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

新井　淳也†　藤原　靖宏†　鬼塚　真††　岩村　相哲†

† 日本電信電話株式会社 ソフトウェアイノベーションセンタ
†† 大阪大学 大学院情報科学研究科
E-mail: †{arai.junya,fujiwara.yasuhiro,iwamura.sotetsu}@lab.ntt.co.jp, ††onizuka@ist.osaka-u.ac.jp

あらまし　データグラフ G とクエリグラフ Q が与えられたとき，Q と同型なサブグラフを G から発見する問題はサブグラフマッチングと呼ばれる．ラベル付きグラフ上のサブグラフマッチングを高速化するため，これまで頂点間のマッチングを取る際の探索順序を最適化する手法についてよく研究されてきた．しかし既存手法は実世界のグラフのように冪乗則に従った次数分布を持つグラフにおいて多数の頂点を訪問するため計算コストが高いという問題がある．そこで本論文では高次数頂点の訪問を避ける動的なマッチング順序と，データグラフ上の局所的な隣接関係に基づく枝刈りを組み合わせたサブグラフマッチングアルゴリズムを提案する．

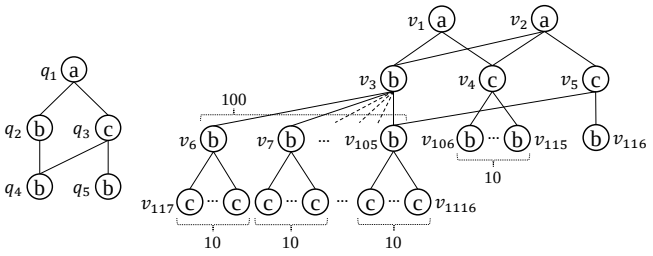
キーワード　グラフアルゴリズム，サブグラフマッチング，ラベル付きグラフ

1. はじめに

多様な事物間の関係性を表現できるグラフデータは，近年ますます多くの分野で分析の対象として利用されるようになっている．グラフ分析において最も頻繁に用いられる処理の一つとして，特定の構造を持つ部分グラフの発見，すなわちサブグラフマッチングがある．より厳密には，サブグラフマッチングは大きいグラフ (データグラフ) G と小さいグラフ (クエリグラフ) Q を受け取り，Q と同型な G のサブグラフへの埋め込みを発見する問題である．この問題は様々な場面で利用されている．Fang ら [4] は人物，学校，専門分野，住所など複数種類の事物を頂点として持つグラフ上のサブグラフマッチングによって，人物間の関係に意味付けを行った．例えば同じ学校と専門分野に接続されている 2 人の人物はクラスメイトであると考えることができる．同様に Hu ら [8] は企業，従業員，役職などを含むグラフをサブグラフマッチングによって企業のみから成るグラフに変換し，営業活動における顧客推薦に利用した．また Park ら [10] はプログラムの振る舞いをグラフとして表現し，新種あるいは自己変異型マルウェアの検出にサブグラフマッチングを使用した．

以上のようにラベル付きグラフ上のサブグラフマッチングの応用は広いが，NP 困難な問題であるため計算に時間を要する [6]．さらに 1 回のグラフ分析の中で繰り返しサブグラフマッチングを用いる場合もある [8,18] ことから，サブグラフマッチングの計算時間は大きな問題となる．

サブグラフマッチングの計算時間を短縮するために重要なのはマッチング順序の最適化である．マッチング順序とは埋め込みとなるクエリ頂点とデータ頂点のペアを探す順番に関する実行プランである．どのようなマッチング順序を取るかによって，一部の頂点ペアについてのみ対応関係が記録された中間結果，すなわち部分的な埋め込みの生成数は大きく変化する．例えば図 1(a) のクエリグラフと図 1(b) のデータグラフのマッチングにおいて，$(q_2, q_3, \ldots)$ というマッチング順序を取ったとする．



(a) クエリグラフ　　(b) データグラフ

図 1　クエリグラフ Q とデータグラフ G の例．頂点内のアルファベットはラベルを，q_i, v_i は頂点の ID を示す．

q_2 はラベル b なので，同じくラベル b を持つ 112 個のデータ頂点 $v_3, v_6, v_7, \ldots, v_{116}$ にマッチし得る．同様に q_3 はラベル c を持つ 1,002 個の頂点 $v_4, v_5, v_{117}, v_{118}, \ldots, v_{1116}$ にマッチし得る．従って最初の 2 頂点 q_2, q_3 への部分的なマッチングだけで $112 \times 1{,}002 = 112{,}224$ 通りもの部分的な埋め込みが生じる．以降の探索ではこれらの部分埋め込みに残りのクエリ頂点とマッチするデータ頂点を追加していくことになるため，計算コストが高い．一方で $(q_1, q_2, \ldots)$ という順序では，最初の 2 頂点が隣接するラベル a と b の頂点である．これを満たすデータ頂点の組合せは $(v_1, v_3), (v_2, v_3)$ の 2 通りのみであるため，部分埋め込みの数は 112,224 通りと比べ大幅に減少する．

このようなマッチング順序の最適化に関して近年 TurboISO [6], BoostISO [11], CFLMatch [1] などでよく研究されてきた．これらの手法はマッチング先のデータ頂点から幅優先探索 (BFS) を行いデータグラフの要約構造を作成し，各クエリ頂点にマッチし得るデータ頂点の数を見積もる．そしてその情報を用いてマッチング順序の最適化を行う．

しかしながら実世界のグラフにおける BFS は計算コストが高いという問題点がある．なぜならば実世界のグラフは一般に冪乗則に従う次数分布，すなわちごく少数の頂点が極端に大きい次数を持つような構造を持っており [13]，BFS で高次数頂点を

訪問すると大量の隣接頂点へのアクセスが発生するためである．例えば図 1(b) のデータグラフにおいては頂点 v_3 が極端に高い次数 (102) を持つ．そのため，仮に v_1 からの BFS で v_3 に訪問したとすると，その先にある 100 個の隣接頂点 $v_6, v_7, \ldots, v_{105}$ へのアクセスが発生する．$v_6, v_7, \ldots, v_{105}$ はそれぞれ次数 11 または 12 であり極端に高くはないが，それらの下には 1000 個の隣接頂点 $v_{117}, v_{118}, \ldots, v_{1116}$ が存在し，これら大量のエッジの走査には時間を要する．このような BFS による要約グラフの作成はクエリグラフのマッチング先領域をデータグラフ上で変更するたびに発生し，そのことが実世界のグラフにおける既存手法の性能を低下させると考えられる．

