알고리즘 과제 2 Report

노규민, 조승한

June 8th, 2021

1 Introduction

NP-complete인 Subgraph Matching Problem은 Big Data Analysis의 핵심 Logic으로, 이를 해결하는 빠른 알고리즘을 찾는 것은 중요한 문제다. 이에 현재까지 알려진 가장 빠른 알고리즘인 DAF를 이해하고, DAF의 Matching Order를 개선하여 보다 좋은 Matching Order를 고안해 보고자 한다.

2 Environment

보고서의 Result에 대한 실행 환경은 아래와 같다.

Machine Specification

- OS : Ubuntu 20.04
- CPU : Intel Core i5-11500B Processor (12M Cache, up to 4.60 GHz)
- Memory : 3GB

Compiler/Interpreter

- C/C++ : gcc 11
- Python : Python 3.7.10

3 Compile and Execute

```
$ mkdir build
$ cd build
/build$ cmake ..
/build$ make
```

Figure 1: Build Command

Figure 1과 같이 shell command를 통해 프로그램을 build 할 수 있다. 프로그램 실행은 Figure 2의 command로 가능하다. option을 주어 output file을 지정할 수 있고, check mode로 진입하여 출력 결과를

```
/build$ ./main/program <data graph file> <query graph file> <candidate set file>

option 1. Specify the output file
  ./main/program <data graph file> <query graph file> <candidate set file> <output file>

option 2. check mode
  ./main/program <data graph file> <query graph file> <candidate set file> -c

option 3. select backtrack mode
  ./main/program <data graph file> <query graph file> <candidate set file> -0
  ./main/program <data graph file> <query graph file> <candidate set file> -1
  ./main/program <data graph file> <query graph file> <candidate set file> -2
  ./main/program <data graph file> <query graph file> <candidate set file> -3
```

Figure 2: Run Command

validate 할 수 있다. 또한, 구현한 총 5가지 알고리즘 중에 하나를 사용하도록 지정할 수 있다.

각 option마다 실행되는 알고리즘은 다음과 같다.

- "-0": IgnoreDAG2 알고리즘
- "-1": DAF 알고리즘
- "-2": ELPSM 알고리즘
- "-3": optimize-da 알고리즘
- "-4": IgnoreDAG 알고리즘

아무런 option을 주지 않으면, stdout에 IgnoreDAG2 알고리즘을 시행한 결과가 보여진다. 또한, python3로 test.py를 수행하여 주어진 24개 query에 대한 각 알고리즘의 score 및 실행 시간을 확인할 수 있다.

4 Implementation

4.1 DAF Algorithm

DAF algorithm [1]의 Candidate-size matching order를 기반으로 구현한 알고리즘이다. code와 header는 "src/daf.cc", "src/daf.h"의 파일명으로 정의했다. 전체 알고리즘은 아래 순서로 진행된다.

- query graph에서 candidate size가 가장 작은 node를 root로 정함
- root를 기준으로 BFS order에 따라 query DAG를 구성
- Candidate-size matching order 기반 Backtracking을 수행

이 알고리즘은 기본적으로 DAF 논문에서 주어진 알고리즘에 Failing Set을 이용한 pruning을 제거하고, candidate-size order를 선택한 것이다. 또한, 논문에서 주어진 $C_M[u]$의 계산을 반복적으로 하는 것을 막고 알고리즘을 최적화하기 위해서 간단한 아이디어를 사용하였다.

