GPGPUによるδ-許容飽和頻出部分グラフマイニングの実装と評価

A GPU Implementation of δ-tolerance Closed Frequent Subgraph Mining

土岐達哉　尾崎知伸 *
Tatsuya Toki　Tomonobu Ozaki

日本大学大学院 総合基礎科学研究科
Graduate School of Intergrated Basic Sciences, Nihon University

Abstract: One of the most fundamental tasks in graph mining is the frequent subgraph discovery, i.e. enumeration of all subgraphs appearing frequently in graph databases. In general, however, frequent subgraph enumeration contains two large drawbacks. However, it is widely recognized that the subgraph enumeration has two essential drawbacks. One is generation of unmanageable number of patterns and the other is requirement of large computation time. To overcome the first shortcoming, efficient algorithms were proposed for the direct enumeration of several representative patterns such as maximal, closed and δ-tolerance closed subgraphs. For the second problem, parallel implementations were developed using Hadoop and GPU. In this paper,to alleviate both drawbacks in frequent subgraph discovery, we propose a GPU implementation of δ-tolerance closed subgraph mining. The effectiveness of the developed subgraph miner is evaluated using synthetic and real graph databases.

1　はじめに

頻出部分グラフマイニングとは，複数のグラフで構成されるデータベースから頻出するパターン，すなわち部分グラフを列挙するマイニング手法である．以前から化合物や文書などをはじめとする多くの実データがグラフで表現され，グラフマイニングの対象とされてきた．近年ではソーシャルネットワークを対象としたビッグデータが注目を集め，それに伴い，再びグラフマイニングの需要が高まりつつある．

これまでに幅優先探索に基づく AGM [1] や FSG [2]，深さ優先探索に基づく gSpan [3] や Gaston [4] など，頻出部分グラフを列挙するアルゴリズムが数多く提案されている．しかし，支持度のみを列挙の基準とした頻出部分グラフマイニングでは，列挙パターンの数が多すぎる点や計算時間がかかりすぎてしまう点が問題とされてきた．

列挙パターンの数が多すぎる問題を解決するために，これまで飽和パターン [5] や極大パターン [6] といった代表元のみを列挙する手法が提案された．また，一定のノイズを許容した δ-許容飽和パターン [7] といった特徴的なパターンも提案されている．

一方で，計算時間がかかりすぎる問題を解決するためには，画像処理ユニットである GPU を用いて GPGPU と呼ばれる技術で並列化を行い高速化したり [8]，複数の PC を用いた Hadoop などの MapReduce 分散処理による高速化の手法 [9–12] が提案されている．

本論文では，これら二つの問題を同時に解決するために，δ-許容飽和頻出部分グラフマイニングを GPGPU で実装する．また，人工データと実データに対し，δ パラメータに着目した実験を行った．

本論文の構成は以下のとおりである．2 章で頻出部分グラフマイニングと列挙パターンの形式的な定義を与える．3 章では提案された種々の頻出部分グラフマイニングの概要を紹介する．4 章では gSpan アルゴリズムの並列実装について紹介する．5 章で本研究である δ-許容飽和の GPGPU 実装について解説する．6 章で実験と考察を行い，最後に 7 章でまとめと今後の課題を述べる．

*連絡先：日本大学 文理学部 情報科学科
〒 156-8550 東京都世田谷区桜上水 3-25-40
tozaki@chs.nihon-u.ac.jp

2 準備

本研究では，グラフを多重辺や自己ループを許さないラベル付き連結無向グラフに限定する．

グラフ $G = (V, E, L)$ を，頂点集合 $V = \{v_1, \cdots, v_m\}$ とエッジ集合 $E \subseteq V \times V$，ラベル付け関数 $L : V \cup E \to \mathcal{L}$ の 3 項組とする．ここで $\mathcal{L}$ はラベルの集合である．

また n 個のグラフから構成されるグラフデータベースを $\mathcal{D} = \{G_1, \cdots, G_n\}$ と表記する．

グラフデータベース $\mathcal{D}$ に対し，パターン P の支持度 (support) を

$$sup_{\mathcal{D}}(P) = |\{G_i \in \mathcal{D} \mid P \subseteq G_i\}| \ / \ |D|$$

