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*.

Студентка гр. 1304 Чернякова А.Д.

Преподаватель Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучение жадных алгоритмов, их сравнение с эвристическими алгоритмами, а также решение задачи поиска кратчайшего пути двумя способами: жадным алгоритмом и алгоритмом 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 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.
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.

Выполнение работы.

Первая задача.

- Класс *Way* - класс пути, который хранит стоимость пути до вершины и саму вершину, к которой ведет данный путь. У данного класса конструктор *__init__ (self, weight, vertex)* принимает на вход стоимость пути и вершину, к которой ведет путь; методы *get_weight(self)* и *get_vertex(self)* возвращают поля класса *self.weight* и *self.vertex* соответственно.
- Функция *inputs()* - функция считывания путей графа(на вход ничего не принимает). В данной функции создается переменная *graph* - словарь вида *vertex : [Way(), Way()...]*. Считывание продолжается до тех пор пока не будет введена пустая строка. Каждая строка при считывании делится на три параметра *first_vert* - вершина, из которой идет путь, *second_vert* - вершина, в

которую идет путь, и *weight* - стоимость (вес) пути, соединяющего данные вершины. Если считывать нечего, то происходит выход из функции и возврат словаря графа. В противном случае идет проверка, есть ли ключ *first_vertex* в словаре, если да, то сохраняем список значений данного ключа в переменную ways, добавляем в список значений еще один путь *Way(float(weight), second_vert)* и присваиваем ключу *first_vertex* новый список значений *ways*, если в словаре нет такого ключа, то просто добавляем в словарь ключ и его значение.

- Функция *is_min_vertex(vertex, graph)* - функция поиска самого дешевого пути для вершины *vertex* в графе *graph*, которая принимает на вход сам словарь графа и вершину, из которой ищем самый выгодный путь. В данной функции в переменную ways сохраняются все пути из вершины *vertex*, если путей вообще не было или были тупиковые, то возвращается сама вершина, если были, то начинается алгоритм поиска самого дешевого пути из данной вершины. Переменной *min_way* (она будет хранить самый выгодный путь) мы присваиваем значение самого первого пути из вершины *vertex*. Переменной *min_vertex* (она будет хранить вершину, к которой ведет самый выгодный путь), присваиваем самую первую вершину, к которой ведет первый путь. Затем проходимся по каждому пути из списка *ways* и ищем самый дешевый, если такой находится, то меняем значения *min_way* и *min_vertex*. Функция возвращает *min_vert*.

- Функция *main()* - главная функция, ничего не принимает на вход. Изначально считываются начальная и конечная вершины и сохраняются в переменные *first_vertex* и *last_vertex* соответственно. Вызывается функция inputs() и ее результат сохраняется в переменную graph. vert - текущая переменная, изначально ей присваивается значение начальной вершины *first_vertex. result* - список вершин итогового пути, изначально содержит одну начальную вершину. Далее идет реализация жадного алгоритма. Текущая вершина передается одним из параметров в функцию *is_min_vertex(vert, graph)* и если функция возвращает эту же вершину, которую и приняла, значит пути из данной вершины в графе нет. Удаляем добавленную вершину из итогового

списка *result*, получаем пути для последней вершины итогового списка (необходимо удалить путь, который ведет в тупиковую вершину), далее с посощью цикл *for* ищем это тупиковый путь, когда находим, удаляем путь и выходим из цикла. Присваиваем графу по данному ключу новый список путей (без тупикового) и текущей вершиной делаем последний элемент итогового списка пути. Если же вершин не оказалась тупиковой, то текущей вершине vert присваивается результат функции *is_min_vertex(vert, graph)* и добавляем текущую вершину *vert* в итоговый список *result*. Так продолжается поиск пути пока текущая вершина не станет последней. В конце функция выводит итоговый список в консоль.

Вторая задача.

- Класс *Vertex* - класс вершины графа, конструктор которого *__init__ (self, vertex)* принимает на вход название вершины, инициализирует поля self.vertex - название вершины, self.ways - список исходящих ребер для вершины (изначально пустой), self.h - эвристика (изначально равна 0), self.g - стоимость пути от начала до этой вершины, self.prev - вершина, из которой пришли в данную (изначально пустая строка)
- Класс Graph - класс графа, конструктор которого *__init__ (self, first_vertex, last_vertex)* принимает на вход начальную и конечную вершины, инициализирует поля *self.first_vertex* - начальная вершина, *self.last_vertex* - конечная вершина, а также *self.vertexes* - словарь всех вершин вида *name:Vertex()* (изначально пустой), *self.viewed* - список вершин, до которых уже найден оптимальный путь, *self.queue* - просматриваемые вершины (очередь с приоритетом). У данного класса реализованы методы *insert(self, way), a_star(self), solve(self), __str__(self).*
- Метод *insert(self, way)* - метод вставки вершин и ребер в граф, принимает на вход путь *way* - список из трех элементов (вершина, из которой идет путь, вершина, в которую идет путь, стоимость пути, соединяющего вершины). Затем идет проверка двух вершин на наличие их в словаре графа,

если их там нет, то они добавляются. А также добавляется в список исходящих путей для первой вершины вторая вершина. Ничего не возвращает.

