creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

자기 동형 사상을 활용한 Minimum Image Based Support 계산

2023년 8월

서울대학교 공과대학

컴퓨터공학부

이연준

자기 동형 사상을 활용한 Minimum Image Based Support 계산

지도교수 박 근 수

이 논문을 공학석사 학위논문으로 제출함
2023 년 6 월

서울대학교 대학원
컴퓨터 공학부
이 연 준

이연준의 석사 학위논문을 인준함
2023 년 7 월

위 원 장 _____ 문 병 로 _____ (인)

부위원장 _____ 박 근 수 _____ (인)

위 원 _____ 송 용 수 _____ (인)

초 록

그래프 마이닝에서 핵심적인 문제 중 하나는 frequent subgraph mining 문제이다. Frequent subgraph mining 문제는 데이터 그래프에서 정해진 support가 가장 큰 부분 그래프들을 찾는 문제이다. 이 때, support는 그래프에서 부분 그래프가 얼마나 자주 등장하는지를 나타내는 지표이다. Minimum Image Based Support (MNI)는 frequent subgraph mining에서 자주 사용되는 support이다. 본 논문에서는 쿼리 그래프의 자기 동형 사상(Automorphism)을 이용하여 MNI의 계산을 효율적으로 계산하는 방법을 제시한다. 추가적으로, MNI 계산 중 백트래킹 과정을 병렬화 하여서 멀티코어 환경에서의 성능을 개선하였다. 또한, 실제 데이터 그래프를 통해 실험을 진행하여 자기 동형 사상과 병렬화를 사용하는 것이 기존의 방법에 비해 효율적임을 입증하였다.

주요어 : 그래프 마이닝, Frequent Subgraph Mining, 자기 동형 사상

목 차

초 록 **i**

목 차 **1**

1 서론 **2**

2 배경 지식 **3**

3 MNI 계산의 개선 **5**

4 실험 설명 및 결과 **11**

5 결론 **20**

Abstract **22**

제 1 장 서론

최근 그래프 마이닝은 소셜 네트워크 검색 [1], 단백질-단백질 상호작용 분석 [2], 네트워크 보안 [3] 등 다양한 분야에서 이용된다. Frequent subgraph mining 문제는 그래프 마이닝에서 가장 핵심적인 문제 중 하나이다 [4]–[6]. Frequent subgraph mining은 데이터 그래프가 주어졌을 때 정해진 support가 가장 큰 그래프들을 찾는 문제이다. 이때, support는 그래프에서 어떤 부분 그래프가 얼마나 자주 등장하는지를 나타내는 지표이다.

Minimum Image Based Support (MNI) [7], [8]는 Frequent subgraph mining에서 가장 많이 사용되는 support로, 패턴 그래프의 정점 중 데이터 그래프에서 동형으로 매칭될 수 있는 정점이 가장 적은 것의 개수로 정의된다. 데이터 그래프와 패턴 그래프가 주어졌을 때, 데이터 그래프에 대한 패턴 그래프의 MNI를 계산하는 것은 패턴 그래프의 한 정점을 데이터 그래프의 각 정점에 대응시킨 후 부분 그래프 동형을 찾아야 하므로 NP-Hard 문제 중 하나이다.

본 논문에서는 패턴 그래프의 자기 동형 사상(Automorphism)을 이용하여 MNI를 계산하는 방법을 제안한다. 또한, 멀티코어 환경에서 성능을 향상시키기 위해 MNI 계산의 백트래킹 과정을 병렬화 하였다. 실제 그래프 데이터에서 기존의 MNI 계산 방법과의 비교 실험을 통해 제안한 방법의 성능을 입증하였다.

본 논문의 구성은 다음과 같다. 2장에서는 논문에 필요한 용어들을 정의하고, MNI와 자기 동형 사상에 대해 설명하고 3장에서는 MNI 개선을 개선하는 방법에 대해 기술한다. 4장에서는 실험 설명 및 결과와 그 분석을 보여준다. 5장에서는 결론으로 마무리한다.

제 2 장 배경 지식

본 문제에서는 정점에 레이블이 존재하는 무향, 연결 그래프에 대해 다룬다. 그래프 $G = (V_G, E_G, L_G)$는 V_G를 정점의 집합, E_G를 간선의 집합으로 하고, 정점 집합을 레이블의 집합 Σ로 대응시키는 레이블 함수 $L_G : V_G \rightarrow \Sigma$를 가지는 그래프이다. 그래프 $q = (V_q, E_q, L_q)$와 그래프 $G = (V_G, E_G, L_G)$가 주어졌을 때, G 안의 q의 임베딩 (embedding)은, 다음 세 조건을 만족하는 매핑 $f : V_q \rightarrow V_G$ 로 정의된다.

1. M은 단사함수(injective)이다. ($f(u) \neq f(u')$ for $u \neq u'$)

2. 모든 $u \in V_q$에 대해, $L_q(u) = L_G(f(u))$

3. 모든 $(u, u') \in E_q$ 에 대해, $(f(u), f(u')) \in E_G$

또한, 그래프 G안에 q의 임베딩이 존재할 때, q가 G의 부분 그래프 동형(subgraph isomorphism)이라고 정의한다.

2.1 Minimum Image Based Support

데이터 그래프 $G = (V_G, E_G, L_G)$와 쿼리 그래프 $q = (V_q, E_q, L_q)$가 주어졌을 때, G 안의 q의 모든 임베딩의 집합이 $f_1, \cdots, f_m$ 이라고 하자. 정점 $v \in v_q$에 대해서, $F(v) = f_1(v), \cdots, f_m(v)$는 v를 G에 대응하는 모든 G의 정점들의 집합이다. 이때, G에 대한 q의 minimum image based support $MNI_G(S)$는,

$$MNI_G(q) = \min_{v \in V_q} \{t \mid t = |F(v)|\}$$

로 정의된다.

예를 들어, 그림 1에서 데이터 그래프 G에서 쿼리 그래프 S의 모든 임베딩은 $\{(u_1, v_1), (u_2, v_6), (u_3, v_4), (u_4, v_2), (u_5, v_8)\}, \{(u_1, v_1), (u_2, v_6), (u_3, v_4), (u_4, v_2), (u_5, v_9)\}, \{(u_1, v_2), (u_2, v_6), (u_3, v_4), (u_4, v_1), (u_5, v_8)\}, \{(u_1, v_2), (u_2, v_6), (u_3, v_4), (u_4, v_1), (u_5, v_9)\}$ 의 4개이다. 이 때, 쿼리 그래프 S의 각 정점 u의 $F(u)$는, $F(u_1) = \{v_1, v_2\}$, $F(u_2) = \{v_6\}$, $F(u_3) = \{v_4\}$, $F(u_4) = \{v_8, v_9\}$ 이다.

