Algorithms: Graph Pattern Matching Challenge

서울대학교 컴퓨터공학부

김도현(2019-19669), 이유나(2019-18585)

1. Introduction

본 과제는 제한시간 안에 data graph G와 query graph q가 주어졌을때 subgraph matching을 최대한 많이 찾는 것을 목표로 한다. 구현에 필요한 여러 자료구조들을 data\_structure.h와 data\_structure.cc에 정의 및 구현하여 사용하였다. 이번 과제에서 핵심적으로 이용되는 backtrack은 상황에 따라 크게 세 종류로 구분하여 정의하였다.

답을 효과적으로 탐색하기 위해 택한 핵심 아이디어는 다음과 같다.

1. Avoid vector의 사용. 특정 기간 동안 반드시 특정 후보 노드를 가져야만 하는 vertex에 대한 정보를 저장해두고, 그 후보 노드를 사용하지 않는 것이다.
2. 탐색 순서는 기본적으로 DAG를 활용한다. Query 그래프의 vertex 중에서는 해당하는 후보 노드들 중 어차피 불가능한 것들 ( 두 가지 경우: 이미 다른 vertex에 의해 선택된 후보 노드 / 부모 vertex가 선택한 노드들과 연결되어 있지 않은 후보 노드 )를 미리 제외시킨 다음 남은 후보 노드의 개수가 가장 적은 vertex를 우선 탐색한다. 한 vertex의 후보 노드들 중에서는 data 그래프 상에서 edge의 개수가 가장 적은 후보 노드를 우선 탐색한다.
3. Extendable vertex들을 하나씩 살펴보면서 다음 탐색할 vertex를 정하려 할 때 2)에서 언급한 두 가지 경우에 의해 모든 후보 노드가 제외되는 경우 바로 백트래킹을 한다. 이 때 모든 후보 노드가 제외된 이유에 따라 그 이유가 해소될 때까지 백트래킹을 한다.

2. How to run and Environment

(1) 실행하는 법

원래 주어진 README.md 에 쓰여 있는 방법과 같다.

mkdir build

cd build

cmake ..

make

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

(2) 구현 및 디버깅한 환경

Windows 10 컴퓨터에서 ubuntu를 활용했다.

3. Implementation

(1) data structures

data\_structure.h, data\_structure.cc 파일에 필요한 자료구조들을 class로 구현해둔 후 추상화하여 사용하였다.

\* 탐색 중에 이미 대응하는 candidate set 중 하나를 선택한 query 그래프의 vertex를 ‘활성화되었다’라고 표현했다.

1) SearchStack

traverse의 정보를 담고 있는 가장 핵심적인 스택 구조이다. 각각 SearchStackElement로 이루어져 당시에 보고 있는 query\_node와 선택한 selected\_cs, 다음에 선택할 후보 노드인 next\_cs vector로 구성된다. 각 항목을 set(), get()하는 함수와 스택을 이용하는데에 필요한 pop(), isEmpty() 등을 정의하여 사용하였다. 백트랙할시에 앞에서부터 순서대로 pop 하여 사용할 수 있어 보다 간단한 구현을 도와준다.

2) ActivatedBitArray

len\_query만큼의 길이를 가지는 boolean bit array로, query node가 활성화 되었는지, 즉 선택되었는지를 표시하게 된다. isActivated(), activate(), inactivate() 등의 함수를 구현해두고 사용하였다.

3) BoundaryCountArray

extendable vertex를 판별하기 위해 필요한 것으로, vertex가 이미 선택된 vertex 몇개와 인접하여 있는지를 저장해두는 size\_t array로, getBoundaryCount(), activate(), inactivate() 등의 함수를 구현해두고 사용하였다.

4) IncomingNumberArray

각 vertex마다 몇개의 vertex에서 들어오는 지를 저장해두는 size\_t array로, 역시 extendable vertex를 판별하기 위해 필요하다. count(), getIncomingNumber() 등의 함수를 구현해두고 사용하였다. DAG를 만들 때 함께 만들어지며, 이후 메인 알고리즘에서 BoundaryCountArray 값과 IncomingNumberArray 값이 같으면 extendable vertex로 판단한다.

5) SelectedArray

arr와 check 항목을 가지는 array로, query node마다 선택한 cs의 정보를 담고있다. select(), unselect(), isSelected(), usFull(), getSelectedOf(), getCS() 등의 함수를 구현해두고 사용하였다. 이 항목에 들어있는 것을 print하면 곧 찾은 subgraph를 얻을 수 있다.

