알고리즘 과제 2 보고서

공과대학 기계공학부 2019-19568 이지훈

공과대학 컴퓨터공학부 2019-ABCDE 박재현

**초록**

Subgraph matching은 주어진 undirected, connected, vertex-labeled query, data 그래프에 대해 query vertices에서 data vertices로의 가능한 모든 injective homomorphism, 즉, embedding을 찾는 문제로 NP-hard 문제이다. 이 문제를 푸는 다양한 알고리즘이 연구되어 왔는데, 그 중 최근에는 DAF라는 높은 성능의 알고리즘이 제안되었고, 이는 DAG-Graph DP와 DAG- ordering을 통해 문제를 푸는 알고리즘이다. 이번 과제에서는 DAG-Graph DP를 통해 이미 filtering한 Candidate Set에 대해 제한 시간 안에 최대한 많은 embedding을 찾을 수 있는 DAG-Ordering을 찾아 implement하는 것을 목표로 했고, 이 때 구현을 위해 C++를 사용했다. 또한, DAG-Ordering을 위한 Adaptive Matching Order를 정하기 위해 각 extendable Vertex 에 대해 가 최소인 순으로 Adaptive Matching Order를 정했고, 을 바꿔가며 구현된 프로그램의 성능을 비교했다. 그 결과 이 0.75일 때 가장 최적의 성능을 보였다. 한편 왜 특정 값에서 높은 성능이 보이는지, 이 값이 정해지는 이론적 근거가 있는지에 대한 연구가 아직 미진해 이에 대해 추가로 연구해보아야 할 것이다. 또한, 의 외 요소들의 지수도 변화시켜 보거나, 다양한 조합의 계산으로 를 정하거나, 새로운 요소를 도입하는등 의 형태도 변화시켜가며 최적의 형태가 무엇인지에 대한 후속 연구 또한 필요할 것이다.

**서론**

Subgraph matching은 주어진 undirected, connected, vertex-labeled query, data 그래프에 대해 query vertices에서 data vertices로의 가능한 모든 injective homomorphism, 즉, embedding을 찾는 문제이다.[[1]](#footnote-1) 이 문제는 NP-hard 문제로 잘 알려져있어[[2]](#footnote-2), Turboiso[[3]](#footnote-3), CFL-Match[[4]](#footnote-4) 등 다양한 알고리즘들이 연구되어 왔고, 최근에는 DAF[[5]](#footnote-5)라는 더 빠른 새로운 알고리즘이 제안되었다.

이러한 알고리즘들 모두 공통적으로 다음과 같은 과정으로 이뤄진다.[[6]](#footnote-6) 먼저, 각 query vertex별로 대응될 가능성이 있는 후보 data vertices 들을 각 알고리즘 별 방법을 통해 filtering한다. 이런 각 query vertex 별 후보 data vertices의 집합을 각 query vertex의 Candidate Set이라 한다. 이 후, query 그래프를 traverse하며 backtracking해, 각 query vertex의 Candidate Set에 있는 vertex가 embedding을 성립시키도록 query vertex와 mapping하면서 가능한 모든 embedding 조합을 찾는다. 이러한 과정으로 Subgraph matching 문제를 해결할 수 있는데, 특히 DAF에서는 DAG-Graph DP를 통한 Filtering 과정과 DAG- ordering을 이용한 Backtracking framework를 사용한 알고리즘으로 높은 성능을 보여준다.[[7]](#footnote-7)

이번 과제에서는 DAG-Graph DP를 통해 이미 filtering한 Candidate Set에 대해 제한 시간 안에 최대한 많은 embedding을 찾을 수 있는 DAG-Ordering을 찾아 implement하는 것을 목표로 했다. 특히, 여러 extendable vertex 중 어떤 것을 먼저 extend할지 그 순서를 Adaptive Matching Order라 하는데 이 순서를 정하기 위해, 각 extendable vertex에 특정한 계산식에 따라 float 값을 대응시켜 이 값이 작은 순서대로 extend하는 방식의 순서를 도입했다. 그 계산식은 다음과 같다.

**Equation 1 Adaptive matching Order를 위한Extendable Vertex의 key**

이 때 은 실험을 통해 적절한 값을 찾아보았다. 자세한 pseudo code는 다음과 같다.

