ** [100] を扱う際にはリスクと規制があることに注意する必要がある．人々の活動から生じるビッグデータは必然的にパーソナルデータを含む．グラフは頂点間の接続関係あるいは頂点とエッジのプロパティとしてパーソナルデータを含む場合がある．例えば

[*1] https://aws.amazon.com/jp/ec2/instance-types/high-memory/

グラフとして表現された交友関係，購買履歴，移動履歴，金融取引はいずれもパーソナルデータである．個人のプライバシを守るため，このような情報は流出しないよう細心の注意を払う必要がある．しかしながらオンライン環境であるクラウドには，セキュリティホール，アカウント情報の漏洩，オペレーションミス，あるいはクラウド事業者による目的外利用などによる情報流出のリスクが存在する．さらにパーソナルデータの利用は各国の法律で規制されている [101]．日本においても，データを提供する個人の合意など所定の条件が満たされない限りクラウド環境へパーソナルデータをアップロードすることは禁止されている [100]．

そこで本研究では，クラウド環境の利用を可能にするために**匿名化**を用いる．匿名化とは個人の特定および元の個人情報の復元ができないようパーソナルデータを加工する処理である．匿名化されたデータは**匿名加工情報**と呼ばれる [100]．匿名加工情報の利用には 2 つの利点がある．まず，仮にクラウド上のデータが流出したとしても，そのデータが匿名加工情報であればプライバシに関する影響を低減できる [82]．さらに，匿名加工情報は法的にもその利用に高い自由度が認められている．例えば，日本では本人の同意を必要とせずに匿名加工情報の目的外利用や第三者への提供が可能である [100]．また EU の一般データ保護規則 (General Data Protection Regulation) において匿名加工情報はパーソナルデータとみなされない[*2]．

匿名化を通じてクラウド環境上でクエリ処理を行うために，本研究では次のような枠組みを考える．クエリ処理エンジンと匿名化プログラムを作成し，クエリ処理エンジンをクラウド上に，匿名化プログラムをデータ所有者の計算環境に配備する．データ所有者は自身の計算環境でパーソナルデータを含むグラフを匿名化し，匿名化されたグラフをクラウドへアップロードする．クエリのリクエストはクラウド上のクエリ処理エンジンに対して行い，クエリ処理エンジンは匿名化されたグラフに基づいて応答する．このような枠組みを用いることにより，パーソナルデータにまつわる障壁をクリアしつつ効率的なクエリ処理が可能になる．

[*2] 前文第 26 項および https://ec.europa.eu/info/law/law-topic/data-protection/reform/what-personal-data_en

1.3 解決すべき課題

匿名化を伴うクラウド利用の枠組みはクエリ処理エンジンと匿名化プログラムの存在を前提とする．大規模なデータを扱うためにはどちらも高速な処理が求められるが，その実現には技術的な課題が存在する．本節ではそれらの課題について述べる．

1.3.1 クエリ処理に関する課題

グラフは事物間の複雑なつながりを表現するため，その構造に対するクエリ処理は時間を要する．処理時間は CPU の命令処理効率と発行命令数によって決まるが，グラフクエリ処理では次に述べるような 2 つの要因がそれぞれ命令処理効率の低下と発行命令数の増加を引き起こしている．

低い局所性

前述のように，サブグラフマッチングを含め，グラフ処理ではグラフデータに対する局所性の低い不規則なランダムアクセスが生じる．本研究はクラウド環境の利用によるインメモリ処理を前提としているため，ストレージへのアクセスを考慮する必要はない．しかしながら，予測困難な不規則アクセスによる CPU のキャッシュヒット率低下という問題が依然として存在する．キャッシュミス時にはメモリアクセスが発生し，その間 CPU は空転 (ストール) するため命令処理効率が低下する．具体的にはグラフ処理において CPU が計算を行っている時間は 10〜45% で，残りはキャッシュストールによって消費されているという報告がある [92]．このように局所性の低いメモリアクセスによってグラフ処理のコストが増大する．

無駄な探索

サブグラフマッチングではデータグラフに含まれるクエリグラフの埋め込みを列挙する．埋め込みはクエリグラフの頂点をデータグラフの頂点へ 1 つず

つ割り当てる手続きを再帰的に適用することで探索される．このとき埋め込みが存在しない箇所に対する無駄な探索が継続されてしまう場合がある．探索では頂点の割り当て方の組み合わせを生成するため，適切に探索が枝刈りされなければ組み合わせ爆発によって極めて長い処理時間を要する．そこで無駄な探索を早期に枝刈りするためのヒューリスティクスがこれまでよく研究されてきた [7, 20, 42, 43, 81, 83, 97]．しかし近年提案された手法群にもそれぞれ苦手とするクエリ形状が存在し，それらの探索に要する時間がクエリ処理のスループットを低下させる支配的要因となることが指摘されている [53]．このように，無駄な探索は依然としてクエリの処理コスト増大の原因となっている．

1.3.2 匿名化に関する課題

クエリ処理のみならず匿名化もまた高い計算コストを要する．匿名化はグラフから人物の名前や ID を除去するのみでは達成できない．なぜならば年齢，性別，居住地などの情報を組み合わせることで個人を特定できてしまう場合があるためである．従って匿名化のためには適切にデータを書き換える必要がある．代表的な匿名化のアプローチとして k-匿名化 [88] がある．k-匿名化はある個人が他の $k-1$ 人以上と区別できないようにデータを加工することで個人の特定を困難にする手法である．例えば k 人以上が同じ情報を持つように年齢などのデータを平均値で置き換える方法がある [26]．このような加工は元データからの乖離，即ち情報損失を生じる．そこで元々似通った情報を持つ k 人以上をグループ化し，そこに含まれる人物を同じ情報で置き換えることにより情報損失を低減する．つまり，k-匿名化ではデータに含まれる人物を k 人以上のグループへと分割する必要がある．本研究の枠組みにおいて匿名化はデータ所有者の計算環境で行われることから，一般的な性能の計算機上で分割可能でなければならない．しかしながら，情報損失が小さくなるように全体最適を目指して分割することは難しい問題である．人物の数 n に対して計算量 $O(n \log n)$ の高速なアルゴリズムが提案されている [61] が，その情報損失は計算量 $O(n^2)$ のアルゴリズムよりも大幅に大きいことが知られている [12]．つまり，既存手法では小さい情報損失と高速な処

理のどちらか一方を選択しなければならず，両立されていない．

1.4 提案技術の概要

以上のような課題を解決するため，本研究では 3 つのアルゴリズムを提案する．それぞれ，(i) 局所性の向上，(ii) 無駄な探索の削減，(iii) 低情報損失かつ高速な匿名化によって 1.3 節で述べた課題を解決する．以下に各アルゴリズムの概要を説明する．

1.4.1 実行時リオーダリング

グラフ処理におけるメモリアクセスの局所性を向上させる方法としてリオーダリングが広く用いられてきた．リオーダリングとは頂点の並び順 (頂点 ID の割り当て) の変更を通じてグラフ処理時のデータ配置とメモリアクセスを最適化する手法である．リオーダリングはサブグラフマッチングを含む様々なグラフ処理に効果があり，さらにグラフ処理プログラムの実装を変更する必要がないという利点がある．しかし一方で，既存のリオーダリングアルゴリズムは効果的な頂点順序の生成に長い時間を要する．そのため，グラフ処理の実行時に前処理としてリオーダリングを行うと，リオーダリングと後続のグラフ処理を合計した全体の処理時間をむしろ増加させてしまう．クエリ処理ではクエリと無関係な頂点やエッジを除外して処理する場合があるため，クエリが投入されてからリオーダリングする必要がある．また実世界の事象を反映することでグラフは時々刻々と変化するため，グラフ分析においても実行時にその都度リオーダリングを行うことが望ましい．このような場合に既存手法で性能を向上させることは難しい．

全体の処理時間を短縮するためには高い局所性を示す頂点順序を短時間で生成しなければならない．そこで提案手法ではグラフに含まれるコミュニティ構造に着目する．コミュニティとはエッジで密に接続された頂点の集合である．Web グラフやソーシャルグラフなどいわゆる実世界のグラフはコミュニティ構造を含むことが知られている [63]．密に接続された頂点はグラフ処理時に互いにデータを参照し合うため，提案手法はそれらをメモリ上で近傍に配置させることにより

局所性を高める．さらに提案手法は逐次集約法 [85] を基にした高速な並列アルゴリズムによってコミュニティ検出を行う．これにより局所性の高さと短時間でのリオーダリングを両立し，リオーダリング時間を含めた全体の処理時間を短縮する．

1.4.2　高速サブグラフマッチング

1.3.1 節で述べたように，サブグラフマッチングを高速化するためには無駄な探索の枝刈りが重要である．既存のサブグラフマッチングアルゴリズムはクエリグラフとデータグラフの構造を分析する前処理に基づいて埋め込みが存在しない探索範囲を見極め，それによって無駄な探索を削減してきた．例えば，クエリグラフの頂点を u，データグラフの頂点を v とする．もし u の次数が v よりも大きければ，u を v に割り当てた後に探索を継続しても埋め込みを発見することはできない [89]．そのため u を v に割り当てた場合についての探索は枝刈りしてよい．しかし，埋め込みが存在する探索範囲を除外してしまうことがないよう，探索の枝刈りは慎重に行わなければならない．探索の開始前に得られる情報のみでは枝刈り可能と判定できる範囲は限定的である．

そこで提案手法は探索の最中に情報を得ながら徐々に探索範囲を削減する．提案手法のアイデアは「失敗から学ぶ」ことである．ここで失敗とは埋め込みを発見することができなかった探索を意味する．失敗が発生した際，提案手法は探索が失敗する条件を導出する．具体的には，失敗の原因となったクエリ頂点とデータ頂点の割り当てを特定し，それを失敗パターンとして記録する．そして記録されている失敗パターンに合致する頂点割り当てが以降の探索において生成された場合，その探索を枝刈りする．失敗パターンの抽出ルールは注意深く設計されており，提案手法は既存手法同様全ての埋め込みを発見することができる．このような枝刈りによってサブグラフマッチングを高速化する．

1.4.3 最近傍グラフを用いた k-匿名化

k-匿名化のための情報損失が少ないグループ化手法としてはこれまでクラスタリングに基づく手法が提案されてきた [86]．しかし既存手法 [86] はデータの類似した k 人を貪欲にグループ化することから，処理の終わりが近づくにつれ互いにデータの類似しない k 人が 1 つのグループを形成するようになる．その結果情報損失が増大するという課題がある．さらに人物の数 n に対し $O(n^2)$ の時間計算量を示すため，大規模データに対する適用が難しい．

そこで提案手法では，情報損失と計算量をそれぞれ次のようなアプローチで低減する．まず情報損失を低減するため，最近傍グラフの構築によってクラスタを考慮したグループ化を行う．具体的には，まず各人物の類似度に基づき人物同士を接続した最近傍グラフを構築する．次に，既存手法と同様にデータの類似した人物を貪欲に収集しグループ化する．その際，クラスタを考慮して 1 グループに含める人数を変化させる．これによってより全体最適に近いグループの作成が可能になる．

最近傍グラフの構築には計算量 $O(n^2)$ を要するが，提案手法では空間分割を組み合わせることで計算量を低減する．具体的には，kd-tree [33] を用いた粗粒度の分割を行う．k-匿名化において k は 5 などの小さい値である [29] ため，データ全体を考慮せず，類似度の高い人物のみを考慮しミクロな観点でグループを作成した場合も情報損失に影響しにくい．そのため，粗粒度の空間分割によって k 人よりも多いグループへ分割し，その内部のみ考慮しながら k 人以上からなる細粒度のグループへ分割するという 2 段階の処理を行う．kd-tree による空間分割は計算量 $O(n \log n)$ であるから，途中まで空間分割を使用することでグループの作成に要する計算量は $O(n^2)$ よりも小さくなる．これにより，小さい情報損失と高速な匿名化を両立する．

1.5　本論文の構成

本論文は全 5 章から構成され，本章以降の内容は次のとおりである．第 2 章と第 3 章は高速なクエリ処理のためのアルゴリズムを説明する．第 2 章ではメモリアクセスの局所性を向上させるための実行時リオーダリングアルゴリズムを，第 3 章では不要な探索の枝刈りによる高速サブグラフマッチングアルゴリズムをそれぞれ説明する．第 4 章では最近傍グラフの利用によって情報損失の少ない k-匿名化を高速に行うアルゴリズムを説明する．最後に第 5 章では本研究の成果を要約し全体のまとめを行うとともに，今後の研究課題について述べる．

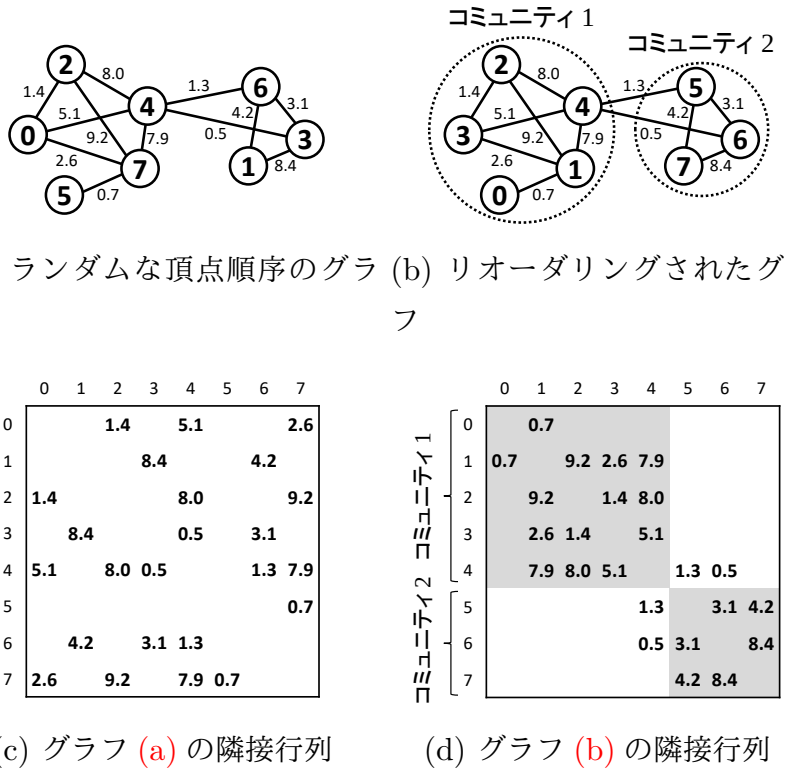
第 2 章

実行時リオーダリング

2.1 序論

1.1 節で述べたように，いくつかのクラウドベンダーは大容量メモリ環境を提供している．そのような環境では大規模なグラフをメモリに格納することができるが，現代の CPU の処理速度と比較するとメモリへのアクセスに要する時間は依然として極めて長い．メモリへのアクセスが発生すると CPU はストール状態となり，命令処理効率が低下する．このような問題を緩和するため，CPU は高速にアクセス可能なキャッシュメモリを備えていることが一般的である．効率的なグラフクエリ処理を実現するためには，局所性の高いメモリアクセスを行い CPU に内蔵されたキャッシュを効果的に利用する必要がある．

ところが，一般にグラフ処理は局所性の低いメモリアクセスを行い，それに伴ってキャッシュミスが頻発することが知られている [70, 92]．実世界のグラフは事物間の非構造的で複雑なつながりを表現しているため，その処理においても不規則なアクセスパターンを生じる．例えば SNS のソーシャルグラフを処理する際，ユーザ ID をキーとしてユーザの情報を保存することが自然である．一方で友人をユーザ ID に基づいて選ぶとは考え難いことから，各ユーザは見かけ



(a) ランダムな頂点順序のグラフ　(b) リオーダリングされたグラフ

(c) グラフ (a) の隣接行列　(d) グラフ (b) の隣接行列

図 2.1: 重み付きグラフとその隣接行列の例．グラフ (b) の点線はコミュニティ構造を示す．

上ランダムな ID のユーザと友人関係にある．従ってあるユーザが与えられたとき，その友人，即ち隣接頂点の情報を取得する際にはメモリへのランダムなアクセスが生じる．グラフは事物間のつながりに着目するデータ表現であることから，このように隣接頂点間のつながりを捉えるためのメモリアクセスはほぼ全てのグラフ処理において見られる．

グラフクエリ処理もまた局所性の低いメモリアクセスを行う．しかしグラフクエリ処理は様々なアルゴリズムが存在しそれぞれ異なる複雑なメモリアクセスを行うため，モデル化が難しい．そこでここでは単純かつよく知られたアルゴリズムとして PageRank に焦点を当てる．例として PageRank が図 2.1(a) に示すグラフにおいて行うメモリアクセスを考える．各頂点のデータは配列に格納されているものとする．先に述べた SNS の例と同様，このグラフは頂点 ID と無関係

な接続関係を持つ．PageRank は頂点を順に選択しその周辺のつながりを調査する．具体的には最初に頂点 0 とその隣接頂点である頂点 2, 4, 7 がアクセスされ，次に頂点 1 とその隣接頂点である 3, 6 がアクセスされる．この時点までで頂点 0, 2, 4, 7, 1, 3, 6 に対応する頂点データが順にアクセスされたことになるが，このようなアクセスはキャッシュミスを多発させる．なぜならばここまで各頂点へのアクセスは一度きりであり，さらにメモリ上で互いに離れた要素にアクセスしているためである．このことはそれぞれ時間的局所性と空間的局所性が低いことを意味する．キャッシュミスはメモリアクセスのための遅延を発生させるのみならず，マルチコア CPU におけるコア間通信やメモリ帯域の飽和によって並列グラフ処理のスケーラビリティに深刻な影響をもたらす [70]．

局所性を向上させるため，さまざまなアプリケーションにおいて**リオーダリング**アルゴリズムが使用されている [17, 32, 36, 56, 78]．リオーダリングによって頂点の並び順，即ち頂点 ID の割り当てを変更することにより，グラフ処理アルゴリズムの実装を変更することなくメモリアクセスの局所性を向上させることができる．具体的には隣接頂点同士に近い ID 番号を与えることで，処理アルゴリズムは直前の処理でアクセスされた頂点に繰り返しアクセスするようになり (高い時間的局所性)，さらにそのデータはメモリ上で近傍に存在するようになる (高い空間的局所性)．図 2.1(a) のグラフに対するリオーダリング結果の例を図 2.1(b) に示す．このグラフにおいて頂点 0 と 1 およびぞれぞれに隣接する接続関係を調べるとき，頂点 0, 1, 1, 0, 2, 3, 4 が順番にアクセスされる．この場合頂点 0 と 1 が複数回アクセスされており，またアクセス先である頂点 0–4 はメモリ上の連続領域に格納されているため，時間的・空間的局所性が向上している．実際に，既存研究 [32, 66, 79] では前もってグラフをリオーダリングしておくことによって PageRank と幅優先探索 (BFS) を高速化している．

しかしながら，効果的な頂点順序を生成するためには長い計算時間を要する．そのため，リオーダリングとグラフ処理を合わせた全体の処理時間はリオーダリングを用いない場合よりもむしろ増加してしまう場合がある．グラフデータを前もって入手し，事前にリオーダリングを済ませておくことができればリオーダリング時間の長さは問題になりにくい．しかし実世界のグラフは時々刻々と構造を

変化させる [55] ため，処理時にその都度リオーダリングすることが望ましい．また，本研究ではグラフクエリ処理のメモリアクセス効率の改善を目指している．クエリ処理ではクエリと無関係な頂点やエッジを無視することでグラフの規模を仮想的に縮小し，処理を効率化することができる．縮小されたグラフはクエリが投入されてから生成されるため，その場でリオーダリングする必要がある．従って，メモリアクセスの効率化で得られる高速化の効果をリオーダリングに要する時間が上回りやすく，全体の処理時間を短縮できない．

そこで本章では，グラフ処理実行時のリオーダリングによる処理時間の短縮を可能にするため，新しいアルゴリズムである *Rabbit Order* を提案する．Rabbit Order は高い局所性と短いリオーダリング時間を 2 つのアプローチによって両立する．1 つ目のアプローチはグラフの階層的なコミュニティ構造を捉えることで局所性を向上させる**階層コミュニティオーダリング**である．実世界のグラフはコミュニティ構造，即ち密に接続された頂点の集合を含むことが知られている [73]．密な接続によりコミュニティ内の頂点はグラフ処理中相互に頻繁なアクセスを行う．そのため，同一コミュニティ内の頂点に連続した ID 番号を与え，メモリ上で近傍に配置することで局所性が向上する．Rabbit Order はさらにコミュニティが入れ子状の階層構造をしばしば有する [19] ことに着目し，コミュニティ内の頂点とその内部コミュニティの頂点を再帰的に近傍に配置する．このような配置はコミュニティの階層と CPU が持つキャッシュの階層 (L1, L2 など) を自然に対応付けることでより効果的なキャッシュの活用を可能にする．2 つ目のアプローチは並列処理によるグラフの縮約とコミュニティ検出によってリオーダリング時間を短縮する**並列逐次集約**である．このアプローチは密に接続された 2 つの頂点を 1 つの頂点に集約する操作を繰り返すことで階層的コミュニティを検出する．さらにその操作を軽量なアトミック命令を使った並行性制御によって並列に実行することで高いスケーラビリティを実現する．

提案手法の特長は次の 3 つにまとめられる：

1. **高い局所性：** Rabbit Order は既存手法と比較して同等以上にキャッシュミスを削減する効果的な頂点順序を生成する．

2. **高速なリオーダリング：** Rabbit Order はリオーダリングを短時間で完了する．そのためリオーダリングを含めた全体処理時間の観点でグラフ処理を高速化することができる．
3. **幅広い有効性：** 実世界のグラフ 10 種類とグラフ処理アルゴリズム 6 種類を用いた実験において，Rabbit Order は様々な特徴を持つグラフとグラフ処理に対して効果を示した．

評価実験の結果，全体処理時間の観点で提案手法は平均 2.2 倍グラフ処理を高速化することが確認された．また同時に全体処理時間について既存手法は高速化効果をほぼ示さないことも判明した．

本章の構成は次の通りである．2.2 節で局所性向上に関する既存研究を紹介し，2.3 節で本章が前提とする知識の説明と問題定義を行う．2.4 節において提案手法である Rabbit Order の詳細を説明し，2.5 節で評価を行う．最後に 2.6 節で結論を述べる．

2.2 関連研究

グラフ処理の局所性向上に関する研究は 2 つに大別できる．1 つ目はリオーダリングの研究 [2, 8, 17, 22, 32, 36, 52, 56, 57, 66, 78, 79, 92] である．リオーダリング手法はグラフ処理の前処理としてグラフデータを最適化することで局所性を向上させる．2 つ目はグラフ処理フレームワークの研究である [35, 71, 95, 96]．グラフ処理フレームワークは局所性の高いデータ構造とデータへのアクセスを提供する API から成る．リオーダリングは前処理であるため，グラフ処理フレームワークよりも柔軟に適用できることが利点である．フレームワークを使用するためにはその API を用いてプログラムを実装する必要があるのに対し，リオーダリングは既存のプログラムも高速化することができる．また，フレームワークを用いて実装されたプログラムをリオーダリングによって一層高速化することもできる．例えば Zhang らは局所性の高い頂点順序によって彼らのグラフ処理フレームワーク Cagra の効率が高まることを確認している [96]．このようにリオーダリングは幅広く効果的なアプローチである．

リオーダリングの研究における Rabbit Order の位置付けを詳しく説明するため，以降ではリオーダリング手法を 4 種類に分類し代表的な手法を紹介する．

2.2.1　分割に基づく手法

グラフを密に接続された頂点のグループへ分割し，グループ内の頂点を連続に配置するというアイデアは過去にも Nested Dissection [37, 57], COM [79], Layered Label Propagation (LLP) [8], SlashBurn [66] などで提案されている．Rabbit Order はコミュニティ検出によるグラフ分割を利用するためこの分類に含まれる．これらの手法はグラフの分割にそれぞれ異なる手法を採用しているため，結果として得られる頂点順序およびリオーダリングに要する時間が異なる．既存手法と比較すると，Rabbit Order の特長は頂点 1 つのコミュニティに辿り着くまでのコミュニティ階層を捉えた階層的リオーダリングによる高い局所性，およびそれを逐次集約に基づくコミュニティ検出で実施することによる高速なリオーダリングにある．

2.2.2　探索に基づく手法

グラフ分割に基づく手法が分割と頂点順序の生成という 2 ステップに分かれるのに対し，一定のルールに従ってグラフを探索 (traverse) する訪問順に直接頂点を並べ替える手法がある．代表的なものとして Cuthill-McKee [22] および Reverse Cuthill-McKee (RCM) [36] が広く使用されている．RCM には NUMA 環境向けの亜種 [32] や並列化手法 [52] も提案されている．これらの手法は高速である一方，複雑なグラフ構造を捉えることができないため，局所性の向上効果は Rabbit Order や LLP よりも小さい．

2.2.3　ソートに基づく手法

頂点のソートによって頂点順序を生成する手法がしばしば用いられる．Shingle ordering [17] はソート時のキーとして MinHash 法 [11] に基づく隣接頂点集合のハッシュ値を用いる．これにより隣接頂点集合に重複が多い頂点同士が近傍

に配置されやすくなる．また，高次数頂点はグラフ処理において頻繁に参照されるため，頂点次数に基づくソートは一定の局所性向上効果を持つ [3, 66, 92]．Frequency Based Clustering (FBC) [96] は高次数頂点のみを次数ソートによって集約し，その他の頂点は頂点順序を変更しない．これにより，元々の頂点順序が持つ局所性を保ちつつ高次数頂点の局所性を向上させる．本章では Shingle ordering と次数ソートについて評価を行っているが，図 2.7 に示されているようにこれらの局所性向上効果は僅かである．また，本章の実験条件では局所性の低いランダムな頂点順序を入力としているため，FBC の局所性向上効果は次数ソート以下になると考えられる．

2.2.4 最適化に基づく手法

局所性の向上を最適化問題へ還元し，直接それを解くことで頂点順序を生成する研究が行われている．最新の手法は Wei らによって提案された Gorder [92] である．Wei らは 2 頂点が共有する隣接頂点の数に着目し，頂点順序の局所性の高さを示すスコア関数を定義した．Gorder はスコアが高くなるような頂点順序を巡回セールスマン問題 (TSP) の近似解法によって発見する．このような頂点順序は高い局所性を示す一方で，生成のために長い時間を要する．例えば Rabbit Order は twitter グラフ (表 2.1) を約 25 秒でリオーダリングするのに対し，Gorder は 5,000 秒以上を要する [92]．文献 [2, 78] の手法は隣接行列の行または列の間に存在する局所性をハイパーエッジとして表現し，ハイパーグラフの分割や TSP を通じて頂点順序を生成する．しかしハイパーグラフに基づくこれらの手法は，n を頂点数，m をエッジ数とするとき，少なくとも $O(\frac{m^2}{n})$ の時間計算量を示す [13]．そのため大規模なグラフへの適用が困難である．このように最適化に基づく手法はグラフ処理実行時のリオーダリングに適さないのに対し，Rabbit Order はリオーダリング時間を含めてグラフ処理の高速化を達成している．

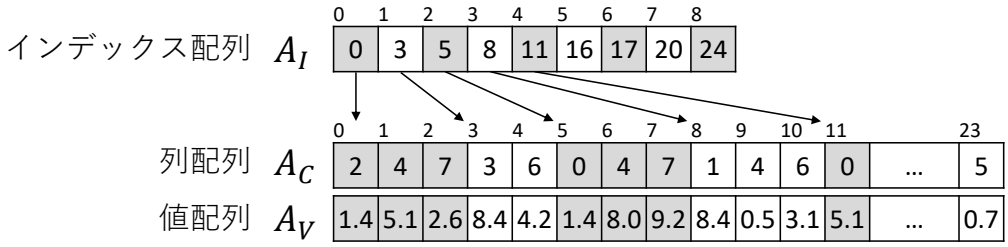


図 2.2: 図 2.1(c) の疎行列を表現する CSR.

アルゴリズム 1 疎行列ベクトル乗算 (SpMV)

入力: $n \times n$ CSR 行列 $A = (A_I, A_C, A_V)$，長さ n のベクトル $\boldsymbol{x}$.
出力: $\boldsymbol{y} = A\boldsymbol{x}$.

