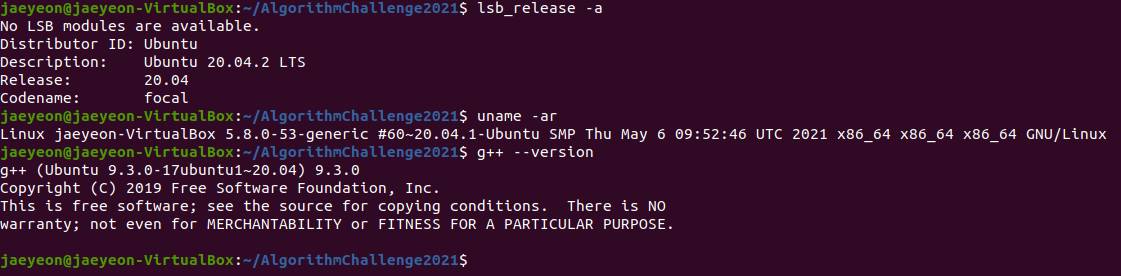
**Graph Pattern Matching Challenge Report**

**2018-10725 김재연, 2018-16371 문보설**

1. **실행환경 및 실행방법**

본 챌린지 진행에 사용된 개발 및 실행환경은 Virtual Box에서 실행된 Ubuntu 20.04LTS이며, 사용된 커널 버전은 linux 5.8.0-53-generic이다. 챌린지 진행을 위한 코드들은 Visual Studio Code에서 C++을 이용해 작성되었고, 사용한 g++ 컴파일러 버전은 9.3.0이다. 구체적인 개발 환경은 다음의 사진과 같다.



필자가 챌린지 진행을 위해 작성한 코드의 실행 방법은 처음에 github을 통해 제시된 코드의 실행 방법과 동일하다. 곧 build 폴더로 들어가 cmake와 make을 이용해 빌드를 해준 후, 다음의 명령어를 입력해 실행한다.

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

1. **구현 방법**

필자가 구현한 코드들이 backtracking을 수행해 제시된 embedding을 찾아내는 과정은 크게 두 부분으로 구성된다. 첫째, 제시된 query graph를 DAG 형태로 만들어준다. 둘째, 이렇게 만들어진 DAG를 바탕으로 DAG-ordering과 candidate size order를 이용해 backtracking을 수행한다. 곧 수업 시간에 배운 내용을 그대로 구현한 것으로, 다만 이를 효율적으로 구현하고자 하였으며, 굳이 다른 matching order를 택하지 않은 것은 위와 같이 구현하였을 때 주어진 test case 전부에서 10만개의 embedding을 1분 안에 찾아냈기 때문이다.

첫 번째 DAG를 만들어주는 부분은 dag.cc 및 graph.cc에 구현되어 있으며, 두 번째 실제 backtracking을 수행하는 부분은 backtracking.cc에 구현되어 있다. 각 부분의 구체적인 구현은 다음과 같다.

1. **DAG**

Dag를 만드는 과정은 Dag.cc와 Dag.h에 정의되어 있는 클래스 Dag의 생성자에 나타나 있다. 클래스 Dag는 graph에 구현되어 있는 생성자를 오버라이드하여 자신의 생성자로 사용한다. 이러한 방식을 택한 이유는 다음과 같다. 첫째로 graph.cc에 정의되어 있는 transferred\_label을 사용하기 위함이다. Data 그래프를 만들 때 그래프의 Label l을 TransferLabel()을 통해 transferred\_label[l]로 변환하여 저장하기 때문에 query graph의 label l도 동일한 변환을 따르게끔 하였다. 두번째로 graph 클래스에 이미 정의 되어 있는 함수들을 활용하기 위함이다. Graph 클래스의 함수 중 GetDegree(Vertex v)나 GetLabel(Vertex v)의 함수들은 Dag에서 그대로 활용할 수 있게 했다. Dag의 child-parent 관계 때문에 내용이 달라지는 IsNeighbor(Vertex v, Vertex u) 함수의 경우 버츄얼로 오버라이드하여 사용했다. 기타 Dag에만 추가적으로 구현한 함수는 GetParentSize(Vertex v), GetChildSize(Vertex v), GetParent(Vertex v, size\_t i), GetChild(Vertex v, size\_t i), GetRoot() 등이 있다. 우선 Dag의 생성자를 소개하면서 Dag를 만든 과정에 대해 설명한 뒤 위에서 언급한 여타 함수들에 대한 설명을 덧붙인다.

**Dag(const std::string& filename, const CandidateSet &candidateSet, bool is\_query)**

: Graph(const std::string& filename, const CandidateSet &candidateSet, bool is\_query)를 호출한다. 기존 graph 생성자와 비교하여 추가적으로 사용한 변수들은 다음과 같다.

