DEIM Forum 2020 H3-2

GPUによる5ノードサブグラフ数え上げの高速化

菅波　柊也†　　天笠　俊之††　　北川　博之††

† 筑波大学 情報学群 情報科学類　〒305–8577 茨城県つくば市天王台1丁目1－1
†† 筑波大学 計算科学研究センター　〒305–8577 茨城県つくば市天王台1丁目1－1
E-mail: †suganami@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.ac.jp

あらまし　サブグラフの数え上げは，ネットワーク分析の基本的な手法の一つであり，バイオインフォマティクスやコンピュータサイエンスなどの様々な分野で利用されている．サブグラフの数え上げを行うアルゴリズムはいくつか提唱されているが，既存のアルゴリズムは，大規模なグラフに対して，実行に多くの時間を要するという問題がある．これは特にノード数が5以上のサブグラフの数え上げに対して, より顕著に現れる．この問題の解決法の一つとして，GPUを用いた並列処理が考えられる．GPUは多くのコア搭載し，高い並列度の処理を得意とする．提案手法ではGPUを用い，数え上げの探索を並列に行うことで高速化を図る．評価実験により，我々の手法は5ノードサブグラフの数え上げを行うstate-of-the-artな手法に比べ，4倍から10倍程度の高速化が可能であることを示した．

キーワード　GPU，サブグラフカウンティング

1　序　　論

グラフ構造はデータをノードとエッジで表したデータ構造である．グラフは実世界において様々な場面で利用されており，グラフを分析することにより様々な情報を抽出する技術が注目を集めている．近年データが大規模化し，ノード数が数億を超えるグラフも現れてきている．例えば，Instagramでは2018の一ヶ月あたりのアクティブユーザー数が10億を超えており，ユーザをノード，ユーザ間のフォロー/フォロワー関係をエッジで表したグラフを考えると大規模なグラフになることが分かる[25]．今後もデータの規模が大きくなっていく事を考えると，大規模なグラフに対する高速な分析手法が必要となる．

グラフ構造の分析手法の一つとしてサブグラフの数え上げが挙げられる．グラフにはトライアングルやクリークのような様々な構造のサブグラフが存在する．サブグラフの数え上げは，これらのサブグラフのグラフ中の出現回数を求めることによりグラフを分析する手法である．サブグラフの数え上げの結果は，複数のグラフで同じ値を取りうる直径などのグローバルな特徴に比べ，よりグラフ特有のローカルな特徴を表すことができ，コンピュータサイエンス，バイオインフォマティクス，ソーシャルサイエンスなどの様々な分野で用いられている[8][22]．

しかし，既存のサブグラフの数え上げアルゴリズムの多くは実行に多くの時間を要するという問題点が存在する．この問題は特に5ノード以上のサブグラフの数え上げに対して顕著に現れる．これは5ノード以上のサブグラフの数え上げにおいて組合せ爆発が起こることに起因する．エッジ数が数百万程度のグラフであっても，5ノードの各サブグラフは数十億から数兆個存在する．さらにサブグラフ数え上げを行うためには，実際の出現回数より多くの候補のパターンについて探索する必要がある．そのため数え上げには多くの探索が必要となる．これらのことにより，ノード数が5を超える場合，サブグラフの数え上げを行うためには多くの時間を要する．

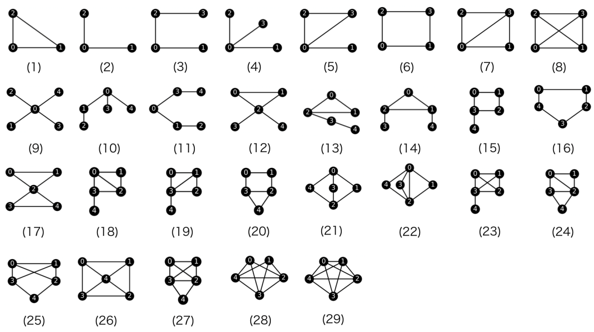
5ノードまでのサブグラフの数え上げを行う手法の中でstate-of-the-artなものとしてPinarらによるESCAPE[16]がある．ESCAPEは組合せを用いることにより，5ノードまでのサブグラフを効率よく求める．しかしESCAPEはグラフによっては探索に数時間から数日を要するため，より高速に5ノードまでサブグラフの数え上げを行う事が必要である．

一方，近年大規模なデータを処理するためにGPU (graphics processing unit) を用いた並列コンピューティングが注目されている．GPUはCPUに比べ，多数のコアを搭載しているため高い並列処理性能を持つ．このGPUの高い並列処理性を利用してGPUを汎用計算に用いる事をGPGPU(general-purpose computing on graphics processing units) と呼び，様々なアプリケーションの高速化に貢献している．

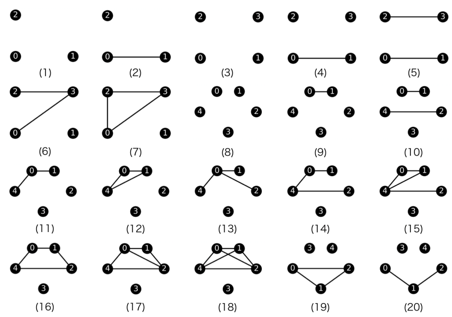
本稿では，GPUを用いてグラフ中の全ての5ノードまでのサブグラフの数え上げの高速化行う．図1に5ノードのグラフを示す．ESCAPE[16]により示されているように，全ての連結なサブグラフの数え上げを行えば，包除原理を用いることによりグラフに対して追加の探索を行うことなく，非連結なサブグラフの数え上げは定数時間で行うことができる．そのため本稿では連結なサブグラフの数え上げについてのみを対象とする．

我々はESCAPEの数え上げのアルゴリズムを注意深く観察し，サブグラフの探索がノードやエッジごとに独立に行えることを発見した．提案手法では，数え上げの独立な探索をGPUを用いてノードやエッジ単位で並列に実行することにより数え上げの高速化を行う．また提案手法の性能を評価するため，実世界のグラフデータセットを用いてESCAPEと実行時間の比較実験を行なった．その結果，我々の手法はESCAPEに比べ約4倍から10倍の高速化を達成した．

本稿の構成は，以下の通りである．2章で関連研究を紹介し，3章でGPUコンピューティングについて述べる．4章で前提となる知識について説明し，5章で提案手法について述べ，6章



