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. 0304 Мажуга Д.Р.

Преподаватель Фирсов М.А.

Санкт-Петербург,

2022

Цель работы.

Изучить алгоритмы построения пути в ориентированном взвешенном графе, такие как жадный алгоритм и алгоритм A*. Реализовать данные алгоритмы и сравнить результаты.

Постановка задач.

Вариант 8.

Перед выполнением A* выполнять предобработку графа: для каждой вершины отсортировать список смежных вершин по приоритету.

Задача №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.

Вход:

В первой строке через пробел указываются начальная и конечная вершины. Далее в каждой строке указываются ребра графа и их вес.

Выход:

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.

Пример ввода:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

Пример вывода:

abcde

Задача №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 ввода:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

Пример вывода:

ade

Выполнение работы.

1. **Описание алгоритма**

Задание 1. Жадный алгоритм.

Смысл алгоритма в том, чтобы на каждом шаге выбирать самую “дешевую” вершину, по которой будет совершен переход. Сам алгоритм начинает в заданной стартовой вершине, а заканчивает в заданной конечной вершине. Так же во время работы алгоритма ребра графа, по которым уже был совершен проход, будут добавляться в список использованных. Это необходимо в случае, если граф циклический. Для нахождения пути в графе, в котором есть тупиковые ветки, введем список закрытых вершин и будем добавлять туда вершины, чья ветка ведет в тупик, и при нахождении такой вершины алгоритм

будет возвращаться к предыдущей вершине и переходить по той, которая не является закрытой.

Задание 2. Алгоритм A*

Для нахождения оптимального пути будем использовать очередь с приоритетом, где приоритет рассчитывается по формуле «предполагаемое расстояние + стоимость»(Обычно обозначается как f(x)). Предполагаемое расстояние определяется эвристической функцией (в данном задании разностью ASCII кодов символов), а под стоимостью имеется ввиду текущий кратчайший путь из стартовой вершины до нужной нам вершины x.

Просчитывая f(x) для каждой смежной вершины и не пересчитывая для тех, чья оценка f(x) лучше(меньше) имеющейся, будем переходить по вершинам с наименьшей оценкой. Таким образом будет достигаться финальная вершина.

2. Оценка сложности алгоритма.

Жадный алгоритм

При оценки сложности алгоритма нужно понимать, что возможен случай обхода всех рёбер графа. Таким образом, в худшем случае каждое ребро графа будет пройдено один раз. Необходимо учесть закрытые вершины графа, при попадании в которые необходимо вернуться к прошлой вершине. Таким образом, оценка жадного алгоритма по сложности O(|V| * log|V|). Граф хранится в виде списков смежности, исходя из этого можно дать оценку по памяти равную O(|E|), где E — множество рёбер графа.

A*

Сложность алгоритма A* по количеству операций зависит от эвристической функции.

В худшем случае сложность по количеству операций алгоритма A* будет экспоненциальной. Количество операций для поиска кратчайшего пути до конечной вершины, длина которого составляет D, будет составлять $O(k^D)$ где k — коэффициент ветвления, т.е. среднее количество ребер, исходящих из вершины. Сложность по памяти в таком случае также будет составлять $O(k^D)$.

С оптимальной эвристикой сложность по количеству операций и памяти составляет O(D), если множество рассматриваемых узлов и рёбер является остовным деревом.

Оптимальной эвристикой является эвристика, удовлетворяющая соотношению

$$|h(x) - h*(x)| = O(\log(h*(x))) \quad ,$$

где h(x) — используемая эвристическая функция, h*(x) — «лучшая» эвристика, которая дает точное расстояние от вершины x до конечной вершины.

3. Описание функций и структур данных

1. Класс Graph. Представляет собой граф, который хранит в словаре вершины. В значения хранится список вершин в которые можно попасть из ключа. Класс имеет методы:

add(self, parent: str, child: str, weight: float) – добавляя ребро в граф, строит дерево.
parent – символ родителя текущей вершины.
child – символ ребенка текущей вершины.
weight – вес ребра между вершинами.

get_to_go(self, vertex: str) – возвращает вершины, по которым можно перейти их текущей.
vertex – вершина для которой необходимо найти все исходящие пути.

