Лабораторная работа № 8 по курсу дискретного анализа: жадные алгоритмы

Выполнил студент группы М80-308Б-22 МАИ *Цирулев Николай*.

Условие

Вариант №4: Откорм бычков
Бычкам дают пищевые добавки, чтобы ускорить их рост. Каждая добавка содержит некоторые из N действующих веществ. Соотношения количеств веществ в добавках могут отличаться. Воздействие добавки определяется как $c_1a_1 + c_2a_2 + \cdots + c_{NaN}$, где a_i- количество i-го вещества в добавке, c_i - неизвестный коэффициент, связанный с веществом и не зависящий от добавки. Чтобы найти неизвестные коэффициенты c_i, Биолог может измерить воздействие любой добавки, использовав один её мешок. Известна цена мешка каждой из $M \leq N$) различных добавок. Нужно помочь Биологу подобрать самый дешевый наобор добавок, позволяющий найти коэффициенты c_i. Возможно, соотношения веществ в добавках таковы, что определить коэффициенты нельзя.

Формат ввода

В первой строке текста - целые числа M и N; в каждой из следующих M строк записаны N чисел, задающих соотношение количеств веществ в ней, а за ними - цена мешка добавки. Порядок веществ во всех описаниях добавок один и тот же, все числа - неотрицательные целые не больше 50.

Формат вывода

Вывести -1 если определить коэффциенты невозможно, иначе набор добавок (и их номеров по порядоку во входных данных). Если вариантов несколько, вывести какой-либо из них.

Метод решения

Цель задачи — найти минимальный по стоимости набор добавок, который позволяет определить коэффициенты c_i.
Идея такова - представим входные данные в виде системы линейных алгебраических уравнений, где добавки - это x_i, а цена мешка добавки - это правая часть уравнения вида: $a_{1j} * x_1 + a_{2j} * x_2 + ... + a_{ij} * x_i + ... + a_{in} * x_n = b_j$.
Это значит, что для решения данной системы нужно, чтобы матрица системы имела N базисных строк. Для нахождения базисных строк матрицы воспользуемся немного модифицированным методом Гаусса. Приведем матрицу к ступенчатому виду, однако всегда среди двух линейно зависимых строк будем выбирать тот, чья стоимость ниже. Данный алгоритм - жадный, так как мы каждом шаге выбираем локально оптимальное решение и в конце у нас получается оптимальное решение.

Описание программы

Структура TTriplet

```
struct TTriplet {
    std::vector<double> vec;
    int initIdx;
    int price;
};
```

Эта структура объединяет данные о каждой добавке:

- `vec` — коэффициенты веществ в добавке.
- `initIdx` — индекс добавки.
- `price` — стоимость добавки.

Ввод данных

```
int m, n;
std::cin >> m >> n;
std::vector<TTriplet> vectors(m, TTriplet{std::vector<double>(n), 0, 0});
for (int i = 0; i < m; ++i) {
    for (int j = 0; j < n; ++j) {
        std::cin >> vectors[i].vec[j];
    }
    std::cin >> vectors[i].price;
    vectors[i].initIdx = i;
}
```

- Считывается m-наборов данных.
- Каждая добавка сохраняется в виде структуры `TTriplet`.

Приведение к ступенчатому виду

```
for (int i = 0; i < n; ++i) {
    int minVecIdx = -1;
    int minPrice = std::numeric_limits<int>::max();

    for (int j = i; j < m; ++j) {
        if (vectors[j].vec[i] != 0.0 && vectors[j].price < minPrice) {
            minPrice = vectors[j].price;
            minVecIdx = j;
        }
```

```
    }

    if (minVecIdx == -1) {
        std::cout << "-1\n";
        return;
    }

    std::swap(vectors[i], vectors[minVecIdx]);
    res.push_back(vectors[i].initIdx);
```

- Для каждого столбца i:
  - Выбирается строка с минимальной стоимостью, имеющая ненулевой коэффициент в текущем столбце.
  - Если такая строка не найдена, выводится `-1`, так как система неразрешима.
  - Найденная строка переставляется на текущую позицию.

Обнуление элементов ниже диагонали

```
for (int j = i + 1; j < m; ++j) {
    double multiplier = vectors[j].vec[i] / vectors[i].vec[i];
    for (int k = i; k < n; ++k) {
        vectors[j].vec[k] -= vectors[i].vec[k] * multiplier;
    }
}
```

- Все строки ниже текущей модифицируются так, чтобы в текущем столбце i коэффициенты стали равны нулю (стандартный шаг метода Гаусса).

