

## Subgraph Matching Problem Report

2014-11217 공은채  
2014-11476 차우진

## A. 프로그램 실행 환경 및 실행 방법

## a. 실행환경

CPU: Intel(R) Core(TM) i9-7900X CPU @ 3.30GHz  
RAM: 64GB  
OS: Ubuntu 20.04  
Compiler: g++ 7.5.0

## b. 컴파일 및 실행방법

## Compile

```
mkdir build
cd build
cmake ..
make
```

## Execute

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

## Program output

Program outputs file at **build/result.txt**

\* 프로그램을 한 번 실행 할 때마다 **result.txt**가 생기므로 실행 전 매번 이전에 존재하던 **result.txt**를 직접 없애주어야 함.

## B. 프로그램 구현 설명

## a. 프로그램 흐름 설명

`main`에서 `data_file`, `query_file`, `candidate_set_file`에 접근해 이들을 각각 `Graph data`, `Graph query`, `CandidateSet candidate_set`에 저장함. 이후 해당 인스턴스들을 `input` 값으로 가지고 `BackTrack`의 `PrintAllMatches` 함수를 돌리면 `result.txt` 파일이 생성되고 `backtracking` 과정에서 `embedding`을 찾을 때마다 이가 기록됨.

`PrintAllMatches` 함수에서는 아래와 같은 절차를 거쳐 `embedding`을 찾음.

- (1) `query graph Q`의 모든 정점 중 `C_ini(v)/degree(v)` 값이 가장 작은 정점을 찾고 이를 `root`로 설정함.
- (2) 2차원 `vector DAG`에 `undirected query graph Q`를 저장함.
- (3) `root`를 시작점으로 `BFS` 함수를 실행해 `query graph`의 방향성을 설정하기 위한 `BFS order`를 구하고 이를 1차원 `vector`인 `order`에 저장함.
- (4) `root`를 시작점으로 `buildDAG` 함수를 실행하여 `DAG`에서 모든 `DAG[parent][child]`를 삭제함으로써 `DAG`를 `directed`로 변환함.

- (5) DAG를 기반으로 가능한 모든 {parent, child} pair를 찾고 이를 2차원 vector인 **match\_order**에 저장함.
- (6) embedding을 찾아 쓸 파일을 염.
- (7) backtracking 함수를 실행하여 embedding을 찾아 해당 파일에 출력함.

b. 함수별 설명

각 함수에 대한 설명은 코드 주석에 더 상세히 서술되어 있으므로 주석을 참고하기를 바람. **match\_order** 구현에 대한 설명은 **get\_match\_order**, **backtracking** 구현에 대한 설명은 **backtracking** 함수에 대한 설명과 더불어 아래에 설명되어 있음.

```
void BFS(std::vector<int> &order, int root, const Graph &query)
```

query graph의 정점 v 중  $C_{ini}(v)/\text{degree}(v)$ 의 최솟값을 가지는 정점인 **root**를 시작점으로 삼아 너비우선탐색을 진행하는 함수.

**queue**가 다 비기 전까지 현재 정점을 **queue**에서 **pop**하고 현재 정점과 연결된 모든 정점들 중 방문되지 않은 정점들을 **queue**에 추가하는 행위를 반복함. 이 결과 **queue**에서 **pop**된 정점들을 그때 그때 순서대로 **order** vector에 집어넣으면 **BFS order**를 **order**에 저장할 수 있음.

```
void buildDAG(std::vector<std::vector<int> &DAG, std::vector<int> order, int idx)
```

**input**으로 주어진 DAG는  $DAG[\text{parent}][\text{child}]$ ,  $DAG[\text{child}][\text{parent}]$ 가 모두 존재하는 undirected query graph임. **order**에 따라 이 DAG에서  $DAG[\text{child}][\text{parent}]$ 를 삭제함으로써 directed query graph로 만드는 함수.

**order**로부터 순서대로 **parent**를 정점을 선택하고 **parent**와 연결된 모든 정점에 대해  $DAG[\text{child}][\text{parent}]$ 가 존재하는 경우 DAG에서 삭제함. 이 때 **BFS order** 순으로 진행되므로  $DAG[\text{parent}][\text{parent's parent}]$ 는 이미 삭제되어 존재하지 않음. **buildDAG** 함수를 **root**부터 query graph Q의 모든 정점들에 대해 반복수행하면 DAG가 undirected graph에서 directed graph 형태를 띠게 변경할 수 있음.

```
void get_match_order(const Graph &data, const Graph &query, const CandidateSet &cs, std::vector<std::vector<int> &DAG, std::vector<std::vector<int>> &path, int current)
```

DAG의 **root**부터 시작하여 **child**가 없을때까지 쭉 내려가다 없으면 올라오는 깊이우선탐색 방식으로 {parent, child}를 가지는 2차원 vector **path**를 업데이트하는 함수. 해당 함수의 결과, **path**에는 DAG에서 가능한 모든 {parent, child} 연관성이 반영됨. 즉, **path**에는 DAG의 모든 directed edge가 {parent, child} 형태로 저장됨.

**current** 정점은 맨 처음 **root**부터 시작함. query graph에서 **current** 정점과 연결된 모든 **child** 정점들의 갯수를 찾은 뒤, **child** 정점을 **index** 순서대로 불러옴. (즉, 이 **match\_order**에는 {parent, child}가 프로그램에서 처음으로 **query\_file**로부터 query Graph Q를 읽어 온 **index** 순서와 동일하게 저장됨.)