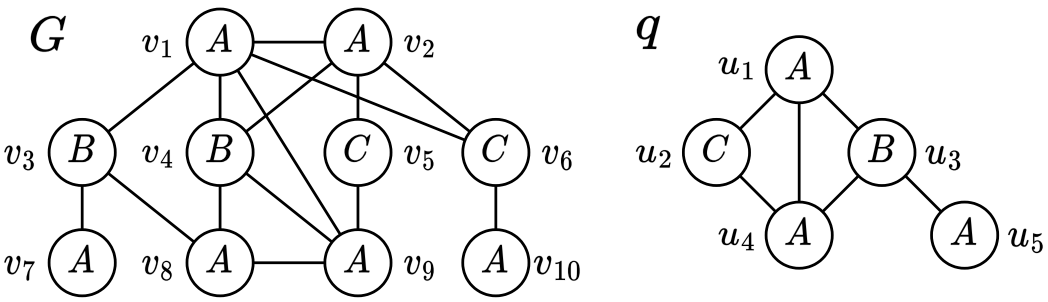


그림 1: Exmaple for computing the MNI

Symbol	Definition
G	데이터 그래프
q	쿼리 그래프
$CS(u)$	$u \in V_q$에 대한 candidate vertices의 집합
$MNI_G(q)$	G에 대한 q의 MNI
$Aut(u)$	q에서 $u \in V_q$의 동치류
$Marked[u][v]$	$u \in V_q$를 $v \in V_G$에 대응시키는 임베딩이 있는지 여부

표 1: Notations frequently used in this paper

따라서 G에 대한 S의 MNI는,

$$MNI_G(S) = \min_{v \in V_S} \{t \mid t = |F(v)|\} = 1$$

이다.

2.2 자기 동형 사상 (Automorphism)

자기 동형 사상 (Automorphism)은 그래프 G에서 자기 자신으로의 동형을 의미한다. G의 두 정점 v, w에 대해 어떤 자기 동형 사상 T가 존재하여 v를 w로 대응할 수 있다면, v와 w를 동치 (equivalent)라고 정의한다. 그래프 G의 정점들을 서로 동치인 것들로 나눈 것을 G의 동치류 (equivalence class)라고 부른다. 예를 들어, 그림 1의 쿼리 그래프 q는 자기 동형 사상으로 $\{(u_1, u_4), (u_2, u_2), (u_3, u_3), (u_4, u_1), (u_5, u_5)\}$을 가지기 때문에, 정점 u_1과 u_4는 동치이고, 같은 동치류에 포함된다. 표 1은 자주 사용하는 정의들을 나타낸다.

제 3 장 MNI 계산의 개선

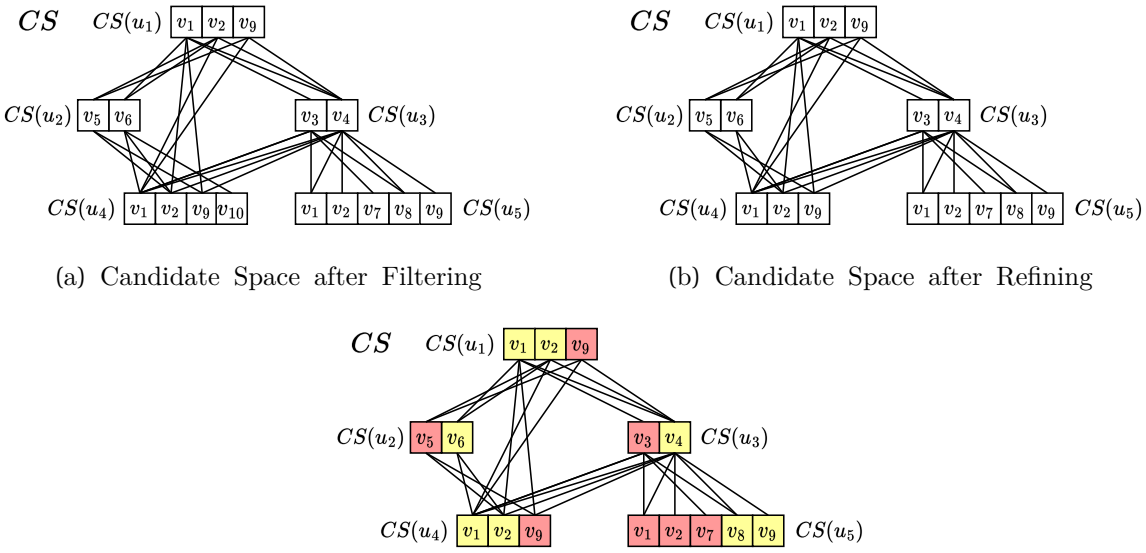
3.1 알고리즘의 개요

데이터 그래프 G와 쿼리 그래프 q가 주어졌을 때, q의 G에 대한 MNI를 계산하기 위해서는 $u \in V_q, v \in V_G$인 각각의 (u, v) 쌍에 대해, u를 v에 대응하는 임베딩이 있는지 계산하여야 한다. 이 때, 쿼리 그래프의 자기 동형 사상을 이용한다면, 하나의 (u, v) 쌍의 대한 동형으로 다른 (u', v') 쌍의 동형이 존재하는지를 알 수 있다. Algorithm 1은 자기 동형 사상을 이용한 MNI 계산의 전체 의사코드를 표현한다.

먼저, 그래프의 자기 동형 사상을 계산하는 알고리즘인 Traces [9]를 사용하여 쿼리 그래프 q의 모든 동치류를 계산한다 (Line 1). 이 때, 동치류의 결과는 정점 집합의 배열로서 Aut 저장된다. 예를 들어, 그림 1에서 쿼리 그래프 q의 동치류는 $[\{u_1, u_4\}, \{u_2\}, \{u_3\}, \{u_5\}]$로 저장된다.

그 후, 부분 그래프 동형 알고리즘 VEQ [10]를 사용하여서 쿼리 그래프 q에 대한 데이터 그래프 G의 초기 후보 공간 (initial candidate space)를 구축하고, 필터링을 통해 후보공간을 줄인다 (Line 2). 그림 2의 (a)는 그림 1의 쿼리 그래프 q에 대한 데이터 그래프 G의 VEQ의 필터링 후의 후보 공간을 나타낸다. VEQ의 필터링은 쿼리 그래프 q의 DAG (directed acyclic graph) q_D와 그의 inverse DAG q_D^-1을 사용하여 top-down, bottom-up, top-down의 세 번의 필터링만을 시행하기 때문에, 같은 동치류인 데이터 그래프 정점의 후보 공간이 동일함을 보장하지 않는다. 예시에서, 쿼리 그래프 q에 대한 데이터 그래프 G의 대한 초기 후보 공간에서 VEQ의 필터링을 거쳤을 때, 쿼리 그래프 q의 정점 u_1 과 u_4는 동치이지만, 두 정점의 후보공간 $CS(u_1)$과 $CS(u_4)$는 일치하지 않는다.