Algorithm 1: Find_dag($root$)

```
Initialize all elements of qdd to 0;
push root into queue;
while queue is not empty do
    u ← front of queue;
    pop queue;
    foreach v ← reachable nodes from u do
        if v has been popped from queue before then
            qdd[u] += 1;
        end
        if v has not been pushed to the queue then
            push v into queue;
        end
    end
end
```

Algorithm 2: Backtrack(id, $matched$)

```
if matched = number of query graph nodes then
    Report M and return;
end
Clear all elements of save_del;
foreach u ∈ reachable nodes from id do
    qdd[u] -= 1;
    if qdd[u] is positive that means there is an edge from id to u in DAG then
        foreach v ∈ C_M[u] do
            if v is not neighbor with M[id] then
                v insert into save_del[u];
            end
        end
        C_M[u] ← difference of C_M[u] with save_del[u];
    end
end
Find u that has minimum C_M[u] size and qdd[u] = 0;
foreach v ∈ C_M[u] do
    if used[v] = false then
        M[u] ← v;
        used[v] ← true;
        Backtrack(u, matched + 1);
        used[v] ← false;
    end
end
foreach u ∈ reachable nodes from id do
    if qdd[u] is positive then
        C_M[u] ← union of C_M[u] and save_del[u];
    end
    qdd[u] += 1;
end
```

Algorithm 1은 query DAG 구성 알고리즘에 대한 pseudocode이다. BFS order로 탐색하면서 모든 edge에 대해 queue에서 먼저 pop된 node를 출발점으로 하는 단방향 edge로 바꾸면 DAG가 된다. qdd array는 DAG로 구성했을 때 자신으로 들어오는 edge 개수를 저장한다. 즉, qdd[x]가 3이라면 DAG에서 node x로 들어오는 edge가 3개라는 의미이다. 이 배열은 부모가 전부 match 되었는지 판단하기 위해서 사용된다.

Algorithm 2는 Candidate-size matching order 기반 Backtraking 알고리즘이다. Backtrack(id, matched)는 전에 matching된 Vertex를 id로 받아 Candidate Set을 update하고 다음 matching할 vertex를 찾아준다. 만약 모든 query vertex에 대한 matching이 완료되었다면 하나의 valid matching을 찾은 것으로, 이를 출력하게 된다. Algorithm 1을 통해 얻은 qdd를 통해 DAG 순서로 Backtracking이 진행된다. save-del은 M[id]에 의하여 지워진 candidate set을 저장하고 있다. id에 대한 matching이 끝났다면 다시 이들을 Union하여 복구한 후 다음 matching을 준비해야한다. 물론, 초기의 $C_M[u]$는 u에 대응되는 candidate의 집합이다.

4.2 ELPSM Algorithm

ELPSM은 (정확하게는 ELP_{SM}) [2]에서 제시된 알고리즘으로, 순수하게 Matching Order만 본다면 DAF와 큰 차이는 없는 알고리즘이다. 가장 핵심적인 차이는 matching order에 Leaf Adaptive Matching이란 것을 추가한다는 점인데, 핵심적인 아이디어는 다음과 같다. 만약 degree가 1인 여러 leaf 노드가 부모 노드가 같고 Label 역시 동일하다면, 이들은 동등한 노드라고 생각할 수 있다. 그러니, 각각을 matching 시키지 말고, 이들을 한꺼번에 여러 노드와 매칭시킨 다음 출력할 때 모든 permutation을 고려해도 충분할 것이다. 즉, 5개의 leaf가 동등하다면, 이들 5개를 data graph의 5개의 노드와 매칭시킨 다음, $5! = 120$개의 실제 매칭을 출력 단계에서 고려하면 된다. 이렇게 matching을 하면, 동등한 노드를 matching 시키느라 backtracking을 여러 번 반복할 필요가 없어 더욱 효율적인 알고리즘이 나온다.

알고리즘이 상당히 길이가 있어, pseudocode는 핵심적인 부분인

- 동등한 leaf를 찾는 방식
- matching order
- matching을 찾은 다음 동등한 leaf를 처리하는 법

만 다루도록 하고, 나머지 부분은 어떤 값을 계산하는지와 그 방법을 간략하게 설명하도록 하겠다.

알고리즘의 흐름은 다음과 같다.