PrintAllEmbeddings(data graph, query graph, Candidate Set)

*ExtendableVertex*=Red-Black Tree<key,Vertex>();

for each Vertex *v* in query {

insert *v* in *ExtendableVertex*;

}

vertex is mapped last time=true;

do {

if (vertex is mapped last time) {

extract minimum Vertex *u* in *ExtendableVertex*;

} else { //backtracked before

Get lastly extended query vertex *u* of the partial embedding;

Restore all filtered extendable Candidates in the last iteration;

}

map *u* with a vertex *v* in Candidate set which is not examined;

if (no such *v* exists) {

insert *u* to *ExtendableVertex* back;

vertex is mapped last time =false;

continue;

}

Filter each vertex of extendable Candidates of neigbor of *u* by checking *v* and the vertex is a neighbor;

If (filtering made one of extendable Candidates empty) {

vertex is mapped last time =false;

continue;

}

Extend partial embedding by (u,v);

If (*size of partial embedding*=|*query Vertices*|) {

print embedding;

vertex is mapped last time =false;

} else {

vertex is mapped last time =true;

}

} while (*size of partial embedding*>0)

**Algorithm 1 Backtracking framework based on the DAG-Ordering**

이번 실험에서는 앞에서 소개한 Backtracking framework를 **Algorithm 1**의 방식으로 c++를 통해 implement했다. 또한 여러 benchmark test case를 통해 시간 제한을 1분을 두고 그 동안 출력하는 embedding을 저장했다. 또한, checker program을 inplement해 출력의 각 embedding이 정말로 injective homomorphism인지 검증하고 출력한 총 embedding의 개수를 얻었다. 또한 앞의 **Equation 1**의 값을 바꿔가며 위 과정을 반복해 성능을 비교했다.

**실험 방법**

먼저 실험 기기로는 macOS Big Sur 11.3의 Macbook Pro (15-inch, 2018) , 2.9GHz 6-core Intel Core i9, 32GB 2400MHz DDR4 SDRAM을 사용했고, c++로 프로그래밍해, Apple clang version 12.0.5을 사용했다. 구현한 코드는 각주[[8]](#footnote-8)의 github repository에 있고, **Algorithm 1**이 c++로 implement되어 있다. 또한, build와 실행 방법은 원 Graph Pattern Matching Challenge의 repository[[9]](#footnote-9)에서의 방법과 같은 방법으로 할 수 있다.

또한, 3개의 data graph 별 각각 8개씩의 query graph의 조합인 총 24개의 benchmark test case에 대해, 각 1분 안에 출력하는 embedding output을 저장하고 이를 checker program에 돌려 correctness와 출력된 embedding 개수를 얻었다.

