

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

Факультет инфокоммуникационных технологий

Домашнее задание №3

Вариант №1

Выполнил:

Бацанова Е. А.

Проверил

Мусаев А.А.

Санкт-Петербург,

2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	3
Задание 1.....	4
Задание 2.....	6
Задание 3.....	8
Задание 4.....	9
ЗАКЛЮЧЕНИЕ.....	11
СПИСОК ЛИТЕРАТУРЫ.....	12
ПРИЛОЖЕНИЕ.....	13

ВВЕДЕНИЕ

Целью данной работы является изучение, реализация и анализ жадных алгоритмов. Основными задачами работы являлось:

- 1) реализация жадного алгоритма для решения задачи о выдаче сдачи заданного размера минимальным числом монет;
- 2) реализация жадного алгоритма для решения задачи о максимизации цены украденных вором вещей, если число заходов, а также максимальный вес, который вор может вынести за заход ограничены;
- 3) сравнение перечисленных ранее задач, анализ применимости и качества работы жадных алгоритмов в данных задачах;
- 4) реализация алгоритма Дейкстры и его тестирование на реальных данных о улицах Санкт-Петербурга.

Задание 1

Задание: Пользователю необходимо дать сдачу N рублей. У него имеется $M1$ монет номиналом $S1$, $M2$ монет номиналом $S2$, $M3$ монет номиналом $S3$ и $M4$ монет номиналом $S4$. Необходимо найти наименьшую комбинацию из заданных монет, которые позволят получить в сумме N .

Решение:

Для нахождения наименьшей комбинации из заданных монет, которая позволит дать сдачу N , реализуем функцию `find_change(N, M1, S1, M2, S2, M3, S3, M4, S4)` (рис. 1). В её основе лежит “жадный алгоритм” – мы берем монету максимально возможного номинала, пока не получим в сумме N .

Сначала программа сортирует массивы монет (и вместе с этим их количества) по убыванию, чтобы обрабатывать первыми монеты большего номинала. Затем программа начинает рассчитывать сдачу с монет наибольшего номинала и идет к монетам меньшего номинала, снижая значение сдачи и число оставшихся монет, пока сдача не окажется равной нулю. После завершения выполнения, программа вернет словарь `result`, содержащий номиналы монет в качестве ключей и то, какое их количество понадобилось, чтобы набрать сдачу N , в качестве соответствующего значения. Если сдачу не удастся дать, программа выдаст сообщение *"Невозможно дать сдачу"*.

```
def find_change(N, M1, S1, M2, S2, M3, S3, M4, S4):
    coins = [S1, S2, S3, S4]
    counts = [M1, M2, M3, M4]
    for i in range(3):
        for j in range(0, 3 - i):
            if coins[j] > coins[j + 1]:
                coins[j], coins[j + 1] = coins[j + 1], coins[j]
                counts[j], counts[j + 1] = counts[j + 1], counts[j]
    change = N
    result = {}
    for i in range(3, -1, -1):
        while change >= coins[i] and counts[i] > 0:
            change -= coins[i]
            counts[i] -= 1
            if coins[i] in result:
                result[coins[i]] += 1
            else:
                result[coins[i]] = 1
    if change == 0:
        return result
    else:
        return "Невозможно дать сдачу"
```

Рисунок 1 – Функция, реализующая поиск наименьшей комбинации из заданных монет, которая позволит дать сдачу заданного размера

Протестируем программу. Видим, что программа действительно выводит оптимальное решение задачи. Стоит отметить, что данный алгоритм всегда выводит оптимальное значение результата.

```
Монет номинала 10: 2  
Монет номинала 5: 1  
Монет номинала 2: 1  
Монет номинала 1: 1
```

Рисунок 2 – Вывод программы, реализующей поиск наименьшей комбинации из заданных монет, которая позволит дать сдачу заданного размера

Задание 2

Задание: Вор пробрался в музей и хочет украсть N экспонатов. У каждого экспоната есть свой вес и цена. Вор может сделать M заходов, каждый раз унося K кг веса. Определить, что должен унести вор, чтобы сумма украденного была максимальной.

