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: Алгоритмы поиска пути в графах

Студентка гр. 9383	____________________	Карпекина А.А.
Преподаватель	____________________	Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Ознакомиться с Жадным алгоритмом и алгоритмом A* и научиться применять их на практике. Написать программу реализующую поиск пути в ориентированном графе Жадным алгоритмом и алгоритмом A*.

Постановка задачи.

1) **Жадный алгоритм.**

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 посещенная вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещенной 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

2) Алгоритм А*.

Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом А*. 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

Вариант 8.

Вывод графического представления графа.

Описание алгоритмов.

1) Жадный алгоритм:

Работа алгоритма начинается в начальной вершине и переходит из нее на вершину с минимальным весом ребра. Данный процесс продолжается, пока не будет найдена конечная вершина и возможно осуществление перехода. Иначе последний переход отменяется и выбирается следующая по минимальности веса ребра вершина. Для избежания циклов все пройденные вершины отмечаются. Также записывается выбранный путь и его вес.

В худшем случае сложность алгоритма будет равна:

O(N*M), где N – количество вершин, M – количество ребер.

Так как необходимо будет пройти все вершины и все ребра графа.

Для хранения графа используется список смежности, поэтому сложность по памяти будет равна:

O(N+M), где N – количество вершин, M – количество рёбер.

2) Алгоритм A*:

В упорядоченную очередь записываются варианты продолжения имеющихся путей с учетом эвристической функции. Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины из начальной, и функции эвристической оценки расстояния от рассматриваемой вершины к конечной. Если возможно сокращение получившегося пути, в очередь добавляются вершины, рассматриваемые при переходе в новую. Выбирается вершина из верха очереди и продолжается соответствующий путь, пока не будет достигнута конечная вершина.

В лучшем случае сложность алгоритма будет равна:

O((N+M), где N – количество вершин, M – количество ребер

Так как эвристическая функция будет правильно выбирать путь до следующей вершины.

Временная сложность алгоритма A* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растет экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(log h^*(x))$$

где h* — оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка *h(x)* не должна расти быстрее, чем логарифм от оптимальной эвристики.

В худшем случае сложность по памяти:

$O(N * (N + M))$, где N – количество вершин, M – количество ребер.

Так как, алгоритму приходится помнить экспоненциальное количество узлов.

Описание структур и функций.

Для работы с графом был создан класс Graph. Поля класса и методы класса:

- self.graph - поле хранит список смежности графа, тип dict
- def __init__(self) - конструктор класса Graph
- def append_edge(self, vertex_out, vertex_in, weight) - метод создает словарь смежности графа
- def greedy(self, start_vertex, end_vertex) - метод, в котором реализован Жадный алгоритм, возвращает список вершин, задающих искомый путь
- def a_star(self, start_vertex, end_vertex) - метод, в котором реализован алгоритм A*, возвращает список вершин, задающих искомый путь
- def draw(self) - метод выводит графическое представление графа

Реализованные функции:

- def read_list() - функция считывает данные в консоли и формирует из них список
- def fill_graph(empty_graph) - с помощью метода append_edge из заданного списка заполняется self.graph

Тестирование.

Таблица 1 - результаты тестирования для Жадного алгоритма

Тест	Входные данные	Ответ Жадного алгоритма	Ответ алгоритма A*
№1	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde	ade
№2	a e a b 7.0 a c 3.0 b c 1.0 c d 8.0 b e 4.0	abe	abe
№3	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0 c m 1.0 m n 1.0	abdefg	ag

