Subgraph matching Algorithm

2018-18699 김준경

2018-17514 유민아

**Ⅰ. Matching order 결정**

Matching order로써 Candidate-size order을 채택하여 구현하였다. 우선, 현재 도달해있는 Ui(query의 vertex)의 children들 중 부모가 모두 맵핑이 되어있는 U들을 찾아 set을 만든다. 이 set의 원소들의 candidate set 중 실제로 지금 연결이 가능한 Vm(cs의 vertex들이면서 Uc에 맵핑될 수 있는 후보 vertex들)의 집합인 extendable candidate set을 찾고, extendable candidate set의 원소의 개수가 최소인 애를 가장 먼저 선택한다. 이때 반드시 최소 원소가 1이므로 extendable candidate set의 원소가 1인 U를 만나면 더 이상 최솟값을 찾지 않고 바로 그 U를 선택하여 수행한다.

**Ⅱ. 백트레킹 구현 방법**

1. backtracking 구현 전 필요한 작업들
   1. root 찾기  
      효율적인 구현을 위해 vertex u에 해당되는 candidate size/ vertex u의 degree의 값이 최소가 되는 지점을 찾아, 해당 u를 root로 간주하고 시작한다.
   2. 각 query vertex마다 parent와 child를 구해주기

vector<vector<int>> parent(n, vector<int>(n, 0));

vector<vector<int>> child(n, vector<int>(n, 0));

다음으로 BFS를 순회하면서 query를 DAG로 만드는 과정을 거친다. BFS 구현 코드를 통해 해당 query vertex의 parent와 child를 각각 찾아주었고 이를 2차원 vector로 구현하였다. DAG 그림에서 a vertex -> b vertex 방향이 코드상에서, parent[b]에는 a, child[a]에는 b가 담긴 식으로 구현된 것이다.

1. backtracking 구현
   1. 함수 형태:

void Backtrack::FindMatches(const Graph &data, const Graph &query, const CandidateSet &cs,  int u, unsigned int count)

인자로 data graph와 query graph, candidate set을 받는다. 인자 중 int u는 현재 방문되고 있는 query vertex의 number이다. 즉 현재 u1에 해당되는 vertex를 고르고 있는 중이라면, int u에는 1이 값으로 들어가게 된다. 다음 매개변수로 주어지는 unsigned int count의 경우 지금까지 방문된 query graph 상에서 u의 개수이다. count를 매개변수로 함수로 디자인한 이유는 count가 query의 전체 vertex와 같아지는 순간 하나의 embedding을 찾은 것이기 때문에 즉시 출력하려고 하기 때문이다. 또한, 하나씩 u를 방문하면서 count를 늘려주거나 다시 backtracking해서 돌아오는 재귀의 과정을 효과적으로 구현하기 위해 매개변수로 count를 넣어주었다.

* 1. FindMatches 함수에서 사용되는 변수

  vector<bool> u\_visited; //u가 방문되었는지를 확인

  vector <bool> v\_visited; //vertex v가 방문되었는지를 확인

  vector<int> each; //각각의 u에 해당되는 vertex를 저장

  map<int, set<int>> candidate\_set;

  set<int> children; //이미 방문한 node 들의 children 후보

* + 1. vector<bool> u\_visited  
       u\_visited는 해당 query vertex가 방문되었는지를 저장한다.
    2. vector <bool> v\_visited:  
       v\_visited는 해당 data graph vertex가 방문되었는지를 저장한다.
    3. vector<int> each  
       each vector에서는 query vertex를 index로 가지고 해당 query에 mapping된 vertex number를 저장해준다.
    4. set<int> children  
       children set은 특정 u에 있을 때 다음 방문될 수 있는 u들을 children에 넣어준다. 즉 children set에는 query vertex의 children을 추가해주고, 자신은 제거해줌으로써 children set이 지속적으로 업데이트된다. 따라서 다음 query vertex가 방문되더라도 다음 가능한 query vertex를 결정할 때 children set에서 하나씩 순회할 수 있는 것이다.
    5. map<int, set<int>> candidate\_set   
       candidate\_set은 map 형태를 사용해서 특정 u에 mapping 될 수 있는 vertex들을 set으로 저장하여 mapping한다. 여기서 candidate\_set은 함수가 호출되기 전에 미리 찾아주는 방식으로 구현했다. 즉, 현재 u vertex가 방문되고 있다면 다음 방문될 u와 mapping될 수 있는 vertex들을 candidate\_set map에 넣어주는 방식으로 진행한다.
  1. 그런 다음, 해당 u를 parameter로 가지는 함수가 호출됐을 때 이미 mapping된 vertex들을 possible\_set에 불러온다. 함수를 호출하기 전 이미 map<int, set<int>> candidate\_set 에 매핑을 시켜놓았기 때문에 key에 해당되는 set을 불러오면 그것이 possible\_set이 된다.
  2. 이후, possible\_set의 원소를 하나씩 순회하면서 다음 방문할 수 있는 vertex들을 찾아본다. 이후 해당 u에서 다음 방문할 수 있는 u들을 children vector에서 하나씩 순회해본다. 이 중, parent가 이미 다 mapping이 되어있는 u에 대해 가능한 candidate vertex들을 구해주고 이들 중 candidate set의 size가 가장 작은 u에 대해 candidate\_set map에 vertex의 set과 함께 mapping 시켜준다. (matching order 관련 부분에서 더 자세히 설명)
  3. 다음으로, candidate set의 size가 가장 작은 u에 대해서 재귀로 다시 호출해주고, count 매개변수도 하나 증가시켜준다. 재귀가 끝난 뒤 방문했던 u와 v는 모두 false 처리해준다.
  4. 이런식으로 재귀 함수를 반복 호출해주면서 함수의 매개변수 count가 query의 vertex 개수와 같아지게 되면 possible\_set을 순회하면서 가능한 embedding을 하나씩 출력해준다. 이렇게 출력한 뒤에 return을 해주게 됨으로써 다시 예전 vertex의 다른 후보들로 돌아가서 계속해서 embedding을 찾는 과정을 진행해준다. 루트 노드의 possible\_set에 해당되는 vertex의 방문이 종료되면 embedding찾는 과정 전체가 종료되는 것이다.

**Ⅲ. Running environment**

1. Running environment

Github 상의 코드를 cloning 혹은 pull 받은 뒤, Linux/ ubuntu 환경에서 실행시켜준다.

그래프 함수에도 사소한 변화가 있었기 때문에, backtrack.cc 파일 이외에도 함께 다 pull 받아주어야 한다.

1. How to run codes

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