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: А.Д. Волков
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б

Дата: 17.11.2024
Оценка:
Подпись:

Москва, 2024

Лабораторная работа №7

Задача: 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.

1 Описание

Требуется реализовать алгоритм динамического программирования для нахождения оптимального пути по матрице. Определим термин динамического программирования.

Динамическое программирование — это метод решения сложных задач, который разбивает их на более простые подзадачи и решает каждую из них только один раз, сохраняя результаты для последующего использования. Этот подход особенно эффективен для задач, которые могут быть разбиты на перекрывающиеся подзадачи, что позволяет избежать повторного вычисления одних и тех же решений. Процесс разработки алгоритмов динамического программирования можно разделить на четыре нижеперечисленных этапа:
1. Описание структуры оптимального решения.
2. Определение значение, соответствующего оптимальному решению, с использованием рекурсии.
3. Вычисление значения, соответствующего оптимальному решению, обычно с помощью метода восходящего анализа.
4. Составление оптимального решения на основе информации, полученной на предыдущих этапах.
Обычно имеется два эквивалентных способоа реализации подхода динамического программирования.
Первый подход - нисходящий с запоминанием. При таком подходе мы пишем процедуру рекурсивно, как обычно, но модифицируем ее таким образом, чтобы она запоминала решение каждой подзадачи. Теперь процедура первым делом проверяет, не была ли эта задача решена ранее. Если была, то возвращается сохраненное значение. Если же подзадача еще не решалась, процедура вычисляет возвращаемое значение, как обычно.
Второй подход - восходящий. Обычно он зависит от некоторого естественного понятия размера подзадачи, такого, что решение любой конкретной подзадачи зависит только от рещения меньших подзадач. Мы сортируем подзадачи по размерам в возрастающем порядке. При решении определенной подзадачи необходимо решить все меньшик подзадачи, от которых она зависит, и сохранить полученные решения. Каждую подзадачу мы решаем только один раз, и к моменту, когда мы впервые с ней сталкиваемся, все необходимые для ее решения подзадачи уже решены. [1]