Решение:

Максимизируем сумму украденного вором с помощью функции *max_steal*(M , K , *exhibits*). Сначала создадим список *may_be_stolen*, в который поместим только те экспонаты, которые вор сможет унести – это поможет сократить количество итераций в будущем. Затем сортируем данный список по убыванию отношения цены к весу, при этом для экспонатов, у которых это отношение равно, первее в списке будут стоять экспонаты с большей ценой. Затем функция начинает проходить по M итерациям, реализуя M заходов, пытаясь украсть элементов на максимальную сумму. Каждый раз пытаемся взять элемент с максимальным соотношением цена / вес, если же вес превышает допустимый – идём по списку дальше, стараясь найти элемент, который вор сможет унести в этот заход. В конце функция возвращает общую стоимость украденных элементов.

```
def max_steal(M, K, exhibits):
    may_be_stolen = [i for i in exhibits if i[0] <= K]
    may_be_stolen.sort(key=lambda x: (x[1] / x[0], x[1]), reverse=True)
    total_value = 0
    num_of_exhibits = len(may_be_stolen)
    cnt = 0
    stolen = []
    for i in range(M):
        total_weight = 0
        for j in range(num_of_exhibits):
            if j not in stolen:
                if total_weight + may_be_stolen[j][0] <= K:
                    total_weight += may_be_stolen[j][0]
                    total_value += may_be_stolen[j][1]
                    stolen.append(j)
                    cnt += 1
    return total_value
```

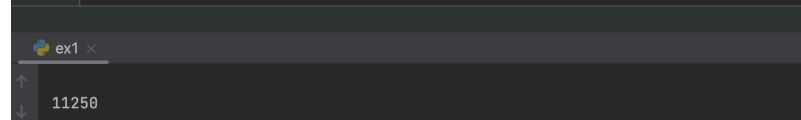
Рисунок 3 – Функция, максимизирующая цену украденных экспонатов

Протестируем работу программы – для примера, изображенного на рисунке 4, алгоритм действительно выводит максимальный результат. Однако для примера с рисунка 5 уже нет. Это связано с тем, что каждый раз мы стремимся взять предмет с

максимальным соотношением цена / вес из-за чего можем использовать заход не оптимально (унесем экспонатов на вес, меньше предельного). В данном случае в последний заход осталось 2 варианта для кражи и оптимальным вариантом был экспонат с меньшим соотношением цена / вес.

Можно сказать, что работа жадных алгоритмов может быть неточной, когда необходимо следить не за одним параметром, как а предыдущей задаче, а за несколькими.

```
20 # пример 1
21 M = 3
22 K = 4
23 exhibits = [(2, 1500), (1, 1000), (4, 5000), (3, 2000), (2, 1250), (3, 2500)]
24 print(max_steal(M, K, exhibits))
```

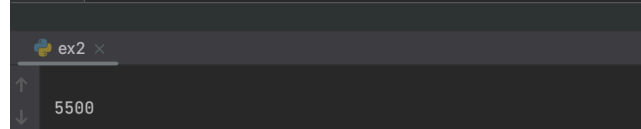


ex1 ×

↑ 11250 ↓

Рисунок 5 – Пример работы программы, максимизирующая цену украденных экспонатов

```
20 # пример 2
21 M = 2
22 K = 4
23 exhibits = [(2, 1500), (1, 1000), (3, 3000), (4, 2000)]
24 print(max_steal(M, K, exhibits))
```



ex2 ×

↑ 5500 ↓

Рисунок 5 – Пример ошибки в работе программы, максимизирующая цену украденных экспонатов

Задание 3

Задание: На основе 1 и 2 задания сделайте выводы о применении «жадных алгоритмов». Всегда ли наилучшие решения на каждом шаге приводят к наилучшему конечному результату?

Решение:

Жадные алгоритмы являются простыми и эффективными, но не всегда гарантируют наилучший результат.

В случае задачи на поиск наименьшей комбинации монет для получения сдачи N , жадный алгоритм успешно находит оптимальное решение. Это связано с тем, что в данной задаче фигурирует всего один параметр и конечная цель, поэтому, выбирая оптимальный вариант на каждом шаге, мы получаем оптимальный вариант для задачи.

В задаче о воре в музее, жадный алгоритм может давать различные результаты в зависимости от выбора критерия оптимальности. Выбранный критерий (отношение цены к весу) хоть и является хорошим и дает верный ответ в большинстве случаев, может также выдавать неоптимальный ответ. Это связано с тем, что максимизируя один параметр, мы можем неоптимально использовать другой (например, за один заход вор может уносить вес, гораздо меньше максимально возможного).

Таким образом, наилучшие решения на каждом шаге не всегда приводят к наилучшему конечному результату при применении жадных алгоритмов. Тем не менее, в некоторых случаях жадные алгоритмы могут быть эффективны и давать оптимальные или близкие к оптимальным результаты.

Задание 4

Задание: Реализовать алгоритм Дейкстры на основе реальных данных по теме «Улицы Санкт – Петербурга».

