알고리즘 기말 과제

Graph-Pattern-Match-Challenge

이재필, 도양훈

1. Goal

   주어진 그래프에 대해 subgraph isomorphism을 찾는 것이 이번 과제의 목표였다. Subgraph isomorphism은 대표적인 NP 문제 중 하나로, 탐색 시간을 최대한 줄여 많은 솔루션을 찾아내는 것이 관건이었다.

2. Trials

   과제가 주어진 이후, 구현을 구분하여 할당하기 애매하다는 생각이 들어 팀원이 서로의 방식대로 간단하게 구현한 뒤 feature를 수합하기로 하였다. 이후 처음부터 구현하기보다는 Efficient Subgraph Matching 논문을 참고하여 구현한 뒤 개선 방안을 찾는 방향을 설정하였다.

   논문과 다소 다르게 한 부분은 pre-built order에 따라 search를 진행했다는 점이다. 효율적인 dynamic 구현 방안이 떠오르지 않아 여러 차례 성능을 테스트한 후 pre-built 방식을 채택하였다.

   구현 후에는 여러 차례 수정과 clean up을 거치면서 속도와 안정성을 모두 고려할 수 있었다. 솔루션을 찾는 것까지는 비교적 짧은 시간 안에 구현하였으나, 개선점을 찾는 데에 어려움을 겪었다.

3. Implement

   우리의 implement는 크게 세 가지 feature를 가진다.

   3.1. DAG

   Query된 subgraph를 Directed Acyclic Graph로 표현한다. 이는 DFS/BFS를 통해 그래프를 순회하며 올바른 pattern match를 찾기 위함이다. 처음에는 DAG 의 각 노드를 struct로 구현하였다. 이후 안정성과 활용성을 위하여 DAGNode라는 클래스를 만들었다. 각 node는 DAG 상에서 자신의 부모 노드와 자식 노드를 기억한다.

   주어진 query는 directed graph가 아니기 때문에, 임의로 루트로부터 DAG의 형태로 바꿀 필요가 있었는데, Build함수로 이를 처리하도록 구현하였다. Build 함수는 모든 버텍스를 방문하여 neighbor를 조사한 뒤, neighbor가 이미 방문된 버텍스라면 부모 노드로, 그렇지 않으면 자식 노드로 간주한다.

   DAG를 빌드할 때에는 query의 root node(0번으로 주어진 버텍스)로부터 search를 해나가며 빌드한다. 우리는 DFS와 BFS 방식의 search를 모두 구현 뒤 비교했다. DFS가 성능 면에서 조금 우수한 것을 확인하였지만 큰 차이가 있던 것은 아니었기 때문에 구현의 난이도 등을 고려하여 BFS 방식을 채택하였다.

   3.2. Search Tree Builder

   Search를 수행할 때, pre-built 된 order를 사용하는 것과 runtime에 결정되는 dynamic한 order를 사용하는 것 사이에 딜레마가 있었다. 참고한 논문에서는 dynamic

ordering을 사용했지만, 우리의 구현에서는 order를 구현하는 데에 추가 연산이 필요해 오히려 속도가 느려지는 것을 확인하였다.

3.3. Optimized Neighbor Search

Neighbor를 확인할 때, 매번 tree를 traverse하면서 확인하지 않도록, O(1) 탐색 시간의 배열을 활용하였다. 이를 위해 각 버텍스의 인덱스를 수정하는 필요가 있었지만 그만큼 시간 효율을 얻을 수 있었다.

이외에도 주어진 메모리 공간이 넉넉하다는 점에서 착안해, 벡터나 배열, unordered map을 최대한 이용하였다. 이러한 접근이 dynamic programming을 이용해 subgraph isomorphism 문제를 해결하려는 참고 논문의 접근과 유사하다는 생각으로 구현할 수 있었다.