2 Исходный код

Сначала мы считываем все данные и сохраняем матрицу в двумерный вектор. Затем мы заводим двумерный вектор, где мы будем хранить матрицу минимальных суммарных штрафов для каждого этапа. Первая строчка матрицы естественно будет аналогична исходной, поэтому записываем сначала ее. Затем переходим к следующим строкам в матрице. Рассмотрим наши ходы. Из клетки мы можем пойти вниз и вправо, просто вниз и вниз и влево. Следовательно в клетку мы можем прийти из верхней левой клетки, верхней клетки и верхней правой клетки. Рассмотрим штрафы, которые получаются после прохода через эти клетки и найдем среди них минимальный и запишем его. Так делаем для всех оставшихся клеток. По итогу в последней строке матрицы мы получим минимальные штрафы, с которыми мы можем добраться до клеток в нижней строке. Но так как нам неважно до какой клетки нам надо добраться, просто среди этих штрафов берем минимальный и получаем нужный ответ. Таким образом мы решили задачу с помощью нисходящего подхода с мемоизацией, так как мы сохраняли матрицу со штрафами. Теперь нам нужно восстановить путь по которому мы прошли. Это можно сделать с помощью нашей матрицы со штрафами. Создаем вектор, для индексов колонок, по которым мы прошли. Строки нам хранить не нужно. Сразу в конец пути, мы записываем индекс колонки, где лежит наш ответ, так как он точно входит в наш путь. Далее мы идем по всем клеткам, из которых мы могли попасть в текущую клетку и находим среди них минимальный штраф. Записываем индекс этого штрафа в путь и так делаем для каждой строки. Таким образом мы получили путь, который нужно пройти для того, чтобы набрать минимальное количество штрафов.

```
#include <bits/stdc++.h>

size_t find_min_idx(std::vector<int64_t> vec)
{
    int64_t min = 0;
    for (size_t i = 1; i < vec.size(); ++i)
    {
        if (vec[i] < vec[min])
        {
            min = i;
        }
    }
    return min;
}

int main()
{
    int64_t rows, columns;
    std::cin >> rows >> columns;
    std::vector<std::vector<int64_t>> matrix(rows, std::vector<int64_t>(columns));
```

```
21      for (size_t i = 0; i < rows; ++i)
22      {
23          for (size_t j = 0; j < columns; ++j)
24          {
25              std::cin >> matrix[i][j];
26          }
27      }
28
29      // min penalty calculating
30      std::vector<std::vector<int64_t>> matrix_penalties(rows, std::vector<int64_t>(
            columns, std::numeric_limits<int64_t>::max()));
31      for (size_t i = 0; i < rows; ++i)
32      {
33          for (size_t j = 0; j < columns; ++j)
34          {
35              if (i == 0)
36              {
37                  // initial penalties
38                  matrix_penalties[i][j] = matrix[i][j];
39              }
40              else
41              {
42                  if (j != 0)
43                  {
44                      // up and left
45                      matrix_penalties[i][j] = std::min(matrix_penalties[i][j], matrix[i][
                            j] + matrix_penalties[i - 1][j - 1]);
46                  }
47                  // up
48                  matrix_penalties[i][j] = std::min(matrix_penalties[i][j], matrix[i][j] +
                         matrix_penalties[i - 1][j]);
49                  if (j != columns - 1)
50                  {
51                      // up and right
52                      matrix_penalties[i][j] = std::min(matrix_penalties[i][j], matrix[i][
                            j] + matrix_penalties[i - 1][j + 1]);
53                  }
54              }
55          }
56      }
57      size_t min_idx = find_min_idx(matrix_penalties[rows - 1]);
58      std::cout << matrix_penalties[rows - 1][min_idx] << "\n";
59
60      // path calculating
61      std::vector<int64_t> path(rows);
62      path[rows - 1] = min_idx;
63      for (int i = rows - 1; i > 0; --i)
64      {
65          int64_t min_penalty_sum = std::numeric_limits<int64_t>::max();
```

```
66          // up and left
67          if (min_idx != 0)
68          {
69              if (min_penalty_sum > matrix_penalties[i - 1][min_idx - 1])
70              {
71                  min_penalty_sum = matrix_penalties[i - 1][min_idx - 1];
72                  path[i - 1] = min_idx - 1;
73              }
74          }
75          // up
76          if (min_penalty_sum > matrix_penalties[i - 1][min_idx])
77          {
78              min_penalty_sum = matrix_penalties[i - 1][min_idx];
79              path[i - 1] = min_idx;
80          }
81          // up and right
82          if (min_idx != columns - 1)
83          {
84              if (min_penalty_sum > matrix_penalties[i - 1][min_idx + 1])
85              {
86                  min_penalty_sum = matrix_penalties[i - 1][min_idx + 1];
87                  path[i - 1] = min_idx + 1;
88              }
89          }
90          min_idx = path[i - 1];
91      }
92
93      for (size_t i = 0; i < path.size(); ++i)
94      {
95          std::cout << "(" << i + 1 << "," << path[i] + 1 << ") ";
96      }
97      std::cout << "\n";
98  }
```

main.cpp	
size_t find_min_idx(std::vector<int64_t> vec)	Функция для нахождения индекса минимального элемента в векторе чисел
int main()	Точка входа программы

3 Консоль

```
lexasy@lexasy$ cat test.txt
3 3
3 1 2
7 4 5
8 6 3
lexasy@lexasy$ make
g++ main.cpp -o solution
lexasy@lexasy$ ./solution <test.txt
8
(1,2) (2,2) (3,3)
```

4 Тест производительности

Тест производительности представляет из себя сравнение времени работы нашего алгоритма и наивного алгоритма, работающего за экспоненциальное время на большом тесте, а именно на матрице 8 на 7.

```
lexasy@lexasy$ make
g++ main.cpp -o solution
lexasy@lexasy$ make benchmark
g++ benchmark.cpp -o benchmark
lexasy@lexasy$ ./solution <test.txt | grep time
time: 107ms
lexasy@lexasy$ ./benchmark <test.txt | grep time
time: 1936ms
```

Как мы видим, скорость работы выше практически в 20 раз, а это лишь на маленьком тесте 8 на 7. Уже на тесте с матрицей 20 на 20, наивный алгоритм не может решить задачу за вменяемое время.

5 Выводы

Выполнив лабораторную работу №7, я узнал как решать сложные задачи с помощью метода динамического программирования. С помощью решения методом динамического программирования, можно ускорить решение в бесчисленное количество раз, а также такое решение будет красивым и читаемым.

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.))