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: С. М. Бокоч
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2018

Лабораторная работа №7

Задача: При помощи метода динамического программирования разработать алгоритм решения задачи, определяемой своим вариантом; оценить время выполнения алгоритма и объём 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: 2. Пустой прямоугольник.

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

Входные данные: в первой строке заданы $1 \leq n, m \leq 500$. В следующих n строках записаны по m символов 0 или 1 - элементы прямоугольника.

Выходные данные: необходимо вывести одно число – максимальную площадь прямоугольника из одних нулей.

1 Метод решения

Для выбора оптимального решения при выполнении задач программирования иногда требуется перебирать большое количество комбинаций данных, что сильно нагружает ОЗУ. К таким методам относится, например, метод программирования «разделяй и властвуй». В данном случае алгоритмом предусмотрено разделение задачи на отдельные мелкие подзадачи. Такой метод применяется только в тех случаях, когда мелкие подзадачи независимы между собой. Для того чтобы избежать выполнения лишней работы в том случае, если подзадачи взаимозависимы, используется метод динамического программирования.

Динамическое программирование заключается в определении оптимального решения n-мерной задачи, разделяя её на **n** отдельных этапов. Каждый из них является подзадачей по отношению к одной переменной.

1 Тривиальный алгоритм.

Подразумевает полный перебор матрицы для каждого элемента. В худшем случае алгоритм справляется с задачей за $\Theta(n^2 * m^2)$.

Ниже приведен листинг алгоритма.

```
1  std::size_t TMatrix::FindMaxSubMatrixTrivial(const std::size_t type) {
2      int ans = 0;
3      for (std::size_t i = 0; i < n; i++) {
4          for (std::size_t j = 0; j < m; j++) {
5              std::size_t right = m;
6              std::size_t down = n;
7              if (mData[i][j] == type) {
8                  std::size_t x = i;
9                  std::size_t y = j;
10                 while (x < down && mData[x][y] == type) {
11                     while (y < right && mData[x][y] == type) {
12                         y++;
13                     }
14                     right = y;
15                     x++;
16                     ans = std::max(ans, static_cast<const int &>((x - i) * (y - j)));
17                     y = j;
18                 }
19             }
20         }
21     }
22     return ans;
23 }
```

Стоит заметить, что если в матрице нули и единицы находятся в приблизительно равном соотношении, то тривиальный алгоритм работает быстрее даже на больших входных данных.

2 Метод динамического программирования

Задача о нахождении наибольшей нулевой матрицы являтся одной из классических задач динамического программирования.

Искомая нулевая подматрица ограничена со всех четырёх сторон границами поля (единицами). Следовательно пропустить возножные решения не получится.

Для реализации требуется завести одномерный массив dp, в котором для каждого **j** столбца будет хранится информация о наибольшем номере строки **i**, если в позиции $matrix[i][j]$ содержится единица, иначе -1. Значение -1 вскоре потребуется для выведения формулы. Пользуясь значением $dp[i]$, мы сразу получаем номер верхней строки нулевой подматрицы. Осталось теперь определить оптимальные левую и правую границы нулевой подматрицы, это значит максимально раздвинуть эту подматрицу влево и вправо от j-го столбца.

Для расширения левой границы необходимо найти индекс такого столбца **x**, для которого будет выполняться условие $dp[j] < dp[x]$. Логично, что x – ближайший такой столбец слева для j. Если индекса x, удовлетворяющему условию нет в матрице, то смело предполагаем, что дальше матрицу расширить нельзя.

Правая граница подматрицы определяется аналогично – ближайший справа от j индекс такой, что $dp[j] < dp[y]$, иначе m (дальше расширить матрицу нельзя).

Площадь нулевой подматрицы подсчитывается с помощью формулы: $(i - dp[j]) \cdot (y - x - 1)$

Для линейной сложности вычисления индексов следует использовать структуру stack.

