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. 9383 _____ Звега А.Р.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучение и практическое освоение жадного и a* алгоритмов.

Задание.

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

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

Задание (Вариант 2).

В A* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных. A* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных.

Выполнение работы.

Для реализации жадного алгоритма была выбрана рекурсия. Этот алгоритм на каждом шаге выбирает наименьший путь и помечает текущий узел как посещенный. Так происходит пока алгоритм не дойдет до конечного узла. Если алгоритм не находит путь, в который можно пойти, то он возвращается на предыдущий узел.

Для реализации A* алгоритма выбран итеративный метод. Этот алгоритм, учитывает по мимо веса - эвристику (модуль разности кодов имени текущего узла от имени конечного узла). A* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный.

Описание общих функций:

create_list_vertex(list_vertex, graph) – создает список вершин и список путей.

create_matrix(matrix_graph, list_vertex, graph) – создает матрицу смежности вершин.

Описание функций жадного алгоритма:

clear(from_a, to_b) – убирает путь(из а в б) из матрицы смежности.

greed_sort(current, prev, prev_ans, ans) – жадный алгоритм, описан выше.

Описание функций A* алгоритма:

heuristic(a, b) – вычисляет эвристику.

add_heuristic(list_vertex) – добавляет собственные эвристики для каждого узла.

a_star_search(graph, start, goal) – A* алгоритм, описан выше.

class **PriorityQueue** – класс очереди, нужен для функционирования алгоритма A*.

Описание тестов:

Тест 1 – проверяет функцию **create_matrix,** так же имеется проверка на некорректные данные.

Тест 2 – проверяет функцию **a_star_search,** так же имеется проверка на некорректные данные.

Тесты проверяют корректность выходных данных из функции.

Таблица 1. Результаты тестирования A* алгоритма

Ввод	Вывод
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	ag
b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	be

Таблица 2. Результаты тестирования жадного алгоритма

Ввод	Вывод
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
b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	bge

Анализ работы алгоритма.

Алгоритм является оптимальным, когда он находит кратчайший путь до конечной вершины.

Жадный алгоритм не является оптимальным, так как у него нет никаких проверок. Если есть несколько путей до конечного узла, то алгоритм выберет путь, у которого первый шаг минимальный, но последующие шаги могут быть в разы затратнее, чем у других путей.

A* имеет более равномерный проход по графу, так как используется очередь, из которой берется узел с наименьшим приоритетом (стоимость пути до узла + эвристика). Этот алгоритм вычисляет не «лучший маршрут», а «лучший маршрут для заданной эвристики», поэтому чтобы получить минимальный маршрут, нужно задать правильную эвристику, в нашем случае алгоритм не является оптимальным.

Выводы.

При выполнении работы были освоены жадный и A* алгоритмы. Была реализована программа, которая позволяет определить путь в графе от начального узла до конечного.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД.

Название файла a_star_task.py

```
import heapq



class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return len(self.elements) == 0

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]



def heuristic(a, b):
    return abs(ord(b)-ord(a))



def a_star_search(graph, start, goal):
    priority = PriorityQueue()
    priority.put(start, 0)
    cost = dict()
```

```
    cost[start] = 0.0
    came_from = dict()
    came_from[start] = None

    while priority:
        current = priority.get()
        if current == goal:
            break
        index = list_vertex.index(current)
        current_index = -1
        for i in graph[index]:
            current_index += 1
            if not i:
                continue
            new_cost = cost[current]+i
            if  list_vertex[current_index]  not  in  cost  or  new_cost  <
cost[list_vertex[current_index]]:
                cost[list_vertex[current_index]] = new_cost
                priority_next           =           new_cost           +
list_heuristic[list_vertex[current_index]]
                # heuristic(list_vertex[current_index], goal) main
                # list_heuristic[list_vertex[current_index]]task
                priority.put(list_vertex[current_index], priority_next)
                came_from[list_vertex[current_index]] =  current
    return came_from


def create_list_vertex(list_vertex, graph):
    while True:
```

```
        try:
            add_way = input().split()
        except EOFError:
            break
        if not add_way:
            break
        add_way[2] = float(add_way[2])
        graph += [add_way]
        skip1 = 1
        skip2 = 1
        if list_vertex:
            for i in list_vertex:
                if add_way[0] == i:
                    skip1 = 0
                if add_way[1] == i:
                    skip2 = 0
            if skip1:
                list_vertex += add_way[0]
            if skip2:
                list_vertex += add_way[1]
        else:
            list_vertex += add_way[0]
            if add_way[0] != add_way[1]:
                list_vertex += add_way[1]
    list_vertex.sort()
    return list_vertex, graph


def create_matrix(matrix_graph, list_vertex, graph):
```

```
    for i in range(len(list_vertex)):
        add_matrix = list()
        for j in range(len(list_vertex)):
            add_matrix += [0]
        for g in graph:
            if g[0] == list_vertex[i]:
                add_matrix[list_vertex.index(g[1])] = g[2]
        matrix_graph += [add_matrix]
    return matrix_graph


def add_heuristic(list_vertex):
    list_heuristic = dict()
    for i in list_vertex:
        print('heuristic ', i, ':')
        list_heuristic[i] = abs(int(input()))
    return list_heuristic


if __name__ == '__main__':
    way = input().split(' ')
    graph = list()
    matrix_graph = list()
    list_vertex = list()

    vertex_and_graph = create_list_vertex(list_vertex, graph)
    list_vertex = vertex_and_graph[0]
    graph = vertex_and_graph[1]
```

```
matrix_graph = create_matrix(matrix_graph, list_vertex, graph)


index_exit = list_vertex.index(way[1])


list_heuristic = add_heuristic(list_vertex)


ans = a_star_search(matrix_graph, way[0], way[1])


add = way[1]
string = add


while add != way[0]:
    add = ans[add]
    string += add


print(string[::-1])
```