Лабораторная работа №6 по курсу Дискретного Анализа: Калькулятор

Выполнил студент группы 08-308 МАИ Жерлыгин Максим Андреевич

Условие

При помощи метода динамического программирования разработать алгоритм решения задачи, определяемой своим вариантом; оценить время выполнения алгоритма и объем затрачиваемой оперативной памяти. Перед выполнением задания необходимо обосновать применимость метода динамического программирования.

Разработать программу на языке C или C++, реализующую построенный алгоритм. Формат входных и выходных данных описан в варианте задания:

Вариант 5:
Задана матрица натуральных чисел **A** размерности **n** × **m**. Из текущей клетки можно перейти в любую из 3-х соседних, стоящих в строке с номером на единицу больше, при этом за каждый проход через клетку **(i, j)** взымается штраф $A_{i,j}$. Необходимо пройти из какой-нибудь клетки верхней строки до любой клетки нижней, набрав при проходе по клеткам минимальный штраф.

Формат входных данных

Первая строка входного файла содержит в себе пару чисел $2 \leq n \leq 10002 \leq m \leq 1000$, затем следует n строк из m целых чисел.

Формат результата

Необходимо вывести в выходной файл на первой строке минимальный штраф, а на второй — последовательность координат из n ячеек, через которые пролегает маршрут с минимальным штрафом.

Метод решения

Для решения данного задания воспользуемся методом динамического программирования, и разобьём задачу на более простые задачи. Так требуется найти кратчайшие через поле размерности n*m при этом на каждом шаге нужно спускаться на одну строку ниже. Всего таких строк N и размер каждой – M. Наивный алгоритм решения данной задачи – полный перебор всех возможных путей. Таким образом на каждом шаге выбираем 1 из 3 (2-ух если находимся на краю поля) переходов, повторяем это действие, запоминаем все длинны всех путей, выбираем из получившихся путей путь с самым маленьким штрафом, восстанавливаем его и получаем результат. Асимптотика такого решения $O(m * 3^n)$.

Теперь используя принципы динамического программирования, разобьём задачу на несколько подзадач. Заданное поле представляет из себя набор из n слоев, где на каждом из них мы должны выбирать путь с минимальным штрафом. И тогда для решения такой подзадачи требуется, чтобы минимальный штраф на более низком уровне уже был найден – получаем индукцию: мы знаем как найти минимальный путь в текущем слое (смотрим на след. слой, выбираем ячейку с минимальным штрафом и прибавляем к ней штраф текущей ячейки) и штрафы для последнего слоя – штрафы самой последней строки исходного массива. После прохода по матрице в верхней строчке выбираем наименшее значение и восстанавливаем путь.

Выполняем n подзадач каждая из которых выполняется за $O(m)$ – итоговое время работы – $O(n*m)$ Память, расходуемая при решении также равна $O(n*m)$ так как хранится сама матрица штрафов.

Исходный код

Я реализовал структуру клетки, где value - значение данной клетки, j - индекс, необходимый для восстановления пути и вывода ответа. Так же перегрузил операторыы сравнения и оператор сложения.

```
struct TCell{
    long long value;
    long long j;
    TCell() : value(-1), j(-1){}
    TCell(long long _value, long long _j) : value(_value), j(_j) {}
    friend bool operator< (const TCell& a, const TCell& b);
    friend bool operator> (const TCell& a, const TCell& b);
    friend bool operator<= (const TCell& a, const TCell& b);
    friend bool operator>= (const TCell& a, const TCell& b);
    friend bool operator== (const TCell& a, const TCell& b);
    friend TCell operator+ (const TCell& a, const long long b);
};
```