と定義する．なお $P \subseteq G$ とは G 中に P が出現する，すなわち P が G の誘導部分グラフであることを表す．

頻出部分グラフマイニングとは，データベース $\mathcal{D}$ と最小支持度 $minsup > 0$ が与えられた時，$sup_{\mathcal{D}}(P) \geq minsup$ を満たす全てのパターン（部分グラフ）を列挙する問題である．

頻出パターンマイニングがある閾値以上に頻出するパターンを列挙するものである一方，より特徴的なパターン列挙の手法として代表元を列挙する飽和パターンや極大パターンなどが提案されてきた．飽和パターンは自身の支持度と同じ支持度をもつ拡張パターンが存在しないパターンを指し，極大パターンは拡張パターンにおける極大元を集めたものである．

定義 1 (飽和パターン) 頻出パターンの集合を $\mathcal{F}$，飽和頻出パターンの集合を $\mathcal{CF}$ としたとき，以下の定義が成り立つ．

$$\mathcal{CF} = \{G \in \mathcal{F} \mid \nexists G' \in \mathcal{F} \ such \ that \ G \subset G' \ and \ sup_{\mathcal{D}}(G) = sup_{\mathcal{D}}(G')\}$$

定義 2 (極大パターン) 極大頻出パターンの集合を $\mathcal{MF}$ としたとき，以下の定義が成り立つ．

$$\mathcal{MF} = \{G \in \mathcal{F} \mid \nexists G' \in \mathcal{F} \ such \ that \ G \subset G'\}$$

さらに，ある程度のノイズを許容する δ-許容飽和パターンという手法も近年提案された．この手法では，列挙するパターンの支持度に関する制約をどの程度許容するかを表すパラメータ δ を新たに導入している．δ は 0.0〜1.0 までの実数値を取り，$\delta = 0$ の時に飽和パターン，$\delta = 1$ の時に極大パターンを求められるなど，柔軟に多様なパターンを求めることができるパラメータである．

定義 3 (δ-許容飽和パターン) δ-許容飽和パターンの集合を $\delta - \mathcal{CF}$ としたとき，以下の定義が成り立つ．

$$\delta - \mathcal{CF} = \{G \in \mathcal{F} \mid \nexists G' \in \mathcal{F} \ such \ that \ G \subset G' \ and \ sup_{\mathcal{D}}(G') \geqq (1 - \delta) \cdot sup_{\mathcal{D}}(G)\}$$

これらのパターンはいずれも頻出パターンの一部であり，パターンの包含関係は δ の値にかかわらず

$$\mathcal{MF} \subseteq \delta - \mathcal{CF} \subseteq \mathcal{CF} \subseteq \mathcal{F}$$

となる．

3 δ-許容飽和頻出部分グラフマイニング

代表的な頻出部分グラフマイニングアルゴリズムである gSpan を拡張し，飽和パターンや極大パターンを求める手法が近年 Takigawa らによって提案された [13].

δ-許容飽和パターンが提案された初期には，頻出パターンを列挙した後に δ-許容飽和パターンのみを取り出す手法が提案されていた．しかし，この後刈りの手法では一度全ての頻出パターンを列挙する必要があり，δ-許容飽和の列挙対象外となるパターンが多い場合には不必要に処理時間が増加してしまう問題が存在した．そこで，探索途中の枝刈りを効率化するために新たなパターンのマッチング手法である出現マッチ（Occurrence-Match）とトランザクションマッチ（Transaction-Match）を導入する．

本節では，まず gSpan について解説し，次に飽和頻出部分グラフマイニング，δ-許容飽和頻出部分グラフマイニングの順に解説する．

3.1 頻出部分グラフの列挙

gSpan は，コードワードを用いた標準形判定を伴う，逆探索（最右拡張）による縦型列挙アルゴリズムである．コードワードは DFS コードと呼ばれる．部分グラフパターンの全域木（DFS 木）を考え，各辺 $e = (v, u)$ を $(v, u, L(v), L(v, u), L(u))$ の 5 つ組のタプルで表し，これらを順に並べることで，部分グラフパターンの文字列表現，すなわち DFS コードとする．Algorithm 1 に，gSpan アルゴリズムの概要を示す．詳細は文献 [3] を参照されたい．

