Graph-Pattern-Matching-Challenge

자유전공학부 2017-18273 이경원

자유전공학부 2019-15478 박홍근

1. 서론

이번 알고리즘 수업 과제로 진행되는 Graph Pattern Matching Challenge에서는 주어진 data graph G와query graph q에 대해서 G 중에서 q에 해당하는 subgraph를 찾아야 하는 것이 핵심 과제이다. 본 알고리즘 수업의 Graph Pattern Matching Challenge를 2인 팀 프로젝트로 진행하기 위해서 본 팀은 상호 논의 후 각자의 알고리즘으로 구현한 뒤, 각자의 알고리즘 방법을 공유하고 더 결과가 좋은 알고리즘을 선택하는 방법으로 진행하였다. 이번 알고리즘 수업 과제로 진행되는 Graph Pattern Matching Challenge에서는 주어진 data graph G와 query graph q에 대해서 G 중에서 q에 해당하는 subgraph를 찾아야 하는 것이 핵심 과제이다. 본 알고리즘 수업의 Graph Pattern Matching Challenge를 2인 팀 프로젝트로 진행하기 위해서 본 팀은 상호 논의 후 각자의 알고리즘으로 구현한 뒤, 각자의 알고리즘 방법을 공유하고 더 결과가 좋은 알고리즘을 선택하는 방법으로 진행하였다. 코드는 WSL ubuntu 18.04 LTS에서 제공받은 readme의 안내와 동일하게 실행하였다.

2. matching order 및 알고리즘 설명

1) Recursive version 알고리즘 설명

Development environment: C++, Mac, Apple clang version 12.0.0 (clang-1200.0.32.29)

backtracking을 통해서 matching order를 출력하기 위해서 큰 뼈대 코드로는 recursive한 알고리즘을 사용하였다. main 함수 안에서 첫번째 노드에 대한 candidate set의 원소 하나씩을 M에 차례로 넣고 recursive한 backtracking을 각 M vector와 함께 호출한다.

```
for (int i = 0; i < static_cast<int>(cs.GetCandidateSize(0)); i++)
{
  vector<int> M;
  M.push_back(cs.GetCandidate(0, i));
  backtracking(data, query, cs, M);
}
```

recursive한 backtracking 함수에서는 호출할 때마다 vector M의 원소의 개수에 따라서 경우를 2개로 나눠서 다른 작업을 수행한다.

경우 1) 만약 M vector에 query의 vertices만큼이 담겨서 온다면 backtracking 과정 중 leaf node에 도달한 것이므로 recursive 호출을 멈추고 vector M에 담겨있는 element들을 하나씩 출력한 후 return한다.

```
if (M.size() == query.GetNumVertices())
{
  cout << "a ";
  for (int i = 0; i < M.size(); i++)
  {
    cout << M[i] << " ";
  }
  cout << endl;
  total_found++;

  if (total_found > 100000) {
    cout << "total_found embeddings are over 10,0000" << endl;
    exit(0);
  }
```

경우 2) 만약 vector M에 충분한 원소가 담겨서 오지 않은 경우 pruning 과정을 통해서 더 재귀를 호출해도 되는지 판단한 후 조건에 만족하면 M vector에 조건을 만족한 원소를 담아서 재귀를 호출한다. 이때 조건을 만족하는지 여부는 기존 M vector의 vertex들과 해당 vertex의 edge가 G에서도 있는지 확인하면 된다. 만약 query에서는 edge가 있지만 G에서는 없는 경우만 제외하고 나머지 경우 모두 조건을 만족하는 것이므로 backtracking 함수를 재귀 호출한다.

```
else
{
  //다음 CS node들에 대해서 edge들을 check하면서 조건을 만족하면 M에 넣어서 다시 backtracking을 돌린다.
  int k = M.size();
  // print_M(M);
  for (int i = 0; i < static_cast<int>(cs.GetCandidateSize(k)); i++)
  {
    Vertex ID = cs.GetCandidate(k, i);
    //중복되는 vertex가 있는지 check
    if (find(M.begin(), M.end(),ID)!= M.end()){
      continue;
    }

    for(l = 0; l < M.size(); l++){
      if(checkifNeighbor(query, l, M.size()) && !checkifNeighbor(data, M[l], ID)){
        //만약에 query에서 edge가 존재하고 data에서도 edge가 존재하면
        break;
      }
    }
    if (l == M.size()){
      vector<int> M_;
      M_.assign(M.begin(), M.end());
      // print_M(M_);
      M_.push_back(ID);
      // cout << "추가할 수 있는 조건을 만족한 vertex : " << ID << endl;;
      backtracking(data, query, cs, M_);
    }
  }
}
```

2) non-recursive version

우선 실행환경은 다음과 같다. WSL에 설치된 Ubuntu 18.04 LTS로 프로젝트를 진행하였고, 언어는 c++을 사용하였다. 메인 backtracking 알고리즘은 다음과 같다.

1) 현재까지 탐색한 vertex들의 set을 V라 할 때, V에 마지막으로 추가된 data graph vertex에 대하여 그 vertex에 해당하는 candidate set 중에서 하나를 query graph의 vertex에 매칭해준다.

2) 앞서 매칭된 정보를 바탕으로 query graph에서 extendable vertex들을 찾는다.

