Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №7 по курсу «Дискретный анализ»

Студент: В. П. Будникова
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б-19

Дата:
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №7

Задача: 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.

Вариант алгоритма: Задана матрица натуральных чисел A размерности n x 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 Ai,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 $2 \leq n \leq 1000$ и $2 \leq m \leq 1000$, 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.

1 Описание

Как сказано в [1]: «Динамическое программирование — способ решения сложных задач путём разбиения их на более простые подзадачи. Он применим к задачам с оптимальной подструктурой, выглядящим как набор перекрывающихся подзадач, сложность которых чуть меньше исходной.»
Данную мне задачу можно разделить на подзадачи, после решения которых можно прийти к ответу. Задана матрица с числами, нужна найти путь из верхней строки в нижнюю, идя по которому набирается самый минимальный штраф, величина штрафа - это числовое значение ячейки. То есть нам нужно найти такой путь, что сумма ячеек будет минимальна. Так как в ответе нужно указать и сам этот минимальный путь(координаты ячеек), то необходимо хранить всю матрицу до конца решения, чтобы потом можно было восстановить последовательность ячеек.
Решение:
Так как в условии задания сказано, что переходить можно только в одну из трех соседних ячеек снизу, то в текущую ячейку мы могли прийти только из одной из трех соседних верхних ячеек. Мы будем идти по элементам строчки матрицы, начиная со второй строчки и заканчивая последней. К значению каждого элемента мы прибавляем значение X, которое является минимальным из значений трех соседних ячеек, стоящих выше, также необходимо запоминать координату ячейки, значение которой мы прибавляем, для восстановления ответа. Таким образом на каждой строчке, в каждой ячейке будет находится минимальная сумма значений ячеек, по которым мы можем добраться до этой ячейки.
В последней строчке матрицы будут находится минимальные пути, до каждой ячейки этой строчки. Поэтому необходимо выбрать минимальное значение - это и будет наш минимальный путь. Так как в ответе необходимо указать путь, то нужно посмотреть, какую координату мы присволии нашей минимальной ячейке, перейти в нее, посмотреть на ее присвоенную координату, перейти в нее и так далее, пока мы не дойдем до первой строчки. Таким образом мы получить последовательность координат ячеек, по которым можно пройти от верхней строчки матрицы до нижней, набрав при этом минимальный штраф.
Так как при решении задачи мы проходимся по каждому элементу ровно один раз, то сложность решения будет - O(n x m). Так как мы храним всю матрицу, то пространственная сложность будет - O(n x m).

2 Исходный код

```
#include <iostream>
#include <vector>
std::pair<long long, short> * Comp(std::pair<long long, short> * v1, std::pair<long
    long, short> * v2) {
    return v1->first <= v2->first ? v1 : v2;
}
std::pair<long long, short> * Min(std::vector<std::pair<long long, short>> &v) {
    std::pair<long long, short> * min = &(*v.begin());
    for (std::vector<std::pair<long long, short>>::iterator i = v.begin(); i != v.end()
        ; ++i) {
        if ((*i).first < (*min).first) min = &(*i);
    }
    return min;
}
int main() {
    short n, m;
    std::cin >> n >> m;
    std::vector<std::vector<std::pair<long long,short>>> vec(n);
    std::vector<std::pair<long long, short>> str(m);
    std::pair<long long, short> * elem;
    std::vector<long long> res(n);
    for (short i = 0; i < n; ++i) {
        for (short j = 0; j < m; ++j) {
            std::cin >> str[j].first;
        }
        vec[i] = str;
    }
    for (short i = 1; i < n; ++i) {
        for (short j = 0; j < m; ++j) {
            elem = Comp(&vec[i - 1][std::max(0, j - 1)], &vec[i - 1][j]);
            elem = Comp(elem, &vec[i - 1][std::min(j + 1, m - 1)]);
            vec[i][j].second = elem - &vec[i - 1][0];
            vec[i][j].first = elem->first + vec[i][j].first;
        }
    }
    elem = Min(vec[n - 1]);
    res[n - 1] = elem - &vec[n - 1][0]; res[n - 2] = elem->second; --n;
    while (n != 1) {
        elem->second = vec[n - 1][elem->second].second; --n;
        res[n - 1] = elem->second;
    }
    std::cout << elem->first << std::endl;
    for (short i = 0; i < res.size(); ++i) {
        std::cout << '(' << i + 1 << "," << res[i] + 1 << ')' << " ";
    }
    std::cout << std::endl;
}
```

3 Консоль

```
Lera:B-M valeriabudnikova$ cat test.txt
10 10
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9
Lera:B-M valeriabudnikova$ ./lab7 <test.txt
45
(1,1) (2,1) (3,1) (4,1) (5,1) (6,1) (7,1) (8,1) (9,1) (10,1)
```

4 Тест производительности

Сравним время работы алгоритма с "наивным"алгоритмом, который для каждой ячейки, находящейся в первый строчке рекурсивно высчитываем минимальный путь по каждой из трех соседних ячеек.
Время работы для матрицы. 10x10:

```
Lera:l7 valeriabudnikova$ make run
./lab7 <test.txt
time_my_lab: 1e-06 s
./time <test.txt
time_naive: 0.001899 s
```

Время работы для матрицы. 15x15:

```
Lera:l7 valeriabudnikova$ make run
./lab7 <test.txt
time_my_lab: 2e-06 s
./time <test.txt
time_naive: 0.284973 s
```

Как можно увидеть, мой алгоритм выиграл у наивного алгоритма, так как сложность у наивного алгоритма намного больше, из-за рекурсивного подсчета минимального пути для каждого элемента матрицы.

5 Выводы

Выполнив седьмую лабораторную работу по курсу «Дискретный анализ», я изучила познакомилась с алгоритмами динамического программирования. Реализовала алгоритм для решения моей задачи. Также сравнив время работы на больших входных данным "наивного"алгоритма, который решает задачу путем рекурсии и перебора и алгоритма динамического программирования, я поняла, на сколько быстрее работает данный алгоритм, так как при времени работы реализованного алгоритма в несколько секунд, "наивный"работал почти за 10 и более секунд.

Список литературы

[1] *Динамическое программирование.*
URL: `https://ru.wikipedia.org/wiki/Динамическое_программирование` (дата обращения: 21.04.2020).