[CS::Algorithm]

Graph Pattern Matching Challenge
Assignment 2

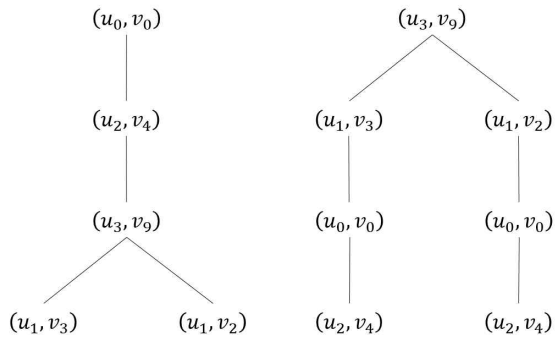
2016-10957 이원석
2016-15788 정원석

[1] Assumption

우리는 backtrack을 구현함에 있어 최대한으로 시간의 이득을 취하기 위해 다음과 같은 가설을 세웠다.

최대한 state search tree에서 travel하는 path의 length가 짧을수록 유리하다.

디자인한 알고리즘은 extendable vertex를 기준으로 next query vertex와 data vertex를 선정한다. 따라서 partial embedding에 새로운 원소를 포함할 때마다 extendable vertex에 대한 정보를 업데이트해줘야 한다. 이러한 이유로 인해 travel하는 path의 length가 짧을수록 업데이트 횟수의 절감효과를 얻을 수 있다. 아래 그림과 같이 간단한 두 state search tree는 결과적으로는 동일한 output을 얻지만 이를 얻기 위해 업데이트를 진행해야 하는 횟수는 다르다. 우리의 알고리즘은 parent node에서 child node로 내려갈 때 한 번의 업데이트가 진행되는데, 이러한 기준에 의하면 왼쪽은 4번의 업데이트가 발생하지만, 오른쪽은 6번의 업데이트가 발생한다.



위에서 세운 가설을 실행하기 위해 next query vertex를 선정하는 기준을 “candidate set ($C_M(u)$) size가 작은 것을 최우선으로 선정한다.”로 설정했다. 이렇게 선정함으로 얻게 되는 이점은 travel path length를 줄일 수 있는 것에 더불어 후순위에 등장하게 될 large size의 $C_M(u)$를 업데이트함으로써 그 크기를 줄일 수 있을 것으로 기대한다.

[2] Backtrack Performance

Subgraph matching에서 설명하고 있는 embedding을 만족하기 위해 본 과제에서는 두 가지 조건만 만족하면 된다. 하나는 injective해야 한다는 것과 다른 하나는 query graph에서 연결된 vertex에 대해서 data graph에서 연결된 성질을 유지해야 한다는 것이다.

embedding을 찾기 위해 partial embedding을 설정했고, next query vertex와 data vertex를 선정하는 효율적인 방법을 찾기 위해 다음과 같은 자료구조를 설정했다.

visited_u[], **visited_v[]** : 우리가 현재까지 찾은 partial embedding이 방문한 query vertex u와 data vertex v를 추적하기 위한 정보를 가지고 있는 boolean vector이다.
extendable_u[level][u] : 현재의 level 단계에서 partial embedding을 정했을 때, 해당 partial embedding으로부터 query vertex u가 extendable한지를 boolean 변수로 저장.
extendable_v[level][u] : 현재의 level 단계에서 partial embedding을 정했을 때, 해당 partial embedding으로부터 extendable한 query vertex u의 candidate set $C_M(u)$를 저장한다. 이 때 partial embedding에 이미 들어있는 원소는 제외된다.

여기서 level은 현재까지 partial_embedding에 들어있는 원소의 개수를 의미한다. 즉, partial_embedding에 새로운 (query vertex, data vertex)를 추가하면 [level]의 정보를 이용해 [level+1]의 정보를 구축하게 된다.

위에서 설정한 네 개의 자료구조를 유지하고 업데이트 하는 방법은 다음과 같다. 우선 **visited_u[]**, **visited_v[]**를 유지하는 방법은 간단하다. 새롭게 partial embedding에 (u, v) 쌍을 추가하면 visited_u[u]=true, visited_v[v]=true로 바꾸고 (u, v)쌍을 제거할 때에는 이 값들을 false로 바꿔준다. **extendable_u[][]**, **extendable_v[][]**를 유지하는 방법은 다음과 같다.

