Graph Pattern Matching Challenge Report

2016-12632 백상원
2017-17288 박성훈

1. Background

본 Graph pattern matching의 구현은 Reference [1]을 상당부분 참고하고 있으므로, [1]의 내용을 중심으로 핵심 알고리즘을 정리하고자 한다.

(1) Notations

Notations	Descriptions
$q,\ G$	query graph, data graph
$V(g),\ E(g)$	vertex set and edge set of graph g
$d(u),\ N(u)$	degree and neighbors of vertex u
$e(u,v)$	edge between u and v
$u.core$	core value of vertex u
π	matching order
$C,\ P$	candidate set and pivot dictionary
$C(u)$	candidates of u in C
$BN_q^{\pi}(u)$	backward neighbors based on the matching order π in the vertex u of query q
BI	bigraph index
$BI_u^{u'}$	bigraph between C(u) and C(u')
$BI_u^{u'}(v)$	neighbors of v in C(u) where v ∈ C(u')
i	index : start from 1

Table1. Notations for algorithm. [2]

다음은 수업시간에 다루지 않은 non-trivial한 요소들에 대한 구체적인 설명이다.

k-core : 그래프 g에 대해 k-core of g는 $d(u) \geq k, \forall u \in V(g')$를 만족하는 g의 maximal connected subgraph g'를 의미한다.

core value : g의 vertex u의 core value. $u.core$란 u가 $k-core$에는 속하지만 어떠한 $(k+1)$-core에도 속하지 않는 경우에서 k값을 말한다.

pivot : query vertex u의 pivot v는 matching order π를 기준으로 u의 이전에 있는 neighbor 중 하나이다 pivot을 통해 u의 후보 중 적절한 것을 택한다.

pivot dictionary : $P[u]$, u의 pivot

bigraph index : $BI_u^{u'}(v)$는 $v \in C(u')$의 neighbor 중 $C(u)$의 원소들의 집합이다. $BI_u^{u'}$는 $C(u)$와 $C(u')$사이의 graph이며 본 알고리즘에서 u'은 $P[u]$이다.

(2) Cost model

matching order π에 대해 $|\pi| = |V(q)|$가 성립한다. 이를 n이라 하자. 그러면 π를 따라 embedding을 찾는 과정을 depth가 $n-1$인 tree로 생각할 수 있다. 이제 i 번째 depth에서의 cost를 T_i라 두고, 전체 cost를 T_{iso}라 하자. 그러면 $T_{\text{iso}} = \sum_{i=0}^{n-1} T_i$ 가 성립한다.

한편, N_i를 i 번째 depth에서 node 개수라고 하고 b_j^i를 i 번째 depth의 j 번째 node의 child 수, b^i를 i 번째 depth에서 평균 child 수라고 하자. 그러면 $T_0 = N_1 = |C(\pi[1])|$, $T_i = \sum_{j=1}^{N_i} b_j^i = N_i \times b^i, (1 \le i \le n-1)$이 성립한다.

이제 a_i를 i 번째 depth의 각 node에서 평균 matching 성공률이라 하자. 그러면 $N_{i+1} = T_i \times a_i$라 할 수 있다. 따라서, 다음 식이 성립한다.

$$T_i = \begin{cases} N_1 & i = 0 \\ N_1 b^1 & i = 1 \\ N_1 b^i \prod_{j=1}^{i-1} a_j b^j & 2 \le i \le n-1 \end{cases} \Rightarrow T_{\text{iso}} = \mathrm{N}_1(1 + \mathrm{b}^1 + \sum_{\mathrm{i}=2}^{\mathrm{n}-1} \mathrm{b}^{\mathrm{i}} \prod_{\mathrm{j}=1}^{\mathrm{i}-1} \mathrm{a}_{\mathrm{j}} \mathrm{b}^{\mathrm{j}})$$

(3) Estimate

T_{iso}를 minimize 하기 위해 parameter a_i, b^i의 estimator를 찾는다.
먼저 $u = \pi[i+1]$, $u' = P[u]$이라 하자. 이때 i 번째 depth의 j 번째 node에서 u'이 v'으로 match된 상태라고 하자. 그러면 u의 matching을 v'과 이웃인 vertex 중에서 고르므로 (아래의 enumeration algorithm 참조) $b_j^i = |BI_u^{u'}(v')|$가 된다.

이를 통해 b^i의 estimator $\tilde{b^i} = \frac{|E(BI_u^{u'})|}{|C(u)|}$를 얻을 수 있다.

또한 $|BN_q^{\pi}(u)|$가 클수록 성공률이 커진다고 생각할 수 있으므로 $\tilde{a_i} = \frac{1}{|BN_q^{\pi}(u)|^2}$

으로 둘 수 있다. 따라서, 이 estimator들을 위 T_{iso}대입한 $\widetilde{T_{\text{iso}}}$를 최소화하는 것이 목표이다. 그러나 이 역시 어려우므로 greedy approach를 통해 대략적으로 근사하는 방식을 택한다. 우리는 [1]에서 제안한 방식에 따라 $\tilde{a_i} * \tilde{b^i}$를 최소화하는 vertex를 greedy하게 선택하여 matching order를 구성하는 것을 목표로 한다.