**결과**

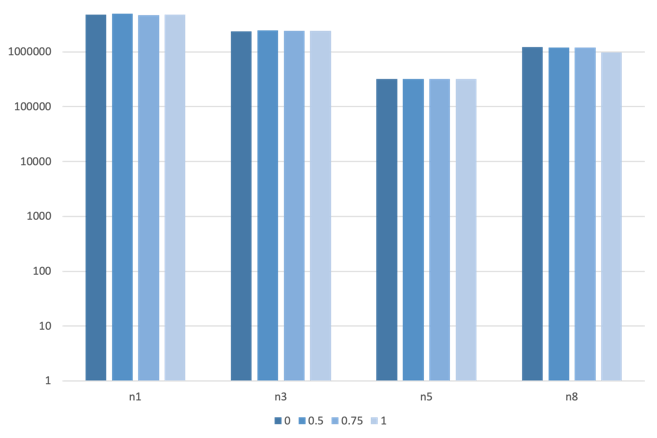


Figure lcc\_yeast n1 n3 n5 n8

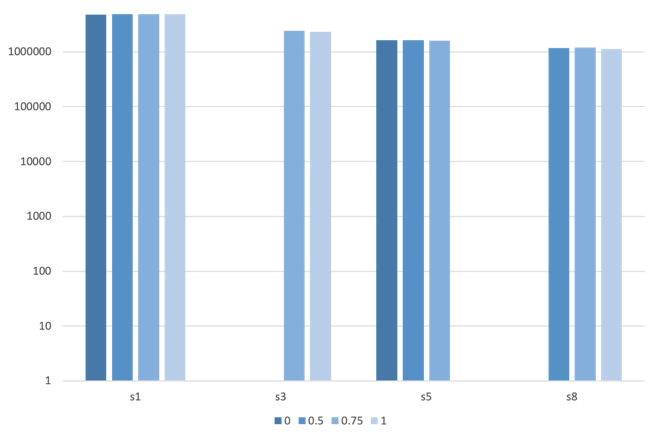


Figure lcc\_yeast s1 s3 s5 s8

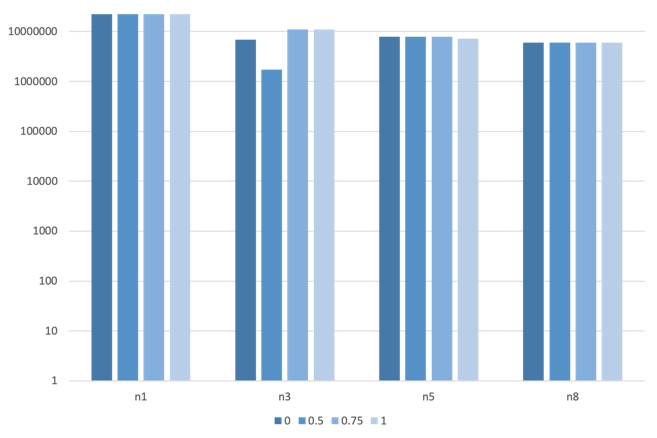


Figure lcc\_human n1 n3 n5 n8

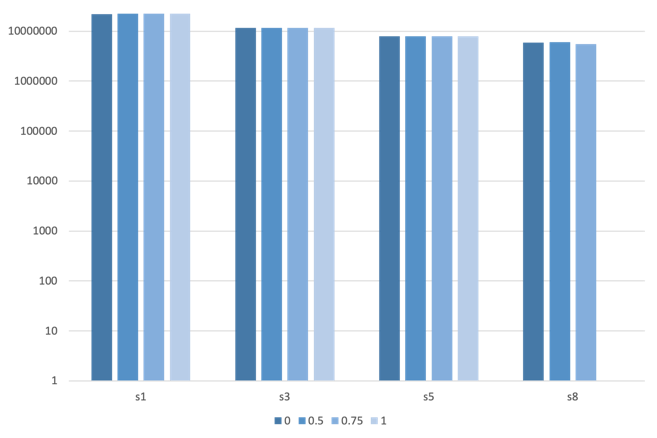


Figure lcc\_human s1 s3 s5 s8

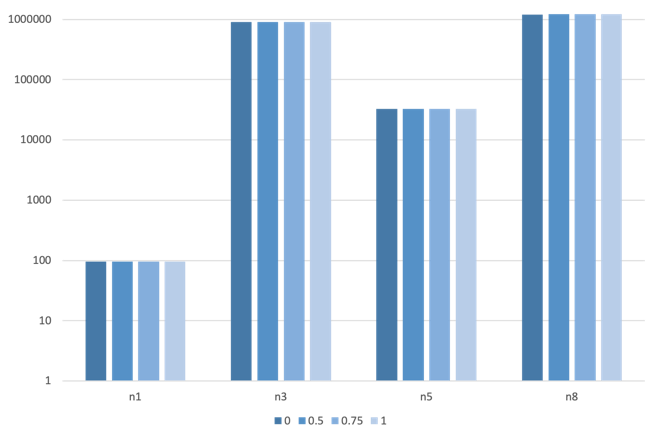


Figure lcc\_hprd n1 n3 n5 n8

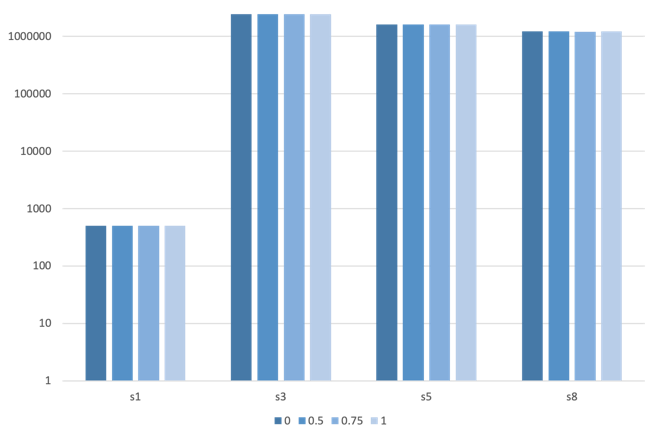


Figure lcc\_hprd s1 s3 s5 s8

먼저 checker program으로 확인한 결과 모든 input에 대해서 구현한 알고리즘은 올바른 embedding 만을 출력한 것을 확인할 수 있었다.

