Algorithm: Graph Pattern Matching Challenge

지구환경과학부 2018-19797 박수빈
수리과학부 2018-19880 팽진희

이번 프로젝트에서는 논문 <Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together>을 참고하여 Subgraph Pattern Matching 알고리즘을 작성했다. 본 보고서에는 논문의 DAP algorithm을 구성하는 세 단계인 buildDAG, buildCS, backtrack 각각의 과정에서 성능을 올리기 위한 시도를 기술했다.

1. BuildDAG

```
// backtrack.cc
std::vector<std::pair<double, Vertex>> rank;
for (size_t i = 0; i < queryVertexNum; i++) {
    rank.push_back(std::make_pair(cs.GetCandidateSize(i) / (0.01 *
        queryVertexNum / data.GetNumLabels() + query.GetDegree(i)), i));
}
std::sort(rank.begin(), rank.end());
```

DAF 알고리즘에서 가장 먼저 하는 일은 DAG ordering을 따라 Backtracking을 진행하기 위해 기존에 주어진 query q로부터 queryDAG q_d를 만드는 일이다. Rooted DAG q_d의 Topological order 를 기반으로 backtrack 기법을 사용하게 되면 non-tree edges들을 사전에 확인함으로써 성능을 개선할 수 있다. DAG를 만드는 데 있어서 가장 먼저 해야하는 일은 적합한 root를 결정하는 것인데, 좋은 root는 (1) candidate set의 크기인 $|CS(u)|$가 작아야 하고, (2) query q에서의 간선의 수인 $deg_q(u)$가 작아야 한다. 이러한 조건을 충족시키기 위해 논문에서는 아래와 같이 Candidate set의 크기와 u의 간선의 개수의 비를 최소화하는 방식으로 DAG의 root를 설정한다.

$$r \leftarrow argmin_{u \in V(q)} \frac{|CS(u)|}{deg_q(u)}$$

하지만 이 방식의 경우 $deg_q(u)$의 크기에 따라 $|C_{ini}(u)|$ 또는 $deg_q(u)$ 두 인자 중 하나의 영향력이 지나치게 커진다는 단점이 있다. 이번 프로젝트에선 $deg_q(u)$가 작아야 유리하다는 아이디어에 더하여, 레이블 당 query q에서의 vertex수 또한 중요하다 생각하여, 이 값을 이용하여 약간의 보정을 진행하였다. 아래는 우리의 최종적 DAG의 root 설정 방식이다. 다양한 k값을 시도해 본 결과, $k = 0.01$일 때의 성능이 가장 우수했다.

$$r \leftarrow argmin_{u \in V(q)} \frac{|CS(u)|}{deg_q(u) + k \times \frac{|V_q|}{|Label_G|}}$$

더하여, DAG의 edge들의 방향성을 선정해주기 위하여 위의 root 설정 방식을 응용하였다. DAG의 결과를 Topological sort한 결과가 결국 아래의 rank값의 오름차순이 될 수 있도록 하였다. 이번 프로젝트에서는 CS optimization의 진행이 불필요하기 때문에 DAG를 생성하진 않았고, 이에 대응되는 Topological order만 줌으로써, Backtracking을 실행할 시, 이 순서를 고려하여 DFS가 진행되도록 설계하였다.

$$rank(u) = \frac{|CS(u)|}{deg_q(u) + k \times \frac{|V_q|}{|Label_G|}}$$

2. BuildCS

```
// candidate_set.cc
if (useCSrank) {
    // code for sorting CS in degree size order
   auto compare = [&data](Vertex u, Vertex v) {
       return (data.GetDegree(u) < data.GetDegree(v));
   };
   std::sort(cs_[id].begin(), cs_[id].end(), compare);
}
```

Backtracking 과정에서는 partial embedding을 확장하기 위해 candidate set의 아직 방문되지 않은 원소들을 차례로 순회하며 candidate set의 원소가 embedding에 포함될 수 있는 조건을 충족시킬 시 partial embedding에 원소를 포함시킨다. 이 과정에서 candidate set의 순회에서 초반에 등장하는 원소들이 partial embedding에 포함될에 포함될 수 있는 확률은 $deg_G(u)$값이 클수록 높다. 이번 프로젝트에서는 성능을 높여주기 위해 CS의 원소들을 $deg_G(u)$가 높은 순으로 정렬한 뒤 저장하도록 개선하였다.