(a) 連結な 5 ノードのグラフ



(b) 非連結な 5 ノードのグラフ

図 1: 5 ノードのグラフ

で実験結果を示す．最後に 7 章で本稿のまとめを述べる．

2 関 連 研 究

グラフ中のサブグラフの数え上げは重要なタスクであり様々なアルゴリズムが提案されている [18]．ここでは本研究と特に関連の強い 4, 5 ノードのサブグラフの数え上げの関連研究について述べる．数え上げのアルゴリズムは大きく厳密解法と近似解法に分かれる．

2.1 厳 密 解 法

2.1.1 4 ノードのまでサブグラフの数え上げ

CPU を用いた手法: 4 ノードまでのサブグラフの数え上げを行う手法として Marcus らによる RAGE [13] や Ortman らによる手法 [15]，Ahmed らによる PGD [3] などさまざまな手法が提案されている．それらの中で PGD は比較的高速にサブグラフの数え上げを行うことができる．この手法では CPU を用いて並列に数え上げを行う．PGD ではエッジ数が 10M 程度のグラフに対して，1 時間以下でサブグラフの数え上げを行うことができる．

GPU を用いた手法: GPU を用いてサブグラフの数え上げの手法として Rossi らによる手法 [20] がある．この手法では，グラフ全体として各 4 ノードのサブグラフが何回出現するかだけでなく，各エッジごとに各サブグラフが何回出現するかまでを求めることができる．

2.1.2 5 ノードまでのサブグラフの数え上げ

CPU を用いた手法: 5 ノードまでのサブグラフの数え上げを行う手法として Hočeva らによる ORCA [9] や Pinar らによる ESCAPE [16] がある．ESCAPE は 5 ノードまでのサブグラフの数え上げる state-of-the-art な手法である．さらに ESCAPE は 4 ノードのサブグラフの数え上げにおいて，PGD よりも高速に数え上げを行うことができる．また ORCA はグラフ全体で各サブグラフが何回出現するかだけでなく，各ノードごとに各サブグラフが何回出現するかまでを求めることができる．しかし ORCA は実行に多くの時間を必要とするため，大規模なグラフに用いることはできない．

GPU を用いた手法: GPU を用いて 5 ノードまでのサブグラフの数え上げを行う手法として Milinković らによる ORCA-GPU [14] がある．しかしこの手法は実行に時間を要し大規模なグラフに対して適用することが出来ない．

2.2 近 似 解 法

近似解法にはランダムウォークを利用した手法やカラーコーディングを利用した手法など様々な手法が提案されている．しかし GPU を用いたサブグラフの数え上げの近似解法はまだ提案されていない．

2.2.1 4 ノードまでのサブグラフの数え上げ

4 ノードまでのサブグラフの近似解法とし Elenberg らによる手法 [6] や Madhav らによる手法 [11] などがある．Elenberg らによる手法では後述する GUISE や GRAFT よりも高い精度で 4 ノードのサブグラフの出現回数を求められ，また実行速度も数十倍から数百倍高速である．

2.2.2 5 ノードまでのサブグラフの数え上げ

5 ノードまでのサブグラフの近似解法として Bhuiyan らによる GUISE [4] や Rahman らによる GRAFT [17]，Wang らによる MOSS-5 [24] などがある．MOSS-5 は高速かつ高精度で数え上げを行える手法であり，GRAFT や GUISE に比べ数百倍から数千倍高速に 5 ノードまでのサブグラフの数え上げを行うことができる．

3 GPU コンピューティング

OpenACC は GPU プログラミングにおいて広く用いられている CUDA に比べ学習コストが低く，また既存のコードに指示文を追加するだけでアクセラレータで実行できるため，可搬性や保守性，生産性に優れる．GPGPU(general-purpose computing on graphics processing units) は本来グラフィック向けに開発されたデバイスである GPU(graphics processingunit) を，汎用の計算に用いることである．GPU は CPU に比べ多くのコアが搭載されているため，高速に並列処理を行うことが可能である．そのため科学技術計算や機械学習など様々なアプリケーションにおいて，処理の高速化に貢献している．

GPGPU は適切に設計されたプログラムにおいては高い性能を発揮するが，不適切な実装やアルゴリズムによって CPU より悪い性能を示す場合がある．そのため GPU を用いて高速化を行うためには，適切なプログラム設計を行う必要がある．

本研究では，NVIDIA 社の GPU と OpenACC を用い GPU プログラミングを行った．以下では GPU の一例として NVIDIA 社の GPU の構造と特徴，及び OpenACC を用いたプログラムについて説明する．

3.1 NVIDIA GPU

GPU は複数の Streaming Multiprocessor(SM) から成り，各 SM は数十から数百の Scalar Processor(SP) を搭載している．SP は単純な処理に適しており，単純な大量の処理を行う場合，大量の SP による並列実行が有効となり高い性能を発揮する．

また，GPU には大きく分けて，グローバルメモリ，シェアードメモリ，レジスタの 3 種類のメモリが存在する．グローバルメモリは SM 外に存在し，シェアードメモリとレジスタは SM 上に存在する．グローバルメモリは 3 種類のメモリの中で最も容量の大きいメモリである．グローバルメモリは GPU 上の全ての SP からアクセス可能であるが，SM 外に位置することから，シェアードメモリやレジスタと比較するとアクセスが低速である．シェアードメモリはグローバルメモリよりも容量が小さく，同じ SM 上の SP からのみしかアクセスできないという制約はあるが，グローバルメモリよりも高速にアクセスできる．レジスタは 3 種類のメモリの中で最も容量の小さいメモリである．レジスタは同じ SM 上の SP からのみアクセス可能であり，シェアードメモリよりもさらに高速なアクセスが可能である．

3.2 OpenACC

OpenACC [2] はメニーコアアクセラレータ向けの並列プログラミング規格である．C/C++・Fortran のコードに対して，指示文を挿入することにより指示領域がアクセラレータにより実行される．OpenACC の指示文には主に，並列領域指示文，データ指示文，ループ指示文の 3 つがある．並列領域指示文では並列に実行する領域を指定する．データ指示文では CPU とアクセラレータ間のデータの転送を指示する．一般にアクセラレータは CPU とは独立したメモリを持つ．そのためアクセラレータ上で実行するためには，事前に CPU 側からアクセラレータ上へデータを転送しておく必要がある．ループ指示文は並列化の方法や並列の粒度を指示する．OpenACC の並列の粒度には gang，worker，vector の 3 段階が存在する．各ギャングは 1 つ以上の worker を持ち，各 worker は vector を用いてベクトル演算を行う．