모든 query vertex u_i는 다음 세 가지 중 하나의 경우에 해당하며 각 경우의 대처방법은 다음과 같다.
(1) 이미 partial embedding에 포함되거나 새롭게 partial embedding에 포함될 u이다.
extendable_u[level][u_i]=false로 설정하고 extendable_v[level][u_i]는 비어있는 vector가 된다.
(2) 이전의 partial embedding으로부터 계산된 extendable vertex이다.
기존에 partial embedding으로부터 extendable하다는 것이 검증완료되었기 때문에 새롭게 포함되는 u, v와 연결관계만 파악하면 된다.
(3) partial embedding에 포함되는 u에 의해 새롭게 extendable vertex가 되는 경우.
기존의 partial embedding과 query graph 상에서 연결되어있지 않았기 때문에 새롭게 추가되는 것이므로 이도 (2)에서와 마찬가지로 partial embedding에 새롭게 포함되는 u, v와 연결관계만 파악하면 된다.

만약 해당 partial embedding이 최종적으로 문제의 solution에 도달하지 못하는 경우

return을 하면서 partial embedding을 하나 제거하게 되는데, 이 경우에는 특별한 작업이 발생하지 않는다. 모든 작업은 partial embedding에 새로운 원소가 추가될 때 기존의 정보를 가지고 새롭게 extendable에 대한 정보를 구축하는 것이다.

아래는 우리가 구상한 업데이트 방법을 pushUV()라는 알고리즘으로 표현한 것이다.

Algorithm : pushUV(P, u, v)

Input : partial embedding P, query vertex u, data vertex v
// level은 u, v를 추가시킨 partial embedding의 size이다.
// P는 현재까지 저장된 partial embedding을 의미한다.

$P \leftarrow P \cup \{(u,v)\}$; partial embedding에 새로운 u, v를 추가;
visited_u[u] ← True; visited_v[v] ← True; u, v를 방문했음을 표현;
level ← $P.size()$;
foreach w in query vertex
  if visited_u[w] = True **then**
    return;
  if extendable_u[level-1][w] = True **then**
    extendable_u[level][w] ← True;
    w를 기준으로 extendable_v[level][w]를 업데이트한다.
  else if w, u가 query graph 상에서 연결되어있으면 **then**
    extendable_u[level][w] ← True;
    w를 기준으로 extendable_v[level][w]를 업데이트한다.

[3] Matching Order

Subgraph matching, 즉 graph isomorphism이 NP-complete 문제임은 이미 학습하였다. 조금 다 자세하게 이야기하자면, NP-hard 문제라고 밝혀진 Clique 문제를 graph isomorphism으로 매핑하는 polynomial reduction 함수가 존재하고, 따라서 graph isomorphism이 NP-hard인 것이다. 이를 다시 말하면, sub-graph를 효과적으로 찾는 알고리즘은 존재하지 않는다는 것이다. 다시 말해, 우리는 어떠한 '가정'에 기본한 approximation을 통해서 matching order를 정해야 할 것이다. 어떠한 휴리스틱에 기본하여 matching order를 선정하였다. 그 어떠한 휴리스틱이란 다음과 같다.

매 순간, 우리는 query graph에서 선택할 vertex u_t를 선택한다. 그리고 그 선택은 선택되지 않은 vertex set $U_{rest} = V(q)/u \in M$의 선택지를 최대한 줄일 확률이 가장 큰 선택이다.

결과적으로, 우리는 매 선택으로 candidate space의 크기를 최대한 줄여야 한다는 DAF 논문의 휴리스틱과도 일치하는 개념이다. 단, 본 휴리스틱에서 "확률"이라는 단어에 집중할 필요가 있다. 매 순간 어떤 선택이 최선인지 파악하는 것은, 각 선택에 수반되는 state space의 sub-tree에 대한 정보를 완전히 알고 있지 않다면 불가능하다. 따라서 우리는 "definitely best approach"가 아니라 "probably best approach"를 기반으로 matching order를 선정해야 할 것이다. (definitely best choice를 알고 있다면 이 문제는 더이상 NP-hard가 아니기 때문이다.)