本論文ではこのようなマッチング順序に関する問題点を解決したアルゴリズムを提案する．提案手法は 2 つのステップから成る．ますステップ 1 はマッチング候補頂点の枝刈りである．ここでマッチング候補頂点とは各データ頂点がマッチし得るクエリ頂点の集合である．既存手法はデータグラフの構造を分析するために BFS によるグラフ要約を行うのに対し，提案手法ではデータグラフ上で隣接頂点の情報を利用した繰り返し処理を行うことにより探索空間の削減とマッチング順序最適化のための情報収集を行う．次のステップ 2 では動的にマッチング順序を最適化しながら埋め込みの探索を行う．提案手法はマッチし得るデータ頂点の数が最も少ないクエリ頂点から順にマッチングを行う．そのため，高次数頂点に隣接する大量の頂点への訪問を避けることができる．さらにクエリグラフにおいて閉路を含む部分から優先的にマッチングを行うことで，隣接頂点集合の交差を用いたマッチング候補頂点の絞り込みを行う．ステップ 1, 2 とも容易に並列化が可能であり，マルチコアまたはマルチソケットのマシンを効率的に利用することができる．評価では提案手法の逐次処理版プロトタイプ実装を用いて実験を行い，現状でも手法提案者などによる既存手法の実装と比肩する性能でサブグラフマッチングが可能であることを確認した．

本論文の構成は次のとおりである．第 2 節において必要な背景知識を説明する．第 3 節で提案手法の詳細を述べ，その評価結果を第 4 節で示す．第 5 節で関連研究を紹介し，最後に第 6 節で結論を述べる．

2. 背景知識

本節ではサブグラフマッチング問題を定義したのち，提案手法の説明に必要となる既存技術について述べる．

2.1 問題定義

本論文ではデータグラフとして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，$E_G \subseteqq V_G \times V_G$ 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させる関数である．同様にクエリグラフ $Q = (V_Q, E_Q, \Sigma, \ell)$ を考える．V_Q と E_Q はそれぞれ頂点とエッジの集合である．ここで次のようなとき Q は G に対してサブグラフ同型であるという: 単射 $M : V_Q \to V_G$ が存在し，$\forall q \in V_Q, \ell(q) = \ell(M[q])$ かつ $\forall (q, q') \in E_Q, (M[q], M[q']) \in E_G$．このような単射 M は埋め込みとも呼ばれる．サブグラフマッチングとは，データグラフ G とクエリグラフ Q が与えられたとき，上記の条件を満たす全ての埋め込み M を発見する問題と定義する．

表 1　本論文で用いる表記

表記	定義
Q, G	データグラフとクエリグラフ
V_Q, V_G	頂点集合; $V_Q = \{q_1, q_2, \ldots\}, V_G = \{v_1, v_2, \ldots\}$
E_Q, E_G	エッジ集合; $E_Q \subseteqq V_Q \times V_Q, E_G \subseteqq V_G \times V_G$
Σ	ラベル集合
$\ell(v)$	頂点 v のラベル
$N(v)$	頂点 v の隣接頂点集合
$N_\sigma(v)$	頂点 v の隣接頂点のうちラベル σ を持つものの集合
M	写像 (埋め込み); $(x, y) \in M$ を $M[x] = y$ とも書く
$\mathrm{dom}(M)$	M の定義域; $\mathrm{dom}(M) = \{ x \mid (x, y) \in M \}$
$\mathrm{ran}(M)$	M の値域; $\mathrm{ran}(M) = \{ y \mid (x, y) \in M \}$
$C[v]$	$v \in V_G$ とマッチし得るクエリ頂点の集合
$C^{-1}[q]$	$q \in V_Q$ とマッチし得るデータ頂点の集合

なお，簡単のため以降は頂点がラベルを持つ無向グラフについてのみ説明するが，エッジがラベルを持つグラフや有向グラフにも容易に拡張することができる．

2.2 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のアルゴリズムはバックトラッキングに基づくものが一般的である．このアプローチではクエリグラフに含まれる頂点に対してマッチング順序を適当に定め，その順でしらみつぶしにマッチする頂点を探索する．マッチする頂点が見つからない場合は，一旦マッチング順序を手前に戻り，別の頂点にマッチさせて再度探索を行う．例として図 1 のクエリグラフとデータグラフを用いてこの処理の流れを説明する．クエリ頂点のマッチング順序はランダムに $(q_4, q_5, q_3, q_1, q_2)$ を選んだとする．最初のクエリ頂点 q_4 はラベル b なので，112 個のデータ頂点 $v_3, v_6, v_7, \ldots, v_{116}$ がマッチング候補となる．ここではまず q_4 に v_3 をマッチさせたとして進める．このことを $M \leftarrow M \cup \{(q_4, v_3)\}$ または $M[q_4] \leftarrow v_3$ と書くことにする．2 番目にマッチを取る q_5 もラベル b であるが，v_3 は使用済みなので，$M[q_5] \leftarrow v_6$ を選んだとして次の q_3 へ進む．q_3 はラベル c を持ち q_1 と q_4 に隣接しているので，q_3 の候補もマッチング済み頂点 $M[q_4]$ $(= v_3)$ と隣接していなければならない．しかしそのような頂点は存在しないので，マッチング順序が 1 つ手前の q_5 のマッチングを行う段階まで戻り，$M[q_5] \leftarrow v_6, v_7, \ldots$ v_{116} とマッチングを変更しながら再度探索する．それでも見つからないためさらにマッチング順序が手前の q_4 まで戻り，同様に探索する．最終的に $M = \{(q_1, v_2), (q_2, v_3), (q_3, v_5), (q_4, v_{105}), (q_5, v_{116})\}$ という唯一の埋め込みを発見し探索を終了する．

バックトラッキングは最新の手法も含め多くの手法の基礎となっている [1]．このような探索に要する時間はクエリ頂点のマッチング順序によって大幅に変化するため，順序を最適化する研究が行われてきた．また不要な探索を枝刈りするためのマッチング候補頂点の枝刈り手法を併用することでさらに探索空間を削減することが可能である [7, 19]．

3. 提案手法

サブグラフマッチングはバックトラッキングで大量の部分埋め込みを生成しながらマッチングを探索するため，計算コストが高い．そこで提案手法では (i) 候補頂点の枝刈りと (ii) 動的なマッチング順序による埋め込み探索という 2 つのステップによ

Algorithm 1 候補頂点枝刈り

入力: データグラフ G，クエリグラフ Q
出力: マッチング候補頂点集合 C

```
1  for v ∈ V_G do
2    C[v] ← { q ∈ V_Q | IsCandidate(v, q) }
3  repeat
4    for v ∈ V_G do
5      C[v] ← { q ∈ C[v] | HasCandidates(v, q, C) }
6      for v' ∈ N(v) do
7        if C[v'] ∩ N(C[v]) = ∅ then
8          エッジ (v, v') を削除
9  until C が変更されていない
```

