Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа № 7 по курсу дискретного анализа: Динамическое программирование

Студент:	А. О. Тояков
Преподаватель:	А. Н. Ридли
Группа:	М8О-307Б-18
Дата:	
Оценка:	
Подпись:	

Москва
2021

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.

Задана матрица натуральных чисел A размерности $n \times m$. 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.

Метод решения

Для реализации поставленной задачи мне понадобится дополнительная структура, куда я буду записывать промежуточные данные.

1. Создание структуры для хранения промежуточных данных.
2. Проходя по каждому элементу, буду определять минимальное значение штрафа для данного элемента, исходя из предыдущих вычислений.
3. Ещё одним проходом выведу в консоль оптимальное решение.

Таким способом с помощью этого алгоритма я буду считать оптимальный результат для ячеек массива, исходя из результатов подсчёта предыдущих ячеек. Я буду считать сумму, идя от конца данного массива к его началу: данное решение значительно упростит вывод оптимального решения. Идя сверху вниз к текущему элементу B[i, j] - A[i, j], я прибавляю минимальное значение штрафа, выбирая из трёх уже подсчитанных значений min(B[i + 1, j - 1], B[i + 1, j], B[i + 1, j + 1]). Сложность подсчёта O(n * n), а сложность вывода оптимального решения O(2n).

Описание программы

Для начала я инициализировал 5 переменных типа int, а также создал две матрицы, которые содержат элементы типа long long. Матрица A служит для хранения исходной матрицы, а в матрице B записывается результат вычислений. В данной программе используется 20 байт под статические переменные, а также выделяется 8 * 2 * n байт, где n - размер квадратной матрицы. Также реализована функция min, которая будет вычислять минимальное значение из трёх, поданных на вход. Для элементов, стоящих в начале или в конце строки массива минимум считается для двух элементов. Начальные значения из матрицы B копируются из матрицы A. Оптимальное решение соответствует спуску по матрице B, проходя минимальные значения.

Исходный код

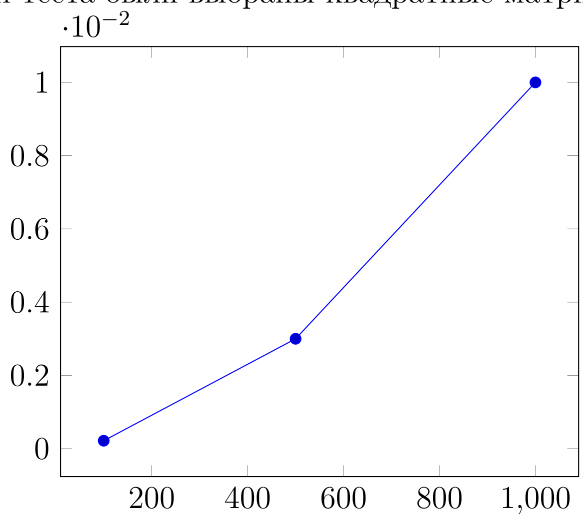
```
#include <iostream>
#include <string>
#include <iomanip>
#include <sstream>
#include <stdlib.h>
#include <vector>

// ошибки на чекере из-за перевыполнения
long long Minimum(long long first, long long second, long long third) {
    if(first <= second && first <= third) {
        return first;
    } else if(first >= second && second <= third) {
        return second;
    } else if(first >= third && third <= second) {
        return third;
    }
    return 0;
}


int main() {
    // initialization
    long long **A;
    long long **B;
    int n,m;
    std::cin >> n >> m;

    int i = 0, j = 0, k = 0;
    A = new long long*[n];
    for(i = 0; i < n; i++) {
        A[i] = new long long[m];
    }
    for(i = 0; i < n; i++) {
        for(j = 0; j < m;j++) {
            std::cin >> A[i][j];
        }
    }
    B = new long long*[n];
    for(i = 0; i < n; i++) {
        B[i] = new long long[m];
    }
```

```
for(i = 0; i < n; i++) {
    for(j = 0; j < m;j++) {
        B[i][j] = 0;
    }
}
// initialization
// algorithm
for(j = 0; j < m; j++) {
    B[n - 1][j] = A[n - 1][j];
}
for(i = n - 2; i >= 0; i--) {
    B[i][0] = (Minimum(B[i + 1][0], B[i + 1][0], B[i + 1][1]) + A[i][0]);
    for(j = 1; j < m - 1; j++) {
        B[i][j] = (Minimum(B[i + 1][j - 1], B[i + 1][j], B[i + 1][j + 1]) + A[i][j])
    }
    B[i][m - 1] = (Minimum(B[i + 1][m - 2], B[i + 1][m - 1], B[i + 1][m - 1]) + A[i]
}
// algorithm
// output
for(j = 1; j < m; j++) {
    if(B[0][k] > B[0][j]) {
        k = j;
    }
}
std::cout << B[0][k] << "\n " << "(1," <<  k + 1 << ") ";
for(i = 1; i < n; i++) {
    if((k > 0) && (B[i][k - 1] < B[i][k])) {
        if((k + 1 < m) && (B[i][k - 1] > B[i][k + 1])) {
            k++;
        } else {
            k--;
        }
    } else {
        if((k + 1 < m) && (B[i][k + 1] < B[i][k])) {
            k++;
        }
    }
    std::cout << "("<< i + 1 << "," <<  k + 1 << ") ";
}
std::cout << "\n";
// output
// clear
```

```
    for(int i = 0; i < n; i++) {
        delete [] A[i];
        delete [] B[i];
    }
    delete [] A;
    delete [] B;
    // clear
    return 0;
}
```

Тест производительности

Тесты представляют из матрицы, заполненные рандомными числами не больше 100000000. Для теста были выбраны квадратные матрицы, порядок которых не превышает 1000.



Пояснения к графику: Ось y - время в секундах. Ось x - количество строк.

Выводы

Для некоторых задач метод рекурсивного разбиения основной задачи на подзадачи является довольно эффективным и полезным. Используя динамическое программирование, можно заметно облегчить процесс решения и уменьшить ассимпотическую сложность. Данная задача была реализована со сложностью O(n * n + 2n). Думаю, что эту же задачу можно решить, не используя методы динамического программирования, а просто, например, перебором, но в этом случае это точно будет менее эффективно и реализация будет довольно сложная.