위와 같은 휴리스틱을 이용해서 우리는 총 세 가지 adaptive matching order를 선정했다. **첫째, DAF 논문에 사용된 것과 같은 candidate size가 최소인 vertex를 next vertex로 선정하는 방법이다.** 이는 다음과 같은 이유에서 기인한다. 우리가 어떠한 시간 t에 vertex u'을 선정하였고, $C_M(u')$의 크기가 1000개라고 이야기했을 때, 다음과 같은 상황이 발생할 수 있다. (u',v_1)의 sub-tree로 travel을 진행했을 때 v_1의 선택과 무관하게 모든 path가 실패했다고 가정하자. 그렇다면 (u',v_2)~(u',v_{1000})의 의미 없는 탐색을 계속해야 한다는 것이다. 이러한 상황을 DAF 논문에서는 failing set으로 간주하여 pruning 기법을 사용하였지만, 해당 기법이 사용 불가하므로, 우리는 이러한 상황을 안 마주치도록 해야 한다. 그러한 첫 번째 방법이 이러한 size가 큰 failing set을 최대한 마지막에 마주쳐, recursion call을 지수적으로 감소하는 방법이다. 또한, $C_M(u)$ size가 적은 vertex를 초반에 마주침으로써, $C_M(u)$ size가 큰 vertex의 candidate set을 줄여나갈 수 있다는 장점도 가지고 있다.

두 번째 방식은 preview $C_M(u)$-size ordering 방식이다. 첫 번째 방식이 가지고 있는 한계는 현재 내가 가지고 있는 candidate set의 size 정보만 가지고 다음 선택을 하기 때문에, 그 그 다음 vertex에서 다음 선택을 진행할 때, 선택지에 candidate size가 큰 vertex밖에 없다면 local에 갇히게 된다. 이러한 단점을 보완하기 위해 t+1번째까지 진행한 선택까지 미리 보고 결정을 내리는 방식이 preview $C_M(u)-size$ ordering이다. 이를 수식으로 표현하면 다음과 같다.

$$u_{next} = argmin(|C_M(u)| + \sum_{u' \in N_u} |C_M(u')|)$$

다만, 매 단계에 다음의 step을 보고 $C_M(u)$ size를 계산하는 것은 비효율적이다. 계산하지 않아도 되는 계산을 지수적으로 하는 꼴이기 때문이다. 따라서, 우리는 이러한 cost의 상한을 $|C_{ini}(u)|$로 근사하였다.

세 번째는, 첫 번째 방법을 조금 보완하고자 고안한 방법이다. 우리는 $C_M(u)$의 크기가 작은 것에만 의존하여 고를 것이 아니라, 그렇게 구한 pair (u,v)로 나머지 candidate space를 최대한 filter를 해야 한다. 우리의 filter 기준은 다음과 같다.

(i) $C_M(u)$의 vertex v를 이미 방문했다면 $C_M(u)$에서 지운다.
(ii) query의 vertex가 연결되었으나, $C_M(u)$의 vertex v가 연결되지 않았으면 지운다.

따라서, 우리는 사전에 candidate space에 있는 vertex v의 frequency를 측정하고, 그 frequency가 높으면 뒤에 그 $C_M(u)$를 filter할 확률이 크다는 것이다. 따라서 만약 candidate size가 같다면, 이러한 frequency로 다음 vertex를 결정하는 order를 사용하였다.

이렇게 세 방법으로 matching order를 선정하여 진행해보았으나, 결과적으로 1번째

matching order가 가장 효과적이었다. 그 performance는 [6] performance에서 설명하도록 하겠다.

[4] Root Selection

위의 matching order를 바탕으로 다양한 root에서 실험을 진행하였다. 각 root마다 recursion call이 크게 차이나는 것을 확인하였으며, 어떠한 데이터셋에 대해서는 root를 어떻게 잡냐에 따라 embedding을 찾기도 하고 아예 못 찾기도 하는 것을 확인하였다. 이는 [3]에서 언급한 failing set의 효과로 여겨진다. Figure 1은 주어진 데이터 셋 중 가장 간단한 셋인 hprd_n1 그래프의 root 변경에 따른 recursion call 횟수를 비교한 그래프이다. 그림에서 확인할 수 있듯이 root를 0으로 무작정 고르는 것보다, 어떠한 특징에 근거하여 root를 골라야 함을 알 수 있다. Figure 2와 같은 큰 데이터 셋에 대해서도 recursion call 수에 대한 root의 역할은 지배적임을 확인할 수 있다.

