Hungarian Algorithm

Getting Started

```
sudo apt install -y cmake g++
mkdir build && cd build
cmake ..
make && make test
./demo
```

Explanation

알고리즘 구현은 "James Munkres" *Algorithms for the Assignment and Transportation Problems* 를 참고하여 구현하였으며, 이를 Python/C++ 로 구현 및 설명한 두개의 문서(*Tutorial on Implementation of Munkres' Assignment Algorithm*, *Munkres Algorithm For Assignment Problem: A Tutorial*)를 참고하였다.

File structure 및 클래스 API는 아래와 같이 구현되어있다.

- File structure

```
├── CMakeLists.txt
├── README.md
├── results.txt
├── src
│   └── hungarian_algorithm
│       ├── include
│       │   └── hungarian_assigner.hpp -> HungarianAssigner Class
Declaration
│       └── src
│           └── hungarian_assigner.cc  -> HungarianAssigner Class
Definition
└── test
    ├── demo.cc -> "demonstration code"
    └── test.cc -> "test code with gtest"
```

- solve API
  - HungarianAssigner::solve API

```
/**
* @brief solve the assignment problem
*
* @param cost_matrix: cost matrix of the assignment problem (n x
m, row
* major)
* @param n: number of rows of the cost matrix
```

```
 * @param m: number of columns of the cost matrix
 * @param mode: 0: minimize the total cost, 1: maximize the total
cost
 * @param assignment_index: index of the assignment, -1 if
unassigned
 * @return float: total cost of the assignment
 */
float solve(const CostType &cost_matrix, const size_t n, const
size_t m,
            const int mode, AssignmentType *assignment_index);
```

  - exmaple

```
HungarianAssigner assigner;
HungarianAssigner::CostType cost_matrix = {
    {3, 7, 5, 11},
    {5, 4, 6, 3},
    {6, 10, 1, 1},
};
HungarianAssigner::AssignmentType assignment_index;
auto cost = assigner.solve(cost_matrix, 3, 4, 1,
&assignment_index);
```

- unvalanced case
  - 기존 Hungarian algorithm은 행/열의 크기가 같은 경우 optimal assignment 를 구하고 있지만, 행렬의 크기가 다른 경우에도 알고리즘이 동작하도록 입력된 matrix에 zero padding 을 수행하였다. zero padding을 통해 행/열의 크기가 같아진 matrix를 이용하여 optimal assignment 를 구하고, 원래 사이즈 정보를 이용하여 assignment index 결과를 저장한다.
- cost maximization
  - 기존 Hungarian algorihtm은 cost를 minimization 하는 알고리즘으로, cost를 maximization 하기 위해서는 입력 cost matrix의 부호를 반대로 하여 알고리즘을 수행하였다.
- Time complexity: `O(N^4)`
  - 구현한 알고리즘은 4단계로 구성되어 `preliminaries` 단계, `step1`, `step2`, `step3` 순서로 수행되며, 조건에 따라 `step1` -> `step3` -> `step1` 또는 `step1` -> `step2` -> `step3` -> `step1` 을 반복한다. 이러한 반복은 알고리즘을 수행하면서 비용함수가 최저가 되는 조합을 만들어 내기 위한 과정으로, `N`개의 assignment가 필요할 경우, `NxN`의 cost matrix가 주어지면, 최악의 경우 `n-1` 번 반복하게 된다. `step1`의 시간복잡도가 `O(N^3)` 이므로(`step2`, `step3`는 `O(N^2)`), 알고리즘의 시간복잡도는 `O(N^4)` 이다.
- `O(N^3)` Hungarian Algorithm
  - David Krouse의 *On implementing 2D rectangular assignment algorithms 2016* 논문을 참고하면 O(N^3) 의 시간 복잡도로 동작하는 알고리즘 구현이 가능하다고 한다. 이부분은 추후 확인해보아야겠다.

Reference

- Algorithms for the Assignment and Transportation Problems (James Munkres)
- Tutorial on Implementation of Munkres' Assignment Algorithm
- Munkres Algorithm For Assignment Problem: A Tutorial
- What is the time/space complexity of scipy.optimize.linear_sum_assignment?