1: **for** $i = 0$ **to** $n - 1$ **do**
2: $\quad \boldsymbol{y}[i] \leftarrow 0$
3: $\quad$ **for** $k = A_I[i]$ **to** $A_I[i+1] - 1$ **do**
4: $\quad\quad \boldsymbol{y}[i] \leftarrow \boldsymbol{y}[i] + A_V[k]\,\boldsymbol{x}[A_C[k]]$

2.3　事前準備

本節ではまずグラフ処理の典型的なメモリアクセスパターンを説明し，次に本章で扱う問題を定義する．

2.3.1　グラフ処理のメモリアクセスパターン

グラフ処理アルゴリズムは頂点間のつながりを捉えるためのメモリアクセスを行うという点で共通するものの，具体的なアクセスパターンは様々である．特にグラフクエリ処理はアルゴリズム毎の差が大きい．例えば PageRank のように繰り返し処理を行うもの [43] や，BFS 状にエッジを辿るもの [7, 42] などがある．

これらはそれぞれ異なる振る舞いをするが，一方で PageRank や BFS を含む典型的なグラフ処理は**疎行列ベクトル乗算** (sparse matrix-vector multiplication; *SpMV*) によって表現可能であることが知られている [51]．そこで本論文では SpMV に基づいてアクセスパターンの議論を行う．SpMV とは密ベクトル $\boldsymbol{x}$ と

疎行列 A についてその積 $\boldsymbol{y} = A\boldsymbol{x}$ を求める計算である．グラフ処理において多くの場合 A は重み付き隣接行列である．

疎行列の表現形式としては *compressed sparse row* (*CSR*) が広く用いられている [2, 52, 78]．CSR は疎行列中の非ゼロ値とその位置を 3 つの配列，即ちインデックス配列 A_I，列配列 A_C，値配列 A_V によって表現する．$A_I[i]$ は行列の $i+1$ 行目に対応するデータの A_C と A_V における開始位置を示し，A_C と A_V の要素はそれぞれ列番号とその列に存在する値を表す．例として図 2.1(c) の行列に対応する CSR 表現を図 2.2 に示す．CSR を用いた SpMV はアルゴリズム 1 に示すように計算される．このアルゴリズムは $\boldsymbol{x}$ 以外の配列に対しては連続アクセスを行うが，$\boldsymbol{x}$ に対しては A_C を通じた間接参照により不規則なアクセスを行う (4 行目)．そのため SpMV は $\boldsymbol{x}$ に対するメモリアクセスの局所性が低い．

2.3.2 本章が扱う問題

本章ではグラフ処理に広く見られる SpMV 型のアクセスパターンを想定し，その局所性をリオーダリングによって向上させる．具体的には，$V = \{0, 1, \ldots, n-1\}$ を正の整数を ID とする n 個の頂点の集合，$E \subseteq V \times V$ をエッジの集合，$G = (V, E)$ をグラフ，A を G の隣接行列とするとき，アルゴリズム 1 で示される SpMV において発生するキャッシュミスを削減するような頂点 ID の置換 $\pi : V \to V$ を生成する．

さらに，本章ではグラフ処理の実行時にリオーダリングを行うことによる高速化を目指す．具体的には，グラフ処理アルゴリズムおよびグラフ G を与えられたとき，キャッシュミスの削減によりリオーダリングとグラフ処理を合わせた全体処理時間を短縮する．

2.4 提案手法

本節では提案手法である Rabbit Order を特徴づける 2 つのアプローチ，即ち階層コミュニティオーダリングと並列逐次集約について説明する．

2.4.1　階層コミュニティオーダリングによる局所性向上

実世界のグラフはしばしばコミュニティ構造を含む [73]．コミュニティとは密に接続された頂点の集合である．密に接続された頂点はグラフ処理において相互に頻繁なメモリアクセスを発生させる．そのため，コミュニティ内の頂点を連続したメモリ領域に配置することで高い空間的・時間的局所性を得られることが知られている [8, 79]．エッジは隣接行列において非ゼロ値として表現されることから，各コミュニティに含まれる頂点を連続して配置することによって隣接行列の対角上に密なブロックが形成される．密なブロックは狭いメモリ領域への頻繁なメモリアクセスを生じさせることから，高い局所性を示す．そのような頂点順序に図 2.1(a) のグラフを並べ替えたものが図 2.1(b) である．図 2.1(d) の灰色部分で示すようにコミュニティ 1, 2 のそれぞれに対応する密なブロックが隣接行列に生じている．コミュニティ 1 に対応するブロック (頂点 0–4) の処理時は頻繁に頂点 0–4 へのアクセスが発生するため局所性が高い．コミュニティ 2 についても同様である．このようにコミュニティ構造を利用することで局所性を向上させることができる．

以上のアイデアをさらに発展させ，本研究では実世界のグラフにみられる階層的なコミュニティ構造 [19] に着目する．階層構造において各コミュニティはより密に接続された内部コミュニティを含んでいる．例えば学生のソーシャルネットワークには，学校，学年，クラスのように段階的に密になる人間関係の階層が見られるはずである．

階層的コミュニティから局所性を抽出するため，階層コミュニティオーダリングを提案する．階層コミュニティオーダリングとは，いずれの階層においても各コミュニティ内の頂点が連続して配置されているような頂点順序である．このような順序は内部コミュニティの頂点についても再帰的に連続した配置を行うことによって得ることができる．階層コミュニティオーダリングでは隣接行列において密なブロックの内側に内部コミュニティと対応する一層密なブロックが形成されるため，高い局所性を得ることができる．さらに階層構造を捉えた頂点順序は

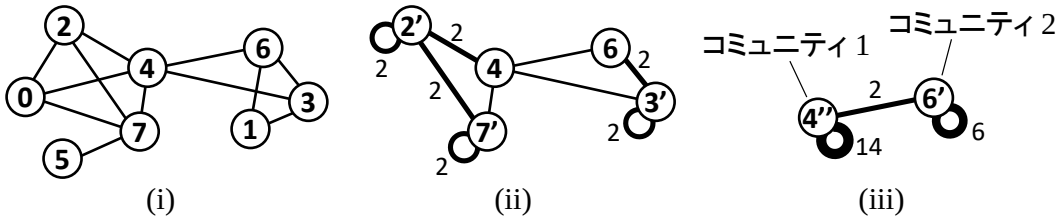
アルゴリズム 2 Rabbit Order の概要

入力: グラフ $G = (V, E)$
出力: 新しい頂点順序を表す置換 $\pi : V \to V$

```
1: dendrogram ← CommunityDetection()
2: return OrderingGeneration(dendrogram)
3: function CommunityDetection()
4:     V を頂点次数順にソート
5:     for each u ∈ V do
6:         v ← ΔQ(u, v) を最大化する u の隣接頂点
7:         if ΔQ(u, v) > 0 then
8:             u を v へ集約し，その操作を dendrogram に記録
9:     return dendrogram
10: function OrderingGeneration(dendrogram)
11:    new_id ← 0
12:    V を dendrogram における DFS 訪問順にソート
13:    for each v ∈ V do
14:        π[v] ← new_id; new_id ← new_id + 1
15:    return π
```

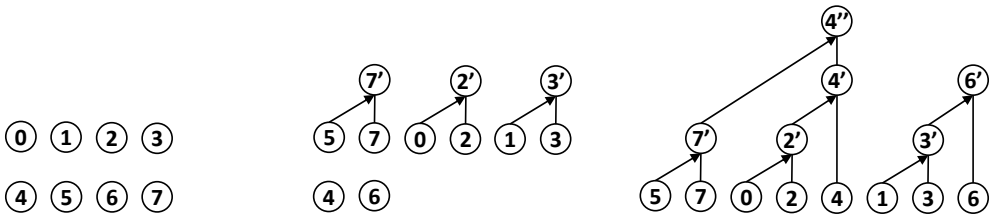
現代の CPU において一般的な階層的キャッシュの効果的な利用も期待できる．即ち，小さく密な内部コミュニティのデータを L1 キャッシュに，その外側のコミュニティのデータを L2, L3 キャッシュに，というような配置を自然に得ることができる．このように，階層コミュニティオーダリングは高い局所性によりグラフ処理の性能を向上させる．

アルゴリズム 2 に Rabbit Order の概要を示す．階層コミュニティオーダリングはコミュニティ検出とそれに基づく頂点順序の生成という 2 つのステップから成る．まず CommunityDetection 関数によりコミュニティの階層を抽出するとともに，それを表現したデンドログラム (樹形図) を得る (1 行目)．次に OrderingGeneration 関数によりデンドログラムから新しい頂点順序を得る (2 行目)．Rabbit Order はどちらのステップにおいても並列処理を行い，短時



(i)　(ii)　(iii)

(a) 逐次集約の進行



(i)　(ii)　(iii)

(b) デンドログラムの構築

図 2.3: コミュニティ検出およびそれと同時に行われるデンドログラム構築の例.

間でリオーダリングを完了する．なお入力された G が対称な有向グラフでない場合，アルゴリズムの開始前に各エッジについて反対方向のエッジを追加することで対称な有向グラフに変換する．これはエッジの向きを無視することでコミュニティ検出の品質が向上するためである [73]．次節以降ではコミュニティ検出と頂点順序生成の詳細について説明する．

2.4.2　高速なコミュニティ検出

本節では逐次処理によるコミュニティ検出アルゴリズムを説明したのち，その並列版，即ち並列逐次集約について述べる．

逐次処理アルゴリズム

階層コミュニティオーダリングに基づくリオーダリングを短時間で行うためには，効率的に階層的コミュニティ構造を検出する必要がある．そこで Rabbit Order は *modularity* [73] に基づく**逐次集約法** [85] と呼ばれるテクニックを採用

している．Modularity はコミュニティ検出の品質を表す指標として最も有名なものの一つである．逐次集約法は頂点のペアが同じコミュニティに分類されるかどうかを modularity の上昇量に基づき判定する．同じコミュニティに分類される場合，2 つの頂点を直ちに 1 つの頂点へと集約する．このように集約を繰り返すことでグラフの規模が縮小されていくため，小さい計算コストでコミュニティを検出することができる．Rabbit Order は集約の過程をコミュニティの階層構造の検出とみなし，その構造をデンドログラムとして記録する．

逐次集約の具体的な手続きをアルゴリズム 2 の CommunityDetection 関数に示す．処理は 3 ステップから成る：(i) 集約元頂点の選択 (4, 5 行目)，(ii) 集約先頂点の決定 (6 行目)，(iii) modularity が上昇するならば集約 (7, 8 行目)．ステップ (i) では頂点次数が小さい順に頂点 u を選択する．次数順の選択は逐次集約の計算コストを削減するためのヒューリスティクスである [85]．ステップ (ii) では頂点を集約した際に得られる modularity 上昇量 ΔQ (式 2.1) を最大化する u の隣接頂点 v を集約先頂点として選択する．

$$\Delta Q(u,v) = 2\left\{\frac{w_{uv}}{2m} - \frac{d(u)d(v)}{(2m)^2}\right\}. \tag{2.1}$$

ただし m は初期状態のグラフにおけるエッジの数，w_{uv} は頂点 u と v の間のエッジの重み，$d(\cdot)$ は頂点の重み付き次数 (接続されたエッジの重みの和) である．初期状態においてエッジの重みは全て 1 であるものとする．もし $\Delta Q(u,v) > 0$ ならば，ステップ (iii) において u と v を集約する．集約後の頂点 v' は u と v 両方の重みを足したエッジを持つ．なおエッジ (u,v) と (v,u) は区別されるため，v' には重み $w_{uv} + w_{vu}$ のループエッジが追加される．一方 modularity が上昇しない場合，u をそのままグラフに残す．このようにグラフに残った頂点を本章では**トップレベル頂点**と呼ぶ．トップレベル頂点はデンドログラムにおいて根となる．図 2.3 に逐次集約による階層的コミュニティ検出の例を示す．低次数頂点から頂点の集約を繰り返すことによりグラフを縮約すると同時にデンドログラムを構成する．最終的にトップレベル頂点 4 と 6 をそれぞれの代表点とするコミュニティ 1 と 2 を得る．Rabbit Order はこのように逐次集約によって頂点数を減少させることで高速にコミュニティを検出する．

並列逐次集約

CommunityDetection 関数は一見すると 5 行目の for 文において各頂点の処理を複数のスレッドで行うことにより容易に並列化可能であるように見える．しかしながら，このナイーブな並列化は頂点の集約 (8 行目) における集約先頂点のエッジの編集および集約元頂点の削除で競合状態を生じる．同様に競合状態はデンドログラムの更新においても発生する．小さいオーバーヘッドでこのような問題を解決するための手段として多くの CPU は compare-and-swap (CAS) などのアトミック命令を提供しているが，CAS がアトミックに変更可能なデータサイズは限られる．

そこで軽量なアトミック命令を活用するため，Rabbit Order では**遅延集約**によって操作するデータサイズを削減する．u を v へ集約することが決定されたとき，遅延集約では v を代表点とするコミュニティに u を登録する操作のみ行う．u と v の実際の集約は v が集約元頂点として別の頂点へ集約されるときまで遅延される．コミュニティへの登録に必要なデータの変更を最小限にすることでアトミック命令による実装が可能となる．また，デンドログラムはこの登録操作の結果構成されるデータ構造から復元する．

並列コミュニティ検出アルゴリズムをアルゴリズム 3 に示す．最初に変数を初期化する (1–6 行目)．配列 *atom* と *sibling* はアトミック命令を利用しつつデンドログラムを表現するために用いる．まず *atom* は遅延集約のためアトミックに操作される構造体の配列である．任意の頂点 u について $atom[u]$ は次の 2 つの変数から成る．(i) *degree*: u を代表点とするコミュニティに含まれる頂点の次数の和．ΔQ の計算において $d(u)$ の値として用いる．(ii) *child*: u へ最後に集約された頂点の ID[*1]．頂点数が $2^{32}-1$，エッジ数が $2^{64}-1$ に達する大規模グラフを想定したとしても，*atom* の各要素のサイズは 64 ビット整数型の *degree* と 32 ビット整数型の *child* を合わせて 12 バイトに留まる．従って x86-64 プロセッサなどにおいて CAS 命令によって操作可能である．配列 *sibling* は同じコ

[*1] それぞれ無効な頂点 ID と次数を表す定数 `INVALID_ID` および `INVALID_DEGREE` としては有効には存在し得ない値を実装都合で定める．

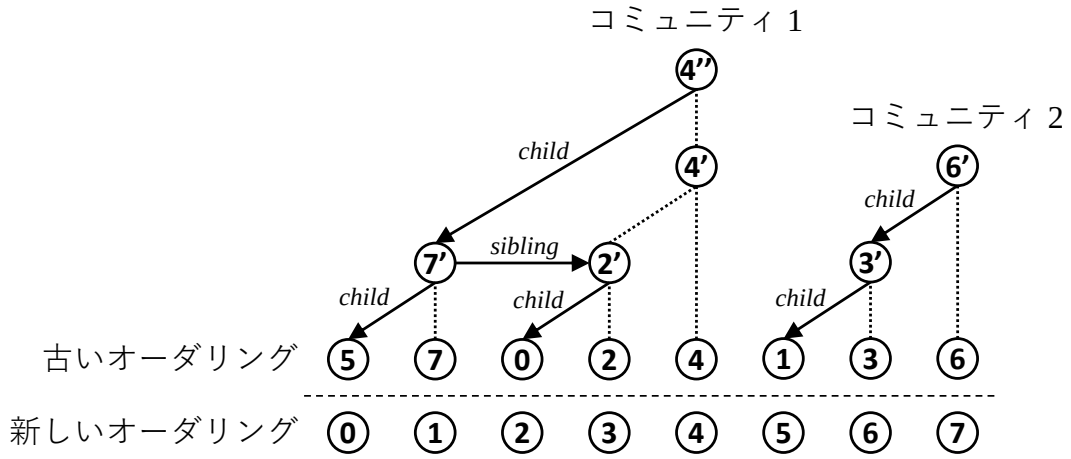


図 2.4: 図 2.3(b) のデンドログラム (iii) を表現するデータ構造，およびそこから導出される新しい頂点順序．

ミュニティに属する頂点の ID を記録し，単方向リスト構造を成す．これら 2 つの変数を用いると，$atom[u].child$ に記録された頂点から始まる *sibling* のリストとリストに含まれる頂点の *child* を再帰的に辿ることで頂点 u のコミュニティのメンバを収集できる．例として図 2.3(b) のデンドログラム (iii) の構造を図 2.4 に示す．頂点 4 のコミュニティのメンバは，$atom[4].child$ $(= 7)$, $atom[7].child$ $(= 5)$, $sibling[7]$ $(= 2)$, $atom[2].child$ $(= 0)$ の 5 頂点である．逆に各頂点からその頂点が所属するコミュニティを取得するためには変数 *dest* を用いる．*dest* は Union-find 木構造によって各頂点が所属するコミュニティを表現する．初期状態において各頂点は自身のみが所属するコミュニティに属する．

初期化の後，アルゴリズムは逐次集約を開始する．まず各スレッドは次数の小さいものから順に集約元頂点 u を選択する (8 行目)[*2]．次に他のスレッドが u を集約先頂点として選択することがないよう，*degree* に無効値を代入する (9, 10 行

[*2] 逐次集約法において次数昇順での選択はあくまでヒューリスティクスであるため，ロックなどを用いて厳密に次数順に選択することよりも処理効率を優先してよい．そこで実装上は頂点を次数昇順にソートした配列を作り，1 番目の頂点をスレッド 1 へ，2 番目の頂点をスレッド 2 へ，3 番目の頂点をスレッド 3 へ，... というように次数順にラウンドロビンで割り当てることによって低次数頂点から優先的に処理している．配列に対する各スレッドのメモリアクセスは局所性が低いが，読み込みのみであるため影響は小さい．

目)．なお AtomicSwap(x, y) はアトミックに変数 x と y の値を入れ替える操作を表す．集約先頂点 v は FindBestDestination 関数によって発見する (11 行目)．得られた v に対して集約による modularity の上昇がない場合，u はトップレベル頂点である．集約先頂点として選択され得るよう *degree* の値を復元するとともに，トップレベル頂点の集合 *Toplevel* へ u を追加する (12–15 行目)．一方 modularity が上昇し，かつ v が集約可能な状態である場合，v のコミュニティのメンバとして u を追加する (16–24 行目)．具体的には，$atom[v].child$ から始まる *sibling* 上のリスト構造の先頭に u を挿入し，v の頂点次数に u の次数を追加するという処理を CAS により行う．CAS が成功した場合，u と v のコミュニティを合併する．コミュニティは *dest* を用いた Union-find 木で保持しているため，$dest[u]$ に v を設定することで合併を表現する．並列処理の影響で集約に失敗した場合，変更された変数の値を復元し，u をリトライ待ちとして保存する (25–27 行目)．集約のリトライは他の頂点の集約が一通り完了した後に行う (28 行目)．

FindBestDestination 関数は引数 u のコミュニティおよび隣接コミュニティの間のエッジの重みを計算し，その値に基づき ΔQ を求める．詳細をアルゴリズム 4 に示す．関数は 4 つのステップから成る．ステップ 1 (2 行目) では u のコミュニティに属する頂点を，図 2.4 に示すように *child* と *sibling* を辿ることで収集する．ステップ 2 (3–5 行目) ではコミュニティに隣接する頂点 v について $dest[v]$ が v の属するコミュニティの ID となるよう更新する．ただし $N(v)$ を頂点 v の隣接頂点集合とするとき $N(C) = \bigcup_{v \in C} N(v)$ である．ステップ 3 (6–9 行目) では遅延集約における実際の集約処理を行う．つまり C に含まれる頂点と，隣接するコミュニティに属する頂点の間のエッジの重みを算出し，頂点 u へ付け替える．ステップ 4 (10 行目) では最も大きな modularity 上昇量をもたらす隣接頂点を探し出し返却する．

2.4.3 頂点順序生成

階層的コミュニティが検出された後，Rabbit Order はその結果を用いて頂点順序を生成する．まず逐次処理版のアルゴリズムを OrderingGeneration 関数 (アルゴリズム 2) として示す．各階層のコミュニティに含まれる頂点を再帰的に近傍に配置するため，アルゴリズムはトップレベル頂点からの深さ優先探索 (DFS) を行い，その訪問順を置換 π として出力する．例えば図 2.4 におけるトップレベル頂点 4 からの DFS は 5, 7, 0, 2, 4 という訪問順になる．このとき生成される π は $\pi[5] = 0, \pi[7] = 1, \pi[0] = 2, \pi[2] = 3$, $\pi[4] = 4$ である．トップレベル頂点 6 からの置換の生成も同様である．なお接続が疎であるためコミュニティ間の順序は定義しない．トップレベル頂点が複数ある場合は任意の順序で選択しそれぞれの木において DFS を行う．このようにして得られる置換が Rabbit Order の最終的な出力となる．

上記の頂点順序生成アルゴリズムは 2 つのステップで容易に並列化することができる．まず各トップレベル頂点から並列に DFS を開始し，そのコミュニティ内部でのローカルな頂点順序を決定する．次に各トップレベル頂点から得られたローカルな頂点順序を逐次処理にて結合しグラフ全体の頂点順序を得る．このように Rabbit Order はコミュニティ検出と頂点順序生成の両方を並列化している．

2.4.4 定性的議論

Rabbit Order の性質を理解するため，以下で定性的な議論を行う．

時間計算量

アルゴリズム 2 に表現されているように，Rabbit Order はコミュニティ検出 (1 行目) と頂点順序生成 (2 行目) の 2 ステップに分けられる．従って，Rabbit Order はこれらのステップの計算量を合計した計算量を持つ．ステップ 1 では逐次集約によるコミュニティ検出を行う．その計算量は $O(m + \frac{\alpha(1-c)}{\alpha-1}n)$ であるこ

とが文献 [84] において示されている．ここで m はグラフのエッジ数，c はクラスタ係数，α は冪乗則に基づく次数分布の偏り (skewness)，n は頂点数である．ステップ 2 の頂点順序生成では，頂点をデンドログラム上の DFS 訪問順にソートし (12 行目)，その順番に ID が振られるように置換を生成する (13, 14 行目)．デンドログラムでは n 個全ての頂点がツリー状に結合しているため，エッジの本数は $O(n)$ である．従って DFS の計算量は $O(n)$ である．さらに置換の生成では n 個それぞれの頂点に連続した ID を生成するため，計算量は $O(n)$ である．以上より頂点順序生成の計算量は $O(n+n) = O(n)$ となる．Rabbit Order の計算量はコミュニティ検出と頂点順序生成の計算量の和であるから，

$$O\left(\left(m + \frac{\alpha(1-c)}{\alpha-1}n\right) + n\right) = O\left(m + \left(1 + \frac{1-c}{1-1/\alpha}\right)n\right) \tag{2.2}$$

である．

式 2.2 に示されているようにクラスタ係数 c と次数分布の偏り α が大きいほど計算量は小さくなる．クラスタ係数が大きいということは多くの頂点が互いに密に接続されているということである．また次数分布の偏りが大きいということは，多くの頂点が小さい次数を持ちごく一部の頂点が極端に大きい次数を持つということである．これらの特徴は実世界のグラフにおいて一般的であることが知られている [30,91]．従って，Rabbit Order は実世界のグラフに対して効率的に動作することが分かる．なお c と α に対する計算量の依存は逐次集約法に由来しており，これらのパラメータに対する逐次集約法の性能変化は文献 [84] において実験的にも確認されている．

並列処理による非決定性

並列逐次集約は並行性制御のコストを抑えることで効率的にコミュニティ検出を行うことができるように設計されている．しかし一方で並列動作時の処理結果は非決定的である．具体的には，アルゴリズム 3 の while ループ (8–27 行目) は各スレッドが独立に処理するため，頂点が集約される順序は各スレッドの処理の進行状況に依存する．集約の順序が入れ替わると最適な集約先頂点の選択 (11 行目) を行う際にスレッドから見えるグラフ構造に差異が生じ，逐次実行時とは異

表 2.1: 実験で使用するグラフ

名前	頂点数	エッジ数	出典
berkstan	0.7M	7.6M	SNAP[*3]の WEB-BERKSTAN
enwiki	4.2M	101.4M	LAW[*4]の ENWIKI-2013
ljournal	4.8M	69.0M	SNAP の SOC-LIVEJOURNAL1
uk-2002	18.5M	298.1M	LAW の UK-2002
road-usa	23.9M	57.7M	DIMACS [5] の ROAD_USA
uk-2005	39.5M	936.4M	LAW の UK-2005
it-2004	41.3M	1150.7M	LAW の IT-2004
twitter	41.7M	1468.4M	LAW の TWITTER-2010
sk-2005	50.6M	1949.4M	LAW の SK-2005
webbase	118.1M	1019.9M	LAW の WEBBASE-2001

なる頂点へ集約される場合がある．従って Rabbit Order を複数スレッドで実行した場合，同一のグラフを与えても実行毎に異なるデンドログラムが生成される可能性がある．ただし逐次集約法は元々 modularity を最大化する厳密解法ではないため，必ずしも逐次実行時の結果が最善とは限らない．またコミュニティ抽出結果の差異がリオーダリング結果の局所性に及ぼす影響は自明でない．そのため並列処理時の非決定性が局所性に影響するかどうかは 2.5.3 節において実験的に確認する．

2.5 評価

本節では全体処理時間に関する Rabbit Order の性能を評価する[*5]．全ての実験は Intel Xeon E5-2697v2 を 2 ソケット搭載し，256GB のメモリを搭載する計算機で行った．Hyper-Threading を有効にすることで計算機としては最大 48 スレッドを実行可能である．特に言及がない限り実験は 48 スレッド実行で行う．

[*3] http://snap.stanford.edu/data/index.html
[*4] http://law.di.unimi.it/
[*5] 実験に用いた Rabbit Order の実装は https://git.io/rabbit にて公開されている．

実験に用いたグラフを表 2.1 に示す．また性能比較は次の 7 つのリオーダリング手法と行う (括弧内は実験結果での略記)：SlashBurn (Slash) [66], Unordered parallel BFS (BFS) [52], Unordered parallel RCM (RCM) [52], Multithreaded Nested Dissection (ND) [57], Layered Label Propagation (LLP) [8], Shingle ordering (Shingle) [17], 頂点次数の昇順 (Degree)．SlashBurn については論文中で示されている最善のパラメータを使用する (S-κH と $k = 0.02n$ の組み合わせ) [66]．また SlashBurn 以外は全て並列アルゴリズムである．Degree と Shingle は本質的には単純なソートであるため，GCC 4.9.2 が提供する `__gnu_parallel::sort` 関数によって並列化した．実験ではさらにベースラインとしてランダムな頂点順序 (Random) を用いる．これはデータセットの配布者がデータ取得元から得られる自然な頂点順序 (Web クローラの訪問順，SNS の会員 ID 順など) に対して変更を加えている場合があるためである [8, 9]．オーダリングをランダムとすることでデータ配布者が与えた影響を排除できる．

実験では主に PageRank を使用する．PageRank は SpMV に基づく最も有名なアルゴリズムの一つである．具体的には $k = 0, 1, 2, \ldots$ について $\boldsymbol{s}$ が収束するまで次の計算を繰り返す：

$$\boldsymbol{s}_{k+1} = (1 - c)W\boldsymbol{s}_k + c\boldsymbol{e}. \tag{2.3}$$

ここで $\boldsymbol{s}_k$ と $\boldsymbol{s}_{k+1}$ は各頂点の PageRank スコアを表すベクトル，c は $0 \leq k \leq 1$ のパラメータ，$\boldsymbol{e}$ はいずれの要素の値も $\frac{1}{n}$ の長さ n のベクトル，$W = (w_{uv})$ はエッジ (u, v) が存在するとき $w_{uv} = \frac{1}{d(v)}$，そうでなければ $w_{uv} = 0$ の行列である．既存研究 [56] に従い，$c = 0.15$ とし，収束の条件は $|\boldsymbol{s}_{k+1} - \boldsymbol{s}_k| < 10^{-10}$ を使用する．標準的な方法として PageRank の並列化はアルゴリズム 1 の最も外側の for 文に対して行った．

2.5.1　全体処理時間と内訳

初めにリオーダリングと PageRank を合わせた全体処理時間の改善について図 2.5 に示す．高速化率はランダム順序で得られた PageRank 処理時間を各リオーダリング手法で得られた全体処理時間で割ることによって求めた．Rabbit

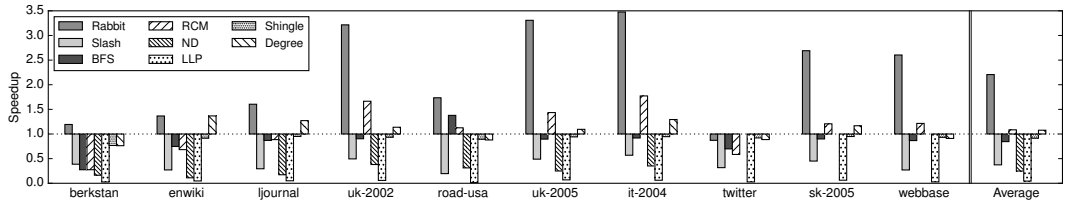


図 2.5: 全体処理時間に関する性能向上率.

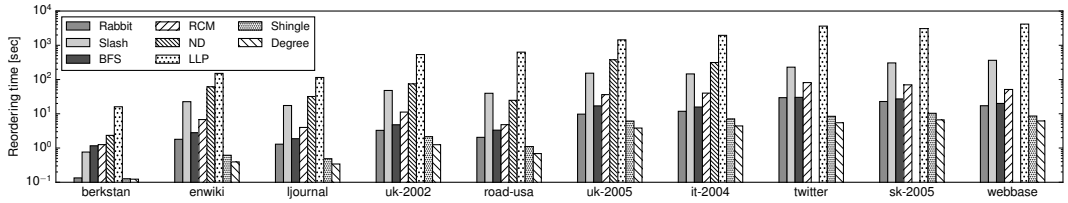


図 2.6: リオーダリングに要した処理時間.