이보다 더 중요한 경우는, 앞에서 언급한 “아예 못 찾을 수 있는 경우”이다. 우리는 이를 abyss라고 표현하도록 하겠다. Backtracking에 근거한 subgraph matching에서 현명하지 못한 방법으로 vertex로 방문한다면 우리는 의미 없는 탐색을 계속해야 하는 “abyss”에 빠지게 된다. 그리고 그러한 abyss가 어디 있는지는 불행히도 탐색 이전에는 판단이 불가능하다. 따라서 대부분의 방법은 pruning을 통해 adaptive하게 abyss를 벗어나지만, 현재 우리에게는 그러한 방법이 불가능하다. Figure 3, 4를 통해서 이러한 경우가 어떤 경우인지 판단할 수 있다.

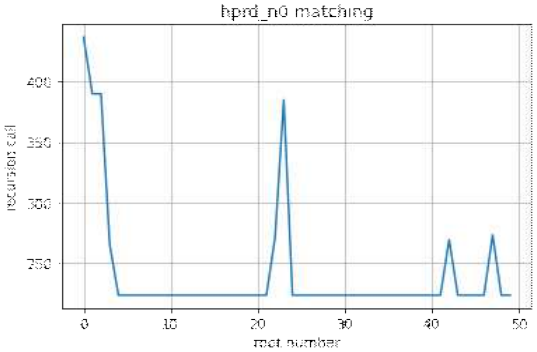


Figure 1. hprd_n1 recursion call

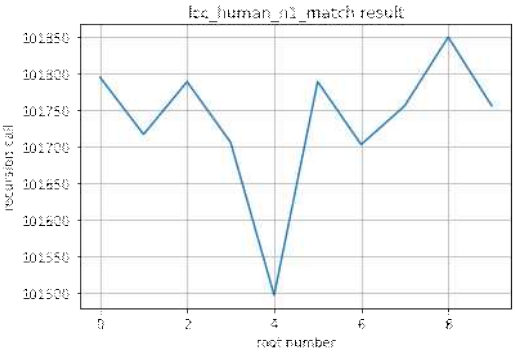


Figure 2. human_n1 recursion call

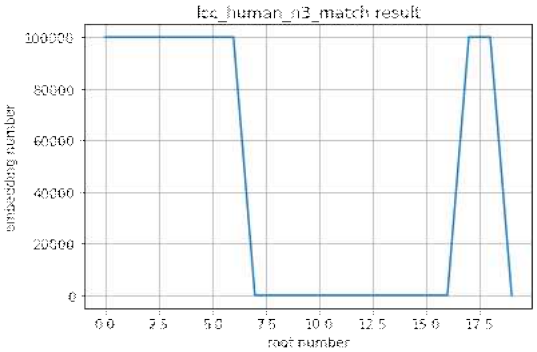


Figure 3. human_n3 embedding number

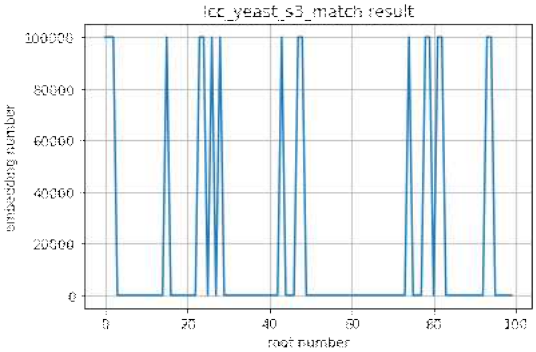


Figure 4. yeast_s3 embedding number

Figure 3은 human_n3의 query graph에 대해서, FIgure 4는 yeast_s3의 query graph에 대해서 root를 바꿔가며 실험을 진행한 결과이다. 두 결과가 완전히 binary하게 구분되는 것을 확인하였다. 이는 embedding의 후보군 어느 지역에 집중해서 몰려있으며, 그렇지 않은 abyss 지역도 존재함을 알 수 있다. 우리는 이러한 결과를 통해서, search space tree가 balance되어 있지 않으며, 결과적으로 한쪽으로 치우쳐져 있는 skewed search space tree를 현재 ordering이 만들어내고 있음을 확인하였다.

