# Algorithms HW2 Report

2017-11785 현화림

2014-13860 이찬우

## Backtracking Implementation

### State Tree Traversal

저희는 가능한 embedding의 search tree를 상정하고, 이를 traverse 하는 것을 바탕으로 backtracking을 구현하였습니다. Search tree의 node는 가능한 query vertex u, data vertex v 의 순서쌍 (u, v)로 구성되어 있으며,  search tree는 다음과 같은 조건을 만족합니다.

- sibling node들은 모두 같은 query vertex u를 가진다.
- tree의 각 node에 대해서, root부터 이 node까지 이어지는 path는 partial embedding을 나타낸다.
- tree의 height는 $|Q(V)|$ (Q는 query graph)와 같으며, height와 같은 level의 leaf node에 도착했을 때 root부터 이 node 까지의 path는 embedding을 나타낸다. 그 외의 leaf node는 더 이상 partial embedding을 확장시킬 수 없는 상태를 의미한다.

저희는 query graph에 대한 traversal을 바탕으로 search tree를 만들어 나가면서 실시간으로 embedding 여부를 판단하고자 하였습니다. 1분이라는 제한시간 동안 최대한 많은 embedding을 찾아낼 수 있는 것이 본 과제의 목표이므로 DFS를 응용하였고, recursive function call의 오버헤드를 줄이기 위해 stack을 사용해 iterative한 방법으로 구현하였습니다.

DFS를 응용하였다고 표현한 이유는 잘 알려진 iterative한 DFS 방법 만으로는 backtracking에 필요한 모든 정보를 다 관리할 수 없기 때문입니다. graph traversal은 모든 node를 한 번씩 방문하는 것을 목표로 하지만, search tree의 traversal은 maximum height를 가지는 root-leaf 간의 path를 찾는 것을 목표로 합니다. 따라서 새로운 노드를 방문할 때마다 root 부터 이 노드까지의 path를 계속 추적해야 하며, 이 과정은 DFS를 통해 연이어 탐색된 두 노드의 LCA(lowest common ancestor)가 두 노드로부터 멀리 위치할수록 복잡해집니다.

이러한 문제를 효율적으로 해결하기 위해 저희는 먼저 search tree를 traverse하는 데 사용되는 operation을 다음의 3가지로 유형화하였습니다.

1. DOWN: current node의 child로 이동. 이는 current node를 partial embedding에 추가한 뒤, 다음으로 탐색할 extendable vertex를 고르는 것에 대응
2. RIGHT: current node의 sibling으로 이동. 이는 current node(u, v)의 subtree에 대한 탐색을 모두 마치고 u의 다른 extendable candidate v'에 대한 탐색을 이어 나가는 것에 대응.
3. UP: current node의 parent로 이동. 이는 current node와 이 node의 모든 sibling들에 대한 탐색을 마친 뒤 parent로 backtrack 하는 것에 대응.

DFS와 달리 모든 operation은 current node의 level을 최대 1만 변경할 수 있으며, 이는 root로부터 current node까지의 path를 추적하는 것을 가능하게 합니다. 이를 보장하기 위해 저희는 각 query vertex u가 path에 새로 등장할 때마다 stack에 (u, NULL)을 먼저 추가하였습니다. 이 dummy node는 stack의 LIFO policy에 의해 extendable candidates를 담은 node를 아직 하나도 보기 이전, 모두 본 이후, 이 두 가지 경우에만 관측할 수 있어서 current node의 level이 바뀌는 경우(즉, query vertex가 달라지는 경우) 이에 맞추어서 달라지는 partial embedding 정보를 정확하게 갱신하는 것을 가능하게 합니다. 위의 세 가지 stack의 동작에 대응하여 state tree traversal은 아래 3 유형의 state를 가지며 하나의 query vertex u에 대해서, State 1 → State 2 → State 3의 순서로 진행됩니다.

`(u, v) = stack.pop()`

[State 1] `if v == NULL and u is not visited then "DOWN"`

[State 2] `if v != NULL then "RIGHT"`

[State 3] `if v == NULL and u is visited then "UP"`

State 1에서는 partial embedding을 바탕으로 u에 대응될 수 있는 extendable candidates를 찾아 stack에 추가하며, state 2에서는 partial embedding에 (u, v)를 추가한 뒤 다음 extendable vertex u'를 결정하고 stack에 (u', NULL)을 추가합니다. 그리고 state 3에서는 partial embedding에서 (u, v)를 제거합니다. u에 대한 visited mark는 state 1에서는 false→true로, state 3에서는 true→false로 변경이 일어납니다.

