[2021-1 알고리즘 – 그래프 패턴 매칭 챌린지 보고서]

팀명	레포지토리	
supernova	https://github.com/supernova-snu/Graph-Pattern-Matching-Challenge	
학과	**학번**	**성명**
기계공학부/컴퓨터공학	2018-16169	신동민
수리과학부/컴퓨터공학	2018-14637	임은성

1. 개요

본 과제의 목적은 빅데이터 그래프 분석의 핵심인 그래프 패턴 매칭 문제를 중심으로, 가장 기본이 되는 subgraph matching (또는 subgraph isomorphism) 문제를 빠르게 해결하는 알고리즘을 구현하는 데에 있다. 이를 위해 candidate set을 적절히 활용하고, matching order를 설계하였으며, backtracking 기법을 적용하였다. 그 결과 준 과제를 정확히 해결해내고, 속도도 빠른 (10만개/20초) 알고리즘을 구현하였다.

2. 프로그램 실행법

```
dongmin@Dongminui-MacBookPro graph-pattern-matching-java % java -version
openjdk version "11.0.10" 2021-01-19
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.10+9)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.10+9, mixed mode)
```

Java 11 SDK 환경에서 코드를 작성하였고, out 디렉토리에 컴파일된 class 파일이 존재한다. 컴파일된 class 파일을 실행해 프로그램을 실행할 수 있다. 루트 폴더에서 out/production/graph-pattern-matching-java 디렉토리로 이동해, 다음과 같이 클래스 파일을 java 커맨드로 실행한다.

```
”cd out/production/graph-pattern-matching-java”
”java com.supernova.Main <data graph file> <query graph file> <candidate set file>”
```

위 커맨드로 프로그램을 실행한다.

```
dongmin@Dongminui-MacBookPro graph-pattern-matching-java % ls
README.md                   candidate_set               graph-pattern-matching-java.iml query
Report_supernova.docx       data                        out                             src
dongmin@Dongminui-MacBookPro graph-pattern-matching-java % cd out/production/graph-pattern-matching-java
dongmin@Dongminui-MacBookPro graph-pattern-matching-java % java com.supernova.Main data/lcc_yeast.igraph query/lcc_yeast_n1.igraph candidate_set/lcc_yeast_n1.cs
```

그림 1. 프로그램 실행 위한 커맨드 예시

실행하면 다음과 같이 최대 100,000 줄의 output 이 출력된다.

```
dongmin@Dongminui-MacBookPro graph-pattern-matching-java % java com.supernova.Main data/lcc_yeast.igraph query/lcc_yeast_n1.igraph candidate_set/lcc_yeast_n1.cs
t 50
a 28 495 921 982 102 86 1270 122 1143 1695 1699 1702 1081 97 776 267 681 1066 85 364 186 267 501 810 754 1525 154 676 269 533 371 157 1662 1067 645 61 249 32 371 2 455 1228 2292 1
62 731 186 919 50 841 691
a 28 495 921 982 102 86 1270 122 1143 1695 1699 1702 1081 97 776 267 681 1066 85 364 186 267 501 810 754 1525 154 676 269 533 371 157 1662 1067 645 61 249 32 371 2 455 1228 2292 1
62 731 186 919 50 841 844
a 28 495 921 982 102 86 1270 122 1143 1695 1699 1702 1081 97 776 267 681 1066 85 364 186 267 501 810 754 1525 154 676 269 533 371 157 1662 1067 645 61 249 32 371 2 455 1228 2292 1
62 731 186 919 50 843 691
a 28 495 921 982 102 86 1270 122 1143 1695 1699 1702 1081 97 776 267 681 1066 85 364 186 267 501 810 754 1525 154 676 269 533 371 157 1662 1067 645 61 249 32 371 2 455 1228 2292 1
62 731 186 919 50 843 844
a 28 495 921 982 102 86 1270 122 1143 1695 1699 1702 1081 97 776 267 681 1066 85 364 186 267 501 810 754 1525 154 676 269 533 371 157 1662 1067 645 61 249 32 371 2 455 1228 2292 1
62 731 186 919 50 1244 691
```