Решение:

Для начала реализуем алгоритм Дейкстры (рис. 6) – алгоритм поиска кратчайшего пути во взвешенном графе. Сам алгоритм реализуем функцией *dijkstra(graph, start)*. Сначала инициализируются переменные *distances* и *visited*. В словаре *distances* для начала инициализируем расстояния от начального узла до других как бесконечность, это необходимо для нахождения в будущем минимального по длине маршрута. Множество *visited* будет содержать вершины, которые мы посетили. Далее, пока длина множества *visited* меньше количества вершин в графе, выбирается текущая вершина как наименьшее расстояние от начальной вершины до вершины, которая еще не была посещена. Затем для каждого соседа текущей вершины пересчитывается расстояние и, если новое расстояние меньше текущего расстояния до соседа, то обновляется значение расстояния до соседа. По окончании работы алгоритма возвращается словарь *distances*, в котором указаны кратчайшие расстояния от начальной вершины до всех других.

```
def dijkstra(graph, start):
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    visited = set()
    while len(visited) < len(graph):
        current_node = min(
            (node for node in graph if node not in visited),
            key=lambda node: distances[node]
        )
        visited.add(current_node)
        for neighbor, weight in graph[current_node].items():
            new_distance = distances[current_node] + weight
            if new_distance < distances[neighbor]:
                distances[neighbor] = new_distance
    return distances
```

Рисунок 6 – Функция, реализующая алгоритм Дейкстры

Протестируем программу на простом примере

```

18 # пример
19 graph = {
20     'A': {'B': 2, 'C': 4},
21     'B': {'A': 2, 'C': 1},
22     'C': {'A': 4, 'B': 1},
23 }
24
25 dijkstra() > while len(visited) < len(graph) >

```

ex4 ×

```

↑
↓
{'A': 0, 'B': 2, 'C': 3}

```

Рисунок 7 – Пример работы программы, реализующей алгоритм Дейкстры

Применим алгоритм к улицам Санкт – Петербурга. Для начала необходимо создать таблицу смежности для улиц Петербурга. Сначала создадим файл, хранящий все улицы Санкт – Петербурга (для этого производился парсинг 2 сайта из списка литературы), после этого с помощью Yandex Geocoding API добавим в файл координаты каждой улицы. По заданным данным с помощью модуля *geopy* рассчитаем расстояния, запишем их в словарь *voc*, а затем сохраним этот словарь в переменную *spb_streets_voc*.

После проведения подготовительных шагов у нас есть таблица смежности, вершинами которой являются улицы Санкт – Петербурга, а весами ребер – расстояния между ними на карте.

Применим к графу реализованный ранее алгоритм Дейкстры, обозначив улицу Ломоносова за стартовую вершину, и выведем некоторые значения на экран. При проверке по картам полученные данные совпадали с реальными с точностью до сотен метров.

```

Садовая --> 1.428216999568983
Восстания --> 1.7568782916702617
Гастелло --> 7.952197612859208
Чайковского --> 2.184059809994296

```

Рисунок 8 – Пример работы программы, реализующей алгоритм Дейкстры
(улицы Санкт – Петербурга)

ЗАКЛЮЧЕНИЕ

В ходе работы были изучены и реализованы в различных задачах жадные алгоритмы.

С помощью жадного алгоритма была реализована задача о выдаче сдачи заданного размера минимальным числом монет. Был сделан вывод о том, что решение этой задачи жадным алгоритмом является удачным – оно всегда выдает оптимальный ответ.

После этого жадным алгоритмом была реализована задача о максимизации цены украденных воров вещей, если число заходов, а также максимальный вес, который вор может вынести за заход, ограничены. В отличие от предыдущей, решение данной задачи не всегда выдавало оптимальный результат. Это случалось из-за большого числа параметров, за которыми необходимо следить и максимизировать их параллельно.

Исходя из этого был сделан вывод о том, что жадные алгоритмы подходят не для всех задач – в некоторых случаях они позволяют получить оптимальное решение, однако не всегда наилучшие решения на каждом шаге приводят к наилучшему конечному результату.

После этого был реализован алгоритм Дейкстры, который производит поиск кратчайшего пути между вершинами взвешенного графа и также относится к жадным алгоритмам. Реализованная программа была протестирована на реальном примере – улицах Санкт – Петербурга с расстояниями между ними на карте. Тестирование было успешно – теоретические значения расстояний (которые выводил навигатор) оказались близки к полученным из алгоритма Дейкстры. Параллельно были изучены модули языка python для работы с картами и географией.

СПИСОК ЛИТЕРАТУРЫ

- 1) Wikipedia. [Жадный алгоритм](#). [Электронный ресурс] – (Дата последнего обращения 19.05.2024);
- 2) Wikipedia. [Алгоритм Дейкстры](#). [Электронный ресурс] – (Дата последнего обращения 19.05.2024).

ПРИЛОЖЕНИЕ

Для удобства все файлы выгружены на GitHub:
<https://github.com/kathykkKk/Algorithms-and-Data-Structures-ICT.git>