Algorithm: HW2

최동석, 구경회

1 작동 환경

컴파일러의 경우 gcc 9.3.0, 운영체제는 ubuntu 20.04, cmake의 버전은 3.16.3이다.

2 빌드 및 실행 방법

주어진 스켈레톤 코드를 기반으로 하여 README.md에 있는 것과 같은 방식으로 동작한다. build를 위해서는 다음과 같이 실행하면 된다.

```
mkdir build
cd build
cmake ..
make
```

실행을 위해서는 다음과 같이 입력한다.

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

3 Matching Order & Backtracking

candidate size를 기반으로 탐색할 경우 탐색이 실패할 때 더 적은 비교가 이루어진다. 예를 들어 0, 1, 2번 vertex에 대응하는 candidate가 1 - 10 - 100개라고 하자. 1, 2번 vertex 모두 아무것도에도 대응되지 않는다. 이 때 2번 노드를 먼저 탐색하는 경우 100번의 비교가 이루어져야 탐색이 끝나지만 1번 노드를 먼저 방문하는 경우 10번이면 상태 공간 트리의 탐색이 끝난다.

3.1 Root Find

```
void Backtrack::FindRoot(const Graph &query, const CandidateSet &cs) {
    for (size_t i = 0; i < query.GetNumVertices(); i++) {
        if (cs.GetCandidateSize(i) < cs.GetCandidateSize(root)) {
            root = i;
        }
    }
}
```

상태 공간 트리의 루트를 찾기 위해 candidate size가 가장 작은 vertex를 찾는다.

3.2 Candidate size 기반 Matching Order

다음 함수를 통해 연장 가능한 Vertex를 찾는다. 이미 임베딩 되어있는 Vertex를 기반으로 (1) 해당 Vertex가 임베딩되었는지 (2) 임베딩된 Vertices와 해당 Vertex가 인접한지 확인한다. 임베딩 되어있지 않고 이미 임베딩된 Vertices 중 하나 이상과 인접하면 연장 가능하다. 이 때 연장 가능한 Vertices중 가장 candidate size가 작은 Vertex를 선택하여 임베딩을 진행한다.
만약 이미 방문했거나 query graph 상에서는 인접하지만 실제 data graph에서는 인접하지 않는 경우에는 건너뛴다.

```
Vertex Backtrack::GetExtendableVertex(const Graph &query, const CandidateSet &cs) {
    vector<pair<size_t, Vertex>> qVertices;

    for (Vertex notEmVertex : not_embedded) {
        for (Vertex emVertex : embedded) {
            bool extendable = query.IsNeighbor(notEmVertex, emVertex);

            if (extendable) {
                qVertices.emplace_back(make_pair(cs.GetCandidateSize(notEmVertex), notEmVertex));
                break;
            }
        }
    }

    if (qVertices.empty()) return null;

    std::make_heap(qVertices.begin(), qVertices.end(), greater<pair<size_t, Vertex>>());

    return qVertices.front().second;
}
```

3.3 Backtrack

실제 Backtrack이 이루어지는 부분의 코드이다. qVertex에 해당하는 embedding을 찾고 임베딩하는 부분이다. 리프 노드에 도달한 경우 embedding의 순서에 맞추어 답을 출력한다. candidate들은 index 순서대로 방문하며 query data에서는 인접하지만 실제 data graph에서는 인접하지 않는 경우나 이미 방문한 경우에는 건너뛰고 다음 candidate를 보게 된다.

```
void Backtrack::PrintMatch(const Graph& data, const Graph& query,
                           const CandidateSet& cs, const Vertex &qVertex) {
    if (qVertex == null) {
        if (count > 100000) {
            exit(0);
        } else {
```

```
            vector<Vertex> ans(query.GetNumVertices(), -1);
            for (size_t i = 0; i < embedded.size(); i++) {
                ans[embedded[i]] = path[i];
            }
            return;
        }
    }

    for (size_t i = 0; i < cs.GetCandidateSize(qVertex); i++) {
        const Vertex current_candidate = cs.GetCandidate(qVertex, i);

        if (v[current_candidate]) {
            continue;
        }

        v[current_candidate] = true; path.push_back(current_candidate);
        embedded.push_back(qVertex);
        not_embedded.erase(std::remove(not_embedded.begin(), not_embedded.end(), qVertex),
            not_embedded.end());
        bool gotoNext = true;

        for (size_t p = 0; p < embedded.size() - 1; p++) {
            if (query.IsNeighbor(embedded[p], qVertex)
                && !data.IsNeighbor(path[p], current_candidate)) {
                gotoNext = false;
                break;
            }
        }

        if (gotoNext) {
            PrintMatch(data, query, cs, GetExtendableVertex(query, cs));
        }

        v[current_candidate] = false; path.pop_back();
        embedded.erase(std::remove(embedded.begin(), embedded.end(), qVertex), embedded.end());
        not_embedded.push_back(qVertex);
    }
}
```