**Tracking Partial Embedding and Extendable Vertices**

Partial embedding을 효율적으로 확장시키기 위해서는, partial embedding에 포함될 수 있는 extendable vertices 역시 search tree traversal과 함께 추적해야 합니다. state 2에서 다음 extendable vertex를 결정하므로 state 1에서 u로 인해 새롭게 partial embedding과 연결되는 u'들을 찾아서 extendable vertices에 추가하고, state 3에서는

partial embedding에서 u가 빠짐으로 인해 partial embedding 과의 연결이 끊어지는 u''를 찾아서 extendable vertices에서 제거합니다.

### Pseudo Code

위 내용을 코드로 정리하면 다음과 같습니다. 이는 실제 저희가 사용한 코드의 간략화된 버전이나, 사용된 자료구조는 동일합니다.

```
std::unordered_set<Vertex> query_next;
std::vector<bool> query_visited(query.GetNumVertices(), false);
std::unordered_set<Vertex> data_visited;
std::stack<std::pair<Vertex, Vertex>> pair_to_visit;
std::unordered_map<Vertex, Vertex> partial_embedding;

Vertex root = find_root();
pair_to_visit.push(std::pair<Vertex, Vertex>(root, NULL));

while (!pair_to_visit.empty()) {
  /* Get current vertex */
  std::pair<Vertex, Vertex> current = pair_to_visit.top(); pair_to_visit.pop();
  Vertex u = current.first;
  Vertex v = current.second;

  if (v == NULL) {
    if (query_visited[u]) {
      /* Case 3: Backtrack */
      query_visited[u] = false;
      query_next.insert(u); // insert u back to query_next

      /* Clean Up: remove vertices in query_next which are not connected to
       * partial embedding anymore */
      clean_up_query_next();

      /* Clean Up: delete the pair in partial embedding */
      if ( u is in partial_embedding ) {
        Vertex old_v = partial_embedding[u];
        data_visited.erase(old_v);
        partial_embedding.erase(u);
      }
    } else {
      /* Case 1: Grow */
      pair_to_visit.push(current); // re-push current for Case 3.
      query_visited[u] = true; // mark visited

      // u is no longer in query_next
      query_next.erase(u);

      /* Add unvisited query vertices to query_next
       * which are adjacent to u and not in query_next */
      for (Vertex neighbor : get_neighbors(u, query)) {
        if ( neighbor is not visited before and not in query_next ) {
          query_next.insert(neighbor);
        }
```

```
      }

      /* get extendable candidates of u
       * and add the pairs into stack for Case 2. */
      std::vector<Vertex> candidates = get_extendable_candidates(u);
      for (Vertex candidate : candidates) {
        if ( candidate has not been visited before ) {
          pair_to_visit.push(std::pair<Vertex, Vertex>(u, candidate));
        }
      }
    }
  } else {
    /* Case 2: (u, v) is an extendable candidate */

    /* Clean Up: if there was (u, v') in partial embedding before (u, v), erase */
    if ( partial_embedding contains u ) {
      Vertex old_v = partial_embedding[u];
      data_visited.erase(old_v);
    }

    partial_embedding[u] = v; // update partial_embedding

    if (partial_embedding.size() == query.GetNumVertices()) {
      print_embedding(partial_embedding);
    }
    data_visited.insert(v);

    /* Transition for Case 1 of next level query vertex. */
    Vertex next = get_extendable_vertex(query_next, partial_embedding);
    pair_to_visit.push(std::pair<Vertex, Vertex>(next, NULL));
    }
  }
}
```