동치류인 쿼리 그래프 정점의 후보공간들이 모두 일치한다면, (u, v)에 대한 임베딩의 존재 여부를 기록한 후, 동치인 쿼리 그래프 정점 u'에 대해 (u', v)에 $O(1)$시간에 접근할 수 있다. 이를 위해, 본 알고리즘에서는 같은 동치류의 후보 공간을 동일하게 맞추는 작업을 진행한다 (Line 5-10). 같은 동치류인 정점에 대해 가장 크기가 작은 후보 공간을 그 동치류 정점들의 후보 공간으로서 통일한다. 모든 후보 공간은 필터링 이후에도 임베딩이 존재하는



(a) Candidate Space after Filtering

(b) Candidate Space after Refining

(c) Candidate Space after Marking

그림 2: Candidate Space in Algorithm

데이터 그래프 정점들을 포함하기 때문에, 동치인 정점의 후보 공간으로 교체하여도, 실제로 임베딩이 존재하는 모든 데이터 그래프 정점들을 포함한다는 것을 보장한다. 그림 2의 (b)는 후보 공간을 동치류 별로 일치 시킨 후의 결과를 나타낸다. 예시에서 $CS(u_1)$과 $CS(u_4)$는 둘 줄 크기가 작은 $CS(u_4)$로 바뀌게 된다.

이어서, 각 후보 공간의 정점 순서쌍 (u, v)에 대해, 쿼리 그래프 u를 데이터 그래프 v에 대응하는 동형이 존재하는지 여부를 VEQ의 백트래킹을 통해 계산하고, 그 결과를 자기 동형 사상을 사용하여 $Marked[u][v]$에 기록한다 (Line 11). $Marked[u][v]$는 아직 방문하지 않은 순서쌍에 대해 0의 값을 가지고, 쿼리 그래프 u를 데이터 그래프 v에 대응하는 동형이 존재한다면 1, 존재하지 않는다면 -1로 기록된다. 후보 공간의 모든 쌍에 대해 기록이 완료되면, 이를 통해 MNI를 계산한다 (Line 12). 그림 2의 (c)는 기록이 끝난 후의 후보 공간을 나타낸다. 각 정점 순서쌍 (u, v)에 대해, $Marked[u][v] = 1$ 이면 노란색, $Marked[u][v] = -1$ 이면 빨간색으로 표시하였다.

Algorithm 1: Computing MNI

Input : Data graph G, Query graph q

Output: $MNI_G(q)$

```
Aut ← FindEquivalentClass(q)                    /* Traces */
CS ← FilterCS(G, q)                             /* VEQ Filtering */
Visited[1..V_q] ← false
Marked ← 0
for u in V_q do
    if not Visited[u] then
        minIdx = argmin_{u'∈Aut(u)} |CS(u')|
        for u' in Aut(u) do
            CS(u') ← CS(minIdx)
            Visited[u'] ← true
MarkingCS(G, q, CS, Aut)
MNI ← min_{u∈V_q} |{v | Marked[u][v] = 1}|
return MNI
```

3.2 자기 동형 사상의 응용

쿼리 그래프의 두 정점 u_1과 u_2가 서로 동치일 때, 만약 u_1을 데이터 그래프의 정점 v로 대응하는 동형이 존재한다면, u_2를 v로 대응하는 동형 또한 존재할 것이다. 반대로, u_1을 v로 대응하는 동형이 존재하지 않는다면, u_2를 v로 대응하는 동형 또한 존재하지 않을 것이다. 따라서, 쿼리 그래프의 동치류를 미리 계산한다면, 같은 동치류에 포함되는 쿼리 그래프의 정점들은 그래프 동형의 존재 여부를 한 번에 계산할 수 있을 것이다.

Algorithm 2는 Algorithm 1의 $MarkingCS$의 의사코드이다. VEQ를 통해 필터링된 후보 공간에서 쿼리 그래프의 정점 u와 데이터 그래프의 정점 v 쌍 (u, v)에 대해 u를 v에 대응시키는 임베딩이 있는지 VEQ의 백트래킹 기법을 사용하여 찾는다. *Backtrack* 함수는

데이터 그래프 G, 쿼리 그래프 q, 쿼리 그래프 정점 u, 데이터 그래프 정점 v를 입력으로 받아서 임베딩이 존재한다면 참을 반환하고, 그 임베딩을 f에 저장한다. 임베딩이 존재하지 않는다면, 거짓을 반환한다. (Line 4).

임베딩 f가 존재한다면, 각 쿼리 그래프 정점 t에 대해, t와 동치인 $t' \in Aut(t)$에 대해, $(t', f(t))$을 대응했을 때 임베딩이 존재하므로, 모든 t, t'에 대해, $Marked[t'][f(t)]$를 1로 기록할 수 있다 (Line 5-7). 그림 2의 (c)에서 (u_1, v_1)을 고정했을 때, 임베딩 $\{(u_1, v_1), (u_2, v_6), (u_3, v_4), (u_4, v_2), (u_5, v_8)\}$을 찾을 수 있고, 이 때, $(u_1, v_1), (u_2, v_6), (u_3, v_4), (u_4, u_2)$와, 추가로 u_1 과 u_4가 동치이기 때문에 (u_4, v_1) 또한 $Marked$ 값을 1로 기록할 수 있다.

반면에, 임베딩이 존재하지 않는다면, $u' \in Aut(u)$에 대해 (u', v)또한 정점 쌍을 대응했을 때 임베딩이 존재하지 않는다는 것을 알 수 있다. 따라서, 모든 $u' \in Aut(u)$에 대해, $Marked[u'][v]$를 -1로 기록한다 (Line 8-10). 그림 2의 (c)에서 (u_1, v_9)을 고정했을 때, 임베딩이 없다는 것을 찾았을 때, u_1 과 u_4가 동치이기 때문에 (u_4, v_9) 또한 기록할 수 있다.

