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:	П. А. Харьков
Преподаватель:	А. А. Кухтичев
Группа:	М8О-206Б-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.

Вариант задания: Задана матрица натуральных чисел **A** размерности $\mathbf{n} \times \mathbf{m}$. 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 $\mathbf{2 \leqslant n \leqslant 1000}$ и $\mathbf{2 \leqslant m \leqslant 1000}$, 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

Для того, чтобы решить задачу давайте сначала используем самый наивный алгоритм, то есть для каждой последней клетки будем искать минимум с помощью рекурсии. Но в этом сложности сложность алгоритма будет равна $O(m * 3^n)$, и уже при матрице 5 на 5 придется делать более тысячи элементарных операций сравнения.

Эту задачу можно решить с помощью метода динамического программирования, идея которого заключается в разделении задачи на более маленькие и решении их, а затем комбинировании ответов. В отличие от рекурсивного метода, если в какой-то момент подзадача была решена, второй раз её не нужно решать, а просто брать результат решения.

Для того, чтобы найти минимум в (i,j) позиции, нам необходимо найти минимум из (i-1, j-1), (i-1, j) или (i-1, j+1) позиций. И так как нам может пригодится это значение, то мы его сохраним, а затем будем использовать в последующих поисках минимума. Благодаря этому сложность моего алгоритма составляет $O(n * m)$ - это минимальная сложность, так как нам, как минимум, нужно пройти по всей матрице.

2 Исходный код

Для начала я считываю n и m - количество строк и стобцов в матрице. Затем создаю матрицу $matrix$, которая является вектором векторов, и записываю в неё штрафы за проход по клетке.
Затем, начиная со 2 строки и заканчивая на последней строке, для каждого ее элемента ищем минимум из предыдущих клеток и добавляем к текущему. Так сложность подсчета минимального штрафа для каждой клетки высчитывается за $O(n * m)$.
После этого нам необходимо найти самый выгодный путь. В последней строке находим столбец с минимальным штрафом и затем до первой строки ищем столбцы на строке выше с минимальным штрафом. Те столбцы, на которые мы перешли, и являются клетками нашего пути. Сложность поиска пути является $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;
using TLl = long long;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    int n, m;
    cin >> n >> m;
    vector<vector<TLl>> matrix(n, vector<TLl>(m));

    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            cin >> matrix[i][j];
        }
    }

    for(int i = 1; i < n; i++){
        for (int j = 0; j < m; j++) {
            if (j == 0){
                matrix[i][j] += min(matrix[i-1][j], matrix[i-1][j+1]);
            }else if (j == m - 1){
                matrix[i][j] += min(matrix[i-1][j-1], matrix[i-1][j]);
            }else{
                matrix[i][j] += min({matrix[i-1][j-1], matrix[i-1][j], matrix[i-1][j
                    +1]});
            }
        }
    }
```

```
    TLl minval = matrix[n-1][0];
    int mincol = 0;

    for(int j = 1; j < m; j++){
        if(minval > matrix[n-1][j]){
            minval = matrix[n-1][j];
            mincol = j;
        }
    }

    vector<int> path(n);
    for(int i = n-1; i >= 0; i--){
        if (mincol == 0) {
            if(matrix[i][mincol+1] < matrix[i][mincol]){
                mincol++;
            }
        }else if (mincol == m - 1) {
             if(matrix[i][mincol-1] < matrix[i][mincol]){
                mincol--;
            }
        }else {
            TLl colval = min({matrix[i][mincol-1], matrix[i][mincol], matrix[i][mincol+1]});
            if (colval == matrix[i][mincol-1]){
                mincol--;
            }else if(colval == matrix[i][mincol+1]){
                mincol++;
            }
        }
        path[i] = mincol;
    }

    cout << minval << '\n';
    for(int i = 0; i < n-1; i++){
        cout << '(' << i+1 << ',' << path[i]+1 << ") ";
    }
    cout << '(' << n << ',' << path[n-1]+1 << ")\n";

}
```

3 Консоль

```
p.kharkov$ make
g++ -std=c++14 -pedantic -Wall src/laba7.cpp -o laba7
p.kharkov$ cat tests/main
3 3
3 1 2
7 4 5
8 6 3
p.kharkov$ ./laba7 <tests/main
8
(1,2) (2,2) (3,3)
```

4 Тест производительности

Тест производительности для наивного алгоритма решения задачи состоит из матрицы 15 на 15 чисел, в каждой клетке находится число от 1 до 100:

```
p.kharkov$ ./benchmark <tests/small.t
Naive solution time: 242ms
Dynamic solution time: 0ms
```

Как можно увидеть, наивный алгоритм отработал за 242 миллисекунды, а мой алгоритм отработал за 0 миллисекунд.
Также я протестировал мой алгоритм в 2 тестах: первый состоит из 10^5 на 10^4 чисел, а второй из 10^5 на 10^5 чисел.

```
p.kharkov$ ./benchmark <tests/01.t
Dynamic solution time: 24ms
p.kharkov$ ./benchmark <tests/02.t
Dynamic solution time: 238ms
```

Как можно увидеть, при увеличении количества чисел в 10 раз, время решения увеличивается в 10 раз. Также, как можно увидеть, только при 10^5 на 10^5 чисел время выполнения программы равно времени выполнения в наивном алгоритме.

5 Выводы

Выполнив седьмую лабораторную работу по курсу «Дискретный анализ», я научился использовать метод динамического программирования для решения задач. Этот метод разбиения задачи на подзадачи с сохранением их результатов для последующего использования может значительно сократить время решение задачи. Это же и мы увидели в тесте производительности - рекурсивное решение значительно дольше выполнялось, чем мое решение.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))

[2] *Динамическое программирование — ИТМО.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Динамическое_программирование`
(дата обращения: 29.04.2021).