2. Implementation

1) GenerateMatchingOrder

matching order를 construct하는 함수이다. 추가적으로 P, BN_q^π도 construct 한다. 전체적인 pseudo code는 다음과 같다.

Input: a query graph q, a data graph G and the candidate set C
Output: a matching order π, a pivot dictionary $\mathcal{P}$ and the backward neighbors BN_q^π

1 **begin**
2 $q^w \leftarrow GenerateWeightedGraph(q, G, C)$;
3 $V_C \leftarrow \{u \in V(q) | u.core \geqslant 2\}, V_{NC} \leftarrow V(q) - V_C$;
4 $\mathcal{P} \leftarrow \{\}$, $w^*[u] \leftarrow |V(G)|$ for all $u \in V(q)$;
5 Set $BN_q^\pi(u)$ to empty for all $u \in V(q)$, set UN to empty;
6 $u^* \leftarrow arg\min_{u \in V_C} \frac{|u.C|}{u.core}$, $\pi \leftarrow (u^*)$;
7 **foreach** $u \in N(u^*) - \pi$ **do**
8 $BN_q^\pi(u) \leftarrow BN_q^\pi(u) \cup \{u^*\}$;
9 **if** $w(u^*, u) \leqslant w^*[u]$ **then**
10 $w^*[u] \leftarrow w(u^*, u)$, $\mathcal{P}[u] \leftarrow u^*$;
11 Add u to UN if $u \notin UN$;
12 **while** $|\pi| < |V_C|$ **do**
13 $u^* \leftarrow arg\min_{u \in UN \cap V_C} \frac{w^*[u]}{|BN_q^\pi(u)|^2}$;
14 Add u^* to π, remove u^* from UN;
15 Same as Lines 7-11;
16 **while** $|\pi| < |V(q)|$ **do**
17 $u^* \leftarrow arg\min_{u \in UN \cap V_{NC}} \frac{w^*[u]}{d^2(u)}$;
18 Add u^* to π, remove u^* from UN;
19 Same as Lines 7-11;
20 **return** $(\pi, \mathcal{P}, BN_q^\pi)$;

Figure1. Pseudo code for GenerateMatchingOrder function. [3]

먼저 $\tilde{b^i}$를 구하기 위해 input query를 이용하여 weighted directed query를 구한다. 이 과정을 GenerateWeightedQuery라는 이름의 함수로 구현하였다.

다음은 query vertex들의 core value를 구한다. ComputeCoreValue라는 이름의 함수로 구현하였다. pivot의 경우 vertex u의 backward neighbor 중 weighted query에서 $e(u^*, u)$가 최소인 vertex u^*로 결정한다.

이제 matching order를 구성하는데, 위에서 설명한대로 $\tilde{a_i}*\tilde{b^i}$가 가장 작은

vertex들을 greedy하게 택한다. 그러나 이때 core structure에 속하는 vertex들이 그렇지 않은 vertex들 보다 우선적으로 택해지도록 만든다. core structure란 core value가 2 이상인 vertex들을 말한다. core value가 높은 것을 우선적으로 배치하는 이유는 query의 dense part에서 진행하는 것이 성능이 좋기 때문이다.

아직 택해지지 않은 non core structure의 vertex의 경우 현재 $|BN_q^{\pi}(u)|$과 u.core가 모두 1이므로 분모 값을 $d(u)$로 대체하여 greedy approach를 적용한다.

core structure에서 $\tilde{a_i}*\tilde{b^i}$가 tie인 경우 다음과 같이 tie handling을 진행한다.

(1) $u.core$가 큰 vertex를 택한다.

(2) $u.core$도 같을 경우 $|C(u)|$가 작은 vertex를 택한다.

한편, [1]에서는 첫 번째 vertex를 택하는 것을 core structure 중 $\frac{|C(u)|}{u.core}$가 최소인 u를 택하는 것으로 하였지만 본 challenge에서 사용되는 graph에 대해서는 $u.core$가 큰 vertex를 택하는 것이 더 좋은 성능을 보였다. 따라서, first vertex를 선택하는 방식으로 core structure에서 $u.core$가 큰 vertex를 택하는 방식을 채택했다.