Algorithm 1 gSpan アルゴリズム [3]

dfs(Pattern P, Database $\mathcal{D}$, Threshold $minsup$)

```
1 if (! is_canonical(P)) return
2 if (sup_D(P) < minsup) return
3 output(P)
4 for q in RME(P) do
5       dfs(q, D, minsup)
6 end for
```

本アルゴリズムにおける is_canonical() 関数が標準形判定を，RME() 関数が最右拡張（RightMost Extension）の役割を担っている．

3.2 飽和頻出部分グラフの列挙

飽和頻出パターンを列挙するための枝刈りを行うために，Blanket と出現マッチ，トランザクションマッチを導入する．

あるグラフ G に対し，拡張可能な一つのエッジを追加したグラフの集合をブランケットと呼び，$B(G)$ と表記する．ここで拡張可能とは，新たにエッジを追加したグラフがデータベース中に存在することを指す．あるグラフ G に対して，G から拡張可能な一つのエッジを追加したグラフの集合を Blanket と呼び，$B(G)$ と表記する．ここで拡張可能とは，新たにエッジを追加したグラフがデータベース中に存在することを指す．$B(G)$ には，Left-Blanket と Right-Blanket の 2 種類があり，それぞれ $B_L(G)$，$B_R(G)$ と表記する．最右拡張で拡張可能な全ての集合が $B_R(G)$ であり，残りの $B(G)$ の集合が $B_L(G)$ である．

また，G と $G' \in B(G)$ について，グラフデータベース $\mathcal{D}$ 内の G が出現する全ての位置に G' が出現する場合，G と G' は出現マッチであると呼び，$OM(G, G')$ と表記する．Blanket と同様に最右拡張によるものかを基準として Right-OM と Left-OM に分けることができ，それぞれ $OM_L(G, G')$，$OM_R(G, G')$ と表記する．

一方で，あるグラフ G に対し，G と同じ支持度を持つ $B(G)$ の集合をトランザクションマッチと呼び，$TM_{\mathcal{D}}(G) = \{G' \in B(G) \mid sup_{\mathcal{D}}(G') = sup_{\mathcal{D}}(G)\}$ と表記する．

Algorithm 1 を基にして，これらを導入した飽和頻出部分グラフアルゴリズムの概要を Algorithm 2 に示す．本アルゴリズムでは，3 行目に OM による枝刈りを追加し，4 行目の出力時において TM によるチェックを追加している．

Algorithm 2 飽和頻出部分グラフアルゴリズム

closed_dfs(Pattern P, Database $\mathcal{D}$, Threshold $minsup$)

```
1 if (! is_canonical(P)) return
2 if (sup_D(P) < minsup) return
3 if (∃P' ∈ B(P) such that OM(P, P')) return
4 if (TM_D(P) ≠ ∅) output(P)
5 for q in RME(P) do
6       closed_dfs(q, D, minsup)
7 end for
```

3.3 δ-許容飽和頻出部分グラフの列挙

命題 1 (δ-許容飽和性の検査 [13]) あるグラフ G について，$B^{\delta}(G) \subseteq B(G)$ は以下によって定義される．

$$B^{\delta}(G) = \{G' \in B(G) \mid sup_{\mathcal{D}}(G') \geq max((1-\delta) \cdot sup_{\mathcal{D}}(G), minsup)\}$$

$B^{\delta}(G) = \emptyset$ のとき，グラフ G は δ-許容であるという．

本命題は，すなわち TM を δ について限定したといえる．このように 3.2 節で導入したマッチングに対して δ の制約を適応することで，δ-許容飽和頻出部分グラフの列挙が可能となる．この手法は完全性が成り立つが，詳細については文献 [13] を参照されたい．

4 gSpan アルゴリズムの並列実装

近年，Kessl らにより，GPGPU を用いた gSpan の実装が提案された [8]．本論文では，便宜上この実装を PGM と呼ぶ．PGM では，gSpan と同様の深さ優先探索に基づく列挙手法を採用している．gSpan との最たる違いとして，gSpan では新しいパターンの候補の部分グラフ同型を最初から計算するのに対し，PGM ではパターンの埋め込みリストを保存している．

