

Звіт з теорії алгоритмів

Домашнє завдання 3, варіант 7

Яквоенко Олена

Завдання полягає у виборі мінімальної вартості призначення п виконавців на п задач так, щоб кожен виконавець отримав рівно одну задачу і кожна задача була призначена рівно одному виконавцю.

Угорський алгоритм (Hungarian Algorithm)

поліноміальниq метод розв'язання задачі призначення з часовою складністю $O(n^3)$. Алгоритм поступово перетворює матрицю витрат так, щоб мінімально можлива сума призначень могла бути знайдена через нульові елементи.

Крок 1. Формування матриці витрат

Створюємо квадратну матрицю С, де кожен елемент - це вартість призначення виконавця і на задачу j.

Крок 2. Row Reduction

Для кожного рядка обчислюємо мінімальний елемент і віднімаємо його від усіх елементів рядка.

Це гарантує наявність принаймні одного нуля в кожному рядку.

Крок 3. Скорочення стовпців

Проводимо аналогічне зменшення для кожного стовпця.

Після цього матриця містить множину нулів, які є потенційними призначеннями.

Крок 4. Покриття нулів мінімальною кількістю ліній

Мета — накрити всі нулі найменшою кількістю горизонтальних та вертикальних ліній.

- Якщо кількість ліній = n , ми можемо знайти оптимальне призначення.
- Якщо ні — матриця потребує подальшої трансформації.

Крок 5. Побудова нової матриці

1. Знаходимо мінімальний некритичний елемент (b).
2. Віднімаємо його від усіх некритичих елементів.
3. Додаємо b до елементів, що знаходяться на перетині ліній.

Це створює нові нулі, дозволяючи побудувати повне призначення.

Крок 6. Повторення

Повертаємося до кроку 4 доти, доки можливе повне призначення.

Крок 7. Побудова оптимального призначення

Серед нулів вибираємо такі, щоб:

- у кожному рядку був рівно один вибір,
- у кожному стовпці — теж рівно один.

Крок 8. Визначення мінімальної вартості

Сумуємо $C_{i,p(i)}$, де $p(i)$ — колонка вибраного нуля.

```
#include "hungarian.hpp"
#include <limits>
#include <queue>

namespace assign {

    // matchR[i] = j — у якій колонці стоїть одиниця (нуль у редукованій матриці
```

```

i) для рядка i
//cost — мінімальна сумарна вартість за оригінальною матрицею C_in
std::pair<std::vector<int>, long long>
hungarian_min_sum(const std::vector<std::vector<long long>>& C_in) {
    const int n = (int)C_in.size();

    // Локальна копія матриці витрат, яку будемо модифікувати
    std::vector<std::vector<long long>> C = C_in;

    // 1) Row reduction (редукція по рядках)
    for (int i = 0; i < n; ++i) {
        long long mn = C[i][0];
        for (int j = 1; j < n; ++j)
            if (C[i][j] < mn) mn = C[i][j];
        for (int j = 0; j < n; ++j)
            C[i][j] -= mn;
    }

    // 2) Column reduction (редукція по стовпцях)
    for (int j = 0; j < n; ++j) {
        long long mn = C[0][j];
        for (int i = 1; i < n; ++i)
            if (C[i][j] < mn) mn = C[i][j];
        for (int i = 0; i < n; ++i)
            C[i][j] -= mn;
    }

    // побудова "графа нулів" (zero-graph)
    // adj[i] — перелік стовпців j, для яких C[i][j] == 0.
    // На цьому двобічному графі будемо шукати максимальне пароспівста
    влення (row ↔ col) використовуючи DFS
    auto build_zero_graph = [&](std::vector<std::vector<int>>& adj){
        adj.assign(n, {});
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (C[i][j] == 0)

```

```

        adj[i].push_back(j);
    };

// matchC[v] = u — стовпець v уже "зайнятий" рядком u; -1 — вільний
// used — відмічає стовпці, які вже пробували в цьому DFS.
auto dfs = [&](auto&& self, int u,
               const std::vector<std::vector<int>>& adj,
               std::vector<int>& matchC,
               std::vector<char>& used) {
    for (int v : adj[u]) if (!used[v]) {
        used[v] = 1;
        if (matchC[v] == -1 || self(self, matchC[v], adj, matchC, used)) {
            matchC[v] = u;
            return true;
        }
    }
    return false;
};

// Граф нулів
std::vector<std::vector<int>> adj;
std::vector<int> matchC(n, -1); // стовпець → рядок (поточне пароспівства
влення)
std::vector<int> matchR(n, -1); // рядок → стовпець (фінальний результат)

// - шукаємо максимальне пароспівставлення на графі нулів;
// - якщо воно вже perfect (розмір n) — готово, обчислюємо відповідь;
// - інакше будуємо мінімальне покриття нулів, знаходимо delta, коригуємо матрицю та повторюємо
while (true) {
    // 2.1) Побудували граф нулів по поточній С
    build_zero_graph(adj);
}

