Graph Pattern Matching Challenge Report

기계공학부 2016-16967 설진환

기계항공공학부 2017-19763 김재형

1. 구현 환경

작성 언어: C++ 11

구현 환경: Ubuntu Linux 20.04 LTS / Windows 10

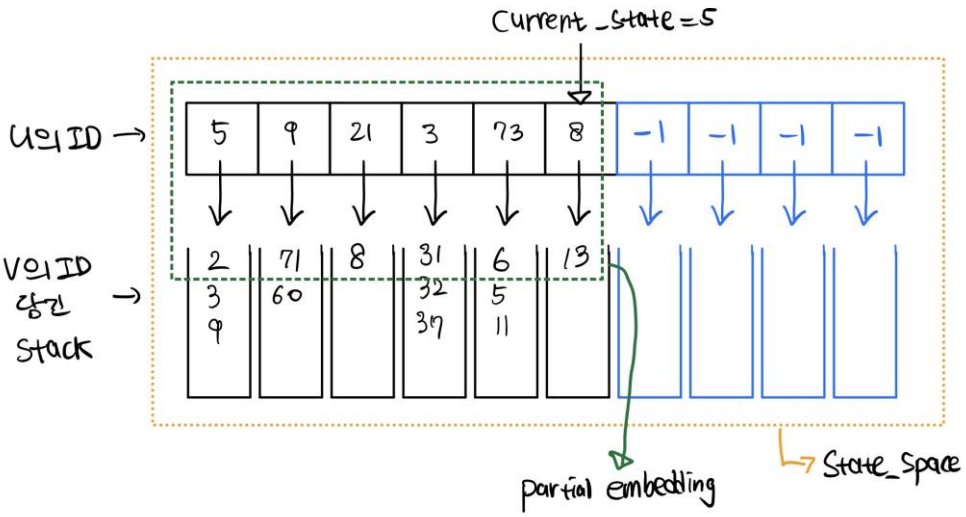
실행 방법: 산출된 결과가 Embedding이 맞는지 확인하기 위한 sanity check 프로그램을 build 하는 코드를 CMakeLists.txt에 추가하였으나 기본적인 프로그램 빌드하는 방법은 Challenge README.md에 나와있는 방법과 동일하다.

2. State space 자료구조

아래 그림은 Backtracking을 실행할 state space를 나타내는 그림이다. u는 query graph의 vertex를 의미하고, v는 data graph의 vertex를 의미한다. 구현한 State space는 u와 해당 u의 candidate set 에 있는 v들 중 embedding condition을 만족하는 것들의 stack(이하 v-stack)을 pair로 갖는 std::vector로 구현하였다.

이렇게 자료구조를 설계하면 각 u마다 v-stack의 top에 있는 점들을 모았을 때 그것이 partial embedding이 되는 것으로 생각하고 backtracking을 진행할 수 있다.

또한 다음 u가 결정되지 않은 위치에는 tombstone(아래그림에서는 -1)을 두어서 구분하였다.



3. 주요함수들의 high level implementation

```
size_t SelectRoot(const Graph &query, const CandidateSet &cs);
```

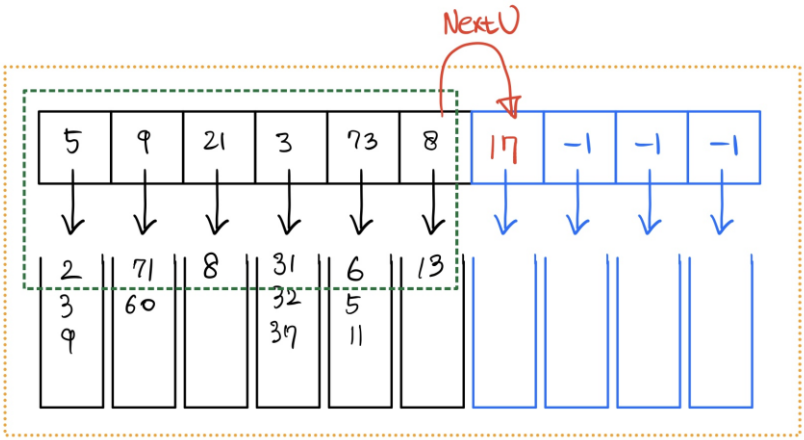
Candidate Set에서의 Size / Query에서의 Degree 가 최소인 u를 root으로 설정하였다.

```
size_t NextU(const Graph &data, const Graph &query, const CandidateSet &cs);
```

기존의 Partial Embedding 이용하여 다음 u의 경로를 결정하여 준다. Extendable vertex 집합을 관리하며 그에 속한 모든 u에 대하여 각각 candidate set에 달린 모든 v들에 대해 embedding condition을 고려하며 그 중 모두 만족하는 개수인 $|C_M(u)|$를 구한다.