- Building DAG - Merging Leaves - Preprocessing DAG로 이루어진 전처리
- Backtracking Phase와 그 과정에서 Matching Order, Fast Leaf Matching, Printing Matches

Building DAG : root의 경우 [1]의 알고리즘처럼 $C_M[u]/\deg(u)$가 가장 작은 vertex를 선택한다. 여기서 C_M은 이미 얻은 candidate set의 크기이며, $\deg$는 query graph에서의 degree이다. 그 후, BFS를 통해 root에서부터의 거리를 계산하고, 거리가 작은 쪽에서 큰 쪽으로, 거리가 같으면 vertex의 번호가 작은 쪽에서 큰 쪽으로 간선이 이어지도록 하여 DAG를 구축한다. 여기서 다음 세 가지 배열을 준비한다.

- query_dag : $query_dag[i]$는 DAG에서 i에서 뻗어나가는 간선의 도착점들이다
- query_dag_rev : $query_dag_rev[i]$는 반대로 DAG에서 i로 들어가는 간선의 시작점들이다.
- qdd : $qdd[i]$는 DAF에서와 같이 i로 들어가는 간선의 개수이다.

Merging Leaves : 이제 동등한 leaf를 계산할 준비를 한다.

Algorithm 3: DAG_merge

$merger$: std::map from (Vertex, Label) to vector of Vertex;
foreach $u \in$ *query graph vertex* **do**
  if u *is a degree one leaf, i.e.* $query\_dag[u]$ *is empty and* $qdd[u] = 1$ **then**
    $parent \leftarrow query\_dag\_rev[u][0]$;
    append u to $merger[(parent, Label(u))]$
  end
end
foreach $(v, l, S) \in merger$ **do**
  (S is the vector of Vertex corresponding to (v, l));
  if $size(S) \geq 2$ **then**
    $merged[S[0]] \leftarrow S$;
    erase all elements of S except $S[0]$ from $query\_dag[v]$;
    denote $S[0]$ as a "large leaf";
    denote all elements of S except $S[0]$ as a "fake leaf";
  end
end

결국 동일한 parent, 동일한 label을 갖는 degree 1 leaf를 모으고, 그 중 하나만을 전체의 "대표"로 보고 나머지를 DAG에서 제거한 것이다. "대표"는 "large leaf"라는 성질을 갖게 되고 제거된 leaf는 "fake leaf"라는 성질을 갖게 된다. 이러한 성질들은 물론 boolean 배열 또는 std::bitset을 이용하여 관리가 된다.

Preprocessing DAG : 이 알고리즘에서도 DAF처럼 backtracking 과정에서 부모가 전부 matching 되어있는지를 확인하게 된다. 앞선 DAF의 구현에서는 qdd 배열을 이용하여 이를 구현하였는데, 여기서는 다른 접근을 사용하였다. 특별한 이유는 없으며, 구현한 사람이 달라 다른 접근을 사용한 것 뿐이다. 이 구현에서는 위상정렬과 DP를 합친 알고리즘을 이용하여 각 정점에 대해 자신의 조상들의 집합을 전부 계산하였다. 또한, backtracking 과정에서 현재 결정한 vertex의 집합을 관리하여, "자신의 조상들의 집합"이 "결정한 vertex의 집합"의 부분집합이면 우리가 원하는 부모가 전부 matching된 상황으로 볼 수 있다. 이때 부분집합 여부를 빠르게 판단하기 위해서는, boolean 배열을 관리하는 것보다는 std::bitset을 사용하는 것이 더욱 효율적이다. 이 부분의 pseudocode는 matching order와 큰 관계가 없으며, elpsm.cc에 작성된 코드 (DAG_preprocess) 역시 간단하므로 설명을 생략한다.

Matching Order : 이제 다음으로 match 할 정점을 고르는 방법을 설명한다.
$matched$는 현재 matching된 정점의 개수이다. 아래 알고리즘 역시 [2]에서 그대로 가져왔다.

아래 알고리즘에서 "fake leaf"는 선택하지 않는다.