Вот сам собственно листинг.

```
1  std::size_t TMatrix::FindMaxSubMatrixFast(const std::size_t type) {
2      int ans = 0;
3      std::vector<int> dp (m, INFINITY), indexLeft(m), indexRight(m);
4      std::stack<int> stack;
5      for (int i = 0; i < n; i++) {
6
7          for (int j = 0; j < m; j++) {
8              if (mData[i][j] != type) {
9                  dp[j] = i;
10             }
11         }
12
13         ClearStack(stack);
14         for (int j = 0; j < m; j++) {
```

```
15            while (!stack.empty() && dp[ stack.top() ] <= dp[j]) {
16                stack.pop();
17            }
18            indexLeft[j] = stack.empty() ? INFINITY : stack.top();
19            stack.push(j);
20        }
21
22        ClearStack(stack);
23        for (int j = m - 1; j >= 0; j--) {
24            while (!stack.empty() && dp[stack.top()] <= dp[j]) {
25                stack.pop();
26            }
27            indexRight[j] = stack.empty() ? m : stack.top();
28            stack.push(j);
29        }
30
31        for (int j = 0; j < m; j++)
32            ans = std::max(ans, (i - dp[j])*(indexRight[j] - indexLeft[j] - 1));
33    }
34    return ans;
35 }
```

Информация взята с сайта *e-maxx.ru*.

2 Код дополнительных файлов

1. checker.py – генератор тестов на языке Python

```
from random import choice
import sys
import random

MAX_SIZE_MATRIX = 5000
MIN_SIZE_MATRIX = 1
COUNT_OF_TESTS = 15
KIND_OF_ELEM = [0, 1]


def generate_matrix(n, m):
  matrix = []
  for i in range(n):
    line = []
    for j in range(m):
      line.append( choice(KIND_OF_ELEM) )
    matrix.append(line)
  return matrix


def find_max_matrix(matrix, n, m):
  ans = 0
  for i in range(n):
    for j in range(m):
      right = m
      down = n
      if matrix[i][j] == 0:
        x = i
        y = j
        while (x < down and matrix[x][y] == 0):
          while (y < right and matrix[x][y] == 0):
            y += 1
          right = y
          x += 1
          ans = max( ans, (x - i)*(y - j) )
          y = j
  return ans

if __name__ == '__main__':
  for enum in range( COUNT_OF_TESTS ):
    test_file_name = "tests/{:02d}".format( enum + 1 )
    with open( "{0}.t".format( test_file_name ), 'w' ) as output_file, \
      open( "{0}.txt".format( test_file_name ), "w" ) as answer_file:
      MIN_SIZE_MATRIX *= 2
      n = random.randint(MIN_SIZE_MATRIX, MIN_SIZE_MATRIX + 1)
      m = random.randint(MIN_SIZE_MATRIX, MIN_SIZE_MATRIX + 1)
```

```
47      output_file.write( "{}\t{}\n".format(n, m) )
48      matrix = generate_matrix(n, m)
49
50      #Print matrix
51      for i in matrix:
52        for j in i:
53          output_file.write( str(j) )
54        output_file.write( "\n" )
55
56      answer_file.write( str(find_max_matrix(matrix, n, m)) )
```

2. makefile

```
1  FLAGS= -std=c++11 -pedantic -Werror -Wno-sign-compare -Wno-long-long -lm -O2
2  CC=g++
3
4  all: matrix main
5
6  main: main.cpp
7    $(CC) $(FLAGS) -o lab7 main.cpp TMatrix.o
8
9  matrix: TMatrix.cpp
10   $(CC) $(FLAGS) -c TMatrix.cpp
```

3. ./wrapper.sh – точка входа в «чекер»

```
1
2  if ! python3 checker.py ; then
3      echo "ERROR: Failed to python generate tests."
4      exit 1
5  fi
6
7  path = './tests/'
8  for test_file in `ls tests/*.t`; do
9      answer_file="${test_file%.*}"
10     echo "Execute ${test_file}"
11     if ! ./src/lab7 < $test_file > "${answer_file}.pl" ; then
12         echo "ERROR"
13         continue
14     fi
15
16     if ! diff -u "${answer_file}.txt" "${answer_file}.pl" > /dev/null ; then
17         echo "Failed"
18     else
19         echo "OK"
20     fi
21 done
```