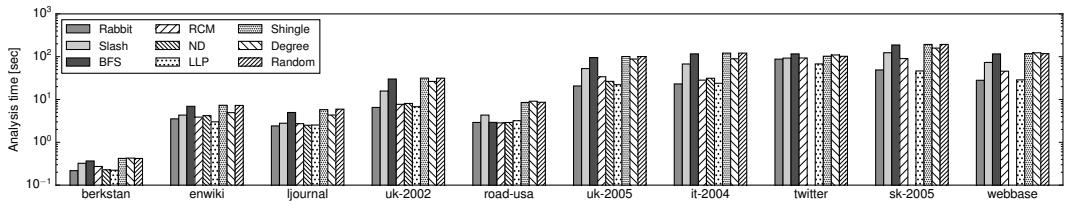


図 2.7: PageRank に要した処理時間.

Order は平均で 2.21 倍，最大では it-2004 において 3.48 倍の高速化率を示し，他手法を大きく上回っている．なお ND は twitter, sk-2005, webbase においてメモリ不足のため動作しなかった．

性能向上の要因を分析するため，全体処理時間をリオーダリング処理時間と PageRank による分析時間に分解する．図 2.6 はランダム順序のグラフにおける各手法のリオーダリング時間を示したものである．ソートに基づく手法 (Degree, Shingle) を除くと Rabbit Order の処理時間は極めて短いことが分かる．例えば 10 億以上のエッジを含む webbase グラフを 13.8 秒でリオーダリングすることができている．さらにそれぞれの頂点順序における PageRank の分析時間を図 2.7 に示す．Rabbit Order はいずれの既存手法と比較しても同等またはそれ以上に分析時間を短縮している．具体的にはランダム順序と比較し平均で 3.37 倍，最

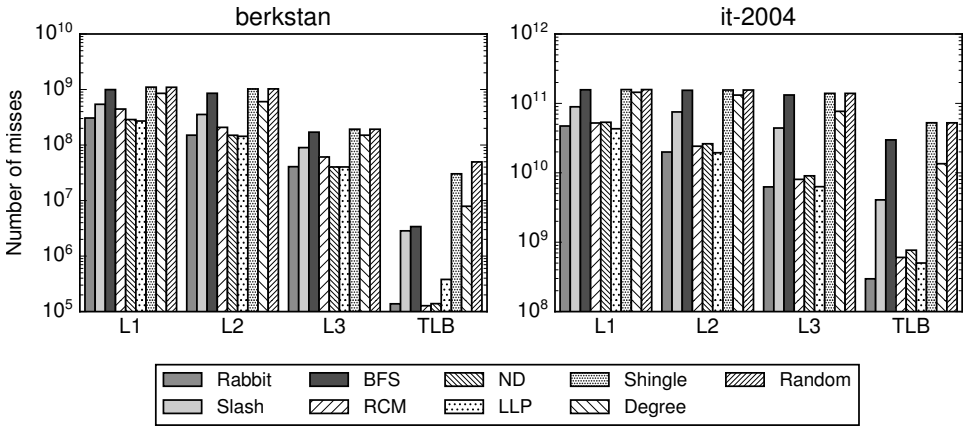


図 2.8: キャッシュミスと TLB ミスの数

大で it-2004 において 5.25 倍分析処理を高速化している．この結果は既存手法の中で最も効果的であった LLP (平均 3.34 倍，最大 5.08 倍) よりも僅かに良い．

2.5.2　キャッシュミス回数

PageRank が高速化した理由を確認するため，各キャッシュレベルにおけるキャッシュミスの回数と translation look-aside buffer (TLB) ミスの回数を図 2.8 に示す．実験では berkstan と it-2004 を使用した．これらはそれぞれ 2.5.1 節の実験で全ての既存手法がリオーダリングできた最小および最大規模のグラフである．実験結果からは図 2.7 に示された PageRank 処理時間とキャッシュミス回数が同じ傾向を示していることを確認できる．具体的には Rabbit Order と LLP はキャッシュミス回数が最も小さく，高速化率の低い BFS, Shingle, および Degree はミス回数の削減率が最も低い．また，berkstan よりも it-2004 においてキャッシュミス削減率が大きいことを読み取れる．これは berkstan が小規模なグラフでありランダム順序でもキャッシュに乗りやすいためである．同様に比較的小規模な enwiki と ljournal においてもリオーダリングによる高速化率は全体的に低い (図 2.7) が，it-2004 のような大規模なグラフではリオーダリングの効果が顕著となる傾向にある．

以上の結果より，Degree, Shingle など短時間でリオーダリング可能な手法は

キャッシュミスの削減率が低く，一方で LLP のようにキャッシュミス削減率が高い手法は長いリオーダリング時間を要することを読み取れる．Rabbit Order はこれら既存手法と異なり，短時間でのリオーダリングに加え既存手法と同等以上のキャッシュミス削減率を両立している．

2.5.3 並列処理の影響

2.4.4 節で述べたように，Rabbit Order の並列逐次集約は非決定的に動作する．そのことがコミュニティ検出の品質とリオーダリング結果に及ぼす影響を確認するため，逐次実行と 48 スレッド実行での Rabbit Order の出力を比較する．表 2.2 にコミュニティ検出の結果得られた modularity と，リオーダリング結果のグラフ上における PageRank の処理時間を示す．表中の「逐次」と「48 スレッド」は Rabbit Order の実行に使用したスレッド数を意味する．他の実験と同様，リオーダリング後の PageRank は常に 48 スレッドで実行している点に注意されたい．実験結果から分かるように，並列処理が示す差異は僅かである．48 スレッド実行時に得られる modularity は逐次実行時と同等かそれ以上の値を示している．同様に，48 スレッド実行で得た頂点順序は逐次実行時と同等の PageRank 処理効率を保っている．以上の結果より，並列逐次集約は高速なリオーダリングと高い局所性を実現しており，非決定的であるものの実用上問題ないことが分かる．

2.5.4 スケーラビリティ

並列処理の効率を評価するため，それぞれのリオーダリング手法を並列実行することで得られる性能向上率を図 2.9 に示す．なお高速化率は表 2.1 の全グラフに対して適用した結果の平均である．SlashBurn は逐次処理手法であるため除外している．12, 24, 48 スレッド実行はそれぞれ，1 ソケットの 12 物理コア，2 ソケットに跨る 24 物理コア，および Hyper-Threading にて行っている．図に示されているように，Rabbit Order はリオーダリング手法の中で最も高い高速化率を達成している．具体的には 48 スレッド実行時に 17.4 倍高速化している．

表 2.2: Rabbit Order によって得られる modularity と PageRank 処理時間．括弧内は逐次実行に対する差異の割合．

グラフ	Modularity		PageRank 処理時間 [秒]		
	逐次	48 スレッド	逐次	48 スレッド	
berkstan	0.930	0.934	0.214	0.218	(+2.0%)
enwiki	0.597	0.602	3.454	3.529	(+2.2%)
ljournal	0.708	0.701	2.421	2.416	(-0.2%)
uk-2002	0.985	0.982	6.415	6.520	(+1.6%)
road-usa	0.997	0.998	2.901	2.914	(+0.5%)
uk-2005	0.979	0.981	21.833	20.719	(-5.1%)
it-2004	0.972	0.975	23.179	23.100	(-0.3%)
twitter	0.360	0.391	90.235	87.709	(-2.8%)
sk-2005	0.971	0.974	47.541	49.067	(+3.2%)
webbase	0.978	0.979	27.474	28.128	(+2.4%)

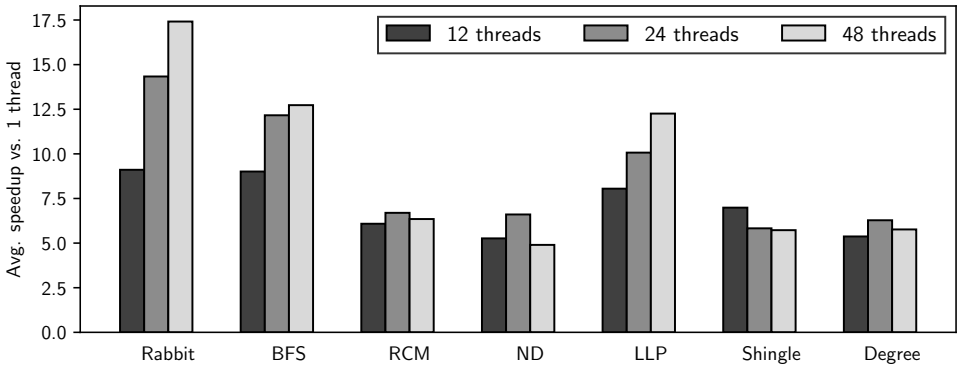


図 2.9: 並列実行によって得られる性能向上率．

BFS と LLP も約 12 倍高速化しているが，他の手法はスレッド数によりほぼ変化していない．Rabbit Order は逐次集約法が内包する高い並列性を軽量なアトミック命令によって引き出しているため，スレッド数に対し高いスケーラビリティを示すことができている．

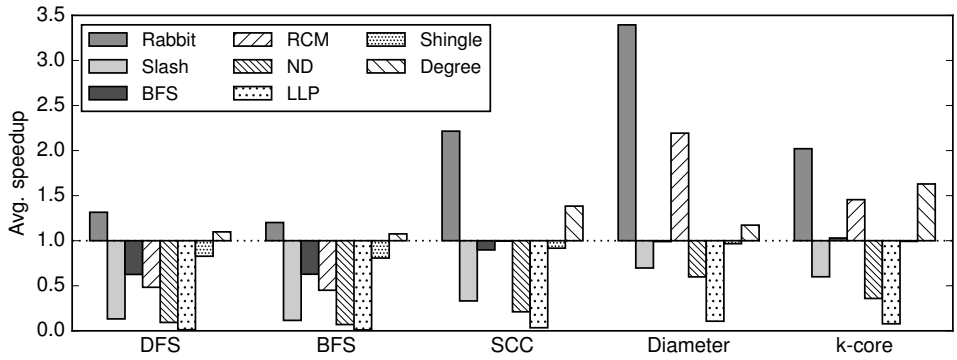


図 2.10: 全体処理時間に関する性能向上率.

2.5.5 様々な処理アルゴリズムに対する効果

ここまで SpMV に基づくグラフ分析アルゴリズムである PageRank を用いてきたが，BFS のように一般的には SpMV を用いず実装されるアルゴリズムに対してもリオーダリングは効果を示すことが知られている [32, 79]．そこで BFS，DFS，強連結成分分解 (SCC)，グラフ直径 (Diameter)，そして k-core 分解の 5 つのアルゴリズムについても全体処理時間の計測を行った．結果を図 2.10 に示す．高速化率はランダム順序の場合を 1 とした相対的な処理時間を全グラフについて平均した値を示している．いずれの処理アルゴリズムに対しても Rabbit Order は最も高い高速化率を示している．全体として SCC, Diameter, k-core の高速化率に比べ DFS と BFS の高速化率が低いのは，それらが軽量な処理であることに起因する．後続のグラフ処理に要する時間が短いほどリオーダリングに要した時間の償却が難しいため，高速化率が小さくなる傾向になる．それにも関わらず Rabbit Order が全体処理時間を短縮できているのは高速なリオーダリングと高い局所性を両立しているためである．

2.6 結論

本章ではグラフ処理実行時のリオーダリングに使用するためのアルゴリズムである Rabbit Order を提案した．Rabbit Order は階層コミュニティオーダリン

グと並列逐次集約によってそれぞれ高い局所性と短時間でのリオーダリングを同時に達成している．実験の結果，リオーダリングとグラフ処理を合わせた全体処理時間の観点で Rabbit Order は既存手法を大きく上回る性能を示すことが確認された．

Rabbit Order は本研究においてクラウド上に配備するクエリ処理エンジンのメモリアクセス効率を改善する手法として位置付けられる．SpMV において見られる隣接頂点情報へのメモリアクセスは，つながりに着目するグラフ処理において本質的なアクセスパターンである．2.5.5 節で実際に示したように，Rabbit Order は SpMV に限らず様々なグラフ処理に効果があることを確認できた．さらにリオーダリングはグラフの頂点順序を入れ替えるのみであるため，高速化しようとしているグラフ処理の実装はほぼ変更する必要がない．サブグラフマッチングとリオーダリングを組み合わせた評価は行っていないものの，以上の理由により組み合わせは容易であり，効果も他のグラフ処理と同様に発揮されることを期待できる．

実験では総合的には Rabbit Order の高い高速化率を確認することができた一方で，2.5 節で確認したように適用対象のグラフによって不均一な高速化率を示すことも明らかになった．これは局所性向上率がグラフが内包するコミュニティ構造に依存していることに由来すると考えられる．例えば図 2.7 に示されているように，twitter グラフにおいて Rabbit Order を含むリオーダリング手法は PageRank の処理時間をほぼ削減できていない．これは twitter グラフが持つ特異な次数分布と複雑な構造に由来すると考えられる [55]．しかし Wei らの手法は，リオーダリングに長い時間を要するものの，twitter グラフにおいて約 1.4 倍程度の性能向上を達成したことが報告されている [92]．従ってリオーダリングによる性能向上の余地自体は存在することが分かる．このように効果がグラフの性質に依存する点に Rabbit Order の限界があるといえる．2.2 節で紹介したように，リオーダリング以外にも局所性を向上するためのアプローチは存在する．そのため，実行時リオーダリングの適用が適当なグラフの性質を定性的に評価しつつ，代替となる手法を検討することが今後の課題である．

アルゴリズム 3 並列逐次集約によるコミュニティ検出

入力: グラフ $G = (V, E)$

出力: トップレベル頂点の集合 *Toplevel*，および *atom* と *sibling* で表現されるデンドログラム．

1: **for each** $u \in V$ **do**
2: $atom[u].degree \leftarrow d(u)$ ▷ 重み付き次数
3: $atom[u].child \leftarrow$ `INVALID_ID` ▷ u にマージされた頂点
4: $sibling[u] \leftarrow$ `INVALID_ID` ▷ 同じコミュニティに属する頂点
5: $dest[u] \leftarrow u$ ▷ u が所属するコミュニティ
6: $Toplevel \leftarrow \emptyset$ ▷ トップレベル頂点の集合
7: **while** 集約元として未選択の頂点が存在する **do in parallel**
8: $u \leftarrow$ 未選択の頂点のうち次数が最小のもの
9: $degree_u \leftarrow$ `INVALID_DEGREE` ▷ 集約不能を表す無効値を代入
10: AtomicSwap($degree_u, atom[u].degree$) ▷ u を集約不能に設定
11: $v \leftarrow$ FindBestDestination(u)
12: **if** $\Delta Q(u, v) \leq 0$ **then** ▷ modularity が向上しない場合
13: $atom[u].degree \leftarrow degree_u$ ▷ 値の復元
14: $Toplevel \leftarrow Toplevel \cup \{u\}$ ▷ u はトップレベル頂点
15: **continue** ▷ 次の頂点へ
16: $atom_v \leftarrow atom[v]$
17: **if** $atom_v.degree \neq$ `INVALID_DEGREE` **then** ▷ v は集約可能か
18: $sibling[u] \leftarrow atom_v.child$ ▷ *child* の上書き前に値を移動
19: $atom'_v.degree \leftarrow atom_v.degree + degree_u$ ▷ 次数更新
20: $atom'_v.child \leftarrow u$ ▷ 最後に集約された頂点として記録
21: CAS: $atom[v] = atom_v$ なら $atom[v]$ と $atom'_v$ を入れ替え
22: **if** CAS で値の入れ替えが実行された **then**
23: $dest[u] \leftarrow v$ ▷ u は v へ集約された
24: **continue** ▷ 次の頂点へ
25: $sibling[u] \leftarrow$ `INVALID_ID` ▷ リトライのため元の値を復元
26: $atom[u].degree \leftarrow degree_u$ ▷ リトライのため元の値を復元
27: u を後でリトライする頂点として保存
28: リトライのため 27 行目で保存された頂点をそれぞれ集約

アルゴリズム 4 FindBestDestination 関数

```
1: function FindBestDestination(u)
2:     デンドログラムを辿り u のコミュニティメンバを集合 C に収集
3:     for each v ∈ N(C) do
4:         while dest[dest[v]] ≠ dest[v] do
5:             dest[v] ← dest[dest[v]]
6:     N' ← { dest[t] | t ∈ N(C) }
7:     for each v ∈ N' do
8:         w_uv ← Σ_{{(s,t)∈E | s∈C, dest[t]=v}} w_st
9:     E ← (E \ {(s,t) | s ∈ C, t ∈ N(s)}) ∪ {(u,v) | v ∈ N'}
10:    return argmax_{v∈N'} ΔQ(u,v)
```

第 3 章

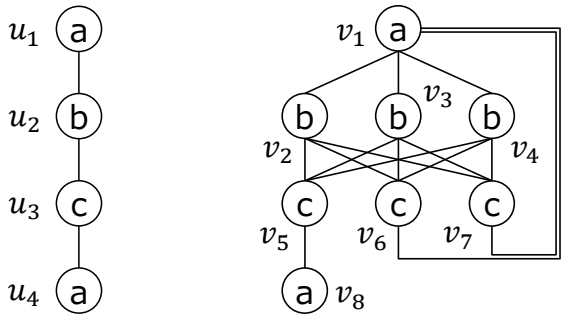
高速サブグラフマッチング

3.1 序論

計算に要する時間は CPU の命令処理効率と発行命令数によって決まる．第 2 章ではグラフ処理のメモリアクセスを効率化するリオーダリング手法を提案した．リオーダリングによって CPU のストールが減少し，命令処理効率を高めることができる．本章ではさらにグラフクエリ処理に焦点を絞り，発行命令数の削減による高速化を目指す．

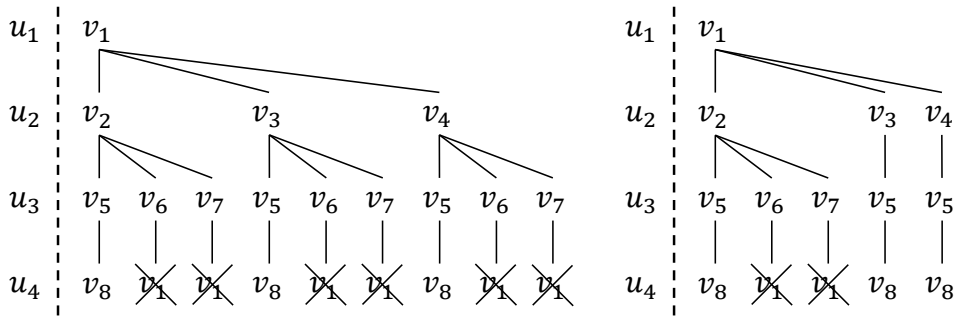
本研究においてグラフクエリとは，具体的にはサブグラフマッチングクエリを意味する．サブグラフマッチングとは大きいグラフ (データグラフ) 中に存在するクエリグラフと同型なサブグラフへの埋め込みを列挙する問題である．サブグラフマッチングはグラフ用データベースにおいて代表的なクエリ言語である SPARQL, Cypher, Gremlin などによって基本的なクエリとしてサポートされている [90]．さらにデータ分析の分野においても，人間関係の分析 [31]，営業活動の支援 [45]，マルウェアの検出 [76] など様々なアプリケーションにサブグラフマッチングは利用されている．特に頂点が表現する事物の種類 (人物，企業，商品など) を頂点ラベルとして持つようなラベル付きグラフに対する適用が盛んである [31,45,76]．しかしながらサブグラフマッチングは NP 困難問題であるため，計算に長い時間を要することが課題となっている [42]．

このような背景からサブグラフマッチングの高速化を目指して多くの研究が行



(a) クエリグラフ Q　　(b) データグラフ G

図 3.1: クエリグラフとデータグラフの例．u_i, v_i は頂点の識別子を，頂点内のアルファベットは頂点ラベルを表す．



(a) 既存手法 (CFL-Match [7] など)　　(b) 提案手法

図 3.2: 図 3.1 の Q と G に対して行われるバックトラッキングの過程を示すツリー．バツ印は探索失敗を表す．

われてきた [7, 20, 42, 43, 81, 83, 97]．これらの手法はいずれもバックトラッキングによる探索 [89] を行う．バックトラッキングは既に得られている部分的な解から新しい解を生成することで探索を行う再帰的な手法である．サブグラフマッチングの場合，クエリグラフの頂点 (クエリ頂点) の一部のみがデータグラフの頂点 (データ頂点) へ割り当てられた部分埋め込みが部分的な解に相当する．未割り当てのクエリ頂点からデータ頂点への割り当てを部分埋め込みに追加することで新しい埋め込みが生成される．新しい埋め込みが全てのクエリ頂点の割り当てを含む完全な埋め込みならば解として報告する．一方で新しい埋め込みから完

全な埋め込みを得られないことが判明した場合は探索失敗となる．失敗した埋め込みを破棄し，前のステップへ戻り別の部分埋め込みについて探索を再開する．バックトラッキングによる探索の過程はツリーとして表現することができる．例えば図 3.1 のクエリグラフ Q とデータグラフ G のサブグラフマッチングでは，図 3.2(a) のような探索過程が得られる．最も左側の過程 v_1-v_2-v_5-v_8 は，u_1 から u_4 を順にそれぞれのデータ頂点へ割り当てる探索を表す．ここでは完全な埋め込みを得られる．次に u_3 の割り当てまで戻り，割り当て先を v_6 へ変更する．v_6 の隣接頂点である v_1 は u_1 の割り当て先として使用済みであるため，探索失敗となる．そこで u_3 の割り当て先を再度変更し，以降も同様に探索を行う．サブグラフマッチングはこのような再帰的な手続きによって行われる．

バックトラッキングにおいて解に至らない部分埋め込みに対する処理は無駄になるため，サブグラフマッチングの性能を向上させるためには探索失敗の削減が必要である．そこで既存手法はバックトラッキングの前処理としてクエリグラフとデータグラフの構造を分析することで探索空間を狭めている．例えば頂点の次数や隣接頂点のラベルなど局所的な情報を比較することで，クエリ頂点 u_i の割り当て先となり得るデータ頂点を候補頂点集合 $C[u_i]$ として事前に抽出する手法は一般的に用いられている [7, 42, 43, 97]．より多くの探索失敗を削減できるよう前処理を改良することでサブグラフマッチングは高速化されてきた．

しかしながら，前処理を用いてもバックトラッキング時の探索失敗は発生する．前処理はバックトラッキングに追加されるオーバーヘッドとなるため複雑な処理を用いることが難しく，探索空間の削減範囲が限られる．例えば既存手法の前処理では，実際にはクエリ頂点 u_i の割り当て先となることができないデータ頂点が偽陽性として候補頂点集合 $C[u_i]$ に含まれてしまう場合がある．ナイーブにはサブグラフマッチングの探索空間は各クエリ頂点 $u_1, u_2, \ldots, u_n$ に対する候補頂点の組み合わせ $C[u_1] \times C[u_2] \times \ldots \times C[u_n]$ である．そのため組み合わせ爆発により僅かな偽陽性でも性能に大きく影響する．実際，図 3.1 の Q と G において，既存手法 [7, 42, 43, 97] は頂点ラベル以外の条件により候補頂点を絞り込むことができない．そのため図 3.2(a) と同様の探索失敗が発生する．この探索では v_1 の重複使用による探索失敗を u_2 と u_3 の割り当て先を変更しながら繰

返し発生させている．既存手法ではこのように無駄な探索が発生してしまう．

以上のような問題点を踏まえ，本章では前処理ではなくバックトラッキングの効率化に着目した新しいサブグラフマッチングアルゴリズムを提案する．提案手法の中心的なアイデアは「失敗から学ぶ」ことによる枝刈りである．具体的には，提案手法は探索失敗となった部分埋め込みから失敗の原因となる頂点の割り当てを抽出し，失敗パターンとして記録する．そして後続のバックトラッキングにおいて処理中の部分埋め込みが既出の失敗パターンと符合したとき，その探索を打ち切る．提案手法の動作を図 3.1 の Q と G を例にとって示す．図 3.2(b) は提案手法による探索の過程を表すツリーである．まずクエリ頂点をそれぞれ v_1, v_2, v_6, v_1 へ割り当てた場合，前述の通り v_1 の重複使用により探索失敗となる．ここで v_6 に隣接するラベル a の頂点は v_1 のみであるから，(u_1, v_1) という割り当てと同時に (u_3, v_6) という割り当てを行った時点で探索失敗となることは確定している．そこで提案手法は $\{(u_1, v_1), (u_3, v_6)\}$ を失敗パターンとして記録する．同様に u_3 を v_7 へ割り当てた場合も探索失敗となるため，失敗パターン $\{(u_1, v_1), (u_3, v_7)\}$ を記録する．次にバックトラッキングにより u_2 の割り当て先が v_3 へ変更される．ここで u_3 を v_6 または v_7 に割り当てた部分埋め込みは記録された失敗パターンに符合するため，探索が枝刈りされる．これにより探索失敗が減少する．u_2 を v_4 へ割り当てた場合も同様である．既存手法が候補頂点の全組み合わせを生成する (図 3.2(a)) のに対し，提案手法は探索失敗を繰り返さないため高速である．

提案手法の特長は次の 3 つにまとめられる：

1. **ロバスト：** 既存手法がクエリグラフとデータグラフの構造を分析することで探索失敗を予防するのに対し，提案手法はバックトラッキング中の情報を利用して高速化する．そのため，図 3.1 のように既存手法が機能しないグラフにおいても探索失敗を削減することができる．つまり，処理効率が与えられたグラフに左右されにくいという意味でロバスト性が高い．
2. **スケーラブル：** クエリグラフとデータグラフが大規模であるほど埋め込みの組み合わせ数は増加するため，処理時間の増大が急激である．提案手

法はロバスト性が高いことから，グラフの規模に対して時間計算量的にスケーラブルである．

3. **厳密：** 提案手法は無駄な探索のみを枝刈りすることが証明されている．つまり全ての埋め込みを列挙する厳密手法である．

実験の結果，提案手法は既存手法と比べ最大 4 桁以上高速であることが確認された．既存手法では処理に 1 日以上を要するクエリセットに対しても提案手法は高速に応答可能であるため，これまで事実上不可能だったクエリにも対応することができる．

本章の構成は次の通りである．第 3.2 節で関連研究を紹介し，本研究の前提知識を第 3.3 節で説明する．第 3.4 節で手法の詳細を述べ，第 3.5 節で実験結果を示す．最後に第 3.6 節で結論を述べる．

3.2 関連研究

サブグラフマッチングの研究はいずれもサブグラフ同型問題を扱うが，その問題設定は大きく 2 つに分類される．

1 つ目は単一の大きなデータグラフからクエリグラフと同型なサブグラフを列挙する問題である．本章ではこちらの問題設定を扱う．この分野ではバックトラッキングに基づく古典的なアルゴリズムである Ullmann [89] がよく知られている．Ullmann 以降，さらに高速化するために候補頂点枝刈りとマッチング順序最適化という 2 つのテクニックを中心に多くの研究がなされてきた．1 つ目の候補頂点枝刈りとは，クエリ頂点 u_i の候補頂点集合 $C[u_i]$ に含まれるデータ頂点を削減するためのフィルタリング手法である．Ullmann [89] では頂点のラベルと次数に基づくフィルタを行う．GraphQL [43] と SPath [97] はさらにクエリグラフとデータグラフの局所的な隣接関係を用いた軽量なマッチングをフィルタに用いる．2 つ目のテクニックであるマッチング順序最適化とは，バックトラッキングにおいて部分埋め込みへクエリ頂点を追加していく順番を変更することで高速化する手法である．バックトラッキングでは前のステップで生成された部分埋め込みから新しい部分埋め込みを生成するため，部分埋め込みの生

成数が少ないクエリ頂点から順に割り当て先を決定することで探索空間を狭められる．VF2 [20] は割り当て済みクエリ頂点が常に 1 つの連結成分になるようにマッチング順序を最適化し，QuickSI [83] はさらにデータグラフにおけるラベルの出現頻度を元に希少度の高い頂点とエッジから順に割り当てを行う．近年は候補頂点枝刈りとマッチング順序最適化を同時に行う手法も提案されている．TurboISO [42] と CFL-Match [7] はクエリグラフの全域木に基づく近似的なマッチングにより候補頂点を枝刈りし，さらに候補頂点の接続関係を踏まえクエリグラフ中の推定埋め込み数が少ない部分から割り当てを行う．以上の既存手法はいずれもバックトラッキングの探索空間を縮小するためにその前処理を工夫している．これに対し，提案手法はバックトラッキングの最中に枝刈りを行うことにより，前処理が十分に機能しないようなクエリグラフとデータグラフの組み合わせにおいても処理を高速化することができる．また既存手法と提案手法はほぼ直交しており，両者を組み合わせて使用することも可能である．

2 つ目の問題設定は多数の (大抵は小規模な) データグラフに対する，クエリグラフと同型なサブグラフを含むかどうかの判定，あるいは同型なサブグラフの列挙である．この問題設定においては，まずデータグラフのインデックス構造を用いて確実にクエリグラフを含まないデータグラフを除外し (filter)，次に残ったデータグラフがクエリグラフを含むことを確認する (verify) という 2 段階の処理が一般的である．インデックスの情報としてはデータグラフに含まれるパス [10,38] や頻出サブグラフ [16] の利用が提案されている．この問題設定においても verify 処理に提案手法を用いて高速化することが可能である．

3.3　事前準備

本節では本章で扱う問題，および以降の説明で用いる表記法を定義する．

3.3.1　問題定義

本章では頂点がラベルを持つ無向グラフ $G = (V_G, E_G, \Sigma, \ell)$ を扱う．V_G は頂点の集合，$E_G \subseteq V_G \times V_G$ はエッジの集合，Σ はラベルの集合，ℓ は頂点とラベ

ルを対応づける関数である．G を特に**データグラフ**と呼ぶ．同様に**クエリグラフ** $Q = (V_Q, E_Q, \Sigma, \ell)$ を考える．クエリグラフの頂点は $u_1, u_2, \ldots, u_n$ というように番号付けされているものとする．

定義 1 (サブグラフ同型)．次の 3 つの制約[*1]を満たす埋め込み $M : V_Q \to V_G$ を定義できるとき，Q は G に対して**サブグラフ同型**であるという：
(1) **ラベル制約**： $\forall u_i \in V_Q,\ \ell(u_i) = \ell(M[u_i])$
(2) **エッジ制約**： $\forall (u_i, u_i') \in E_Q,\ (M[u_i], M[u_i']) \in E_G$
(3) **単射制約**： $\forall u_i, u_i' \in V_Q,\ u_i \neq u_i' \Rightarrow M[u_i] \neq M[u_i']$

定義 2 (サブグラフマッチング)．クエリグラフ Q とデータグラフ G が与えられたとき，G に含まれる Q のサブグラフ同型な埋め込みを全て列挙する問題を**サブグラフマッチング**と呼ぶ．

本章はサブグラフマッチングを高速に解く手法を与える．ただし実際には組み合わせ爆発によって膨大な数の埋め込みが生じる場合があるため，その全てを列挙することは現実的でない．そこで一定数，例えば 1,000 個の埋め込みを列挙した時点でクエリ処理を打ち切ることが一般的である [7, 42, 59]．

3.3.2 諸定義

本章で使用する主なシンボルを表 3.1 に示す．さらにいくつかの定義を行う．

定義 3 (埋め込みの表現)．データグラフ G に含まれるクエリグラフ Q の埋め込み $M : V_Q \to V_G$ を，クエリ頂点 u_i とデータ頂点 v のペアの集合で表す．u_i の割り当て先が v である (つまり $v = M[u_i]$) とき $(u_i, v) \in M$ である．

定義 4 (埋め込みの定義域と値域)．埋め込み M の定義域と値域をそれぞれ次のように書く: $\mathrm{dom}(M) = \{\, u_i \mid (u_i, v) \in M \,\}$, $\mathrm{ran}(M) = \{\, v \mid (u_i, v) \in M \,\}$.