Algorithm 4: nxt_extend($matched$)

if $matched = 0$ **then**
  return root;
end
if *there is a non-decided vertex with empty* C_M **then**
  return -1;
end
if *there is a extendable large leaf* u *with* $|C_M[u]| = |merged[u]|$ **then**
  return u;
end
if *all extendable vertices are large leaves* **then**
  return the large leaf with minimum C_M size;
end
if *there is a extendable vertex that is not large leaves* **then**
  return the non-large-leaf with minimum C_M size;
end

핵심은 $|C_M[u]| = |merged[u]|$로, 이 경우 하나로 묶인 동등한 leaf들에 대응되어야 할 data graph의 vertex들이 고정된다. extendable 여부의 판단은 앞서 언급한대로 "현재 결정한 vertex의 집합"과 "조상들의 집합" 사이의 포함관계를 따져서 할 수 있다. -1이 반환되는 것은 extend가 불가능하다는 것이다.

Backtracking Phase : 기본적으로 DAF의 backtracking과 비슷하지만, 아주 약간의 차이가 있다. 우선 backtracking에 들어가기 전 (사실 DAG를 계산하는 부분에서) 다음 두 배열을 준비한다.

- cands : $cands[i]$는 i번 query vertex가 대응될 수 있는 candidate set으로, std::set이다.
- cand_rev : $cand_rev[i]$는 i번 data vertex가 대응될 수 있는 query vertex의 집합으로, std::vector이다.

이들의 초기화 과정은 복잡하지 않으므로, elpsm.cc 코드를 참조하자.

Algorithm 5: ELPSM($matched$)

if $matched$ = *number of query graph nodes* **then**
  print_matches(0) and return;
end
$nxt \leftarrow nxt\_extend(matched)$;
if $nxt = -1$ **then**
  return;
end
if nxt *is not a large leaf* **then**
  foreach $match \in C_M[nxt]$ **do**
    $M[nxt] = match$;
    mark nxt as decided;
    foreach $v \in query\_dag[nxt]$ **do**
      ($N(match)$ is the neighbors of $match$ in data graph);
      $C_M[v] \leftarrow C_M[v] \cap N(match)$;
    end
    foreach $v \in cand\_rev[match]$ **do**
      erase $match$ from $C_M[v]$;
    end
    ELPSM($matched + 1$);
    mark nxt as undecided;
    recover all previous values of $C_M[v]$;
  end
end
if nxt *is a large leaf* **then**
  (here, we take the std::set $cands[nxt]$ and make it an array);
  $cand\_nxt \leftarrow cands[nxt]$;
  fastleafmatch($matched$, nxt, 0, 0);
end

nxt가 large leaf가 아닌 경우, 가능한 candidate을 nxt에 대응시키고, 이에 따라 nxt의 DAG 상 자식들이 C_M 값이 달라지는 것을 미리 계산한다. 특히, nxt에 $match$가 대응된 경우 $match$를 candidate으로 가지고 있는 vertex를 순회하여 이를 제거한다. 이 경우는 DAF와 크게 다르지 않다는 것을 알 수 있다.

nxt가 large leaf인 경우가 문제인데, 현재 갖고 있는 $C_M[nxt]$를 nxt에 묶여있는 동등한 leaf들에 대응시켜야 하기 때문이다. 이 부분을 처리하는 것이 위 pseudocode에서도 등장하는 "fastleafmatch" 알고리즘이다. 이 알고리즘은 복잡하지 않으므로 pseudocode 대신 글로 설명하도록 한다.

Fast Leaf Matching : 현재 nxt에 n개의 leaf가 묶여있고 $|C_M[nxt]| = m$이라고 하자. 그러면 우선 대응시킬 수 있는 $C_M[nxt]$의 부분집합은 총 $\binom{m}{n}$개가 있다. 이들을 backtracking으로 나열하고, matching 된 data graph의 vertex에 대하여 앞선 Algorithm 5의 $cand_rev$를 이용한 처리를 해준다.