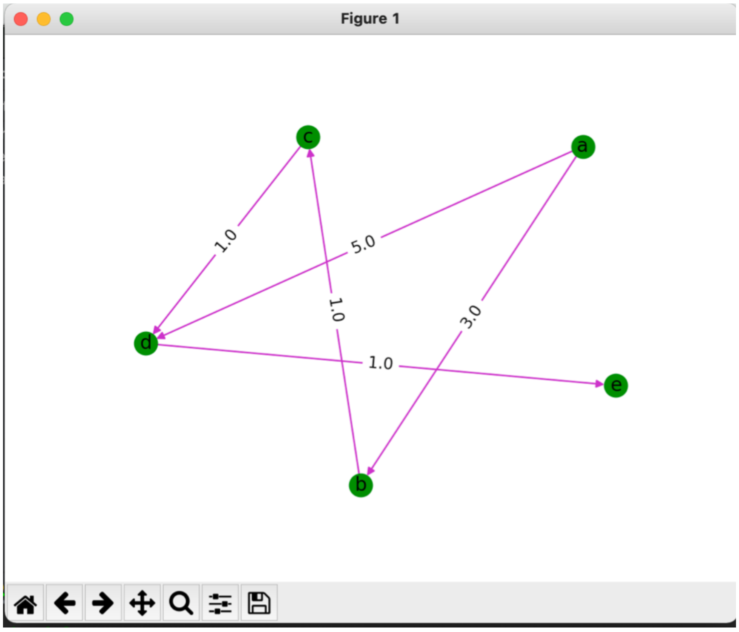

Рисунок 1 - Изображение графа для теста №1

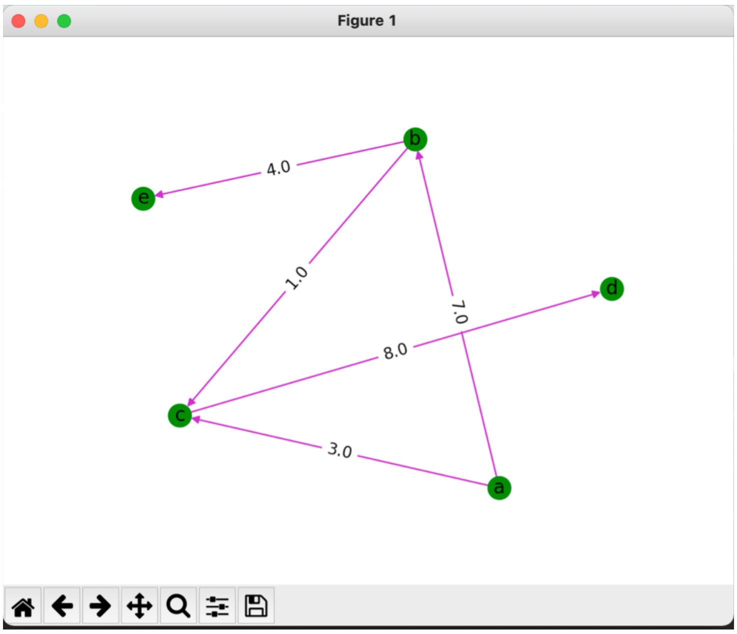

Рисунок 2 - Изображение графа для теста №2

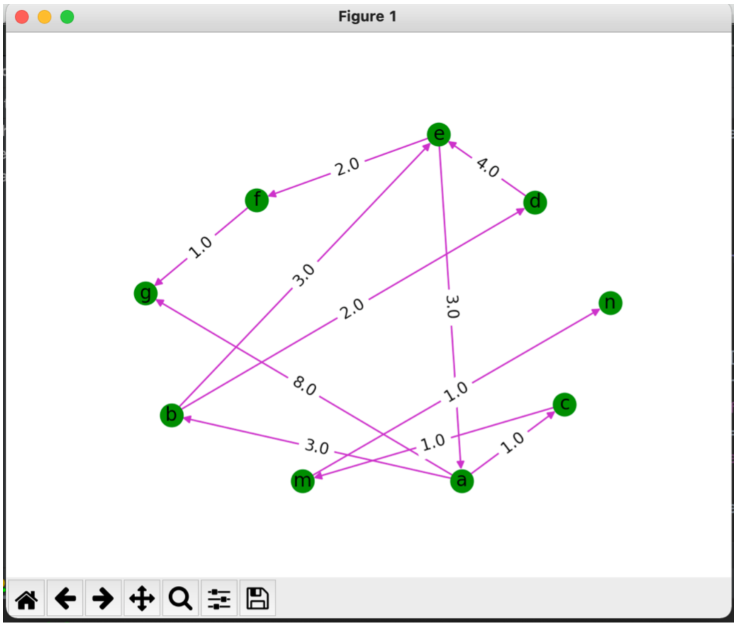

Рисунок 3 - Изображение графа для теста №3

Вывод.