例えば図 3.1 の Q と G において，$M = \{(u_1, v_1), (u_2, v_2), (u_3, v_5), (u_4, v_8)\}$

[*1] それぞれの制約の名前は以降の説明のために本研究で便宜上与えたものであり，一般的ではない．

表 3.1: 本章で用いる主なシンボル

シンボル	定義
Q, G	クエリグラフ, データグラフ
V_Q, V_G	クエリグラフとデータグラフの頂点集合
E_Q, E_G	クエリグラフとデータグラフのエッジ集合
u_i, v	クエリ頂点 ($u_i \in V_Q$) とデータ頂点 ($v \in V_G$)
$C[u_i]$	u_i の候補頂点集合
$M, \hat{M}$	完全な埋め込みと部分埋め込み
D, Γ	Dead-end パターンと dead-end マスク
$N(\cdot)$	頂点の隣接頂点集合
$\ell(\cdot)$	頂点のラベル
$\mathrm{dom}(\cdot), \mathrm{ran}(\cdot)$	埋め込みの定義域と値域
$\mathrm{Dead}(\cdot)$	述語: 埋め込みは dead-end パターンであるか？

は埋め込みである．また $\mathrm{dom}(M) = \{u_1, u_2, u_3, u_4\}$, $\mathrm{ran}(M) = \{v_1, v_2, v_5, v_8\}$ である．

定義 5 (完全な埋め込み)**.** 埋め込み M について，$\mathrm{dom}(M) = V_Q$ であるときに限り，M を**完全な埋め込み**と呼ぶ**.**

なお埋め込みが完全な埋め込みでない (かもしれない) ことを強調する文脈においては，埋め込みを $\hat{M}$ と表記し，$\hat{M}$ を**部分埋め込み**と呼ぶ．完全な埋め込みも部分埋め込みの一種として扱うことに注意されたい**.**

3.3.3　バックトラッキングによるサブグラフマッチング

サブグラフマッチング手法は近年提案されたものも含めバックトラッキングによる探索 [59, 89] を行うものが主流である．ナイーブなバックトラッキングによるサブグラフマッチングアルゴリズムをアルゴリズム 5 に示す．再帰関数 Search は部分埋め込み $\hat{M}$ と候補頂点集合 C を受け取る．C の抽出方法としては，最も単純には次のように u_i とラベルが一致するデータ頂点を候補頂点と

アルゴリズム 5 バックトラッキングによる探索

入力: クエリグラフ Q，データグラフ G

出力: G に含まれる全ての Q の埋め込みを報告する．

```
1: function Search(M̂, C)
2:     k ← |M̂|
3:     if k = |V_Q| then
4:         M̂ を完全な埋め込みとして報告する
5:         return
6:     C′ ← C のうち M̂ についてエッジ制約を満たす候補頂点 (式 3.2)
7:     for each v ∈ C′[u_{k+1}] do
8:         if v ∉ ran(M̂) then
9:             Search(M̂ ∪ {(u_{k+1}, v)}, C′)
```

する方法がある [89]：

$$C[u_i] = \{\, v \in V_G \mid \ell(v) = \ell(u_i) \,\}. \tag{3.1}$$

関数呼び出し $\textsc{Search}(\emptyset, C)$ によって探索が開始される．関数は最初に $\hat{M}$ が完全な埋め込みかどうかを確認し，そうであれば完全な埋め込みとして報告する (2–5 行目)．$\hat{M}$ が完全な埋め込みでない場合，次の再帰呼び出しで使用するため C からエッジ制約を満たさない候補頂点を除外する (6 行目)．エッジ制約は，クエリ頂点 u_i と u'_i が隣接するならばそれらの割り当て先 $M[u_i]$ と $M[u'_i]$ も隣接していることを要求する．従ってクエリ頂点 u_i の割り当て先は，$\hat{M}$ に含まれる全ての隣接頂点 u'_i の割り当て先と隣接する候補頂点でなければならない．つまり，

$$C'[u_i] = C[u_i] \cap \bigcap\nolimits_{u'_i \in N(u_i) \cap \mathrm{dom}(\hat{M})} N(\hat{M}[u'_i]). \tag{3.2}$$

$\hat{M}$ には $u_1, u_2, \ldots, u_k$ の割り当てが含まれるので，次は u_{k+1} の候補頂点 v についてループする (7 行目)．v が $\hat{M}$ で使用されていないならば (8 行目)，$\hat{M}$ に u_{k+1} と v の割り当てを追加した部分埋め込み $\hat{M} \cup \{(u_{k+1}, v)\}$ と C' を引数として関数 Search を再帰呼び出しする．このようにラベル制約 (式 3.1)，エッジ

制約 (6 行目)，単射制約 (8 行目) を満たす割り当てを再帰的に部分埋め込みへ追加していくことにより完全な埋め込みを探索する．

3.4　提案手法

本節ではまず提案手法の中心的なアイデアとアルゴリズムの全体像を説明し，その後アルゴリズムの詳細を述べる．

3.4.1　提案手法のアイデア

候補頂点枝刈りやマッチング順序最適化のようなバックトラッキングの前処理によって探索範囲を狭めることができるが，依然として多くの探索失敗が発生し処理時間を増大させる．提案手法のアイデアはバックトラッキング中に発生した探索失敗から学び，失敗を繰り返さないことによって探索を高速化することである．具体的には，探索失敗は部分埋め込みに含まれる頂点の割り当てが定義 1 で示した 3 つの制約のいずれかに違反することで発生する．違反の原因となる割り当てを部分埋め込みが含んでいる限り，それ以外のクエリ頂点の割り当てをどのように変更しても探索失敗となる．そこで提案手法では制約に違反する割り当ての組み合わせ (パターン) を探索失敗時に部分埋め込みから抽出し，それに符合する部分埋め込みを枝刈りすることによって以降の探索失敗を削減する．既存手法による探索範囲の絞り込みと提案手法を組み合わせることによって，既存手法では削減しきれなかった探索失敗を削減し高速化することが可能である．

3.4.2　Dead-end パターンによる枝刈り

本節ではナイーブなバックトラッキングによる探索 (アルゴリズム 5) を参照しつつ提案手法による探索の枝刈りを説明する．まず以降の説明で使用するため次の通り定義を行う．

定義 6 (Dead-end パターン)**.** D を部分埋め込みとする．$D \subseteq M$ であるような完全な埋め込み M が存在しないとき，D を *dead-end* **パターン**と呼ぶ．記述を

簡潔にするため，単に「D は dead-end である」とも書く．また D が dead-end である場合に限り $\mathrm{Dead}(D)$ が真になるような述語 Dead を導入する．

Dead-end (行き止まり) とは図 3.2 のような探索ツリーにおける探索失敗 (図中バツ印) を指す．Dead-end パターンを利用した枝刈りを行うため，提案手法は関数 Search に対して (i) 関数末尾における dead-end パターンの抽出と (ii) 再帰呼び出しの前 (8, 9 行目の間) における dead-end パターンとの照合という 2 つの変更を加える．1 つ目の dead-end パターンの抽出では部分埋め込み $\hat{M}$ から一部の頂点割り当てを dead-end パターン D として抽出し，dead-end パターンの集合 $\mathcal{D}$ へ追加する．具体的な抽出方法は後述する．Dead-end パターンの抽出は関数呼び出し $\textsc{Search}(\hat{M}, C)$ またはそこからの再帰呼び出し内において埋め込みの報告がなかった場合にのみ行われる．もし埋め込みの報告があったならば $\hat{M}$ に関する探索は成功したことになるため，$\hat{M}$ から dead-end パターンを抽出することはできない．2 つ目の dead-end パターンとの照合では，新しい部分埋め込み $\hat{M} \cup \{(u_{k+1}, v)\}$ が $D \in \mathcal{D}$ のいずれかに符合するかどうかを確認する．

定義 7 (Dead-end パターンとの符合)**.** 部分埋め込み $\hat{M}$ と dead-end パターン D が与えられたとき，$D \subseteq \hat{M}$ であるならば「$\hat{M}$ は D に符合する」と言う．

もし dead-end パターンに符合するならば再帰を行わないことで探索を打ち切る．このような dead-end パターンの抽出と照合がナイーブなバックトラッキングに対する差分である．

3.4.3 Dead-end パターンの抽出

前述の通り部分埋め込み $\hat{M}$ が dead-end パターン抽出の対象となるとき，$\hat{M}$ を含む完全な埋め込みは存在しない．従って定義上 $\hat{M}$ 自体も dead-end パターンである．しかし dead-end パターンとしての $\hat{M}$ に符合する部分埋め込みは関数呼び出し $\textsc{Search}(\hat{M}, C)$ において探索済みであるため，$\hat{M}$ に枝刈り効果はない．以降の探索において多くの部分埋め込みを dead-end パターンと符合させるため，パターンは少数のクエリ頂点に対する割り当てのみを含むことが望ま

しい．しかし $\hat{M}$ から一部の割り当てのみを取り出した部分埋め込みが必ずしも dead-end であるとは限らない．そこで dead-end パターンの抽出では，$\hat{M}$ の部分集合であり，かつ依然として dead-end でもあるような部分埋め込みを作成する．その際次のような dead-end マスクという考え方を用いる．

定義 8 (Dead-end マスク)**.** $\hat{M}$ を dead-end である部分埋め込みとする．このとき次式を満たすクエリ頂点の集合 Γ を $\hat{M}$ の *dead-end* **マスク**と呼ぶ．

$$\Gamma \subseteq \mathrm{dom}(\hat{M}) \ \wedge \ \mathrm{Dead}\left(\left\{ (u_i, v) \in \hat{M} \,\middle|\, u_i \in \Gamma \right\}\right) \tag{3.3}$$

$\hat{M}$ から抽出された Dead-end パターン D は，$\hat{M}$ と Γ を用いて $D = \left\{ (u_i, v) \in \hat{M} \,\middle|\, u_i \in \Gamma \right\}$ によって得られる．

Dead-end マスクの選択方法は探索失敗の原因によって異なる．アルゴリズム 5 において探索失敗となる原因は次の 3 つがある: (i) ラベル制約 (式 3.1) とエッジ制約 (式 3.2) を満たす候補頂点 $v \in C'[u_{k+1}]$ が存在しない (7 行目)，(ii) 新しい部分埋め込み $\hat{M} \cup \{(u_{k+1}, v)\}$ が単射制約を満たさない (8 行目)，(iii) 再帰呼び出しした関数 Search が探索に失敗する (9 行目)．さらに提案手法では dead-end パターンとの照合を 8, 9 行目の間で行うため，探索失敗となる原因は合わせて 4 つ存在する．以降ではこれら 4 つの原因それぞれについて dead-end マスクの選択方法を説明し，最後に失敗の原因別に得られたマスクを集約し $\hat{M}$ のマスクを得る方法を述べる．

候補頂点が存在しない場合

候補頂点が存在しない状態とは $C'[u_{k+1}] = \emptyset$ である．実際には u_{k+1} に限らず，いずれかのクエリ頂点が候補頂点を失った時点で $\hat{M}$ の探索失敗が確定する．このとき，dead-end マスクは次のように得られる．

補題 1 (候補頂点が存在しない場合の dead-end マスク)**.** クエリ頂点 u_i について $C'[u_i] = \emptyset$ であるとき，$\Gamma = N(u_i) \cap \mathrm{dom}(\hat{M})$ は $\hat{M}$ の dead-end マスクである．

証明. Γ が dead-end マスクでない，即ち式 3.3 を満たさないと仮定する．こ

のとき $\Gamma \subseteq \mathrm{dom}(\hat{M})$ は自明なので，$\{\,(u_j, v) \in \hat{M} \,|\, u_j \in \Gamma\} \subseteq M$ であるような埋め込み M が存在する．M はエッジ制約を満たす埋め込みなので，式 3.2 の $\hat{M}$ に M を代入して $C'_M[u_i] = C[u_i] \cap \bigcap_{u'_i \in N(u_i)} N(M[u'_i])$ と置くと $M[u_i] \in C'_M[u_i]$．一方 $\Gamma \subseteq N(u_i)$ 及び $\forall u_j \in \Gamma, M[u_j] = \hat{M}[u_j]$ より，$C'_M[u_i] \subseteq C[u_i] \cap \bigcap_{u'_i \in \Gamma} N(\hat{M}[u'_i]) = C'[u_i] = \emptyset$ である．従って $M[u_i] \in C'_M[u_i]$ と矛盾する．ゆえに Γ は式 3.3 を満たす dead-end マスクである． □

単射制約を満たさない場合

説明のため新しい部分埋め込みを $\hat{M}_+ = \hat{M} \cup \{(u_{k+1}, v)\}$ と置く．アルゴリズム 5 では $v \notin \mathrm{ran}(\hat{M})$ (8 行目) によって $\hat{M}_+$ が単射制約を満たすことを確認する．$\hat{M}_+$ が単射制約を満たさない場合の dead-end マスクは次のように得られる．

補題 2 (単射制約を満たさない場合の dead-end マスク)**.** $\hat{M}_+$ を単射制約を満たさない部分埋め込みとする．単射でないので $\hat{M}_+[u_{i_1}] = \hat{M}_+[u_{i_2}]$ であるようなクエリ頂点 u_{i_1}, u_{i_2} が存在する．ここで $\Gamma_+ = \{u_{i_1}, u_{i_2}\}$ は $\hat{M}_+$ の dead-end マスクである．

証明. $\{\,(u_i, v) \in \hat{M}_+[u_i] \,|\, u_i \in \Gamma_+\,\} \subseteq M$ であるような埋め込み M は単射制約を満たさない．また $\Gamma_+ \subseteq \mathrm{dom}(\hat{M}_+)$．ゆえに Γ_+ は $\hat{M}_+$ の dead-end マスクである． □

補題 1 が $\hat{M}$ の dead-end マスクを与えるのに対し，補題 2 は $\hat{M}_+$ のマスクを与えることに注意されたい．$\hat{M}_+$ のマスクを $\hat{M}$ のマスクへ変換する方法は後述する．

Dead-end パターンに符合する場合

新しい部分埋め込み $\hat{M}_+$ が記録済みの dead-end パターン D に符合する場合，dead-end マスクは D より得られる．

補題 3 (Dead-end パターンに符合する場合の dead-end マスク)**.** D を dead-end

パターンとする．$D \subseteq \hat{M}_+$ であるとき，$\Gamma_+ = \mathrm{dom}(D)$ は $\hat{M}_+$ の dead-end マスクである．

証明. $D \subseteq \hat{M}_+$ より $\Gamma_+ \subseteq \mathrm{dom}(\hat{M}_+)$．$D$ は dead-end パターンなので $\mathrm{Dead}(D)$．ゆえに Γ_+ は $\hat{M}_+$ の dead-end マスクである． □

再帰呼び出しが探索失敗となる場合

アルゴリズム 5 は新しい部分埋め込み $\hat{M}_+$ について単射制約の確認と dead-end パターンへの照合を行ったのち，関数 SEARCH を再帰呼び出しする．再帰呼び出しから戻るまでに埋め込みの報告がなかった場合，$\hat{M}_+$ は dead-end であったことが分かる．このとき，$\hat{M}_+$ からの dead-end パターンの抽出と記録は再帰呼び出しされた関数内で完了している．つまり，再帰呼び出しから戻ったときには $\hat{M}_+$ と符合する dead-end パターンが記録されている．従って補題 3 と同様にして $\hat{M}_+$ の dead-end マスクを得られる．

Dead-end マスクの集約

関数 SEARCH は部分埋め込み $\hat{M}$ から dead-end パターンを抽出するために $\hat{M}$ の dead-end マスクを必要としている．補題 1 では直接 $\hat{M}$ のマスクを得られるのに対し，補題 2 及び 3 で得られるのは新しい部分埋め込み $\hat{M}_+$ のマスクである．式 3.3 より $\hat{M}$ のマスク Γ は $\Gamma \subseteq \mathrm{dom}(\hat{M}) = \{u_1, u_2, \ldots, u_k\}$ を満たすが，$\hat{M}$ に u_{k+1} の頂点割り当てを追加することで作られた $\hat{M}_+$ のマスクは u_{k+1} を含む可能性がある．従って $\hat{M}_+$ のマスクから $\hat{M}$ のマスクを得るためには変換が必要である．

補題 2 と 3 は $v \in C'[u_{k+1}]$ についてのループ (7 行目) の内部で適用されるため，ループ内で生成される新しい部分埋め込み $\hat{M}_+$ のそれぞれに対応するマスクが得られる．従って次のような集合 Γ_* を得ることが可能である:

$$\Gamma_* = \bigcup_{v \in C'[u_{k+1}]} \left(\hat{M} \cup \{(u_i, v)\} \text{ の dead-end マスク} \right). \tag{3.4}$$

ここで Γ_* について次のことが言える．

補題 4 (Dead-end マスクの集約)**.** $C'[u_{k+1}] \neq \emptyset$ であるとき，次の Γ は $\hat{M}$ の dead-end マスクである:

$$\Gamma = \begin{cases} \Gamma_* & \text{if } u_{k+1} \notin \Gamma_* \\ (\Gamma_* \cup N(u_{k+1})) \cap \mathrm{dom}(\hat{M}) & \text{if } u_{k+1} \in \Gamma_*. \end{cases} \tag{3.5}$$

証明. $D = \left\{ (u_i, v) \in \hat{M} \;\middle|\; u_i \in \Gamma \right\}$ と置く．$\Gamma \subseteq \mathrm{dom}(\hat{M})$ と $\mathrm{Dead}(D)$ を，$u_{k+1} \notin \Gamma_*$ と $u_{k+1} \in \Gamma_*$ の場合ぞれぞれについて示す．

■$u_{k+1} \notin \Gamma_*$ **のとき**　補題の前提として，任意の $v' \in C'[u_{k+1}]$ について新しい部分埋め込み $\hat{M}_+ = \hat{M} \cup \{(u_{k+1}, v')\}$ は dead-end であるため，その dead-end マスク Γ_+ が存在する．$\Gamma_+ \subseteq \mathrm{dom}(\hat{M}_+)$ であり，今は $u_{k+1} \notin \Gamma_*$ の場合を考えているので，$\Gamma_* \subseteq \mathrm{dom}(\hat{M}_+) \setminus \{u_{k+1}\}$．また $\Gamma_* = \Gamma$ および $\mathrm{dom}(\hat{M}_+) \setminus \{u_{k+1}\} = \mathrm{dom}(\hat{M})$ より $\Gamma \subseteq \mathrm{dom}(\hat{M})$．さらに $\Gamma_+ \subseteq \Gamma_*$ より，

$$\forall v' \in C'[u_{k+1}], \mathrm{Dead}\left(\left\{ (u_i, v) \in \left(\hat{M} \cup \{(u_{k+1}, v')\}\right) \;\middle|\; u_i \in \Gamma_* \right\}\right). \tag{3.6}$$

$u_{k+1} \notin \Gamma_*$ なので v' の全称量化子は除去できて $\mathrm{Dead}\left(\left\{ (u_i, v) \in \hat{M} \;\middle|\; u_i \in \Gamma_* \right\}\right)$．$\Gamma_* = \Gamma$ なので $\mathrm{Dead}(D)$．

■$u_{k+1} \in \Gamma_*$ **のとき**　まず Γ は $\Gamma_* \cup N(u_{k+1})$ と $\mathrm{dom}(\hat{M})$ の積集合なので $\Gamma \subseteq \mathrm{dom}(\hat{M})$．次に $\mathrm{Dead}(D)$ を背理法で考える．Γ が $\hat{M}$ の dead-end マスクでないと仮定すると，$D \subseteq M$ であるような完全な埋め込み M が存在する．$N(u_{k+1}) \cap \mathrm{dom}(\hat{M}) \subseteq \Gamma$ および $\forall u_i \in \Gamma, \hat{M}[u_i] = D[u_i] = M[u_i]$ より，

$$\forall u_i \in N(u_{k+1}) \cap \mathrm{dom}(\hat{M}), \hat{M}[u_i] = M[u_i]. \tag{3.7}$$

M はエッジ制約を満たすため，

$$M[u_{k+1}] \in C[u_{k+1}] \cap \bigcap_{u'_i \in N(u_{k+1})} N(M[u'_i]) \tag{3.8}$$

$$\subseteq C[u_{k+1}] \cap \bigcap_{u'_i \in N(u_{k+1}) \cap \mathrm{dom}(\hat{M})} N(\hat{M}[u'_i]) \quad \because \text{式 3.7} \tag{3.9}$$

$$= C'[u_{k+1}]. \quad \because \text{式 3.2} \tag{3.10}$$

ところで，$u_{k+1} \in \Gamma_*$ の場合において式 3.6 を変形すると，

$$\forall v' \in C'[u_{k+1}], \mathrm{Dead}\left(\left\{ (u_i, v) \in \hat{M} \;\middle|\; u_i \in \Gamma_* \setminus \{u_{k+1}\} \right\} \cup \{(u_{k+1}, v')\}\right). \tag{3.11}$$

$\Gamma_* \setminus \{u_{k+1}\} \subseteq \Gamma$ より，

$$\forall v' \in C'[u_{k+1}], \mathrm{Dead}\left(\left\{ (u_i, v) \in \hat{M} \;\middle|\; u_i \in \Gamma \right\} \cup \{(u_{k+1}, v')\}\right) \tag{3.12}$$

$$\Leftrightarrow \forall v' \in C'[u_{k+1}], \mathrm{Dead}\left(D \cup \{(u_{k+1}, v')\}\right). \tag{3.13}$$

ところが，完全な埋め込み M は $D \subseteq M$ かつ式 3.10 より $M[u_{k+1}] \in C'[u_{k+1}]$ を満たすので，式 3.13 と矛盾する．ゆえに $\mathrm{Dead}(D)$.

以上より，$u_{k+1} \notin \Gamma_*$ と $u_{k+1} \in \Gamma_*$ の両方の場合において $\Gamma \subseteq \mathrm{dom}(\hat{M})$ かつ $\mathrm{Dead}(D)$ であるから，Γ は $\hat{M}$ の dead-end マスクである． □

関数 SEARCH の末尾では以上のように抽出された dead-end マスク Γ を用いて dead-end パターンを $\hat{M}$ から抽出する．

3.4.4　Dead-end パターンの管理

3.4.2 節では簡単のため，dead-end パターンを集合 $\mathcal{D}$ に記録し，照合時には $\mathcal{D}$ に部分埋め込みと符合するパターンが存在するかどうかを確認すると述べた．しかしながら実際には空間的，時間的制約がそれぞれ存在し，実装は現実的でない．まず空間的には，dead-end パターンは最悪の場合存在し得る部分埋め込みと同数，即ち $|C[u_1]||C[u_2]|\ldots|C[u_n]|$ 個抽出される．そのため，$\mathcal{D}$ はメモリに収まらない可能性がある．さらに時間的には，性能向上のために部分埋め込み $\hat{M}$ と符合する dead-end パターンを高速に検索できなければならない．そこで本節では，実用的な枝刈りを可能にするための 2 つのテクニックを説明する．

連想配列への dead-end パターンの格納

探索中に抽出された全ての dead-end パターンを保存しておくことは，メモリ消費量とパターンの検索速度の両面において現実的でない．そこで dead-end パ

ターンの一部をキーとして使用し，連想配列へ保存する方法を取る．キーが一致したパターンは上書き保存することによって，一定数以上のパターンが保存されないようにする．具体的には，探索に失敗した部分埋め込みへ最後に追加されたクエリ頂点とその割り当て先データ頂点のペアをキーとする．つまり，Δ を連想配列，$\hat{M} = \{(u_1, v_{i_1}), (u_2, v_{i_2}), \ldots, (u_k, v_{i_k})\}$ を dead-end である部分埋め込みだとすれば，$\hat{M}$ から抽出されたパターンは $\Delta[u_k, v_{i_k}]$ に格納される．これにより，保存される dead-end パターンの数は候補頂点数の合計 $\sum_{u_i \in V_Q} |C[u_i]|$ 以下に保たれる．最後に割り当てたクエリ頂点とデータ頂点をキーにするのは，その組み合わせが最も上書きされる頻度が少ないためである．枝刈りのために dead-end パターンを検索する際は，部分埋め込みへ最後に割り当てたクエリ頂点とデータ頂点をキーとして連想配列を参照する．この処理は $O(1)$ で可能である．

Dead-end パターンの整数表現

Dead-end パターンを連想配列に保存することでメモリ消費と検索コストは実用レベルまで低下する．しかしながら，dead-end パターン D には最大で $|V_Q|$ 個の割り当てが含まれることから，部分埋め込み $\hat{M}$ との符合を確認する ($D \subseteq \hat{M}$ をテストする) ためには $O(|V_Q|)$ の時間計算量を要する．符合の確認は頻繁に行われるためこの計算コストの影響は大きく，枝刈り処理がかえって処理時間の増加を招く原因となる．

そこで提案手法では dead-end パターンを整数によって表現することで符合の確認を $O(1)$ 時間で行う．Dead-end パターンの整数表現のアイデアはバックトラッキングの再帰呼び出しの性質に基づく．具体的には，関数 Search に引数として渡される部分埋め込み $\hat{M}$ が呼び出しごとに必ず異なることを利用する．これにより，$\hat{M}$ を一意に識別する ID 番号 (埋め込み ID) として Search の通算呼び出し回数を使用することができる．部分埋め込み $\hat{M}$ からその埋め込み ID を対応付ける関数を $\nu(\hat{M})$ と置くと，例えば図 3.2(b) の探索において，$\nu(\{(u_1, v_1)\}) = 1$, $\nu(\{(u_1, v_1), (u_2, v_2)\}) = 2$, $\nu(\{(u_1, v_1), (u_2, v_2), (u_3, v_5)\}) = 3$, ... となる．