Find_path(graph: Graph, start: str, finish: str) - метод, находящий путь в графе.
graph - граф, хранящий все вершины поданные на вход
start, finish – начальная и конечная вершины пути

2. Алгоритм A* представлен функцией find_path(graph, start, finish)
graph – граф, хранящий все вершины, поданные на вход
start, finish – начальная и конечная вершины пути

Для хранения вершин используется очередь с приоритетом, в которой в качестве приоритета указана оценка функцией f(x). Стоимости переходов в вершины и пути хранятся в качестве словарей.

Функция вернёт словарь соответствий вершин и как в них можно попасть.

Для получения требуемого результата необходимо восстановить полученный путь.

get_cost(self, parent: str, child: str) – возвращает стоимость ребра между указанными вершинами. Порядок имеет значение.

4. Индивидуальное задание

Перед выполнением A* выполнять предобработку графа: для каждой вершины отсортировать список смежных вершин по приоритету.

Для реализации этого индивидуального задания в функции find_path(self,graph,start,finish) перед началом работы алгоритма A* был дописан цикл, который перебирает всех детей вершин, считает их приоритет, где приоритет рассчитывается по формуле «предполагаемое расстояние + стоимость»(Обычно обозначается как f(x)). Предполагаемое расстояние определяется эвристической функцией (в данном задании разностью ASCII кодов символов). Под стоимостью понимается вес ребра указанный при формировании графа. Сохраняет значения приоритета для каждого из детей в словарь. После этого новый порядок детей записывается в значения для каждой из вершин.

Выводы

Были изучены алгоритмы нахождения кратчайшего пути в взвешенном ориентированном графе — жадный алгоритм и алгоритм A*. Реализации данных алгоритмов были протестированы и исследованы их сложности.

Тестирование 1

№	Входные данные	Вывод программы	Комментарий
1.	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 8.0 f g 1.0	abdeag	OK
2.	a f a b 1.0 b d 2.0 d f 1.0 a c 0.0 c e 1.0 e f 2.0	acef	OK
3.	a b a b 1.0	ab	OK
4.	a c a b 1.0 b c 1.0 a c 3.0	abc	OK

Тестирование 2

№	Входные данные	Вывод программы	Комментарий
1.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade	OK

2.	a f a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 c f 2.0 b g 1.0 d h 1.0 h o 2.0	abcf	OK
3.	a e a c 4 a e 14 b a 7 b d 2 c b 8 c d 6 d a 3 d e 1 e c 13 e b 5	acde	OK
4.	a b a b 1.0	ab	OK
5.	a c a b 1.0 b c 1.0 a c 3.0	abc	OK

Тестирование индивидуального задания

№	Входные данные	Вывод программы	Комментарий
1.	a e a c 4.0 a b 3.0 a d 5.0 c g 3.0 c f 5.0 f e 7.0	Рассмотрим детей вершины a b : 6.0 c : 6.0 d : 6.0 Рассмотрим детей вершины c g : 5.0 f : 6.0 acfe	Одинаковый приоритет, сортируем по дальности сомволов по таблице ascii
2.	a e a c 4.0 a b 4.0 a f 7.0 a e 11.0	Рассмотрим детей вершины a c : 6.0 b : 7.0 f : 8.0 e : 11.0 a e	OK
3.	a e	Рассмотрим детей вершины a	OK

	a c 4.0 a b 4.0 a f 7.0 f g 5.0 f h 8.0 h e 11.0	c : 6.0 b : 7.0 f : 8.0 Рассмотрим детей вершины f g : 7.0 h : 11.0 Рассмотрим детей вершины h e : 11.0 afhe	
4.	a e a f 4.0 a d 7.0 a g 5.0 g c 4.0 g j 7.0 j e 5.0	Рассмотрим детей вершины a f : 5.0 g : 7.0 d : 8.0 Рассмотрим детей вершины g c : 6.0 j : 12.0 Рассмотрим детей вершины j e : 5.0 agje	OK
5.	a j a d 3.0 a h 5.0 a t 7.0 t j 5.0	Рассмотрим детей вершины a h : 7.0 d : 9.0 t : 17.0 Рассмотрим детей вершины t j : 5.0 atj	ok