3) 각 extendable vertex들에 대해 대응되는 candidate set에서 extendable한 candidate의 집합을 구하고, 그 크기가 가장 작은 extendable vertex를 다음으로 탐색할 vertex로 지정한다.

4) 3)의 과정에서 간단한 pruning 기법을 사용하였다. 만약 extendable candidate의 수가 0개인 extendable vertex가 존재한다면, 그 root는 더 이상 탐색해도 의미가 없으므로 그 이전에 탐색했던 vertex로 돌아가 다른 candidate을 매칭한다.

5) 만약 모든 vertex를 다 탐색했다면, 매칭된 결과들을 출력하고 이전 vertex로 돌아간다.

6) 만약 한 vertex의 extendable candidate를 모두 탐색했다면, 그 이전 vertex로 돌아간다.

위와 같은 방법을 계속하면서 embedding을 찾는데, 만약 이 때까지 찾은 embedding의 숫자가 10만개 이상이라면 그 즉시 program을 멈춘다.

Matching order를 위와 같이 정한 이유는 이번 과제의 시간 제한 때문이다. 이번 과제에서는 1분의 시간내에 최대한 많은 embedding을 출력하는 것이 목적이기 때문에, 빠른 시간내에 최대한 많은 올바른 root를 탐색하는 것이 중요하다고 생각했다. 만약 extendable candidate의 수가 많은 vertex부터 탐색을 했다면, 다음 루트를 탐색하기 위해 시작 노드까지 거슬러 올라와야 하는 일이 많아질 것이다. 그렇기 때문에 효율적으로 비교적 많은 embedding들을 찾을 수 있을 것이라 기대되는 extendable candidate가 가장 적은 vertex들을 고르는 방식을 채택하였고, extendable candidate이 0개일 때 더 이상 탐색을 하지 않아도 되는 pruning 기법이라는 부산물까지 얻을 수 있었다.

3. check program 알고리즘 설명

```
//checker process
int num_vertices = query.GetNumVertices();
bool correct = true;
for (int query_vertex = 0; query_vertex < num_vertices; query_vertex++){
  for (int offset = query.GetNeighborStartOffset(query_vertex); offset < (int) query.GetNeighborEndOffset(query_vertex); offset++){
    Vertex neighbor = query.GetNeighbor(offset);
    if (data.IsNeighbor(M[neighbor], M[query_vertex]) == false){
      correct = false;
    }
  }
  if (correct == false) {break;}
}

if (correct) {correct_embedding += 1;}
```

checker program은 최종적으로 출력하는 vertex들이 담긴 M vector의 vertex들이 query에도 edge가 있고 graph에도 edge가 있는지 확인만 해주었다. correct_embedding을 하나씩 늘려서 최종 결과와 correct embedding의 개수가 같은지 확인하면 된다.

4. 선택한 알고리즘과 그 이유

Recursive version과 non-recursive version 중에서 non-recursive version을 선택하였다. 우선 재귀호출로 인한 overhead로 인한 속도저하가 우려되어 속도가 중요한 이번 과제에 영향을 미칠 것이라 생각되어 non-recursive version을 선택하였다.

두 번째 이유는 정확도 때문이다. 앞서 설명한 Check program을 사용한 결과, 10만개 이하로 나오는 test case들은 총 3개로 hprd_n1, hprd_s1, hprd_n5로 각각 96, 504, 32832개의 embedding들이 있다는 것을 확인할 수 있었다. 그래서 두 알고리즘 중에서 non-recursive 알고리즘이 각각 96, 504, 32832개의 embedding을 정확히 찾아내었고, 10만개 이상의 embedding을 가지는 test case에 대해서도 찾아낸 10만개의 matching이 embedding을 만족했기 때문에 non recursive 알고리즘을 채택하였다.

5. 결론 및 보안할 점

빠른 그래프 탐색을 위해서 extendable candidate이 가장 적은 vertex 우선이라는 matching order를 선택하여 구현한 결과, candidate set의 개수가 너무 많아 탐색이 오래 걸리는 lcc_yeast_s8 이외에는 모두 좋은 결과를 얻어낼 수 있었다.

하지만 lcc_yeast의 경우, 탐색 과정에서 중간 노드로 candidate_set이 많은 vertex들이 겹쳐져서 나와 시간이 많이 소요되었다. 이를 해결하기 위해 다음과 같은 방법을 고안해 보았지만 결국 유의미한 결과를 보이지 않아 현재의 알고리즘을 선택하였다. Extendable candidate를 계산하는 과정에서 candidate set의 크기가 일정 이상이라면 계산하지 않고 pass 하는 기법도 고안해 보았지만 이로 인해 얻어지는 시간 단축 효과보다 기초적인 pruning 기법으로 얻어지는 시간 단축

효과가 더 뛰어났으며, candidate set의 크기와 extendable candidate의 크기가 비례하지 않는 경우가 존재하기 때문에 사용하지 않았다. 만약 시간이 더 주어져 알고리즘을 수정할 수 있다면, 부적절하게 사용된 자료구조를 바꾸거나 더욱 빠르게 모든 vertex를 한번씩은 탐색할 수 있는 matching order를 고안하여 lcc_yeast_s8도 통과할 수 있도록 할 것이다.