しかしながら，埋め込み ID を使った dead-end パターンの記録にはさらに 2 つの課題がある．1 つ目は，埋め込み ID で表現可能な部分埋め込みが必ず $u_1, u_2, \ldots$ と連続したクエリ頂点に対する割り当てを含むことである．そのため，dead-end マスクによって一部のクエリ頂点のみを抽出した dead-end パターンを表現できない．この問題の影響を緩和するため，dead-end パターンを連想配列へ格納していることを利用する．Dead-end パターンの連想配列 Δ において，$\hat{M}$ から抽出されたパターンの保存先は $\Delta[u_k, \hat{M}[u_k]]$ (ただし $k = |\hat{M}|$) に固定されている．その場所より自明であるため，dead-end パターンが $(u_k, \hat{M}[u_k])$ を含んでいることを明示する必要はない．そこで $\hat{M}$ の dead-end マスク Γ から u_k を除いて dead-end パターンを作成し，その埋め込み ID を連想配列へ格納する．2 つ目の課題は，部分埋め込みから埋め込み ID への変換である．埋め込み ID は関数 Search の通算呼び出し回数と同じ数だけ存在するため，全てを記録しておくことはできない．そこで現在探索中の部分埋め込み $\hat{M}$ に含まれる埋め込みについてのみ ID を保持する．具体的には，$\hat{M}$ と別に埋め込み ID の配列 Φ を定義し，$\Phi[\mu] = \nu(\{(u_i, v) \in \hat{M} \mid i \leq \mu\})$ となるように保持する．これには関数呼び出し $\textsc{Search}(\hat{M}, C)$ において $\Phi[u_k]$ へその時点の関数呼び出し回数を記録するだけでよい．

これら 2 つの課題とその対応策により，dead-end である部分埋め込み $\hat{M}$ とその dead-end マスク Γ が与えられたとき，dead-end パターンは (i) 埋め込み ID，(ii) 埋め込み ID が表す部分埋め込みの要素数，(iii) オリジナルの dead-end マスクの 3 つ組により表現される．具体的には次のように連想配列へ dead-end パターンを保存する:

$$\Delta[u_k, \hat{M}[u_k]] = (\Phi[\mu], \mu, \Gamma)\,. \tag{3.14}$$

ただし $\mu = \max_{u_i \in \Gamma, i<k} i$ である．3 つ組に含まれる dead-end マスク Γ は補題 3 に基づき dead-end パターンを抽出するために必要である．部分埋め込み $\hat{M}$ $(k = |\hat{M}|)$ とそれに対応する埋め込み ID の配列 Φ が与えられたとき，$\hat{M}$ が dead-end パターンに符合するかどうかは次のように確認できる:

$$\Phi[\mu] = \phi, \text{ ただし } (\phi, \mu, \Gamma) = \Delta[u_k, \hat{M}[u_k]]. \tag{3.15}$$

Δ の参照と埋め込み ID の比較は共に $O(1)$ なので，照合処理全体としても $O(1)$ である．複雑になるため次に示す詳細なアルゴリズムでは dead-end パターンの整数表現について省略しているが，内部で以上のような dead-end パターンの表現を用いることで枝刈りのオーバーヘッドを削減している．

3.4.5 詳細なアルゴリズム

ここまでに説明した dead-end パターンの抽出及び管理手法を統合した具体的な探索アルゴリズムをアルゴリズム 6 に示す．なおアルゴリズムは dead-end パターンの連想配列 Δ を大域変数として持つものとする．関数 SEARCH は部分埋め込み $\hat{M}$ と候補頂点集合 C を受け取り，$\hat{M}$ が dead-end である場合は $\hat{M}$ の dead-end マスクを，そうでない場合は空集合を返却する．探索は関数呼び出し $\textsc{Search}(\emptyset, C)$ によって開始される．アルゴリズム 5 と比較したときアルゴリズム 6 の差分は，dead-end マスクの抽出 (8 行目など)，dead-end パターンとの照合 (14 行目)，記録 (20 行目) の 3 つを行う点である．1 つ目の dead-end マスクの抽出は探索失敗の確認とセットで行う．補題 1 の候補頂点が存在しない場合 (7, 8 行目)，補題 2 の単射制約を満たさない場合 (12, 13 行目)，補題 3 の dead-end パターンに符合する場合 (14, 15 行目)，そして 3.4.3 節で述べた再帰呼び出しが探索失敗となる場合 (17 行目) である．候補頂点が存在しない場合を除き，直接得られるのは新しい部分埋め込み $M \cup \{(u_{k+1}, v)\}$ の dead-end マスクである．そこでそれらは Γ_* に蓄積し，候補頂点についてのループを終えた後で式 3.5 により $\hat{M}$ の dead-end マスクに変換される (18 行目)．2 つ目の dead-end パターンとの照合は，連想配列の 1 要素 $\Delta[u_{k+1}, v]$ に対してのみ行われるため高速である．なおもし dead-end パターンの記録がまだ行われておらず $\Delta[u_{k+1}, v]$ が未定義であった場合，$\Delta[u_{k+1}, v] \subseteq \hat{M}$ は偽になるものとして扱う．3 つ目の dead-end パターンの記録は，再帰呼び出し中に埋め込みの報告がない，即ち $\hat{M}$ が dead-end であり，かつ $\hat{M}$ が頂点割り当てを含むときにのみ行う (19 行目)．後者の条件は $\hat{M} = \emptyset$ のとき $\hat{M}[u_k]$ が未定義となってしまうため確認している．最後に，$\hat{M}$ が dead-end であるときは dead-end マスクを (21 行目)，そうでな

アルゴリズム 6 探索アルゴリズムの詳細

入力: クエリグラフ Q，データグラフ G.

出力: G に含まれる全ての Q の埋め込みを報告する.

```
1:  function SEARCH(M̂, C)
2:      k ← |M̂|
3:      if k = |V_Q| then
4:          M̂ を埋め込みとして報告する
5:          return ∅
6:      C′ ← C のうち M̂ についてエッジ制約を満たす候補頂点 (式 3.2)
7:      if C′[u_i] = ∅ であるクエリ頂点 u_i が存在する then
8:          Γ ← N(u_i) ∩ dom(M̂)                                  ▷ 補題 1
9:      else
10:         Γ_* ← ∅
11:         for each v ∈ C′[u_{k+1}] do
12:             M̂_+ ← M̂ ∪ {(u_{k+1}, v)}
13:             if v ∈ dom(M̂) then
14:                 Γ_* ← Γ_* ∪ { u_i ∈ dom(M̂_+) | M̂_+[u_i] = v }   ▷ 補題 2
15:             else if Δ[u_{k+1}, v] ⊆ M̂_+ then
16:                 Γ_* ← Γ_* ∪ Δ[u_{k+1}, v]                     ▷ 補題 3
17:             else
18:                 Γ_* ← Γ_* ∪ SEARCH(M̂_+, C)
19:         Γ ← 式 3.5 により変換された Γ_*                       ▷ 補題 4
20:     if 再帰呼び出し中に埋め込みの報告がなく，かつ M̂ ≠ ∅ then
21:         Δ[u_k, M̂[u_k]] ← { (u_i, v) ∈ M̂ | u_i ∈ Γ }
22:         return Γ
23:     return ∅
```

ければ空集合を返却する (22 行目)．以上のようにして実用的な時間計算量と空間計算量の下で dead-end パターンによる枝刈りを実装することが可能である．

3.4.6 定性的議論

本節では提案手法の定性的性質として，全ての埋め込みを列挙すること (完全性) を証明し，さらに dead-end パターンで枝刈り可能な探索について考察する．

完全性

提案手法は dead-end パターンによって探索を打ち切る．打ち切られる探索は必ず完全な埋め込みを発見しない探索であるため，列挙される埋め込みに影響はない．具体的には，アルゴリズム 6 に示した提案手法は次の性質を満たす．

定理 1 (完全性)**.** アルゴリズム 6 はデータグラフ G に含まれるクエリグラフ Q の全ての埋め込みを報告する．

証明. ナイーブなバックトラッキング (アルゴリズム 5) からの探索範囲の変化は 14 行目における dead-end パターンとの照合によって発生する．補題 1, 2, 3, 4 より $\Delta[u_{k+1}, v]$ は (値が代入されているならば) dead-end であるから，$\Delta[u_{k+1}, v] \subseteq \hat{M}$ である部分埋め込み $\hat{M}$ も dead-end である．定義 6 より dead-end である部分埋め込みは完全な埋め込みを生じないので，探索を打ち切っても報告される埋め込みは変化しない． □

Dead-end パターンにより枝刈り可能な範囲

3.4.3 節ではシンプルなバックトラッキング (アルゴリズム 5) のコードに基づき探索失敗になる状況を分類し，それぞれに対して dead-end パターンの抽出方法を示した．ここでは具体的にどのような状況が dead-end パターンによって枝刈りされるかを考察する．

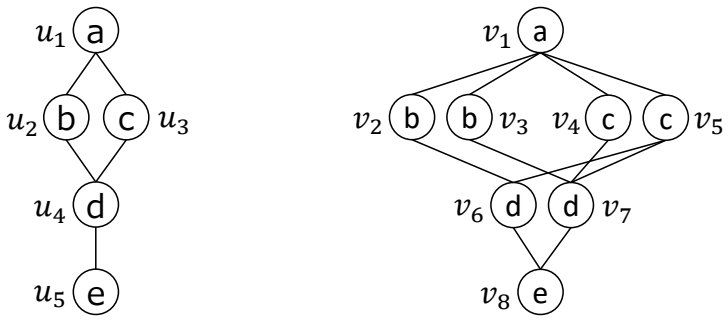
■候補頂点が存在しない場合　候補頂点が存在しない状態とはアルゴリズム 5 の 7 行目で $C'[u_{k+1}] = \emptyset$ となった場合を意味する．候補頂点の絞り込みは 6 行目においてエッジ制約に基づいて行われている．それによって候補頂点集合が空になる具体的な状況を図 3.3 のクエリグラフ Q とデータグラフ G を例に考える．

探索中に部分埋め込み $\hat{M} = \{ (u_1, v_1), (u_2, v_2), (u_3, v_4) \}$ が発生したとする．次に割り当てを行う u_4 は u_2 と u_3 に隣接していることから，候補頂点は v_2 と v_4 に共通して隣接するラベル d の頂点となる．しかしそのようなデータ頂点は存在しないため，候補頂点集合は空となる．このときの dead-end マスク Γ は補題 1 より，

$$\Gamma = N(u_4) \cap \mathrm{dom}(\hat{M}) \tag{3.16}$$
$$= \{ u_2, u_3, u_5 \} \cup \{ u_1, u_2, u_3 \} \tag{3.17}$$
$$= \{ u_2, u_3 \} \tag{3.18}$$

である．従って抽出される dead-end パターンは $\{ (u_2, v_2), (u_3, v_4) \}$ となる．このパターンが意味するところは「u_2 と u_3 をそれぞれ v_2 と v_4 に割り当てると探索が失敗する」であり，データグラフ G の構造を踏まえると正しいことが分かる．クエリグラフ Q のように閉路を含む場合，候補頂点が複数のデータ頂点に共通する隣接頂点に絞り込まれるため候補数が減少しやすい．この性質に着目し，前処理としてクエリグラフ中の閉路部分から先に割り当てを行うマッチング順序最適化手法が提案されている [7]．一方でデータグラフに含まれる閉路を発見することは極めて高コストであるため，これまで閉路に基づく効果的な枝刈りは行われてこなかった．それに対し，提案手法は dead-end パターンによってデータグラフ上の閉路を捉えた枝刈りを行うことで高速な探索を可能にする．

■単射制約を満たさない場合　1 つのデータ頂点に複数のクエリ頂点を割り当てると単射制約違反となる．アルゴリズム 5 では 8 行目の条件で同じデータ頂点を重複して使用していないか確認している．この条件は容易に確認できるため，一見 dead-end パターンによる枝刈りは不要であるように見える．しかしながら，単射制約はクエリグラフに含まれるパスのマッチングにおいて大きな性能問題となる．パスの探索で単射制約違反が発生するのはデータグラフ中の閉路に遭遇した場合である．具体例として 3.1 節では図 3.1 のクエリグラフ Q とデータグラフ G を用いて v_1 の重複使用によって探索が失敗する状況を示した．この例においてはラベル a–b–c–a のパス状クエリをラベル a, b, c の 3 頂点から成るデータグラフ上の閉路に割り当てようとして探索失敗が生じている．実世界のグラフは



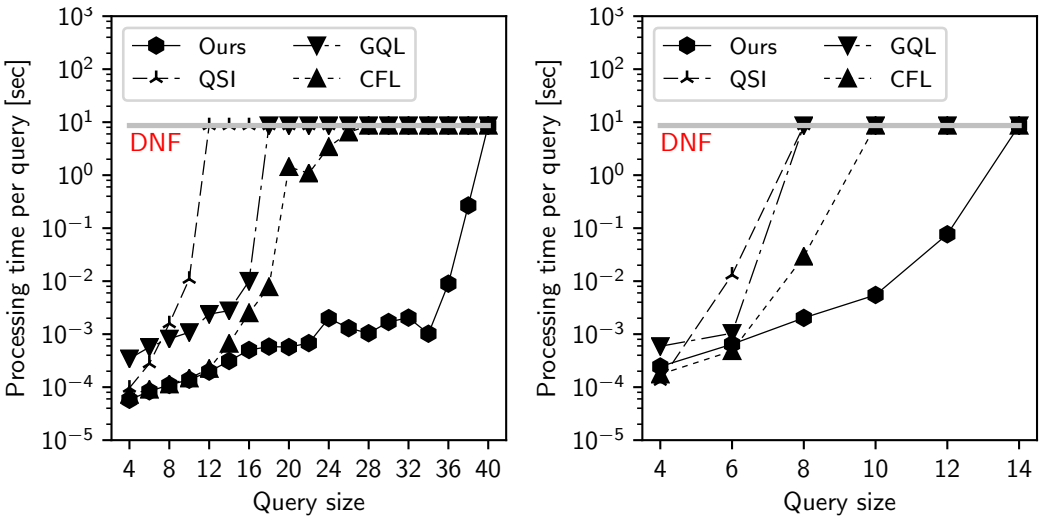
(a) クエリグラフ Q (b) データグラフ G

図 3.3: 候補頂点が空になるクエリグラフとデータグラフの例.

複雑な接続関係を有し，大量の閉路が存在する．そのため，パス状クエリの探索において閉路で単射制約違反を生じる場面は多いと考えられる．前述したようにデータグラフ上の閉路を前処理で発見するためには高い計算コストを要するため，既存手法において閉路への割り当てを回避するような枝刈りは行われてこなかった．それに対し，dead-end パターンは単射制約違反を回避する効果的な枝刈りが可能である．

3.5 評価

本節では dead-end パターンによる枝刈りの効果を評価する．提案手法の実装ではバックトラッキングによる探索に提案手法を用い，候補頂点の枝刈りとマッチング順序最適化に最新手法である CFL-Match [7] を組み合わせる．比較対象の手法としては CFL-Match に加え，文献 [59] において高い性能が確認されている QuickSI [83] と GraphQL [43] の合計 3 つを用いる．QuickSI と GraphQL の実装は文献 [59] の著者から，その他は論文の著者から入手した．実験に用いた計算機は Intel Xeon E5-2697 v2 を搭載し，256GB のメモリを持つ．グラフは既存研究 [7, 42, 43, 81, 83, 97] で広く利用されている yeast と human を用いる．共にタンパク質相互作用ネットワークであり，yeast は 3,112 頂点，12,519 エッジ，71 種類の頂点ラベルを，human は 4,674 頂点，86,282 エッジ，44 種類の頂



(a) yeast データセット　　(b) human データセット

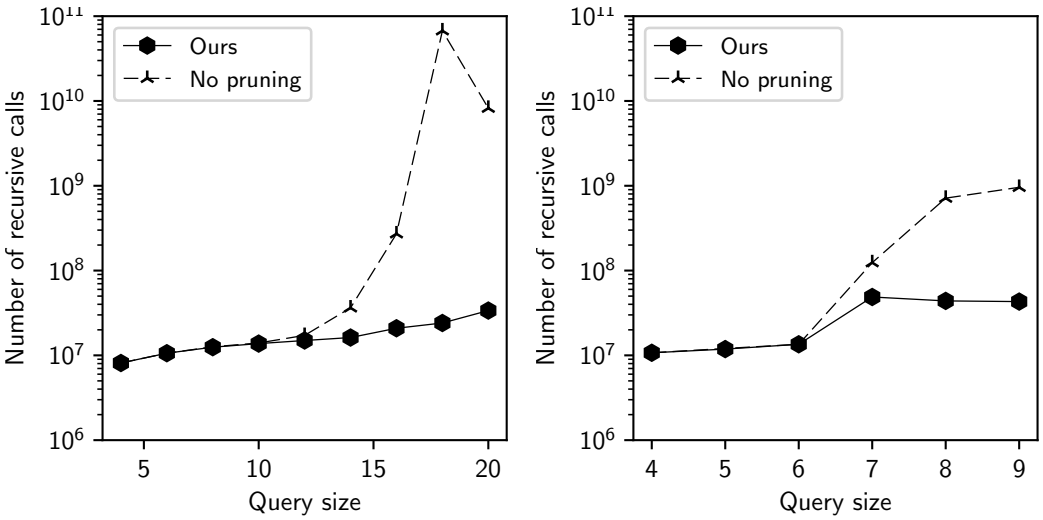
図 3.4: 各手法の 1 クエリあたりの平均処理時間の比較

点ラベルをそれぞれ含む．クエリグラフはランダムウォークによってデータグラフからサブグラフを抽出することで生成する．クエリは頂点数を変化させながら生成し，頂点数ごとに 1 万クエリを 1 つのクエリセットとする．クエリセットの処理に 1 日以上を要する場合は処理不能 (Do-Not-Finish, DNF) とみなす．クエリによっては膨大な数の埋め込みが存在し全ての列挙が現実的でない場合があるため，文献 [42] と同様 1,000 個の埋め込みを発見した時点で探索を打ち切る．

3.5.1　クエリ処理時間

まず dead-end パターンによる枝刈りによって得られる性能を確認するため，既存手法とクエリの処理時間を比較する．実験結果を図 3.4 に示す．図中の Ours, QSI, GQL, CFL はそれぞれ提案手法，QuickSI, GraphQL, CFL-Match を意味する．yeast データセットにおいては 40 頂点，human データセットにおいては 14 頂点で全ての手法が DNF となった．

全体傾向を見ると，提案手法は小規模なクエリから大規模なクエリまでバランスよく高速である．その中でも特に大規模なクエリにおいて性能差が顕著で



(a) yeast データセット (b) human データセット

図 3.5: 各クエリサイズにおいて 10,000 クエリの探索で発生した再帰呼び出し回数の合計

ある．特に yeast の 26–36 頂点のクエリでは既存手法比でおよそ 4 桁の高速化がみられる．図 3.1 で例示したように，既存手法は候補頂点枝刈りなどバックトラッキングの前処理が有効でないクエリグラフにおいて網羅的な割り当ての試行に陥ることがある．クエリが大規模化すると候補頂点の組み合わせ数が増大するため，処理時間が爆発的に増加する傾向にある．これに対し提案手法はクエリグラフの構造によらず dead-end パターンに基づく枝刈りによって無駄な探索を削減する．そのため大規模なクエリにおいても処理時間の増加を抑えることができている．

3.5.2 Dead-end パターンによる枝刈りの回数

次に提案手法の性能が何によってもたらされているかを理解するため，Search 関数 (アルゴリズム 6) の再帰呼び出し回数に着目して評価を行う．本実験では dead-end パターンによる枝刈りを行う通常の提案手法 (“Ours”) と，アルゴリズム 6 の 14, 15 行目をコメントアウトすることで枝刈りを省略した手

法 (“No pruning”) について，再帰呼び出しが行われた回数を比較する．これらの再帰回数の差が枝刈りの効果である．結果を図 3.5 に示す．yeast では 20 頂点，human では 9 頂点より大きいクエリに対して No pruning が DNF となったため，それ以降の頂点数はプロットしていない．再帰回数を見ると，yeast, human とも小規模なクエリでは枝刈りがほぼ発生していないものの，クエリが大規模化するに従って枝刈りの効果が増大している．yeast の 18 頂点のクエリにおける再帰回数は Ours が約 2.4×10^7 回，No pruning が約 6.7×10^{10} 回であり，3 桁以上の再帰を削減している．大規模なクエリほど削減率が大きいのは探索失敗の発生頻度が高いためと考えられる．クエリの頂点数が多いほど同一データ頂点への複数のクエリ頂点の割り当て (単射制約違反) が発生しやすい．またエッジ数が増加するためエッジ制約違反も発生しやすい．それらによる探索失敗を削減することで提案手法は大幅に性能を向上させている．対照的に小規模なクエリでは探索失敗の削減率が小さい．それでも図 3.4 の実験結果において既存手法と比肩する性能を示すことができているのは，提案手法が 3.4.4 節で述べたように dead-end パターンの効率的な管理と照合を行い，枝刈りに要するオーバーヘッドを極めて小さく抑えているためである．

3.5.3　議論

ここでは yeast と human を用いた本章の実験から読み取れる提案手法の性質について議論する．

サブグラフマッチングにおいては一般に頂点ラベルの種類が少なく，クエリグラフのエッジ密度が高いほど複雑な問題となる．なぜならばラベルの種類が少ないほどラベルに基づくフィルタリングで除外可能な候補頂点が減少し，またエッジが多いほどエッジ制約による探索失敗が多発するためである．yeast と human は頂点数が近い一方で，yeast と比べ human はラベルが少なくエッジ密度が高い．またデータグラフから抽出したサブグラフをクエリとしているため，human からはエッジ密度の高いクエリが生成されやすい．従ってラベル数とエッジ密度の観点で human は yeast よりも複雑さの大きい問題設定となる．図 3.4 に示さ

れているように，クエリサイズのスケールは異なるものの，yeast と human の処理時間はどちらも似た形状のプロットとなっている．具体的には，性能は提案手法，CFL-Match, GraphQL, QuickSI の順で高く，提案手法は既存手法が DNF となるクエリサイズのおよそ 2 倍のサイズまで処理できている．yeast と human という 2 つのグラフにおいて同じ傾向を示した本実験結果は，ラベル数とエッジ密度に依存せず提案手法が既存手法よりも効率的であることを示している．

データグラフの規模もまたサブグラフマッチングの複雑さに影響する．実験では最新手法である CFL-Match 上に提案手法によるバックトラッキングを実装しているため，データグラフに対する性能の変化は CFL-Match の評価結果を基に議論することができる．人工的に合成したグラフを用いた実験 (文献 [7] 6.2 節) において CFL-Match のクエリ処理時間はデータグラフの頂点数とエッジ数に対しておおよそ線形に増加している．これは CFL-Match がデータグラフとクエリグラフを前処理した結果生成される索引構造 (Compact Path Index [7]) がデータグラフの規模に対して線形のサイズを示すためである．前処理後のバックトラッキングでは探索失敗を繰り返すことで索引サイズに対して線形以上に処理時間が増加する．しかし提案手法を用いることで探索失敗を CFL-Match よりも減らすことができる．従って処理時間に占める索引サイズの影響はより顕著になるため，CFL-Match を基にする限り提案手法もまたデータグラフの規模に対して線形に処理時間が増加すると推測される．また CFL-Match の評価では 100 万頂点の合成グラフが使用されており，提案手法もその程度の規模までは現実的な時間で処理可能である可能性が高い．

3.6 結論

サブグラフマッチングは様々な場面で利用されるが，NP 困難に属し高速化が課題となっている．そこで本章では，失敗から学び，失敗を繰り返さないことにより高速化するサブグラフマッチングアルゴリズムを提案した．具体的にはバックトラッキングによる探索中に発生した探索失敗の原因に基づき dead-end

パターンを生成し，パターンに符合する部分埋め込みの探索を打ち切る．実験の結果提案手法は既存手法比で最大 4 桁高速であることが確認された．

本研究では第 2 章のリオーダリング手法で命令処理効率を向上させ，本章の枝刈りによって発行命令数を削減するという役割分担をとっている．本章ではメモリアクセスについて言及していないが，サブグラフマッチング処理を開始する前に第 2 章の手法でグラフをリオーダリングすることにより局所性の向上を期待できる．このように 2 つの技術の組み合わせによってクラウド上に高速なクエリ処理エンジンを実現することができる．クラウド上でパーソナルデータを使用するための匿名化手法については次の第 4 章で説明する．

提案手法の評価では yeast と human という 2 つのデータグラフを用い，クエリグラフの頂点数を変化させつつ実験を行った．実験ではラベル数およびクエリグラフのエッジ密度と頂点数に関する提案手法の傾向を確認することができた．しかしながら，データグラフの規模による性能の変化や，パス，クリークなどクエリグラフの形状別の性能評価など，よりよく提案手法の性質を理解するために必要な実験の追加は今後の課題である．実験では提案手法に基づくバックトラッキングと CFL-Match [7] を組み合わせた実装を使用しているため，特に CFL-Match の評価に使用されている実験設定においてどのような差分が生じるか確認することが重要である．

また，本章の実験では既存手法と同様に提案手法もクエリサイズが大きくなるにつれて処理時間が増大することが確認された (図 3.4)．その原因としては次の 2 つの可能性が考えられる．1 つ目の可能性は dead-end パターンの上書きによる枝刈り効果の低下である．3.4.4 節で説明したように，dead-end パターンはハッシュテーブルに格納される．キーが衝突した場合は上書きによって古いパターンが削除される．このとき枝刈り効果の大きいパターンが削除されると枝刈りの効果が大幅に低下してしまう．2 つ目の可能性は「失敗から学ぶ」という提案手法のアイデア自体の限界である．実験においてはクエリグラフの頂点数を固定したが，エッジ数はデータグラフからランダムに抽出された部分によって大きく異なる．エッジ数が多いほど同型であるための制約は厳しくなるため，制約のチェックに要する時間が増加する．このように，探索失敗を繰り返さないだけで

は解消することが難しい計算コストが性能に影響している可能性がある．これらの可能性は，どちらも追加の実験を行うことで検証することができる．1 つ目の可能性については保存する dead-end パターンの数を変更しながら性能を比較することで影響を確認できる．また 2 つ目の可能性についても，頂点数に対してエッジ数が最大となるグラフ，即ちクリークをクエリとした場合の性能を既存手法と比較することで提案手法の効果を確認できる．これらの確認が今後の課題となる．

第 4 章

最近傍グラフを用いた k-匿名化

4.1 序論

個人情報を含むデータは統計や医療のために様々な組織で収集されている．収集されたデータは情報源として利用価値が高い反面，個人のプライバシを侵害する危険を孕んでいる．そこで，データ内の情報と個人を結び付けられない形へのデータの匿名化が行われる．匿名化はクラウド環境にデータを保存する際の安全性を高めるために用いることができる [82]．本研究においても，1.2 節で説明した枠組みに基づきクラウド上には匿名化したグラフデータを保存し，クエリ処理に利用する．

匿名化の文脈においては表形式のデータが広く扱われてきた．例として架空の Web ページの閲覧履歴を表 4.1 に示す．この表から閲覧したページを知られないよう匿名化するためには，名前や個人番号 (マイナンバーや米国の社会保障番号) のような明確に個人を識別し得る属性を取り除くのみでは不十分である．なぜなら，年齢，性別，居住地といった単体では個人を特定できないような属性でも，複数の属性を組み合わせることで個人を特定できてしまう場合があるからである． このような他者でも比較的情報を入手しやすく且つ複数組み合わせることで個人を特定できてしまうような属性群を**準識別子** (quasi-identifier) [23] と

本章は日本データベース学会論文誌 (©2014 The Database Society of Japan) に含まれる ‘クラスタリングと空間分割の併用による効率的な k-匿名化’ [99] に基づく．

表 4.1: ページ閲覧データ

名前	年齢	性別	居住地	閲覧ページ
阪大 太郎	22	男	大阪府吹田市	https://www.osaka-u.ac.jp/
佐藤 一子	41	女	大阪府大阪市	https://sato.com/
鈴木 二子	30	女	大阪府大阪市	https://suzuki.edu/
高橋 次郎	22	男	京都府京都市	https://takahashi.gov/
田中 三郎	8	男	京都府京都市	https://tanaka.info/
伊藤 三子	59	女	神奈川県横浜市	https://ito.net/
渡辺 四子	65	女	東京都千代田区	https://watanabe.org/

呼ぶ．

準識別子から個人を特定できないようにする手法としては **k-匿名化** [88] が代表的である．k-匿名化とは，データを k-匿名性のある状態，即ち k 人以上を準識別子から区別できないようなデータへ変換することである．$k = 2$ として表 4.1 を k-匿名化した結果の例を表 4.2 と表 4.3 に示す．表 4.2 は数値を平均値へ，カテゴリ値を最頻値へ置き換える**ミクロアグリゲーション** [21, 27, 94] によって k-匿名化されている．一方，表 4.3 は年齢を数値の範囲へ置き換え，性別や居住地のようなカテゴリ値をより大きな区分へ置き換える**一般化** [87] によって k-匿名化されている．表 4.2 と 4.3 のどちらも 2 人以上が同じ準識別子を持つため，準識別子を基にした閲覧ページの特定が難しくなっている．ただし，一般化による匿名化ではデータを扱うプログラムが数値の範囲などの一般化された表現を扱えるよう変更する必要がある．一方でミクロアグリゲーションによる匿名化では具体的な値が保たれるため，第 2 章と第 3 章の手法を用いたクエリ処理エンジンをそのまま利用可能である．そこで，本章ではミクロアグリゲーションによる k-匿名化に焦点を当てる．