4.1 データ構造

PGM では，GPU 向けの様々なデータ構造が提案された．ここで言う GPU 向けとは，GPU の持つ限られたメモリ量の中で使用するメモリを節約し，また GPU が得意とするメモリアクセス方法が適用できる形を指す．これらは GPU が得意な一次元配列を複数組み合わせて使うことで高速化を図っている．主にデータベース，埋め込み，拡張の 3 つに適したデータ構造がそれぞれ提案された．以下では，それらについて簡単に解説する．より詳細な構造については文献 [8] を参照されたい．

まず，データベース中のグラフの全ての頂点に一意な番号（*global vertex id*）を割り当てる．次に近傍を保存する配列 N を用意し，全ての頂点の近傍を頂点ごとに昇順に格納する．N は一次元配列であるので，それぞれの頂点の近傍の最初の index を別の一次元配列 O に保存する．O の添え字が *global vertex id* に対応しており，$N[O[\text{global vertex id}]]$ により，N 中の *global vertex id* の近傍がスタートするアドレスへ飛ぶことが可能である．また，*global vertex id* を i としたとき，$N[O[i]]$〜$N[O[i+1]-1]$ が i における全ての近傍であり，連続することを意味する．

Embedding は，パターンがデータベース中に現れる位置を意味する．その為，保持するデータの一部が重複することが多く，単純に保存すれば多くのメモリを

Algorithm 3 Graph Mining on GPUs [8]

GRAPH_MINING(Database $\mathcal{D}$, Threshold σ,
Pattern P, Embeddings $\sum_{\mathcal{D}}(P)$)

```
1 ( GPU step ) Get all possible edge extensions ε_D(P) of P
2 ( GPU step ) Compute support for all extensions in ε_D(P)
                and remove infrequent extensions
3 for each e = (i, j, l_i, l_(i,j), l_j) ∈ ε_D(P) do
4     P' ← P extended by e
5     if P' = minDFS(P') then
6         output P'
7         ( GPU step ) Create Σ_D(P')
8         GRAPH_MINING( D, σ, P', Σ_D(P') )
9     end if
10 end for
```

消費してしまう．そこで PGM では，パターン中の前半部分の Embedding を共通化することで，prefix tree（探索木）を構築した．そのうえで，prefix tree の各層ごとに前の層への index と頂点番号をペアで持つ $Q_i(i = 1, ..., height(tree))$ を構築し，保存する．但し，$i = 1$ の場合に限り $index = -1$ とする．例えば，(index, 頂点番号) の 2 項組で考えた場合，$Q_{height(tree)} \rightarrow Q_{height(tree)-1} \rightarrow \cdots \rightarrow Q_1$ の順に index を辿ることでパターンの再構築が可能である．

Extension では，探索中のパターンから拡張可能な頂点とエッジの情報を記録する．具体的には，DFS と from 頂点と to 頂点のグローバル頂点番号を保持する．これらを保持することで，支持度計算の際の並列化計算が容易になる．データ構造の例は 5.1 に示す．

4.2 アルゴリズム

PGM の疑似コードを Algorithm3 に示す．アルゴリズム内では，親パターン P のエッジ拡張 $\varepsilon_{\mathcal{D}}(P)$ を得ることは最右拡張に一致する．GPU が並列処理を担当するのは (GPU step) と記載された 1, 2, 7 行目の処理である．1 行目が最右拡張を得る処理であり，Algorithm 1, 2 における RME() 関数に対応している．2 行目は支持度計算に関する処理であり，GPU では複数の拡張候補の支持度を並列に計算することが可能である．7 行目では実際に拡張を行う段階で Embedding を得るための処理を行っている．