3.3 병렬화를 통한 MNI 계산의 개선

Algorithm 2의 반복문 (Line 1-2)은 모두 병렬적으로 진행이 가능한 연산들이다. 병렬화가 되었을 때, 서로 다른 쓰레드에서 같은 $Marked[u][v]$에 접근하여 기록하는 상황이 발생할 수 있으나, 항상 쓰레드들이 기록하고자 하는 답은 동일할 것이므로, 후보 공간의 모든 (u, v) 순서쌍들이 기록된 후에 그 결과는 병렬화를 하기 전과 동일한 것을 보장한다. 각 쓰레드에서 백트래킹을 개별적으로 하게 병렬화 하였다. 또한, 백트래킹에 필요한 변수들과 자료구조들을 각 쓰레드 별로 저장하게 하여서 쓰레드에서 다른 쓰레드에서 진행하고 있는 백트래킹 변수에 접근하는 일이 없도록 하였다. Algorithm 3는 병렬화된 $MarkingCS$의 의사코드를 나타낸다.

Algorithm 2: *MarkingCS*

Input : Data graph G, Query graph q, Filtered Candidate Space CS, Equivalence Class Aut

```
for u in V_q do
    for v in CS(u) do
        if Marked[u][v] is unmarked then
            if Backtrack(G, q, u, v, f) then
                /* When Backtrack finds an embedding, it returns isomorphism
                in f */
                for t in V_q do
                    for t' in Aut(t) do
                        Marked[t'][f(t)] ← 1          /* (t', f(t)) has embedding */
            else
                for u' in Aut(u) do
                    Marked[u'][v] ← −1                /* (u', v) has no embedding */
```

Algorithm 3: $MarkingCS(Parallel)$

Input : Data graph G, Query graph q, Filtered Candidate Space CS, Equivalence Class Aut

```
do in parallel
    for u in V_Q do
        for v in CS(u) do
            if Marked[u][v] is unmarked then
                if Backtrack(G, q, u, v, f) then
                                /* When Backtrack finds an embedding, it returns
                    isomorphism in f */
                    for t in V_q do
                        for t' in Aut(t) do
                            Marked[t'][f(t)] ← 1          /* (t', f(t)) has embedding */
                else
                    for u' in Aut(u) do
                        Marked[u'][v] ← −1            /* (u', v) has no embedding */
```

제 4 장 실험 설명 및 결과

4.1 실험 개요

본 연구의 실험에서는 세 가지 알고리즘을 비교한다. VEQ만을 사용하여 MNI를 계산하는 알고리즘 (Baseline)과, 자기 동형 사상을 이용하여 MNI를 계산하는 알고리즘 (Automorphism), 자기 동형 사상과 병렬화를 모두 사용한 알고리즘 (Parallel)을 사용한다. 모든 알고리즘에서 그래프를 읽는 입력시간과 VEQ의 필터링 시간은 동일하기 때문에, 이를 제외한 VEQ의 백트래킹 시간과 MNI를 계산하는 시간만을 비교하였다. 실험 데이터 그래프로는 frequent subgraph mining 문제에서 주로 사용된 3개의 실험 그래프 MiCo, Yeast, Patent를 이용하였다. 사용된 그래프의 특성은 표 2과 같다. 표 2에서 $|V(G)|$는 정점의 개수, $|E(G)|$는 간선의 개수, $|\Sigma|$는 레이블의 개수를 의미한다.

- Baseline : VEQ 백트래킹
- Automorphism : VEQ 백트래킹 + 자기 동형 사상
- Parallel : VEQ 백트래킹 + 자기 동형 사상 + 병렬화 (24 thread)

본 연구의 실험은 두 가지로 구성된다. 먼저, MNI 계산을 주로 frequent subgraph mining에서 알고리즘의 효율성을 측정하기 위해, 실험 1은 데이터 그래프의 부분 그래프 중 MNI가 가장 큰 k개의 그래프의 MNI를 계산하는 시간을 비교하는 실험이다. Frequent subgraph mining 알고리즘을 사용하여 데이터 그래프의 부분 그래프 중 MNI가 가장 큰 k개의 부분 그래프를 미리 계산한 후, 그 그래프들을 쿼리 그래프로 하여 MNI를 계산하는 시간의 평균 시간을 측정하였다. 실험 2는 쿼리 그래프의 형태에 따른 MNI의 계산 시간의 비교로, star, clique, tree 그래프에 대해 실험을 진행하였다. 각 쿼리 그래프의 크기를 변화시키면서 자기 동형 사상과 병렬화의 효율을 측정하였다. 모든 실험은 Intel Xeon E5-2680 v3 2.50GHz CPU 256GB 와 메모리를 가진 CentOS Linux 7 환경에서 진행되었다. 병렬화 알고리즘 Parallel의 경우, 쓰레드 갯수를 24개로 하여서 실험을 진행하였다.

Dataset	$\|V(G)\|$	$\|E(G)\|$	Σ	k	$\|Q\|$	$Avg(Aut)$
MiCo	100,000	422,402	29	50	31	1.92
Yeast	3,112	12,519	71	50	34	2.96
Patent	2,938,754	13,965,411	419	15	15	1.89

표 2: Statistics of Data Graphs

4.2 Top-k 그래프에 대한 실험

4.2.1 실험 설명

본 실험은 데이터 그래프의 부분 그래프 중 MNI가 가장 큰 k개의 MNI를 계산하는 시간을 비교한다. MiCo와 Yeast는 상위 50개의 부분 그래프를, Patent의 경우 상위 15개의 부분 그래프를 쿼리 그래프로 사용하였다. Speedup의 비교를 위해 Baseline 알고리즘과 자기 동형 사상을 사용한 알고리즘 양쪽에 대해 모두 제한 시간 30분 안에 풀어낸 쿼리에 대해 비교를 진행하였다. 표 1의 k, $|Q|$는 각각 데이터 그래프에 대해 k값과 제한 시간 안에 풀어낸 쿼리의 개수를 보여준다. $Avg(Aut)$는 풀어낸 쿼리 그래프들에 대해 정점 개수를 동치류의 개수로 나눈 값의 평균을 나타낸다. 이 값이 클수록 자기 동형 사상을 사용하여 더 빠르게 MNI를 계산할 것으로 기대할 수 있다.

4.2.2 실험 결과

그림 3는 각 데이터 그래프마다 자기 동형 사상을 이용한 MNI 계산 알고리즘 (Automorphism)과 사용하지 않은 알고리즘 (Baseline), 그리고 병렬화 알고리즘(Parallel)의 Top-k 그래프의 평균 MNI 계산 시간을 보여준다.