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл astar.py

```
import sys
import heapq

debug = True


def heuristic(a: str, b: str):
    """Эвристическая функция расстояния между вершинами"""
    return abs(ord(a) - ord(b))


class Vertex:
    """Класс являющийся представлением вершины"""

    def __init__(self, ch, parent, cost):
        self.parents = {parent: cost}  # соответсвтвие родитель :
стоимость
        self.ch = ch  # символ вершины

    def __str__(self):
        s = f"Вершина '{self.ch}', с ребрами от вершин: "
        for k, v in self.parents.items():
            s += f"'{k}' весом {v};"
        return s


class Graph:

    def __init__(self):
        self.graph = {}  # Представление графа словарём вершина : дети

    def add(self, parent: str, child: str, weight: float):
        """Метод добавления вершины в граф
                    weight - вес ребра между вершинами parent и child
        """
        if parent not in self.graph:  # если родитель не в ключах
            self.graph[parent] = []   # добавляем
        if child not in self.graph:  # если ребенок не в ключах
            self.graph[child] = []  # добавляем
        self.graph[parent].append(Vertex(child, parent, weight))
        if debug: print(f"Добавим в граф ребро с весом {weight} между
вершинами '{parent}' и {child}' ")

    def get_to_go(self, vertex):
        """Метод возвращающий детей переданной вершины"""
        return self.graph[vertex]

    def get_cost(self, parent: str, child: str):
        """Метод возвращающий вес ребра от вершины parent к child"""
        for elem in self.graph[parent]:
```

```
            if elem.ch is child:
                return elem.parents[parent]


def find_path(graph, start, finish):
    """Алгоритм A*
    """
    if debug: print("Для нахождения первого оптимального пути будем
использовать очередь с приоритетом\n"
                    "Приорететом служит сумма эврестической оценки
вершины и значение веса ребра для её достижения\n")
    q = []  # очередь с приорететом
    heapq.heappush(q, (0, start))  # Добавляем в очередь вершину старта
    if debug: print("Для начала итерирования алгоритма поместим в очередь
вершину старта")
    came_from = {}  # словарь вершин хранящие в качестве значений вершину
из которой мы попали в вершину ключа
    costs = {}  # словарь для хранения величин оценки стоимости пути до
вершины
    came_from[start] = None
    costs[start] = 0
    while len(q) != 0:  # Повторяем алгоритм или пока очередь не
кончится, или пока не найдём нужную вершину
        current = heapq.heappop(q)[1]  # Берём самую приорететную вершину
        if debug: print(f"Берём из очерди приоритета вершину
'{current}'")

        if current == finish:  # Если она конечная, то заканчиваем
итерацию
            if debug: print(f"Полученная вершина является терминальной\
nПоиск пути окончен")
            break
        if debug: print(f"Для построения пути проверим все вершины в
которые можно попасть из вершины "
                        f"'{current}':\n", *graph.get_to_go(current))
        for next in graph.get_to_go(current):  # Для каждой вершины в
которую можно попасть из текущей
            new_cost = costs[current] + graph.get_cost(current, next.ch)
# Находим стоимость
            if debug: print(f"  До вершины '{next.ch}' можно добраться за
{new_cost}")
            if next.ch not in costs or new_cost < costs[next.ch]:  # если
стоимость меньше или неизвестна
                if debug: print(f"  Полученная стоимость меньше
известной\n"
                                f"  Запомним стоимость\n"
                                f"  Добавим вершину в очередь с
приоритетом с значением приоритета {new_cost + heuristic(finish,
next.ch)} = {new_cost} + {heuristic(finish, next.ch)}\n")
                costs[next.ch] = new_cost  # Добавляем новую стоимость
                priority = new_cost + heuristic(finish, next.ch)  #
вычисляем приоритет
                heapq.heappush(q, (priority, next.ch)) # добавляем
вершину в очередь с приоритетом
                came_from[next.ch] = current  # Дополняем словарь новым
путём в вершну next.ch из current
```

```
    return came_from


if __name__ == "__main__":
    reader = sys.stdin
    start, finish = input().split()
    graph = Graph()
    while True:
        try:
            parent, child, weight = input().split()
        except EOFError as e:
            break
        except ValueError as e:
            break
        if not parent or not child or not weight:
            break
        graph.add(parent, child, float(weight))

    if debug:
        print('Вершины обработаны и добавлены в граф')
        for key in graph.graph.keys():
            print(key, end=':')
            for v in graph.graph[key]:
                print(v.ch, v.parents[key], sep='-', end=' ')
            print()

    res = find_path(graph, start, finish)
    if debug: print(f"\nВозвращаемся по всем пройденным вершинам,
последняя '{finish}'")
    res_string = back = finish
    while back is not start:
        if debug: print(f"В вершину '{back}' пришли из вершины
{res[back]}")
        back = res[back]
        res_string += back
    print(res_string[::-1])
```