4. Result

결과 출력 예시

```
$ ./main/program ../data/lcc_hprd.igraph ../query/lcc_hprd_n1.igraph ../candidate_set/lcc_hprd_n1.cs
t 50
a 1047 2330 1293 161 1126 684 2805 1469 541 221 5672 5671 211 5669 5670 123 4468 43 69 1995 685 4455 1106 160 1932 2806 3088 5008 303 131 831 280 404 0 5138 5137 1517 4700 179 8785 1929 110 1376 86 1168 1379 687 266 4723 1684
a 1047 2330 1293 161 1126 684 2805 1469 541 221 5672 5671 211 5669 5670 123 4468 43 69 1995 685 4455 1106 160 1932 2806 3088 5008 303 131 831 280 404 0 5138 5137 1517 4700 179 8785 1929 110 1376 730 1168 1379 687 111 4723 1684
a 1047 2330 1293 161 1126 684 2805 1469 541 221 5672 5671 211 5669 5670 123 4468 43 69 1995 685 4455 1106 160 1932 2806 3088 5008 303 131 831 280 404 0 5138 5137 1517 4700 179 8785 1929 110 1376 937 1168 1379 687 266 4723 1684
a 1047 2330 1293 161 1126 684 2805 1469 541 221 5672 5671 211 5669 5670 123 4468 43 69 1995 685 4455 1106 160 1932 2806 3088 5008 303 131 831 280 404 0 5138 5137 1517 4700 179 8785 1929 110 1376 2599 1168 1379 687 266 4723 1684
(생략)
```

실제 구현 중에는 주어진 그래프가 워낙 방대하기 때문에 작은 크기의

5. Feedback

Candidate set의 수가 많을 때 연산이 현저히 느려지는 것을 확인할 수 있었다. 결국 모든 candidate set을 방문해야 하는 문제의 특성 상, 이를 최대한 효율적으로 구현하는 것이 성능에 핵심적인 역할을 할 것이라고 기대했지만 마땅한 아이디어를 모으지 못한 점이 아쉬웠다.

특히 query의 버텍스 수가 많을수록 candidate set의 수가 exponential하게 증가한다고 분석했는데, 이를 해결하기 위해 여러 매칭 솔루션을 고려해봤지만 time complexity 자체를 획기적으로 개선하는 방안은 찾지 못했기 때문에, 각 연산 과정을 효율적으로 조정하여 최대한 많은 솔루션을 확인할 수 있도록 구현하는 수밖에 없었다.

우리가 시도한 방안은 크게 세 가지였다.

5.1. Clustering candidates

하나의 query 버텍스에 대하여 candidate가 너무 많은 경우, 앞의 candidate에 대해 모든 traverse가 이뤄진 이후 뒤의 candidate에 방문하게 된다는 점이 가장 첫 번째로 지적할 수 있는 문제였다. 이렇게 될 경우, 앞의 candidate가 legal하지 않은 candidate였다면, 무의미한 traverse를 하기 때문에 오히려 legal한 candidate를 traverse하지 못한다는 단점이 존재한다. 이를 해결하기 위해 candidate를 그룹으로 접근하는 방안을 고안하였다.

하지만 결국 필요한 연산량은 같다(결국 모든 candidate set에 방문해야 한다)는 점 때문에 큰 성능 상승은 확인하지 못하였다.

5.2. Randomly select candidates

위의 문제에서 이어지는 trial이었다. 결국 모든 솔루션을 찾는 것이 아니라 최대한 많은 솔루션을 찾는 것을 목표로 한다면, 솔루션이 많이 존재할 것으로 예상되는 구역을 위주로 search한다는 구상이다. 실제로 naïve implementation이 해결하지 못하는 문제 중 하나를 해결하는 성과를 거두었지만 실제 체계적인 구현으로 옮기기에 시간이 부족하여 적용하지 못하였다.

5.3. Clustering query

Candidate set의 총 개수가 (각 candidate set의 수) ^ (query 버텍스의 수) 라는 점에서 착안하여 query 버텍스의 수를 줄이는 시도 역시 해보았다. query를 여러 개의 subquery로 나누어 각 경우의 matching graph를 찾은 뒤 recursive한 접근법이었다. 하지만 matching된 subgraph 사이의 matching을 다시 확인하는 것에 구현 난이도가 높아 완성하지 못하였다. 또한 각 subquery간의 연결 관계를 다시 각 버텍스에 대해 확인해야 하는데, 이것이 기존 방법보다 연산량을 줄인다고 하기 어려우며, 솔루션을 더 빨리 많이 찾는 방법도 아니라고 판단하였다.

참고문헌

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