우리는 이러한 trade-off 선택의 문제를 마주치게 된다. 첫 번째, 어떠한 ordering의 조절을 통하여 balanced된 search space tree를 만들어내냐. 두 번째, skewed search space tree를 만들고, 이를 확인할 수 있는 적절한 root를 고른다. 현재 우리는 두 번째 알고리즘을 선택하였으며, 현재의 root는 $|C_{ini}(u)|$가 가장 작은 root를 선택한다. 기존 DAF 논문에서는 pruning을 위하여 degree를 위의 값으로 나누어준 것으로 root를 선택하지만, 현재 우리의 알고리즘은 DAG-ordering을 사용하고 있지 않으므로 degree의 영향이 없을 것이라고 가정하였다. 마찬가지로, 이 알고리즘으로 인한 performance는 [7]에서 자세하게 다루겠다.

[5] Check Function

위와 같이 구현한 알고리즘을 통해 subgraph를 찾았다고 해도 올바른 정답이 도출되었는지 파악하기 위해 일일이 비교할 수는 없다. 따라서 우리는 Check()라는 함수를 별도로 만들어서 디버깅할 때 확인하는 과정을 거쳤다. 최종적으로 제출하는 코드에서는 해당 함수를 비활성화 해놓았다. Check()는 다음과 같은 방식으로 작동한다.

Algorithm : Check(P)

Input : embedding P

for i ← 0 **to** $|V_q|$-1

  for j ← i+1 **to** $|V_q|$

    $u_1 \leftarrow P[i]$.first //query vertex를 의미한다.

    $u_2 \leftarrow P[j]$.first //query vertex를 의미한다.

    $v_1 \leftarrow P[i]$.second //data vertex를 의미한다.

    $v_2 \leftarrow P[j]$.second //data vertex를 의미한다.

    if ($v_1 = v_2$) **or** (u_1, u_2 is connected while v_1, v_2 is not connected)

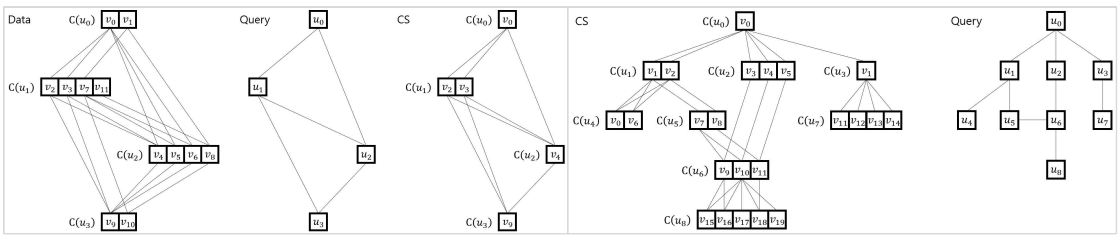
      embedding P 가 틀린 embedding이라고 출력

      return;

embedding P 가 올바른 embedding이라고 출력

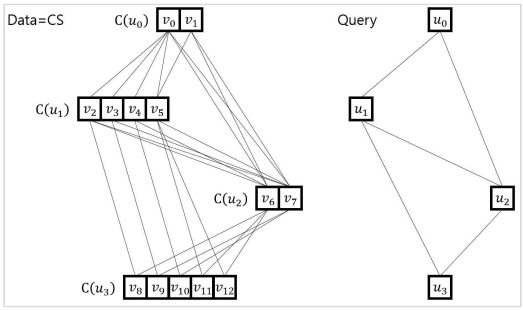
[6] Toy Example

사이즈가 큰 그래프를 가지고는 우리가 올바른 알고리즘을 구현했는지 판단하기 용이하지 않았다. 따라서 우리는 한눈에 과정과 결과를 추적할 수 있도록 다음과 같은 데이터셋을 구축했다.



homework1 homework2

homework1, homework2 데이터셋은 강의자료에 나온 간단하누 예제를 구현해 보았다. 매우 간단한 경우에 해당하기 때문에 중간 출력값을 우리 눈으로 직접 확인하기 용이할 것으로 판한하여 다음과 같이 구현했으며, github repository에 data, query, candidate_set 모두 업로드되어있다.



homework3

homework3은 우리가 문제를 해결하기 위해 선택한 전략에 맞춤형으로 준비했다. 해당 데이터셋의 특징은 root vertex u_0를 선택할 때, v_0, v_1중 어떤 vertex를 선택하느냐에 따라 다음에 선택해야 할 query vertex가 달라지도록 의도적으로 edge를 바꿔놓았다.