그림 2. 출력 예시

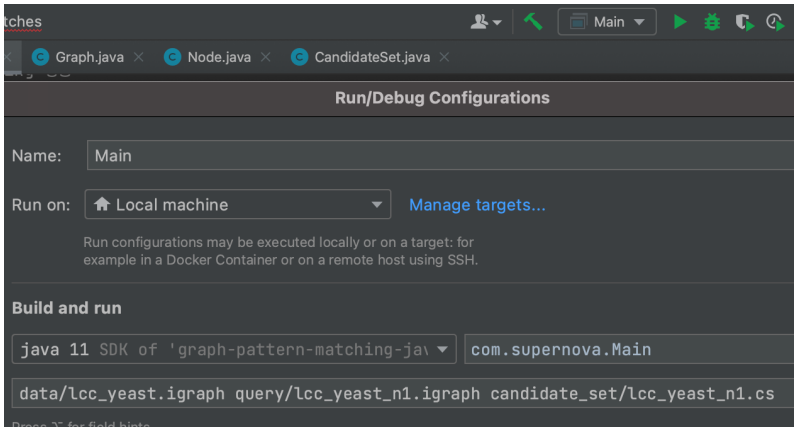

그림 3. IntelliJ 상에서 실행할 경우의 run configuration

3. 프로그램 설명

기본적으로, data graph와 query graph 파일을 읽어들여 Node와 Graph 객체로 변환한다. 또한 candidate set 파일을 읽어 CandidateSet 객체를 생성한다. 이후 BackTrack 객체에서 백트래킹을 담당해, 임베딩을 찾을때마다 최대 10만개 임베딩 그래프의 노드 아이디를 순서대로 출력한다.

세부사항은 아래서 backtracking 구현법 위주로 다룬다.

1) Backtracking 구현

Query graph에서 노드 id의 오름차순으로 matching order를 정해 0부터 작업을 실행한다. 첫 번째로, query id가 0이 맵핑되는 data vertex를 우리가 출력할 currentPath에 저장한다. 이 후 query id가 1이 맵핑되는 data vertex를 다음 path에 올 수 있는 후보지로 선정하고, 각 vertex에 대해 query graph를 참고하여 path에 포함시킬 수 있는지를 판단한다.

이를 판단하는 기준은 다음과 같다. currentPath가 새로운 data vertex를 받을 때는 이 vertex를 맵핑했던 query에 대하여 이전 query id들과의 연결관계를 확인한다. 추가하려는 vertex에 해당하는 query vertex에 연결되는 것이 있는지 이전 query id중에서 확인하고, 있다면 해당하는 query id를 mapping하고 있는 data vertex가 currentPath안에 있는지 확인한다. 이렇게 모든 query edge에 대해서 존재한다면, 이 새로운 vertex는 현재의 currentPath에 포함시켜도 되는 상황이기 때문에 포함시키고, 어떠한 query edge에 대해 해당하는 query id를 가지고 있는 vertex가 currentPath 안에 없을 때는 invalid한 경우이기 때문에 포함시키지 않는다. .

이렇게 재귀적으로 currentPath가 모든 query id를 포함하고 있을 때까지 재귀적으로 currentPath에 query id에 대응하는 data vertex id를 넣어준다. 최종적으로 currentPath가 모든 query id에 해당하는 vertex를 맵핑했을 때 그 맵핑된 data vertex의 조합을 결과로 출력한다.

아래에서부터 자세한 코드설명을 하겠습니다.

```
public class BackTrack {

    Graph data;
    Graph query;
    CandidateSet cs;

    BackTrack(Graph data, Graph query, CandidateSet cs){
        this.data = data;
        this.query = query;
        this.cs = cs;
    }
```

그림 4. BackTrack class specification

위와 같이 각 input data를 구조화한 data graph, query graph, candidateSet을 클래스 변수로 정의한다. 그리고 생성자에서 주어진 값을 모두 받아 클래스를 정의한다.

```
public void printAllMatches() {

        int count = 0; // for exit program after count >= 100,000

        System.out.println("t " + query.getNumNodes());

        int flag;
        ArrayList<Integer> currentPath = new ArrayList<>(query.getNumNodes());
        // □□□ □□□ □□ □□ □
        ArrayList<Integer> tempIndex = new ArrayList<Integer>(Collections.nCopies(query.getNumNodes(),-1));
        Integer nextVertexId;
        // BackTracking
        while(currentPath.size()<query.getNumNodes()){
            flag = 0;
            ArrayList<Integer> possibleNextVertices = findPossibleVerticeIds(currentPath);
            // □□□ x_i□ □□ □□□
            //System.out.println("currentPath: "+currentPath+", tempIndex"+tempIndex);

            for(Integer i=tempIndex.get(currentPath.size())+1;i<possibleNextVertices.size();i++){

                nextVertexId = possibleNextVertices.get(i);
                // □□ □□□ □
                if(isPossiblePath(currentPath,nextVertexId)){

                    flag = 1;
                    tempIndex.set(currentPath.size(),i);
                    currentPath.add(nextVertexId);

                    // matching □□
                    if(currentPath.size() == query.getNumNodes()){
                        System.out.print("a");
                        for(Integer matchingVertices : currentPath){
                            System.out.print(" "+matchingVertices);
                        }
                        System.out.println();
                        flag = 0;
                        count++;
                        if(count>=100000){ // 100000□ □□ □□□ □□□□
                            System.exit(0);
                        }
                    }
                    break;
                }
            }
            // □□□ □□□ nextVertice□ □□ □ □□□ □□□ □□□ □ □□
            if(flag==0){
                if(currentPath.size()==0) return;
                if(currentPath.size()<query.getNumNodes()) {
                    tempIndex.set(currentPath.size(), -1);
                }
                currentPath.remove(currentPath.size()-1);
            }
        }

    }
```