り部分埋め込みの生成数を削減する．候補頂点の枝刈りによって，データグラフにおいて隣接頂点がどのクエリ頂点にマッチし得るかを知ることができる．動的なマッチング順序の生成ではその情報を用いることにより，マッチし得るデータ頂点が最も少ないクエリ頂点から順にマッチングを行っていく．本節ではこれらの 2 ステップの詳細についてそれぞれ説明する．

3.1 データグラフ上の候補頂点枝刈り

最初のステップである候補頂点枝刈りでは，各データ頂点 v について，v がマッチし得るクエリ頂点の集合 (candidates) $C[v]$ を求める．このステップの目的は，各データ頂点の隣接頂点から得られる情報を用いて $C[v]$ からなるべく多くのクエリ頂点を除外することである．アルゴリズム 1 に候補頂点枝刈り処理の全体像を示す．このアルゴリズムは隣接頂点のラベルを用いた局所的な枝刈り (1, 2 行目) と，繰り返し処理による枝刈り情報の伝播 (3–9 行目) という 2 つの処理に分けられる．以降でこれら 2 つの処理について詳しく説明する．

3.1.1 隣接頂点のラベルを用いた局所的な枝刈り

局所的な枝刈りではデータ頂点 v とクエリ頂点 q のラベルに加え，特定のラベルを持つ隣接頂点の出現回数に着目してマッチする可能性を判定する．すなわち，あるラベル σ について，ラベル σ を持つ q の隣接頂点の数がラベル σ を持つ v の隣接頂点の数よりも大きい場合，q と v はマッチし得ないことを利用する．この条件では次数が高いクエリ頂点ほど C から除外されやすい．v が q にマッチし得ることを IsCandidate(v, q) と表すと，次のように条件を定義できる：

$$\begin{aligned}&\textsc{IsCandidate}(v, q) = \\ &\quad \ell(v) = \ell(q) \wedge \forall \sigma \in \ell(N(q)), |N_\sigma(q)| \leqq |N_\sigma(v)| \,.\end{aligned} \tag{1}$$

ただし $\ell(N(q)) = \{\, \ell(q') \mid q' \in N(q) \,\}$，$N_\sigma(q)$ は q の隣接頂点のうちラベル σ を持つものの集合 $\{\, q' \in N(q) \mid \ell(q') = \sigma \,\}$ とする．$N_\sigma(v)$ も同様である．例えば図 1 においてクエリ頂点 q_4 はラベル b を持ち，ラベル b と c の頂点に隣接している．従ってラベル b を持ちながらもラベル b の隣接頂点を持たないデータ頂点 v_{106} は q_4 にマッチし得ない，すなわち $q_4 \notin C[v_{106}]$ であることがわかる．同様に $v_{107}, \ldots, v_{116}$ も q_4 とはマッチし得ない．隣接頂点の情報を用いた枝刈りでは，このように式 1 を用いることでマッチし得ないクエリ頂点とデータ頂点の組合せを排除した C を生成する．

3.1.2 枝刈り情報の伝播

次の枝刈り情報の伝播では，局所的な枝刈りの結果を利用しさらに多くのクエリ頂点を候補から除外するとともに，データグラフから不要なエッジを削除することでグラフの規模を縮小する．この処理では，データ頂点 v とクエリ頂点 q がマッチし得るならば，v の隣接頂点は q の隣接頂点にマッチし得るという性質を利用する．この判定を行う関数 HasCandidates の定義は後で議論する．例えば図 1 において，IsCandidate(v_4, q_3) が成立する．しかし，q_3 は q_4 と隣接しているにもかかわらず，前段の枝刈りの結果 $v_{106}, \ldots, v_{115}$ は q_4 にマッチし得ないことが分かっているので，v_4 は q_4 とマッチし得る頂点と隣接していない．従って v_4 と q_3 にマッチし得ないことが分かる．このように枝刈りの情報を伝播させることでより多くのクエリ頂点を C から除外することができる．

さらにこのように C に含まれる頂点を減少させていくと，マッチング結果に影響しない不要なエッジが見つかる．データグラフのエッジ (v, v') を削除してよい条件は，クエリグラフにおいて $C[v]$ と $C[v']$ の間にエッジがないこと，すなわち $C[v'] \cap N(C[v]) = \emptyset$ である．例えばラベル a と c の頂点を接続するエッジはクエリグラフでは (q_1, q_3)，データグラフでは (v_1, v_4), (v_2, v_4), (v_2, v_5) が存在する．ここで v_4 は q_3 にマッチしないことが分かっているので，エッジ (v_1, v_4), (v_2, v_4) を含む埋め込みは存在せず，削除してもよいことが分かる．エッジを削除することでグラフの規模が縮小され，探索に要する時間が減少する．

以上のような枝刈りの効果を高めるためには適切に HasCandidates 関数を定義する必要がある．求められる性質は，「データ頂点 v の隣接頂点がクエリ頂点 q の隣接頂点にマッチし得る」ことを偽陰性なく判定すること，偽陽性が少ないこと，計算が軽量であることの 3 点である．考えられる条件式の中で最低限満たすべきものとしては，v の隣接頂点がマッチし得るクエリ頂点集合の和集合に q の隣接頂点が含まれる，すなわち

$$N(q) \subseteqq C[N(v)] \tag{2}$$

という条件がある．ただし $C[N(v)] = \bigcup_{v' \in N(v)} C[v']$ とする．$N(q)$ が大きければ大きいほど，つまり次数が高いクエリ頂点ほど，この条件を満たすデータ頂点は少なくなる．しかしながら式 2 の定義では，$|N(q)| \geqq 2$ の場合に 1 つのデータ頂点と 2 つ以上のクエリ頂点をマッチさせてしまうことによる偽陽性を生じる可能性がある．例えば v の隣接頂点 v' について $N(q) \subseteqq C[v']$ を満たすものが 1 つ存在し，その他の v の隣接頂点は $C[v'] = \emptyset$ であったとする．このとき式 2 は成立するが，同じ頂点は 1 度しか埋め込みに使用できないため，これは偽陽性である．$N(q)$ に含まれるクエリ頂点をそれぞれ異なるデータ頂点にマッチさせる組合せ (単射) が存在することを確認すれば偽陽性を排除できるが，この処理は計算量が大きい．He ら [7] の方法を用いた場合，繰り返し処理 1 イテレーションあたりの最悪計算量は，クエリ頂点とデータ頂点の最大次数をそれぞれ d_Q^{max}, d_G^{max} とすると $O(|V_Q||V_G|(d_Q^{max} + d_G^{max})^{2.5})$ となる．最大次数はグラフの規模が増大するほど大きくなると考えられるため，この計算量は問題がある．

そこで偽陽性の生成数と計算量の間でバランスを取り，提案手法ではクエリ頂点の隣接頂点のうち同じラベルを持つものが

$$\textsc{HasCandidates}(v,q,C) = \begin{cases} N(q) \subseteqq C[N(v)] \wedge (\forall \sigma \in \ell(N(q)), |N_\sigma(q)| \leqq |N_\sigma(v)|) & \text{if } |N_\sigma(q)| = 1 \\ N(q) \subseteqq C[N(v)] \wedge \left(\forall \sigma \in \ell(N(q)), |N_\sigma(q)| \leqq |N_\sigma(v)| \wedge \left|\{\, v' \in N_\sigma(v) \mid q_\alpha \in C[v'] \vee q_\beta \in C[v'] \,\}\right| \geqq 2\right) & \text{if } |N_\sigma(q)| \geqq 2 \end{cases} \tag{3}$$

ただし q_α, q_β は $N_\sigma(q)$ の中で最も次数が高い 2 頂点.