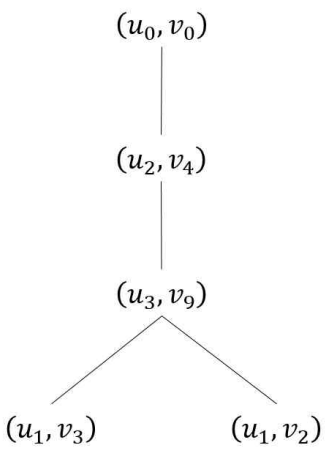
[7] Performance

[6]번 파트에서 생성한 toy example을 우리의 알고리즘으로 돌린 결과를 출력한 콘솔 창을 다음과 같이 첨부했다. 첫 번째 이미지는 subgraph를 (u, v) 순서대로 출력한 것이고, 두 번째 이미지는 state search tree를 유추할 수 있도록 각 iteration에서 partial embedding을 출력한 것이다.

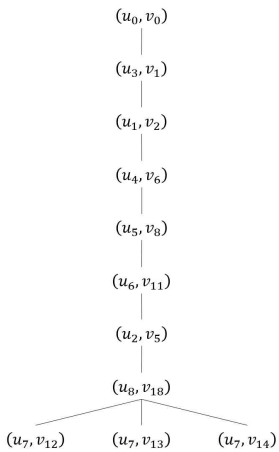
```
vagrant@vagrant:~/algorithm/subgraph_match_challenge/build$ ./main/
k.igraph ../candidate_set/homework.cs
t 4
[TRUE][1]a (0,0) (1,2) (2,4) (3,9)
[TRUE][2]a (0,0) (1,3) (2,4) (3,9)
vagrant@vagrant:~/algorithm/subgraph_match_challenge/build$ ./main/
rk2.igraph ../candidate_set/homework2.cs
t 9
[TRUE][1]a (0,0) (3,1) (1,2) (4,6) (5,8) (2,5) (6,11) (8,18) (7,12)
[TRUE][2]a (0,0) (3,1) (1,2) (4,6) (5,8) (2,5) (6,11) (8,18) (7,13)
[TRUE][3]a (0,0) (3,1) (1,2) (4,6) (5,8) (2,5) (6,11) (8,18) (7,14)
vagrant@vagrant:~/algorithm/subgraph_match_challenge/build$ ./main/
rk3.igraph ../candidate_set/homework3.cs
t 4
[TRUE][1]a (0,0) (1,2) (2,6) (3,8)
[TRUE][2]a (0,0) (1,2) (2,6) (3,12)
[TRUE][3]a (0,0) (1,3) (2,7) (3,9)
[TRUE][4]a (0,0) (1,4) (2,6) (3,12)
[TRUE][5]a (0,0) (1,5) (2,6) (3,11)
[TRUE][6]a (0,0) (1,5) (2,6) (3,12)
[TRUE][7]a (0,1) (1,5) (2,6) (3,11)
[TRUE][8]a (0,1) (1,5) (2,6) (3,12)
```

```
vagrant@vagrant: ~/algorithm/subgraph_match_challenge/build
vagrant@vagrant:~/algorithm/subgraph_match_challenge/build$ ./main/program ../data/homework_data.igraph ../query/homewor
k.igraph ../candidate_set/homework.cs
t 4
[PATH] level : 0 embedding : (0)a
[PATH] level : 1 embedding : (1)a (0,0)
[PATH] level : 2 embedding : (2)a (0,0) (2,4)
[PATH] level : 3 embedding : (3)a (0,0) (2,4) (3,9)
[PATH] level : 4 embedding : (4)a (0,0) (2,4) (3,9) (1,3)
[PATH] level : 4 embedding : (5)a (0,0) (2,4) (3,9) (1,2)
Elapsed time: 0.000156 sec
vagrant@vagrant:~/algorithm/subgraph_match_challenge/build$ ./main/program ../data/homework2_data.igraph ../query/homewo
rk2.igraph ../candidate_set/homework2.cs
t 9
[PATH] level : 0 embedding : (0)a
[PATH] level : 1 embedding : (1)a (0,0)
[PATH] level : 2 embedding : (2)a (0,0) (3,1)