`temp vector`를 만들어 지금 `current` 정점과 `child` 정점을 하나의 `vector` `{current, child}`로 만들고 만약 이 `vector`가 현재까지 따라가고 있는 `path` 상에 존재하지 않으면 `path`에 추가함. 추가 후에는 재귀적으로 `get_match_order` 불러 `path`를 지속적으로 `update`함. (`current` 정점 자리에 `child` 정점을 넣고 추가된 `path`를 집어넣는 방식)

#### \* Matching order 설명

즉, 상단의 `get_match_order` 함수를 보면 알수있듯이 해당 프로그램에서는 `query DAG`를 만든 뒤, `query DAG`의 모든 `edge`를 `match_order` `vector`에 넣어둠. 이 `match_order`를 `index` 순서대로 가져와 `{parent, child}`가 `data graph`의 `candidate set`과 `mapping` 되는지 확인 함. 이 때, `data graph`의 `candidate set` 중 어떤 순서로 `matching` 하느냐는 단순히 `candidate set`에 담긴 `index` 순서대로 `matching` 해보는 방식을 따름. 수업시간에 배운 DAF에서는 `candidate_size order`를 활용했으나 우리 팀에서는 해당 방식을 취하지 않고 단순히 `candidate set`에 담겨있는 순서대로 앞에서부터 맨 끝까지 매칭해보는 방식을 취함.

#### \* backtracking 설명

`backtracking`은 아래 함수와 같이 세가지 경우로 나누어 구현하였음.

```
void backtrack(const Graph &data, const Graph query, const CandidateSet &cs, std::vector<std::vector<int>> match_order, std::vector<int> &M, int idx)
```

#### input 설명

- `match_order` : `query DAG` 내에서 `edge`가 존재하는 모든 `{parent, child}`의 정보를 가진 2차원 `vector`임. `query graph`의 `embedding`을 확인하기 위해서는 `match_order`내에 표현된 모든 `edge`가 `query graph`에 매칭될 각각의 `data graph vertex` 사이에 존재해야함.
- `idx` : `match_order` 중 `backtracking` 함수를 통해 `partial embedding` `M`에서 존재 여부를 확인 한 `{parent, child}` 페어의 수.
- `M` : `query graph`의 정점 순서대로 `mapping` 되는 `data graph`의 정점을 저장하는 1차원 `vector`. `partial embedding`이 저장되는 `vector`라 보면 됨.

`match_order`에 기반하여 `data graph`에서 `partial embedding` `M`을 찾는 함수. `idx`가 `match_order` 크기와 동일할 때 `partial embedding` `M`을 `complete embedding`으로 판단하여 `result.txt` 상에 출력함.

`backtracking` 함수는 크게 3가지 경우로 나누어 수행됨.

##### (1) Complete embedding 찾은 경우

(`idx`가 `match_order.size`와 동일한 경우)

`query DAG`의 모든 `{parent, child}`간의 `edge`가 존재하는 `data graph`의 정점들을 매핑하여 `M`에 저장해 두었으므로 `M`을 그대로 출력하면 됨.

##### (2) `backtrack`을 처음 시작하는 경우

(partial embedding M의 크기가 0인 경우)  
 M의 크기가 0인 경우는 최초로 root에서 시작하는 경우임. root가 parent인 `match_order[idx][0]`를 root로 설정함. (\* 이 때 `idx = 0`임.)  
 root 정점과 매칭될 수 있는 `data graph`의 정점 후보군 중 하나를 가져오고, 이를 `new_M`을 만들어 `new_M[root]`에 넣어주고 이후 `new_M`을 partial embedding, `idx`로 `idx(*idx = 0임.)`로 넣어 다시 `backtrack`을 불러줌. 해당 행위를 root와 매칭될 수 있는 모든 candidate set 정점들에 대해 진행함.

### (3) `backtrack`에서 partial embedding을 찾아가는 경우(1, 2 외의 경우)

이는 partial embedding M을 가지고 M을 넓혀가는 모든 경우가 속함. root의 경우는 이미 (2)에서 수행되므로 (3)에서 수행될 때 `M[parent]`는 당연히 특정 정점이 매핑되어 있을 수 밖에 없음. 반면 `M[child]`는 이미 매핑되어 있을 수도 있고 그렇지 않을 수도 있음. 따라서 두 경우를 나누어 수행해 줌.

#### (3)-1. `M[child]`에 이미 특정 정점이 매핑되어 있는 경우.

이는 child의 다른 parent가 존재하고 해당 parent와 backtracking을 수행하는 과정에서 `M[child]`에 특정 정점이 매핑되어 M에 저장되어 있는 경우임. 따라서 `M[child]`에 기존에 매핑되어 있는 정점과 현재 `match_order`에서 가져온 `parent_candidate` 사이에 `data graph`에서 `edge`가 존재하는 경우만 해당 M을 유지하고 현재 `match_order[idx]`를 체크하였으므로 `idx+1`을 넣어 backtracking을 마저 수행함. 반면 `edge`가 존재하지 않으면, 지금까지 찾은 M은 embedding이 아니므로 함수를 종료함.

#### (3)-2. `M[child]`에 아무 정점도 매핑되어 있지 않은 경우.

이 경우에는 현재 `query graph`의 child 정점에 매칭될 수 있는 모든 경우를 고려해야 함. 따라서 child의 candidate set 중 맨 앞의 정점을 하나 가져와 `child_candidate`으로 저장함. `M[parent]`에 매핑된 정점과 `child_candidate` 사이에 `edge`가 존재하는 경우 새로운 partial embedding을 저장할 `new_M`을 만들고 여기에 기존 M의 정보를 저장하고 `M[child]`에 `child candidate`를 저장해 partial embedding을 늘림. 이후 `new_M, idx+1`을 넣어 backtracking을 지속함.