표 2에서, MiCo와 Patent의 경우 제한 시간 내에 풀어낸 쿼리 그래프에서 평균 동치류의 크기가 크지 않아 자기 동형 사상의 효과가 크지 않은 것을 볼 수 있다. 반면에, 동치류의 크기가 큰 Yeast 데이터 그래프의 경우에는 자기 동형 사상을 사용할 경우 Baseline 알고리즘과 비교하여 빠른 속도로 MNI를 계산할 수 있는 것을 볼 수 있다. 모든 데이터에 대해, 풀어내지 못한 쿼리는 모두 cycle 그래프이다. Cycle 그래프란, n개의 쿼리 그래프 정점

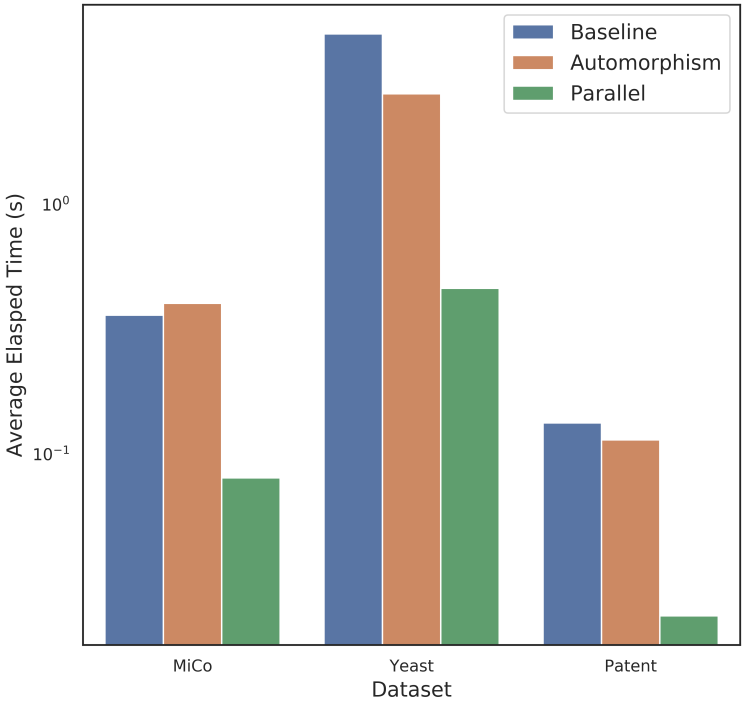


그림 3: Average MNI computation Time on Top-k Graph

$v_1, \cdots, v_n$이 각각 모든 i에 대해, v_i 와 $v_i + 1$이 연결되어 있고, v_n과 v_1이 연결되어 있는 순환 그래프를 의미한다. 정점이 n개이고, 모든 정점의 레이블이 동일한 쿼리 그래프에 대해, cycle 그래프의 모든 정점은 서로 동치이기 때문에, 자기 동형 사상을 통한 MNI 계산이 빨라질 것으로 기대할 수 있다. 실제로는, 대부분 Baseline 알고리즘과 Automorphism 알고리즘 모두 제한시간 내에 해결하지 못하지만, 몇몇 경우에 대해서는 Automorphism 알고리즘만 제한 시간 내에 정답을 출력하는 경우가 존재하였다.

병렬화 알고리즘인 Parallel 알고리즘의 경우, 한개의 쓰레드만을 사용하는 Automorphism 알고리즘에 비해 데이터에 따라 5-10배 정도 빨라지는 것을 볼 수 있었다. 병렬화를 위해 백트래킹에 필요한 자료구조들을 할당하고 초기화 하는 과정이 쓰레드의 갯수만큼 더 소요되는 것이 병렬화의 overhead로 작용하였다고 분석할 수 있다. 또한, 데이터셋에 따라 서로 다른 쓰레드에서 쿼리 그래프의 정점 u와 데이터 그래프의 정점 v 쌍 (u, v)의 $Marked$ 값인 $Marked[u][v]$에 접근할 가능성이 존재한다. 이러한 경우 단일 쓰레드에서는 한 번 진행하였

을 작업이 중복으로 진행되었기 때문에, 이 역시 overhead로서 작용할 수 있다.

4.3 쿼리 그래프의 형태에 따른 실험

4.3.1 실험 설명

본 실험은 쿼리 그래프의 형태에 따른 MNI 계산 시간의 차이를 비교한다. 쿼리 그래프로는 clique, star, tree 그래프로 세 가지 형태에 대해 크기를 변화시키며 실험을 진행하였다.

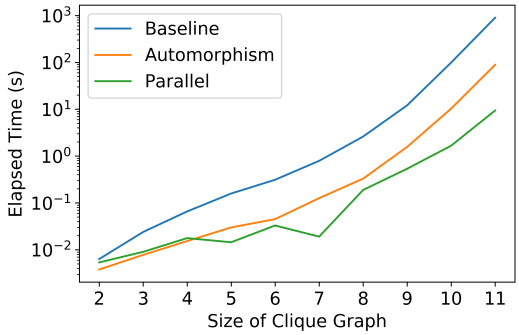
Clique 그래프의 경우 데이터 그래프에서 가장 빈번하게 등장하는 상위 5개의 레이블을 노드들의 레이블로 지정하였다. Clique 그래프의 크기를 2-11로 하여 실험하였다.

Star 그래프의 경우 데이터 그래프에서 가장 빈번하게 등장하는 레이블 L_f를 말단 노드들의 레이블로 하고, 레이블이 L_f인 데이터 그래프 정점들과 가장 많이 연결된 상위 5개의 레이블을 중심 노드의 레이블로 하였다. 각 레이블에 대해 말단 노드가 5 - 50개 달린 star 그래프를 쿼리 그래프로서 실험하였다.

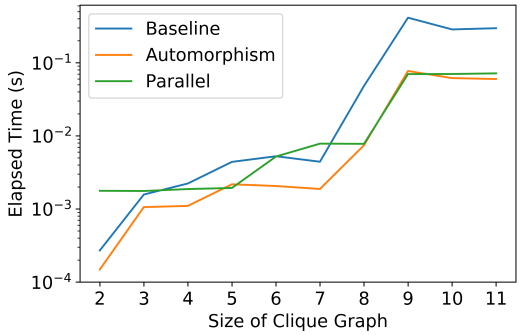
Tree 그래프의 경우 데이터 그래프에서 random walk를 수행하여 방문한 정점들과 그 간선들을 추출하여 생성하였다. 임의의 데이터 그래프 정점으로부터, 그 정점의 이웃 정점들중 하나를 무작위로 방문하고, 이 때 방문하지 않은 정점이라면 방문한 정점들과의 데이터 그래프 간선들을 이어준다. 이러한 random walk 작업을 정점 갯수가 원하는 크기가 될 때 까지 반복하여 무작위의 tree 그래프들을 추출하였다. Tree Graph의 정점 개수를 1-10, 11-20, 21-30, 31-40, 41-50으로 변화시키며 실험하였다. Tree 그래프의 정점 레이블은 데이터 그래프에서 가장 빈번하게 등장하는 레이블 L_f로 하였다.