[PATH] level : 3 embedding : (3)a (0,0) (3,1) (1,2)
[PATH] level : 4 embedding : (4)a (0,0) (3,1) (1,2) (4,6)
[PATH] level : 5 embedding : (5)a (0,0) (3,1) (1,2) (4,6) (5,8)
[PATH] level : 6 embedding : (6)a (0,0) (3,1) (1,2) (4,6) (5,8) (6,11)
[PATH] level : 7 embedding : (7)a (0,0) (3,1) (1,2) (4,6) (5,8) (6,11) (2,5)
[PATH] level : 8 embedding : (8)a (0,0) (3,1) (1,2) (4,6) (5,8) (6,11) (2,5) (8,18)
[PATH] level : 9 embedding : (9)a (0,0) (3,1) (1,2) (4,6) (5,8) (6,11) (2,5) (8,18) (7,12)
[PATH] level : 9 embedding : (10)a (0,0) (3,1) (1,2) (4,6) (5,8) (6,11) (2,5) (8,18) (7,13)
[PATH] level : 9 embedding : (11)a (0,0) (3,1) (1,2) (4,6) (5,8) (6,11) (2,5) (8,18) (7,14)
Elapsed time: 0.000692 sec
vagrant@vagrant:~/algorithm/subgraph_match_challenge/build$ ./main/program ../data/homework3_data.igraph ../query/homewo
rk3.igraph ../candidate_set/homework3.cs
t 4
[PATH] level : 0 embedding : (0)a
[PATH] level : 1 embedding : (1)a (0,0)
[PATH] level : 2 embedding : (2)a (0,0) (2,6)
[PATH] level : 3 embedding : (3)a (0,0) (2,6) (1,2)
[PATH] level : 4 embedding : (4)a (0,0) (2,6) (1,2) (3,8)
[PATH] level : 4 embedding : (5)a (0,0) (2,6) (1,2) (3,12)
[PATH] level : 3 embedding : (6)a (0,0) (2,6) (1,4)
[PATH] level : 4 embedding : (7)a (0,0) (2,6) (1,4) (3,12)
[PATH] level : 3 embedding : (8)a (0,0) (2,6) (1,5)
[PATH] level : 4 embedding : (9)a (0,0) (2,6) (1,5) (3,11)
[PATH] level : 4 embedding : (10)a (0,0) (2,6) (1,5) (3,12)
[PATH] level : 2 embedding : (11)a (0,0) (2,7)
[PATH] level : 3 embedding : (12)a (0,0) (2,7) (3,9)
[PATH] level : 4 embedding : (13)a (0,0) (2,7) (3,9) (1,3)
[PATH] level : 3 embedding : (14)a (0,0) (2,7) (3,10)
[PATH] level : 1 embedding : (15)a (0,1)
[PATH] level : 2 embedding : (16)a (0,1) (1,5)
[PATH] level : 3 embedding : (17)a (0,1) (1,5) (2,6)
[PATH] level : 4 embedding : (18)a (0,1) (1,5) (2,6) (3,11)
[PATH] level : 4 embedding : (19)a (0,1) (1,5) (2,6) (3,12)
[PATH] level : 3 embedding : (20)a (0,1) (1,5) (2,7)
Elapsed time: 0.001601 sec
```

두 번 째 이미지를 가지고 state search tree를 재구성하면 다음과 같다.