В ходе выполнения лабораторной работы были изучены и реализованы два алгоритма для поиска пути в ориентированном графе: Жадный алгоритм и алгоритм A*. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, A* отличает то, что при выборе вершины он учитывает, весь пройденный до неё путь, поэтому найденный путь будет минимальным возможным. Был реализован вывод графического представления графа.

ПРИЛОЖЕНИЕ А
Исходный код программы

Название файла: lb2_greedy+a_star.py

```
import networkx as nx
from pylab import show as graph_show

from operator import itemgetter
from math import inf
import sys



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

    def append_edge(self, vertex_out, vertex_in, weight):
        if vertex_out not in self.graph:
            self.graph[vertex_out] = {}
        self.graph[vertex_out][vertex_in] = weight

    def greedy(self, start_vertex, end_vertex):
        processed_vertex = []
        while True:
            el = start_vertex
            answer = []
            while el in self.graph and any(self.graph[el]):
                answer.append(el)
                min_size = inf
                next_vertex = None
                for vertex in self.graph[el]:
```

```
                if self.graph[el][vertex] < min_size and vertex not in
processed_vertex:
                    if vertex in self.graph or vertex == end_vertex:
                        next_vertex = vertex
                        min_size = self.graph[el][vertex]
            el = next_vertex
            processed_vertex.append(el)
            if el == end_vertex:
                answer.append(el)
            if answer[-1] == end_vertex:
                return answer


    def a_star(self, start_vertex, end_vertex):
        answer = {}
        vertex_queue = [(start_vertex, 0)]
        vertex_queue.sort(key=itemgetter(1), reverse=True)
        vec = [start_vertex]
        answer[start_vertex] = (vec, 0)
        while not vertex_queue == []:
            if vertex_queue[-1][0] == end_vertex:
                return answer[end_vertex][0]
            top_queue = vertex_queue[-1]
            vertex_queue.pop()
            if top_queue[0] in self.graph:
                for i in list(self.graph[top_queue[0]].keys()):
                    cur_size = self.graph[top_queue[0]][i] + answer[top_queue[0]][1]
                    if i not in answer or answer[i][1] > cur_size:
                        cur_path = []
                        for j in answer[top_queue[0]][0]:
                            cur_path.append(j)
```

```
                cur_path.append(i)
                answer[i] = (cur_path, cur_size)
                vertex_queue.append((i, abs(ord(end_vertex) - ord(i)) +
answer[i][1]))
                vertex_queue.sort(key=itemgetter(1), reverse=True)
        return answer[end_vertex][0]

    def draw(self):
        g = nx.DiGraph()

        for i in self.graph:
            for j in self.graph[i]:
                g.add_edges_from([(i, j)], weight=self.graph[i][j])

        edge_labels = dict([((u, v,), d['weight'])
                            for u, v, d in g.edges(data=True)])
        pos = nx.spring_layout(g, scale=100, k=100)
        nx.draw_networkx_edge_labels(g, pos, edge_labels=edge_labels)
        nx.draw(g, pos, node_size=200, with_labels=True, node_color='g',
edge_color='m')
        graph_show()


def read_list():
    for line in sys.stdin:
        line = line.rstrip()
        if line:
            yield line
        else:
            break
```

```
def fill_graph(empty_graph):
    input_list = list(read_list())
    for el in range(len(input_list)):
        input_list[el] = input_list[el].split(" ")
        if el > 0:
            empty_graph.append_edge(input_list[el][0], input_list[el][1],
float(input_list[el][2]))

    start_node = input_list[0][0]
    end_node = input_list[0][1]
    return start_node, end_node


if __name__ == '__main__':

    graph = Graph()
    filled_graph = fill_graph(graph)
    path_greedy = graph.greedy(filled_graph[0], filled_graph[1])
    path_a_star = graph.a_star(filled_graph[0], filled_graph[1])

    print("\nGreedy: ", end='')
    for p in path_greedy:
        print(p, end='')

    print("\nA*: ", end='')
    for p in path_a_star:
        print(p, end='')

    graph.draw()
```