4.3.2 Clique 그래프의 크기에 따른 실험

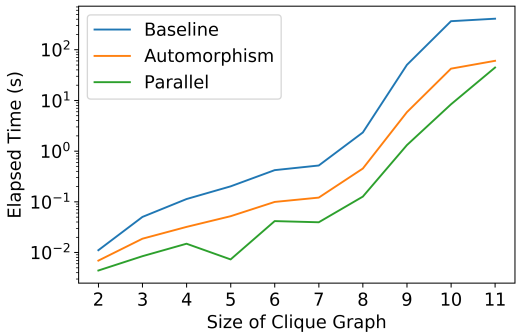
그림 4은 clique 그래프의 크기에 따른 세 알고리즘의 MNI 계산 시간을 보여준다. MiCo와 Patent 데이터에 대해서, Automorphism 알고리즘이 Baseline 알고리즘에 비해 최대 10배까지 효율이 증가하는 것을 볼 수 있다. 가장 큰 clique 쿼리의 크기가 11이고, 이 때 11개의 정점이 모두 서로 동치이므로, 이론상 최대 11배의 성능 증가를 기대할 수 있으므로, 합리적인 결과로 볼 수 있다. 반면에, Yeast 데이터에 대해서는 다른 데이터에 비해 데이터 그래프의 크기가 작기 때문에 후보 공간의 크기가 작고, 백트래킹에 걸리는 시간 또한



(a) MiCo



(b) Yeast

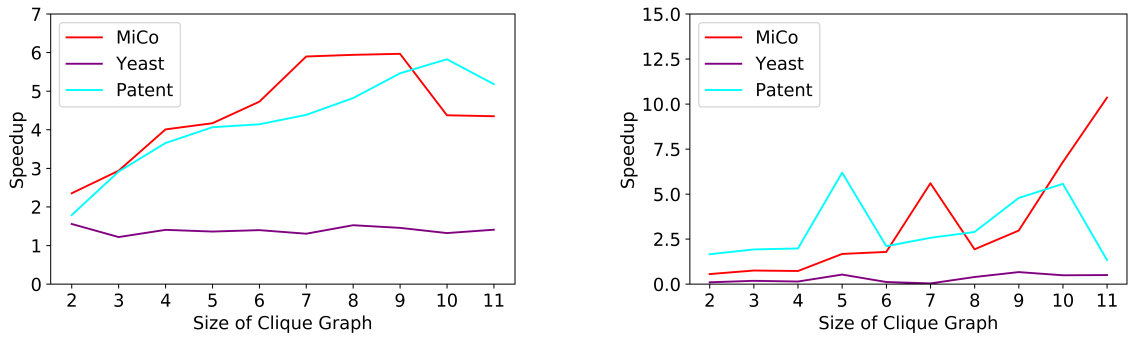


(c) Patent

그림 4: MNI Computation Time for Clique Graph

작다. 이로 인해, 자기 동형 사상에 의한 speedup이 예상되는 값에 비해 적은 것이라 볼 수 있다. 실험 5.2에서 제한 시간 내에 MNI를 계산하지 못한 cycle 쿼리 그래프와 달리, clique 그래프는 모두 제한 시간 내에 MNI를 계산할 수 있다. 이는 clique 그래프의 경우 cycle 그래프에 비해 간선의 갯수가 많기 때문에 VEQ의 필터링을 통해 많은 정점들이 후보 공간에서 제거 되기 때문에 각 백트래킹에 소요되는 시간이 clique 그래프가 cycle 그래프에 비해 적기 때문이다.

그림 5의 (a)는 clique 그래프의 크기에 따른 각 데이터의 Automorphism 알고리즘의 Baseline 알고리즘 대비 Speedup을 나타낸다. 각 clique 그래프의 크기에 대해, 상위 5개의 레이블을 부여한 그래프에서 Baseline 알고리즘의 수행시간을 Automorphism 알고리즘의 수행시간으로 나눈 값의 기하평균을 표시하였다. MiCo와 Patent 데이터의 경우 clique 그래프의 크기가 증가함에 따라 동치인 정점의 갯수가 많아지기 때문에, 자기 동형 사상의



(a) Automorphism

(b) Parallel

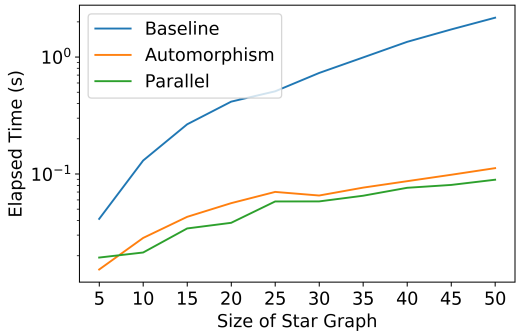
그림 5: Speedup for Clique Graph

효과가 증가하는 것을 볼 수 있다. 반면에, Yeast 데이터의 경우에는 수행시간이 너무 짧기 때문에 쿼리 그래프의 크기가 커짐에 따라 speedup이 증가하는 것을 관찰하기 어렵다.

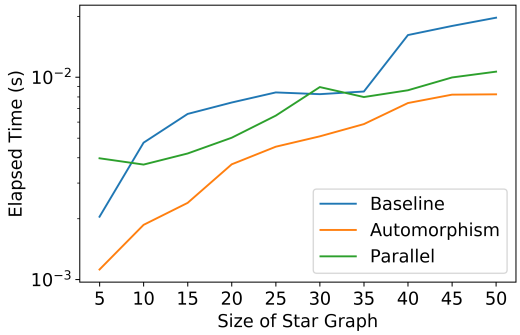
그림 5의 (b)는 clique 그래프의 크기에 따른 Parallel 알고리즘의 Automorphism 알고리즘 대비 Speedup을 나타낸다. 각 clique 그래프의 크기에 대해, 상위 5개의 레이블을 부여한 그래프에서 Automorphism 알고리즘의 수행시간을 Parallel 알고리즘의 수행시간으로 나눈 값의 기하평균을 표시하였다. 병렬화에 의한 speedup은 대체로 크기와 무관하나, 수행시간이 짧은 Yeast의 경우, 쓰레드마다 자료구조를 할당하고 초기화 하는 시간에 인해 Parallel 알고리즘이 단일 쓰레드의 Automorphism 알고리즘에 비해 느린 것을 볼 수 있다.