4 前提知識

本稿では重みなし無向グラフ $G = (V, E)$ に対してグラフ中の 5 ノードまでのサブグラフの数え上げを行う．また数え上げを行う構造をパターンと呼び，パターンを $H = (V(H), E(H))$ と表す．ただし $V(H)$ はパターン H のノード集合，$E(H)$ はパターン H のエッジ集合である．

4.1 記　　法

本稿において用いる記号について説明する．本稿で用いる記号を表 1 に示し，いくつかのパターンの名称を図 2 に示す．$G^{\rightarrow}$ は下で定義される degree ordering に基づいて作成される DAG(directed acyclic graph) $G^{\rightarrow}$ を表す．

表 1: 記号の定義

記号	定義
V	G 中のノードの集合
$V(H)$	パターン H 中のノードの集合
E	G 中のエッジの集合
$E(H)$	パターン H 中のエッジの集合
$H\|_C$	ノード集合 C より誘導される H の部分グラフ
$d(i)$	ノード i の次数
$N(i)$	i の隣接ノード集合
$N^+(i)$	i より degree ordering の大きい隣接ノード集合
$N^-(i)$	i より degree ordering 小さい隣接ノード集合
$W(G)$	G 中のウェッジの数
$W(i,j)$	ノード i, j 間のウェッジの数
$W_{++}(i,j)$	ノード i, j 間のアウトウェッジの数
$W_{+-}(i,j)$	ノード i, j 間のインアウトウェッジの数
$t(i)$	ノード i を含むトライアングルの数
$T(i,j)$	エッジ (i,j) とトライアングルとなるノードの集合
$T^+(i,j)$	エッジ (i,j) とトライアングルとなり i, j より degree ordering の大きいノードの集合
$C_4(G)$	G 中の 4 サイクルの数
$C_4(i)$	ノード i が含まれる 4 サイクルの数
$C_4(i,j)$	エッジ (i,j) が含まれる 4 サイクルの数
$K_4(G)$	G 中の 4 クリーク
$K_4(i)$	ノード i が含まれる 4 クリーク
$K_4(i,j)$	エッジ (i,j) が含まれる 4 クリーク
$k_4(i,j,k)$	トライアングル (i,j,k) と 4 クリークとなるノードの集合
$k_4^+(i,j,k)$	トライアングル (i,j,k) と 4 クリークとなり， i, j, k より degree ordering の大きいノードの集合
$D(G)$	G 中のダイヤモンド
$TT(G)$	G 中の tailed-triangle の数

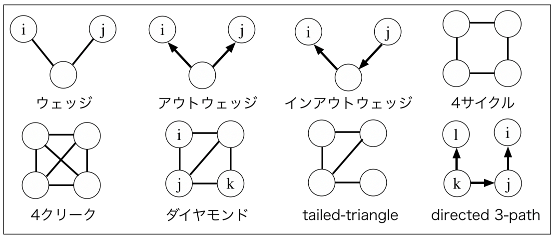


図 2: いくつかの基本的なパターンの名称

グラフの *degree ordering* を $\prec$ で表す．ノード i, j について $d(i) < d(j)$ または $d(i) \leqq d(j)$ かつ $i < j$ の時，i $\prec$ j と表す．G の全てのエッジについて $\prec$ に基づいて方向づけを行うことにより $G^{\rightarrow}$ を得る．degree ordering $\prec$ は全順序関係であるため $G^{\rightarrow}$ が閉路を持たない，つまり DAG であることは保証される．また，図 1a の連結なグラフの i 番目のグラフの G 中のサブグラフとしての出現回数を N_i と表す．

4.2 ESCAPE

ESCAPE は 5 ノードまでのサブグラフの数え上げを行う手法である．ESCAPE は多くの数え上げのアルゴリズムで起こる組合せ爆発を避けることにより，5 ノードまでのサブグラフの数え上げを行う．組合

せ爆発を回避するために ESCAPE では以下の 2 つのアイデアを利用する．

アイデア 1: パターンをより小さなパターンに分割する． クリークを除く全てのパターンにおいて，いくつかのノードを取り除くとそのパターンをいくつかの連結なグラフに分割できるようなノード集合が存在する (これらのノード集合をカットセットと呼ぶ)．数え上げを行うパターンを H とし，そのカットセットを S とする．H から S を取り除いてできるいくつかの連結なグラフをそれぞれ $C_1, C_2, \ldots$，S と C_i の和集合により誘導される H の部分グラフを H_i とする．このアイデアは以下のことが分かれば，H 数え上げが行えることに基づいている．

- G 中の全ての S について，S のノードが含まれる $H_1, H_2, \ldots$ の出現回数．
- パターン H よりノード数の少ない全てのパターン H' の G 中の出現回数．

分割の詳細については 4.2.1 で述べる．

アイデア 2: エッジに方向付けを行う． 入力される無向グラフ G のエッジについて $\prec$ に基づいて方向付けを行い $G^{\rightarrow}$ を作成する．この $G^{\rightarrow}$ について探索を行うことで方向付けにより同じ部分を重複して探索することを防ぎ，探索範囲を削減する．エッジに方向付けを行う手法はトライアングルカウンティング [21] やクリークカウンティング [5] においても用いられている．

ESCAPE ではこれらのアイデアを組み合わせパターンをより小さいパターンに分割し，その分割したパターンについて $G^{\rightarrow}$ を探索することでカウンティングを行う．

4.2.1 分割フレームワーク

パターンをより小さなパターンに分割するフレームワークについて説明する．$H = (V(H), E(H))$ をカウンティングを行いたいパターンとする．$H|_C$ を H 中のいくつかのノードの集合 C により誘導される H の部分グラフとする．まずはじめに同型を定義する．

定義 1. $G = (V, E), G' = (V', E')$ において，全単射 $\tau : V \rightarrow V'$ が $(\tau(u), \tau(v)) \in E'$ であるときに限り $(u, v) \in E$ である場合，G' は G と同型であるいい，G と同型であるグラフの集合を $\mathrm{AUT}(G)$ と表す．

次にマッチを定義する．

定義 2. 全単射 $\pi : S \rightarrow V(H)$ において $(\pi(s_1), \pi(s_2))$ が H のエッジであるときに限り，$S \subseteq V$ かつ $\forall s_1, s_2 \in S, (s_1, s_2)$ が G のエッジである場合，π を H のマッチという．G 中の H の異なるマッチの集合を $\mathrm{match}(H)$ と表す．