図 2　関数 HasCandidates の定義

2 つ以下の場合に限り偽陽性を生じさせない条件式を用いる．クエリグラフは一般に小さいため，同じラベルを持つ隣接頂点が 2 つ以下になる場合は少なくない．HasCandidates の定義を式 3 に示す．この定義は 2 通りの場合に分かれている．まず $|N_\sigma(q)| = 1$ の場合は式 2 でも偽陽性を生じさせることがない．そのため式 1 に含まれる隣接頂点数の条件と式 2 を組み合わせたものを定義として用いる．隣接頂点数の条件を含めるのは，処理の途中でエッジを削除していくので $|N_\sigma(v)|$ の値が変化するためである．次に $|N_\sigma(q)| \geqq 2$ の場合は，偽陽性を減らすための条件として 2 つのクエリ頂点 $q_\alpha, q_\beta \in N_\sigma(q)$ に対してそれぞれ異なるデータ頂点をマッチさせられることを確認する．式 2 の条件によって q_α と q_β にそれぞれマッチし得るデータ頂点が 1 つ以上存在することは確認できているため，q_α または q_β にマッチし得るデータ頂点が 2 つ以上あれば，q_α, q_β にそれぞれ異なるデータ頂点を割り当てることができる．そのため条件 $|\{\, v' \in N_\sigma(v) \mid q_\alpha \in C[v'] \vee q_\beta \in C[v'] \,\}| \geqq 2$ が追加されている．ここで q_α と q_β は，$N_\sigma(q)$ の中で最も次数が高い 2 頂点を選ぶ．なぜならば式 1, 2 ともに次数が高いクエリ頂点ほど C から除外されやすい条件であり，そのような希少なデータ頂点 2 つ以上に隣接する頂点もまた希少であるためである．

この処理の最悪計算量は次のように考えられる．アルゴリズム 1 の 1, 2 行目では $v \in V_G$ と $q \in V_Q$ で 2 重のループが行われる．ループ内で使用する IsCandidate 関数はラベル $\sigma \in \ell(N(q))$ に対する条件確認を含む．$\sum_{q \in V_Q} |\ell(N(q))| \leqq 2|E_Q|$ なので，1, 2 行目の最悪計算量は $O(|E_Q||V_G|)$ である．同様に 4, 5 行目では $v \in V_G$ と $q \in C[v]$ で 2 重のループが行われる．$|C[v]| \leqq |V_Q|$ なのでループの回数は $O(|V_G||V_Q|)$ である．ループ内で使用する HasCandidates 関数は，ラベル $\sigma \in \ell(N(q))$ についてデータ頂点 $v' \in N_\sigma(v)$ に対するループによって条件確認を行う．$\sum_{\sigma \in \ell(N(q))} |N_\sigma(v)| \leqq |N(v)|$ かつ $\sum_{v \in V_G} |N(v)| \leqq 2|E_G|$ なので，4, 5 行目の最悪計算量は $O(|V_Q||E_G|)$ である．よって全体の計算量は $O(|E_Q||V_G| + |V_Q||E_G|)$ となる．この計算量であれば，クエリグラフまたはデータグラフのどちらか片方のサイズが一定である限り，もう片方が大規模化したとしても線形の計算時間増加で押さえることができる．既存手法の計算量 $O(|V_Q||V_G|(d_Q^{max} + d_G^{max})^{2.5})$ はグラフのサイズに対して線形より大きく計算時間が増加することに比べると，提案手法はより大規模なクエリグラフおよびデータグラフに対応しやすい計算量となっていることが分かる．

3.1.3　並列化の方針

局所的な枝刈りと枝刈り情報の伝播のいずれにおいても，各データ頂点に対して条件式を適用し，その結果によってデータ頂点自身の情報 (マッチング候補集合 C) を書き換える処理は容易に並列化することができる．具体的には 1 行目と 4 行目の for 文で各データ頂点について並列に処理を行わせる．これに対しエッジの削除 (8 行目) はグラフを書き換える処理であるが，エッジ (v, v') に対して削除済みの印をつける処理で代替することによりスレッド間の競合を回避することができる．

提案手法ではこのように軽量な計算で枝刈りの情報を伝播させていくことにより，隣接頂点の情報のみを用いて C からクエリ頂点を除外していく．

3.2　埋め込みの探索

埋め込みの探索では枝刈りの結果を利用しつつ埋め込みを列挙する．その際，提案手法では 2 つのアイデアに基づいてマッチング順序を決定する．本節ではその 2 つのアイデアをはじめに述べ，続いてアルゴリズムの詳細を説明する．

3.2.1　探索高速化のアイデア

1 つ目のアイデアは低次数頂点の隣接頂点から優先してアクセスすることにより部分埋め込みの生成数を削減することである．これは部分埋め込み M について，マッチし得るデータ頂点の数が最も少ないようなクエリ頂点を順次選択する動的なマッチング順序によって行われる．例えば図 1 において，$M = \{(q_1, v_2), (q_2, v_3)\}$ であるとき q_4 にマッチし得るデータ頂点は $v_6, v_7, \ldots, v_{105}$ の 100 個である．従ってもし次に q_4 のマッチングを行ったとすると，部分埋め込みも同じく 100 個 $(M \cup \{(q_4, v_6)\}, M \cup \{(q_4, v_7)\}, \ldots, M \cup \{(q_4, v_{105})\})$ 生成される．v_3 が高次数の頂点であるため，このように大量の部分埋め込みが生じる．一方で q_3 にマッチし得るデータ頂点は v_5 のみである．そのため，q_3 のマッチングを次に行うほうが部分埋め込みの生成数を少なく抑えることができる．このように動的なマッチング順序を使用することで部分埋め込みの生成数を削減することができる．