또한, **Figure 1~6**은 **Equation 1**의이 0, 0.5, 0.75, 1인 경우에 각각의 benchmark test case에서 1분동안 출력한 embedding의 개수를 나타낸 그래프이다. 위 그래프를 보면 상당 수의 benchmark test case에서 출력 embedding의 수가 에 무관했지만, Figure 2, 4의 경우 일부 에서 embedding을 출력하지 못하는 경우가 존재했다. 또한, 모든 benchmark test case에서 embedding을 하나 이상 출력한 의 값은 0.75였다. 또, 이 0.75일 때 모든 benchmark test case에 대해 다른 값에 비해 더 많은 embedding이나 비슷한 수의 embedding을 출력했다.

**토의**

먼저 결과를 통해 값에 따라 출력하는 embedding의 수가 차이가 나는 것을 확인할 수 있었고 일부 경우에서 특정 값은 embedding을 출력하지 못하는 것을 확인할 수 있었다. 이는 **Algorithm 1**을 보면 첫 번째 embedding을 출력하기 위해 구조 상 많은 backtracking이 발생할 수 있기 때문에 첫 embedding을 출력하기까지 비교적 시간이 걸리는 것으로 생각할 수 있다. 따라서 별로 corner case가 존재해 유독 첫 embedding 출력까지 시간이 많이 걸리는 경우도 존재한다는 것 또한 알 수 있다.

반면 상당수의 benchmark test case에서 별 출력 embedding 수가 큰 차이가 나지 않았는데, 이는 실험 기기가 노트북이여서 발열에 따라 성능이 고르게 나타나지 못한다는 것을 고려하면, 앞의 corner case를 제하면 Adaptive matching order에 따른 알고리즘 성능이 크게 차이가 나지 않는다고 생각할 수 있다.

실험 설계에서 우리는 vertex degree의 지수만을 바꿔가며 실험했지만, 나머지 요소에 대해서도 지수를 바꾸거나 또한 key를 계산하는 식을 좀 다양한 형태의 계산 조합으로 바꿔보거나 다른 요소를 도입하는 등의 변화를 줘가면서 성능이 차이가 나는지 추가적으로 연구해보아야 할 것이다. 그리고 우리는 이 0.75 인 경우가 종합적인 benchmark test case에서 최적이라고 판단할 수 있었지만, 이러한 이 왜 존재하는지 또, 이 값의 의미는 무엇인지 등에 대한 이론적 근거가 부족하기 때문에, 이것이 subgraph matching 문제의 속성에 의한 것인지 아니면 우연에 의한 것인지 후속 연구 또한 해보아야 할 것이다.

**참고문헌**

1. M. Han, H. Kim, G. Gu, K. Park, and W. 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.
2. M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., 1979.

W. Han, J. Lee, and J. Lee. Turbo iso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In Proceedings of SIGMOD, pages 337–348, 2013.

F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient Subgraph Matching by Postponing Cartesian Products. In Proceedings of SIGMOD, pages 1199–1214, 2016.

1. M. Han, H. Kim, G. Gu, K. Park, and W. 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. [↑](#footnote-ref-1)
2. M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., 1979. [↑](#footnote-ref-2)
3. W. Han, J. Lee, and J. Lee. Turbo iso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In Proceedings of SIGMOD, pages 337–348, 2013. [↑](#footnote-ref-3)
4. F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient Subgraph Matching by Postponing Cartesian Products. In Proceedings of SIGMOD, pages 1199–1214, 2016. [↑](#footnote-ref-4)
5. M. Han, H. Kim, G. Gu, K. Park, and W. Han, op. cit. [↑](#footnote-ref-5)
6. loc. cit. [↑](#footnote-ref-6)
7. loc. cit. [↑](#footnote-ref-7)
8. https://github.com/dlwl0088/PMC21.git [↑](#footnote-ref-8)
9. https://github.com/SNUCSE-CTA/Graph-Pattern-Matching-Challenge.git [↑](#footnote-ref-9)