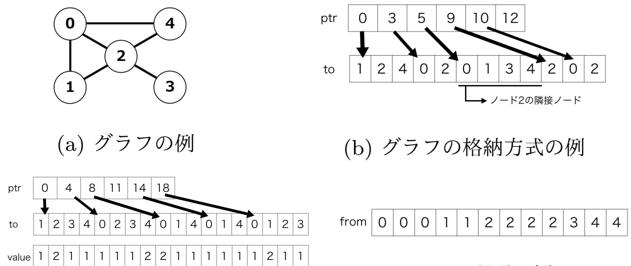
定義よりパターン H のグラフ中のサブグラフとしての出現回数は $|\mathrm{match}(H)|/|\mathrm{AUT}(H)|$ である．$\mathrm{AUT}(H)$ は事前に求めることができるため，$match(H)$ を求めることができれば H のグラフ中のサブグラフとしての出現回数を求めることができる．また π が単射である，つまり $|S| < |V(H)|$ であるとき，π を部分マッチという．

定義 3. 部分マッチ $\sigma : T \rightarrow V(H)$ とマッチ $\pi : S \rightarrow V(H)$ について，$S \supset T$ かつ $\forall t \in T, \pi(t) = \sigma(t)$ である場合，部分マッチ $\sigma : T \rightarrow V(H)$ はマッチ $\pi : S \rightarrow V(H)$ に拡張されるという．

定義 4. σ を G 中の H の部分マッチとする．σ を拡張し H のマッチとなる数を σ の H-degree といい，$\deg_H(\sigma)$ と表す．

次に H を小さなパターンに分割することにより得られるフラグメントについて定義する．

定義 5. C をカットセットとする．カットセットの定義より，H から C を取り除くと H はいくつかの連結な要素 $S_1, S_2, \ldots$ に分割される．この分割されたそれぞれの要素と C の和集合，つまり $C \cup S_1, C \cup S_2, \ldots$ によって誘導される H の部分グラフを H の C-フラグメントといい，この集合を $\mathrm{Frag}_C(H)$ と表す．



(a) グラフの例

(b) グラフの格納方式の例

(c) wedge の格納方式の例

(d) from 配列の例

図 3: データの格納方式の例

ここで $H|_C$ のマッチ σ を考える．σ を H に拡張するためには，σ を $Frag_C(H)$ の全ての要素に対して拡張できれば十分である．σ を拡張した $Frag_C(H)$ の要素 $F_i, F_j (i \neq j)$ において $H|_C$ を除いた F_i と F_j のノード集合が互いに素である場合，σ を拡張した $F_1, F_2, \ldots, F_{|Frag_C|}$ を合併すると H のマッチとなる．一方で，F_i と F_j が互いに素でない場合，$F_1, F_2, \ldots, F_{|Frag_C|}$ の合併させた物は H とは異なるパターン H' となる．この H' を *shrinkage* と呼ぶ．

定義 6. C をカットセット，$\mathrm{Frag}_C(H)$ の要素を $F_1, F_2, \ldots$ とする．パターン H を異なるパターン H' にする *C-shrinkage* を以下を満たす集合 $\{\sigma, \pi_1, \pi_2, \ldots, \pi_{|Frag_C(H)|}\}$ とする．

- $\sigma : H|_C \rightarrow H'$ が H' の部分マッチである．
- 各 $\pi_i : F_i \rightarrow H'$ が H' の部分マッチである．
- 各 π_i は σ を拡張したものである．
- H' の全てのエッジ (u, v) について，$\pi_i(a) = u$ かつ $\pi_i(b) = v$ であるような $F_i \in Frag_C(H)$ のエッジ (a, b) が存在する．

定義 7. H からの *C-shrinkage* が少なくとも一つ存在するパターン H' の集合を $Shrink_{C(H)}$ とする．$H' \in Shrink_C(H)$ について，互いに素な *C-shrinkage* の数を $\mathrm{numSh}_C(H, H')$ と表す．

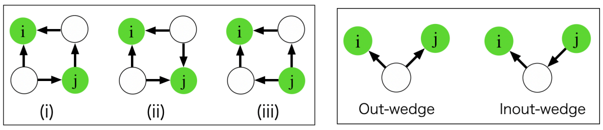
次に $\mathrm{match}(H)$ を求めるための補題を示す．この補題より全ての $H|_C$ の σ の $\deg_H(\sigma)$，全ての C-フラグメント F，起こりうる全ての shrinkage の出現回数が分かれば $\mathrm{match}(H)$ を求めることができる．

補題 1. C をカットセットとする．

$$match(H) = \sum_{\sigma \in match(H|_C)} \prod_{F \in Frag_C(H)} deg_{F(\sigma)} - \sum_{H' \in Shrink_C(H)} numSh_C(H, H') match(H')$$

5 提案手法

本章では GPU を用いた 5 ノードまでのサブグラフの数え上げについて述べる．提案手法では前節で説明した ESCAPE の分割フレームワークに基づいた数え上げアルゴリズムを GPU 用いて並列に実行することにより，5 ノードまでのサブグラフの数え上げを高速化する．パターン H の探索において，カットセットを C とすると探索はグラフ中の全ての $H|_C$ のマッチに対して行う．例えば，$H|_C$ のマッチがエッジの場合，グラフ中の全てのエッジが $H|_C$ のマッチとなるため，全エッジについて探索を行う．この探索は各 $H|_C$ 毎に独立であるため，$H|_C$ 単位でのスレッド並列化を行うことにより並列に探索を行うことができる．提案手法ではこれに基づきノード単位またはエッジ単位でスレッド並列化し探索を行う．



(a) DAG である 4 サイクル　　(b) フラグメント

図 4: DAG である 4 サイクルとそのフラグメント

(a) DAG である 4 クリーク　　(b) DAG である 5 クリーク

図 5: DAG であるクリーク

5.1　データ構造

数え上げを行うために，グラフやトライアングルの基本的な情報の他に，各ノードごとにウェッジを形成するノードと各ウェッジごとにダイヤモンドを形成するノードをそれぞれ記録しておく必要がある．ここで，あるノードとウェッジを形成するノードとは図 2 のウェッジにおいて，ノード i に対してノード j の位置となるノードのことである．ただし i, j 間のエッジの接続関係は問わない．またウェッジとダイヤモンドを形成するノードとは図 2 のダイヤモンドにおいてウェッジ (i, j, k) に対してノード l となる位置のノードである．ただし i, k 間の接続関係は問わない．我々の検証により，実世界のグラフにおいてウェッジとダイヤモンドの配列は比較的疎となるが分かった．そのため本稿ではウェッジとダイヤモンドの配列を CSR (compressed sparse row) 形式に基づいた表現法により表す．CSR はグラフを表現するために一般的に使用されているデータレイアウトであり，重みなしグラフの場合 ptr 配列と to 配列からなる．to 配列は各頂点の隣接ノードを格納しており，ptr 配列はある頂点の隣接ノードが to 配列中のどこに格納されているかを表している．例えば図 3a のグラフを CSR 方式で表すと図 3b のようになる．ノード 2 の隣接ノードの情報は to 配列の $ptr[2] = 4$ から $ptr[3] = 7$ の範囲で表される．また本稿においてグラフは CSR 方式で表されるとする．