本研究では図 1.1 に示すようなグラフデータを対象としているが，このようなデータも個人に関する属性という観点で匿名化することができる．例えば表 4.1 に含まれる阪大太郎氏の情報は図 1.1 の情報を表形式に変換したものである．年齢と性別は人物に直接付与されたプロパティである．さらに LiveIn エッジで接

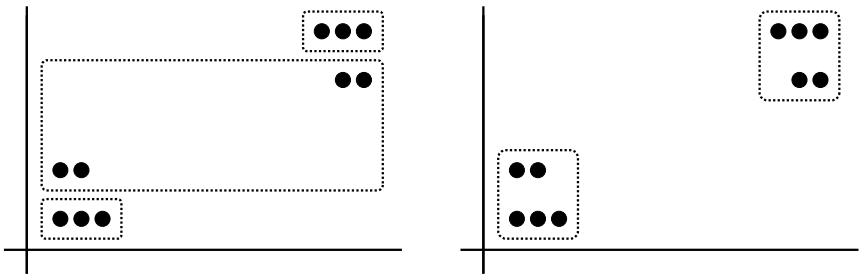
表 4.2: ミクロアグリゲーションにより k-匿名化されたページ閲覧データ

名前	年齢	性別	居住地	閲覧ページ
—	31	女	大阪府大阪市	https://www.osaka-u.ac.jp/
—	31	女	大阪府大阪市	https://sato.com/
—	31	女	大阪府大阪市	https://suzuki.edu/
—	15	男	京都府京都市	https://takahashi.gov/
—	15	男	京都府京都市	https://tanaka.info/
—	62	女	神奈川県横浜市	https://ito.net/
—	62	女	神奈川県横浜市	https://watanabe.org/

表 4.3: 一般化により k-匿名化されたページ閲覧データ

名前	年齢	性別	居住地	閲覧ページ
—	[22–41]	人	大阪府	https://www.osaka-u.ac.jp/
—	[22–41]	人	大阪府	https://sato.com/
—	[22–41]	人	大阪府	https://suzuki.edu/
—	[8–22]	男	京都府京都市	https://takahashi.gov/
—	[8–22]	男	京都府京都市	https://tanaka.info/
—	[59–65]	女	関東	https://ito.net/
—	[59–65]	女	関東	https://watanabe.org/

続された住所 (居住地) や，Browsed エッジで接続された URL (閲覧ページ) なども個人に関する情報とみなすことができる．匿名化による書き換えはプロパティを直接変更するか，LiveIn, Browsed などのエッジの接続先を変更することで元のグラフに反映できる．例えば表 4.2 では匿名化によって阪大太郎氏の居住地が大阪府吹田市から大阪府大阪市に変更されている．この場合は大阪府大阪市を表現する頂点を (無ければ) 作成し，LiveIn エッジの接続先を大阪府大阪市に変更すればよい．このようにグラフと表は相互に変換可能である．k-匿名化の研究においては表データを対象とすることが多いため，本章でも表データに基づいて議論を行う．



(a) 分散の大きいグループ　　(b) 理想的なグループ

図 4.1: 既存手法が分散の大きいグループを作る例 ($k = 3$)

以上のようにグラフや表から個人を特定できないように匿名化することができる．しかしながら，匿名化処理ではデータの変換によって情報が失われる．例えば表 4.1 から表 4.2 への匿名化によって，阪大太郎氏の年齢，性別，居住地は全て書き換えられてしまっている．情報に損失の大きいデータに対するクエリは元のデータと全く異なる結果を返却するため，有用性が低下する．情報損失を小さくするためには互いにデータの似通ったレコードを集めてグループを作り，同じ準識別子に置き換える必要がある．k-匿名化では k 人以上に同じ準識別子を与えるため，各グループは k レコード以上を含む分割になっていなければならない．このように分割された状態はしばしば **k-分割**と呼ばれている [26, 47, 86]．つまり，k-匿名化のためには情報損失の小さい k-分割を作成する必要がある．

情報損失の小さい k-分割を作成するため，これまで多くの手法が研究されてきた．それらはクラスタリングに基づく手法 [12, 26, 47, 58, 67, 86] と kd-tree [33] 構築アルゴリズム等の空間分割に基づく手法 [48, 61] に分けられる．レコード数 n に対する計算量は前者が $O\left(n^2\right)$，後者が $O\left(n \log n\right)$ である．つまり空間分割に基づく手法のほうが計算量は小さい．診療記録や購買履歴のように複数のレコードが 1 人に対応するデータは人口を超えて際限なく増大し得るため，レコード数に対する計算量の小ささは重要である．一方，クラスタリングのほうが空間分割よりも柔軟に分割を行うことが可能であるため情報損失は小さい [12]．このように，既存の手法では計算量と情報損失の小ささを両立できていない．

さらに，クラスタリングに基づく既存手法であってもレコードのクラスタ構

造を適切に捉えられない場合がある．例として図 4.1 のように 2 次元空間上に分布したデータの k-分割を考える．図中の点はそれぞれ 2 次元情報のレコードを表す．図 4.1(b) に示されているように，このデータには左下と右上に 2 つのクラスタが存在する．ところがこのデータに対して既存手法 [12,47] を適用すると，作成されるグループの数が $\lfloor n/k \rfloor$(n はレコード数) に固定されているため図 4.1(a) のようにクラスタを無視したグループが作成されてしまう．グループ数が可変である手法 [86] も同様にクラスタを正確に捉えられず図 4.1(a) のようなグループを作ってしまう．

そこで本研究では 2 つの手法を提案する．それらは (i) 新しい k-分割アルゴリズムの**友引法**，並びに (ii) 空間分割とクラスタリングの併用である．

1 つ目の提案である友引法は，ミクロアグリゲーションによる k-匿名化のための k-分割クラスタリングアルゴリズムである．友引法の特徴はグラフを用いて既存手法より小さい情報損失を実現する点にある．グラフは k 近傍グラフのようにレコードを頂点とし近傍点を接続して構築する．友引法は近傍のレコードを貪欲に収集し分散の小さいグループを形成しつつ，グラフから発見されるクラスタを考慮して総グループ数を変化させる．これによってより全体最適に近い分割が可能になる．例えば図 4.1(b) のような分割を作成する．

2 つ目の提案である空間分割とクラスタリングの併用では，2 段階の処理を行うことで情報損失を抑えつつ高速な分割を可能にする．具体的には，まずレコード数 n に対し計算量 $O(n \log n)$ の空間分割アルゴリズムによって大まかな分割を行い，次にクラスタリングに基づく計算量 $O\left(n^2\right)$ のアルゴリズムでさらに細かく分割する．レコード全体の分布を分析する一般的なクラスタリングと異なり，k-分割では近傍のレコードでグループを作成できさえすればよい．そのためには局所的なレコードの分布が分かっていれば十分である．従って，クラスタリングの対象が空間分割によって作られた狭い空間内のレコードに限定されても情報損失への影響は小さい．計算量の小さい空間分割を途中まで使用することで k-分割処理全体の計算量は $O\left(n^2\right)$ より小さくなるため，空間分割の併用によって小さい情報損失と高速な処理を両立できる．本章では友引法と空間分割の併用に焦点を当てるが，その他のクラスタリングに基づく分割手法と空間分割を組み

合わせることも可能である．

提案手法の特長は次の 3 つにまとめられる：

1. **低情報損失**：友引法は最近傍グラフの構築を通じてレコードのクラスタ構造を発見する．これにより情報損失の少ない k-分割を作成することができる．
2. **高速**：友引法と高速な空間分割を併用することによって大量のデータを高速に匿名化することができる．
3. **パラメータによる制御**：空間分割を併用する際には分割の粒度 $k_\sharp$ をパラメータとして与える．この値によって情報損失の大きさと処理性能のどちらを優先するか利用者が調整可能である．

友引法は既存手法と比べ最大 16% 情報損失を減少させることが実験によって分かった．空間分割を併用するとクラスタリングのみ用いた場合と比べ情報損失が増大するが，友引法を使用することでこの損失を補うことができる．実験では，空間分割と友引法の併用はクラスタリングベースの既存手法のみを用いた場合と同等の情報損失量を約 10 倍高速に実現した．

本章の構成は次の通りである．4.2 節で k-匿名化に関する研究を紹介し，4.3 節で前提知識の説明を行う．4.4 節においてクラスタリングに基づく k-分割アルゴリズムである友引法，および空間分割とクラスタリングを併用するアイデアの詳細を説明し，4.5 節で評価を行う．最後に 4.6 節で結論を述べる．

4.2　関連研究

k-匿名化は (i) k-分割の作成と (ii) k レコードに同じ準識別子を与えるための準識別子の変換という 2 段階に分けられる．k-分割手法をクラスタリングと空間分割に，準識別子の変換手法をミクロアグリゲーションと一般化に分類し，これまでの研究と本研究を分類したものが表 4.4 である．それぞれの分類について詳しく説明した後，本研究の位置づけを述べる．

表 4.4: 既存研究と本研究の分類

		準識別子の変換手法	
		ミクロアグリゲーション	一般化
分割手法	クラスタリング	[26, 47, 58, 86], 友引法	[12, 44, 67]
	空間分割	—	[48, 61]

4.2.1 分割手法

クラスタリングに基づく分割手法

クラスタリングに基づく手法はレコード間の距離を用いて分割を行う．一般的なクラスタリングアルゴリズムでは 1 つのグループに k レコード以上含まれることを保証できないため，k-分割専用のアルゴリズムが研究されてきた [12, 26, 47, 58, 67, 86]．

クラスタリングに基づく手法はグループ数が固定であるものと可変であるものに分けられる．前者においては Maximum Distance to Average Vector (MDAV) [27, 47] が，後者においては MDAV を基にした Variable-size Maximum Distance to Average Vector (V-MDAV) [86] が代表的である．MDAV はグループの中心とするレコードからユークリッド距離が最も近い $k-1$ レコードを収集して 1 つのグループにする操作を繰り返す．グループのレコード数が k に固定されているため，クラスタを無視して分散の大きいグループを作ってしまう場合がある．このような結果になる原因の一つは，グループの作成によって周囲から孤立したレコードが取り残されてしまうことにある．例えば図 4.1(a) の中央のグループは，左下と右上のグループから取り残された 4 つのレコードで作られている．そこで V-MDAV は MDAV で k レコードのグループを作成した後，グループの周囲に孤立したレコードが存在する場合はそれらもグループに追加することでまとまりのよいグループを作成する．しかしながら，V-MDAV で孤立レコードと判定されるのは 1 つのレコードが単独で存在している場合のみであるため，2 個以上 k 個未満のレコードがまとまって孤立している場合には無視されて

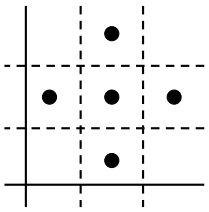


図 4.2: 空間分割に基づく手法で分割できない状態の例 ($k = 2$)

しまう．従って図 4.1 の例では MDAV と同様に図 4.1(a) のような分散の大きいグループが作られる．さらに V-MDAV は孤立状態を判定するためのパラメータ γ を要求し，適切な γ はデータによって異なる [86] ため設定が難しい．以上のように，既存手法には解決されていない問題がある．

空間分割に基づく分割手法

空間分割に基づく手法 [48, 61] ではレコードを多次元空間上の点と見做し，k レコード以上含む空間を作れなくなるまで空間の分割を繰り返す．ここで言う空間分割とは，例えばレコード全体を年齢が 30 歳以下の集合と 30 歳より大きい集合に分割するような操作を指す．分割には kd-tree [33] や R-tree [41] のような空間インデックスの構築アルゴリズムが使用される．

これらの手法は，レコード数に対する計算量がクラスタリングに基づく手法よりも小さく高速である一方で，情報損失が大きい [12]．図 4.2 がその理由の一端を示している．図中には 5 つのレコード (点) が存在するため，$k = 2$ において 2 グループへ分割できるはずである．しかし実際には図中に示したいずれの点線で分割してもレコード数 1 のグループが生じるため，分割できない．結果として 5 レコード全体で 1 つのグループとなってしまい，情報損失が増加する．

4.2.2　準識別子の変換手法

ミクロアグリゲーションによる k-匿名化

ミクロアグリゲーションとは同じグループに属する k 人以上のレコードの値を平均値に書き換えることにより個人の特定を防ぐ手法で，統計的開示抑制技術

として 1980 年代から存在している [21, 94]．元々ミクロアグリゲーションによる値の書き換えはレコードの全属性に対し行われていたが，書き換えを準識別子に限定することによって k-匿名化にも利用される [27]． 表 4.1 をミクロアグリゲーションによって 3-匿名化したものが表 4.2 である．

ミクロアグリゲーションは一般化に対して 2 つの利点がある．1 つ目の利点は匿名化前のデータと匿名化後のデータに対して同じデータ分析手法を適用できることである．表 4.3 に示されているように，一般化では数値データが数値の範囲と置き換えられる．つまり，データの種類が変化する．これに対し，ミクロアグリゲーションではデータの種類が匿名化前後で維持される．例えば数値は同じ数値データである平均値と置き換えられる．従って元データと同じアルゴリズムで分析が可能である．2 つ目の利点は数値データの情報がより多く残ることである．一般化によって数値の範囲に置き換えられた場合，元データの数値の分布情報は残らない．例えば 5 つの整数の集合 $\{0, 0, 0, 0, 10\}$ と $\{0, 10, 10, 10, 10\}$ はどちらも [0–10] へと一般化されるため区別できない．一方，ミクロアグリゲーションではそれぞれ平均値の 2 と 8 に置き換えられるため，集合に含まれる値の大小関係が残る．これにより匿名化後のデータを用いた分析で有用な情報を抽出できる可能性が高まる．

一般化による k-匿名化

表 4.3 のように，数値を値の範囲へ，カテゴリをより大きな概念へ置き換える変換手法は一般化 [87] と呼ばれる．

一般化による k-匿名化はさらに**大域的再符号化** (global recoding) と**局所的再符号化** (local recoding) に分けられる [93]．大域的再符号化ではレコードが持つ属性値を一般化された値と対応付ける全レコード共通の関数が存在する．これに対し，局所的再符号化ではそのような関数を定義できない．例えば表 4.1 から表 4.3 への変換において，年齢の 22 という値は [22–41] と [8–22] の両方に一般化されている．このことから使用された一般化関数が全レコード共通でないことが分かる．局所的再符号化は大域的再符号化よりも柔軟な一般化が可能であるため情報損失は減少する．一方で，一般的なデータ分析手法では局所的再符号化が

行われたデータを正しく処理できないことが指摘されている [34]．例えば決定木の作成において分類条件を定義するためには再符号化の意味を解釈する必要がある．具体的には，「神奈川県横浜市」が「関東」に含まれる地名であることや，重なりや包含関係を持つ数値の範囲を考慮しなければならない．このように大域的再符号化と局所的再符号化にはそれぞれ利点と欠点がある．

4.2.3　本研究の位置付け

1 つ目の提案手法である友引法は分割手法としてクラスタリングを用い，準識別子の変換手法としてミクロアグリゲーションを前提とした k-分割手法の一つである (表 4.4)．友引法の主な特長はレコードのクラスタ構造を捉えることで既存手法よりも小さい情報損失を実現することである．2 つ目の提案手法である空間分割とクラスタリングの併用は，2 つの異なるアイデアに基づく分割アルゴリズムを組み合わせることで低情報損失かつ高速な k-分割を可能にする処理の枠組みである．このような組み合わせは我々が知る限りこれまで試みられていない．

4.3　事前準備

4.3.1　空間分割に基づく k-分割

提案手法では空間分割に基づく k-分割手法として Mondrian [61] を使用するため，その概要を述べる．Mondrian の疑似コードをアルゴリズム 7 に示す．Mondrian 関数はレコードの集合 R を引数として受け取り，k-分割されたグループの集合，即ちレコードの集合の集合を返却する．k-分割処理は全レコードの集合を引数として Mondrian 関数を呼び出すことで開始される．Mondrian 関数はまず R を分割可能であるかどうかを判定する (2 行目)．前述した図 4.2 のように，k レコード以上のグループに 2 分割可能な次元が存在しなければ分割できない．もし R が分割不可能であれば，そこに含まれる全てのレコードを 1 つのグループとして返却する (3 行目)．そうでなければ R をさらに

アルゴリズム **7** Mondrian

1: **function** MONDRIAN(R)
2: **if** R は分割不可能 **then**
3: **return** $\{R\}$
4: $d \leftarrow R$ を k レコード以上のグループに 2 分割可能な次元
5: $m \leftarrow d$ に関する R の中央値
6: $P \leftarrow \{\, r \in R \mid r[d] \leq m \,\}$
7: $Q \leftarrow \{\, r \in R \mid r[d] > m \,\}$
8: **return** MONDRIAN(P) $\cup$ MONDRIAN(Q)

分割するため，分割する次元を決定する (4 行目)．k レコード以上のグループに分割可能な次元が複数存在する場合は，R に含まれる値の幅が最も大きい (最小値と最大値の距離が最大であるような) 次元を選ぶ．次に分割する次元に関して R の中での中央値を探す (5 行目)．中央値で分割することにより 2 つの分割に含まれるレコード数が均等に近くなる．最後に，中央値より大きいレコード群と小さいレコード群の 2 つのグループを作成し (6, 7 行目)，それぞれについて再帰的に MONDRIAN 関数を適用した結果の和集合を返却する (8 行目)．

4.3.2 情報損失指標

ミクロアグリゲーションにおける情報損失の指標としては sum of squared errors (SSE) を total sum of squares (SST) で割った値 SSE/SST が広く用いられている [26, 47, 74, 86]．そのため本研究でも同じ指標を用いる．SSE はグループ内のデータがどの程度似通っている (同質である) かを表し，次のように定義される．

$$\mathrm{SSE} = \sum_{i=1}^{g} \sum_{j=1}^{n_i} (x_{ij} - \overline{x}_i)^2. \tag{4.1}$$

ただし g は構成されたグループの数，n_i は i 番目のグループの要素数，x_{ij} は i 番目のグループの j 番目のレコード，$\overline{x}_i$ は i 番目のグループに属するレコード値の平均を表す．SSE が小さいほどグループ内のデータの同質性は高い．さらに

グループ間のデータの差異の大きさを treatment sum of squares (SSA) として次のように定義する．

$$\mathrm{SSA} = \sum_{i=1}^{g} n_i(\overline{x}_i - \overline{x})^2. \tag{4.2}$$

ただし $\overline{x}$ は匿名化対象レコードすべてのデータの平均である．SSE と SSA の和として SST を次のように定義する．

$$\mathrm{SST} = \mathrm{SSE} + \mathrm{SSA} = \sum_{i=1}^{g}\sum_{j=1}^{n_i}(x_{ij} - \overline{x})^2. \tag{4.3}$$

最適な k-分割とは SSE を最小化する (同時に SSA を最大化する) ことなので，$[0, 1]$ に正規化された情報損失は SSE/SST で表される．k-分割が最適に近いほど SSE/SST は小さくなる．

最適な k-分割の作成は NP 困難である [74] ため，ヒューリスティクスが用いられる．提案手法もまたヒューリスティクスである．

4.4　提案手法

本節では k-分割のためのクラスタリングアルゴリズムである友引法と，空間分割とクラスタリングを併用した k-分割について詳しく説明する．友引法は情報損失を減少させ，空間分割の併用は計算量を下げることから，これらを組み合わせることで低情報損失かつ高速な匿名化が可能となる．

4.4.1　最近傍グラフを用いたクラスタリング

友引法の処理は 2 段階に分けられる．即ち，友引法はまず最近傍グラフを構築することで連結成分としてレコードのクラスタを発見する．次に連結成分を頂点数 k 以上の小さいグループへ分割することで k-分割を作成する．以降ではこれら 2 段階の処理について順に詳しく説明する．

グラフの構築では，レコードを頂点とみなし，近傍に存在する頂点同士の間にエッジを追加する．なおエッジの向きは考慮しない．近傍点を接続したグラフと

アルゴリズム 8 (k,m)-近傍グラフの構築

入力: レコードの集合 R，匿名化のパラメータ k，グラフ構築のパラメータ m.
出力: (k,m)-近傍グラフ G

1: $G \leftarrow (R, \emptyset)$ ▷ レコードを頂点としエッジのないグラフ
2: **repeat**
3: $C \leftarrow \{\, G_c \in G$ 内の連結成分 $\mid |V(G_c)| < k \,\}$
4: **for each** $G_c \in C$ **do**
5: $V(G_c)$ と $V(G) \setminus V(G_c)$ の間で最近傍の頂点 m 組を G 上で接続
6: **until** $C = \emptyset$

しては，各頂点をその最近傍にある k 頂点と接続した k 近傍グラフ [28] が知られている．k 近傍グラフと k-匿名性は同じ k を用い混乱を招くため，ここでは k 近傍グラフを m 近傍グラフと呼ぶことにする．友引法の第 2 段階では連結成分を分割することで k-分割を作成するため，構築されるグラフの連結成分は全て k 頂点以上を含む必要がある．しかし $m < k$ のとき，m 近傍グラフでは k 頂点未満の連結成分が生じ得る．そこで，友引法では k 頂点未満の連結成分がなくなるまでエッジの追加を繰り返すことで構築される (k,m)-**近傍グラフ**を用いる．具体的な構築手順をアルゴリズム 8 に示す．ただし $V(G)$ と $E(G)$ はそれぞれグラフ G に含まれる頂点の集合とエッジの集合を表す．アルゴリズムはレコードの集合 R に加え，k-匿名化の k と，最近傍グラフ構築のパラメータ m を受け取る．初期状態のグラフ G はレコードを頂点としエッジを含まない (1 行目)．まずアルゴリズムはそのグラフから k 頂点未満の連結成分を抽出する (3 行目)．次に，各連結成分 G_c に含まれる頂点 $v \in G_c$ と連結成分に含まれない頂点 $v' \in V(G) \setminus V(G_c)$ の組み合わせ (v, v') のうち，最も近傍にある m 組の間にエッジを追加する (5 行目)．このような処理を k 頂点未満の連結成分がなくなるまで繰り返す (6 行目)．以上のようにエッジを追加することで，全ての連結成分が k 頂点以上含むグラフを構築することができる．

次に友引法の第 2 段階の処理であるグラフの分割を説明する．グラフの分割では (k,m)-近傍グラフの連結成分を k 頂点以上のさらに小さいグループへ分割

アルゴリズム 9 グラフの分割 (PARTITION 関数)

入力: (k, m)-近傍グラフ G，匿名化のパラメータ k

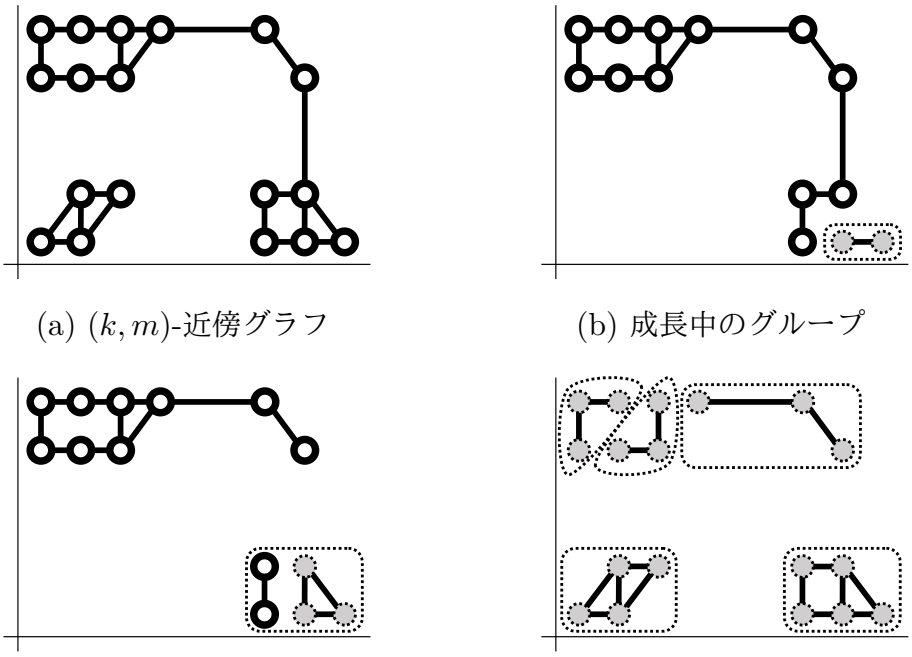
出力: k-分割の結果 (レコードの集合の集合)

```
1: function PARTITION(G)
2:     return ⋃_{G_c∈G 内の連結成分} PARTITIONCC(G_c)
3: function PARTITIONCC(G_c)
4:     if |V(G_c)| < 2k then
5:         return {V(G_c)}
6:     G_1 ← G_c
7:     v ← G_c 上でランダムに選んだ頂点から最も遠い頂点
8:     loop
9:         G_1 から v を切除し，さらに k 頂点未満の連結成分も取り除く
10:        G_2 ← G_c から V(G_1) を除いたグラフ
11:        if |V(G_2)| ≥ k then
12:            break
13:        v ← G_c における V(G_2) の隣接頂点で V(G_2) の重心に最も近い頂点
14:    if V(G_1) = ∅ then
15:        return {V(G_c)}
16:    return PARTITION(G_1) ∪ PARTITION(G_2)
```

する．この際グラフの連結成分を利用してクラスタ構造を捉える点が友引法の特徴である．(k, m)-近傍グラフでは近傍の頂点が接続されていることから，近傍に存在するレコードの集団は互いに密に接続された連結成分として抽出される．この性質を利用し情報損失の小さいグループを形成するため，k 頂点以上のグループとなるまで次の 2 つの処理を繰り返す．まず形成中のグループの重心から最も近傍にある頂点をグループに追加すると共に，連結成分から切除する．このとき，切除された頂点を経由して大きい連結成分に接続されていた頂点は別の連結成分に分離される．そこで，次に k 頂点未満の連結成分をグループに追加する．これにより密に接続された頂点を一つのグループに含めることができる．詳細な処理の手順をアルゴリズム 9 に PARTITION 関数として示す．PARTITION

関数は (k,m)-近傍グラフ G に含まれる各連結成分に対して PartitionCC 関数を適用した結果を集約して返却する．PartitionCC 関数は連結成分 G_c を k 頂点以上の 2 つのサブグラフ G_1 と G_2 へ再帰的に分割する．処理は $G_1 = G_c$ から開始し，徐々に G_1 から G_2 へ頂点を移動することによって行われる．まとまりの良い連結成分へ分離するため，移動は G_c の端にある頂点から開始する．具体的には，G_c 内でランダムに選んだ頂点から最も遠い頂点を選び，v と置く (7 行目)．次に，G_1 から v を切除し，それによって生成された k 頂点未満の連結成分も取り除く (9 行目)．G_c から $V(G_1)$ を除いたグラフは G_2 と置く (10 行目)．G_2 の頂点数が k 以上になった場合はループを抜ける (11, 12 行目)．そうでない場合は，$V(G_2)$ の重心に最も近い頂点を v と置き直し，もう一度ループする (13 行目)．重心から遠い頂点に対する無駄な計算を避けるため，重心の最近傍点は $V(G_2)$ の隣接頂点から探す．$V(G_2)$ の隣接頂点集合とは $\{\, v \mid (u,v) \in E(G_c), u \in V(G_2), v \notin V(G_2) \,\}$ である．この手順を繰り返すと，k 頂点未満の連結成分の除去 (9 行目) において G_1 の頂点がなくなる場合がある．その場合は G_c を分割せずそのまま返却し (15 行目)，そうでなければさらに分割できる可能性があるため，再帰する (16 行目)．友引法は以上のような分割によって情報損失の少ない k-分割を得る．

アルゴリズムの流れを図 4.3 を用いて説明する．ただし $m = 2, k = 3$ とする．まずアルゴリズム 8 により構築される (k,m)-近傍グラフが図 4.3(a) である．図 4.3(a) の左下の連結成分は $2k$ 頂点未満なのでこれ以上分割されず，そのまま一つのグループとなる．グループとなった頂点は取り除き，図 4.3(a) の大きい連結成分の分割だけを考える．グループ形成の始点が右下の頂点になったとすると，最初に最近傍の 2 頂点が収集されグラフから取り除かれる (図 4.3(b))．次にその 2 頂点の重心から最も近い頂点を収集する (図 4.3(c))．これにより頂点数 2 の連結成分ができるため，それも同じグループに含める．結果として図 4.3(c) の右下点線で囲われた頂点が一つのグループとなる．同様にしてグループに含まれない頂点がなくなるまでグループの作成を繰り返す．最終的に作成される k-分割の一例は図 4.3(d) である．グループ形成の始点がどのように選ばれるかによって分割結果は変化するため，結果は一意に定まらない．図 4.3(c) に見られ



(a) (k, m)-近傍グラフ　(b) 成長中のグループ

(c) グループ完成　(d) 最終的にできる k-分割

図 4.3: 友引法の過程

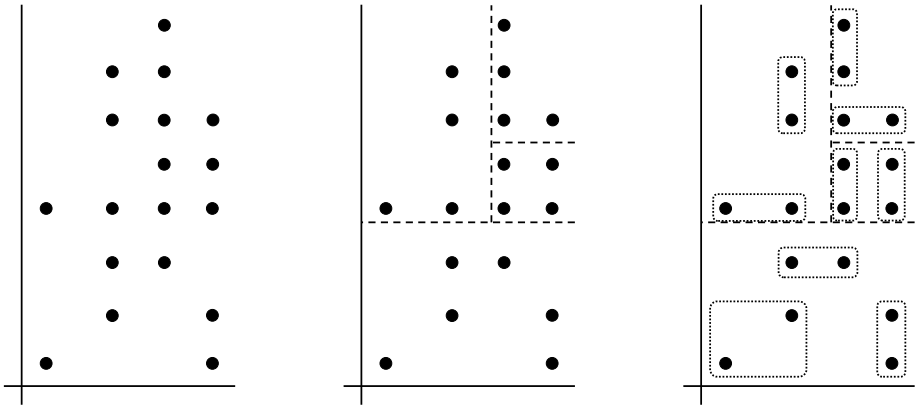
るように連結成分から切除した頂点と接続されている頂点 (友人) も一緒に切除されることから，本研究ではこのアルゴリズムを友引法と呼んでいる．

4.4.2　空間分割とクラスタリングの併用

空間分割とクラスタリングの併用は，具体的には次の 2 段階の処理によって行われる．

1. **粗粒度分割**　パラメータ $k_\sharp$ $(k_\sharp \geq k)$ を受け取り，空間分割によってレコード全体を $k_\sharp$ レコード以上含むグループへ分割する．
2. **細粒度分割**　粗粒度分割で得た各グループに対してさらにクラスタリングに基づく分割を行い，k-分割を作成する．

本章では粗粒度分割時の分割手法として Mondrian を用いるが，Iwuchukwu らによる R-tree ベースの手法 [48] 等を用いることもできる．ただし格子状に空間



(a) 与えるレコード (b) 粗粒度分割後 (c) 細粒度分割後

図 4.4: 空間分割とクラスタリングを併用した匿名化の過程 ($k = 2$, $k_\sharp = 4$)

を分割するアプローチは匿名化に適さない．理由は 2 つある．まず $k_\sharp$ レコード未満の空間が作成されてしまう可能性がある．さらに各空間に含まれるレコードの数に偏りが生じる可能性がある．細粒度分割では計算量 $O(n^2)$ 以上であるアルゴリズムの利用を想定しているため，全体の計算量を最小化するため粗粒度分割で作られるグループはなるべく均等な数のレコードを含むことが望ましい．Mondrian を併用して k-分割を行う過程の例を図 4.4 に示す．図 4.4(b) では空間分割によって 4 レコード以上含むグループへ分割され，図 4.4(c) ではクラスタリングによって 2 レコード以上のグループへ分割されている．

クラスタリングによる匿名化に空間分割を併用する利点は 3 つある．第 1 に，小さい情報損失と高速な処理を両立できる．匿名化は実用上 $k = 5$ 程度の利用が多い [29]．そのためレコード全体の大域的分布を考慮せず，粗粒度分割で作られたグループに含まれる局所性なレコードの分布のみ考慮して k-分割を行ったとしても情報損失への影響は限定的である．第 2 に，情報損失と処理時間のどちらを優先するか，利用時の状況に応じて連続的に調整できる．小さい情報損失と高速な処理を両立する中でも $k_\sharp$ の大きさによって結果は変化する．大きい $k_\sharp$ では空間分割の特徴が強く現れ高速・高情報損失になる．逆に小さい $k_\sharp$ ではクラスタリングの特徴が現れ低速・低情報損失となる．第 3 に，メモリに格納しき

れないほど巨大なデータに対する k-分割処理を高速化できる．細粒度分割時は粗粒度分割で生成されたグループを一つずつ処理するため，処理中でないグループはストレージ上へスワップアウトできる．また，buffer tree [4, 24] を用いることにより，空間分割で巨大なデータを扱う際の I/O 効率を高められることが Iwuchukwu らによって指摘されている [48].