Файл greedy.py

```
import sys
from collections import namedtuple

pair = namedtuple('Pair', 'destination weight')

debug = False


class Graph:
    """Класс графа, на котором будет происходить поиск пути"""

    def __init__(self):
        self.graph = {}  # инициализация словаря, который будет
представлять собой граф

    def add(self, parent: str, child: str, weight: float):
        """Метод добавления вершины в граф
            weight - вес ребра между вершинами parent и child
```

```
        """
        if parent not in self.graph:
            self.graph[parent] = []
        if child not in self.graph:
            self.graph[child] = []
        self.graph[parent].append(pair(child, weight))
        if debug: print(f"Добавим в граф вершину '{parent}' с ребром
весом {weight} между вершиной '{child}' ")

    def get_to_go(self, vertex: str):
        """Метод возвращающий детей переданной вершины"""
        return self.graph[vertex]


def find_path(graph: Graph, start: str, finish: str):
    """Жадный алгоритм поиска кратчайшего пути
    """
    closed = set()  # множество вершин в которые алгоритм не будет
просматривать
    used_edges = set()  # ребра по которым алгоритм уже прошел
    lastv = start  # последняя посещённая вершина
    lastv_weight = 0  # вес этой вершины
    path = [start]  # путь найденный алгоритмом

    # ищем вершину finish
    while lastv != finish:
        if debug: print("Текущий путь: ", *path, f"; Последняя посещённая
вершина '{lastv}' с весом ", lastv_weight)
        # получаем вершины в которые можем пойти из последней, в которой
были исключая закрытые
        to_go = set(graph.get_to_go(lastv)).difference(closed)
        if debug: print("Из неё можно добраться до вершин ",
*set(graph.get_to_go(lastv)).difference(closed))
        # проверка, что из вершины можно перейти куда-то
        if to_go:
            current_vertex, current_vertex_weight = min(to_go, key=lambda
vertex: (vertex[1], vertex[0]))
            if debug: print(f"Доступная вершина с минимальным весом
'{current_vertex}', её вес {current_vertex_weight}")
            # берём из списка вершину, в которую можно попасть по
минимальному ребру
            if (lastv, current_vertex) in used_edges:  # если ребро уже
использовалось
                if debug: print(f"Ребро от вершина {lastv} к вершине
{current_vertex} уже было использовано")
                # добавляем ребро в список закрытых
                closed.add((current_vertex, current_vertex_weight))
            else:  # если ребро используется впервые
                # добавляем вершину в путь, ребро в список использованных
                if debug: print(
                    f"Ребро '{lastv}---{current_vertex}' ранее не
использовалось, добавляем '{current_vertex}' в путь\nДобавляем ребро в
список использованных\n")
                path.append(current_vertex)
                used_edges.add((lastv, current_vertex))
                lastv = current_vertex  # меняем последнюю посещённую
вершину
```

```
                lastv_weight = current_vertex_weight  # меняем её вес
        else:
            # Вершина является листом
            if debug: print(f"Вершина '{lastv}' является листом, из неё
некуда пойти, возвращаемся к вершине '{path[-1]}'\n"
                            f"Заносим текущую вершину в список закрытых\
n")
            closed.add((lastv, lastv_weight))  # добавляем лист в список
закрытых вершин
            path.pop()  # убираем её из пути
            lastv = path[-1]  # возвращаемся в предыдущей вершине
        if debug: print("Путь найден:")
    return path


if __name__ == "__main__":
    reader = sys.stdin
    start, finish = input().split()
    graph = Graph()
    while True:
        try:
            parent, child, weight = input().split()
        except EOFError as e:
            break
        except ValueError as e:
            break
        if not parent or not child or not weight:
            break
        graph.add(parent, child, float(weight))
    if debug:
        print('Вершины обработаны и добавлены в граф')
        for key in graph.graph.keys():
            print(key, end=':')
            for elem in graph.graph[key]:
                print(elem.destination, elem.weight, sep='-', end=' ')
            print()
    print(*find_path(graph, start, finish), sep='')
```