次にウェッジを CSR に基づいた方式で表現する方法を説明する．実際には数え上げにおいてウェッジを利用するためには，あるノードとどのノードがウェッジを形成するかだけでなく，あるノードがどのノードと何個ウェッジを形成するかまでを知っている必要がある．ptr 配列と to 配列を用いることで，あるノードがどのノードとウェッジを形成するかの情報は得られるが，何個のウェッジを形成するかの情報を得ることができない．そこで本稿ではあるノードがどのノードと何個のウェッジを形成するかを表す value 配列を導入する．

例として図 3a のグラフはウェッジは図 3c のように表される．図 3a において，ノード 0 はノード 2 と $(0, 1, 2)$ と $(0, 4, 2)$ の 2 つのウェッジを形成するため value は 2 となる．ダイヤモンドについてもウェッジと同様に CSR 方式で表す．ただしダイヤモンドについては value 配列の情報のみが必要となるため，value 配列のみを保持する．

5.2　サブグラフ数え上げのアルゴリズム

5.2.1　4 ノードのサブグラフの数え上げ

4 ノードのサブグラフの数え上げについて説明する．

N_3, N_4, N_5, N_7 の数え上げ：　N_3, N_4, N_5, N_7 の数え上げは以下の式により行う．これらは容易に並列に実行できる．

$$N_3 = \sum_{(i,j \in E} (d(i) -)(d(j) - 1) - 3N_1$$

$$N_4 = \sum_{i \in V} \binom{d(i)}{3}$$

$$N_5 = \sum_{i \in V} t(i)(d(i) - 2)$$

$$N_7 = \sum_{(i,n) \in E} \binom{|T(i,j)|}{2}$$

N_6 (4 サイクル) の数え上げ：　4 サイクルの数え上げには分割とエッジの方向付けを用いる．DAG である 4 サイクルは対称となる場合を除くと図 4a に示すように 3 種類存在する．これらの 4 サイクルに対して degree ordering が最も大きいノードを i，その対角のノードを j とし i，j (図において色付けされているノード) をカットセットとする．したがってフラグメントはウェッジとなる．このウェッジにはエッジの方向を考慮すると図 4b のようにアウトウェッジとインアウトウェッジの 2 種類存在する．そのため以下の式より N_6 の数え上げを行うことができる．

$$N_6 = \sum_{i \in V} \sum_{j \prec i} \binom{W_{++}(i,j) + W_{+-}(i,j)}{2}$$

Algorithm 1　4 クリーク数え上げ

1: $N_8 \leftarrow 0$
2: **for each** $(i, j) \in E$ **do in parallel** // $i \prec j$
3: 　$sort(T(i, j), [\,]\ (a,\ b)\{\ return\ a \prec b\})$ // $T^+(i, j)$ を degree ordering の昇順にソート
4: 　**for each** $k_u \in T^+(i, j)$ **do** // $i \prec j \prec k_u$
5: 　　**for each** $k_v \in \{k_{u+1}, \dots k_{|T|}\}$ **do** // 　$i \prec j \prec k_u \prec k_v$
6: 　　　**if** $k_v \in N(k_u)$ **then** // k_u, k_v 間にエッジが張る時
7: 　　　　$N_8 \leftarrow N_8 + 1$
8: 　　　**end if**
9: 　　**end for**
10: 　**end for**
11: **end for**

N_8 (4 クリーク) の数え上げ： クリークにはカットセットが存在しない．そのためエッジの方向付けのみを用いて数え上げを行う．DAG である 4 クリークは対称となる場合を除くと図 5a の 1 種類のみである．そのため $G^{\rightarrow}$ に対して，図 5a となる 4 クリークを探索することで数え上げを行う．4 クリークの数え上げのアルゴリズムを Algorithm1 に示す．Algorithm1 では 3 行目で $T^+(i, j)$ を degree ordering の昇順になるようにソートする．4,5 行目で $T^+(i, j)$ から $k_u \prec k_v$ となるように k_u と k_v を選び 6 行目で k_u と k_v の間にエッジを張るかを確認する．k_u と k_v がエッジを張る場合 $\{i, j, k_u, k_v\}$ は 4 クリークとなる．

5.2.2　5 ノードのサブグラフの数え上げ

次に 5 ノードのサブグラフの数え上げについて説明する．ここでは 4 ノードまでのサブグラフの数え上げは完了しているとする．図 6 はノード数 1.69M，エッジ数 28.8M の tech-as-skitter に対して，ESCAPE を実行した時の 5 ノードの各パターンの探索にかかる時間である．これより，実行時間のほとんどがいくつかのパターンの数え上げによって占めていることが分かる．これはほとんどのパターンの数え上げは単純なノードやエッジのループで実行できるのに対し，いくつかのパターンの探索は深いループが必要となるためである．そのためここでは，特に時間を要するパターンの数え上げについて詳細に説明する．

N_{16}(5 サイクル) の数え上げ： DAG である 5 サイクルは対称となる場合を除くと図 7a のように 2 種類存在する．図 7a の i, l (色のついたノード) をカットセットとするとフラグメントは directed 3-path とウェッジである．このウェッジには 2 種類のエッジの方向が存在する．そのため $G^{\rightarrow}$ に対して図 7a となる 5 サイクルを探索することで探索を行う．5 サイクルの数え上げのアルゴリズムを Algorithm2 に示す．Algorithm2 では 2 行目から 4 行目で図のような directed 3-path となるノード i, j, k, l を選ぶ．5 行目でノード i がノード l とウェッジを形成するかを判定する．ノード i がノード l とウェッジを形成

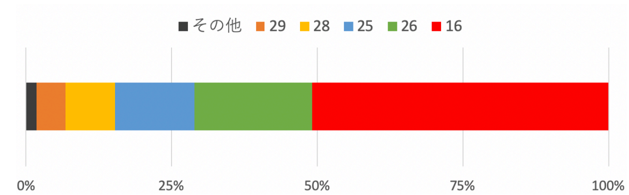
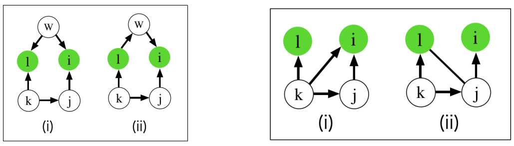