В main() мы идём снизу вверх, просчитывая штрафы для каждой клетки на данном слое, пользуясь только информацией с прошлого слоя. Затем мы идём по последнему слою, находя минимальный элемент и восстанавливаем путь.

```cpp
int main() {
    std::ios::sync_with_stdio(false);

    long long n, m;
    std::cin >> n >> m;

    std::vector<std::vector<long long>> field;
    std::vector<std::vector<TCell>> myField;
    field.resize(n);
    myField.resize(n);
    for (int i = 0; i < n; ++i) {
        field[i].resize(m);
        myField[i].resize(m);
    }
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            std::cin >> field[i][j];
        }
    }
    for (int i = 0; i < m; ++i) {
        myField[myField.size() - 1][i] = TCell(field[field.size()-1][i], -1);
    }
    for (int i = n - 2; i >= 0; --i) {
        for (int j = 0; j < m; ++j) {
            if (j == 0) {
                myField[i][j] = MyMin(TCell(myField[i + 1][j].value, j), TCell(myField[i + 1][j + 1].value,j+1)) + field[i][j];
            }
            else if (j == m - 1) {
                myField[i][j] = MyMin(TCell(myField[i + 1][j].value, j), TCell(myField[i + 1][j - 1].value, j-1)) + field[i][j];
            }
            else {
                myField[i][j] = MyMin(TCell(myField[i + 1][j - 1].value,j-1), MyMin(TCell(myField[i + 1][j].value, j), TCell(myField[i + 1][j + 1].value, j+1))) + field[i][j];
            }
        }
    }
    TCell min(myField[0][0]);
    long long minJ = 0;
```

```
    for (int i = 1; i < m; ++i) {
        if (min > myField[0][i]) {
            min.value = myField[0][i].value;
            min.j = myField[0][i].j;
            minJ = i;
        }
    }
    std::cout << min.value << std::endl;
    std::cout << '(' << 1 << ',' << minJ + 1 << ") ";
    for (int i = 1; i < n-1; ++i) {
        std::cout << '(' << i + 1 << ',' << min.j + 1 << ") ";
        min.j = myField[i][min.j].j;
    }
    std::cout << '(' << n << ',' << min.j + 1 << ")" << std::endl;
    return 0;
}
```

Пример работы

```
mmaxim2710@DESKTOP-RDPBU3D:/mnt/c/Users/mmaxi/Desktop/coursera/DA_ex/lab7$ make clean
rm -f *.o solution
mmaxim2710@DESKTOP-RDPBU3D:/mnt/c/Users/mmaxi/Desktop/coursera/DA_ex/lab7$ make
g++ -std=c++17 -pedantic -Wall -O2 -c main.cpp -o main.o
g++ -std=c++17 -pedantic -Wall -O2 main.o -o solution
mmaxim2710@DESKTOP-RDPBU3D:/mnt/c/Users/mmaxi/Desktop/coursera/DA_ex/lab7$ ./solution
3 3
3 1 2
7 4 5
8 6 3
8
(1,2) (2,2) (3,3)
```

Вывод

Динамическое программирование — это когда у нас есть задача, которую непонятно как решать, и мы разбиваем ее на меньшие задачи, которые тоже непонятно как решать. (c) А.Кумок

Динамическое программирование стоит применять для решения задач, которые обладают двумя характеристиками:

1. Можно составить оптимальное решение задачи из оптимального решения ее подзадач.

2. Рекурсивный подход к решению проблемы предполагал бы многократное (не однократное) решение одной и той же подпроблемы, вместо того, чтобы производить в каждом рекурсивном цикле все новые и уникальные подпроблемы.

Так, в решаемой мной задаче, вместо того, чтобы перебирать каждый путь за $O(m*3^n)$, где n и m - количество строк и столбцов матрицы соотвествтенно, мы можем разбить задачу на подзадачи: нахождение пути с минимальным штрафом на каждом из слоёв. Решив эти подзадачи оптимально, мы оптимально решаем и главную задачу. В моём случае сложностью $O(n * m)$.

Так же Динамическое программирование часто сравнивают с принципом «Разделяй и влавствуй». Я бы не считал их чем-то совершенно различным, потому что обе эти концепции рекурсивно разбивают проблему на две или более подпроблемы одного и того-де типа до тех пор, пока эти подпроблемы не станут достаточно легкими. Но у этих подходов есть как пересекающиеся задачи, так и не пересекающиеся (например, бинарный поиск нельзя реализовать с помощью ДП).