Санкт-Петербургский Национальный Исследовательский
Университет Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

Лабораторня работа №5

Выполнили:
Стафеев И.А., Лапшина Ю.С., Килебе Нтангу Дьевина
Проверил
Мусаев А.А.

Санкт-Петербург,
2023

СОДЕРЖАНИЕ

Стр.

ВВЕДЕНИЕ

Цель работы: познакомиться с применением жадных алгоритмов, метода динамического программирования и метода "разделяй и властвуй" для решения практических задач. Для достижения цели были поставлены следующие задачи:

- изучить концепцию жадных алгоритмов;
- изучить метод динамического программирования;
- изучить метод "разделяй и властвуй";
- написать алгоритмы для трех задач, предполагающих применение названных методов.

1 Нахождение наибольшей суммы, которую может получить вор

Для решения данной задачи можно использовать несколько подходов. Поскольку "Жадный алгоритм" в данном случае не всегда выдавал бы лучший результат, было был выбран способ решения с помощью динамического программирования.

Была написана функция *knapsack*, в которую подаётся кортеж из всех пар значений вес/цена для каждого экспоната, а также массу груза, который может унести вор (в дальнейшем - вместимость рюкзака). При заполнении двумерного массива *dp* было использовано динамическое программирование. Для каждого экспоната мы рассматриваем всевозможные вместимости для рюкзака от 0 до K. $dp[i][j]$ - текущее значение для экспоната i и вместимости рюкзака j (изначально принимаем его равным $dp[i-1][j]$). Это значение представляет собой максимальную стоимость, которую можно получить с учетом всех экспонатов до текущего и доступной вместимости рюкзака. Если вес рассматриваемого экспоната больше или равен рассматриваемой вместимости рюкзака, то в ячейку $dp[i][j]$ мы записываем максимальное значение из стоимости рюкзака без взятия экспоната ($dp[i-1][j]$) и стоимости рюкзака с массой, меньшей на вес экспоната, и цены экспоната ($dp[i-1][j-weight] + price$). Таким образом, в ячейке $dp[-1][-1]$ окажется наибольшая возможная стоимость рюкзака.

Чтобы узнать, какие экспонаты вор унёс, нужно восстановить набор предметов, которые были использованы при составлении максимальной ценности рюзака. Для этого осуществляется проход по массиву *dp* для всех экспонатов, начиная с последнего (т.е мы отслеживаем изменение значений с самого конца массива). Если значение в ячейке $dp[i][capacity]$ (цена рюкзака с рассматриваемым экспонатом) отличается от значения в $dp[i-1][capacity]$ (цена рюкзака без экспоната), то рассматриваемый эскпонат был взят, и он добавляется в массив *result*, который и является результатом выполнения функции (список всех взятых экспонатов). Далее вместимость рюкзака уменьшается на вес взятого экспоната и поиск продолжается до самого начала массива.

Поскольку у вора есть M заходов, то для каждого захода вызывается функция *knapsack*, похищенные экспонаты добавляются в общий список похищенного ($general\_stolen\_stuff$) и удаляются из массива $stuff$. В конце выводится информация по всем заходам об украденных экспонатах.

Код реализованной программы представлен на рисунке: 1.

```python
def knapsack(stuff, capacity):
    n = len(stuff)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    #заполнение dp динамически
    for i in range(1, n + 1):
        for j in range(capacity + 1):
            weight, price = stuff[i - 1]
            dp[i][j] = dp[i - 1][j]
            if weight <= j:
                dp[i][j] = max(dp[i][j], dp[i - 1][j - weight] + price)


    #получение забранных предметов
    result = []

    for i in range(n, 0, -1):
        if dp[i][capacity] != dp[i - 1][capacity]:
                result.append(stuff[i - 1])
                capacity -= stuff[i - 1][0]

    return result



if __name__ == '__main__':
    N, M, K = map(int, input().split())
    # stuff = [(random.randint(1, 30), random.randint(1, 30)) for _ in range(N)]
    stuff = [tuple(map(int, input().split())) for _ in range(N)]
    # stuff = [(1, 2), (2, 1), (1, 4), (2, 10), (1, 14), (6, 10), (3, 8)]
    print(stuff)
    general_stolen_stuff = []
    for _ in range(M):
        if len(stuff) == 0:
            break

        stolen_stuff = knapsack(stuff, K)
        general_stolen_stuff.append(stolen_stuff)

        # Удаляем вещи, которые унесли
        for item in stolen_stuff:
            stuff.remove(item)
```

Рисунок 1 — Код нахождения наибольшей суммы, которую может украсть вор

Пример работы программы также представлен на рисунке: 2.

```
7 3 6
1 2
2 1
1 4
2 10
1 14
6 10
3 8
[(1, 2), (2, 1), (1, 4), (2, 10), (1, 14), (6, 10), (3, 8)]
Заход 1:
weight: 3, price: 8
weight: 1, price: 14
weight: 2, price: 10
Заход 2:
weight: 6, price: 10
Заход 3:
weight: 1, price: 4
weight: 2, price: 1
weight: 1, price: 2
```

