Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №7 по курсу «Дискретный анализ»

Студентка: А. А. Довженко
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б

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 зададния.

Вариант 5: Обход матрицы. 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

Как правильно заметил Кормен [1], динамическое программирование – это способ решения сложных задач путём разбиения их на более простые подзадачи. Он применим к задачам с оптимальной подструктурой, выглядящим как набор перекрывающихся подзадач, сложность которых чуть меньше исходной: в этом случае время вычислений можно значительно сократить. Как правило, чтобы решить поставленную задачу, требуется решить отдельные части задачи (подзадачи), после чего объединить решения подзадач в одно общее решение. Часто многие из этих подзадач одинаковы. Подход динамического программирования состоит в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений. Это особенно полезно в случаях, когда число повторяющихся подзадач экспоненциально велико.

Этапы построения алгоритма решения задач, решаемых методом динамического программирования:

- Описать структуру оптимального решения.
- Составить рекурсивное решение для нахождения оптимального решения.
- Вычисление значения, соответсвующего оптимальному решению, методом восходящего анализа.
- Непосредственное нахождение оптимального решения из полученной на предыдущих этапах информации.

Если нас интересует только значение оптимального решения, последний шаг можно опустить.

2 Исходный код

В один вектор (rowPrev) поочередно записываем значения элементов строк. В другом векторе (rowCur) будем считать минимальную сумму пройденного пути. На каждой итерации, проходя по вектору rowPrev, находим, какой элемент из каждых трех соседних наименьший. Его записываем в вектор rowCur, а координату столбца этого элемента запишем в массив allPath, который понадобится для восстановления пути. Пройдя всю матрицу, находим наименьший элемент в векторе rowCur. Это и есть ответ. Для восстановления пути будем проходить по матрице allPath с нижней строки до верхней. В качестве индекса столбца используем значение из матрицы на предыдущем шаге.

```
#include <iostream>
#include <vector>
#include <limits>
#include <algorithm>

const size_t MAX = 1000;

int main(void)
{
    int n = 0, m = 0;
    std::cin >> n >> m;
    long long allPath[MAX][MAX];
    std::vector<long long> rowPrev(m + 2), rowCur(m);

    rowPrev[0] = std::numeric_limits<long long>::max();
    rowPrev[m + 1] = std::numeric_limits<long long>::max();

    for (int i = 0; i < m; ++i) {
        std::cin >> rowPrev[i + 1];
    }

    for (int i = 1; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            std::cin >> rowCur[j];
        }

        for (int j = 1; j <= m; ++j) {
            long long minCost = std::numeric_limits<long long>::max();
            if (rowPrev[j - 1] < minCost) {
                minCost = rowPrev[j - 1];
                allPath[i][j - 1] = j - 1;
            }
            if (rowPrev[j] < minCost) {
                minCost = rowPrev[j];
                allPath[i][j - 1] = j;
            }
```

```
37              if (rowPrev[j + 1] < minCost) {
38                  minCost = rowPrev[j + 1];
39                  allPath[i][j - 1] = j + 1;
40              }
41              rowCur[j - 1] += minCost;
42          }
43
44          for (int j = 0; j < m; ++j) {
45              rowPrev[j + 1] = rowCur[j];
46          }
47      }
48
49      auto minEl = std::min_element(rowCur.begin(), rowCur.end());
50      long long ans = *minEl;
51      int end = std::distance(rowCur.begin(), minEl) + 1;
52
53      std::vector<long long> path;
54      for (int i = 0; i < n; ++i) {
55          path.push_back(end);
56          end = allPath[n - i - 1][end - 1];
57      }
58
59      std::cout << ans << std::endl;
60      for (int i = 0; i < n; ++i) {
61          i == 0 ? std::cout << "(" : std::cout << " (";
62          std::cout << i + 1 << "," << path[n - i - 1] << ")";
63      }
64      std::cout << std::endl;
65
66      return 0;
67  }
```

3 Консоль

```
karma@karma:~/mai_study/DA/lab7$ make
g++ -std=c++11 -O3 -pedantic -lm -o da7 main.cpp
karma@karma:~/mai_study/DA/lab7$ cat test
3 3
3 1 2
7 4 5
8 6 3
karma@karma:~/mai_study/DA/lab7$ cat test | ./da7
8
(1,2) (2,2) (3,3)
```

4 Тест производительности

Тест производительности представляет собой сравнение с наивным решением этой задачи, в котором считаются все возможные обходы матрицы, а потом выбирается обход с наименьшим штрафом.

Генерирую 3 файла, содержащие матрицы размерами 10x10 (input10), 20x20 (input20) и 30x30 (input30). Значения элементов лежат в диапазоне от 0 до 10^4.

```
karma@karma:~/mai_study/DA/lab7$ cat input10 | ./naive
Naive time: 0.000728 sec.
karma@karma:~/mai_study/DA/lab7$ cat input10 | ./dp
DP time: 7.4e-05 sec.
karma@karma:~/mai_study/DA/lab7$ cat input20 | ./naive
Naive time: 79.2608 sec.
karma@karma:~/mai_study/DA/lab7$ cat input20 | ./dp
DP time: 0.000163 sec.
karma@karma:~/mai_study/DA/lab7$ cat input30 | ./dp
DP time: 0.000285 sec.
```

Прождав около полутора часов, я так и не получила результатов вычисления наивного алгоритма для теста с матрицей размером 30x30. Предполагаемая сложность наивного алгоритма – $O(m * 3^n)$, а оптимизированного – $O(m * n)$. Посмотрев на результаты тестирования, нетрудно заметить, что так оно и вышло. Наивный алгоритм работает непозволительно долго. При решении этой задачи лучше его не использовать. Динамический алгоритм, напротив, показал очень хороший результат. Даже на матрице размером 1000x1000 он решает задачу за считанные доли секунды.

5 Выводы

К выводам можно отнести всё то, что сказано в описании: динамическое программирование – это метод, который зачастую позволяет построить радикально ускоренную и улучшенную версию алгоритма. Сфера его применений очень широка: всюду, где имеются перекрывающиеся подаздачи, и где эти самые подзадачи относительно легко выделить, динамическое программирование может оказать неоценимую поддержку быстродействию программы. И примеры тоже бывают самыми разными, некоторые из них представлены в вариантах этой лабораторной. Такое разнообразие вызвано именно тем, что ДП – это метод построения алгоритмов, а не просто какой-то алгоритм.

Список литературы

[1] Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. *Алгоритмы: построение и анализ, 3-е изд.* — Издательский дом «Вильямс», 2013. Перевод с английского: ООО «И. Д. Вильямс». — 1328 с. (ISBN 978-5-8459-1794-2 (рус.))