2 つ目のアイデアはクエリグラフにおいて閉路を含む部分から先にマッチングを行うことである．これによりマッチング候補の絞り込みが可能になり，部分埋め込みの数を削減できる．具体的な説明のため，クエリ頂点 $q_{i_1}, q_{i_2}, \ldots, q_{i_k}$ が環状につながった閉路をこの順序でマッチングする場合を考える．最後に q_{i_k} のマッチングを行う際，閉路における隣接頂点 $q_{i_1}, q_{i_{k-1}}$ はマッチング済みであるため，q_{i_k} にマッチするデータ頂点は $M[q_{i_1}]$ と $M[q_{i_{k-1}}]$ の両方に隣接していなければならないという制約が生じる．すなわち，q_{i_k} にマッチし得るデータ頂点の集合は $M[q_{i_1}]$ と $M[q_{i_{k-1}}]$ の隣接頂点集合の交差 $N(M[q_{i_1}]) \cap N(M[q_{i_{k-1}}])$ と一致する．実世界のグラフは一般に疎であるため，特定の 2 頂点に同時に隣接する頂点はまれである．もしそのようなデータ頂点が存在しなければマッチング不可能ということなので，処理を中断しバックトラッキングを行う．両頂点に隣接するデータ頂点が存在したとしてもその数は $M[q_{i_1}]$ および $M[q_{i_{k-1}}]$ の隣接頂点数以下

Algorithm 2 埋め込みの探索

入力: クエリグラフ Q，データグラフ G，マッチング候補頂点集合 C，実数パラメータ $c \in (0, 1]$
出力: 埋め込みの集合 $\mathcal{M}$

1 $K[v, q'] = \left|N(v) \cap C^{-1}[q']\right|$ **for** $v \in V_G, q' \in V_Q$
2 $T = Q$ の 2-core に含まれる頂点
3 **if** $T \neq \emptyset$ **then**
4 $\quad q_s = \mathrm{argmin}_{q \in T} \left|C^{-1}[q]\right|$
5 **else**
6 $\quad q_s = \mathrm{argmin}_{q \in V_Q} \left|C^{-1}[q]\right|$
7 $\mathcal{M} = \bigcup_{v \in C^{-1}[q_s]} \textsc{Match}(q_s, v, \emptyset, \emptyset)$
8 **function** $\textsc{Match}(q, v, F, M)$
9 $\quad M[q] \leftarrow v$
10 $\quad$ **if** $|M| = |V_Q|$ **then**
11 $\quad\quad$ **return** $\{M\}$
12 $\quad$ **for** $q' \in N(q) \setminus \mathrm{dom}(M)$ **do**
13 $\quad\quad P = N(q') \cap \mathrm{dom}(M)$
14 $\quad\quad F[q'] \leftarrow (c/2)^{|P|-1} \min_{q'' \in P} K[M[q''], q']$
15 $\quad$ **if** $\mathrm{dom}(F) \cap T \neq \emptyset$ **then**
16 $\quad\quad q_* = \mathrm{argmin}_{q' \in \mathrm{dom}(F) \cap T} F[q']$
17 $\quad$ **else**
18 $\quad\quad q_* = \mathrm{argmin}_{q' \in \mathrm{dom}(F)} F[q']$
19 $\quad F$ から q_* を削除
20 $\quad V_* = \bigcap_{q'_* \in N(q_*) \cap \mathrm{dom}(M)} \left(N(M[q'_*]) \cap C^{-1}(q_*)\right) \setminus \mathrm{dom}(M)$
21 $\quad$ **return** $\bigcup_{v_* \in V_*} \textsc{Match}(q_*, v_*, F, M)$

であるため，マッチング候補の数を減らすことができる．例えば図 1 において $M = \{(q_1, v_2), (q_2, v_3), (q_3, v_5)\}$ という部分埋め込みがある状態において，q_4 にマッチする頂点の集合は $N(M[q_2]) \cap N(M[q_3]) = N(v_3) \cap N(v_5) = \{v_{105}\}$ より 1 つまで絞り込むことができる．そのため v_3 と v_5 の隣接頂点を一つずつ選択して部分埋め込みを生成するよりも高速である．

3.2.2 埋め込み探索アルゴリズム

以上のように動的な順序で探索を行うためのアルゴリズムをアルゴリズム 2 に示す．アルゴリズムは 1–6 行目で以降の計算に必要な前準備を行ったあと，7 行目で関数 Match を呼び出す．関数 Match は 1 回の呼び出しごとにクエリ頂点とデータ頂点のペア 1 つを埋め込みに追加し，再起呼び出しを行うことでバックトラッキングに基づくサブグラフマッチングを行う．なおアルゴリズム中で使用している $C^{-1}[q]$ は q にマッチし得るデータ頂点の集合 $\{\, v \in V_G \mid q \in C[v] \,\}$ である．

前準備における重要な処理はマッチング開始点 q_s の選択 (3–6 行目) である．マッチングを開始する頂点の数は少ない方が部分埋め込みの生成数を少なくすることができるため，マッチング候補となるデータ頂点の数が最も少ないクエリ頂点をマッチング開始点として選択する．この際閉路に含まれる頂点から先にマッチングを行うため，クエリグラフの 2-core 構造を利用する．2-core に含まれる頂点は互いに 2 本以上のエッジで接続されているため，その 2 つの隣接頂点がマッチング済みとなれば交差によってマッチング候補を得ることができる．2-core は次数 1 の頂点をクエリグラフから取り除く操作を繰り返すことで得ることができる．その結果を用いてマッチング開始点を 2-core に含まれる頂点から探す．ただし，クエリグラフが 2-core 構造を含まない場合は全てのクエリ頂点から探す．マッチング開始点 q_s を決定したら，q_s と q_s にマッチし得るデータ頂点の全ての組合せについて Match 関数を呼び出す．

Match 関数はクエリ頂点 q，データ頂点 v に加え，次にマッチングを取るクエリ頂点を決定するための優先度付きキュー F と (部分) 埋め込み M を受け取り，埋め込みの集合を返却する．もしマッチング (q, v) を M に追加することでマッチングが完了するならば，その埋め込みを返却する (9–11 行目)．

未マッチングの頂点が残っている場合は，次にマッチングを行うクエリ頂点 q_* を選択する (12–18 行目)．q_* としては現在の部分埋め込み M においてマッチング候補となるデータ頂点が最も少ないものを選びたい．そのために優先度付きキュー F を用いる．F はマッチング済みクエリ頂点の隣接頂点それぞれに対するマッチング候補の数を保持している．12–14 行目は q の隣接頂点を新しく F に追加すると共に，マッチング候補の数を更新する．q によって新しく追加されるクエリ頂点，すなわちマッチング済み隣接頂点が q のみであるようなクエリ頂点 q' の場合，マッチング候補の数は q の隣接頂点のうち q' にマッチし得るものの数である．しかし複数のマッチング済み頂点に隣接し交差によって発見されようとしているクエリ頂点，すなわち $|P| \geqq 2$ であるような q' の場合は見積もりの数を使用する．実際に交差を取ればマッチング候補集合と要素数が分かるが，他のクエリ頂点がマッチング不可能だった場合に交差の結果が使われず，計算が無駄になってしまうためである．q' にマッチし得るデータ頂点の数 $\left|\bigcap_{q'' \in P} \left(N(M[q'']) \cap C^{-1}[q']\right)\right|$ は次の式によって見積もる：