```

```

std::fill(matchC.begin(), matchC.end(), -1);

// 2.3) Прагнемо знайти matching максимальної потужності
int msize = 0;
for (int i = 0; i < n; ++i) {
    std::vector<char> used(n, 0);
    if (dfs(dfs, i, adj, matchC, used))
        ++msize;
}

// 2.4) Якщо вже є perfect matching ( $|M|=n$ ), формуємо matchR і повертаємо результат
if (msize == n) {
    for (int j = 0; j < n; ++j)
        if (matchC[j] != -1)
            matchR[matchC[j]] = j;

    long long cost = 0;
    for (int i = 0; i < n; ++i)
        cost += C_in[i][matchR[i]];

    return {matchR, cost};
}

// 2.5) Побудова мінімального лінійного покриття нулів
// - matchRow[i] = j якщо рядок i у matching парований зі стовпцем j, інакше -1
// - Починаємо з усіх Непарованих рядків (matchRow[i] == -1), позначаємо їх як "видимі" (visRow=1) і кладемо в чергу
// - З видимого рядка переходимо по нулях у стовпці (робимо стовпець видимим)
// - Якщо стовпець має парний рядок, робимо той рядок видимим і теж додаємо в чергу
//
// Після обходу:

```

```

// - Лінії малюємо через НЕвидимі рядки (0) та видимі стовпці (1)
std::vector<char> visRow(n, 0), visCol(n, 0);
std::queue<int> q;

// matchRow: рядок → стовпець за поточним matching
std::vector<int> matchRow(n, -1);
for (int j = 0; j < n; ++j)
    if (matchC[j] != -1)
        matchRow[ matchC[j] ] = j;

// Старт: усі непаровані рядки позначаємо й додаємо в чергу
for (int i = 0; i < n; ++i)
    if (matchRow[i] == -1) {
        visRow[i] = 1;
        q.push(i);
    }

// BFS по черзі: рядок → (нулями) → стовпець → (парним ребром) → р
ядок ...
while (!q.empty()) {
    int r = q.front(); q.pop();
    for (int c : adj[r]) {
        if (!visCol[c]) {
            visCol[c] = 1;
            if (matchC[c] != -1 && !visRow[ matchC[c] ]) {
                visRow[ matchC[c] ] = 1;
                q.push(matchC[c]);
            }
        }
    }
}

// 2.6) Обчислення delta — мінімального ненакритого елемента
//      delta = min{ C[i][j] | visRow[i] == 1 && visCol[j] == 0 }
long long delta = std::numeric_limits<long long>::max();

```

```

for (int i = 0; i < n; ++i) if (visRow[i])
    for (int j = 0; j < n; ++j) if (!visCol[j] && C[i][j] < delta)
        delta = C[i][j];

// 2.7) Оновлення матриці
//
// - Відняти delta з усіх ненакритих клітин (visRow[i]=1 і visCol[j]=0).
// - Додати delta до клітин, накритих двома лініями
// (тобто visRow[i]=0 і visCol[j]=1; це "перехрещення" ліній).
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) {
        if (visRow[i] && !visCol[j])    C[i][j] -= delta; // "uncovered"
        else if (!visRow[i] && visCol[j]) C[i][j] += delta; // "covered twice"
        // інші комірки (covered once) не змінююємо
    }
    // Після оновлення повертаємося на початок циклу: будуємо новий гр
    // аф нулів і намагаємося побудувати більше пароспівставлення.
}
}

}

```

Нижче наведено два приклади роботи алгоритму (вручну), що демонструють усі ключові кроки:

row reduction column reduction

$\begin{bmatrix} 4 & 1 & 3 \\ 2 & 0 & 5 \\ 3 & 2 & 2 \end{bmatrix}$	$\min=1$
$\begin{bmatrix} 3 & 0 & 2 \\ 2-0 & 0-0 & 5-0 \\ 2 & 0 & 5 \end{bmatrix}$	
$\begin{bmatrix} 1 & 0 & 0 \\ 3-2 & 2-2 & 2-2 \\ 0 & 0 & 2 \end{bmatrix}$	$\min=1, 0, 2$

$\begin{bmatrix} 3 & 0 & 2 \\ 2-1 & 0-0 & 2-0 \\ 2 & 0 & 2 \end{bmatrix}$	
$\begin{bmatrix} 1 & 0 & 5 \\ 2-1 & 0-0 & 5-0 \\ 1 & 0 & 5 \end{bmatrix}$	
$\begin{bmatrix} 0 & 0 & 0 \\ 1-1 & 0-0 & 0-0 \\ 0 & 0 & 0 \end{bmatrix}$	

$\begin{bmatrix} 2 & 0 & 2 \\ 1 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix}$	$\min=1$
$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 4 \\ 0 & 1 & 0 \end{bmatrix}$	$\min=1$
$\begin{bmatrix} 4 & 1 & 3 \\ 2 & 0 & 5 \\ 3 & 2 & 2 \end{bmatrix}$	