만약 backtracking으로 $C_M[nxt]$의 크기 n 부분집합을 결정했다면, 다시 ELPSM을 호출한다.

Printing Matches : ELPSM에서는 large leaf를 이용한 특수 처리가 있기 때문에, 출력 과정도 난해한 점이 있다. large leaf가 아닌 경우는 단순히 출력하면 되지만, large leaf의 경우 처리할 점이 있다.

u가 large leaf라고 하고, u에 n개의 leaf가 묶여있다고 가정하자. 그렇다면 여기서 $n!$개의 permutation 중 하나를 선택해야 한다. 이를 위해서 std::algorithm의 next permutation 함수를 사용하여, permutation을 고르고 재귀호출을 하는 것을 반복한다. 아래 pseudocode를 참고하라.

Algorithm 6: print_matches(cur)

```
if cur = number of query graph nodes then
    Report M and return;
end
if cur is not a large leaf then
    print_matches(cur + 1);
end
if cur is a large leaf then
    n ← size(merged[cur]);
    foreach i ∈ [0..n) do
        partial_match ← M[merged[cur][i]]
    end
    foreach σ ∈ S_n do
        foreach i ∈ [0, n) do
            M[merged[cur][i]] ← partial_match[σ[i]]
        end
        print_matches(cur + 1)
    end
    reset M[merged[cur][i]]'s to original
end
```

알고리즘을 전체적으로 보면, leaf들이 잘 merge 되어야 효율적인 알고리즘이 됨을 알 수 있다. 특히, leaf가 merge 되어야 얻어지는 최적화를 위해서 기존에 간단하게 짤 수 있는 알고리즘도 훨씬 복잡하게 구현되었다. 그런만큼, leaf들이 잘 merge되지 않는다면 이 알고리즘은 DAF에 비해 비효율적이다. 안타깝게도, 실험 결과에서는 leaf들이 잘 merge가 되지 않았다. 이는 다음 알고리즘의 구현으로 이어졌다.

4.3 optimize-da Algorithm

ELPSM 알고리즘을 구현한 결과, 동등한 leaf가 자주 등장하지 않아 최적화가 잘 되지 않는다는 점을 확인하였다. 그래서 동등한 leaf를 구하는 과정 및 동등한 leaf를 처리하는 과정을 전부 제거하여, optimize-da라는 구현체를 만들었다. 이 알고리즘은 사실상 DAF와 거의 동일한데, 단순히 구현 방식에 약간의 차이가 있을 뿐이다. 큰 차이가 없으므로, 앞선 알고리즘과의 차이만 간략하게 설명한다.

- ELPSM의 "ELPSM" 함수는 그대로 적용되는데, large leaf를 처리하는 부분은 필요가 없다.
- DAG_merge 부분과 fastleafmatch 역시 필요가 없어, 구현이 매우 간단해진다.
- extendable 한 것 중 $|C_M[u]|$의 크기가 가장 작은 것을 다음 정점으로 선택한다.
- matching을 출력하는 과정 역시 이제 단순히 M의 원소들을 순서대로 출력하면 된다.

전체적으로 ELPSM보다 훨씬 간단한 알고리즘이지만, 더 좋은 성능을 보인다.

4.4 IgnoreDAG/IgnoreDAG2 Algorithm

이번 알고리즘에서는 DAG 자체를 구축하지 않는다. DAG를 사용하지 않으니, 알고리즘을 IgnoreDAG라고 부른다. 제출한 파일에는 IgnoreDAG에 2가지 version이 존재하는데, 그 차이는 크지 않다.

