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: Кратчайшие маршруты. Жадный алгоритм и A*

Студент гр. 1384 Усачева Д.В.

Преподаватель Шевелева А. М.

Санкт-Петербург

2023

Цель работы.

Изучить принцип работы алгоритмов поиска кратчайшего пути. Написать две программы, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма и алгоритма 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.

Пример входных данных:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

В первой строке через пробел указываются начальная и конечная вершины.

Далее в каждой строке указываются ребра графа и их вес.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет:

abcde

Задание 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.

Пример входных данных:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

В первой строке через пробел указываются начальная и конечная вершины.

Далее в каждой строке указываются ребра графа и их вес.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет:

ade

Выполнение работы

Для выполнения заданий был создан класс Graph, содержащий поле *info* – словарь, в котором ключом является какая-то вершина-родитель графа, а в качестве значения выступает массив, состоящих из кортежей (пара: вершина-ребенок и вес ребра). Для данного класса был реализован конструктор и метод *insert,* который добавляет новый элемент в граф.

Также для обоих алгоритмов была реализована функция считывания графа из стандартного потока ввода *input_graph.*

Для реализации первого задания была написана программа, которая в качестве ответа выдает строку, состоящую из вершин, из которых строится путь.

Созданы глобальные переменные *graph* (объект класса *Graph*), пустая строка *result* (для дальнейшей записи пути), переменные *start* и *end* (начальная и конечная вершины в пути).

Для поиска минимального пути в графе была реализована рекурсивная функция *greedy_alg*, действующая по принципу поиска с возвратом. В данной функции сначала проводится проверка текущей вершины на наличие вершин-детей и на совпадение с конечной вершиной пути. Начиная с стартовой вершины, ведется поиск вершины-ребенка с минимальной длиной ребра. Если у вершины нет детей, то мы возвращаемся на шаг назад. Функция продолжает рекурсивно вызываться от ребенка, пока мы не дойдем до конечной вершины. В процессе работы алгоритма все посещенные вершины извлекаются из графа.

Для выполнения второго задания так же была написана программа на python, которая с помощью алгоритма A* находит кратчайший путь в графе и выводит полученный результат.

Созданы глобальные переменные *graph* (объект класса *Graph*), пустая строка *result* (для дальнейшей записи пути), переменные *start* и *end* (начальная и конечная вершины в пути) и массив *queue*, содержащий необходимые для рассмотрения вершины.

Реализована *find_min_func* функция для поиска в массиве *queue* вершины с наименьшим значением эвристической функции *func*.

Реализована функция *heuristic_evaluation*, которая нужна для эвристической оценки. В качестве эвристики выступает модуль разности кодов по таблице ASCII рассматриваемой вершины и конечной.

Для поиска минимального пути в графе была реализована функция *alg_A_star*. В ней объявлены переменные *min_path* (ключ — вершина, значение — кратчайшее расстояние от начальной вершины до данной) *func* (ключ — вершина, значение — кортеж: сумма эвристической оценки пути из данной вершины и минимальной длины пути из стартовой вершины в данную, строка, в которой записан самый короткий путь из стартовой вершины в данную). В массив рассматриваемых вершин *queue* добавляется стартовая

вершина. До тех пор, пока массив необходимых вершин не станет пустым, будем искать в этом массиве вершину с наименьшим значением эвристической функции. Затем начинается проход по всем вершинам, в которые можно добраться от текущей, подсчитывается стоимость пути от стартовой вершины до ребенка текущей. Далее проверяется, смотрели ли мы эту вершину или стоимость пути от стартовой до ребенка текущей меньше, чем значение *min_path* по ключу этого ребенка, если одно из этих условий выполняется, в *min_path* заносится значение стоимости пути, подсчитанное ранее, так же обновляются значения *func* для данной вершины, и она добавляется в массив *queue* для дальнейшего поиска пути через ее потомков. Когда текущая вершина будет являться конечной вершиной, алгоритм выведет минимальный пройденный путь, хранящийся в *func* и вернет *True*.

Тестирование

Результаты тестирования представлены в таблице 1.

Таблица 1 — Результаты тестирования

№	Входные данные	Выходные данные	Комментарии
1.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde	Правильный ответ для первой задачи
2.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade	Правильный ответ для второй задачи

Выводы.

Были реализованы жадный алгоритм, жадность которого заключается в том, что на каждом шаге выбирается последняя посещенная вершина, и алгоритм A* для поиска наименьшего по стоимости пути в ориентированном графе. Жадный алгоритм был реализован рекурсивно, а алгоритм A* был стандартно реализован с использованием в качестве эвристической функции расстояние между символами в таблице ASCII. Обе программы успешно прошли все тесты на платформе «Stepik».

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл: greedy_alg.py

```
import sys


class Graph:
    def __init__(self):
        self.info = {}

    def insert(self, parent, child, len_way):
        if not (parent in self.info.keys()):
            self.info[parent] = [(child, len_way)]
        else:
            self.info[parent].append((child, len_way))
        if not (child in self.info.keys()):
            self.info[child] = []


graph = Graph()
start, end = input().split()


def input_graph():
    graph.info[start] = []
    graph.info[end] = []
    for line in sys.stdin:
        try:
            parent, child, len_way = line.split()
        except:
            break
        graph.insert(parent, child, int(float(len_way)))


input_graph()
result = ''
```

```
    def greedy_alg(cur):
        global result
        result += str(cur)
        if cur == end:
            print(result)
            return True
        min_id = 0
        if len(graph.info[cur]) != 0:
            min_len = graph.info[cur][0][1]
            for i in range(len(graph.info[cur])):
                if graph.info[cur][i][1] < min_len:
                    min_len = graph.info[cur][i][1]
                    min_id = i
            cur = graph.info[cur].pop(min_id)[0]
            greedy_alg(cur)
        else:
            graph.info.pop(cur)
            cur = result[-2]
            result = result[:-2]
            greedy_alg(cur)
        return False



greedy_alg(start)
```

Файл: alg_A_star.py

```
import sys
import math


class Graph:
    def __init__(self):
        self.info = {}

    def insert(self, parent, child, len_way):
        if not (parent in self.info.keys()):
            self.info[parent] = [(child, len_way)]
        else:
            self.info[parent].append((child, len_way))
        if not (child in self.info.keys()):
            self.info[child] = []


graph = Graph()
start, end = input().split()
```

```
    def input_graph():
        graph.info[start] = []
        graph.info[end] = []
        for line in sys.stdin:
            try:
                parent, child, len_way = line.split()
            except:
                break
            graph.insert(parent, child, int(float(len_way)))


    input_graph()


    def heuristic_evaluation(cur):
        return abs(ord(end) - ord(cur))


    queue = []


    def find_min_func(func, Q):
        min_f = math.inf
        for edges in Q:
            if func[edges][0] <= min_f:
                min_f = func[edges][0]
                min_edge = edges
        return min_edge


    def alg_A_star():
        global start, end
        queue.append(start)
        func = {start: (0, start)}
        min_path = {start: 0}
        while len(queue) != 0:
            cur = find_min_func(func, queue)
            if cur == end:
                print(func[end][1])
                return True
            queue.remove(cur)
            for node in graph.info[cur]:
                tmp = min_path[cur] + node[1]
                if node[0] not in min_path.keys() or tmp <
min_path[node[0]]:
                    min_path[node[0]] = tmp
                    func[node[0]] = (min_path[node[0]] +
heuristic_evaluation(node[0]), func[cur][1] + node[0])
                    if node[0] not in queue:
                        queue.append(node[0])
        return False


    alg_A_star()
```