図 6: ノード数 1.69M，エッジ数 28.8M の tech-as-skitter に対し ESCAPE を実行した時の 5 ノードの各サブグラフの数え上げに要する時間



(a) DAG である 5 サイクル　(b) shrinkage

図 7: DAG である 5 サイクルと shrinkage

する場合，5 サイクルとなるためその分カウントを増やす．8 行目で i と l 間にエッジが張るかを判定する．i と k の間にエッジが張る時 (i,k,l) でウェッジを形成する．その場合 6 行目において図 7b(i) のような時も 5 サイクルとしてカウントしてしまう．そのため 9 行目において shrinkage となる分のカウントを減らす．11 行目についても同様である．

Algorithm 2　5 サイクルの数え上げ

1: $N_{16} \leftarrow 0$
2: **for each** $(i,j) \in E$ **do in parallel** // $i \prec j$
3: **for each** $k \in N^{-}(j)$ **do** // $k \prec j$
4: **for each** $l \in N^{+}(k)$ **do** // $k \prec l$
5: **if** ノード i がノード l とアウトウェッジまたはインアウトウェッジを形成する **then**
6: $N_{16} \leftarrow N_{16} + W^{++}(i,l) + W^{+-}(i,l)$
7: **end if**
8: **if** $i \in N(k)$ **then** // i と k の間にエッジが張る時
9: $N_{16} \leftarrow N_{16} - 1$
10: **end if**
11: **if** $j \in N(l)$ **then** // j と l の間にエッジが張る時
12: $N_{16} \leftarrow N_{16} - 1$
13: **end if**
14: **end for**
15: **end for**
16: **end for**

N_{25}(diamond-wedge) の数え上げ:　diamond-wedge のフラグメントはダイヤモンドとウェッジである．そのためダイヤモンドとウェッジについて探索する．diamond-wedge の数え上げのアルゴリズムを Algorithm3 に示す．2 行目から 4 行目でダイヤモンドとなる i,j,k,l を選ぶ．5 行目で i が l とウェッジを形成するかを判定し，ウェッジを形成する場合 diamond-wedge となるためカウントを $W(i,l)$ だけ増やす．また i と l がウェッジを形成するかの判定は to 配列の ptr[i] から ptr[i+1] の範囲に l が存在するかを二分探索を用いて確認する．

N_{26}(wheel) の数え上げ:　wheel のフラグメントはダイヤモンドである．Algorithm4 に wheel の数え上げのアルゴリズムを示す．前述したようにダイヤモンドの value 配列の値はあるウェッジが何個のダイヤモンドを形成するかを表している．そのため value 配列の値の組みわせにより wheel の出現回数を求めることができる．我々の手法ではダイヤモンドを CSR 方式を用いて表しているため，単純なループにより wheel の数え上げを行うことができる．

Algorithm 3　diamond-wedge の数え上げ

1: $N_{25} \leftarrow 0$
2: **for each** $(i,j) \in E$ **do in parallel**
3: **for each** $k \in T(i,j)$ **do** // i,j,k はトライアングル
4: **for each** $l \in T(j,k)$ **do** // i,j,k,l はダイヤモンド
5: **if** ノード i がノード l とウェッジを形成する **then**
6: $N_{25} \leftarrow N_{25} + W(i,l)$
7: **end if**
8: **end for**
9: **end for**
10: **end for**

Algorithm 4　wheel の数え上げ

1: $N_{26} \leftarrow 0$
2: **for each** $u < |value|$ **do in parallel**
3: $N_{26} \leftarrow N_{26} \binom{diamond\_value[u]}{2}$
4: **end for**

Algorithm 5　almost-5clique の数え上げ

1: $N_{28} \leftarrow 0$
2: **for each** $(i,j) \in E$ **do in parallel** // $i \prec j$
3: **for each** $k \in T(i,j)$ **do** // i,j,k はトライアングル
4: $N_{28} \leftarrow N_{28} + \binom{k_4(i,j,k)}{2}$
5: **end for**
6: **end for**

N_{28}(almost-5clique) の数え上げ:　almost-5clique のフラグメントは 4 クリークである．Algorithm5 に almost-5clique 数え上げのアルゴリズムを示す．2 行目から 3 行目でトライアングルとなる i,j,k を選ぶ．その i,j,k について 4 クリークとなるものから 2 つ選ぶとその 5 ノードは almost-5clique となる．

N_{29}(5 クリーク) の探索:　DAG である 5 クリークは対称となる場合を除くと図の 1 種類のみである．そのため $G^{\rightarrow}$ に対して図となる 5 クリークを探索することで数え上げを行う．5 クリークの数え上げを行うアルゴリズムを Algorithm6 に示す．2 行目から 4 行目で $i \prec j \prec k$ となるトライアングル (i,j,k) を選ぶ．5 行目で $k_4^{+}(i,j,k)$ を degree ordering の昇順にソートする．6,7 行目で $k_4^{+}(i,j,k)$ から $l_u \prec l_v$ となるように 2 ノードを選ぶ．8 行目において $l_u \prec l_v$ がエッジを張る場合 $\{i,j,k,l_u,l_v\}$ は 5 クリークとなる．

Algorithm 6　5 クリークの数え上げ

1: $N_{29} \leftarrow 0$
2: **for each** $(i,j) \in E$ **do in parallel** // $i \prec j$
3: **for each** $k \in T^{+}(i,j)$ **do** // $i \prec j \prec k$ のトライアングル
4: $sort(k_4^{+}(i,j,k), [\,]\ (a,\ b)\{\ return\ a \prec b\})$ // $k_4^{+}(i,j,k)$ を degree ordering の昇順にソート
5: **for each** $l_u \in k_4^{+}(i,j,k)$ **do** // i,j,k,l_u は 4 クリーク
6: **for each** $l_v \in \{l_{u+1}, \ldots l_{|k_4^{+}(i,j,k)|}\}$ **do** // i,j,k,l_u は 4 クリーク
7: **if** $l_v \in N(l_u)$ **then** // l_uと l_v がエッジを張る
8: $N_{29} \leftarrow N_{29} + 1$
9: **end if**
10: **end for**
11: **end for**
12: **end for**
13: **end for**

Algorithm 7 node-centric

```
G = (V, E)
for each u ∈ V do in parallel
    for each v ∈ N(u) do
        // do counting
    end for
end for
```