Файл astar_ind.py

```
import sys
import heapq

debug = True


def heuristic(a: str, b: str):
    """Эвристическая функция расстояния между вершинами"""
    return abs(ord(a) - ord(b))


class Vertex:
    """Класс являющийся представлением вершины"""

    def __init__(self, ch, parent, cost):
```

```
        self.parents = {parent: cost}  # соответсвтвие родитель :
стоимость
        self.ch = ch  # символ вершины

    def __str__(self):
        s = f"Вершина '{self.ch}', с ребрами от вершин: "
        for k, v in self.parents.items():
            s += f"'{k}' весом {v};"
        return s


class Graph:

    def __init__(self):
        self.graph = {}  # Представление графа словарём вершина : дети

    def add(self, parent: str, child: str, weight: float):
        """Метод добавления вершины в граф
                    weight - вес ребра между вершинами parent и child
        """
        if parent not in self.graph:  # если родитель не в ключах
            self.graph[parent] = []   # добавляем
        if child not in self.graph:  # если ребенок не в ключах
            self.graph[child] = []  # добавляем
        self.graph[parent].append(Vertex(child, parent, weight))
        if debug: print(f"Добавим в граф ребро с весом {weight} между
вершинами '{parent}' и {child}' ")

    def get_to_go(self, vertex):
        """Метод возвращающий детей переданной вершины"""
        return self.graph[vertex]

    def get_cost(self, parent: str, child: str):
        """Метод возвращающий вес ребра от вершины parent к child"""
        for elem in self.graph[parent]:
            if elem.ch is child:
                return elem.parents[parent]


def find_path(graph, start, finish):
    """Алгоритм A*
    """
     print()
        print("Перед началом работы алгоритма отсортируем всех детей
вершин в соответсвии с их приоритетом")
        dics = {}
        for vertexes in self.graph:
            print()
            print("Рассмотрим детей вершины",vertexes)
            for children in self.graph[vertexes]:
                #print(vertexes,end='          ')
                #print(children.ch,children.parents[vertexes])
                dics[children.ch] = children.parents[vertexes] +
heuristic(finish,children.ch)
            if(len(dics) != 0):
                print("Неотсорированные дети этой вершины и их
приоритет:",end=' ')
```

```
                self.print_dict(dics)
            else:
                print("Данная вершина детей не имеет")
            count = 0
            sorted_values = sorted(dics.values())  # Sort the values
            sorted_dict = {}
            dics_copy = copy.deepcopy(dics)
            for i in sorted_values:
                a = min(dics_copy.items(),key= lambda x : (x[1], x[0]) )
                sorted_dict[a[0]] = a[1]
                dics_copy[a[0]] = 1000
                count += 1
                print("Вершина",a[0],"с приоритет",i,"будет
являться",count,"по счету в отсортированном списке детей для данной
вершины")
                continue
            if(len(sorted_dict) != 0):
                print("Итого имеем:",end=' ')
                self.print_dict(sorted_dict)
            else:
                pass
            self.graph[vertexes].clear()
            for i in sorted_dict.keys():
                self.graph[vertexes].append(Vertex(i,vertexes,dics[i]-
heuristic(finish,i)))
            dics.clear()
        print("Все вершины были отсортированны")