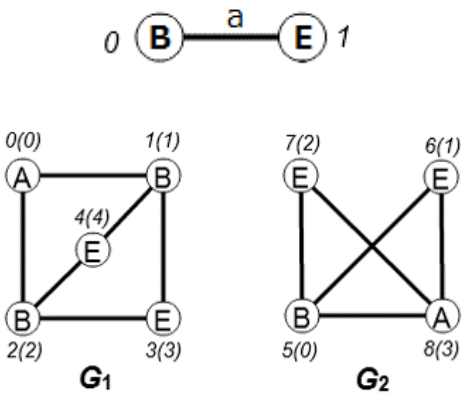


図 1: パターン B-E とグラフデータベース

k	$\mathbf{EXT_0}$					
v_i^g	1	1	2	2	5	5
v_j^g	4	3	4	3	7	6
l_j	E	E	E	E	E	E

表 1: Right-Blanket

5 δ-許容飽和頻出部分グラフマイニングの GPU 実装

我々は 3 章と 4 章の技術を組み合わせることで，GPU による δ-許容飽和頻出部分グラフマイニングを実現する方法を提案する．先行研究 [13] では，支持度の計算手法として，CPU 上で優れた性能を示す partial support 技術を提案した．しかし，GPU 上で計算する際には，多数のスレッドによる並列計算が可能であり，PGM と同様の手法を使うことが可能である．

5.1 データ構造

図 1 に示す G1 と G2 からなるグラフデータベースと，パターン P(B-E) の拡張について考える．なお，G1, G2 の頂点に付与された数字は，各頂点の global vertex id と local vertex id である．これらの Right-Blanket を 4.1 節内で解説した形式で保存すると，表 1 のような構造になる．論文 [8] に従い，from 頂点を v_i^g，to 頂点を v_j^g，to 頂点のラベルを l_j と記述する．

次に，Left-Blanket について考える．あるグラフ G が飽和であるかどうかをチェックするには，Left-Blanket を利用する必要がある．Left-Blanket は，実際の拡張には必要ないが，OM と TM を計算する為に必要なため，4.1 節で述べた Extension の形で保存する必要があ

k	$\mathbf{EXT_0}$												$\mathbf{EXT_1}$					
v_i^g	1	1	1	1	2	2	2	2	5	5	5	5	3	4	3	4	6	7
v_j^g	0	4	0	3	0	4	0	3	7	8	6	8	2	2	1	1	8	8
l_j	A	E	A	E	A	E	A	E	E	A	E	A	B	B	B	B	A	A

表 2: Left-Blanket：Speed 重視 (Right-Blanket を含む)

k	$\mathbf{EXT_0}$												$\mathbf{EXT_1}$					
v_i^g	1	1	1	1	2	2	2	2	5	5	5	5	3	4	3	4	6	7
v_j^g	0	4	0	3	0	4	0	3	7	8	6	8	2	2	1	1	8	8
l_j	A	E	A	E	A	E	A	E	E	A	E	A	B	B	B	B	A	A
$L/R\ Blanket$	L	R	L	R	L	R	L	R	R	L	R	L	L	L	L	L	L	L

表 3: Left-Blanket：Memory 重視

る．GPU 上の実装では Right-Blanket の場合と同様に Extension の保存や支持の計算が可能である．

単純に Left-Blanket 用のデータ構造を別に持つと，それは Right-Blanket の場合と同様の並列計算を適用できる為，容易に高速化が可能となる．速度重視の場合の Left-Blanket のデータ構造を表 2 に示す．

しかし，GPU ではメモリが少ないことから，単純に Left-Blanket 用のデータを増やすと Right-Blanket とのデータの重複が発生してしまい，メモリ不足により非常に大きなデータベースに対する計算が出来なくなってしまう．データの重複とメモリ不足を避けるために，本論文では，Right-Blanket と Left-Blanket のデータ構造を一体化したデータ構造を提案する．

具体的には，新たに Left か Right かを判定する変数の配列を用意し，Extension の保存時に同時に保存する．一体化したデータ構造を表 3 に示す．これらの工程は確かにメモリ量を減らすことが出来るが，一方で各 EXT ごとに並列処理を試みたときに Left/Right Blanket でそれぞれ異なる処理，すなわち GPU が不得意とする分岐処理を増やすことになってしまう．どの程度の影響が出るかは，比較実験を通じて評価する．

5.2 出現マッチの確認方法

PGM では，Embedding のデータ構造を prefix tree の形で保持している．図 1 における例を表 4 に示す．なお，表 2,3 における Extension の情報と表 4 との間で，4.1 節で述べたように Q の後方から順にマッチングを取っていくことで，出現位置が同じかどうかを確認することが可能である．

