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 Кардаш Я.Е.

Преподаватель Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучить принципы работы жадного алгоритма и алгоритма A* для ориентированных взвешенных графов. Применить алгоритмы на практике, решив задачу построения пути с наименьшей стоимостью ребер.

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.

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:

Abcde (жадный алгоритм)

Ade (A*)

Выполнение работы.

Для большего удобства и понимания алгоритмы реализованы в разных файлах, однако некоторые конструкции совпадают.

В main происходит вызов метода, который считывает данные, затем создание объекта класса Solve, вызов метода, запускающего решение и вывод результата.

Метод input_data() считывает введенные пользователем данные, заносит их в созданный объект класса Solve и возвращает этот объект.

Для прохождения проверок тестирующей системой импортирован stdin библиотеки sys.

Реализация жадного алгоритма.

Для хранения данных о вершине создан класс Node. При создании объекта инициализируются следующие поля:

Name – Название вершины.

Ways – список, каждый элемент которого – кортеж из названия вершины, в которую можно попасть из текущей вершины, а также из длины ребра до нее.

Хранение графа и основная логика алгоритма происходит в классе Solve.

Поля класса:

Start – Вершина, из которой алгоритм начинает работу.

Finish – Вершина, до которой считается оптимальный путь.

Nodes – словарь, где ключи – названия вершин, а значения по ключам – объекты класса Node.

Way – итоговый путь в формате вывода.

Viewed – список уже просмотренных вершин (для корректной работы алгоритма в некоторых случаях).

Для работы алгоритма реализованы следующие методы:

Insert_way(way) – записывает в словарь nodes поступившую от пользователя информацию о графе. Ничего не возвращает.

Solve() - запускает работу алгоритма. Ничего не принимает и не возвращает.

__greedy_search(cur,way) – в этом методе происходит работа жадного алгоритма по поиску пути. Рекурсивно происходит перемещение к той непросмотренной вершине, путь до которой минимален. Если алгоритм заходит в тупик, происходит возврат на верхние уровни рекурсии, а при достижении конечной вершины путь сохраняется в поле класса и алгоритм завершается.

__find_min_way(node) – метод вызывается из метода __greedy_search. В нем происходит нахождение такой непросмотренной вершины, путь до которой из текущего узла минимален.

__str()__ - преобразует вывод класса к необходимому по условию задачи.

Реализация алгоритма A*

Для реализации алгоритма A* импортирован класс PriorityOueue библиотеки queue.

Для алгоритма A* использован тот же метод ввода, в main происходит те же действия, однако некоторые поля и методы классов Node и Solve изменены.

Класс Node:

Кроме уже описанных в работе жадного алгоритма полей name и way добавлены следующие поля:

H – эвристика, необходимая для работы A*.

G – Путь от начального узла (до работы алгоритма устанавливается в большое число, затем улучшается).

Prev – название вершины, из которой проложен путь.

В классе Solve оставлены поля start и finish начальной и конечной вершины, а также поле словаря с вершинами nodes и поле просмотренных

вершин viewed. Также оставлены методы заполнения графа insert_way(way), метод solve(), запускающий решение задачи (изменено только название вызываемого из него метода) и метод __str__(), формирующий выходную строку (реализация формирования немного изменилась, но логика осталась та же).

Добавленное поле:

Queue – Приоритетная очередь, в которую помещаются просматриваемые вершины до нахождения оптимального пути к ним.

Добавленный метод:

__astar_search() – метод, реализующий алгоритм A*. Принцип работы алгоритма следующий: в приоритетную очередь добавляется сумма пройденного пути и эвристики, а также название вершины. Сумма в данном случае является показателем приоритета для извлечения из очереди. Далее в цикле из очереди извлекается вершина с минимальным приоритетом, эта вершина помещается в просмотренные, а для всех смежных с ней вершин пересчитывается g. Если посчитанное расстояние до вершины меньше текущего, то поле g и prev узла перезаписываются. Затем каждая из таких смежных вершин помещается в очередь (для нее уже посчитана функция g, а ранее задана эвристика). Таким образом при достижении конечной вершины в каждом узле будет записано, из какого узла до него был построен оптимальный путь. Собрав такие пути от конечного, до начального можно получить требуемый результат.

Исходный код обеих программ находится в приложении А.

Выводы.

Изучены и применены на практике жадный алгоритм на графах и алгоритм A*. Реализованы задачи поиска минимального по весу ребер пути в ориентированном графе с помощью этих алгоритмов.