$$\left(\frac{c}{2}\right)^{|P|-1} \min_{q'' \in P} K[M[q''], q']. \tag{4}$$

ただし $c \in (0, 1]$ は実数のパラメタ，$K[v, q']$ は v の隣接頂点のうち q' にマッチし得るものの数である．式の意味について P の要素数を変えながら説明する．まず $P = \{q_i\}$ のとき，式 4 は $K[M[q_i], q']$ となり，見積もりでない実際のマッチング候補数を表す．次に $P = \{q_i, q_j\}$ のとき，式 4 は

$$\left(\frac{c}{2}\right) \min(K[M[q_i], q'], K[M[q_j], q']) \tag{5}$$

となる．これは 2 つの隣接するデータ頂点 $M[q_i]$ と $M[q_j]$ のうち，q' のマッチング候補である隣接頂点が少ない方の要素数に $c/2$ をかけた値となる．$c/2$ は交差を取った後の隣接頂点集合に残る要素の割合と考えることができる．また c はデータグラフ G のクラスタ係数に相当しており，既存の推定手法によって高速に求めることができる [14]．従ってサブグラフマッチングの実施時に自動的に求めることも可能である．P が 3 要素以上の場合も同様に，q' のマッチング候補である隣接頂点が最も少ないものに交差の回数だけ $c/2$ をかけた値となる．

マッチング候補数を更新した後，マッチング候補数が最小であるようなクエリ頂点を発見する (15–18 行目)．この際，2-core に含まれる頂点で未マッチングのものがある場合は優先的に選択する．その後実際に隣接頂点の交差をとり，関数 Match を再帰的に呼び出す．

3.2.3 並列化の方針

埋め込みの探索の並列化は Cilk [2] 等を用いて関数 Match の呼び出しを並列化することにより可能である．しかしながら

計算コストの小さい関数呼び出しを並列化するとオーバーヘッドが大きくなってしまう．そこで未マッチのクエリ頂点数および優先度付きキュー F に含まれる頂点のマッチング候補数から計算コストを見積もり，一定の閾値を超えた場合に並列化する．

以上のような手続きによって，高次数頂点に隣接する大量の候補頂点に対するマッチングを動的なマッチング順序によって避けつつ埋め込みを探索する．

4. 評　　価

本節では提案手法の性能を既存手法と比較するとともに，マッチング順序の最適化と候補頂点の枝刈りによる探索空間の削減効果を評価する．実験には提案手法の逐次処理版プロトタイプ実装を用いる．性能比較は次の 3 つの手法と行う：QuickSI [12], GraphQL [7], TurboISO [6]．QuickSI と GraphQL は様々なサブグラフマッチングアルゴリズムに関する性能調査 [9] において最も高い性能を示した手法である．TurboISO は BFS によるグラフ要約を行う近年の手法として選択した．QuickSI と GraphQL の実装は文献 [9] の著者より入手し，TurboISO の実装も著者 [6] より入手した．これらは全て逐次処理を行う手法である．実験に用いたマシンは CPU として Intel Core i7-5600U を搭載し，16GB のメモリを持つ．グラフデータは既存研究 [1, 6, 7, 9, 11] で広く利用されている yeast を用いる．yeast は 3112 頂点，12519 エッジ，71 種類の頂点ラベルを持つタンパク質相互作用ネットワークである．クエリグラフは既存研究 [1, 6, 9, 11] に倣い，データグラフからランダムに連結成分を抽出する．本論文では次の 2 種類のクエリを生成した．

（1）**BFS:** ランダムに選んだ点から BFS を行い，訪問した頂点から誘導されるグラフ．閉路を含むものやツリーなど，様々な形状が現れる．

（2）**パス:** 一本道で閉路を含まない開いたパス．

それぞれの種類のクエリについて，頂点数 4, 6, 8,10,12,14 のものをそれぞれ 1000 パターンずつ生成する．以降ではそれら 1000 パターンの平均値を報告する．またクエリによっては膨大な数の埋め込みが存在し列挙に極めて長い時間を要する場合があるため，文献 [6] と同様 1000 個の埋め込みを発見した時点で探索を打ち切る．提案手法のパラメータ (クラスタ係数) c としては yeast のクラスタ係数 0.108 を使用した．

4.1 クエリ処理速度

最初に提案手法のクエリ処理速度を評価する．表 2 は提案手法および既存手法の処理速度を 1 秒あたりのクエリ処理数 (queries per second, QPS) で示したものである．“DNF” (do not finish) は 1000 クエリの処理を 1000 秒以内に完了しなかったことを表す．提案手法は既存手法と比べ特に BFS クエリにおいて高い性能を示している．また BFS クエリとパスクエリの両方においてクエリ頂点数に対する性能低下が緩やかである．QPS では全体として TurboISO が最高性能をマークしているクエリが最も多いものの，TurboISO は BFS の 6 頂点で一度 DNF になったあと 8 頂点および 14 頂点で再び高い処理性能を見せるという不安定な結果になっている．いずれの手法も全く同じクエリグラフのセットを使っているにもかかわらず TurboISO だけがこのような振る舞いを示していることから，

表 2　提案手法と既存手法のクエリ処理速度．クエリ名のカッコ内は頂点数．数値の単位は queries per second (QPS).

クエリ	GraphQL	QuickSI	TurboISO	提案手法
BFS (4)	2919.14	3665.27	**14043.16**	6357.88
BFS (6)	1804.63	1731.52	DNF	**4359.75**
BFS (8)	1314.79	334.58	**7108.22**	3364.74
BFS (10)	1038.14	DNF	DNF	**2978.83**
BFS (12)	**783.93**	DNF	DNF	605.11
BFS (14)	372.63	DNF	**3136.86**	1653.72
パス (4)	2994.44	8314.83	**21331.37**	5734.06
パス (6)	1917.06	5275.79	**17216.29**	4272.02
パス (8)	1406.62	3248.18	**11084.05**	3402.75
パス (10)	1070.58	2174.05	**7578.22**	2639.71
パス (12)	887.40	561.07	**5154.19**	2545.50
パス (14)	723.49	318.92	**3030.60**	2223.82

TurboISO は特定のクエリ対して極端に性能を低下させる場合があるのではないかと推測される．提案手法も BFS の 12 頂点と 14 頂点の間で処理性能が逆転する振る舞いが見られるものの，総合的には安定した挙動を見せている．

4.2 部分埋め込みの生成数

以上のように提案手法は安定した性能を示すことが確認された．ここからはその理由を理解するため，提案手法のみに着目して詳細な調査を行う．なお以降で使用する機能を制限された提案手法では 10 頂点以上のクエリを処理するために長い時間を要するため，4, 6, 8 頂点のクエリのみ使用する．