Vertex root : Rooted-dag의 루트를 저장한다.

std::vector<std::vector<Vertex>> dag\_adj : 각 vertex를 인덱스로 하여 vertex들의 vector를 저장하는 vector이다. Vertex v를 인덱스로 하는 vector dag\_adj[v]는 v로부터의 outgoing edge가 있는 vertex들의 vector이다. Dag 처리가 끝나고 나면 dag\_adj[v]는 v의 child vertex들의 vector라고 할 수 있다.

std::vector<std::vector<Vertex>> parents : dag\_adj와 마찬가지로 각 vertex를 인덱스로 하여 vertex들의 vector를 저장하는 vector이다. 그러나 각 vertex의 child의 vector를 저장하는 dag\_adj와 반대로, 이 변수는 각 vertex의 parent의 vector를 저장한다.

Int mark[num\_vertices] : Dag를 만들기 위한 BFS를 할 때에 vertex v의 방문 여부를 기록하기 위한 변수이다. 만약 mark[v]가 1이면 v가 과거 방문 되었다는 뜻이고, 0이면 아직 방문 되지 않았음을 뜻한다.

std::vector<std::pair<double, Vertex>> priority : 각 vertex의 방문순서와 그래프 전체의 루트를 결정하기 위한 우선순위를 저장한다.

생성자 내에서 이루어지는 일은 다음과 같다.

184~235 : 파일을 한 줄 씩 읽어 들이면서 값을 저장한다. Vertex 정보일 경우 해당 vertex의 Label을 label\_[]에 id를 인덱스로 하여 저장하고, edge 정보일 경우 두 vertex에 대해 각 vertex의 child로 서로를 추가한다. 여기서는 directed graph를 가정하고 있으므로 edge가 두개 증가한 것으로 카운트한다.

236~285 : 각 vertex에 대해 우선순위를 계산한 후 저장한다. 우선순위로 해당 vertex의 Candidate set의 사이즈를 degree로 나눈 것을 사용한다. 우선순위 값이 가장 작은 것을 root로 설정한 뒤, 방문표시를 하고(mark[root]=1) 큐에 넣어 BFS를 시작한다. BFS 과정에서 queue 맨 앞의 vertex v의 incoming edge들을 모두 지운다. 이는 v의 이웃들에 대해 그 이웃들의 인접 vertex들의 vectord에서 v를 모두 지움으로써 이루어진다. 지우면서 num\_edges\_도 1씩 감소시킨다. 이 작업이 끝난 v의 이웃에 대해 방문처리를 하고 queue에 넣는다. 모든 v의 이웃이 해당 작업을 마치면 v를 queue에서 빼낸다. v의 이웃이 여러 개 일 때는 우선순위가 작은 것부터 방문한다. Queue가 빌 때 까지 이를 반복하고 나면 모든 에지에 대해 단방향 edge만이 남게 된다.

290~ : 이후의 과정은 위의 과정을 통해 재조정된 dag\_adj를 기반으로 adj\_array\_, num\_labels\_, max\_label\_. label\_frequency\_, start\_offset\_by\_label\_ 등의 값들을 재 조정하는 것으로, 제공된 graph 생성자에서 해당 값들을 초기화하는 과정과 동일하다. adj\_list 대신 dag\_adj를 이용한다는 차이만이 존재한다.

아래는 클래스 dag가 추가적으로 구현한 함수와 graph로부터 오버라이딩한 기타 함수에 대한 설명이다. 대부분이 직관적이므로 간단하게만 소개한다.

**GetParentSize(Vertex v)** : v의 부모 vertex 개수를 리턴한다.

**GetChildSize(Vertex v)** : v의 자식 vertex 개수를 리턴한다.

**GetParent(Vertex v, size\_t i)** : v의 부모 vertex 중 i번째 vertex를 리턴한다.

**GetChild(Vertex v, size\_t i)** : v의 부모 vertex중 i번째 vertex를 리턴한다.

**GetRoot()** : 루트노드를 리턴한다.

1. **backtracking**

구체적인 실행과정을 설명하기에 앞서 제출한 backtrack.cc 및 backtrack.h의 멤버 변수들의 역할을 설명하면 다음과 같다. cnt는 현재까지 구한 embedding의 개수로, 10만개 이상의 embedding을 찾을 경우 실행을 중단하기 위해 사용한다. embedding[ ]은 크기가 query graph의 size인 벡터로, 현재까지 진행된 partial embedding의 정보를 가지고 있다. 이 때 embedding[i]는 query graph의 vertex i에 mapping된 data graph의 vertex가 무엇인지를 나타내며, 이 값이 -1일 경우 아직 query graph의 vertex i에 대한 mapping이 이루어지지 않은 것이다. embedding\_size는 partial embedding에서 현재까지 mapping된 query graph의 vertex의 개수를 나타낸다. extendable[ ]은 backtracking 도중 extendable vertex들을 관리하기 위한 것으로서, extendable[i]에는 query graph의 vertex i에 대해 해당 vertex의 extendable candidate의 개수와, extendable candidate들을 모아놓은 vector가 pair 형태로 저장 되어있다. 그 외 root는 DAG의 root를, q\_size는 query graph의 vertex의 개수를 나타낸다. 마지막으로 embedding\_list는 앞서 출력된 embedding들을 모아놓은 것으로서 실제 실행 시에는 사용되지 않으며, 다만 결과 검증 시 중복된 embedding의 존재 여부를 확인하기 위해 사용된다.