4. make_graph.py – генератор графика по заданным точкам на языке Python для теста производительности.

```
import matplotlib.pyplot as plt
import numpy as np

PATH_TIME = "benchmark/"

def load_time(search):
    with open(PATH_TIME + search[0] + ".txt", 'r') as time_fast, \
           open(PATH_TIME + search[1] + ".txt", "r") as time_trivial, \
           open(PATH_TIME + "count", "r") as count_file :
        fast = np.array( [ float(i) for i in time_fast.read().split() ] )
        trivial = np.array( [float(i) for i in time_trivial.read().split() ] )
        count_of_elements = np.array( [ int(i) for i in count_file.read().split()
            ] )
    return count_of_elements, fast, trivial


if __name__ == '__main__':
    search = ["fast", "trivial"]

    count, fast, trivial = load_time(search)

    line_main, line_ntl = plt.plot(count, fast, 'red', count, trivial, 'blue')

    plt.title(u'    !')

    plt.xlabel(u'   ')

    plt.ylabel(u'')
    plt.legend((line_main, line_ntl),
        (u'fast', u'trivial'), loc="best")
    plt.savefig( 'oneqetnil.png', format='png')
```

3 Проверка на утечки памяти

Теперь проверим программу на наличие утечек памяти с помощью утилиты **valgrind** на матрице размером $10^3 \cdot 10^2$ элеметов.

```
[bokoch@MacKenlly lab_2]$ valgrind --leak-check=yes ./lab7 < tests/07.t > dev
==4648==
==4648== HEAP SUMMARY:
==4648==     in use at exit: 72,704 bytes in 1 blocks
==4648==   total heap usage: 10204 allocs, 10203 frees, 3,728,034 bytes allocated
==4648==
==4648== LEAK SUMMARY:
==4648==    definitely lost: 0 bytes in 0 blocks
```