Backtracking Framework 자체는 optimize-da와 다르지 않은데, 하나의 작은 차이점이 있다. 앞서 우리는 Algorithm 5에서 $query_dag[nxt]$에 속한 v에 대하여 $C_M[v] \leftarrow C_M[v] \cap N(match)$을 하는데, 이를 nxt의 자식인 v에만 적용하는 이유는 nxt의 조상들은 이미 matching order 상 대응되는 data graph vertex가 결정되었기 때문이다. IgnoreDAG 알고리즘에서는 matching order를 정하기 위해서 DAG 자체를 이용하지 않으므로, 이 과정에서 $query_dag[nxt]$를 사용하지 않고 query graph의 neighbor 전부를 사용한다.

이 알고리즘에서는 DAG를 구축하지 않고 간단한 전처리 후 바로 backtracking으로 들어간다.

Framework가 optimize-da와 거의 동일하므로, 실제로 논의해야 할 것은 matching order 하나 뿐이다. 이는 매우 간단한데, 단순히 결정되지 않은 vertex 중 $|C_M[u]|$가 가장 작은 것이다. 즉, 앞서 DAF와 ELPSM에서는 조상들이 다 결정되어야 한다는 제약을 걸었는데, 여기서는 이를 고려하지 않고 정점을 선택하는 것이다. 이것이 IgnoreDAG에서의 정점 선택 알고리즘이다.

IgnoreDAG2에서는 정점 선택 과정에서 마찬가지로 $|C_M[u]|$가 가장 작은 것을 고르는데, tie-breaking을 하나 추가한다. 즉, $|C_M[u]|$가 가장 작은 정점 u가 여러 개 있는 경우, 그 중 하나를 선택하는 알고리즘을 추가하였다. tie-breaking 방법은, 각 정점의 neighbor 중 결정되지 않은 것이 가장 많은 정점을 고르는 것이다. 이를 효율적으로 구현하려면 각 정점의 neighbor 중 결정되지 않은 것의 개수를 관리해야 하는데, 이는 backtracking 과정에서 어렵지 않게 할 수 있다. ignore_dag_2.cc를 참고하라.

두 알고리즘의 성능은 비슷했지만, IgnoreDAG2가 우위를 가졌다. repository의 result를 참고하라. IgnoreDAG/IgnoreDAG2 알고리즘은 거의 동일하므로, 아래 Result 표에는 IgnoreDAG2만 수록하였다.

		DAF		ELPSM		optimize-da		ignore dag2	
Data	**Query**	score	time (s)	score	time (s)	score	time (s)	score	time (s)
lcc_hprd	***lcc_hprd_n1***	96	0.058	96	0.06	96	0.061	96	0.059
	lcc_hprd_n3	100,000	1.285	100,000	1.418	100,000	1.349	100,000	1.356
	lcc_hprd_n5	32,832	0.695	32,832	0.757	32,832	0.711	32,832	0.701
	lcc_hprd_n8	100,000	2.649	100,000	2.801	100,000	2.619	100,000	2.657
	lcc_hprd_s1	504	0.075	504	0.077	504	0.075	504	0.075
	lcc_hprd_s3	100,000	1.499	100,000	1.446	100,000	1.361	100,000	1.338
	lcc_hprd_s5	100.000	2.236	100,000	2.139	100,000	2.026	100,000	2.003
	lcc_hprd_s8	100,000	2.856	100,000	2.823	100,000	2.642	100,000	2.643
lcc_human	***lcc_human_n1***	100,000	0.26	100,000	0.231	100,000	0.224	100,000	0.235
	lcc_human_n3	100,000	0.38	0	time out	0	time out	100,000	0.349
	lcc_human_n5	0	time out	100,000	14.893	100,000	14.644	100,000	0.535
	lcc_human_n8	100,000	0.641	0	time out	0	time out	100,000	0.671
	lcc_human_s1	100,000	0.263	100,000	0.212	100,000	0.206	100,000	0.212
	lcc_human_s3	100,000	0.574	100,000	0.504	100,000	0.485	100,000	0.432
	lcc_human_s5	0	time out	0	time out	0	time out	100,000	0.505
	lcc_human_s8	0	time out	0	time out	0	time out	100,000	0.631
lcc_yeast	***lcc_yeast_n1***	100,000	0.656	100,000	0.724	100,000	0.676	100,000	0.684
	lcc_yeast_n3	0	time out	0	time out	0	time out	100,000	1.305
	lcc_yeast_n5	100,000	2.051	100,000	2.122	100,000	1.907	100,000	1.968
	lcc_yeast_n8	100,000	2.793	100,000	2.563	100,000	2.497	100,000	2.501
	lcc_yeast_s1	100,000	1.108	100,000	0.863	100,000	0.827	100,000	0.807
	lcc_yeast_s3	0	time out	0	time out	0	time out	0	time out
	lcc_yeast_s5	0	time out	0	time out	0	time out	100,000	1.878
	lcc_yeast_s8	0	time out	0	time out	0	time out	0	time out