- `query_next`: 현재의 partial embedding을 기준으로, partial embedding에 본인이 소속되어 있진 않지만 소속된 query vertex와 연결되어 있는 vertex들을 저장합니다. 이는 본문의 extendable vertices와 같습니다.
- `query_visited(query.GetNumVertices(), false)`: 각 query vertex가 방문된 적이 있는지, 없는지 체크합니다. state1과 3을 구별하기 위해 사용됩니다.
- `data_visited` : query vertex u 마다 injective를 보장하면서 mapping될 수 있는 data vertex v를 찾기 위해 이미 partial embedding 안에 포함된 data vertex들을 보관합니다. query vertex와는 다르게 data vertex는 candidate space에 포함되지 않는 이상 방문될 일이 없고, data vertex의 개수도 query vertex에 비해 매우 많기 때문에 공간 절약을 위해 여기서는 hashing을 사용하였습니다.
- `pair_to_visit` : search tree를 DFS order로 traverse 하기 위해 사용되는 stack 입니다. stack의 각 node (u, v)는 각각 mapping 관계에 있는 query vertex, data vertex를 의미합니다. State 1과 3의 경우 v는 data vertex 대신 NULL이 될 수 있습니다.

- `partial_embedding`: 현재까지의 partial embedding 정보를 저장하는 KV(key-value) store입니다. key는 query vertex, value는 이에 대응되는 data vertex이며, insertion과 deletion을 빠르게 하기 위해 hash 기반의 map을 사용하였습니다. embedding이 완성이 되었을 때는 partial embedding을 key를 기준으로 오름차순으로 정렬하여 출력합니다.

## Determining Matching Order

Matching Order를 정하는 문제란 곧 가능한 extendable vertices 들 중에서 이후에 가장 최소의 탐색을 할 수 있게 하는 최선의 vertex를 고르는 문제로 정리됩니다. 저희는 extendable candidates의 size를 기준으로 extendable vertex를 결정하였으며 (candidate-size order), vertex마다 extendable_candidate의 수를 구하는 과정은 다음과 같습니다.

- query vertex u의 neighbor 중 partial embedding에 소속된 vertex를 찾는다 (= u의 parents)
- u의 전체 candidate 중에서, u의 모든 parent에 대해 M[parent]와 모두 연결된 candidate만 찾아서 센다

```
// u: query vertex to calculate number of extendable candidates
int count = 0;
std::vector<Vertex> parents = get_parents(u, query, embedding);
for (int i=cs.GetCandidateSize(u)-1; i>=0; i--) {
  Vertex candidate = cs.GetCandidate(u, i);
  /* for every parent of vertex u, if M[u_parent] and candidate
   * are not connected, fail */
  bool is_connected = true;
  for (Vertex parent : parents) {
    // parent is guaranteed to be included in embedding
    if (!data.IsNeighbor(embedding.find(parent)->second, candidate)) {
      is_connected = false;
      break;
    }
  }
  if (is_connected) {
    count++;
  }
return count;
```

## Search Result

제공된 `lcc_hprd`, `lcc_human`, `lcc_yeast` 3개의 데이터 그래프와 쿼리 그래프들에 대해 embedding 개수를 출력한 결과는 다음과 같습니다. 문제 스펙에 따라, 100000개가 넘는 결과가 나올 경우 처음으로 발견된 100000개만 출력하였습니다. 모든 경우에 대해 결과는 1분 이내에 다 도출되었습니다.

| | lcc_hprd | lcc_human | lcc_yeast |
|---|---|---|---|
| n1 | 96 | 100,000+ | 100,000+ |
| n3 | 100,000+ | 100,000+ | 100,000+ |
| n5 | 32832 | 100,000+ | 100,000+ |
| n8 | 100,000+ | 100,000+ | 100,000+ |
| s1 | 504 | 100,000+ | 100,000+ |
| s3 | 100,000+ | 100,000+ | 100,000+ |
| s5 | 100,000+ | 100,000+ | 100,000+ |
| s8 | 100,000+ | 100,000+ | 100,000+ |

## Environment

조교님들께서 challenge repository에 올려주신 cpp skeleton code를 사용하여 구현하였습니다. README에 적어주신 build 및 실행 방법 그대로 실행해주시면 동작합니다. 사용한 머신의 스펙은 다음과 같습니다.

```
processor       Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
memory          256KiB L1 cache
memory          1MiB L2 cache
memory          8MiB L3 cache
memory          256KiB L1 cache
memory          32GiB System Memory
memory          8GiB DIMM DDR4 Synchronous 2133 MHz (0.5 ns)
memory          8GiB DIMM DDR4 Synchronous 2133 MHz (0.5 ns)
memory          8GiB DIMM DDR4 Synchronous 2133 MHz (0.5 ns)
memory          8GiB DIMM DDR4 Synchronous 2133 MHz (0.5 ns)
```