4.3.3 Star 그래프의 크기에 따른 실험

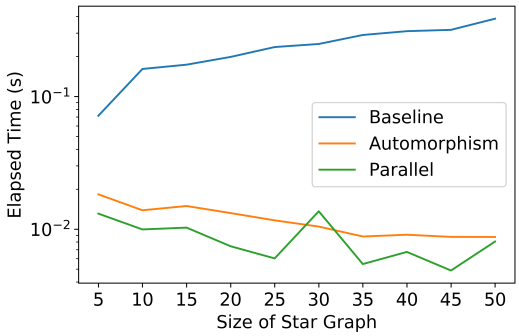
그림 6은 star 그래프의 크기에 따른 세 알고리즘의 MNI 계산 시간을 보여준다. Star 그래프의 경우 중심 정점을 제외한 모든 말단 정점들이 서로 동치이다. Patent의 경우 Automorphism 알고리즘이 Baseline 알고리즘에 비해 최대 45배까지 효율이 증가하는 것을 볼 수 있고, MiCo의 경우 약 20배까지 효율이 증가하였다. 가장 큰 star 그래프의 말단 노드의 갯수가 50이고, 이 때 50개의 정점이 모두 서로 동치이므로, 이론상 최대 50배의 성능 증가를 기대할 수 있으므로, 합리적인 결과로 볼 수 있다. 반면에, Yeast 데이터에 대해서는 최대 2.4배의 성능 증가가 이루어졌는데, 이는 clique 그래프와 마찬가지로, 데이터 그래프의 크기가 작기 때문에 자기 동형 사상에 의한 speedup이 예상되는 값에 비해 적은 것이라 볼 수 있다.



(a) MiCo



(b) Yeast



(c) Patent

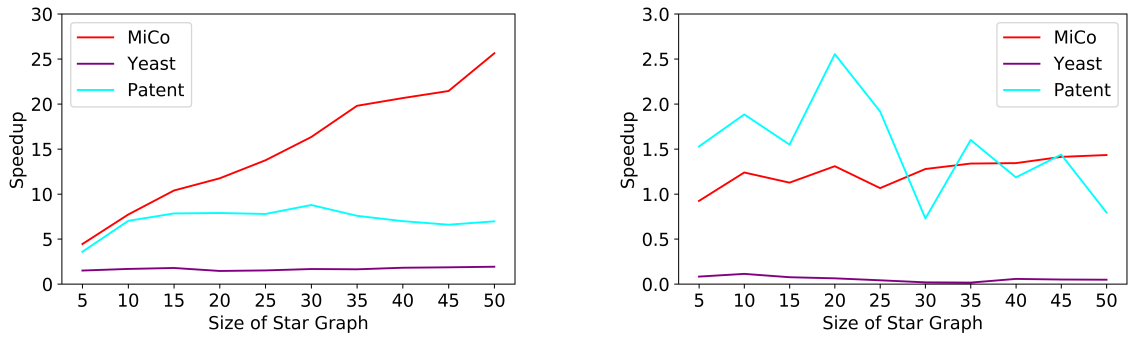
그림 6: MNI Computation Time for Star Graph

그림 7의 (a)는 star 그래프의 크기에 따른 각 데이터의 Automorphism 알고리즘의 Baseline 알고리즘 대비 Speedup을 나타낸다. 모든 데이터에 대해, star 그래프의 크기가 증가함에 따라 동치인 정점의 갯수가 많아지기 때문에, 자기 동형 사상의 효과가 증가하는 것을 볼 수 있다.

그림 7의 (b)는 star 그래프의 크기에 따른 Parallel 알고리즘의 Automorphism 알고리즘 대비 Speedup을 나타낸다. Clique 그래프와 마찬가지로, 병렬화에 의한 speedup은 대체로 크기와 무관한 것을 볼 수 있고, Yeast 역시 데이터 그래프의 크기가 작기 때문에 Parallel 알고리즘이 단일 쓰레드의 Automorphism 알고리즘에 비해 느린 것을 볼 수 있다.

4.3.4 Tree 그래프의 크기에 따른 실험

그림 8은 tree 그래프의 크기에 따른 세 알고리즘의 MNI 계산 시간을 보여준다. 실험



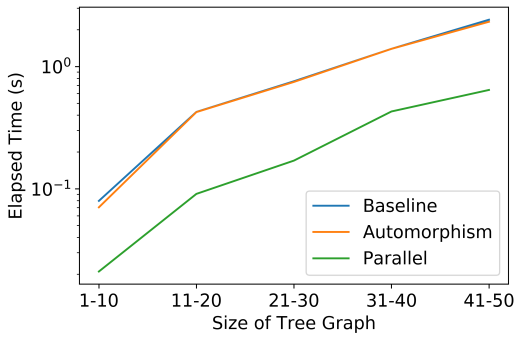
(a) Automorphism

(b) Parallel

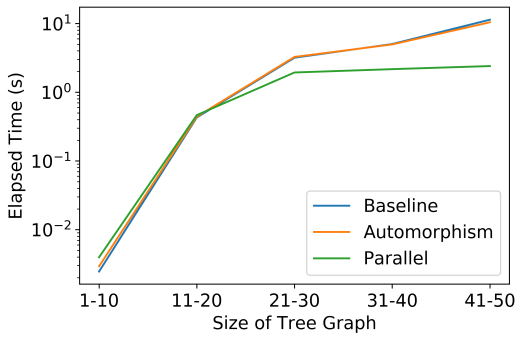
그림 7: Speedup for Star Graph

결과, 데이터 그래프에 따라 Automorphism 알고리즘은 Baseline 알고리즘 대비 최대 1.3배의 성능 향상을 보였다. Tree 그래프는 random walk를 통해 생성된 쿼리 그래프이기 때문에 대체로 동치인 정점들이 존재하지 않는 것을 볼 수 있었다. 따라서, 자기 동형 사상에 의한 speedup이 clique 그래프나 star 그래프에 비해 작다고 분석할 수 있다.