그림 5. BackTracking 구현의 핵심파트

currentPath에는 tracking을 하면서 valid한 set를 담고있는 경로이고, data vertex id의 배열이다. tempIndex는 currentPath에 넣을 때 candidate set에서 query id에 해당하는 data vertex id의 배열 중의 index를 넣는 자리이다. BackTracking을 할 때 어느 위치에서 재시작할지 지시하는 지표다. BackTracking은 DFS와 같은 방식으로 시행된다. currentPath 다음에 들어올 vertex는 currentPath 다음 query id에 해당하는 data vertex를 candidate set에서 고른다. 이 때 들어올 수 있는 vertex를 findPossibleVerticeIds 함수를 통해 구하고, 각 vertex들에 대해 isPossiblePath함수를 통해서 currentPath에 추가시켜도 되는지에 대한 여부를 확인한다. 그리고 추가시킬 수 있는 vertex가 없

으면 flag를 조회함으로써 currentPath에서 없애준다. DFS와 같이 될 수 있는 path를 뒤에서부터 바꾸어가며 찾는다. 아래는 위에서 사용한 findPossibleVerticeIds 함수와 isPossiblePath 함수에 대한 설명이다.

```
private ArrayList<Integer> findPossibleVerticeIds(ArrayList<Integer> currentVerticeIds){
        int nextQueryId = currentVerticeIds.size();
        return this.cs.mappingSet.get(nextQueryId);
    }

    // f(x_1,x_2,...,x_i)
    private boolean isPossiblePath(ArrayList<Integer> currentPathIds,Integer newVerticeId ){

        int newQueryId = currentPathIds.size();

        ArrayList<Node> shouldConnentQueries = this.query.adjacencyMap.get(newQueryId);

        // newVerticeId □ □□□□□□ targetQuery
        for(Node targetQuery: shouldConnentQueries){
            // □□ sorting□□□□□□ □□□□□□ □□□□ id□□ □□ □□□□□
            if(targetQuery.getId()>newQueryId){
                break;
            }

            boolean valid =false;
            // targetQuery□ □□□□ data vertex id□ □□ currentPath□ □□□□ ok, □□□□ return false
            for(Integer dataVertexId : this.cs.mappingSet.get(targetQuery.getId())){
                // newVertice□ □□□□ node□□ id □□□□ □□
                ArrayList<Integer> neighborIds = new ArrayList<Integer>(data.adjacencyMap.get(newVerticeId).size());
                for(Node node : data.adjacencyMap.get(newVerticeId)){
                    neighborIds.add(node.getId());
                }
                // newVertice□ □□□□ dataVertextId□ □□ □ valid
                if(neighborIds.contains(dataVertexId)){
                    valid =true;
                    break;
                }
            }
            // targetQuery□ □ □□□□ □□□□ □□□□ □□□□ return false
            if(!valid){
                return false;
            }
        }

        return true;
    }
```

그림 6. findPossibleVerticeIds와 isPossiblePath 함수의 구현 코드

findPossibleVerticeIds 함수는 currentPath의 vertex를 인자로 받아서 다음 query id에 매핑이 되는 vertex set을 배열로 반환한다.

isPossiblePath는 새로 들어올 newVerticeId에 대하여 기존에 있던 currentPath의 vertex와 비교하면서 query graph에 해당하는 edge가 mapping된 형태로 있는지에 대한 여부를 확인한다. 만약 모두 있으면 true를 반환하고, 하나라도 연결되는 edge가 없으면 false를 반환함으로써 새로 들어올 vertex가 valid한지 invalid한지 여부를 결정할 수 있다.

[끝]

감사합니다.