Рисунок 2 — Пример работы программы по нахождению наибольшей суммы, которую может украсть вор

2 Минимизация скалярных операция для перемножения матриц

Формализуем условие и решим общем виде. Пусть есть последовательность натуральных чисел $a_0, a_1, \cdots, a_n$. Размер матрицы A_i, $i \geq 1$ равен $a_{i-1} \times a_i$. Получается, количество столбцов матрицы A_i и количество строк матрицы A_{i+1} равны, то их можно перемножить. Так как такое равенство выполняется для всех пар подряд идущих матриц, то выполняется свойство ассоциативности. Для перемножения матриц, требующего миимального количества скалярных операций, необходимо определить оптимальный порядок перемножения матриц, то есть расставить скобки.

Заметим, что перемножение всех матриц получается из перемножения каких-то двух матриц, например, $A_1 \times A_{n-1}$ и A_n. Каждая из участвующих в финальном перемножении матриц получается также перемножением двух каких-то матриц, и так далее. Тогда мы можем применить метод динамического программирования, вычисляя последовательно минимальное количество скалярных операций для перемножения части матриц, а потом на основе сделанных вычислений получим ответ для всех матриц.

Пусть $min\_cost[i][j]$ - минимальное количество скалярных операций для умножения матриц $A_i, \cdots, A_j$. Ответ на задачу будет содержаться в $min\_cost[1][n]$. Понятно, что $min\_cost[i][j] = 0$. Будем последовательно вычислять $min\_cost[i][j]$ для $j - i = 1, j - i = 2, \cdots, j - i = n$. Как уже было сказано, произведение матриц получается из перемножения двух других матриц.

Для каждого $k, i \leq k < j$ посчитаем значение $min\_cost[i][k] + min\_cost[k+1][j] + a_i \times a_{k+1} \times a_{j+1}$, где последнее слагаемое - количество скалярных операций для умножения матрицы $A_{i \cdots k}$ на матрицу $A_{k+1 \cdots j}$, если s - массив размеров графа. Значение $min\_cost[i][j]$ есть минимальное из посчитанных значений для каждого k.

Описанный алгоритм реализован в функции $find\_min\_cost$ и представлен на рисунке 3.

```python
def find_min_cost(sizes: list[int], print_table_flag=False):
    """Нахождение минимального числа операций для умножения матриц"""
    n = len(sizes) - 1
    min_cost = [[0 for _ in range(n)] for _ in range(n)]
    parents = {}

    for i_j in range(1, n):
        for i in range(n - i_j):
            j = i + i_j
            min_cost[i][j] = 10**12
            for k in range(i, j):
                num_of_operations = sizes[i] * sizes[k + 1] * sizes[j + 1]
                value = min_cost[i][k] + min_cost[k + 1][j] + num_of_operations
                if value < min_cost[i][j]:
                    min_cost[i][j] = value
                    parents[i, j] = k

    if print_table_flag:
        print_min_cost_table(min_cost)

    return min_cost[0][n - 1], get_tree(0, n - 1, parents)
```

Рисунок 3 — Алгоритм поиска наибольшей возрастающей подпоследовательности

Описанная функция возвращает минимальное количество операций, требуемых для перемножения матриц, а также дерево, отображающее порядок перемножения матриц, при котором используется минимальное количество операций. Это дерево можно получить, если для каждой пары вершин (i, j) запоминать k, при котором перемножение соответстствующих матриц требует минимального числа операций. Получается, при обходе этого дерева в ширину порядок обхода вершин будет задавать обратный оптимальный порядок умножения матриц.