3. Backtrack

Backtrack의 경우 논문에 주어진 pseudocode는 재귀적 방법을 사용하였으나, 프로젝트의 구현에서는 재귀적으로 함수를 호출하는 과정에서 발생한는 overhead를 최소화하기 위해 stack 구조를 이용해 iterative하게 구현했다. Main 함수에서 주어진 조건에 따라 q 또는 q_D의 원소를 차례로 방문하며 candidate set의 원소 중 아래와 같이 injective mapping 조건을 충족시킬 경우 partial embedding에 정점을 추가했다.

```
// backtrack.cc
// for all vertices in advance of vertex(queryVertex)
for (Embedding ::const_iterator iter = embedding.begin(); iter != embedding.end(); iter++) {
    if (!data.IsNeighbor(dVertex, iter->second) && query.IsNeighbor(qVertex, iter->first)) {
        edgeMatches = false;
        return edgeMatches;
    }
}
```

4. Query/Candidate Set 조합

이번 알고리즘 성능 개선 과정에서는 buildDAG와 CS sorting을 순차적으로 수정했는데, 도입 과정에서 모든 input case에 대해 성능 개선이 이루어지는 대신 일부 input에서만 성능이 개선되고 또 다른 input에서는 성능이 오히려 악화된다는 사실을 발견했다. 이에 따라 최적화된 query와 candidate set 조합을 찾기 위해 아래와 같이 (1) rank에 따라 query DAG를 생성한 경우와 그렇지 않은 경우, (2) candidate set을 degree 순으로 정렬한 경우와 그렇지 않은 경우에 대한 실험을 진행했다. 주어진 모든 input에 대하여 각 query와 candidate set 조합을 이용해 성능을 측정한 결과는 아래와 같다.

대부분의 input 경우 모든 조건에서 동일한 개수의 embedding을 찾아냈으나, 초록색으로 강조된 일부 input의 경우 특정 DAG와 candidate set 조합 하에서만 embedding을 찾아내는 경우가 있었다. 에 따라 다른 개수의 embedding을 찾는 경우가 있었다. Human N8, Human S8, Yeast N8, Yeast S1의 경우 DAG/No CS order의 조건 하에서는 embedding을 찾지 못하는 대신 new DAG/CS order의 조건 하에서는 embedding을

빈 실행시간의 경우 완료 전 30초 제한으로 timeout			DAG				New DAG			
			No CS order		CS order		No CS order		CS order	
			# of outputs	time(s)	# of outputs	time(s)	# of outputs	time(s)	# of outputs	time(s)
HPRD	HPRD_N	1	96	0.08	96	0.09	96	0.1	96	0.08
		3	100000	6.44	100000	6.49	100000	6.72	100000	6.53
		5	32832	4.18	32832	4.08	32832	4.2	32832	4.38
		8	100000	20.24	100000	19.26	100000	20.56	100000	20.26
	HPRD_S	1	504	0.08	504	0.09	504	0.09	504	0.09
		3	100000	6.5	100000	6.17	100000	6.53	100000	6.55
		5	100000	12.75	100000	12.07	100000	12.97	100000	12.84
		8	94850		65009		97751		60514	
HUMAN	HUMAN_N	1	100000	0.79	100000	0.73	100000	0.73	100000	0.78
		3	100000	5.45	0		0		0	
		5	100000	1.49	0		100000	1.46	0	
		8	0		100000	2.31	0		100000	2.4
	HUMAN_S	1	100000	1.77	100000	0.66	100000	2.34	100000	0.74
		3	100000	23.99	100000	4.95	100000	1.49	100000	9.55
		5	100000	1.41	0		100000	1.49	0	
		8	0		0		0		100000	1.91
YEAST	YEAST_N	1	100000	2.45	100000	2.29	100000	2.55	100000	2.5
		3	0		0		0		0	
		5	100000	21.18	100000	20.97	100000	21.14	100000	22.15
		8	0		100000	28.46	0		100000	29.22
	YEAST_S	1	0		100000	3.49	100000	6.09	100000	2.83
		3	0		0		0		0	
		5	0		0		0		0	
		8	0		0		0		0	

