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:	М. А. Волков
Преподаватель:	А. А. Кухтичев
Группа:	М8О-207Б-19
Дата:	22 апреля 2021 г.
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. Формат входных и выходных данных описан в варианте задания.

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

Формат входных данных:
В первой строке заданы $1 \leq n \leq 500$ и $1 \leq m \leq 500$. В следующих n строках записаны по m символов 0 или 1 – элементы прямоугольника.

Формат результата:
Необходимо вывести одно число – максимальную площадь прямоугольника из одних нулей.

1 Описание

Как описано в [1], динамическое программирование — это метод решения задач, при котором сложная задача разбивается на более простые, решение сложной задачи составляется из решений простых задач.

Этот метод очень похож на «разделяй и властвуй», но динамическое программирование допускает использование метода восходящего анализа, который позволяет изначально решать простые задачи и получать на их базе решение более сложных.

Так же метод запоминает решения подзадач, потому что часто для построения нужно обращаться за оптимальным решением к одним и тем же малым задачам.

В моем случае малыми задачами – являются подсчет площади прямоугольника для каждой строчки, запоминая при этом возможные основания прямоугольника и их высоты.

Высоты прямоугольника считаются очень просто – это количество нулей над нулем, включая его.

Возможные основания прямоугольника считаются несколько сложнее. Предположим нам нужно высчитать $S_i(L, R)$ при $1 \leq L \leq R \leq m$. Для этого нам нужно как-то высчитать L и R. Чтобы сделать это быстро, мы будем проходиться по нашей высчитанной высоте и таким образом записывать в L и R самые левые и правые совпадающие по высоте значения соответственно.

2 Исходный код

Для подсчета высот, я буду высчитывать для каждой строчки массив p, означающий высоту, и массивы L и R, отвечающие за тот самый индекс, описанный в описании для каждого соответствующего j-ого элемента строки i.

Для того, чтобы высчитать это все за всего 2 прохода по массиву p, используется стек, в который записывается элемент, больше $p[j]$.

Таким образом я добиваюсь ассимптотики $O(n * m)$.

```
#include <iostream>
#include <stack>
#include <vector>

int Atoi(const char& c){
  return c - '0';
}

int main(){
  int n,m;
  std::cin >> n >> m;
  std::vector<int> p(m,0);
  std::vector<std::vector<int>> A(n,std::vector<int>(m,0));
  int sMax = 0;

  //input
  for (int i = 0; i < n; ++i) {
    std::string input;
    std::cin >> input;
    for (int j = 0; j < m; ++j) {
      int digit = Atoi(input[j]);
      A[i][j] = digit;
    }
  }

  for (int i = 0; i < n; ++i) {
    //calculating p fot i's row
    for (int j = 0; j < m; ++j) {
      p[j] = (p[j] + 1) * !A[i][j];
    }

    //calculating R fot i's row
    std::vector<int> R(m,0);
    {
      std::stack<int> S;
      for (int j = 0; j < m; ++j) {
        while(!S.empty() && p[j] < p[S.top()]){
          R[S.top()] = j - 1;
          S.pop();
```

```
        }

        S.push(j);
      }
      while(!S.empty()){
        R[S.top()] = m-1;
        S.pop();
      }
    }

    //calculating L fot i's row
    std::vector<int> L(m,0);
    {
      std::stack<int> S;
      for (int j = m-1; j >= 0; --j) {
        while(!S.empty() && p[j] < p[S.top()]){
          L[S.top()] = j + 1;
          S.pop();
        }

        S.push(j);
      }
      while(!S.empty()){
        L[S.top()] = 0;
        S.pop();
      }
    }

    int s_iMax = 0;
    for (int j = 0; j < m; ++j) {
      s_iMax = std::max(s_iMax, p[j]*(R[j] - L[j] + 1));
    }

    sMax = std::max(sMax, s_iMax);
  }

  std::cout << sMax << std::endl;

  return 0;
}
```

3 Тест производительности

Сравниваться будут 2 алгоритма: наивный, который работает за $O(n^3 * m^3)$ и мой алгоритм.
Тесты состоят из прямоугольников $10 * 10$, $100 * 100$ и $500 * 500$.

```
test1 10*10 reqtangle
Naive time 0.0 ms
My prog time 0.0 ms

test2 100*100 reqtangle
Naive time 0.418 ms
My prog time 0.2 ms

test3 250*250 reqtangle
Naive time 16.32 ms
My prog time 0.12 ms
```

Видно, что при малых значениях алгоритмы сильно не отличаются по времени, но чем дальше, тем больше проявляет себя алгоритм за экспоненциальное время.

4 Выводы

В ходе выполнения лабораторной работы я изучил классические задачи динамического программирования и их методы решения, реализовал алгоритм для своего варианта задания.

Рассмотрев свой вариант из предущей лабораторной работы (ЛР 5), я понял, что мою задачу можно было бы решить методом Динамического Программирования, но ассимптотически она бы выполнялась медленнее и затрачивала бы на много больше памяти.

Это меня натолкнуло на мысль, что ДП – не является панацеей для всех возможных задач, но обычно данным методом программирования код пишется быстрее, что может мне пригодится на олимпиадах или на работе, где важна не опитимизация кода, а его быстрое написание.

Список литературы

[1] *Динамическое программирование — Викиконспекты*
URL: https://neerc.ifmo.ru/wiki/index.php?title=Динамическое_программирование
(дата обращения: 02.04.2021).