2) ConstructBigraph

query vertex u와 그 pivot u'에 대해 $C(u)$와 $C(u')$사이의 bigraph를 나타내는 data structure BI를 construct하는 함수이다. 전체적인 pseudo code는 다음과 같다.

```
   Input: a candidate set C and a pivot dictionary P
   Output: the bigraph index BI
1  begin
2      foreach (u, u') ∈ P do
3          Set BI_u^{u'}(v) to empty for all v ∈ u'.C;
4          Set v.count to 1 for all v ∈ u.C;
5          foreach v ∈ u'.C do
6              foreach v' ∈ N(v) do
7                  if v'.count = 1 then
8                      BI_u^{u'}(v) ← BI_u^{u'}(v) ∪ {v'}
9          Reset v.count to 0 for all v ∈ u.C;
10     return BI;
```

Figure2. Pseudo code for ConstructBigraph function. [4]

전체적인 flow는 위 pseudo code와 같지만 우리의 구현에서 구체적인 BI의 구조는 다음과 같다.

BI : pair의 벡터. 길이는 $|V(q)|$.

$BI[u]$: $C(P[u])$의 정보와 $C(u)$, $C(P[u])$ 사이의 bigraph 정보.

$BI[u].first$: 길이 $|V(G)|$의 벡터. $C(P[u])$의 정보가 담겨있다.

$BI[u].second$: vertex vector의 vector. $C(P[u])$의 각 원소에 대해 해당 원소와 이웃하는 $C(u)$의 원소에 대한 정보가 담겨있다.

3) Enumerate

backtracking을 구현한 함수이다. 전체적인 pseudo code는 다음과 같다.

```
   Input: a query graph q and a data graph G
   Output: all matches from q to G
1  begin
       /* The indexing phase.                                   */
2      C ← ExtractCandidates(q, G);
3      (π, P) ← GenerateMatchingOrder(q, G, C);
4      BI ← ConstructIndex(C, P);
       /* The enumeration phase.                                */
5      l ← 1, u ← π[l], M ← {};
6      foreach v ∈ u.C do
7          M[u] ← v, v.visited ← true;
8          Enumerate(G, π, P, M, BI, l + 1);
9          v.visited ← false, remove (u, v) from M;
10 Procedure Enumerate(G, π, P, M, BI, l)
11     if l = |π| + 1 then Output M, return;
12     u ← π[l], u' ← P[u];
13     foreach v ∈ BI_u^{u'}(M[u']) do
14         if v.visited is false and Validate(G, M, u, v, u') is
             true then
15             Same as Lines 7-9;
16 Function Validate(G, M, u, v, p)
17     foreach u' ∈ BN_q^π(u) with u' ≠ p do
18         if e(M[u'], v) ∉ E(G) then return false;
19     return true;
```

Figure3. Pseudo code for Enumerate, Validate function. [5]

Enumerate에서는 $\pi[1]$을 시작으로 π에 따라 embedding을 구성한다. 이때 $u=\pi[i]$라 두면, BI를 이용하여 $C(u)$ 중 적절하지 않은 원소들은 먼저 택하지

않는다.

Figure3의 Line 14는 embedding 조건 (1), (3)을 check한다. 특히 Validate 함수는 조건 (3)을 check하는데, 이 함수에 $P[u]$를 인자로 받는 이유는 u와 $P[u]$ 사이에 이미 edge가 있음을 Enumerate 함수에서 확인했으므로 같은 작업을 반복하지 않기 위함이다.

4) Matching order 재구성

실제 구현은 위에서 설명한 알고리즘과 다소 차이점을 갖는다. 그 핵심은 matching order 구성이 실행동안 여러 번 이루어진다는 것에 있다. Matching order의 구성은 first vertex에 따라 크게 달라지며, 프로그램의 실행 시간 역시 이에 크게 영향을 받는 것으로 보였다. 따라서, 실제 구현에서는 해당 matching order가 적절하지 않다고 판단될 경우 first vertex를 바꾸어 matching order를 재구성하는 방식을 사용하였다. 자세한 구현은 다음과 같다.

먼저 Enumerate 함수에 들어갈 때마다 error라는 전역변수를 1씩 증가시키고 이 값이 미리 설정된 상수인 ERROR_LIMIT을 넘어갈 때까지 embedding을 하나도 구성하지 못했다면 해당 matching order를 버리고 새로운 order를 구성하였다. order는 first vertex에 따라 결정되므로 first vertex를 바꾸어 주는데, 기준은 core value이다. 즉 query vertex의 core value를 저장해놓은 table인 core table을 내림차순으로 정렬하여 error가 ERROR_LIMIT을 넘어가면 다음으로 큰 core value를 가진 query vertex를 first vertex로 택하는 것이다.

3. Execution environment

본 구현에서 실행 환경은 Github으로 제공된 Graph Pattern Matching Challenge의 skeleton을 그대로 사용하였다. 백트래킹의 구현을 위하여 'src/backtrack.cc'를 수정하였고 이외의 파일은 수정하지 않았다. 따라서, 원본과 동일하게 이하의 과정을 통하여 프로그램을 실행할 수 있다.

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

References

[1] Shixuan Sun, Qiong Luo. 2020. “Subgraph Matching with Effective Matching Order and Indexing.” TKDE.
[2] Ibid. p. 3.
[3] Ibid. p. 9.
[4] Ibid. p. 8.
[5] Ibid. p. 5.