이상의 변수들을 활용하여 backtracking이 수행되는 과정을 함수 별로 설명하면 다음과 같다.

**PrintAllMatches( ):** main에서 backtracking을 위해 호출하는 함수로, embedding descriptor line을 출력한 뒤 backtrack(root)를 호출해 backtracking을 수행한다.

**backtrack(Vertex curr):** 주어진 query graph의 vertex인 curr부터의 backtracking을 수행하는 함수이다. 주어진 vertex가 root인 경우와 root가 아닌 경우를 구분하여 각 경우에 대해 다른 방식으로 수행된다. 주어진 vertex가 root인 경우, 그에 대응하는 Candidate Space의 모든 vertex들이 mapping될 수 있으므로, for문을 통해 이를 하나씩 mapping해주며, 그에 맞게 embedding[ ]과 embedding\_size를 업데이트 해준다. 이 때 만약 embedding\_size==q\_size, 곧 embedding이 구해졌다면 printembedding( )을 통해 주어진 embedding을 출력하고 return한다. 그러지 않을 경우 방금 mapping된 정보를 바탕으로 update\_extendable( )을 통해 앞서 언급한 extendable[ ]을 update한 뒤, 이를 이용해 현재의 extendable vertices중 candidate size가 가장 작은 것은 골라 해당 vertex에 대한 backtrack을 호출한다.

주어진 vertex가 root가 아닌 경우에는 extendable에 존재하는 해당 vertex의 extendable candidate, 곧 embedding 조건을 만족함이 이미 검증된 candidate들만을 for문을 통해 mapping 해주게 되며, 그 외의 과정은 위와 동일하게 진행된다.

이후 주어진 vertex에 대한 가능한 모든 mapping을 진행한 후, update\_extendable(curr)을 통해 주어진 vertex가 partial embedding에서 제거된 것과 관련하여 extendable[ ]의 정보를 업데이트 해준 후 return 한다.

**update\_extendable(Vertex curr):** 주어진 query\_graph의 vertex curr가 partial embedding에 추가되어 mapping이 이루어지거나, partial embedding에서 제거되었을 때, 이와 관련해 extendable[ ]의 정보를 업데이트하는 함수이다. 이를 위해 curr의 모든 child에 대해, 만약 embedding[curr]=-1, 곧 curr가 partial embedding에서 제거되었다면 extendalbe[child]를 extendable vertex가 아니게 수정해준다. 반대로 curr가 partial embedding에 추가되어 새롭게 mapping이 이루어진 경우, child의 모든 parent에 대해 embedding 여부를 확인해 extendable vertex인지를 확인하고, 만약 extendable vertex라면 check\_candidate( ) 함수를 이용해 embedding 조건을 만족시킬 수 있는 candidate들의 개수와 정보를 extendable[child]에 저장한다.

**check\_candidate(Vertex curr, Vertex curr\_cs, const vector<Vertex> &curr\_parent)**: 주어진 query graph의 vertex curr와 그에 대응하는 candidate space의 graph curr\_cs가 embedding 조건을 만족시키면서 mapping될 수 있는지를 확인하는 함수이다. embedding[ ]에 curr\_cs가 이미 존재하는지를 확인해 injectivity를 검증하며, curr\_parrent를 활용해 data graph에 query graph의 edge에 대응되는 부모 vertex와의 edge들이 존재하는지를 확인한다.

**printembedding( ):** embedding을 찾았을 때, 그 결과를 출력 형식에 맞춰 stdout으로 출력하는 함수이다. 주석 처리된 부분은 embedding의 조건을 만족시키지 못하는 잘못된 embedding이 없는지, 중복된 embedding이 없는지의 여부를 함께 출력해주는 부분이다. 이는 코드 검증을 위해 활용한 것으로 실제 수행 시에는 실행되지 않는다.

**check( )**: 구한 embedding이 embedding의 조건 3가지를 만족시키는지 확인하고, 만약 조건을 위배할 경우 어떤 조건을 위배하는지 확인해주는 함수로, 검증을 위해 사용하였다.

**check\_replica( ):** 구한 embedding이 이미 앞서 구했던 embedding가 중복되지 않는지 확인해주는 함수로, 이 또한 검증을 위해 사용하였다.

이상의 backtracking 구현의 핵심은 결국 DAG Ordering과 Candidate Size order를 이용하되, extendable vertex와 candidate에 대한 정보를 extendable[ ]이라는 별도의 공간을 통해 관리하여 이를 효율적으로 업데이트 및 활용할 수 있도록 함에 있다.