4.4.3　定性的議論

友引法と空間分割併用のそれぞれについて，入力データおよびパラメータに対する振る舞いを分析するため計算量を求める．なお，空間分割の併用では本研究の提案である友引法と組み合わせた場合について分析を行う．

友引法

n レコード含むデータを k-分割する場合について考える．議論をシンプルにするため，k は n の約数とし，全てのグループが k レコードを含むように分割されるものとする．友引法は (k, m)-近傍グラフの構築とその分割という 2 段階に分けることができる．まずグラフの構築では，各連結成分の大きさが k 頂点以上となるように最近傍の m 頂点と接続する．連結成分の大きさはエッジ追加の際に 2 つの連結成分の大きさの和を求め随時更新することで得られる．一方最近傍の連結成分を発見するために全頂点間で距離の計算が必要になる．従ってグラフ構築の計算量は $O(n^2)$ である．

次にグラフの分割について考える．Partition 関数 (アルゴリズム 9) に示されているように，分割は連結成分ごとに行う．ある連結成分 G_c が与えられたときの分割処理は PartitionCC 関数が行う．この関数内で計算量に注目すべき処理は次の 3 つである．1 つ目は 7 行目のランダムに選んだ頂点から最も遠い頂点の選択である．最も遠い頂点の探索は G_c の全頂点が対象になるため，計算量は $O(|V(G_c)|)$ である．2 つ目は 9 行目における連結成分の検出である．DFS による連結成分検出の計算量は $O(|V(G_c)| + |E(G_c)|)$ である．3 つ目は 13 行目における重心に最も近い頂点の探索である．ここではグラフ構造に基づき，$V(G_2)$ の隣接頂点の中から探索を行う．ここでは全てのグループが k 頂点含むように

分割される場合を考えているため，G_2 の頂点数は最大で k である．従って G_c の平均次数を d とおくとグループの隣接頂点数は $O(dk)$ である．以上の処理を最大 k 回繰り返すことでグループを 1 つ作成する．G_2 はちょうど k 頂点を含むので，16 行目の Partition(G_2) の呼び出しはこれ以上再帰しない．従って連結成分 G_c 内で 1 つのグループを分割するための計算量は次のとおりである：

$$O\left(k\left(|V(G)| + (|V(G)| + |E(G)|) + dk\right)\right) = O\left(k\left(|V(G)| + |E(G)|\right)\right). \tag{4.4}$$

この変形では $dk \leq |V(G)|$ を利用している．式 4.4 から読み取れるように，1 つのグループの分割はグラフサイズに対して線形の計算量である．従って，ここまで 1 つの連結成分 G_c について考えてきたが，グラフ全体 G についても同様に計算量を考えることができる．グラフ全体では頂点数 n であり，また最近傍の m 頂点と接続しているのでエッジ数は最大で mn である．従ってグラフ全体から 1 つのグループを分割する計算量は

$$O\left(k\left(n + mn\right)\right) = O\left(kmn\right) \tag{4.5}$$

となる．グループは $\frac{n}{k}$ 個作成されるので，全体の計算量は次のとおりである．

$$O\left(kmn\frac{n}{k}\right) = O\left(mn^2\right). \tag{4.6}$$

この計算量は (k, m)-近傍グラフのエッジ数を最悪ケースである mn と見積もることで得られている．しかし実際には 2 つの頂点が相互にエッジを張り 1 つのエッジに統合される場合があるため，実際にはこれよりも小さい計算量になると考えられる．

空間分割の併用

Mondrian による粗粒度分割で $k_\sharp$ レコードを含むグループへ分割し，その後各グループ内に友引法を適用して k-分割を作成する場合について考える．議論をシンプルにするため，$k_\sharp$ は n の約数であり，粗粒度分割ではちょうど $k_\sharp$ レコードのグループへ分割できるものとする．k-分割処理全体の計算量は粗粒度分割と友引法それぞれの計算量の和である．まず粗粒度分割の計算量を考え

る．Mondrian で 1 回分割するときの計算量は $O(n)$ である [61]．分割の度にグループの数は 2 倍になる．分割を繰り返し $\frac{n}{k_\sharp}$ グループを作成するので，粗粒度分割の計算量は $O\left(n \log \frac{n}{k_\sharp}\right)$ である．次に友引法の計算量を考える．粗粒度分割の結果 $k_\sharp$ レコードを含むグループが $\frac{n}{k_\sharp}$ 個作られているので，計算量は $O\left(\frac{n}{k_\sharp} m k_\sharp^2\right) = O\left(nmk_\sharp\right)$ である．以上より空間分割を併用した場合の計算量は次のように求められる：

$$O\left(n \log \frac{n}{k_\sharp} + nmk_\sharp\right) = O\left(n\left(\log \frac{n}{k_\sharp} + mk_\sharp\right)\right). \tag{4.7}$$

ここで $k_\sharp$ に対する振る舞いを見るために n と m を定数とみなすと，

$$O\left(n\left(\log \frac{n}{k_\sharp} + mk_\sharp\right)\right) = O\left(-\log k_\sharp + k_\sharp\right) \tag{4.8}$$

$$= O\left(k_\sharp\right) \tag{4.9}$$

となる．つまり m を固定して同じデータを匿名化する限りは $k_\sharp$ に線形に計算量が増加することが分かる．

4.5　評価

実際にいくつかのデータセットを匿名化し，友引法および空間分割併用の効果を情報損失と処理速度の観点で既存手法と比較する．実験に使用した計算機の CPU は Intel Core i7-4770K 3.50GHz，メモリは 32.0GB である．実験用プログラムの実装には Haskell を用いた．実験に使用した統計データを表 4.5 に示す．Census, EIA, Tarragona はミクロアグリゲーションの，Adult は k-匿名化の既存研究においてそれぞれデファクトスタンダードのデータセットである [6, 12, 27, 49, 58, 60, 61, 64, 86]． Census と Tarragona は配布データに含まれる全ての属性を用いる．EIA からは UTILITYID, UTILNAME, YEAR の 3 属性を除く．UTILITYID と UTILNAME は個人 (団体) を特定する一意な識別子であり，YEAR は全レコードで同一の値を持つためである．Adult は既存研究 [6, 49, 61] に倣い，次の 8 属性を用いる：age, work class, education, marital status, occupation, race, gender, native country．カテゴリ値には適当な数値

表 4.5: 匿名化するデータセット

名前	レコード数	属性数
Census[*1]	1080	13
EIA[*1]	4092	12
Tarragona[*1]	834	13
Adult [25]	30162	8
Random (一様分布)	3000	2

表現を与えた上で，各データセットのそれぞれの属性について，データセット中に登場する値の最大値が 1.0，最小値が 0.0 となるように正規化する．値の範囲が大きいレコードがあるとその属性がレコード間距離の算出において支配的になってしまうためである．比較対象のアルゴリズムには V-MDAV を使用する．V-MDAV はミクロアグリゲーションの研究で手法のベースや比較対象としてしばしば使用されている [46, 50, 54, 68]．さらに友引法と同様生成されるグループの数が可変であるため，比較に適している．グループの大きさ k は実用上 $k = 5$ の利用が多い [29] ことから，実験でも特に指定がない限り $k = 5$ を使用する．

評価は友引法に次いで空間分割併用の順序で行う．空間分割併用の評価には併用するためのクラスタリング手法が必要となるため，先にクラスタリング手法である友引法を単独で評価し，その性能を明らかにする．その後に，空間分割併用によって情報損失と処理速度がどのように変化するかを評価する．

4.5.1 友引法の評価

まず空間分割を併用せず友引法単体の性能を評価する．

m の最適値

友引法ではグラフの構築時に各頂点を接続する最近傍頂点の数 m を受け取り (k, m)-近傍グラフを構築する．m の値によって構築されるグラフの構造が変化

[*1] http://neon.vb.cbs.nl/casc/CASCtestsets.htm

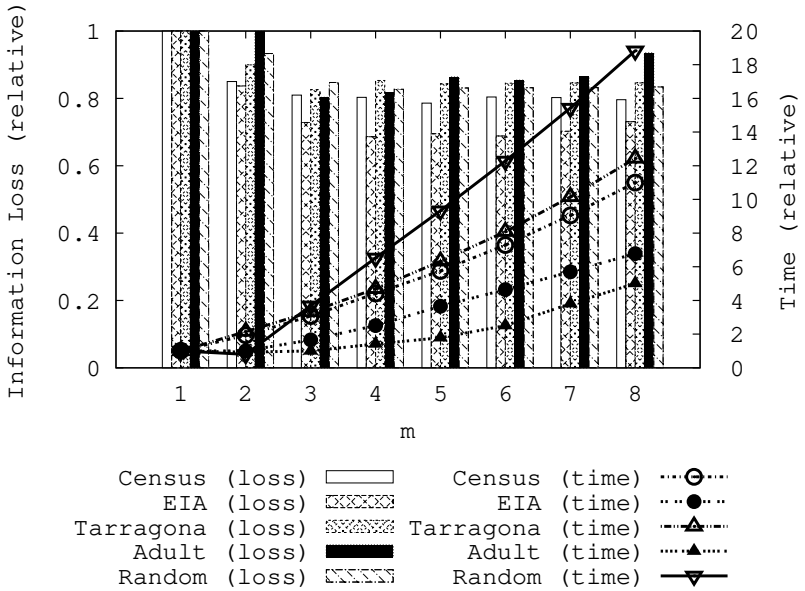


図 4.5: m に対する情報損失と処理時間

するため，k-分割の結果も変化する．最適な m を探すため，m の値を変化させつつ各データセットの匿名化で生じる情報損失と実行時間を調査した．結果を図 4.5 に示す．データセットごとの差異が大きいため，各データセットについて $m = 1$ で得られる値を 1 として情報損失と実行時間を正規化した．実行時間はいずれのデータセットでも m の値と共に増加している．m が大きいほど (k, m)-近傍グラフが多くのエッジを含むようになり，グラフ構造の処理コストが増大するためである．特に Tarragona, Adult, Random はほぼ m に対して線形に増加しており，4.4.3 節で明らかにした友引法の計算量 $O(mn^2)$ に符合する結果となった．一方，情報損失は $m = 3$ までに大きく減少し，そこからはデータセットごとに差はあるものの小幅の増減が続いている．グループに追加する頂点はグループの隣接頂点から選択されるため，m が大きいほど隣接頂点の数が増加し選択肢が広がる．そのため，$1 \leq m \leq 3$ の間では選択肢の増加によってまとまりの良いグループを作成しやすくなり，情報損失が減少したと考えられる．しかし実験結果では $m < 3$ においてほぼ情報損失が減少していないため，$m = 3$ の時点でほとんどのデータセットにおいて十分な選択肢が存在していることが分か

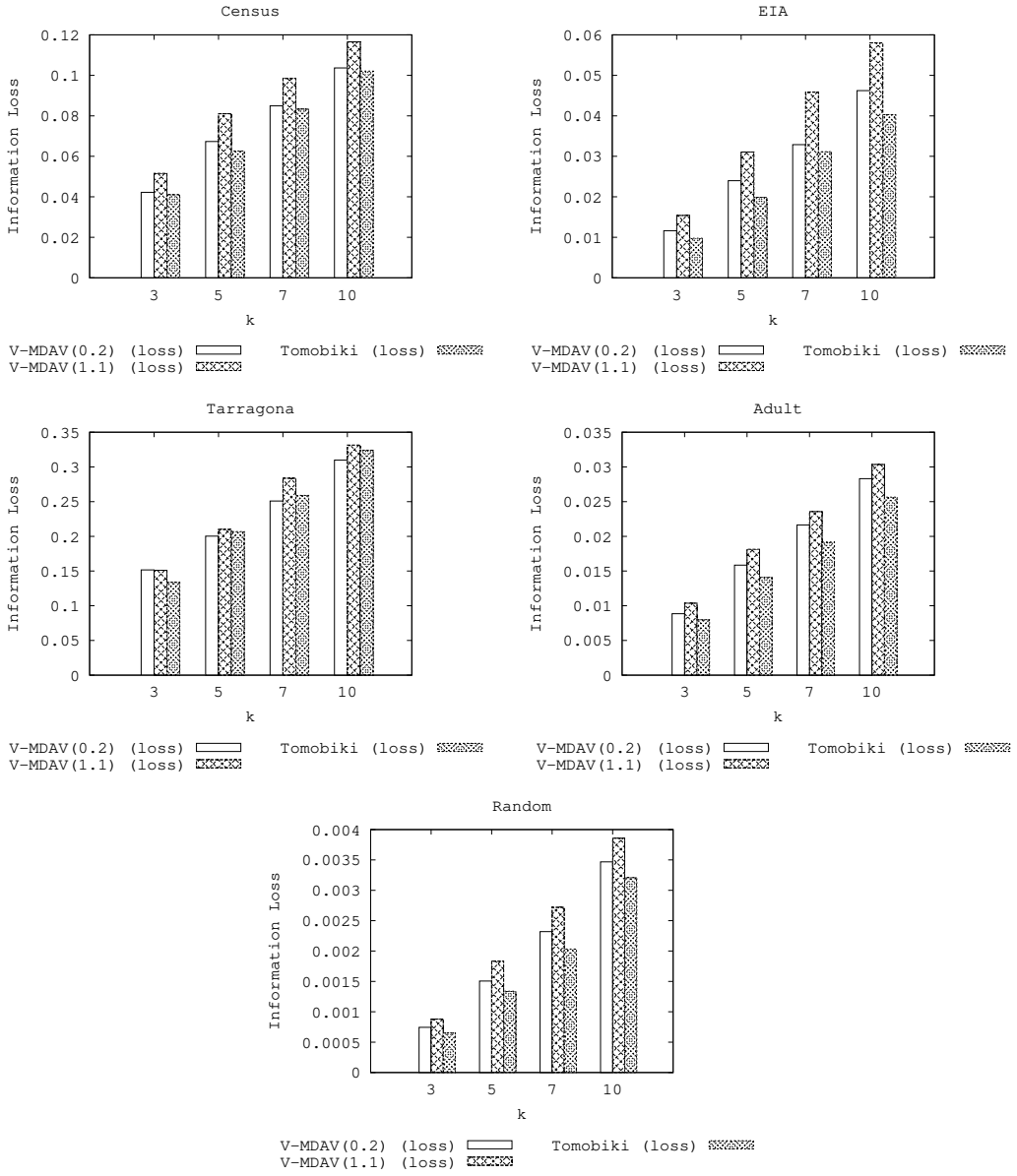


図 4.6: k に対する情報損失量の変化

る．全データセットで最小の情報損失を示す共通の m は見つからないものの，m を大きくするにつれ実行時間が増大することも踏まえると $m = 3$ が標準のパラメータとして適当であると言える．

情報損失

次に，データセット毎にグループの大きさ k を変化させ情報損失の大きさを比較した．文献 [86] にならい，比較対象である V-MDAV に与えるパラメータ γ は 0.2 と 1.1 を用いた．また最高の性能を調査するため (k, m)-近傍グラフの m はデータセット毎に最適値を用いた．即ち Census は 5，EIA と Random は 4，他は 3 である．結果を図 4.6 に示す．全体の傾向として k が大きいほど情報損失も大きい．グループに含まれるレコードが多いほど値の分散も大きく，平均値で置き換えたときに値の変動が大きいためである．また情報損失量は使用するデータセットによって大きく異なる．レコードの属性数が少ない Adult と Random は比較的情報損失が小さく，属性数 12 または 13 である他のデータセットは大きい．この差は次元の呪い [1] によって属性数が多いほどレコード間の距離が遠くなることの影響と考えられる．友引法の性能に着目すると，多くのデータセットにおいて V-MDAV より小さい情報損失量となっている．特に EIA の $k = 3$ では V-MDAV ($\gamma = 0.2$) と比べ約 16% 情報損失を減少させている．一方で Tarragona の $k \geq 5$ では V-MDAV ($\gamma = 0.2$) よりも友引法の情報損失が大きい．Tarragona は Census と比べ疎なデータの分布になっていることが指摘されている [14]．友引法はグラフの構築を通じてレコードのクラスタを発見しようとしているが，疎なデータ分布のためにクラスタを発見できないことが情報損失増大の原因となっている可能性がある．

レコード数に対する実行時間の変化

友引法のスケール性を評価するため，Adult データセットの一部を用い，処理対象のレコード数に対する実行時間の変化を調査した (図 4.7)．ただし $m = 3$ である．レコード数 n に対し $O\left(n^2\right)$ である V-MDAV と沿うように友引法も実行時間が増加している．従って友引法も n^2 に比例して計算時間が増大していると考えられる．この振る舞いは 4.4.3 節で得た計算量 $O(mn^2)$ とも符合する．

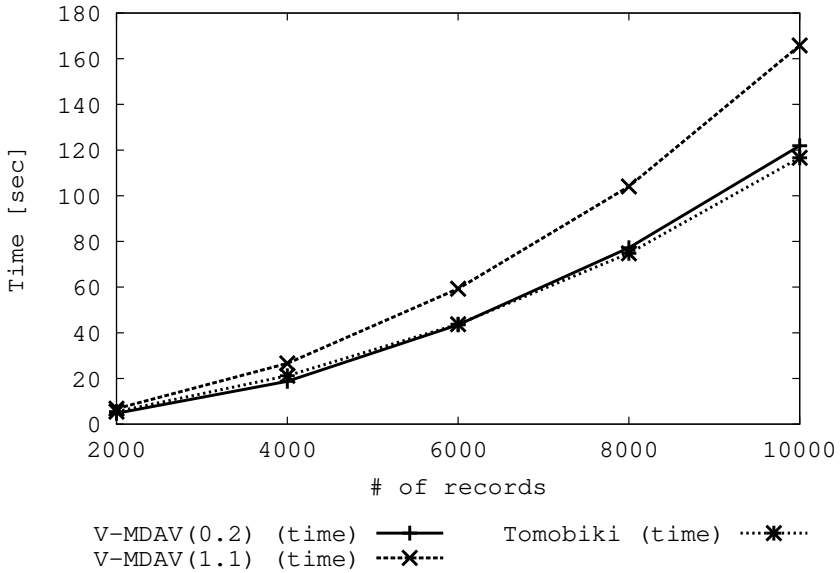


図 4.7: レコード数に対する情報損失と処理時間の変化

4.5.2 空間分割併用の評価

次に空間分割と友引法を併用した場合の情報損失と実行時間について調査した．便宜上空間分割を併用する友引法を友引 $\sharp$ と呼ぶことにする．データセットはレコード数が多い EIA と Adult を用いた．Mondrian のみ用いた場合，および空間分割によって作られる空間のレコード数 $k_\sharp$ を 5 から各データセットのレコード数まで変化させた場合の結果を図 4.8 に示す．横軸左端の Mondrian は最も大きい情報損失を示す一方，最も高速である．そこから横軸右へ向かって $k_\sharp$ が増大するにつれ徐々に友引法の影響が支配的となり，処理時間が延びる代わりに情報損失は減少していく．Adult では Mondrian よりも $k_\sharp = 5$ の場合の情報損失が小さい．これは Mondrian のみの場合図 4.2 のように $2k$ レコード以上含むグループが作られ得るが，$k_\sharp = 5$ では友引法によってそのようなグループをさらに分割できるためである．$k_\sharp$ がデータセットのレコード数と等しい場合 (横軸右端) は一切空間分割が行われず，全て友引法で処理される．そのため情報損失は最小となるが，同時に処理時間は最大となる．計算時間は $k_\sharp$ に対してほぼ

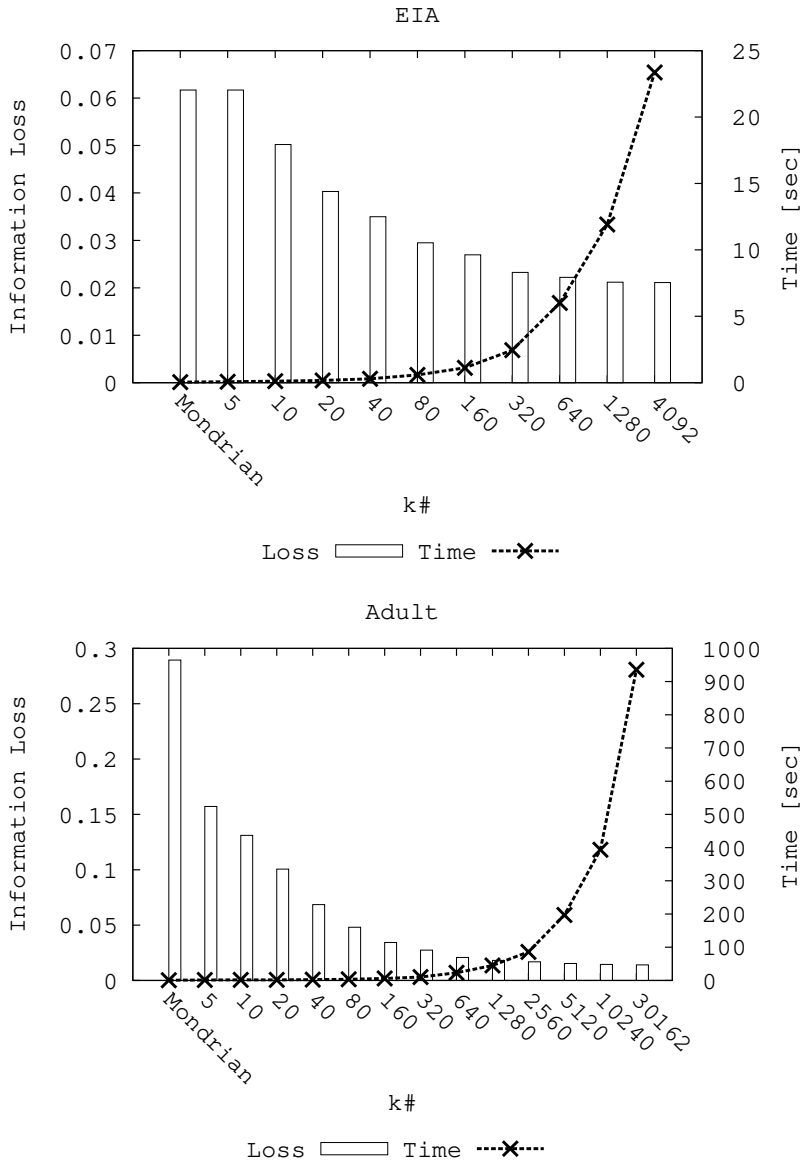


図 4.8: $k_\sharp$ に対する情報損失と処理時間の変化

線形に増加しており，この振る舞いは 4.4.3 節で得た計算量 $O(k_\sharp)$ と符合する．

k が小さい場合，レコード全体を見ずに近傍にあるレコードにのみ着目しても良い k-分割が可能である，という仮説が空間分割の併用の基にあるアイデアである．そこで空間分割の粒度と情報損失の観点で図 4.8 を見ると，この仮説に従った結果を得ることができている．細かい数値での比較のため EIA において

表 4.6: EIA データセットにおける友引 ♯ と V-MDAV の比較

アルゴリズム	情報損失	処理時間 [秒]
友引 ♯ ($k_\sharp = 5$)	0.0617	0.0776
友引 ♯ ($k_\sharp = 320$)	0.0233	2.47
友引 ♯ ($k_\sharp = 4092$)	0.0211	23.5
Mondrian	0.0617	0.0400
V-MDAV ($\gamma = 0.2$)	0.0240	25.0
V-MDAV ($\gamma = 1.1$)	0.0311	40.0

特徴的な点の値を表 4.6 にまとめた．空間分割を用いない場合 ($k_\sharp = 4092$) と $k_\sharp = 320$ における情報損失はそれぞれ 0.0211 と 0.0233 であり，ほぼ変化がない．一方で処理時間はそれぞれ 23.5 秒と 2.47 秒であるため，$k_\sharp = 320$ では情報損失の変化を抑えつつ空間分割の併用により約 10 倍の高速化を達成している．また V-MDAV との比較では，V-MDAV ($\gamma = 0.2$) で 25.00 秒かかった情報損失量 0.0240 を友引 ♯ は $k_\sharp = 320$ において 2.47 秒で実現しており，こちらも約 10 倍高速である．

以上のように，友引法は多くのデータセットにおいて既存手法よりも小さい情報損失を示し，さらに友引 ♯ は空間分割の併用によって情報損失の増大を抑えつつ大幅な高速化が可能であることを確認できた．

4.6 結論

プライバシ保護のため k-匿名化が利用されている．これまでクラスタリングに基づく手法と空間分割に基づく手法が k-匿名化のために提案されてきたが，前者は情報損失が大きく，後者は処理が遅い．そこで本研究では 2 つの手法を提案した．1 つ目は，最初にレコードを頂点とするグラフを構築することでクラスタ構造を捉える新しい k-分割アルゴリズムの友引法である．2 つ目は，空間分割を併用することにより，クラスタリングに基づく手法が持つ情報損失の少なさを保ちつつ匿名化処理を高速化する手法である．実際の統計データを用いた実験に

より，友引法は既存手法と比べ最大 16% 情報損失を減少させることが分かった．さらに空間分割を併用することで既存手法と同等の情報損失のデータを約 10 倍高速に得ることができた．

