**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 _____ Байков Е.С.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

**Цель работы.**

Изучение жадного алгоритма и алгоритма A*.

**Задание.**

Для жадного алгоритма:

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 дешевым из последней посещенной вершины. Каждая вершина в графе имеет буквенное обозначение («a», «b», «c»…), каждое ребро имеет неотрицательный вес.

Для алгоритма A*:

Разработайте программу, которая решает задачу построения 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.

Входные и выходные данные:

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.

**Основные теоретические положения.**

Жадный алгоритм — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Алгоритм A* — в информатике и математике, алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как f(x)). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как g(x) и может быть, как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).

**Выполнение работы.**

В ходе выполнения лабораторной работы был создан класс *Graph*, в котором и осуществляется основная реализация алгоритмов. Были описаны методы, выполняющие требуемые в задании алгоритмы, – жадный алгоритм и A*.

**Структура**

Далее описаны классы, методы и функции используемые в коде программы для решения поставленной задачи.

1) Класс *Graph*: данный класс хранит в себе стартовый и конечный узел пути, а также информацию о дугах графа в виде словаря, где ключом является начальный узел дуги, а в качестве значения список списков из двух элементов. Первый элемент внутреннего списка – конечный узел, а второй элемент – вес дуги. Внутри класса реализованы методы: *get_neighbours*, *heuristic_function*, *a_star_algorithm*, *greedy_algorithm* и *__init__*.

2) Метод *get_neighbours*: принимает на вход узел графа и возвращает список из смежных ему узлов и весов соответствующих ребер. Если же переданный узел является листом, то возвращаемым значением будет None.

3) Метод *heuristic_function*: эвристическая функция, которая возвращает разницу между значениями ASCII кодов переданного в нее и конечного узлов. Метод необходим для реализации алгоритма A*.

4) Метод *a_star_algorithm*: реализует алгоритм поиска кратчайшего пути A*. Внутри метода созданы открытое и закрытое множества (*open_set*, *closed_set*), в первом находится стартовый узел. Затем были созданы два словаря *distance* – хранит в себе пары узел (ключ) и длину пути до стартовой вершины (значение), и *adjacent_nodes*, хранящий в себе узел назначения в качестве ключа и стартовый узел (откуда осуществлен переход в узел назначения) в качестве значения. После начинается цикл *while*, который длится до тех пор, пока в открытом множестве есть узлы требующие проверки. Внутри цикла происходит поиск минимального (по сумме эвристики и расстояния) узла из открытого множества. Если вдруг такого узла не существует, то пути не существует и метод возвращает *None*. Затем идет проверка на то, не является ли выбранный узел финишным узлом. При достижении финиша происходит проход по ключам и значениям списка смежных узлов, по которым строится путь от финиша к старту, а затем возвращается инвертированный путь, т.е. от старта к финишу. В ином случае происходит проверка списка смежных узлов с помощью метода *get_neighbours*. При отсутствии смежных узлов из открытого множества узел удаляется и добавляется в закрытый, чтобы его больше не просматривать, затем осуществляется переход к следующей итерации. При наличии происходит просмотр всех узлов (*node*) и весов (*weight*), которые хранятся в списке *neighbours*. Если *node* не находится в открытом и закрытом списках, значит данный узел не просмотрен и должен быть добавлен в открытое множество, а также обновлены значения в словарях *adjacent_nodes* и *distance*. Если значение *distance* от узла в открытом или закрытом множестве больше, чем значение расстояния текущего узла и добавляемого веса, то значения обновляются, т.е. расстояние *node* становится равным значению расстояния текущего узла и добавляемого веса, а также происходит перенаправление пути из *current_node* в *node* (*adjacent_nodes[node] = current_node*) если же *node* находится в закрытом списке, то из него происходит удаление этого узла и добавление его в открытое множество. После цикла текущий узел удаляется из открытого множества и

добавляется в закрытое. Если верхний цикл завершился, а путь не был найден, то возвращается *None*.

5) Метод *greedy_algorithm*: для каждого ключа из набора дуг списки-значения сортируются по весу дуги. Заводится переменная *path* равная стартовому узлу. До тех пор, пока конечная вершина в пути не будет равна финишной, будет добавляться первый узел из набора смежных узлов, при условии того что он уже не находится в пути. Если данный узел уже лежит в пути, то он извлекается из набора всех дуг, для того чтобы удалить цикличность. Метод возвращает значение переменной *path*.