찾아냈으며, Human N3, Human N5, Human S5의 경우에는 new DAG/CS order 하에서는 embedding을 찾지 못했지만 DAG/No CS order의 조건 하에서는 embedding을 잘 찾았다. 따라서 DAG/No CS order, 그리고 new DAG/CS order의 두 조합을 통해 대부분의 input에서 embedding을 찾아낼 수 있을 것이라고 생각해 아래와 같이 new_candidate_set 함수와 PrintAllMatches 함수에 DAG 생성과 CS ordering을 조정하기 위한 boolean input을 추가하고, new DAG/CS order를 먼저 시도 후 충분한 시간이 지난 후에도 embedding을 하나도 찾지 못했을 경우 DAG/no CS order 조합으로 다시 embedding을 찾을 수 있도록 코드를 수정했다.

```
// main.cc
bool success = backtrack.PrintAllMatches(data, query, candidate_set, true);
if (!success) {
    CandidateSet new_candidate_set(candidate_set_file_name, data, false);
    backtrack.PrintAllMatches(data, query, new_candidate_set, false);
}
```

5. 최종 결과

이번 프로젝트에서는 1차적으로 주어진 graph와 query set, 그리고 candidate set에 대하여 (1) query DAG를 앞에서 기술한 rank에 따라 정렬하고, candidate set을 degree에 대한 내림차순으로 정렬 후 backtracking을 시도한 후 실패 시 (2) 주어진 기존의 query와 candidate set을 이용해 다시 backtracking을 시도했다. 그 결과 Yeast N3와 Yeast S3, S5, S8을 제외한 모든 input에 대하여 성공적으로 embedding을 찾아내는 것을 확인할 수 있었다. 아래는 최종 알고리즘의 실제 작동 시간(Real)과 CPU time(User)을 측정하여 표로 나타낸 것이다.

			Real	User	Sys	# of embeddings found
HPRD	HPRD_N	1	0m1.177s	0m0.073s	0m0.016s	96
		3	0m2.828s	0m2.532s	0m0.273s	100000
		5	0m1.364s	0m1.247s	0m0.104s	32832
		8	0m5.340s	0m4.942s	0m0.334s	100000
	HPRD_S	1	0m0.107s	0m0.082s	0m0.022s	504
		3	0m2.845s	0m2.538s	0m0.283s	100000
		5	0m4.807s	0m4.471s	0m0.310s	100000
		8	32.605s	32.014s	0m0.461s	100000
HUMAN	HUMAN_N	1	0m0.725s	0m0.469s	0m0.247s	100000
		3	0m25.508s	0m24.886s	0m0.558s	100000
		5	0m21.260s	0m19.587s	0m1.583s	100000
		8	0m1.382s	0m1.116s	0m0.251s	100000
	HUMAN_S	1	0m0.642s	0m0.406s	0m0.230s	100000
		3	0m8.995s	0m8.647s	0m0.331s	100000
		5	0m21.161s	0m20.555s	0m0.568s	100000
		8	0m1.408s	0m1.147s	0m0.247s	100000
YEAST	YEAST_N	1	0m1.595s	0m1.333s	0m0.251s	100000
		3	NOT_FOUND	NOT_FOUND	NOT_FOUND	0
		5	0m8.710s	0m8.314s	0m0.343s	100000
		8	0m7.583s	0m7.108s	0m0.441s	100000
	YEAST_S	1	0m2.056s	0m1.762s	0m0.276s	100000
		3	NOT_FOUND	NOT_FOUND	NOT_FOUND	0
		5	NOT_FOUND	NOT_FOUND	NOT_FOUND	0
		8	NOT_FOUND	NOT_FOUND	NOT_FOUND	0

6. 실행 방법 및 구현 환경

이번 프로젝트를 진행한 환경은 아래와 같다.

- OS : macOS Big Sur 11.4
- CPU : 2.6 GHz 6코어 Intel Core i7
- Mem : 16GB 2667 MHz DDR4
- C++ : gnu++14

또한, 이를 터미널에서 실행시키기 위해서는 다음의 커맨드를 입력하면 된다.

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