Figure 3: Result

5 Result

lcc-hprd, lcc-human, lcc-yeast의 3개 data graph와 24개의 query graph를 input으로 하여 4가지 알고리즘에 적용시킨 결과 위 Figure의 결과를 얻었다. Score는 프로그램이 찾은 valid matching 개수이고 1분 안에 프로그램이 끝나지 않으면 중단하고 Score를 계산한다. time은 프로그램의 실행 시간이고 1분이 넘어 프로그램이 중단된 경우 time out으로 기록한다. 추가로 CHECK MODE를 통해 확인한 결과 모든 output이 correct하다. 흥미로운 사실은 프로그램이 중단된 경우 모두 Score가 0이라는 것이다. 하나의 가능한 matching을 찾았다면 멀지 않은 곳에서 다음 matching을 쉽게 찾을 수 있다는 것이 원인이었다. 여기서 멀지 않은 곳이란 Bracktracking의 탐색 순서에서 다음 matching 까지의 거리를 의미한다. 즉, 하나만 찾으면 나머지 100,000 개를 찾는 것은 금방이지만 이를 1분 내에 찾지 못했을 경우 중단되기 때문에 Score가 모두 0이었다. 알고리즘의 성능을 비교해보면, 모든 query에 대해 가장 높은 score와 유의미하게 빠른 실행시간을 가지는 IgnoreDAG2가 가장 좋은 알고리즘이라고 볼 수 있다. optimize-da와 DAF는 동일한 Candidate-size matching order 기반 백트래킹 알고리즘이지만 구현의 방법에 차이가 있었다. 두 알고리즘은 human-n3, human-n5, human-n8의 3가지 query에 대해 다른 결과를 보였다. 하지만 query에 따라 더 빠른 알고리즘이 달라지기 때문에 성능을 비교하기 힘들다. ELPSM은 optimize-da에서 같은 성질을 가진 leaf들을 한꺼번에 처리하는 최적화를 추가한 알고리즘이다. 하지만 그 최적화가 input query에 대해서 높은 효율을 띄지 않았고, 이에 따라 아쉽지만 성능이 개선되지 않았다.

6 Conclusion

Matching order, 최적화 등에서 차이가 있는 여러 알고리즘을 제시하고 직접 구현한 후 24개의 input query에 대해 그 성능을 비교했다. 결과를 분석했을 때 모든 알고리즘이 correct했으며, 가장 좋은 성능을 가진 알고리즘은 IgnoreDAG2였다. 그러나 실전에서는 pruning 역시 강력한 성능을 발휘하는 만큼, Matching Order와 Pruning을 잘 조합하여 최고의 알고리즘을 만드는 것이 실제 연구에서는 더욱 중요할 것이다. 지금은 IgnoreDAG2가 좋지만, Pruning을 추가하면 다른 order가 우수한 성능을 보일수도 있기 때문이다.

7 References

[1] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1429–1446. DOI:https://doi.org/10.1145/3299869.3319880

[2] 김현준. *Fast Graph Query Processing Algorithms Using Dynamic Programming.* 2020. PhD Thesis. 서울대학교 대학원.