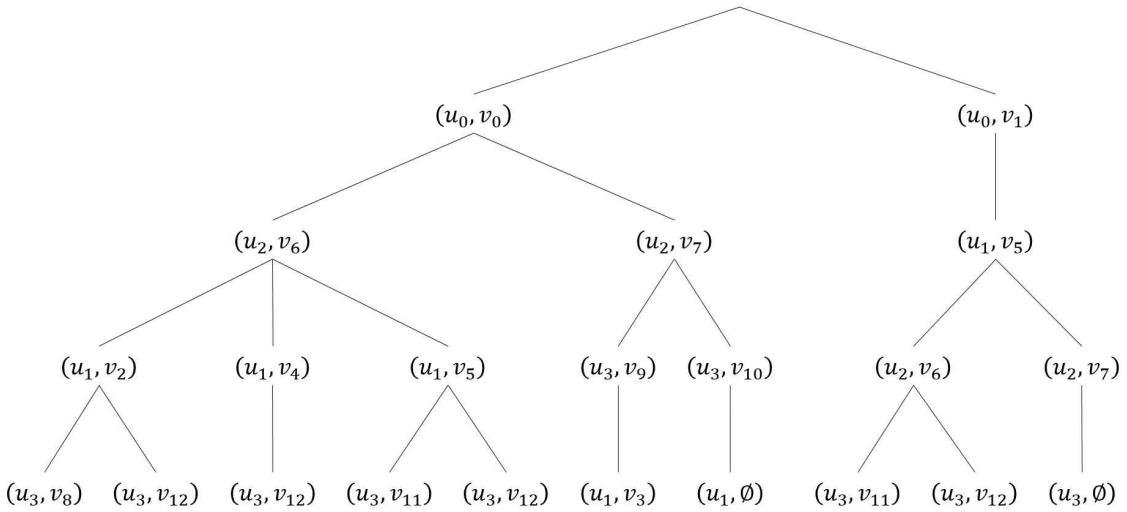


homework1



homework2

homework1, homework2의 경우에는 최대한 root에서 내려갈 때 하나의 candidate data vertex만 고르도록 query vertex를 선정하는 것을 알 수 있다.



homework3

homework3의 state search tree를 살펴보면 우리가 의도한 것처럼 u_0에서 v_0, v_1을 선택할 때마다 다음 query vertex를 선정하는 양상이 달라진다는 것을 알 수 있다. 이러한 능동적 선택을 통해 우리의 알고리즘이 최적의 시간복잡도를 가진다는 것을 확인할 수 있다.

[8] 결론 및 향후 연구 방향

현재 우리의 알고리즘은 분명히 최선으로부터 거리가 멀다. 다만 다음과 같은 지점에서 공헌점이 있다고 생각한다. 첫 번째, backtracking 과정을 Dynamic programming을 이용하여 해결하였다. Memory space는 $O(|V_G|*|V_E|)$만큼 사용한 것에 비해서, 우리는 backtrack을 진행할 때 새롭게 partial embedding에 추가된 vertex만 기준으로 계산을 하면 됨을 밝혀내었다. 이러한 bactracking algorithm은 redundant한 계산을 줄여주는 효과가 있을 것이다. 두 번째, 고정된 matching order에 대해서 root를 현명하게 고르는 방법에 대한 직관을 제공하였다. 총 24개의 데이터 셋에 대해서 root 분석을 진행하였고, 그래프에 대한 고차원적인 feature를 뽑아낼 수 있다면, abyss에 빠질 확률이 가장 적은 root를 고를 수 있을 것이다.
대표적인 방법은, query graph와 candidate space만을 이용하여 Hamilton Path problem의 approximation version을 적용하는 것이다. 결과적으로 recurrence call을 최소화할 수 있는 search space tree의 path는 $|C_M(u)|$ size가 큰 node를 마지막에 방문하는 것이다. 따라서 정해진 root로부터 Breadth First Search를 통해 path의 length를 측정한다. 그리고 이 값을 $|C_{ini}(u)|$와 곱하여 cost를 정하고, 해당 graph 상에서 Hamilton Path problem의 approximation version을 통해 모든 vertex를 방문하면서 가장 작은 weight를 가지는 path를 찾으면 된다. 비록 Hamilton Path 문제도 NP-hard 문제이지만, query의 vertex 개수가 주어진 데이터-셋처럼 작다면, 이러한 문제를 처음에 풀고 얻은 root에서 subgraph isomorphism 문제를 풀기 시작한다면 좋은 결과를 얻을 것이라고 기대한다.

[9] Environments / How to run

운영체제: Ubuntu 20.04.2
GNU/Linux 5.8.0-44-generic x86_64
gcc 9.3.0

Written in C++

▶ How to run
git clone https://github.com/goldenhazard/subgraph_match_challenge.git
rm -rf build
mkdir build
cd build
cmake ..
make
./main/program <data graph file> <query graph file> <candidate set file>