6) AvoidVector

AvoidVectorElement로 구성되는데, 각각은 cs, yield, desperate을 가진다. 즉, 백트래킹 도중에 특정 후보 노드는 반드시 어떤 vertex가 가질 수 있어야 함을 발견하면, 그 vertex를 desperate에, 그 후보 노드를 cs에 저장하고, desperate을 위해 양보해준 vertex를 yield에 저장한다. AvoidVectorElement에는 각 항목들의 get(), set() 함수를 구현해두고 사용하였다.

(2) algorithm

1) previous task

search\_stack, len\_query, activated\_bit\_arr, boundary\_count\_arr, incoming\_number\_arr, activated\_count, selected\_arr, avoid\_vector, finish\_flag, DAG, RDAG 등 필요한 자료구조에 대한 포인터 및 및 변수들을 전역으로 선언하고 main 함수 내에서 생성 및 초기화해주었다.

RootForDAG() 함수를 이용하여 cs.GetCandidateSize()/query.GetDegree()가 min 값을 가지는 vertex를 root로 지정해주었다. BuildDAG 함수에서는 이 root에서 시작하여 BFS로 DAG를 vector<Vertex> 자료형으로 만들고, 이때 incoming\_number\_arr도 적절하게 구해주었다. BuildReverseDAG에서 DAG의 directed edge 방향이 반대로 된 RDAG도 만들어 두었다. DAG를 통해서는 특정 vertex의 자식 노드들을 찾기 쉽고, RDAG를 통해서는 특정 vertex의 부모 노드들을 찾기 쉽다.

2) main loop

a. (activated\_count == len\_query)이면 query의 모든 vertex를 본 것으로, selected array를 정답으로 출력하고 backtrack\_3()을 하여 다른 정답을 찾아나선다.

b. 변수 temp\_smallest\_extendable, temp\_smallest\_cs, temp\_smallest\_cs\_num를 정의 및 초기화하였다.

c. (!activated\_bit\_arr[k].isActivated(k)) && (boundary\_count\_arr.getBoundaryCount(l) == incoming\_number\_arr.getIncomingNumber(k))를 만족하는 vertex k를 선택한다. 다시 말해 활성화되지 않은 vertex 중 extendable한 vertex k를 고른다.

d. vertex k의 selected\_cs를 temp\_cs에 저장한다.

e. selected\_array와 activated\_bit\_array를 활용해 temp\_cs 중 이미 선택된 상태인 후보 노드들과 avoid vector에 들어있는 후보 노드들을 제거하고 제거된 후보 노드들을 illuminated 벡터에 저장한다. 그 결과 temp\_cs에 아무것도 남지 않았으면 backtrack\_1()을 수행하게 되며, 이때 매개변수로 illuminated vector과 vertex k를 넘겨준다.

f. RDAG를 이용하여 vertex k의 부모 vertex를 찾고, selected\_arr를 보며 부모 vertex에 해당하는 후보들 중 이미 고른 것들을 확인하여 이들과 모두 edge를 가지는 cs만 temp\_cs에 남긴다. 그 결과 temp\_cs에 아무것도 남지 않았으면 backtrack\_2()을 수행하게 되며, 이때 매개변수로 illuminated vector과 vertex k를 넘겨준다.

g. temp\_cs에 남은 개수가 temp\_smallest\_cs보다 작으면, 작은것부터 보는것이 유리하다고 판단하여 temp\_smallest\_extendable, temp\_smallest\_cs, temp\_smallest\_cs\_num을 모두 vertex k에 대한 값으로 교체한다.

h. 위의 c~g의 과정을 모든 query 그래프의 노드들에 대해 시행한다. temp\_smallest\_cs를 edge 수가 작은 순서대로 정렬한다. search\_stack에 temp\_smallest\_extendable, temp\_smallest\_cs의 맨앞 원소, temp\_smallest\_cs의 두번째부터 마지막까지의 원소들을 push 한다.

i. temp\_smallest\_extendable의 activated\_bit\_arr를 set해준다. DAG를 사용하여 temp\_smallest\_extend의 자식 vertex들의 activated\_bit\_arr, boundary\_count\_arr를 1 증가시키고, selected\_arr도 업데이트한다.