本研究において匿名化手法はクラウド環境上でパーソナルデータを利用するために利用される．そのため，匿名化時の情報損失が小さいことはクラウド上でのグラフクエリ処理自体の有用性に関わる．また，匿名化はデータ所有者の計算環境で行われるため，限られた計算資源で大量のデータを処理できなければならない．提案手法はこのような要求条件を小さい情報損失と高速な処理の両立によって達成した．提案手法の大きな特徴の一つは，表形式のデータを入力としつつも匿名化のためにグラフを利用することである．4.1 節で示したように，プロパティグラフはパーソナルデータに着目すると表として捉えることができる．そのため本章では k-匿名化の研究としてより一般的な表形式のデータを題材に手法の説明と評価を行った．一方で，提案手法は表形式のデータからグラフ ((k,m)-近傍グラフ) を再構築する．このことは情報損失の削減に役立つと同時に，第 2 章のリオーダリング手法を適用することによる局所性向上の余地を生んでいる．(k,m)-近傍グラフは匿名化処理時に作成されるグラフであるため，処理中に直ちにリオーダリングする必要がある．従って高速なリオーダリングが可能な Rabbit Order にとって特に適した用途であると言える．

将来の課題としてはパラメータフリー化が挙げられる．空間分割の併用では $k_\sharp$ をパラメータとして指定する必要がある．また，友引法で使用する (k,m)-近傍グラフの m や V-MDAV の γ のように，グループ内のレコード数が可変であるような k-分割アルゴリズムはパラメータを持っている．最適なパラメータを推定する技術や，パラメータを受け取らずに多様なレコード分布のデータに対応できるアルゴリズムが必要である．

第 5 章

結論と今後の課題

5.1　本論文のまとめ

大規模なグラフデータを活用するため，本論文ではクラウド上の計算資源を利用した高速なクエリ処理を実現する手法について議論した．

第 1 章では本研究が取り組む研究の背景と目的を説明した．グラフは柔軟性と表現力が高いデータ形式であり，実世界の様々な場面で利用されている．特にグラフに対するサブグラフマッチングクエリは最も典型的なクエリである．クラウド環境の計算資源は大規模なグラフをインメモリで処理するために有用であることから，本研究ではクラウド環境上のグラフクエリ処理に焦点を当てることを説明した．ただしクラウド環境ではパーソナルデータの利用にリスクと規制が存在するため，本研究では匿名加工情報と高速なクエリ処理エンジンをクラウド環境に配置し，クラウド環境上でクエリ処理を行うという枠組みを用いることを述べた．

第 2 章では，グラフ処理全般において性能上の足枷となる局所性の低さを改善するためのリオーダリング手法を提案した．グラフ分析におけるメモリアクセスの局所性を向上させるため，リオーダリングによってデータ配置を事前に最適化するアプローチが広く用いられている．しかしながら既存のアルゴリズムは効果的な頂点順序の生成に長い時間を要するため，リオーダリングと後続のグラフ分析を合計した全体の処理時間をむしろ増加させてしまうという問題がある．本章

において全体の処理時間を短縮するための並列リオーダリングアルゴリズムである Rabbit Order を提案した．Rabbit Order は高い局所性と高速なリオーダリングを同時に達成するとともに，効率的な並列処理により高いスケーラビリティを示す．実験の結果，全体処理時間の観点で Rabbit Order は平均 2.2 倍グラフ分析を高速化することが確認された．

第 3 章ではグラフクエリ処理における無駄な探索を削減するため，効果的な枝刈りを行うサブグラフマッチングアルゴリズムを提案した．サブグラフマッチングの解法としてはバックトラッキングによる探索が一般的である．即ち前のステップで得られた部分的な埋め込みを基に新しい埋め込みを生成する手続きを再帰的に適用することで完全な埋め込みを発見する．探索中，完全な埋め込みに到達しないことが判明した部分埋め込みは探索失敗として破棄される．既存のサブグラフマッチング手法はグラフの構造を分析する前処理によってバックトラッキングの探索範囲を狭めることで失敗を削減してきた．しかし前処理は単純なルールに基づいているため，削減可能な失敗は限定的である．本章では，「失敗から学ぶ」サブグラフマッチングアルゴリズムを提案した．提案手法はバックトラッキングの最中に探索失敗の履歴から失敗となる部分埋め込みのパターンを抽出する．そしてパターンに符合した部分埋め込みを早期に破棄することで余分な探索を枝刈りする．実験の結果提案手法は既存手法比で最大 4 桁高速であることが確認された．

第 4 章では，クラウド環境へアップロードする匿名加工情報を小さい情報損失で高速に作成するための k-匿名化手法を提案した．k-匿名化のためには与えられたレコードの集合を k レコード以上から成るグループの集合へと分割する必要がある．その際，データの変換で生じる情報損失を抑えられるようなグループを高速に作成できることが望ましい．これまで空間分割に基づく分割手法とクラスタリングに基づく分割手法が提案されてきたが，これらは小さい情報損失と高速な処理を両立できていない．本章では 2 つの手法を提案した．1 つ目は，レコードを頂点とするグラフの構築によりクラスタを捉え，情報損失の小さい分割を作成するアルゴリズムである．2 つ目は，小さい情報損失と高速な処理を両立するための，空間分割とクラスタリングの併用である．2 つの手法を組み合わせ

ることで，既存手法と比べ最大 10 倍高速に情報損失の小さい分割を作成可能であることが確認された．

パーソナルデータの匿名化を通じて安全にクラウド環境を利用するという枠組みは新しいものではない [82]．本研究の貢献として本質的なのは，その枠組みにおいてグラフクエリを可能にするにあたり必要となる要素技術について効率的なアルゴリズムを提案したことである．情報社会で生み出されるデータはますます増加している．グラフ表現を用いることでデータに含まれる事物を柔軟に関連付けて蓄積することが可能である．しかし大規模なデータを蓄積するのみでは意味がなく，そこから情報を得ることができなければならない．サブグラフマッチングによるグラフクエリはグラフから情報を得るための最も基礎的な技術である．本研究の提案手法を用いることにより，データ所有者は手元にある大規模データを小さい情報損失で高速に匿名化し，クラウド上ではリオーダリングによって局所性の向上したグラフに対する効率的な探索によってクエリを処理することができる．手元の計算資源に縛られることなく大量の情報を活用していくために本研究の成果は大きく貢献すると期待される．

5.2　今後の課題

データの規模およびそれを活用するために必要な計算資源の増大と呼応するように，クラウド環境で提供される計算機の性能も向上し柔軟性を増している．実際に，AWS において 12TB ものメモリを搭載したインスタンスの提供が始まったのは 2018 年 9 月末のことである[*1]．このような計算資源の活用が大規模データの利用において重要であることは今後も変化しないと考えている．また個人データに対する法的な規制も今後大幅に緩和されるとは考え難い．2017 年施行の改正個人情報保護法において匿名加工情報に対し自由度の高い利用が認められたことから，匿名化は個人データの利用において暗号化など他の安全確保手段とは一線を画す技術となるだろう．以上のように本研究の背景となった状況は今後

[*1] https://aws.amazon.com/jp/about-aws/whats-new/2018/09/introducing-amazon-ec2-high-memory-instances-purpose-built-to-run-large-in-memory-databases/

も続いていくと考えられる．

このような状況に鑑み，本研究の今後の展開として考えられる課題は 3 つある．1 つ目は効率的な並列処理の実現である．1CPU コアあたりの性能向上は鈍化傾向にあるため，並列処理の重要性はますます高まっている [77]．また，クラウドサービスにおいて大容量のメモリを搭載したインスタンスは同時に多くの (仮想) コアを搭載している．例えば 2019 年 1 月時点では，パブリッククラウドの主要なベンダである AWS, Microsoft, Google において 1TB 以上のメモリを搭載したインスタンスは少なくとも 64 コアを備える[*2]．並列化は逐次処理アルゴリズムを基にしつつ，さらにコア間通信や並行性制御などのコストを考慮しながら行われる．そのため第 2 章の提案手法に対して行ったように，まず効率的な逐次処理アルゴリズムを確立し，次にそれを並列化するという 2 段階のアプローチが効果的である．本研究ではサブグラフマッチングと匿名化について高速な逐次処理アルゴリズムを提案した．そこで今後は第 2 段階としてこれらを並列化し，計算資源の効果的な利用を可能にすることが課題である．

2 つ目は本研究で提案した手法を一つのシステムとして統合することである．本研究はクラウド環境上のクエリ処理に必要とされる要素技術の確立を目的としているため，各提案手法の位置づけを明確にしつつ，評価は個別に行った．提案手法はそれぞれ本研究全体としての目的にとらわれない高い汎用性を持ち，単独でも多くの利用シーンを持つ重要な技術である．しかしながらクラウド環境上のクエリ処理を実際に行うためにはこれらを一体化して動作させる必要がある．また一体化によって明らかになる新たな課題が存在する可能性もある．そこで提案手法をインメモリグラフデータベースとして統合すると共に，その状態で評価を行うことで本研究の貢献はより実用に寄り添ったものになると期待される．

3 つ目は省メモリかつ効率的なグラフクエリ処理の検討である．本研究の枠組みは大容量のメモリを搭載したクラウド上の計算環境を利用するために匿名化を

[*2] AWS の x1e.16xlarge は 64 コア (https://aws.amazon.com/jp/ec2/instance-types/x1e/)，Microsoft の Standard_M64s も同じく 64 コア (https://docs.microsoft.com/ja-jp/azure/virtual-machines/linux/sizes-memory)，Google の n1-ultramem-80 は 80 コア (https://cloud.google.com/compute/docs/machine-types) を提供する．

用いる．しかし匿名化はデータの変動を伴う．第 4 章の匿名化手法は変動を低減するものの，多くの場合匿名化前のデータと匿名化後のデータではクエリの結果が一致しない．従ってその差異を許容できない用途では本研究の枠組みを適用することが困難である．低コストに大規模グラフの利用を可能にするためのアプローチとしては，他にも外部メモリの利用やグラフの圧縮がある．外部メモリの利用では，グラフデータの一部のみをメモリに格納し，その他を SSD のようなメモリよりも低速だが安価なストレージに格納する．グラフの圧縮では，メモリ上のデータを圧縮したグラフに置き替えることで大規模グラフ処理のメモリフットプリントを削減する．これらのアプローチの先行研究としては文献 [15, 80] があるが，非圧縮のグラフデータ全体をメモリに格納する場合と同等の性能の実現は極めて挑戦的な課題である．なぜならば外部メモリの利用ではメモリにないデータへのアクセスが発生した際のレイテンシが大きく，グラフの圧縮では展開のための計算時間が追加で生じるためである．しかし 2 章のリオーダリングアルゴリズムによって局所性を向上させることで，メモリ上に存在するデータへのアクセスを増やすことができる．また 3 章のサブグラフマッチングアルゴリズムでは探索の枝刈りによってグラフデータに対するメモリアクセスが削減される．従ってメモリにグラフデータ全体を格納できない場合においても高い性能を達成する余地はあると考えている．

謝辞

本研究の実施にあたり，懇切なるご指導と格別のご高配を賜りました大阪大学大学院情報科学研究科マルチメディア工学専攻 鬼塚真教授に謹んで御礼申し上げます．本論文の執筆にあたり多数の有益なご助言をくださいました大阪大学大学院情報科学研究科マルチメディア工学専攻 原隆浩教授，藤原融教授に心より感謝申し上げます．講義，学生生活を通じて，学問に取り組む姿勢をご教授頂きました大阪大学大学院情報科学研究科マルチメディア工学専攻 松下康之教授，下條真司教授に厚く感謝申し上げます．本研究を共に推進し，熱心にご指導くださいました筑波大学計算科学研究センター 塩川浩昭助教，北海道大学産学・地域協働推進機構 岩村相哲産学協働マネージャー，日本電信電話株式会社サービスイノベーション総合研究所 藤原靖宏特別研究員，山室健研究主任に深謝いたします．研究環境の確保にご配慮，ご協力いただき，また学術と産業の両観点からご助言をくださいました日本電信電話株式会社サービスイノベーション総合研究所 木原誠司主席研究員，飯田恭弘主任研究員，内山寛之主任研究員，及び同総合研究所ソフトウェアイノベーションセンタ分散処理基盤技術プロジェクト分散システムアーキテクチャ基盤グループの諸氏に御礼申し上げます．最後に，私のこれまでの人生，そして研究生活を送る上で暖かい支援と理解をいただきました家族や友人，特に妻の真理に心から感謝いたします．

参考文献

[1] Charu C. Aggarwal. On k-Anonymity and the Curse of Dimensionality. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pp. 901–909, 2005.

[2] Kadir Akbudak, Enver Kayaaslan, and Cevdet Aykanat. Hypergraph Partitioning Based Models and Methods for Exploiting Cache Locality in Sparse Matrix-Vector Multiplication. *SIAM Journal on Scientific Computing*, Vol. 35, No. 3, pp. C237–C262, 2013.

[3] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium*, pp. 22–31, 2016.

[4] Lars Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms. In Selim G. Akl, Frank Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures*, Vol. 955, pp. 334–345. 1995.

[5] David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for Graph Clustering and Partitioning. In Reda Alhajj and Jon Rokne, editors, *Encyclopedia of Social Network Analysis and Mining*, pp. 73–82. 2014.

[6] Roberto J. Bayardo and Rakesh Agrawal. Data Privacy through Optimal k-Anonymization. In *Proceedings of the 21st IEEE International*

Conference on Data Engineering, pp. 217–228, 2005.

[7] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data*, Vol. 1, pp. 1199–1214, 2016.

[8] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th International Conference on World Wide Web*, pp. 587–596, 2011.

[9] Paolo Boldi and Sebastiano Vigna. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web*, pp. 595–602, 2004.

[10] Vincenzo Bonnici, Alfredo Ferro, Rosalba Giugno, Alfredo Pulvirenti, and Dennis Shasha. Enhancing Graph Database Indexing by Suffix Tree Structure. In *Proceedings of the 5th IAPR International Conference on Pattern Recognition in Bioinformatics*, pp. 195–203, 2010.

[11] Andrei Z. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, pp. 21–29, 1997.

[12] Ji-Won Byun, Ashish Kamra, Elisa Bertino, and Ninghui Li. Efficient k-Anonymization Using Clustering Techniques. In Ramamohanarao Kotagiri, P. Radha Krishna, Mukesh Mohania, and Ekawit Nantajeewarawat, editors, *Advances in Databases: Concepts, Systems and Applications SE - 18*, Vol. 4443, pp. 188–200. 2007.

[13] Umit V. Catalyurek, Mehmet Deveci, Kamer Kaya, and Bora Ucar. Multithreaded Clustering for Multi-level Hypergraph Partitioning. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, pp. 848–859, 2012.

[14] Chin-Chen Chang, Yu-Chiang Li, and Wen-Hung Huang. TFRP: An

Efficient Microaggregation Algorithm for Statistical Disclosure Control. *Journal of Systems and Software*, Vol. 80, No. 11, pp. 1866–1878, 2007.

[15] Xiaoyang Chen, Hongwei Huo, Jun Huan, and Jeffrey Scott Vitter. Efficient Graph Similarity Search in External Memory. *IEEE Access*, Vol. 5, pp. 4551–4560, 2017.

[16] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: Towards Verification-free Query Processing on Graph Databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pp. 857–872, 2007.

[17] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On Compressing Social Networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 219–228, 2009.

[18] Avery Ching. Giraph: Large-scale graph processing infrastructure on Hadoop. In *Hadoop Summit*, 2011.

[19] Aaron Clauset, Cristopher Moore, and Mark E. J. Newman. Structural Inference of Hierarchies in Networks. In Edoardo Airoldi, David M. Blei, Stephen E. Fienberg, Anna Goldenberg, Eric P. Xing, and Alice X. Zheng, editors, *Proceedings of the Statistical Network Analysis: Models, Issues, and New Directions*, pp. 1–13, 2007.

[20] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 26, No. 10, pp. 1367–1372, 2004.

[21] Lawrence Cox, Bruce Johnson, Sarah-Kathryn McDonald, Dawn Nelson, and Violeta Vazquez. Confidentiality Issues at the Census Bureau. In *Proceedings of the First Annaul Census Bureau Research Conference*, pp. 199–218, 1985.

[22] Elizabeth H. Cuthill and Jon McKee. Reducing the Bandwidth of Sparse Symmetric Matrices. In *Proceedings of the 1969 24th National Conference*, pp. 157–172, 1969.

[23] Tore Dalenius. Finding a needle in a haystack or identifying anonymous census records. *Journal of Official Statistics*, Vol. 2, No. 3, pp. 329–336, 1986.

[24] Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. A Generic Approach to Bulk Loading Multidimensional Index Structures. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pp. 406–415, 1997.

[25] Dua Dheeru and Efi Karra Taniskidou. UCI Machine Learning Repository, 2017.

[26] Josep Domingo-Ferrer and Josep M. Mateo-Sanz. Practical Data-Oriented Microaggregation for Statistical Disclosure Control. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 14, No. 1, pp. 189–201, 2002.

[27] Josep Domingo-Ferrer and Vicenç Torra. Ordinal, Continuous and Heterogeneous k-Anonymity Through Microaggregation. *Data Mining and Knowledge Discovery*, Vol. 11, No. 2, pp. 195–212, 2005.

[28] Wei Dong, Charikar Moses, and Kai Li. Efficient K-nearest Neighbor Graph Construction for Generic Similarity Measures. In *Proceedings of the 20th International Conference on World Wide Web*, pp. 577–586, 2011.

[29] Khaled El Emam, Fida Kamal Dankar, Romeo Issa, Elizabeth Jonker, Daniel Amyot, Elise Cogo, Jean-Pierre Corriveau, Mark Walker, Sadrul Chowdhury, Regis Vaillancourt, Tyson Roffey, and Jim Bottomley. A Globally Optimal k-Anonymity Method for the De-Identification of Health Data. *Journal of the American Medical Informatics Association*, Vol. 16, No. 5, pp. 670–82, 2009.

[30] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On Power-law Relationships of the Internet Topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 251–262, 1999.

[31] Yuan Fang, Wenqing Lin, Vincent W. Zheng, Min Wu, Kevin Chen-Chuan Chang, and Xiao-Li Li. Semantic Proximity Search on Graphs with Metagraph-based Learning. In *Proceedings of the 32nd IEEE International Conference on Data Engineering*, pp. 277–288, 2016.

[32] Michael Frasca, Kamesh Madduri, and Padma Raghavan. NUMA-aware Graph Mining Techniques for Performance and Energy Efficiency. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2012.

[33] Jerome H. Freidman, Jon Louis Bentley, and Raphael Ari Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, Vol. 3, No. 3, pp. 209–226, 1977.

[34] Benjamin C. M. Fung, Ke Wang, Rui Chen, and Philip S. Yu. Privacy-preserving Data Publishing: A Survey of Recent Developments. *ACM Computing Surveys*, Vol. 42, No. 4, pp. 14:1—-14:53, 2010.

[35] Pin Gao, Mingxing Zhang, Kang Chen, Yongwei Wu, and Weimin Zheng. High Performance Graph Processing with Locality Oriented Design. *IEEE Transactions on Computers*, Vol. 66, No. 7, pp. 1261–1267, 2017.

[36] J. Alan George. *Computer Implementation of the Finite Element Method*. PhD thesis, 1971.

[37] J. Alan George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, Vol. 10, No. 2, pp. 345–363, 1973.

[38] Rosalba Giugno, Vincenzo Bonnici, Nicola Bombieri, Alfredo Pul-

virenti, Alfredo Ferro, and Dennis Shasha. GRAPES: A Software for Parallel Searching on Biological Graphs Targeting Multi-Core Architectures. *PLoS One*, Vol. 8, No. 10, 2013.

[39] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pp. 599–613, 2014.

[40] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. 1st edition, 2013.

[41] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, Vol. 14, p. 47, 1984.

[42] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 77–85, 2013.

[43] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp. 405–418, 2008.

[44] Xianmang He, HuaHui Chen, Yefang Chen, Yihong Dong, Peng Wang, and Zhenhua Huang. Clustering-Based k-Anonymity. In Pang-Ning Tan, Sanjay Chawla, ChinKuan Ho, and James Bailey, editors, *Advances in Knowledge Discovery and Data Mining SE - 34*, Vol. 7301, pp. 405–417. 2012.

[45] Qingbo Hu, Sihong Xie, Jiawei Zhang, Qiang Zhu, Songtao Guo, and Philip S. Yu. HeteroSales: Utilizing Heterogeneous Social Networks

to Identify the Next Enterprise Customer. In *Proceedings of the 25th International Conference on World Wide Web*, pp. 41–50, 2016.

[46] Kuan Lun Huang, Salil S. Kanhere, and Wen Hu. Towards Privacy-Sensitive Participatory Sensing. In *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications*, pp. 1–6, 2009.

[47] Anco Hundepool, Aad van de Wetering, Ramya Ramaswamy, Luisa Franconi, Silvia Polettini, Alessandra Capobianchi, Peter-Paul de Wolf, Josep Domingo, Vicenc Torra, Ruth Brand, and Sarah Giessing. Âţ-ARGUS version 4.2 User's Manual, 2008.

[48] Tochukwu Iwuchukwu and Jeffrey F. Naughton. K-anonymization As Spatial Indexing: Toward Scalable and Incremental Anonymization. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pp. 746–757, 2007.

[49] Vijay S. Iyengar. Transforming Data to Satisfy Privacy Constraints. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 279–288, 2002.

[50] Han Jian-min, Cen Ting-ting, and Yu Hui-qun. An Improved V-MDAV Algorithm for l-Diversity. In *Proceedings of the 2008 International Symposiums on Information Processing*, pp. 733–739, 2008.

[51] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the Ninth IEEE International Conference on Data Mining*, pp. 229–238, 2009.

[52] Konstantinos I. Karantasis, Andrew Lenharth, Donald Nguyen, María J. Garzarán, and Keshav Pingali. Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 921–932, 2014.

[53] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms. In *Proceedings of 20th International Conference on Extending Database Technology*, pp. 25–36, 2017.

[54] Sarat kr. Chettri and Bhogeswar Borah. MDAV2K: a Variable-Size Microaggregation Technique for Privacy Preservation. In *Proceedings of the International Conference on Information Technology Convergence and Services*, pp. 105–118, 2012.

[55] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web*, pp. 591–600, 2010.

[56] Amy N. Langville and Carl D. Meyer. A Reordering for the PageRank Problem. *SIAM Journal on Scientific Computing*, Vol. 27, No. 6, pp. 2112–2120, 2006.

[57] Dominique LaSalle and George Karypis. Multi-threaded Graph Partitioning. In *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing*, pp. 225–236, 2013.

[58] Michael Laszlo and Sumitra Mukherjee. Minimum Spanning Tree Partitioning Algorithm for Microaggregation. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, No. 7, pp. 902–911, 2005.

[59] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proceedings of the VLDB Endowment*, Vol. 6, No. 2, pp. 133–144, 2012.

[60] Kristen LeFevre, David J. DeWitt, and Raghu Ramakrishnan. Incognito: Efficient Full-Domain K-Anonymity. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 49–60, 2005.

[61] Kristen LeFevre, David J. DeWitt, and Raghu Ramakrishnan. Mon-

drian Multidimensional K-Anonymity. In *Proceedings of the 22nd International Conference on Data Engineering*, pp. 25–25, 2006.

[62] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel Graph Analytics. *Communications of the ACM*, Vol. 59, No. 5, pp. 78–87, 2016.

[63] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, Vol. 6, No. 1, pp. 29–123, 2009.

[64] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-Closeness: Privacy Beyond k-Anonymity and l-Diversity. In *Proceedings of the 23rd IEEE International Conference on Data Engineering*, pp. 106–115, 2007.

[65] Zhongmou Li, Hui Xiong, Yanchi Liu, and Aoying Zhou. Detecting Blackhole and Volcano Patterns in Directed Networks. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, pp. 294–303, 2010.

[66] Yongsub Lim, U Kang, and Christos Faloutsos. SlashBurn: Graph Compression and Mining beyond Caveman Communities. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 26, No. 12, pp. 3077–3089, 2014.

[67] Jun-Lin Lin and Meng-Cheng Wei. An Efficient Clustering Method for k-Anonymization. In *Proceedings of the 2008 International Workshop on Privacy and Anonymity in Information Society*, pp. 46–50, 2008.

[68] Jun-Lin Lin, Tsung-Hsien Wen, Jui-Chien Hsieh, and Pei-Chann Chang. Density-based microaggregation for statistical disclosure control. *Expert Systems with Applications*, Vol. 37, No. 4, pp. 3256–3263, 2010.

[69] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo

Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, Vol. 5, No. 8, pp. 716–727, 2012.

[70] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, Vol. 17, No. 01, pp. 5–20, 2007.

[71] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pp. 527–543, 2017.

[72] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at what COST? In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems*, 2015.

[73] Mark E. J. Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review E*, Vol. 69, No. 2, p. 026113, 2004.

[74] Anna Oganian and Josep Domingo-Ferrer. On the Complexity of Optimal Microaggregation for Statistical Disclosure Control. *Statistical Journal of the United Nations Economic Comission for Europe*, Vol. 18, No. 4, pp. 345–353, 2001.

[75] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. 1999.

[76] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast Malware Classification by Automated Behavioral Graph Matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, pp. 45:1—-45:4, 2010.

[77] Juan C. Pichel, David E. Singh, and Jesús Carretero. Reordering Algorithms for Increasing Locality on Multicore Processors. In *Proceedings of the 2008 10th IEEE International Conference on High Performance*

Computing and Communications, pp. 123–130, 2008.

[78] Ali Pinar and Michael T. Heath. Improving Performance of Sparse Matrix-vector Multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[79] Arnau Prat-Pérez, David Dominguez-Sal, and Josep L. Larriba-Pey. Social Based Layouts for the Increase of Locality in Graph Operations. In *Proceedings of the 16th International Conference on Database Systems for Advanced Applications - Volume Part I*, pp. 558–569, 2011.

[80] Miao Qiao, Hao Zhang, and Hong Cheng. Subgraph Matching: On Compression and Computation. *Proceedings of the VLDB Endowment*, Vol. 11, No. 2, pp. 176–188, 2017.

[81] Xuguang Ren and Junhu Wang. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. *Proceedings of the VLDB Endowment*, Vol. 8, No. 5, pp. 617–628, 2015.

[82] Jeff Sedayao. Enhancing Cloud Security Using Data Anonymization. *Intel White Paper*, pp. 1–8, 2012.

[83] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proceedings of the VLDB Endowment*, Vol. 1, No. 1, pp. 364–375, 2008.

[84] Hiroaki Shiokawa. *Efficient Clustering Algorithms for Large-scale Graphs*. PhD thesis, University of Tsukuba, 2015.

[85] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. Fast Algorithm for Modularity-Based Graph Clustering. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, pp. 1170–1176, 2013.

[86] Agusti Solanas and Antoni Martinez-Balleste. V-MDAV: A Multivariate Microaggregation With Variable Group Size. In *Proceedings of the 17th COMPSTAT Symposium of the IASC*, 2006.

[87] Latanya Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, Vol. 10, No. 5, pp. 571–588, 2002.

[88] Latanya Sweeney. k-anonymity: a model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, Vol. 10, No. 5, pp. 557–570, 2002.

[89] Julian R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, Vol. 23, No. 1, pp. 31–42, 1976.

[90] Hannes Voigt. Declarative Multidimensional Graph Queries. In Patrick Marcel and Esteban Zimányi, editors, *Business Intelligence: 6th European Summer School, eBISS 2016, Tours, France, July 3-8, 2016, Tutorial Lectures*, pp. 1–37. 2017.

[91] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, Vol. 393, p. 440, 1998.

[92] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pp. 1813–1828, 2016.

[93] Leon Willenborg and Ton de Waal. *Elements of Statistical Disclosure Control*. 2000.

[94] Michael K. Wolf. Microaggregation and Disclosure Avoidance for Economic Establishment Data. In *Annual Meeting of the American Statistical Association*, 1988.

[95] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware Graph-structured Analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 183–193, 2015.

[96] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making Caches Work for Graph Analytics.

In *Proceedings of 2017 IEEE International Conference on Big Data*, pp. 293–302, 2017.

[97] Peixiang Zhao and Jiawei Han. On Graph Query Optimization in Large Networks. *Proceedings of the VLDB Endowment*, Vol. 3, No. 1-2, pp. 340–351, 2010.

[98] 新井淳也, 塩川浩昭, 山室健, 鬼塚真, 岩村相哲. 効率的なグラフ分析のための実行時並列リオーダリング. 電子情報通信学会和文論文誌 D, Vol. J102-D, No. 4, 2019.

[99] 新井淳也, 鬼塚真, 塩川浩昭. クラスタリングと空間分割の併用による効率的な k-匿名化. 日本データベース学会和文論文誌, Vol. 13-J, No. 1, pp. 72–77, 2014.

[100] 総務省. 平成 29 年版 情報通信白書. 2017.

[101] 総務省. 平成 30 年版 情報通信白書. 2018.