Пример работы программы для 10 матриц представлен на рисунке 4. Понятно, что асимтотическая сложность алгоритма составляет $O(n^3)$.

```
PS D:\ProgrammingProjects\itmo_algos_labs> & "D:/Program Files/Python311/python.exe" d:/Progr
5/task2.py
Массив размеров: [5, 7, 6, 4, 6, 4, 3, 6, 6, 6, 4]
Результат умножения матриц:
[-82756649, 116152642, -98096209, -44848726]
[93824018, -155494789, 85078288, 150750397]
[20269328, -34379773, 17265478, 56674669]
[82242936, -125307308, 87366336, 26085788]
[86230136, -118735169, 104926726, 20779649]
Количество операций при умножении матриц подряд: 1230


Количество скалярных операций для вычисления произведения матриц Ai...Aj
i\j     1       2       3       4       5       6       7       8       9       10
1       0       210     308     428     484     447     537     645     753     795
2       0       0       168     336     360     342     468     576     684     714
3       0       0       0       144     192     216     324     432     540     576
4       0       0       0       0       96      144     216     324     432     480
5       0       0       0       0       0       72      180     288     396     432
6       0       0       0       0       0       0       72      180     288     336
7       0       0       0       0       0       0       0       108     216     288
8       0       0       0       0       0       0       0       0       216     288
9       0       0       0       0       0       0       0       0       0       144
10      0       0       0       0       0       0       0       0       0       0

Результат множения матриц в оптимальном порядке (такой же):
[-82756649, 116152642, -98096209, -44848726]
[93824018, -155494789, 85078288, 150750397]
[20269328, -34379773, 17265478, 56674669]
[82242936, -125307308, 87366336, 26085788]
[86230136, -118735169, 104926726, 20779649]
Количество операций при умножении в оптимальном порядке: 795
Порядок умножения:
((A1) * ((A2) * ((A3) * ((A4) * ((A5) * (A6)))))) * ((((A7) * (A8)) * (A9)) * (A10))
PS D:\ProgrammingProjects\itmo_algos_labs>
```

Рисунок 4 — Алгоритм поиска наибольшей возрастающей подпоследовательности

3 Нахождение наибольшей возрастающей подпоследовательности

Создадим вспомогательный массив dp длиной n, в котором каждый элемент будет содержать длину наибольшей последовательности, оканчивающейся в этой позиции.

Начальное значение $dp[i]$ для всех элементов будет равно 1, так как любой элемент массива является последовательностью длины 1 сам по себе.

Затем, начиная с позиции 1, мы будем перебирать элементы массива, проверяя, если предыдущий элемент меньше текущего элемента. Если это так, то мы можем продлить последовательность, добавив текущий элемент, иначе текущая последовательность обрывается в этой позиции и мы начнем новую последовательность с текущего элемента.

В процессе перебора всех элементов, мы будем обновлять значения $dp[i]$ - если текущая последовательность длиннее, чем уже найденная на данной позиции.

Наконец, мы найдем максимальное значение в массиве dp, которое представляет длину наибольшей непрерывной возрастающей последовательности, и соответствующую ему позицию. Зная индекс максимального значения в dp (то есть конец подпоследовательности), легко найти индекс начала, вычев из индекса максимального значения само максимальное значение и прибавив 1.

Реализация описанного словесного алгоритма представлена на рисунке 5. Функция $find\_longest\_increasing\_seq$ принимает массив и возвращает индексы начала и конца наибольшей возрастающей подпоследовательности.

```python
def find_longest_increasing_seq(arr):
    n = len(arr)
    dp = [1] * n

    for i in range(1, n):
        if arr[i] > arr[i-1]:
            dp[i] = dp[i-1] + 1

    max_length = max(dp)
    max_length_index = dp.index(max_length)
    start_index = max_length_index - max_length + 1
    return start_index, max_length_index
```

Рисунок 5 — Алгоритм поиска наибольшей возрастающей подпоследовательности

Пример работы программы представлен на рисунке 6.

```
PS D:\ProgrammingProjects\itmo_algos_labs> & "D:/Program Files/Pytho
5/task3.py
20
Исходный массив
-4 47 74 -69 25 98 89 74 18 -73 -88 -2 -92 -57 -9 54 60 -74 7 71
С выделенной подпоследовательностью:
-4 47 74 -69 25 98 89 74 18 -73 -88 -2 -92 -57 -9 54 60 -74 7 71
PS D:\ProgrammingProjects\itmo_algos_labs>
```

Рисунок 6 — Пример работы программы по поиску наибольшей возрастающей подпоследовательности

ЗАКЛЮЧЕНИЕ

Результатом выполнения работы стало повышение навыков по использованию динамического программирования, жаных алгоритмов и метода "разделяй и властвуй". В ходе работы были написаны: алгоритм, определяющий наибольшую сумму, которую может получить вор в результате ограбления музея; алгоритмы, определяющий порядок перемножения матриц, при котором используется минимальное число скалярных операций; алгоритм, находящий наибольшую возрастающую подпоследовательность. Таким образом, полученные знания позволят более эффективно применять описанные методы в практической деятельности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ахо А.В., Хопкрофт Д.Э., Ульман Дж.Д. Алгоритмы и структуры данных [Электронный ресурс] – URL: https://vk.com/wall-114485185_260 (дата обращения: 09.11.2023)

2. Knapsack problem [Электронный ресурс] - URL: https://youtu.be/nTABeARo3HU?si=DYZAV7R_9KqLyRFN (дата обращения: 13.11.2023)

3. Matrix Chain Multiplication - Dynamic Programming [Электронный ресурс] - URL: https://youtu.be/prx1psByp7U?si=QDVx3aInn1pACCR6 (дата обращения: 09.11.2023)