3) backtrack1

a. search\_stack이 비어있으면 탐색종료한다.

b. search\_stack.pop() 하여 temp에 저장한다. temp의 selected\_cs가 illuminated 벡터에 포함되어 있을 경우 c로 간다. 그렇지 않은 경우 해당 query vertex를 inactivate 한 후 a로 돌아간다.

c. avoid vector에 temp의 selected\_cs, query vertex, 그리고 반드시 이 cs를 필요로 하는 k를 추가한다.

d. temp의 next\_cs가 avoid\_vector에 속해있으면 그렇지 않은 후보가 나올때까지 temp.popNextCS()하여 가능한 후보 노드를 찾아, 있을 경우 그 후보를 선택된 후보로 저장한 temp를 search\_stack에 push해준 후 main loop에서의 탐색으로 돌아가 계속해서 수행하게 된다. next\_cs가 남아있지 않은 경우 query vertex를 inactivate하고 한 번 더 pop한 후 d 시작 부분으로 돌아간다.

4) backtrack 2

a. search\_stack이 비어있으면 탐색종료한다.

b. search\_stack.pop() 하여 temp에 저장한다. temp의 selected\_cs가 illuminated 벡터에 포함되어 있거나 query node가 vertex k의 부모 노드일 경우 c로 간다. 그렇지 않은 경우 해당 query vertex를 inactivate 한 후 a로 돌아간다.

c. temp의 next\_cs가 avoid\_vector에 속해있으면 그렇지 않은 후보가 나올때까지 temp.popNextCS()하여 가능한 cs를 찾아, 있을 경우 그 후보를 선택된 후보로 저장한 temp를 search\_stack에 push해준 후 main loop에서의 탐색으로 돌아가 계속해서 수행하게 된다. next\_cs가 남아있지 않은 경우 해당 query vertex를 inactivate하고 b로 돌아가서 같은 과정을 거친다.

5) backtrack 3

a. search\_stack이 비어있으면 탐색 종료한다.

b. search\_stack.pop() 하여 temp에 저장한다. 이때 next\_cs가 avoid\_vector에 속해있으면 그렇지 않은 후보가 나올때까지 temp.popNextCS()하여 가능한 cs를 찾아, 있을 경우 그 후보를 선택된 후보로 저장한 temp를 search\_stack에 push해준 후 main loop에서의 탐색으로 돌아가 계속해서 수행하게 된다. next\_cs가 남아있지 않은 경우 해당 query vertex를 inactivate하고 a로 돌아가서 같은 과정을 거친다.

4. Other Module

알고리즘 구현 및 디버깅에 도움이 될 만한 추가 모듈을 만들었다. performance\_test.h와 performance\_test.cc 파일로 구현했다.

Performance 객체를 생성하고 PrintAllMatches() 이전에 start(), 이후에 end()를 호출하면 백트래킹에 걸린 총 시간을 저장한다. check\_result() 함수에 출력 결과를 정해진 형식대로 저장해 놓은 파일의 경로를 매개변수로 넘겨주면 전체 답 중에서 몇 개의 답이 맞았고, 몇 번째 답이 틀렸는지를 판단해 저장한다. print\_result() 함수를 호출하면 저장된 총 시간, 출력 답 중 맞은 답의 개수 등을 출력한다.

5. Result

주어진 입력에 대해서 모두 실행시킨 뒤 자체적으로 만든 performance\_test 모듈을 활용해 몇 개의 답이 맞는 지 확인했다. 출력되기까지 걸린 시간이 1초 미만이라면 0초라고 기록하였다. (맞은 답 개수) / (구한 답 개수) 로 표기했다.

lcc\_hprd\_n1: 0초, 7/7

lcc\_hprd\_n3: 0초, 46/46

lcc\_hprd\_n5: 0초, 22/22

lcc\_hprd\_n8: 0초, 25/25

lcc\_hprd\_s1: 0초,10/10

lcc\_hprd\_s3: 0초, 43/43

lcc\_hprd\_s5: 0초, 37/37

lcc\_hprd\_s8: 0초, 31/31

lcc\_human\_n1: 0초, 360/360

lcc\_human\_n3: 0초, 0/0

lcc\_human\_n5: 0초, 129/129

lcc\_human\_n8: 0초, 0/0

lcc\_human\_s1: 0초, 74/74

lcc\_human\_s3: 0초, 311/311

lcc\_human\_s5: 0초, 128/128

lcc\_human\_s8: 0초, 244/244

lcc\_yeast\_n1: 0초, 59/59

lcc\_yeast\_n3: 0초, 25/25

lcc\_yeast\_n5: 0초, 18/18

lcc\_yeast\_n8: 0초, 89/89

lcc\_yeast\_s1: 0초, 36/36

lcc\_yeast\_s3: 0초, 0/0

lcc\_yeast\_s5: 0초, 0/0

lcc\_yeast\_s8: 0초, 0/0