	Q_0			Q_1					
idx	-1	-1	-1	0	0	1	1	2	2
vid	1	2	5	3	4	3	4	6	7

表 4: Embedding のデータ構造

6 実験と考察

6.1 実験環境とデータセット

実験には，3072 コアと 12GB のメモリを持ち，CUDA compute capability 5.2 をサポートする GeForce GTX TITAN X を使い，Windows 10 上で実験を行った．開発ツールには Nvidia 社が提供する CUDA 8.0 を利用した．

本研究では，実験のために 2 つの実データセットと 3 つの人工データセットを用意した．人工データセットは GephGen[1] を用いて生成し，グラフ数，エッジの平均数，エッジラベルの数，density のパラメータについて調整を行った．実データセットでは，DragBank[2] と PTC[3] のデータを用意した．各データセットの基本的な統計量を表 5 に示す．

Parameter	syn_1	DrugBank	PTC
# of graphs	5000K	976	417
Avg.# of edges	100	27	15
# of edge labels	20	34	21
Avg. density	0.5	0.1	0.1

表 5: 各データセットの統計量

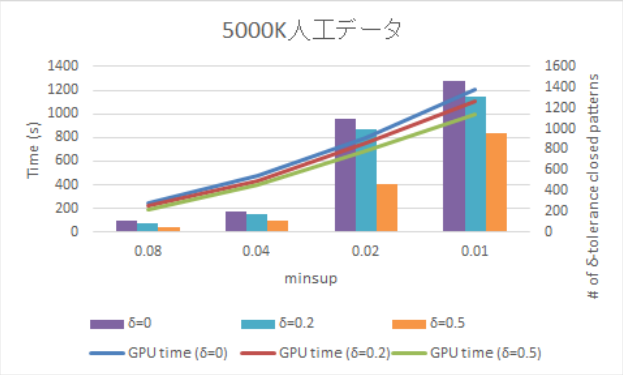
[1] http://www.cse.ust.hk/graphgen/
[2] https://www.drugbank.ca/
[3] http://www.predictive-toxicology.org/ptc/

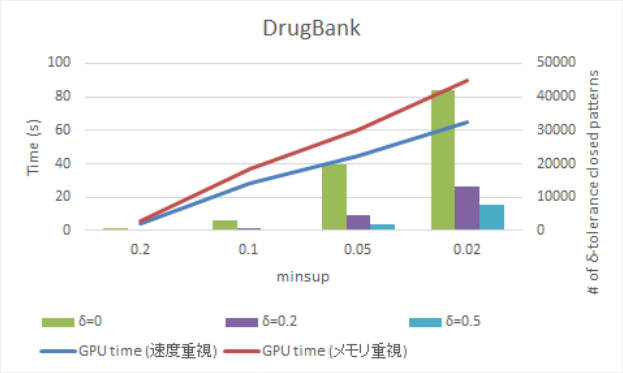
6.2 実験結果

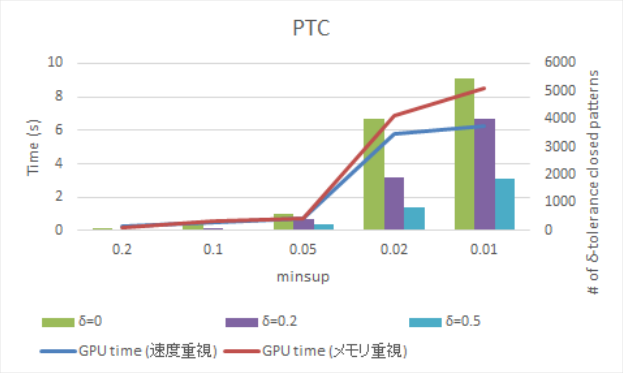
人工データに対して，minsup 及びグラフ数を変化させた場合と，実データに対する $\delta \in \{0, 0.2, 0.5\}$ による実験を図 2 に示す．

図中の棒グラフは列挙パターン数を表し，右側の縦ラベルに対応する．折れ線グラフは計算時間（GPU time）を表し，左側の縦ラベルに秒単位で記している．実データの計算時間は速度重視とメモリ重視の実装の違いによるものであるが，5000K 人工データについては如何なる最小支持度の場合でも速度重視のプログラムがメモリ不足を引き起こしてしまう為，δ の値ごとの計算時間を記載している．