DAF에서는 $|C_M(u)|$ 만을 고려해서 u의 순서를 정했다면 본 프로젝트 팀은 u의 label frequency와 degree를 추가로 고려하여 u의 순서를 정했다. 구체적으로 u들에 대하여 각각 $|C_M(u)|$를 비교하여 작은 것을 선택하고, $|C_M(u)|$가 같다면 다음으로 query에서의 u의 label frequency를 비교하여 작은 것을 선택하고, label frequency까지 같다면 query에서의 u의 degree가 최소가 되도록 u를 정한다.

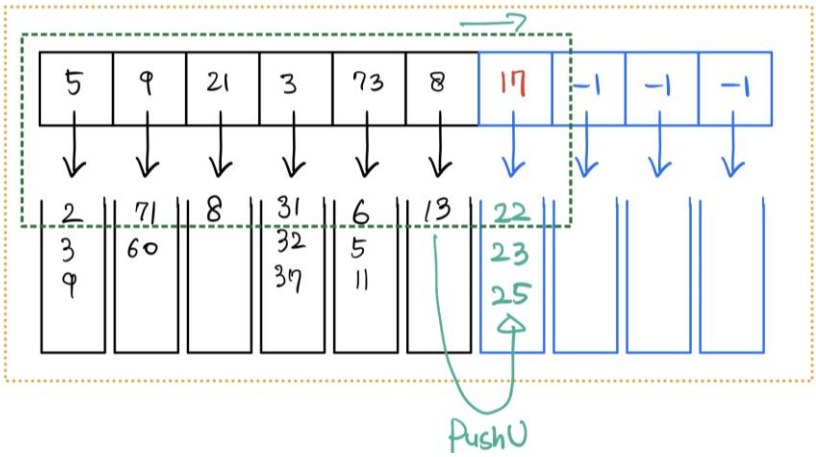
이렇게 경로를 잡는 이유는, extendable한 u 중 $|C_M(u)|$가 같은 경우가 많아 $|C_M(u)|$ 최소값만으로 u를 찾는 것이 의미가 없었고, label frequency와 degree가 높은 u가 초반에 선택된다면 차후 가능한 v가 없어 GoBack 함수 호출 시 돌아와야 하는 거리가 길어져 수행 시간이 많이 늘어나기 때문이다. 실제로 위와 같은 방법으로 $|C_M(u)|$만 적용했을 때는 Embedding을 찾지 못하거나 오래 걸렸던 case들에서도 결과를 산출해내는 것을 확인하였다.



```
bool PushU(size_t u, size_t current_state, const CandidateSet &cs, const Graph
```

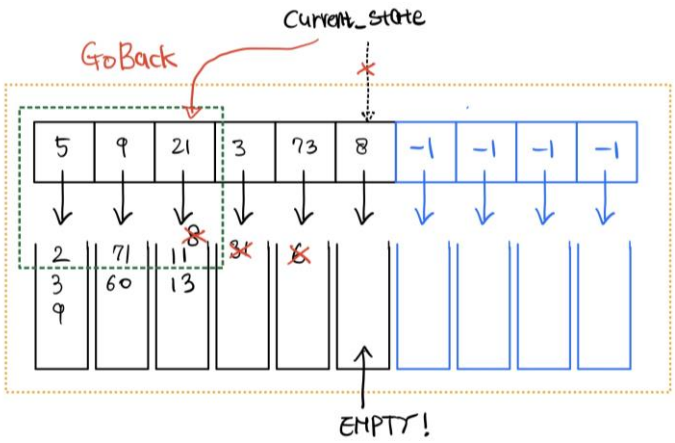
NextU에서 결정된 다음 u를 받아서 가능한 v들을 data graph에서의 degree가 작은 순서대로 정렬하여 state_space와 current_state 업데이트 해준다. 이 때, 추가적인 temp stack을 두어서 stack 내부에서 degree가 작은 순서대로 정렬하는 알고리즘을 사용하였다. 이렇게 degree 순서로 정렬을 할 경우 불가능한 경우의 vertex를 빠르게 제거할 수 있도록 해준다.

