МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Жадный алгоритм и A*

Студент гр. 1304 Поршнев Р.А.

Преподаватель Шевелева А.М.

Санкт-Петербург

2023

Цель работы.

Написать программу, которая решает задачу построения пути в ориентированном графе с помощью жадного алгоритма и A*.

Задание.

1. Разработайте программу, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

2. Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A*. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

Основные теоретические положения.

1. Жадный алгоритм реализован с помощью рекурсии. На каждом шаге рекурсии происходит проверка: если путь найден, то нужно выйти из текущего состояния рекурсии. Гарантируется, что первый найденный путь будет самым оптимальным согласно алгоритму в задании, ведь среди всех не посещённых узлов на каждом шаге выбираются в первую очередь узлы с минимальным весом. Также происходит следующая проверка: если текущий узел равен конечному, то следует зафиксировать путь, отметить, что ответ найден, и выйти из текущего состояния рекурсии. Флаг о том, что ответ найден, потребуется для проверки выше. Реализована ещё одна проверка: если из текущего узла не выходят дуги, то следует выйти из текущего состояния рекурсии. После данных проверок всех смежных узлов для текущего узла выбираются не посещённые узлы, а затем сортируются по неубыванию весов. Следующим шагом текущий узел отмечается

посещённым и для каждого отсортированного узла-соседа вызывается рекурсивная функция.

2. Начало работы алгоритма A* выглядит следующим образом: в очередь с приоритетом добавляется старт-узел и его приоритет, равный нулю. Далее инициализируется словарь, в котором в качестве ключа выступает узел, а в качестве значения – минимальный вес рёбер, ведущих к данному узлу. Также инициализируется словарь, в котором в качестве ключа выступает узел, а значение – узел, из которого можно попасть в ключ-узел. Данный словарь нужен для восстановления пути. Далее начинает работу цикл, который работает до тех пор, пока очередь с приоритетом не станет пуста. Внутри цикла происходит извлечение из очереди элемента с наименьшим приоритетом. Затем следует проверка: если текущий узел равен конечному, то цикл останавливается. Затем для каждого узла, смежного с текущим, пересчитывается новое значение веса рёбер, ведущих к смежному узлу. Если смежный узел не находится в очереди или новое значение веса лучше текущего для данного смежного узла, то ему присваивается значение новое значение веса, в словарь для восстановления пути в ключ записывается название смежного узла, а в значение – текущий узел. Затем происходит расчёт приоритета, который равен новому значению веса и эвристической функции. Последним шагом итерации в данном блоке является добавление данного смежного узла и его приоритета в очередь узлов, которые следует рассмотреть. После окончания работы алгоритма остаётся восстановить путь, опираясь на записи в словаре.

Выполнение работы.

1. Для решения поиска пути в орграфе с помощью жадного алгоритма создан класс *Graph*. В конструкторе происходит объявление переменной для сохранения пути, стартового и конечного узла, словаря для хранения посещённых узлов и самого графа, инициализация флага о том, что путь не найден. Реализованы следующие методы:

- *def solution(self)* – данный метод вызывает функционал для считывания исходного графа, поиска пути с помощью жадного алгоритма и печати пути в консоль;
- *def __print_path(self)* – метод для вывода ответа на исходную задачу в консоль;
- *def __read_graph(self)* – данный метод считывает стартовый и конечный узел, а так же весь граф. Граф сохраняется следующим образом: в словарь в качестве ключа записываются все узлы, из которых выходят дуги, а значение для каждого такого ключа – список кортежей, где первый элемент кортежа – название узла, в которую входит дуга из ключа, а второй – вес ребра между ними. Все узлы в контексте посещения устанавливаются в значение *false*;
- *def __get_unexplored_neighbors(self, neighbors_info)* – данный метод предназначен для получения не посещённых узлов, которые смежны рассматриваемому узлу. Входные данные: смежные узлы. Выходные данные: смежные не посещённые узлы;
- *def __greedy_algorithm(self, current_node, path)* – данный метод является реализацией жадного алгоритма. Принцип его работы описан в разделе Основные теоретические положения, п. 1. Входные данные: текущий узел и путь.

Код представлен в Приложении А, greedy.py.