    if debug: print("Для нахождения первого оптимального пути будем
использовать очередь с приоритетом\n"
                    "Приорететом служит сумма эврестической оценки
вершины и значение веса ребра для её достижения\n")
    q = []  # очередь с приорететом
    heapq.heappush(q, (0, start))  # Добавляем в очередь вершину старта
    if debug: print("Для начала итерирования алгоритма поместим в очередь
вершину старта")
    came_from = {}  # словарь вершин хранящие в качестве значений вершину
из которой мы попали в вершину ключа
    costs = {}  # словарь для хранения величин оценки стоимости пути до
вершины
    came_from[start] = None
    costs[start] = 0
    while len(q) != 0:  # Повторяем алгоритм или пока очередь не
кончится, или пока не найдём нужную вершину
        current = heapq.heappop(q)[1]  # Берём самую приорететную вершину
        if debug: print(f"Берём из очерди приоритета вершину
'{current}'")

        if current == finish:  # Если она конечная, то заканчиваем
итерацию
            if debug: print(f"Полученная вершина является терминальной\
nПоиск пути окончен")
            break
        if debug: print(f"Для построения пути проверим все вершины в
которые можно попасть из вершины "
                        f"'{current}':\n", *graph.get_to_go(current))
        for next in graph.get_to_go(current):  # Для каждой вершины в
которую можно попасть из текущей
```

```
            new_cost = costs[current] + graph.get_cost(current, next.ch)
# Находим стоимость
            if debug: print(f"  До вершины '{next.ch}' можно добраться за
{new_cost}")
            if next.ch not in costs or new_cost < costs[next.ch]:  # если
стоимость меньше или неизвестна
                if debug: print(f"  Полученная стоимость меньше
известной\n"
                                f"  Запомним стоимость\n"
                                f"  Добавим вершину в очередь с
приоритетом с значением приоритета {new_cost + heuristic(finish,
next.ch)} = {new_cost} + {heuristic(finish, next.ch)}\n")
                costs[next.ch] = new_cost  # Добавляем новую стоимость
                priority = new_cost + heuristic(finish, next.ch)  #
вычисляем приоритет
                heapq.heappush(q, (priority, next.ch)) # добавляем
вершину в очередь с приоритетом
                came_from[next.ch] = current  # Дополняем словарь новым
путём в вершну next.ch из current

    return came_from


if __name__ == "__main__":
    reader = sys.stdin
    start, finish = input().split()
    graph = Graph()
    while True:
        try:
            parent, child, weight = input().split()
        except EOFError as e:
            break
        except ValueError as e:
            break
        if not parent or not child or not weight:
            break
        graph.add(parent, child, float(weight))

    if debug:
        print('Вершины обработаны и добавлены в граф')
        for key in graph.graph.keys():
            print(key, end=':')
            for v in graph.graph[key]:
                print(v.ch, v.parents[key], sep='-', end=' ')
            print()

    res = find_path(graph, start, finish)
    if debug: print(f"\nВозвращаемся по всем пройденным вершинам,
последняя '{finish}'")
    res_string = back = finish
    while back is not start:
        if debug: print(f"В вершину '{back}' пришли из вершины
{res[back]}")
        back = res[back]
        res_string += back
    print(res_string[::-1])
```