그래프 패턴 매칭 경진대회

그래프 패턴 매칭은 빅데이터 그래프 분석에 있어서 핵심이 되는 문제이다. 많은 빅데이터가 그래프로 모델링되므로 이를 분석하기 위해 대규모 그래프에 대하여 빠른 시간에 그래프 패턴 매칭 문제를 푸는 알고리즘의 개발이 중요해지고 있다.

본 경진대회에서는 그래프 패턴 매칭에서 가장 기본이 되는 subgraph matching (또는 subgraph isomorphism) 문제를 빠르게 푸는 소프트웨어를 개발하고자 한다. (자세한 내용은 아래의 설명 참고).

- 참가 자격: 본교 대학(원)생
- 시상 내역

구분	팀수	상금
금상	1	200 만원
은상	1	100 만원
동상	2	각 50만원

- 대회 일정

구분	기간	상세내용
참가 신청	5월 12일 – 5월 25일	대회 GitHub 저장소를 fork한 뒤, 팀원 목록(2인)과 연락처, GitHub 저장소 링크를 이메일로 제출
과제 진행	5월 12일 – 6월 8일	GitHub 저장소에서 과제 진행
시상	7월 중	순위 발표 및 수상자 시상

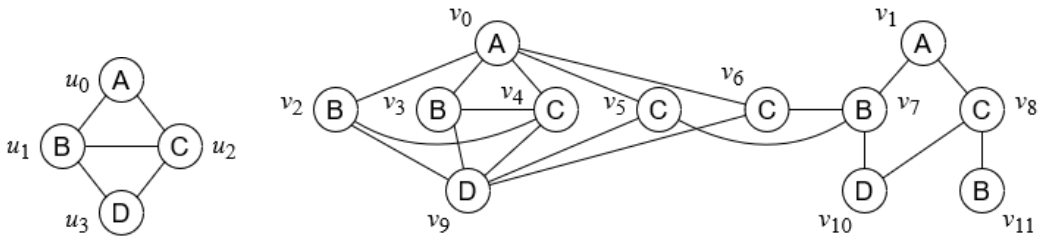
- 문의처
  - 서울대학교 컴퓨터공학부 컴퓨터이론 및 응용 연구실
  - Tel. 82-2-880-1828
  - E-mail. `yhnam@theory.snu.ac.kr`
  - GitHub. `https://github.com/SNUCSE-CTA/Graph-Pattern-Matching-Challenge`

* 본 경진대회는 과학기술정보통신부 재원으로 정보통신기획평가원의 지원을 받아 SW컴퓨팅 산업원천기술개발사업 SW스타랩 과제로 개최됨.

Graph Pattern Matching Challenge

Given a data graph G and a query graph q, *subgraph matching* (also known as *subgraph isomorphism*) is the problem of finding all distinct embeddings (i.e., matches) of q in G. Most practical solutions for subgraph matching are based on the backtracking approach, which recursively extends a partial embedding by mapping query vertices to data vertices one by one. The performances of a subgraph matching algorithm that uses the backtracking framework may differ depending on the choice of a matching order. For example, the state-of-the-art subgraph matching algorithm DAF uses the backtracking framework based on *DAG ordering*, in which the matching order follows a topological order of a directed acyclic graph q_D of q.

In this challenge, one team consists of two people, and you are to design your matching order and solve subgraph matching using backtracking. You are to write a program that gets as input (1) a data graph $G = (V_G, E_G, L_G)$, (2) a query graph $q = (V_q, E_q, L_q)$, and (3) a candidate set $C(u)$ for each vertex $u \in V_q$ ($C(u)$ is a set of vertices in G which u may be mapped to) and outputs at most 10^5 distinct embeddings of q in G. For example, for query graph q and data graph G in Figure 1, there are two embeddings of q in G, i.e., $\{(u_0, v_0), (u_1, v_2), (u_2, v_4), (u_3, v_9)\}$ and $\{(u_0, v_0), (u_1, v_3), (u_2, v_4), (u_3, v_9)\}$. You are not supposed to use any pruning techniques such as failing sets in DAF. For simplicity, we focus on undirected, connected, and vertex-labeled graphs in this challenge. To assist you in carrying out the challenge, (1) a set of data graphs, (2) a set of query graphs, and (3) an executable program that outputs a candidate set for each query vertex for a given data graph and a query graph will be provided in a GitHub repository (`https://github.com/SNUCSE-CTA/Graph-Pattern-Matching-Challenge`). You must fork the repository and do the challenge there. Your scores will be based on the last commit before the deadline.