- Метод *a_star(self)* - метод, который реализует алгоритм *A** (ничего не принимает на вход и ничего не возвращает). Изначально переменным *h* и *g* присваиваются соответствующие поля начальной вершины. Создается переменная текущей вершины *vert* - список, первый элемент которого результат сложения *h*+*g* (в будущем приоритет), второй элемент является начальной вершиной. Далее *vert* добавляется в очередь с приоритетом и начинается обход графа. Когда текущая вершина окажется равной конечной вершине, то происходит выход из цикла. В противном случае извлекается вершина с самым высоким приоритетом, затем она добавляется в просмотренные, а далее происходит обход соседей текущей извлеченной вершины. Если найден более дешевый путь (то есть *g* меньше), то текущая вершина добавляется в поле *prev* для следующей, обновляется поле *g* у вершины (изначально выбранное большое число заменяется реальным), переменным *h* и *g* присваиваются соответствующие поля вершины и элемент добавляется в очередь с приоритетом. Обход графа происходит до тех пор, пока очередь не станет пустой.
- Метод *solve(self)* - метод, который вызывается в функции *main()* для решения задачи (ничего не принимает и ничего не возвращает). Для всех ключей словаря рассчитывается эвристика (заполняются поля *h*). Для первой вершины поле *g* приравнивается к 0, так как вершина начальная и путь до нее 0. Вызывается метод *a_star(self)*, реализующий основной алгоритм.
- Метод *__str__(self)* - метод, который выводит итоговый путь *res* (изначально пустая строка) от начальной вершины до конечной. Текущей вершиной *vert* назначается конечная вершина. К итоговому пути добавляется конечная вершина, текущей вершине присваивается поле prev конечной, так строка будет заполняться до тех пор, пока поле *prev* не будет пустой строкой (а это только у начальной вершины). Так будет найден путь с конца, поэтому метод

вернет строку *res* в обратном порядке, чтобы получился путь от начальной вершины к конечной.

Вне классов реализовано две функции *inputs()* и *main()*.

- Функция *inputs()* - функция считывания путей графа(на вход ничего не принимает). В данной функции изначально считывается начальная и конечная вершины, создается объект класса *Graph*. Далее каждая строка при считывании делится на три параметра вершина, из которой идет путь, вершина, в которую идет путь, и - стоимость (вес) пути, соединяющего данные вершины. Если считывать нечего, то происходит выход из функции и возврат графа. В противном случае стоимость пути приводится к типу *float* и список из трех параметров передается в метод *insert()*.
- Функция *main()* - на вход ничего не принимает и ничего не возвращает, вызывает функцию считывания inputs(), функцию solve() и выводит итоговый путь в консоль.

Исходный код программы представлен в Приложении А.

Выводы.