6) Метод *__init__*: конструктор класса, принимает набор ребер в виде словаря, стартовый и финишный узлы, а также инициализирует данные поля класса.

7) Функция *solve*: функция считывает стартовую и конечную позицию, а также заполняется словарь с ребрами *arcs_kit*. Был создан объект типа *Graph*, в который передаются словарь ребер, стартовый узел и финишный узел. Печатает результат методов *greedy_algorithm* и *a_star_algorithm*.

8) Функция *main*: вызывает функцию *solve*.

Разработанный программный код смотреть в приложении А.

**Выводы.**

Была разработана программа, реализующая алгоритмы – жадный и А*, для поиска кратчайшего пути в графе. Успешно пройдены тесты на платформе *Stepik*. Оба алгоритма представлены в виде методов класса *Graph*. Жадный алгоритм реализован итеративно и был немного оптимизирован путем сортировки набора смежных узлов. В алгоритме А* использована эвристическая функция, которая была приведена на платформе *Stepik* – разность символов, текущего и конечного узлов.

# ПРИЛОЖЕНИЕ А
# ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
# Graph class stores in itself start and end nodes and kit of arcs
class Graph:
    # Graph constructor that creates an object of the Graph type
    def __init__(self, arcs_kit, start_node, finish_node) -> None:
        self.start_node = start_node
        self.arcs_kit = arcs_kit
        self.finish_node = finish_node

    # method returns list which includes adjacent nodes and weight
of arc
    def get_neighbours(self, node):
        return self.arcs_kit[node] if node in self.arcs_kit else None

    # method that returns heuristic approximation
    def heuristic_function(self, node):
        return ord(self.finish_node) - ord(node)

    # method implements A* algorithm
    def a_star_algorithm(self):
        open_set = set(self.start_node) # stores nodes requiring
consideration
        closed_set = set() # stores checked nodes

        distance = {} # stores node as key and distance to the start
as value
        distance[self.start_node] = 0

        adjacent_nodes = {} # stores destination node as key and
start node as value
        adjacent_nodes[self.start_node] = self.start_node

        while len(open_set) > 0:
            current_node = None

            for node in open_set:
                if current_node == None or\
                    distance[node] + self.heuristic_function(node)
<\
                    distance[current_node] +
self.heuristic_function(current_node):
                    current_node = node

            if not current_node:
                return None

            if current_node == self.finish_node:
                reverse_path = ''
                while adjacent_nodes[current_node] != current_node:
                    reverse_path += current_node
                    current_node = adjacent_nodes[current_node]
```

```
                    reverse_path += current_node
                    return reverse_path[::-1]

                neighbours = self.get_neighbours(current_node)
                if neighbours == None:
                    open_set.remove(current_node)
                    closed_set.add(current_node)
                    continue

                for (node, weight) in neighbours:
                    if node not in open_set and node not in closed_set:
                        open_set.add(node)
                        adjacent_nodes[node] = current_node
                        distance[node] = distance[current_node] + weight
                    elif  distance[node]  >  distance[current_node]  +
weight:
                        distance[node] = distance[current_node] + weight
                        adjacent_nodes[node] = current_node
                        if node in closed_set:
                            closed_set.remove(node)
                            open_set.add(node)
                open_set.remove(current_node)
                closed_set.add(current_node)

            return None

        # method implements greedy algorithm
        def greedy_algorithm(self):
            for node in self.arcs_kit:
                self.arcs_kit[node].sort(key=lambda x: x[1])

            path = self.start_node
            while path[-1] != self.finish_node:
                current_last_node = path[-1]
                if   self.arcs_kit[current_last_node][0][0]   not   in
self.arcs_kit \
                    and    self.arcs_kit[current_last_node][0][0]    !=
self.finish_node:
                    self.arcs_kit[current_last_node].pop(0)
                path += self.arcs_kit[current_last_node][0][0]
            return path

    # function that reads input data and creates the Graph object
    # then it outputs the returned values of the algorithms
    def solve():
        start_node, finish_node = input().split()
        arcs_kit = {}

        while True:
            try:
                arc = input().split()
                if arc[0] in arcs_kit:
                    arcs_kit[arc[0]].append([arc[1], float(arc[2])])
                else:
                    arcs_kit[arc[0]] = [[arc[1], float(arc[2])]]
            except:
                break
```

```
    graph = Graph(arcs_kit, start_node, finish_node)
    print(graph.greedy_algorithm())
    print(graph.a_star_algorithm())

def main():
    solve()

main()
```