Вывод результата

```
std::sort(res.begin(), res.end());
for (int elem : res) {
    std::cout << elem + 1 << " ";
}
std::cout << std::endl;
```

- Индексы выбранных строк (добавок) сортируются в порядке возрастания.
- Выводятся с учётом 1-индексации.

```cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;
using ll = long long;

int main() {
    ll n;
    cin >> n;
    vector<ll> steps(n + 1, 0);
    vector<ll> c(n + 1, numeric_limits<ll>::max());
    c[1] = 0;

    for (ll i = 2; i <= n; ++i) {
        if (c[i - 1] + i < c[i]) {
            c[i] = c[i - 1] + i;
            steps[i] = -1;
        }
        if (i % 2 == 0 && c[i / 2] + i < c[i]) {
            c[i] = c[i / 2] + i;
            steps[i] = 2;
        }
        if (i % 3 == 0 && c[i / 3] + i < c[i]) {
            c[i] = c[i / 3] + i;
            steps[i] = 3;
        }
    }
    vector<string> res;
    ll i = n;
    while (i > 1) {
        if (steps[i] == -1) {
            res.push_back("-1");
            i -= 1;
        } else if (steps[i] == 2) {
            res.push_back("/2");
            i /= 2;
        } else if (steps[i] == 3) {
            res.push_back("/3");
            i /= 3;
        }
    }
```

```
    cout << c[n] << endl;
    for (ll i = 0; i < res.size(); i++) {
        cout << res[i] << (i == res.size() - 1 ? "" : " ");
    }
    cout << endl;
    return 0;
}
```

Дневник отладки

После отправки решения в чекер не было обнаружено ошибок.

Тест производительности

Асимптотика написанного алгоритма - $O(MN^2)$. Для лучшей наглядности приведём таблицу, в которой время работы алгоритма сопоставляется с вводимыми данными. Для удобства расчетов приравняем N к M.

N	Время работы алгоритма, мс
125	9327
250	70629
500	587350
1000	4674816

Как показано в таблице, время выполнения алгоритма $T = CN^3, C = const$. Это соответствует ожидаемой сложности нашего алгоритма. Ниже приведена программа benchmark.cpp, использовавшаяся для определения времени работы алгоритма:

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>
#include <random>
#include <chrono>

struct TTriplet {
    std::vector<double> vec;
    int initIdx;
    int price;
};

void GaussWithPriority(int m, int n, std::vector<TTriplet> vectors) {
    std::vector<int> res;
```

```
    for (int i = 0; i < n; ++i) {
        int minVecIdx = -1;
        int minPrice = std::numeric_limits<int>::max();

        for (int j = i; j < m; ++j) {
            if (vectors[j].vec[i] != 0.0 && vectors[j].price < minPrice) {
                minPrice = vectors[j].price;
                minVecIdx = j;
            }
        }

        if (minVecIdx == -1) {
            //std::cout << "-1\n";
            return;
        }

        std::swap(vectors[i], vectors[minVecIdx]);
        res.push_back(vectors[i].initIdx);

        for (int j = i + 1; j < m; ++j) {
            double multiplier = vectors[j].vec[i] / vectors[i].vec[i];
            for (int k = i; k < n; ++k) {
                vectors[j].vec[k] -= vectors[i].vec[k] * multiplier;
            }
        }
    }

    std::sort(res.begin(), res.end());
    //for (int elem : res) {
    //    std::cout << elem + 1 << " ";
    //}
    //std::cout << std::endl;
}

int main() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution int_dist(0, 50);
    for (int k = 125; k <= 1000; k *= 2) {
        std::cout << k << " & ";
        int m = k;
        int n = k;
```

```
        std::vector<TTriplet> vectors(m, TTriplet{std::vector<double>(n), (
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                vectors[i].vec[j] = int_dist(gen);
            }
            vectors[i].price = int_dist(gen);
            vectors[i].initIdx = i;
        }
        auto start1 = std::chrono::high_resolution_clock::now();
        GaussWithPriority(m, n, vectors);
        auto finish1 = std::chrono::high_resolution_clock::now();
        auto duration1 = std::chrono::duration_cast<std::chrono::microsecon
        std::cout << duration1.count() << " \\" << "\\ ";
        std::cout << '\n';
    }
    return 0;
}
```

Выводы

В ходе выполнения данной работы была подробно изучена концепция жадных алгоритмов, а также углублены знания в области линейной алгебры, в частности, метода Гаусса. На практике полученные теоретические знания были применены для разработки эффективного алгоритма, который сочетает оптимальную временную сложность и рациональное использование памяти.
Особое внимание было уделено модификации классического метода Гаусса с учётом дополнительных критериев, таких как минимизация стоимости, что позволило продемонстрировать практическое применение математических методов для решения реальных оптимизационных задач.
Полученные результаты могут служить основой для дальнейшего изучения оптимизационных методов и применения их в более сложных задачах.