Algorithm 8 edge-centric

```
G = (V, E)
for each (u, v) ∈ E do in parallel
    // do counting
end for
```

その他のサブグラフの数え上げ: その他のサブグラフについては単純なループにより並列に数え上げを行うことができる．各サブグラフの数え上げの式を以下に示す．

$$N_9 = \sum_{i \in V} \binom{d(i)}{4}$$

$$N_{10} = \sum_{(i,j) \in E} [(d(i)-1)\binom{d(j)-1}{2} + (d(j)-1)\binom{d(i)-1}{2}] - 2N_5$$

$$N_{11} = \sum_{i \in V} \sum_{(i,j) \in E} (d(j)-1) - 4N_6 - N_5 - 3N_1$$

$$N_{12} = \sum_{i \in V} t(i)(d(i)-2)$$

$$N_{13} = \sum_{(i,j) \in E} ((d(i)-1)t(j) + (d(j)-1)t(i)) - 4N_7 - 6N_1 - 2N_5$$

$$N_{14} = \sum_{(i,j) \in E} |T(i,j)|(d(i)-2)(d(j)-2) - 2N_7$$

$$N_{15} = \sum_{i \in V} C_4(i)(d(i)-2) - 2N_7$$

$$N_{17} = \sum_{i \in V} \binom{t_i}{2} - 2N_7$$

$$N_{18} = \sum_{(i,j) \in E} \sum_{k \in T(i,j)} (d(k)-2)(|T(i,j)|-1) - 12N_8$$

$$N_{19} = \sum_{(i,j) \in E} \binom{|T(i,j)|}{2} (d(i)+d(j)-6)$$

$$N_{20} = \sum_{(i,j) \in E} C_4(i,j)|T(i,j)| - 4N_7$$

$$N_{21} = \sum_{i \in V} \sum_{i \prec j} \binom{W(i,j)}{3}$$

$$N_{22} = \sum_{(i,j) \in E} \binom{|T(i,j)|}{3}$$

$$N_{23} = \sum_{i \in V} K_4(i)(d(i)-3)$$

$$\begin{aligned} N_{24} = \sum_{(i,j) \in E} \sum_{k \in T(i,j)} & ((|T(i,j)|-1)(|T(i,k)|-1) \\ & + (|T(i,j)|-1)(|T(j,k)|-1) + (|T(j,k)|-1)(|T(i,k)|-1)) \end{aligned}$$

$$N_{27} = \sum_{(i,j) \in E} K_4(i,j)(|T(i,j)|-2)$$

5.3 ウェッジとダイヤモンドの転送の分割

5.1 節で述べたようにウェッジとダイヤモンドを CSR 方式で表す．CSR 方式は疎な配列を効率良く表現することができる．しかしグラフによってはウェッジやダイヤモンドの配列のサイズが大きくなり，全てを GPU のメモリに乗せることができないことがある．そこで提案手法では配列の要素を b ごとに分割して GPU に送り，要素数 b の配列を用いて数え上げを行う．これを配列の全ての要素を GPU に送るまで繰り返す．これによりウェッジやダイヤモンドが GPU のメモリに乗らない場合に対処する．

5.4 ロードバランスの改善

Algorithm7 と Algorithm8 はどちらも全てのエッジについてのループである．2 つのアルゴリズムの違いは各ノードごとに並列化を行うか各エッジごとに並列化を行うかである．一般に実世界のグラフの次数分布はべき乗則に従うことが知られている [7]．そのため node-centric により並列化を行なった場合，各ノードの隣接ノード集合の大きさの偏りから，スレッド間のタスクの割り当てに偏りが生じ，並列の効率が低下する可能性がある．[10] そのため本稿では edge-centric により並列化を行うことを目指す．しかし前述したようにグラフは CSR 方式で表される．CSR は空間効率やキャッシュ効率に優れているが，to 配列から元のノードを参照できないため edge-centric に並列化を行えないという問題点がある．そこで図 3d に示すように to 配列の要素と対応した from 配列を導入する [23]．from 配列の i 番目の値は to 配列の i 番目の要素がどのノードと隣接しているのかを表す．from 配列を用いることにより，CSR 方式の利点を得ながら edge-centric による並列化を行うことが可能となる．

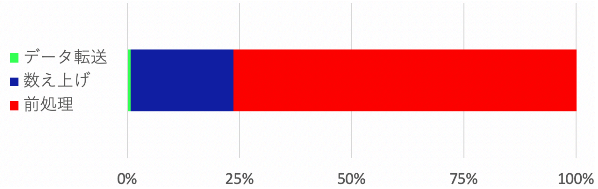


図 8: ノード数 1.69M，エッジ数 28.8M の tech-as-skitter に対し提案手法を用いて 5 ノードのサブグラフ数え上げを行った時の実行時間の割合

6 実　　験

本節では提案手法の性能を評価するため，4 ノードと 5 ノードのサブグラフの数え上げの state-of-the-art な手法である ESCAPE と実行速度の比較評価を行う．ESCAPE はソースコードが公開されており本実験では ESCAPE の実行にそのコードを利用した [1]．また本実験には CPU として Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz, 64GB, GPU として NVIDIA Tesla V100, 32GB を搭載しているマシンを使用した．提案手法のコンパイルには pgc++ 18.5-0，ESCAPE のコンパイルには g++ (GCC) 4.8.5 を用いた．データセットとして Citation Network Dataset [19] と SNAP [12] のグラフを用いのグラフを用いた．グラフはエッジの方向を無視し，重複したエッジと自己ループを取り除いて使用する．データセットの詳細を表 2 に示す．

実行時間の比較: ESCAPE と提案手法の実行時間の比較を表 2 に示す．ESCAPE-k は ESCAPE による k ノードのサブグラフの数え上げの結果であり，proposal-k は提案手法による k ノードのサブグラフの数え上げの結果である．ただし実行時間が 6 時間を超えた場合タイムアウトとした．結果から 4 ノード，5 ノードどちらの場合においても，提案手法が全てのデータセットにおいて高速化していることが示された．

4 ノードのサブグラフの数え上げにおいて，ノード数 2.25M，エッジ数 21.6M の tech-ip に対して最も大きく高速化し，提案手法は既存手法に比べ 5 倍の高速化を達成している．また，提案手法において soc-pokec, tech-ip のどちらも約 11 秒で終了しているが既存手法においては soc-pokec は約 36 秒，tech-ip は約 60 秒で終了している．これは提案手法の実行時間のほとんどが CPU でウェッジなどを作成する前処理の時間で占められていることに起因すると考えられる．soc-pokec, tech-ip の 2 つのデータセットにおいて，前処理に必要な時間はほぼ同程度である．しかし既存手法では tech-ip は soc-pokec に比べ前処理後の数え上げにより多くの時間を必要とする．そのため 2 つのデータセットの実行時間に差が生じた．一方で提案手法において前処理後の数え上げに必要な時間はどちらのデータセットでも 1 秒に満たないため実行時間が同程度になった．