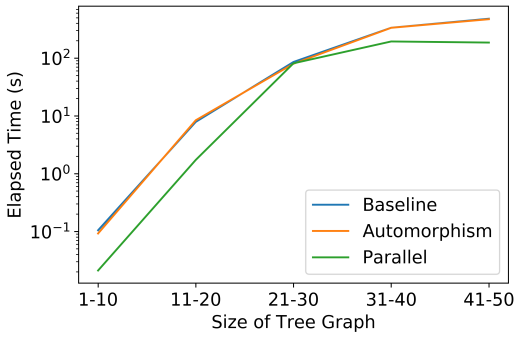
그림 9의 (a)는 tree 그래프의 크기에 따른 각 데이터의 Automorphism 알고리즘의 Baseline 알고리즘 대비 Speedup을, (b)는 그래프의 크기에 따른 Parallel 알고리즘의 Automorphism 알고리즘 대비 Speedup을 나타낸다. 자기 동형 사상과 병렬화 모두 tree 그래프의 크기와 speedup은 무관한 것을 볼 수 있다. 다만, 수행시간이 지나치게 짧은 Yeast의 경우에는 쿼리 그래프가 커짐에 따라 Parallel 알고리즘에서 자료구조를 할당하고 초기화 하는 시간이 차지하는 비중이 작아지기 때문에, 병렬화의 speedup이 소폭 증가하는 것을 관찰할 수 있었다.



(a) MiCo

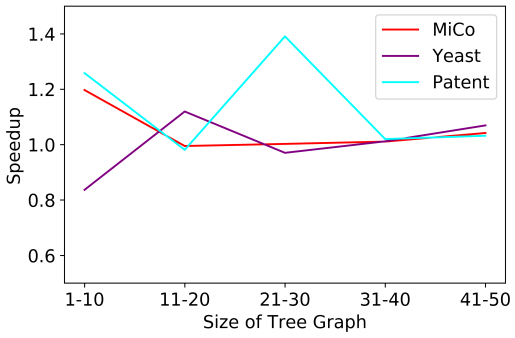


(b) Yeast

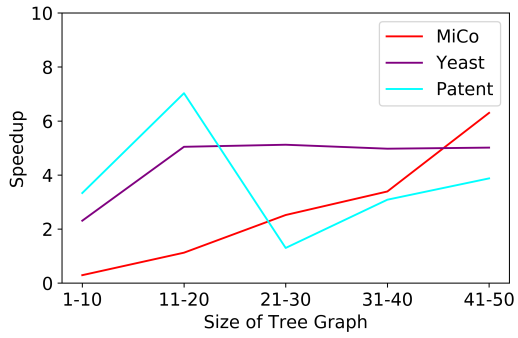


(c) Patent

그림 8: MNI Computation Time for Tree Graph



(a) Automorphism



(b) Parallel

그림 9: Speedup for Tree Graph

제 5 장 결론

본 연구에서는 자기 동형 사상을 이용하여 MNI를 효율적으로 계산하는 알고리즘을 제안하였다. 제안된 알고리즘은 대칭적인 쿼리 그래프들에서 Baseline 알고리즘과 비교하여 개선점을 보였다. 더불어, 병렬화를 통해 제안한 알고리즘이 멀티코어 환경에서 더 빠르게 작동할 수 있게 개선하였다. 또한, MiCo, Yeast, Patent의 실제 그래프 데이터를 통해 clique, star, tree 그래프 들에서 제안된 알고리즘이 효율적임을 입증하였다.

실험 결과, 자기 동형 사상을 사용할 때 동치류의 크기가 큰 쿼리 그래프에 대해 더 빠른 MNI 계산이 가능하다는 것을 보여주었다. 병렬화를 포함한 최종적인 알고리즘은 기존 Baseline 알고리즘 대비 최대 45배 가량 빠른 것을 볼 수 있었다. 병렬화의 경우, 대부분의 경우에 대해 성능이 향상되었지만, 수행시간이 짧은 몇몇 경우에 대해 overhead로 인해 성능이 느려지는 것을 관찰할 수 있었다. 본 연구는 대칭적인 그래프의 MNI 계산을 더 빠르게 처리하기 위한 효율적인 방법을 통해 frequent subgraph mining 등 MNI를 사용하는 그래프 기반 문제들에 기여할 수 있을 것으로 기대된다.

참 고 문 헌

[1] L. Tang and H. Liu, "Graph mining applications to social network analysis," Managing and mining graph data, pp. 487–513, 2010.

[2] X.-L. Li, C.-S. Foo, S.-H. Tan, and S.-K. Ng, "Interaction graph mining for protein complexes using local clique merging," Genome Informatics, vol. 16, no. 2, pp. 260–269, 2005.

[3] Z. Futai, G. Yuxiang, P. Bei, P. Li, and L. Linsen, "Web phishing detection based on graph mining," in 2016 2nd IEEE international conference on computer and communications (ICCC), IEEE, 2016, pp. 1061–1066.

[4] S. U. Rehman and S. Asghar, "Online social network trend discovery using frequent subgraph mining," Social Network Analysis and Mining, vol. 10, no. 1, p. 67, 2020.

[5] M. Seeland, T. Girschick, F. Buchwald, and S. Kramer, "Online structural graph clustering using frequent subgraph mining," in Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2010, Barcelona, Spain, September 20-24, 2010, Proceedings, Part III 21, Springer, 2010, pp. 213–228.

[6] T. K. Saha, A. Katebi, W. Dhifli, and M. Al Hasan, "Discovery of functional motifs from the interface region of oligomeric proteins using frequent subgraph mining," IEEE/ACM transactions on computational biology and bioinformatics, vol. 16, no. 5, pp. 1537–1549, 2017.

[7] B. Bringmann and S. Nijssen, "What is frequent in a single graph?" In Advances in Knowledge Discovery and Data Mining: 12th Pacific-Asia Conference, PAKDD 2008 Osaka, Japan, May 20-23, 2008 Proceedings 12, Springer, 2008, pp. 858–863.

[8] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "Grami: Frequent subgraph and pattern mining in a single large graph," Proceedings of the VLDB Endowment, vol. 7, no. 7, pp. 517–528, 2014.

[9] B. D. McKay and A. Piperno, "Practical graph isomorphism, ii," Journal of symbolic computation, vol. 60, pp. 94–112, 2014.

[10] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han, "Versatile equivalences: Speeding up subgraph query processing and subgraph matching," in Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 925–937.

Abstract

One of the core problems in graph mining is the frequent subgraph mining problem. The frequent subgraph mining problem is to find subgraphs with the largest *support* in a data graph, where the support refers to how frequently a subgraph appears in the graph. MNI is a commonly used support measure in frequent subgraph mining. In this paper, we propose an efficient method for computing the MNI using automorphism of the query graph. Additionaly, we parallelized the backtracking process during MNI computation to improve performance in a multicore environment. Experiments conducted on real data graphs demonstrate that our method of using automorphism and parallelization is efficient for computing the MNI.

Keywords: subgraph matching, adaptive matching order, graph data analysis