(a) Query graph q (b) Data graph G

Figure 1: Query graph and data graph of subgraph matching

Since subgraph matching is an NP-hard problem, finding 10^5 matches may take too much time. In this challenge, therefore, the time limit for a query graph is set to 1 minute, regardless of the language used. The performance of your program will be measured by

$$\sum_{G \in \mathbb{G}} \sum_{q \in \mathbb{Q}_G} \max\left(\frac{X(q,G) - Y(q,G)}{X^*(q,G)}, 0\right),$$

where $\mathbb{G}$ is a set of data graphs, $\mathbb{Q}_G$ is a set of query graphs for data graph G, $X(q,G)$ is the number of embeddings of q in G in your output, $Y(q,G)$ is the number of mappings in your

output that are not embeddings of q in G, and $X^*(q, G)$ is the maximum number of embeddings of q in G found by contestants. Programs that are tied will be ranked by their running times, regardless of the language used. Note that whenever an embedding is found during backtracking, it should be printed out immediately. Otherwise, your program may not be able to print the embeddings, since programs that exceed the time limit will be killed.

If you have any questions about the challenge, you may leave your questions on the issue board of the repository (`https://github.com/SNUCSE-CTA/Graph-Pattern-Matching-Challenge/issues`).

- Your program must take a data graph file path as the first command line argument, a query graph file path as the second command line argument, a candidate set file path as the third command line argument, and write the embeddings to stdout, following the file formats described below.
- Explain in your report how your program chooses a matching order and performs backtracking.
- Write down the environment you run your program and how to run your program in your report.
- Write comments appropriately in your program.
- Upload your report and source code to your GitHub repository.
- Email your team member list and GitHub repository address (`yhnam@theory.snu.ac.kr`).

Graph file format. A line in a file must be one of the followings.

1. **Tag.** There is one tag line per file, which must appear before any vertex or edge descriptors. The tag line has the following format.
   ```
   t ID VERTICES
   ```
   The lower case character `t` signifies that this is a tag line. `ID` field contains an integer value specifying the id of the graph. `VERTICES` field contains an integer value specifying the number of vertices in the graph.

2. **Vertex descriptors.** There is one vertex descriptor line for each vertex of the graph. The vertex descriptor line has the following format.
   ```
   v ID LABEL
   ```
   The lower case character `v` signifies that this is a vertex descriptor line. `ID` field contains an integer value specifying the id of the vertex. `LABEL` field contains an integer value specifying the label of the vertex.

3. **Edge descriptors.** There is one edge descriptor line for each edge of the graph. The edge descriptor line has the following format.
   ```
   e ID1 ID2 LABEL
   ```
   The lower case character `e` signifies that this is an edge descriptor line. `ID1` field contains an integer value specifying the id of one vertex. `ID2` field contains an integer value specifying the id of another vertex. `LABEL` field contains an integer value specifying the label of the edge. Note that, in this challenge, `LABEL` field contains 0 for any edge descriptor line, since we focus on vertex-labeled graphs.

Here is a sample instance for the query graph in Figure 1a.

```
t 0 4
v 0 0
v 1 1
v 2 2
v 3 3
e 0 1 0
e 0 2 0
e 1 2 0
e 1 3 0
e 2 3 0
```

Candidate set file format. A line in a file must be one of the followings.

1. **Tag.** There is one tag line per file, which must appear before any CS descriptors. The tag line has the following format.
   `t VERTICES`
   The lower case character `t` signifies that this is a tag line. `VERTICES` field contains an integer value specifying the number of vertices in the query graph.

2. **CS descriptors.** There is one CS descriptor line for each vertex of the query graph. The CS descriptor line has the following format.
   `c ID SIZE ID1 ID2 ...`
   The lower case character `c` signifies that this is a CS descriptor line. `ID` field contains an integer value specifying the id of the query vertex. `SIZE` field contains an integer value specifying the size of the candidate set. `ID1`, `ID2`, ... fields contain integer values specifying the ids of each data vertex in the candidate set.

Here is a sample instance for the query graph and the data graph in Figure 1.

```
t 4
c 0 1 0
c 1 2 2 3
c 2 3 4 5 6
c 3 1 9
```

Output file format. A line in a file must be one of the followings.

1. **Tag.** There is one tag line per file, which must appear before any Embedding descriptors. The tag line has the following format.
   `t VERTICES`
   The lower case character `t` signifies that this is a tag line. `VERTICES` field contains an integer value specifying the number of vertices in the query graph.

2. **Embedding descriptors.** There is one embedding descriptor line for each embedding of the query graph in the data graph. The embedding descriptor line has the following format.
   `a ID1 ID2 ...`
   The lower case character `a` signifies that this is a embedding descriptor line. `ID1` field contains an integer value specifying the id of the vertex in the data graph that is the mapping of a query vertex 0 in this embedding, `ID2` field contains an integer value specifying the id of the vertex in the data graph that is the mapping of a query vertex 1 in this embedding, and so on.

Here is a sample instance for the query graph and the data graph in Figure 1.

```
t 4
a 0 2 4 9
a 0 3 4 9
```