2. Для решения поиска пути в орграфе с помощью жадного алгоритма создан класс *Graph*. В конструкторе происходит объявление переменной для сохранения пути, стартового и конечного узла, словаря для хранения графа. Метод для вывода ответа на исходную задачу такой же, как и в реализации задачи поиска пути жадным алгоритмом, считывание графа в алгоритме A* отличается от реализации в жадном лишь тем, что все узлы в контексте посещения никак не помечаются.

- *def solution(self)* – данный метод вызывает функционал для считывания исходного графа, поиска пути алгоритмом A* и печати пути в консоль;

- *def __heuristic(self, current_node)* – данный метод предназначен для расчёта эвристической функции, которая равна модулю разности кодов символов в таблице ASCII, где в качестве символов выступают буквенные обозначения текущего узла и конечного. Входные данные: буквенное обозначение текущего узла;
- *def __a_star(self)* – данный метод реализует алгоритм A*. Принцип его работы описан в разделе Основные теоретические положения, п. 2;
- *def __recover_path(self)* – данный метод предназначен для восстановления пути по записям из словаря *self.__came_from*.

Код представлен в Приложении А, a_star.py.

Выводы.

В ходе выполнения работы было реализовано два алгоритма, которые ищут минимальный путь в графе между заданными вершинами.

Было реализовано две программы на языке *Python*, каждая из которых ищет минимальный путь в орграфе между заданными узлами. Первая программа ищет минимальный путь с помощью жадного алгоритма, а вторая с помощью алгоритма A*. Жадный алгоритм является рекурсивным и в данной задаче зачастую уступает в оптимальности найденного пути алгоритму A*.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: greedy.py

```
class Graph:
    # инициализация необходимых переменных и структур данных
    def __init__(self):
        self.__path = ''  # для сохранения ответа на задачу
        self.__graph_storage = {}  # структура для хранения графа
        self.__start_node = ''  # старт-узел
        self.__end_node = ''  # финиш-узел
        self.__path_was_found = False
        self.__visited_nodes = {}

    # данный метод запуск логику решения задачи
    def solution(self):
        self.__read_graph()
        self.__greedy_algorithm(self.__start_node, self.__start_node)
        self.__print_path()

    # выход: путь из стартового узла в финишный
    # данцный метод предназачен для вывода ответа на исходную задачу
    def __print_path(self):
        print(self.__path)

    # данный метод предназначен для считывания графа
    def __read_graph(self):
        self.__start_node, self.__end_node = list(input().split())    #
считывание стартового и финишного узла
        while True:
            try:
                node_pair_data = input().split()  # первый узел, второй и
вес дуги между ними
                if node_pair_data[0] not in self.__graph_storage.keys():
# если для первого узла это первая дуга
                    self.__graph_storage[node_pair_data[0]]            =
[(node_pair_data[1], float(node_pair_data[2]))]
                else:  # если для первого узла это не первая дуга

self.__graph_storage[node_pair_data[0]].append((node_pair_data[1],
float(node_pair_data[2])))
                self.__visited_nodes[node_pair_data[0]] = False  # первый
узел ещё не посещён
                self.__visited_nodes[node_pair_data[1]] = False  # как и
второй, это нужно для жадного алгоритма
            except:
                break  # закончить считывание

    # данный метод предназначен для отбора непросмотренных вершин в текущем
состоянии рекурсии
    # входные данные: список смежных вершин с данной
    # выходные данные: список непросмотренных смежных вершин с данной
    def __get_unexplored_neighbors(self, neighbors_info):
        unexplored_neighbors = []
```

```
        for node_info in neighbors_info:
            if not self.__visited_nodes[node_info[0]]:  # если вершина не посещалась ранее
                unexplored_neighbors.append(
                    node_info)  # то добавить её и вес входящего в неё ребра из данной вершины в список
        return unexplored_neighbors

    # данный рекурсивный метод предназначен для поиска пути в графе по заданному жадному алгоритму
    def __greedy_algorithm(self, current_node, path):
        if self.__path_was_found:
            return
        elif current_node == self.__end_node:  # если финишный узел достигнут
            self.__path = path  # записать полученный путь
            self.__path_was_found = True  # отметить, что ответ найдён
            return
        elif current_node not in self.__graph_storage:  # если узел висячий и не финишный узел
            return  # то из него точно не выходят дуги, можно остановить поиск
        unexplored_neighbors = self.__get_unexplored_neighbors(self.__graph_storage[current_node])
        sorted_neighbors_weight = sorted(unexplored_neighbors, key=lambda node_data: node_data[
            1])  # сортировка непосещённых узлов по весам входящих в них дуг из текущей вершины
        self.__visited_nodes[current_node] = True  # пометить текущую вершину, что она посещена
        for node_info in sorted_neighbors_weight:
            self.__greedy_algorithm(node_info[0], path + node_info[0])


if __name__ == "__main__":
    graph = Graph()
    graph.solution()
```