$\min\text{-cost}=5$

row reduction column reduction

$\begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$	$\min=2$
$\begin{bmatrix} 7 & 0 & 5 & 6 \\ 6-3 & 4-3 & 3-3 & 7-3 \\ 3 & 1 & 0 & 4 \\ 5-1 & 8-1 & 1-1 & 8-1 \end{bmatrix}$	$\min=3$
$\begin{bmatrix} 4 & 7 & 0 & 7 \\ 7-4 & 8-4 & 0-4 & 4-4 \\ 3 & 2 & 5 & 0 \end{bmatrix}$	$\min=3, 0, 0, 0$

$\begin{bmatrix} 7-3 & 0-0 & 5-0 & 6-0 \\ 3-3 & 1-0 & 0-0 & 4-0 \\ 0 & 1 & 0 & 4 \\ 4-3 & 7-0 & 0-0 & 7-0 \end{bmatrix}$	
$\begin{bmatrix} 1 & 7 & 0 & 7 \\ 3-3 & 8-0 & 5-0 & 0-0 \\ 0 & 2 & 5 & 0 \end{bmatrix}$	

$\begin{bmatrix} 4 & 0 & 5 & 6 \\ 0 & 1 & 0 & 4 \\ 1 & 7 & 0 & 7 \\ 0 & 2 & 5 & 0 \end{bmatrix}$	
$\begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$	

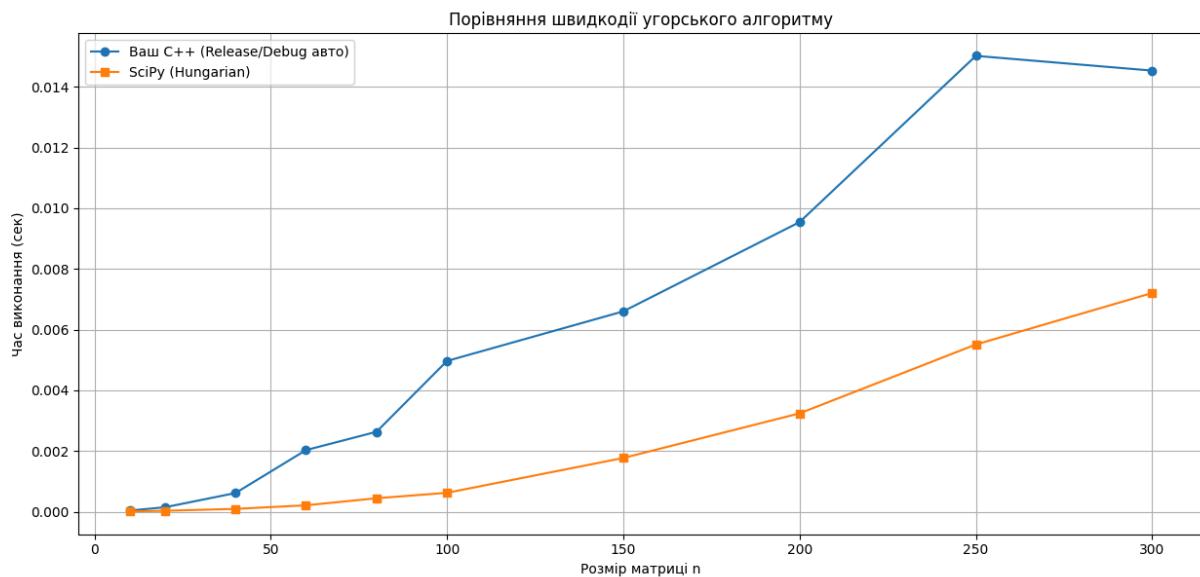
$\min\text{-cost}=13$

Порівняння з бібліотичною реалізацією (SciPy)

Для порівняння використовувалась функція `scipy.optimize.linear_sum_assignment`

Створено бенчмарк, який на python, який:

- генерує випадкові квадратні матриці різних розмірів,
- запускає мою C++ реалізацію,
- запускає SciPy для тієї ж матриці,
- усереднює час за 5 прогонів,
- будує графік.



Аналіз:

- Обидві реалізації демонструють зростання часу приблизно як $O(n^3)$.
- Реалізація SciPy швидша ($\approx 2\text{--}3\times$), що очікувало, адже:
 - писана з високим рівнем оптимізації
 - використовує сучасні покращення алгоритму (я реалізовувала "класичну версію")
 - мінімізує виділення пам'яті

Моя реалізація, хоча й повністю коректна, має більше допоміжних кроків і ґрунтуються на простішій версії алгоритму, що впливає на швидкодію.

Порівняння результатів (коректність)

На всіх розмірах матриць результати: SciPy та моя C++ реалізація повністю збігалися за сумарною вартістю.