Лабораторная работа № 7 по курсу дискретного анализа: Динамическое программирование

Выполнил студент группы М8О-308Б-21 МАИ *Белоносов Кирилл.*

Условие

1. Общая постановка задачи: Задан прямоугольник с высотой n и шириной m, состоящий из нулей и единиц. Найдите в нём прямоугольник наибольшой площади, состоящий из одних нулей.

2. Вариант задания(2):

   Формат ввода: В первой строке заданы $1 \leq n \leq 500$ и $1 \leq m \leq 500$. В последующих n строках записаны по m символов 0 или 1 - элементы прямоугольника.
   Формат вывода: Необходимо вывести одно число – максимальную площадь прямоугольника из одних нулей.

Метод решения

Для решения данной задачи используем метод динамического программирования. Заметим, что если мы будем хранить в нашей динамике расстояние до ближайше 1 сверху, мы легко сможем получить высоту нашей матрицы в указанной точке. Далее необходимо получить ширину нашей матрицы и мы точно найдем ее максимальную площадь. Для этого используем две дополнительных динамики: максимальное продолжение влево и максимальное продолжение вправо. Для этого используем стек, будем туда добавлять "граничные элементы". Если стек пуст, значит для текущего элемента нет границы и запишем -1. Если же стек не пуст (Нашлись такие индексы, что их высота до 1 больше текущей) то запишем верхушку стека как границу. Такой динамикой мы саможем находить элементы за левой (правой) границей и разница их и будет нашей шириной. Сложность алгоритма O(mn)

Описание программы

Программа почти полностью соответствует описанному выше алгоритму, за исключением того, что мы можем не хранить всю динамику, а только для конкретной строки

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
```

```
    int n, m;
    cin >> n >> m;
    vector <vector<int>> a(n, vector<int>(m));
    for (int i = 0; i < n; ++i) {
        string str;
        cin >> str;
        for(int j = 0; j < m; ++j)
            a[i][j] = str[j] - '0';
    }
    vector<int> dp(m, -1), dp_left(m), dp_right(m);
    int square = 0;
    for(int i = 0; i < n; ++i) {
        for(int j = 0; j < m; ++j)
            if(a[i][j] == 1)
                dp[j] = i;
        stack<int> st1, st2;
        for(int j = 0; j < m; ++j) {
            while(!st1.empty() && dp[st1.top()] <= dp[j])
                st1.pop();
            while(!st2.empty() && dp[st2.top()] <= dp[m - j - 1])
                st2.pop();
            dp_left[j] = (st1.empty()) ? -1 : st1.top();
            dp_right[m - j - 1] = (st2.empty()) ? m : st2.top();
            st1.push(j);
            st2.push(m - j - 1);
        }
        for(int j = 0; j < m; ++j) {
            square = max(square, (i - dp[j]) * (dp_right[j] -
               dp_left[j] - 1));
        }
    }
    cout << square << "\n";
    return 0;
}
```

Дневник отладки

Попытка 1

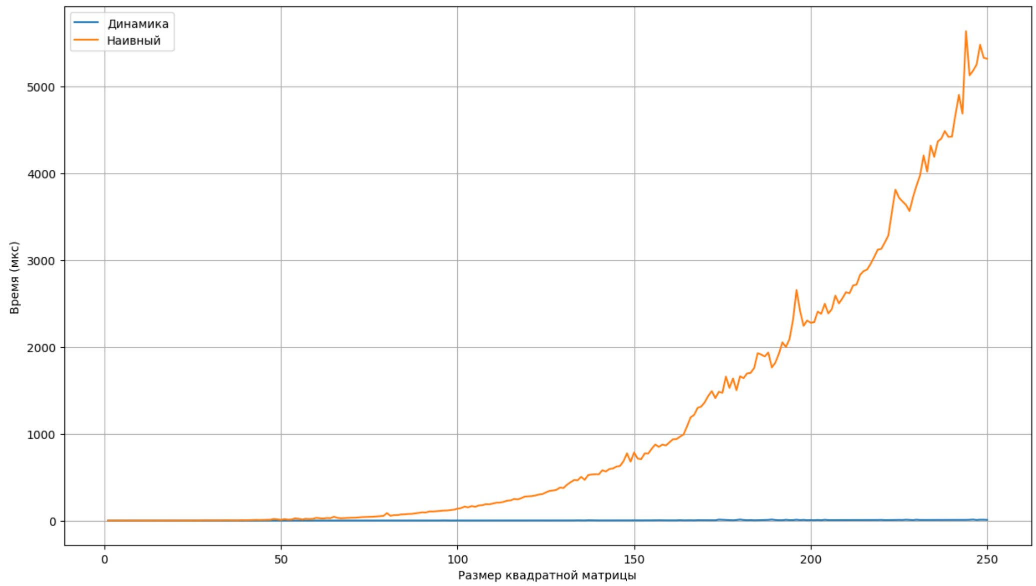
Данная задача была решена с первой попытки. Скорее всего это связано с тем, что я заранее покрыл ее тестами

0.1 Результаты тестирования

Тест производительности

Тестирование программы производилось на 250 сгенерированных тестах. Каждый тест представлял собой квадртаную матрицу размером i * i. Для сравнения использовался наивный алгоритма за $O(n^2 * m^2)$

Вставка:



Получившиеся результаты особо не удивляют. Реализация с использованием динамического программирования имеет намного более лучшую ассимптотику.

Выводы

В данной лабораторной работе я познакомился с принципом решения задач, с помощью динамического программирования. Его основная идея заключается в том, чтобы разбить текущую задачу на элементарные подзадачи, которые мы уже умеем решать. В данном задаче подзадачи заключались в том, чтобы для каждого элемента находить максмальную высоту и максимальную ширину нулевой матрицы. Метод динамического программирования позволяет решать большой класс задач за сложность, намного меньшую, чем наивную (которая чаще всего сводится к перебору). В данном случае динамика позволила перейти от сложности $O(n^2 * m^2)$ к сложности O(nm), что по результатам тестирования значительно ускорило программу. Конечно, выделение зависимости, которая приводит к ДП является одной из сложнейших задач, но меньшая ассимпотика стоит того.

Помими представленного алгоритма присутствует целый ряд других, такие как:

Наибольшая общая подпоследовательность, Задача о рюкзаке, Расстояние Левенштейна