5 ノードの数え上げでは，ノード数 1.63M，エッジ数 22.3M の soc-pokec において最も高速化しており，10 倍の高速化を達成している．ノード数 1.69M，エッジ数 28.8M の tech-as-skitter において最も高速化率が低く約 4 倍の高速である．高速化率のばらつきの要員としてグラフ中のウェッジやダイヤモンドの出現回数が考えられる．ウェッジやダイヤモンドを多く含む場合，前処理に多くの時間を必要とするため実行時間が増加する．

ボトルネックの分析: tech-as-skitter の提案手法による 5 ノードサブグラフの実行時間を，CPU によりウェッジやダイヤモンドなどを作成する前処理の時間，GPU により数え上げを行う時間，CPU と GPU 間のデータ転送の時間の 3 つに分けた時の実行時間の割合を図 8 に示す．CPU による前処理が最も多くの時間を要していることがわかる．

7 ま　と　め

本稿では数え上げの独立性に着目し，GPU を用いて探索を並列に行うことで 5 ノードまでのサブグラフの数え上げの手法を提案した．提案手法は 5 ノードの

表 2: サブグラフの数え上げの実行時間 (単位は全て秒)

データセット	$\|V\|$	$\|E\|$	$\|T\|$	ESCAPE-4	proposal-4	ESCPE-5	proposal-5
soc-brightkite	56.7K	426K	494K	0.13	0.03	7.16	1.79
soc-pokec	1.63M	22.3M	32.6M	36.05	11.2	1.79K	174.1
tech-as-skitter	1.69M	28.8M	28.8M	11.3	2.68	1.27K	321.5
web-wiki-ch-internal	1.93M	8.5M	18.2M	12.8	3.87	1.73K	210.4
web-hudong	1.98M	14.43M	21.6M	22.2	5.37	2.55K	396.7
web-baidu-baike	2.14M	17.01M	25.2M	27.1	9.41	3.61K	596.7
tech-ip	2.25M	21.6M	2.3M	60.8	11.67	-	18.1K

サブグラフの数え上げを行う手法 state-of-the-art な手法に比べ，約 4 倍から 10 倍の高速化を達成した．また 4 ノードまでのサブグラフの数え上げに対しても，約 3 倍から 5 倍の高速化を達成した．今後の課題としては，手法のボトルネックとなっているウェッジとダイヤモンドの構築時間の高速が挙げられる．現在の手法ではウェッジとダイヤモンドを CPU による逐次処理により作成しているため，これらを並列に行うことで数え上げの高速化が可能だと考えられる．

文　献

[1] Escape. https://bitbucket.org/seshadhri/escape.

[2] Openacc-standard.org. `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf`.

[3] Nesreen K Ahmed, Jennifer Neville, Ryan A Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining*, pages 1–10. IEEE, 2015.

[4] Mansurul A Bhuiyan, Mahmudur Rahman, Mahmuda Rahman, and Mohammad Al Hasan. Guise: Uniform sampling of graphlets for large graph analysis. In *2012 IEEE 12th International Conference on Data Mining*, pages 91–100. IEEE, 2012.

[5] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*, pages 589–598. International World Wide Web Conferences Steering Committee, 2018.

[6] Ethan R Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G Dimakis. Distributed estimation of graph 4-profiles. In *Proceedings of the 25th International Conference on World Wide Web*, pages 483–493, 2016.

[7] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM computer communication review*, volume 29, pages 251–262. ACM, 1999.

[8] Wayne Hayes, Kai Sun, and Nataša Pržulj. Graphlet-based measures are suitable for biological network comparison. *Bioinformatics*, 29(4):483–491, 2013.

[9] Tomaž Hočevar and Janez Demšar. A combinatorial approach to graphlet counting. *Bioinformatics*, 30(4):559–565, 2014.

[10] Yang Hu, Hang Liu, and H Howie Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182. IEEE, 2018.

[11] Madhav Jha, C Seshadhri, and Ali Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings of the 24th International Conference on World Wide Web*, pages 495–505, 2015.

[12] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[13] Dror Marcus and Yuval Shavitt. Rage–a rapid graphlet enumerator for large networks. *Computer Networks*, 56(2):810–819, 2012.

[14] Aleksandar Milinkovi ć , Stevan Milinkovi ć , and Ljubomir Lazi ć . A contribution to acceleration of graphlet counting. In *Infoteh Jahorina Symposium*, volume 14, pages 741–745.

[15] Mark Ortmann and Ulrik Brandes. Quad census computation: Simple, efficient, and orbit-aware. In *International Conference and School on Network Science*, pages 1–13. Springer, 2016.

[16] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1431–1440. International World Wide Web Conferences Steering Committee, 2017.

[17] Mahmudur Rahman, Mansurul Alam Bhuiyan, and Mohammad Al Hasan. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2466–2478, 2014.

[18] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: Concepts, algorithms and applications to network motifs and graphlets. *arXiv preprint arXiv:1910.13011*, 2019.

[19] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[20] Ryan A Rossi and Rong Zhou. Leveraging multiple gpus and cpus for graphlet counting in large networks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1783–1792. ACM, 2016.

[21] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*, pages 606–609. Springer, 2005.

[22] Daizaburo Shizuka and David B McDonald. A social network perspective on measurements of dominance hierarchies. *Animal Behaviour*, 83(4):925–934, 2012.

[23] Tomokatsu Takahashi, Hiroaki Shiokawa, and Hiroyuki Kitagawa. Scan-xp: Parallel structural graph clustering algorithm on intel xeon phi coprocessors. In *Proceedings of the 2nd International Workshop on Network Data Analytics*, page 6. ACM, 2017.

[24] Pinghui Wang, Junzhou Zhao, Xiangliang Zhang, Zhenguo Li, Jiefeng Cheng, John CS Lui, Don Towsley, Jing Tao, and Xiaohong Guan. Moss-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 30(1):73–86, 2017.

[25] Xin L Wong, Rose C Liu, and Deshan F Sebaratnam. Evolving role of instagram in# medicine. *Internal medicine journal*, 49(10):1329–1332, 2019.