サブグラフマッチングでは部分的な埋め込みを生成しながらバックトラックによって探索を行うため，部分埋め込みの生成数が性能の指標となる．部分埋め込みには最終的に完全な埋め込みとして抽出されるものと，途中でマッチング不可能であることが分かり無駄になるものの 2 種類がある．前者を減らすことはできないが，後者は枝刈りや効率的なマッチング順序で減らすことができる．そこで本節ではアルゴリズム 2 の関数 Match の呼び出しのうち，生成された埋め込みが完全な埋め込み結果に結びつかなかったものの回数を数える．

比較は提案手法のアプローチを有効にした場合と無効にした場合の 4 パターンで行う．1 つ目に，候補頂点の枝刈り (3.1 節) と動的なマッチング順序による探索 (3.2 節) の両方を行うものを「提案手法」と呼ぶことにする．2 つ目に，候補頂点の枝刈りのみ行うものを「枝刈り」と呼ぶ．枝刈りを行わない場合，任意のデータ頂点 v について $C[v]$ は v とラベルが一致する全てのクエリ頂点を含む．3 つ目に，動的なマッチング順序のみ用いるものを「順序最適化」と呼ぶ．最適化を行わない場合もマッチング開始点は同様に選択するが，動的なマッチングは行わずクエリグラフ上の深さ優先探索の訪問順でマッチングを行う．4 つ目に，枝刈りと順序最適化のどちらも行わない場合を「ベースライン」と呼ぶ．このベースラインは既存手法で言えば VF2 [3] に相当する．

実験結果を表 3 に示す．ベースラインと比べ，提案手法は無駄になる部分埋め込みの数を大幅に減少させていることが分かる．特に 8 頂点のパスでは 4 桁減少させている．さらに枝刈りと順序最適化の結果について見ていくと，それぞれの効果の大小関係はクエリによって異なっている．BFS クエリでは，4 頂

表 3　マッチングに失敗した部分埋め込みの生成数.

クエリ	ベースライン	枝刈り	順序最適化	提案手法
BFS (4)	211.55	70.31	163.40	32.56
BFS (6)	4587.29	1504.06	501.68	41.49
BFS (8)	110040.92	42604.63	1122.11	157.68
パス (4)	80.69	0.27	63.82	0.22
パス (6)	374.90	1.71	189.84	2.18
パス (8)	11624.19	8.95	709.37	2.49

表 4　枝刈り後に残る候補頂点の総数．カッコ内は「ラベルのみ」に対する残存候補頂点数の割合.

クエリ	ラベルのみ	局所	伝播
BFS (4)	1299.60	1126.23 (86.67%)	298.13 (22.94%)
BFS (6)	1962.27	1629.85 (83.06%)	293.75 (15.00%)
BFS (8)	2630.47	2146.51 (81.60%)	292.08 (11.10%)
パス (4)	1282.40	1128.03 (87.96%)	299.13 (23.33%)
パス (6)	1972.30	1659.76 (84.15%)	401.68 (20.37%)
パス (8)	2672.01	2195.40 (82.16%)	498.34 (18.65%)

点のときは枝刈りのほうが部分埋め込みの生成数が少なく，6 頂点と 8 頂点では順序最適化のほうが少ない．これは 4 頂点の時に順序最適化の効果が弱いというよりも，枝刈りの効果が高いのだと考えられる．枝刈りでは隣接頂点のラベルがクエリ頂点とマッチするかどうかを検査するため，直径 2 で閉路を含まないクエリであれば枝刈りの段階で厳密なマッチングを取ることができる．例えば次数 3 の頂点 1 つと次数 1 の頂点 3 つからなるツリー型のクエリは直径 2 なので，次数 3 の頂点の隣接関係を検査するだけでマッチするかしないかを厳密に判定することができる．しかし頂点数が増えてくると枝刈りだけでは厳密にマッチを取ることができなくなり，代わりに順序最適化の効果が強く現れてくる．例えばラベル a, b, c から成る三角形のクエリに対してラベル c, a, b, c, a のデータ頂点から成る開いたパスはマッチしないが，これを枝刈りでは検出することができない．以上のような効果により，提案手法の部分埋め込み生成数は枝刈りと順序最適化のいずれよりも小さくなっている．

一方，パスクエリではサイズに関わらず枝刈りのほう部分埋め込みの生成数は少ない．枝刈りでは閉路の存在を確認できないが，パスは閉路を含まないためこの欠点が問題になりにくい．逆に閉路であってはならないところが閉路になっていたケースが部分埋め込みの生成数に現れていると考えられる．

このように，枝刈りと順序最適化のどちらも探索空間の削減に貢献している．特に閉路を含まないクエリでは枝刈りの効果が強く表れ，閉路を含むクエリでは順序最適化の効果が大きいことが分かる．提案手法では 2 つのアプローチを組み合わせることによりどちらのクエリにも対応することができる．

4.3　候補頂点枝刈りの効果

次に候補頂点の枝刈りによって C に含まれるクエリ頂点の数が減少した割合を求める．結果を表 4 に示す.「局所」は 3.1.1 節にある隣接頂点の情報を用いた局所的な枝刈りを,「伝播」は 3.1.2 節にある枝刈り情報の伝播を意味する．ベースラインであるラベルの一致のみを用いる枝刈りと比べ，局所枝刈りでは 13–18%の削減率に留まっている．クエリグラフが小さいため，隣接頂点数だけで枝刈りできるケースが少なかったものと考えられる．しかし伝播を行うことで全体の 77–90%を削減できている．特にマッチングに失敗した部分埋め込みの数 (表 3) においてパスクエリはゼロに近い数値を示しているため，限界に近い程度まで候補頂点の枝刈りを行えているといえる．繰り返し処理 (アルゴリズム 1, 3–9 行目) が行われた回数も調査したが，いずれのクエリの場合も 3–5 回であり，クエリ毎の傾向は見られなかった．グラフ直径などデータグラフ側の性質に依存している可能性がある．

5.　関 連 研 究

サブグラフマッチングはアルゴリズムの改良と並列化という 2 つの方向性で高速化されてきた．特にアルゴリズムの改良においては主にマッチング順序の最適化とマッチング候補頂点の枝刈りという 2 つのアイデアの組合せがよく研究されている．そこで本節ではまず既存のアルゴリズムをマッチング順序最適化と枝刈りの観点から俯瞰し，最後に並列処理手法を紹介する．