図 2: 速度重視とメモリ重視による実験結果







6.3 考察

δ が大きくなるにつれて制約が大きくなるので，どの結果においても列挙パターン数が減少していることが見て取れる．500K 人工データにおいて，パターン数が減少するにつれて劇的というほどではないが，確かに計算時間も減少している．

また，実データにおいては全体的にメモリ重視の実装よりも速度重視の実装の方が約 1.3〜1.5 倍高速であった．いずれの場合にも実行時間が短い場合には差が殆ど見られないが，長くなるほど差が顕著になった．

7 まとめと今後の課題

我々は，δ-許容飽和頻出部分グラフマイニングアルゴリズムの GPU 実装を行った．単純に出現マッチとトランザクションマッチを導入しただけの実装ではグラフ数が多い場合に解が求められない場合が存在した．そこで，データ構造の改善を行い，メモリ重視の実装を行ったことで，多少の速度を犠牲にしてもグラフ数が多い場合にも解が求まるようになった．

今後は，単一グラフ [14, 15] や Approximate 頻出パターン [16] などのマイニングアルゴリズムに対して GPGPU 実装のデータ構造を検討する．

参考文献

[1] A. Inokuchi, T. Washio and H. Motoda: An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. *Proc. PKDD2000, Lyon, France.* 2000.

[2] M. Kuramochi and G. Karypis: Frequent Subgraph Discovery. *Proc. of the 2001 IEEE International Conference on Data Mining, pp.313-320*, 2001.

[3] X. Yan and J. Han: gSpan: Graph-Based Substructure Pattern Mining. *Proc. 2002 of Int. Conf. on Data Mining (ICDM'02)*, 2002.

[4] S. Nijssen and J. N. Kok : A quickstart in frequent structure mining can make a difference. *Proc. of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining : pp.647-652*, 2004.

[5] X. Yan and J. Han: CloseGraph: Mining Closed Frequent Graph Patterns. *Proc. of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.286-295*, 2003.

[6] J. Huan, W. Wang, J. Prins and J. Yang: SPIN: Mining Maximal Frequent Subgraphs from Graph Databases. *Proc. of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, pp.581-586*, 2004

[7] J. Cheng, Y. Ke, W. Ng and A.Lu FG-Index: Towards Verification-Free Query Processing on Graph Databases. *Proc. of the 2007 ACM SIGMOD international conference on Management of data, pp.857-872*, 2007.

[8] R. Kessl, N. Talukder, P. Anchuri and M. Zaki: Parallel Graph Mining with GPUs. *JMLR Workshop and Conference Proceedings 36 : 116*, 2014.

[9] M. A. Bhuiyan and M. A. Hasan: An iterative MapReduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering, Vol.27, Issue 3, pp.608-620*, 2015.

[10] W. Lin, X. Xiao and G. Ghinita: Large-scale frequent subgraph mining in MapReduce. *Proc. of 2014 IEEE 30th International Conference on Data Engineering (ICDE), pp.644-855*, 2014.

[11] F. N. Afrati, D. Fotakis, and J. D. Ullman: Enumerating subgraph instances using map-reduce. *Proc. of 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp.62-73*, 2013.

[12] S. Hill, B. Srichandan and R. Sunderraman: An Iterative MapReduce Approach to Frequent Subgraph Mining in Biological Datasets. *Proc. of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine pp.661-666*, 2012.

[13] I. Takigawa and H. Mamitsuka: Efficiently mining δ-tolerance closed frequent subgraphs. *Machine Learning, Vol.82, Issue 2, pp 95-121*, 2011.

[14] B. Bringmann and S. Nijssen: What is frequent in a single graph? *Proc. of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, pp.858-863*, 2008.

[15] M. Elseidy, E. Abdelhamid, S. Skiadopoulos and P. Kalnis: GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph. *Proc. of the VLDB Endowment, Vol.7, No.7, pp.517-528*, 2014.

[16] R. Li and W. Wang: REAFUM: Representative Approximate Frequent Subgraph Mining. *Proc. of the 2015 SIAM International Conference on Data Mining, pp.757-765*, 2015.