Были изучены основные алгоритмы на графах, такие как A* и жадный алгоритм, на платформе *Stepik* обе задачи успешно прошли все тесты, а значит алгоритмы были реализованы верно. При сравнении двух алгоритмов было замечено и подтверждено на практике, что жадный алгоритм, выбирая локально лучший результат не всегда вычисляет глобально лучшее решения, поэтому более точным и верным будет алгоритм A*, так как он еще использует и эвристику.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: greed.py

```
class Way:  # класс пути, хранит стоимость пути до вершины и саму вершину
    def __init__(self, weight, vertex):
        self.weight = weight
        self.vertex = vertex

    def get_weight(self):
        return self.weight

    def get_vertex(self):
        return self.vertex


def inputs():  # функция считывания входных параметров
    graph = {}  # словарь графа вида vertex : [Way(), Way()...]
    while True:
        try:
             first_vert, second_vert, weight = input().split()  # считывание из какой вершины путь, в какую вершину и стоимость пути
            if not first_vert:
                return graph
             if graph.get(first_vert):  # если в словаре уже есть такой ключ
                ways = graph.get(first_vert)
                ways.append(Way(float(weight), second_vert))
                graph[first_vert] = ways
            else:  # если в словаре нет такого ключа
                        graph[first_vert] = [Way(float(weight), second_vert)]
        except (ValueError, EOFError):
            return graph


def is_min_vertex(vertex, graph):    # функция поиска самого дешевого пути для вершины vertex в графе graph
    ways = graph.get(vertex)
     if ways is None or ways == []:  # если путей вообще не было или были тупиковые, то возвращается сама вершина vertex
        return vertex
    min_way = ways[0].get_weight()
    min_vert = ways[0].get_vertex()
     for way in ways:  # рассмотрение всех путей из вершины vertex и поиск самого дешевого
        if way.get_weight() < min_way:
            min_way = way.get_weight()
```

```
            min_vert = way.get_vertex()
    return min_vert


def main():
      first_vertex, last_vertex = input().split()   # считывание
начальной и конечной вершин
    graph = inputs()
    vert = first_vertex  # текущая вершина
    result = [first_vertex]
    while vert != last_vertex:
         if vert == is_min_vertex(vert, graph):  # если  пути из
данной вершины в графе нет
            result.pop()
            ways = graph.get(result[-1])
            for i in range(len(ways)):
                if ways[i].get_vertex() == vert:
                    ways.pop(i)
                    break
            graph[result[-1]] = ways
            vert = result[-1]
        else:
            vert = is_min_vertex(vert, graph)
            result.append(vert)
    print("".join(result))


if __name__ == "__main__":
    main()
```

Название файла: a_star.py

```
from queue import PriorityQueue

class Vertex:  # класс вершины графа
    def __init__(self, vertex):
        self.vertex = vertex
        self.ways = []  # список исходящих ребер из вершины
        self.h = 0  # эвристика
        self.g = 10000000  # стоимость пути от начала до вершины
        self.prev = ''


class Graph:  # класс графа
    def __init__(self, first_vertex, last_vertex):
        self.first_vertex = first_vertex
        self.last_vertex = last_vertex
             self.vertexes = {}   # словарь всех вершин вида
name:Vertex()
```

```
        self.viewed = []  # список вершин, до которых уже найден оптимальный путь
        self.queue = PriorityQueue()  # просматриваемые вершины (очередь с приоритетом)

    def insert(self, way):  # добавление вершин и ребер в граф
        for i in range(2):
            if way[i] not in self.vertexes.keys():
                self.vertexes[way[i]] = Vertex(way[i])
        self.vertexes[way[0]].ways.append((way[1], way[2]))  # добавление второй вершины в список исходящих путей для первой

    def a_star(self):  # алгоритмом A*
        h = self.vertexes[self.first_vertex].h
        g = self.vertexes[self.first_vertex].g
        vert = [h + g, self.first_vertex]
        self.queue.put(vert)  # добавление начальной вершины в очередь
        while not self.queue.empty():  # обход графа
            if vert[1] == self.last_vertex:
                break
            vert = self.queue.get()  # извлечение вершины с самым высоким приоритетом
            self.viewed.append(vert)
                for way in self.vertexes[vert[1]].ways:  # обход соседей текущей извлеченной вершиной
                    if self.vertexes[vert[1]].g + way[1] < self.vertexes[way[0]].g:  # если найден более дешевый путь
                        self.vertexes[way[0]].prev = vert[1]
                        self.vertexes[way[0]].g = self.vertexes[vert[1]].g + way[1]
                        h = self.vertexes[way[0]].h
                        g = self.vertexes[way[0]].g
                        self.queue.put([h + g, way[0]])

    def solve(self):  # метод, вызывающийся в функции main() для решения задачи
        for elem in self.vertexes.keys():  # для всех ключей словаря рассчитывается эвристика
            self.vertexes[elem].h = abs(ord(elem) - ord(self.last_vertex))
        self.vertexes[self.first_vertex].g = 0  # для первой вершины функция g равна 0
        self.a_star()

    def __str__(self):  # вывод итогового пути
        res = ""
        vert = self.vertexes[self.last_vertex].vertex
        while vert != '':
```

```
            res += vert
            vert = self.vertexes[vert].prev
        return res[::-1]


def inputs():   # функция считывания входных параметров
    first_vertex, last_vertex = input().split()
    graph = Graph(first_vertex, last_vertex)
    while True:
        try:
            data = input().split()   # считывание из какой вершины
путь, в какую вершину и стоимость пути
            if data == []:
                return graph
            data[2] = float(data[2])
            graph.insert(data)  # вставка пути в граф
        except (ValueError, EOFError):
            return graph


def main():
    graph = inputs()
    graph.solve()
    print(graph)


if __name__ == '__main__':
    main()
```