5.1　マッチング順序最適化

最も古いサブグラフマッチングアルゴリズムと言われる Ullmann [17] はクエリグラフで与えられたままの順序でマッチングを行う．VF2 [3] はクエリグラフで最初に現れる頂点から順に，マッチング済みのクエリ頂点と隣接するクエリ頂点を順に選択する．QuickSI [12] はさらにラベルの出現頻度を考慮し，より頻度が小さい (希少な) ラベルを持つクエリ頂点から優先的にマッチングを行うことで探索を削減する．TurboISO [6] は neighborhood equivalence class (NEC) ツリーと呼ばれる構造へクエリグラフを要約する．NEC ツリーでは同じ頂点群に隣接する頂点が 1 つの頂点へとまとめられており，複数のクエリ頂点に対するマッチを一度に行うことができる．マッチング順序に関しては，マッチング開始点をデータ頂点にマッチさせたのち，そこから BFS を行って各 NEC 頂点に対応するデータ頂点数を計測し，探索範囲が少なくなる順序で NEC ツリーをマッチさせる．BoostISO [11] はデータグラフに関しても隣接頂点の包含関係に基づく圧縮を TurboISO に対して適用し，計算を削減している．さらに CFLMatch [1] はクエリグラフ中の密に結合された構造 (コア) に探索空間を大きく削減する効果があることに着目した．具体的にはクエリグラフのコアからマッチングを行い，その後他の末端構造のマッチングを行うことにより，マッチし得ない箇所の計算を早期に打ち切る．

以上の中に 1 頂点のマッチングごとに動的に順序を生成する手法は存在しない．TurboISO 以前の手法は 1 つの固定されたマッチング順序をグラフ全体に対して使用している．TurboISO 以降の手法はマッチング開始点をマッチさせてからその他のクエリ頂点のマッチング順序を決定しているため，その意味では動的にマッチング順序を生成している．しかしながら 1 頂点毎のマッチングで順序を生成しているわけではない．これに対して提案手法では現在進行中のマッチングの状況に応じて探索範囲が小さくなるように動的にマッチング順序を変更している．これにより高次数のデータ頂点に遭遇した際に生じる部分埋め込みの数を抑える．またクエリグラフ中の密に接続された構造は閉路を含むため CFLMatch と同じ構造に着目していることになるが，提案手法は隣接頂点集合の交差でマッチング候補頂点を抽出する点が異なる．

5.2 マッチング候補頂点の枝刈り

GraphQL [7] と SPath [19] は我々の提案手法でも用いているように，隣接頂点におけるラベルの出現回数をクエリ頂点とデータ頂点の間で比較することによる枝刈りを導入した．GraphQL はさらに簡易的なサブグラフマッチングを行うことでマッチング候補頂点を削減する．SPath は事前計算において 2 ホップ以上先の隣接頂点まで含めてラベルの出現回数を数えておくことで枝刈りの効果を向上させている．

提案手法は GraphQL および SPath と近いアプローチをとっている．しかしながら提案手法は 2 つの点で異なる．まずそれらの既存手法と異なり，ユーザーによる指定が必要なパラメータがない．このパラメータは最適値の発見が難しい上に性能への影響が大きいことが知られている [6]．さらに計算量の小さな枝刈り判定方法を用いることでより短時間に多くの枝刈りを実現している．

5.3 並列サブグラフマッチング

STwig [15] は分散メモリ環境を対象とした並列アルゴリズムである．クエリグラフを STwig と呼ばれる深さ 1 のツリー構造へ分解し，STwig 毎に並列にマッチングを行った後に，STwig 間で同一でなければならない頂点がマッチするよう結合を行う．STwig は GPU 向けの改良研究 [16] においても基礎となっている．また Gao ら [5] は分散メモリ環境において動的に変化するグラフ向けの近似サブグラフマッチングアルゴリズムを提案している．

並列化のアプローチはアルゴリズム毎に大きく異なるが，共通して重要なのはスレッドまたはプロセス間で共有されるデータの操作を減らすことである．提案手法においてはマッチング候補頂点の枝刈りを各データ頂点に対して独立に行っているため，並列化が容易である．

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．そこで本論文では次の 2 つのステップから成る高速なサブグラフマッチング手法を提案する．まずステップ 1 ではデータグラフの各頂点について繰り返し処理によってマッチング候補頂点の枝刈りを行う．そしてステップ 2 では部分埋め込みの生成数を小さく保つようにマッチング順序を動的に生成しながら埋め込みを発見する．実験ではこれら 2 つのステップが部分埋め込みの生成数を大幅に削減し，既存手法の実装に比肩する性能を示すことを確認した．

今後の課題としては更なる評価の実施が挙げられる．CFLMatch [1] など最新の手法との性能比較のほか，提案手法の並列処理効率についても調査する．また動的なマッチング順序の効果を確認するため，次数分布の異なる複数のデータグラフを用いた実験を行う．

文　　献

[1] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient Subgraph Matching by Postponing Cartesian Products. *Proceedings of the 2016 ACM SIGMOD Conference on Management of Data*, 1:1199–1214, 2016.

[2] R. Blumofe. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, aug 1996.

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs, 2004.

[4] Y. Fang, W. Lin, V. W. Zheng, M. Wu, K. C. C. Chang, and X. L. Li. Semantic proximity search on graphs with metagraph-based learning, 2016.

[5] J. Gao, C. Zhou, J. Zhou, and J. X. Yu. Continuous pattern detection over billion-edge graph using distributed framework, 2014.

[6] W.-S. Han, J. Lee, and J.-H. Lee. TurboISO: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 337–348, New York, NY, USA, 2013. ACM.

[7] H. He and A. 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*, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM.

[8] Q. Hu, S. Xie, J. Zhang, Q. Zhu, S. Guo, and P. S. Yu. HeteroSales: Utilizing Heterogeneous Social Networks to Identify the Next Enterprise Customer. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 41–50, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[9] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proceedings of the 39th international conference on Very Large Data Bases*, pages 133–144, 2012.

[10] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel. Fast Malware Classification by Automated Behavioral Graph Matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 45:1—-45:4, New York, NY, USA, 2010. ACM.

[11] X. Ren and J. Wang. Exploiting Vertex Relationships in Speeding Up Subgraph Isomorphism over Large Graphs. *Proc. VLDB Endow.*, 8(5):617–628, 2015.

[12] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, 2008.

[13] H. Shiokawa, Y. Fujiwara, and M. Onizuka. Fast Algorithm for Modularity-Based Graph Clustering. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI 2013)*, pages 1170–1176, 2013.

[14] J. Shun and K. Tangwongsan. Multicore Triangle Computations Without Tuning. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 149–160, 2015.

[15] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.*, 5(9):788–799, 2012.

[16] H.-N. Tran, J.-j. Kim, and B. He. *Fast Subgraph Matching on Large Graphs using Graphics Processors*, pages 299–315. Springer International Publishing, Cham, 2015.

[17] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1):31–42, 1976.

[18] Y. Yuan, G. Wang, J. Y. Xu, and L. Chen. Efficient Distributed Subgraph Similarity Matching. *The VLDB Journal*, 24(3):369–394, 2015.

[19] P. Zhao and J. Han. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.*, 3(1-2):340–351, 2010.