Graph Pattern Matching Challenge Report

컴퓨터공학부 18학번 김동현

수리과학부 18학번 안정현

0. Refining Candidate Sets

우선, 주어지는 Candidate Set을 (Soundness를 유지하면서) 더 줄일 수 있음을 확인하였습니다. 아래와 같은 iteration을 일정 횟수 반복하여 적용합니다.

1. 논문에서 설명한 DAG DP를 적용합니다. 단, q_D를 사용하는 것이 아닌 Random하게 구성한 Rooted DAG를 사용합니다.
2. 정점의 Local Characteristic을 사용하여 조건에 맞지 않는 정점을 Candidate set에서 제외합니다. 아래와 같은 조건들을 고려합니다.
    - Query Graph의 각 정점 u에서 인접한 정점들의 Degree들을 모은 multiset을 생각합니다. 각 $v \in C(u)$에 대해, v에서 인접한 정점들의 Degree multiset이 u의 multiset을 Cover하지 않는다면 v를 $C(u)$에서 제거합니다. 여기서 multiset A가 B를 Cover한다는 것은 A의 각 원소를 자신보다 같거나 큰 B의 원소에 (중복 없이) 대응시킬 수 있다는 것입니다.
    - Query Graph의 각 정점 u에서 인접한 정점들의 Label들을 모은 multiset을 생각합니다. 각 $v \in C(u)$에 대해, v에서 인접한 정점들의 Label multiset이 u의 multiset을 포함하지 않으면 (일반적인 multiset 포함관계) v를 $C(u)$에서 제거합니다.

위 알고리즘은 `/executable` 폴더에 주어진 binary로 생성한 candidate set을 많은 경우에서 실제로 줄여줍니다. 특히 Data Graph가 Yeast인 경우 Candidate Set의 총 크기 합을 20~30% 정도 줄이는 경우가 빈번하게 발생함을 확인하였습니다. 이는 성능의 개선으로까지 이어집니다.

1. making DAG

논문에서 주어진 q_D는 아래와 같이 구성됩니다.

1. 쿼리 그래프에서 $\frac{|C(u)|}{deg(u)}$ 값이 가장 작은 정점을 루트로 정합니다.
2. 정한 루트로부터 BFS를 수행한 뒤, 거리가 증가하는 방향으로 간선을 연결합니다. 두 정점의 거리가 같다면 Label Frequency, Degree 등을 기준으로 미리 정한 순서에 따라 방향을 정합니다.

루트를 정하는 것은 필수적인 과정이므로 그대로 유지하고, 2번 과정에서도 루트를 정하는 데 사용한 가중치를 사용할 수 있다면 더욱 좋을 것 같아서 아래와 같이 수정하였습니다.

2'. 집합 S에 정점을 하나씩 추가하면서 DAG를 만들어 나갑니다. 초기에 S에는 루트만 있습니다. 이제, 매 iteration마다 "S에 없는 정점 중 S에 속한 정점과 인접한 정점들" 중 $\frac{|C(u)|}{deg(u)}$ 값이 가장 작은 정점을 w라고 합시다. S에서 w에 인접한 모든 간선에 대해 방향을 w로 향하도록 설정해 주고, w를 S에 추가합니다.

위 과정을 통해 만든 DAG가 올바른 rooted DAG임은 귀납법으로 쉽게 증명할 수 있습니다. 이렇게 만든 DAG를 Backtracking 과정에서 q_D 대신 사용한 결과 성능이 개선되는 것을 확인하였습니다.

(성능 개선의 확인은 Appendix. 에서 언급하는 `experiment/stress_test.py`를 통해서 주어진 Test Case보다 더 정밀하게 할 수 있었습니다.)

2. Backtracking

백트래킹의 기본 골자는 논문에서 서술한 방법과 같습니다. Extendable한 정점 중 가장 적합한 것을 고를 때에는 $|C(u)|$를 기준으로 사용합니다. 백트래킹 과정은 아래와 같습니다.

1. Query Graph의 모든 정점이 Embedding이 된 상태라면, 즉 Valid Embedding을 하나 찾았다면 출력한다.
2. 그렇지 않다면, 현재 Extendable한 정점들 중 $|C(u)|$ 값이 가장 작은 정점을 새로 Extend할 정점으로 고른다.

3. $C(u)$에 속한 각 v에 대해, (u, v) Embedding을 Partial Embedding에 추가하고 함수를 재귀호출한다. 호출하기 전/후에 갱신된 Partial Embedding에 따라 $C(u)$를 적절히 변화시켜 준다.

논문에 나온 방법과의 차이점은 이미 한 번 쓰인 Data Graph의 정점을 다시 쓰지 않기 위한 방법입니다. 논문에서는 3번 과정에서 v가 이미 쓰였다면 무시하는 식으로 구현을 하였는데, 저희는 v가 쓰일 때마다 아직 extend되지 않은 Query Graph의 정점 w들에 대해 $C(w)$에 v가 들어있다면 그때그때 지워주는 식으로 구현을 하였습니다. 이 방법의 장점은 $C(w)$의 크기 정보가 더 정확해지는 데 (쓸 수 없는 정점이 그때그때 바로 빠지므로) 있습니다.

Data Graph의 $|V|$가 그렇게 크지 않은 데 착안하여, 간선이 연결되어 있는지에 대한 빠른 판별을 위해 인접 행렬을 추가로 구성하여 사용하였습니다.

3. Experiment Environment

실험은 Google Cloud Platform의 Computer Engine 인스턴스에서 진행하였습니다. Machine 사양은 아래와 같습니다.

- OS : Ubuntu 20.04.2 LTS
- CPU : (Intel(R) Xeon(R) CPU @ 2.00GHz) × 2
- RAM : 4 GB

구현은 Skeleton 코드 바로 위에서 C++로 수행하였습니다. 실행 방법 역시 처음 Skeleton 그대로입니다. 즉, 아래와 같습니다.

```
mkdir build
cd build
cmake ..
make
./main/program <data graph file> <query graph file> <candidate set file>
```

Appendix. About Repository

Challenge 진행 과정에서 새로 추가한 폴더에 대해서만 서술하겠습니다.

`archive/` : `backtrack.cc`에 해당하는 여러 version의 코드들을 모아놓은 폴더. 최종 제출본에 해당하는 코드(`src/backtrack.cc`와 동일한 코드)는 `archive/a.cc` 입니다.

`experiment/` : 코드 성능 실험을 위해 만든 폴더. 폴더 내에서 `/stress_test.py <code filename (in archive/)> <# of trial for each parameter type> <time limit for each trial>` 커맨드를 실행하면 총 36가지의 Parameter type (Data graph 3종류 × Query Graph 정점 수 4종류 × Query Graph 정점 평균 Degree 3종류) 에 대해 해당하는 parameter로 query graph를 랜덤하게 만들고, 코드를 실행하여 그 결과를 취합합니다. `experiment/logs/` 폴더 내에 여러 실험 결과들이 들어 있습니다.

`report/` : 보고서가 있는 폴더입니다.