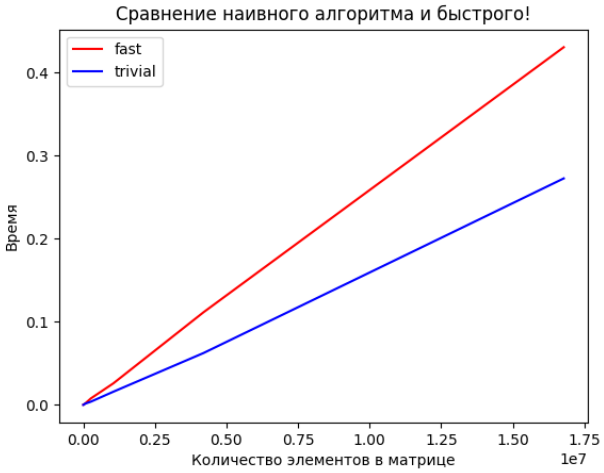
```
==4648==    indirectly lost: 0 bytes in 0 blocks
==4648==      possibly lost: 0 bytes in 0 blocks
==4648==    still reachable: 72,704 bytes in 1 blocks
==4648==         suppressed: 0 bytes in 0 blocks
==4648== Rerun with --leak-check=full to see details of leaked memory
==4648==
==4648== For counts of detected and suppressed errors, rerun with: -v
==4648== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

4 Тест производительности

Сравнение производилось с тривиальным алгоритмом, который работает за $\Theta(n^2 \cdot m^2)$.
Количество тестов: 12.
Размеры матриц: $[6 \dots 6 \cdot 2^{11}]$ с шагом $x \cdot 2$.

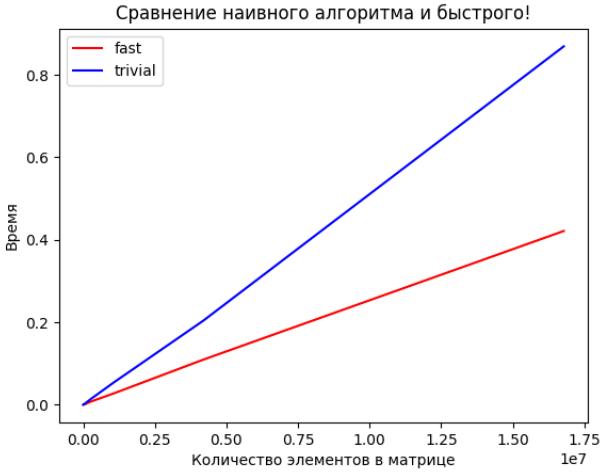
```
1  #include <ctime>
2  #include <cstdio>
3  #include "TMatrix.h"
4
5  double FindMaxSubMatrixTrivialTime(TMatrix& matrix) {
6    clock_t start = clock();
7    matrix.FindMaxSubMatrixTrivial(0);
8    clock_t finish = clock();
9    return (double) (finish - start)/CLOCKS_PER_SEC;
10 }
11
12 double FindMaxSubMatrixFastTime(TMatrix& matrix) {
13   clock_t start = clock();
14   matrix.FindMaxSubMatrixFast(0);
15   clock_t finish = clock();
16   return (double) (finish - start)/CLOCKS_PER_SEC;
17 }
18
19 int main(int argc, char *argv[]) {
20   FILE* fout_trivial;
21   FILE* fout_fast;
22   FILE* fout_count;
23
24   fout_count = fopen("./count", "a+");
25   fout_trivial = fopen("./trivial.txt", "a+");
26   fout_fast = fopen("./fast.txt", "a+");
27
28     std::size_t n, m;
29     std::cin >> n >> m;
30     TMatrix matrix(n, m);
31     std::cin >> matrix;
32
33     fprintf( fout_fast, "%0.15f\n", FindMaxSubMatrixFastTime(matrix) );
34     fprintf( fout_trivial, "%0.15f\n", FindMaxSubMatrixTrivialTime(matrix) );
35     fprintf( fout_count, "%d\n", n*m );
36
37     fclose(fout_trivial);
38     fclose(fout_fast);
39     fclose(fout_count);
40     return 0;
41 }
```

Честно сказать, результаты меня удивили. При приблизительно равном соотношении нулей и единиц тривиальный алгоритм работает значительно быстрее оптимального.



Причина очень проста, перед поиском размера матрицы присутствует проверка на равенство нулю левого верхнего края матрицы. Поэтому при встрече единицы, алгоритм переходит к следующему элементу.

Но при соотношении 5 : 1 нулей и единиц соответственно, оптимальный алгоритм доказывает своё превосходство.



5 Выводы

Динамическое программирование применяется при наличии двух характерных признаков:

- оптимальность для подзадач;
- наличие в задаче перекрывающихся подзадач.

Действительно в данной задаче требуется идти от простой задачи к более сложной. При оптимальном алгоритме экономится не только время, но и память – $\Theta(n)$. Задачу, где можно применить динамическое программирование легко увидеть, но сложно написать. Я, к сожалению, к данному алгоритму пришел не сам(кроме тривиального), поэтому стоит потренироваться решать подобные задачи на платформе *codeforces*.

Замечание: с указанными входными данными даже тривиальный алгоритм прошел все тесты.

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.))(дата обращения: 11.03.2018).

[2] *Понятие динамического программирования*
URL:`http://fb.ru/article/44641/dinamicheskoe-programmirovanie-osnovnyie-printsipyi`
(дата обращения: 12.03.2018).

[3] *Описание алгоритма о поиске максимальной нулевой подматрицы*
URL:`http://www.e-maxx-ru.1gb.ru/algo/maximum`$_z ero_s ubmatrix$(: 12.03.2018).