Название файла: a_star.py

```
from queue import PriorityQueue


class Graph:
    # инициализация необходимых переменных и структур данных
    def __init__(self):
        self.__path = ''  # для сохранения ответа на задачу
        self.__graph_storage = {}  # структура для хранения графа
        self.__start_node = ''  # старт-узел
        self.__end_node = ''  # финиш-узел

    # данный метод запуск логику решения задачи
    def solution(self):
        self.__read_graph()
        self.__a_star()
        self.__print_path()

    # данный метод предназачен для вывода ответа на исходную задачу
```

```
    def __print_path(self):
        print(self.__path)

    # данный метод предназначен для считывания графа
    def __read_graph(self):
        self.__start_node, self.__end_node = list(input().split())    #
считывание стартового и финишного узла
        while True:
            try:
                node_pair_data = input().split()  # первый узел, второй и
вес дуги между ними
                if node_pair_data[0] not in self.__graph_storage.keys():
# если для первого узла это первая дуга
                    self.__graph_storage[node_pair_data[0]]             =
[(node_pair_data[1], float(node_pair_data[2]))]
                else:  # если для первого узла это не первая дуга

self.__graph_storage[node_pair_data[0]].append((node_pair_data[1],
float(node_pair_data[2])))
            except:
                break  # закончить считывание

    # входные данные: текущая вершина
    # выходные данные: значение эвристической функции
    # данный метод предназначен для нахождения значения эвристической
функции, которая представляет собой
    # модуль разности ASCII-кодов текущего узла и конечного
    def __heuristic(self, current_node):
        return abs(ord(current_node) - ord(self.__end_node))

    # данный метод предназначен для нахождения минимального по стоимости
пути между старовым и конечным узлом
    # с помощью алгоритма A*
    def __a_star(self):
        open_nodes = PriorityQueue()  # узлы, которые следует рассмотреть
        open_nodes.put((0, self.__start_node))  # начиная с начального
        weight = {self.__start_node: 0}    # ключ - узел, значение -
суммарный вес рёбер, необходимый для достижения узла
        self.__came_from = {
            self.__start_node: None}  # ключ - узел, значение - из какой
вершины можно попасть в ключ - узел
        while not open_nodes.empty():
            current_node_name = open_nodes.get()[1]  # извлечь из очереди
имя самого приоритетного узла
            if current_node_name == self.__end_node:
                break  # конечный узел достигнут
            if current_node_name not in self.__graph_storage:
                self.__graph_storage[current_node_name] = []  # узел без
исходящих из него дуг
            for neighbor_node in self.__graph_storage[current_node_name]:
# для каждого узла смежного с текущим узлом
                neighbor_name = neighbor_node[0]
                weight_between = neighbor_node[1]
                new_weight = weight[current_node_name] + weight_between  #
новый текущий вес для соседнего узла
                if neighbor_name not in weight or new_weight <
weight[neighbor_name]:
                    # если соседний узел ещё не находится в открытом списке
```

```
                    # или вес узла, смежного с текущим, можно уменьшить
                    weight[neighbor_name] = new_weight
                    self.__came_from[neighbor_name] = current_node_name
                    priority = new_weight + self.__heuristic(neighbor_name)
# приоритетность соседнего узла
                    open_nodes.put((priority, neighbor_name))
        self.__recover_path()

    # данный метод предназначен для восстановления пути по записям из
словаря между стартовым и конечным узлом
    def __recover_path(self):
        current_node = self.__end_node
        while current_node:
            self.__path += current_node
            current_node = self.__came_from[current_node]
        self.__path = self.__path[::-1]


if __name__ == "__main__":
    graph = Graph()
    graph.solution()
```