이렇게 빨리 안되는 경우를 제거할 수 있는 v를 고르게 되면 $C_M(u)$ 에 영향을 끼치기 때문에 u의 순서에도 영향을 끼치게 된다. 즉, 본 프로젝트 팀이 $|C_M(u)|$ 를 변형한 알고리즘을 사용한만큼 v를 degree 순서로 정렬하여서 u의 순서도 더 효과적으로 바뀌었다고 할 수 있다.



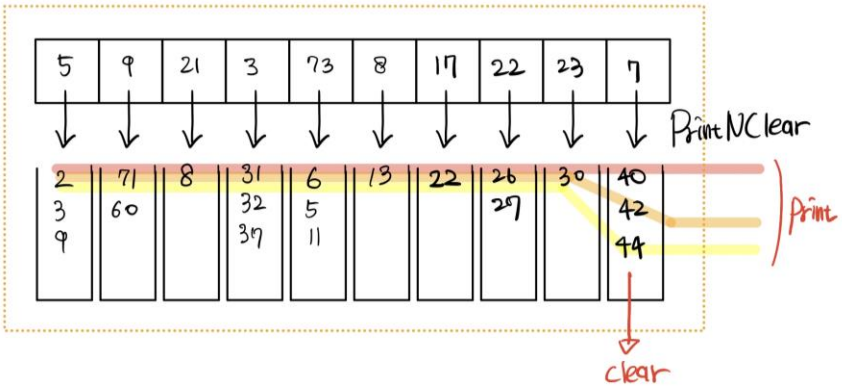

```
size_t GoBack(size_t current_state);
```

PushU 시 가능한 v 없어 앞으로 돌아가는 경우 DFS Search와 같은 원리로 v stack pop 해주며 empty 아닌 경우까지 내려가고 state_space와 current_state 업데이트 해준다.



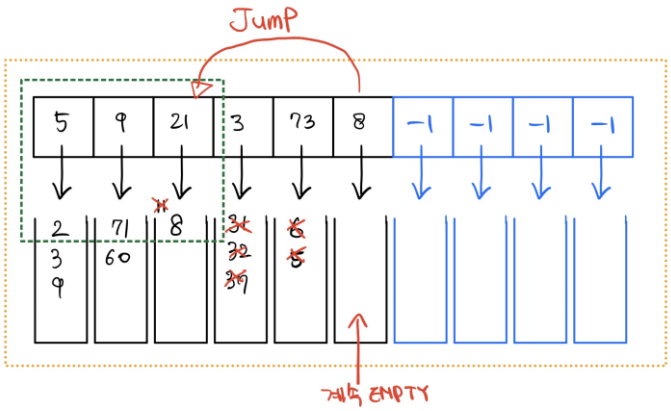
```
void PrintnClear(size_t current_state);
```

마지막 u에 도달할 시 v-stack의 모든 원소 포함하여 출력 후 clear 해준다.




```
size_t Jump(size_t current_state);
```

30초동안 current_state 최대치를 갱신하지 못하는 경우 v의 stack들을 clear해주며 강제로 u를 앞쪽으로 이동시킨다.



4. Result sanity check

출력된 결과가 Embedding이 맞는지 확인하는 sanity check 프로그램을 추가 구현하였다. program의 stdout을 file로 redirect 한 뒤 해당 file을 input으로 받아서 sanity를 체크하는 프로그램이다. Embedding의 정의에 따른 세가지 조건을 아래와 같이 구현하였다

조건 1. Injective – 입력 받은 v끼리 중복 여부를 확인

```
for(size_t i = 0; i<array_size; i++)
{
    for(size_t j = 0; j<i; j++)
        {
            if(array[i]==array[j])
```

조건 2. Label 일치 – i번째 v의 label == u_i의 label 확인

```
for(size_t i = 0; i<array_size; i++)
{
    if(query.GetLabel(i)!=data.GetLabel(array[i]))
    {
        printf("Label not match\n");
        printf("u1 : %zu, v1 : %zu\n", i, array[i]);
        printf("Not Valid\n");
        return 0;
    }
}
```

조건 3. Query의 모든 edge가 data에 mapping 된 채로 연결 확인

```
for(size_t i = 0; i<query.GetNumVertices(); i++){
    for(size_t j = 0; j<i; j++)
    {
        if(query.IsNeighbor(i, j))
        {
            if(!data.IsNeighbor(array[i], array[j]))
            {
                printf("Edge not match\n");
                printf("u1 : %zu, v1 : %zu, u2 : %zu, v2 : %zu\n", i, array[i]
                printf("Not Valid\n");
                return 0;
            }
        }
    }
}
```

을 통하여 얻어낸 embedding의 validity를 확인해 볼 수 있다.