Жадный алгоритм реализован на основе обхода графа в глубину вплоть до достижение конечной вершины. Алгоритм A* реализован с помощью обхода в глубину и приоритетной очереди, а в качестве эвристического расстояния использована разность между ASCII значениями названий вершин.

Оба алгоритма успешно прошли все тесты проверяющей системы Stepik.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
#импорт потока ввода stdin
#для корректного ввода для stepik
from sys import stdin

#класс вершины графа
class Node:
    def __init__(self, node_name):
        self.node_name = node_name #название вершины
        self.out_edges = [] #исходящие ребра

#Класс ориентированного графа, решающего задачу
class Solve:
    def __init__(self, start_node, finish_node):
        self.start_node = start_node #начальная вершина
        self.finish_node = finish_node #конечная вершина
        self.nodes_dict = {} #словарь всех вершин (name:node)
        self.final_way = "" #конечный путь
        self.viewed_nodes = [] #просмотренные вершины

    #Заполнение графа вершинами и ребрами
    def insert_way(self,way):
        for i in range(2):
            if way[i] not in self.nodes_dict.keys():
                self.nodes_dict[way[i]] = Node(way[i])
        self.nodes_dict[way[0]].out_edges.append((way[1], way[2]))

    # метод, вызывающийся для решения задачи
    def solve(self):
        for elems in self.nodes_dict.keys():
            self.nodes_dict[elems].out_edges.sort()
        current_way = ''
        self.__greedy_search(self.start_node,current_way)
```

```
    # Жадный алгоритм поиска пути
    def __greedy_search(self, current_node, current_way):
        if current_node == self.finish_node or self.finish_node in
self.final_way:
            current_way+=self.finish_node
            self.final_way = current_way
            return
        #выбрать минимальную непросмотренную вершину
        # и переместиться к ней
        current_way += current_node
        while self.finish_node not in self.final_way:
            min_way = self.__find_min_way(self.nodes_dict[current_node])
            if min_way in self.nodes_dict.keys():
                self.viewed_nodes.append(min_way)
                self.__greedy_search(min_way, current_way)
            else:
                return

    # Нахождение минимальной вершины для жадного алгоритма
    def __find_min_way(self,node):
        if node.out_edges == []:
            return ''
        min_way = 1000
        min_name = ''
        for elems in node.out_edges:
            if elems[0] in self.viewed_nodes:
                continue
            if elems[1]<min_way:
                min_name = elems[0]
                min_way = elems[1]
        return min_name
    #Формат для вывода
    def __str__(self):
        return self.final_way

#ввод данных для построения графа
def input_data():
    start_node_name, finish_node_name = input().split()
    graph = Solve(start_node_name, finish_node_name)
```

```
    input_edges = ' '
    for input_edges in stdin:
        input_edges = input_edges.split()
        if input_edges == []:
            break
        input_edges[2] = float(input_edges[2])
        graph.insert_way(input_edges)
    return graph

if __name__ == '__main__':
    graph = input_data()
    graph.solve()
    print(graph)
    pass
```

Название файла: main_2.py

```
#импорт потока ввода stdin
#для корректного ввода для stepik
from sys import stdin
#импорт очереди с приоритетом
#для построения алгоритма A*
from queue import PriorityQueue
#класс вершины графа
class Node:
    def __init__(self, node_name):
        self.node_name  = node_name #название вершины
        self.out_edges = [] #исходящие ребра
        self.heuristic = 0 #эвристика
        self.cost_path = 100000 #+Б - пока не найден путь до вершины
        self.prev_node = '' #откуда попал в вершину

#Класс ориентированного графа, решающего задачу
class Solve:
    def __init__(self,start,finish):
        self.start_node = start #начальная вершина
        self.finish_node = finish #конечная вершина
```

```
        self.nodes = {} #словарь всех вершин (name:node)
        self.viewed = [] #вершины, до которых найден оптимальный путь
        self.queue = PriorityQueue() #просматриваемые вершины

    #Заполнение графа вершинами и ребрами
    def insert_way(self,way):
        for i in range(2):
            if way[i] not in self.nodes.keys():
                self.nodes[way[i]] = Node(way[i])
        self.nodes[way[0]].out_edges.append((way[1], way[2]))

    #метод, вызывающийся для решения задачи
    def solve(self):
        for elems in self.nodes.keys():
            self.nodes[elems].out_edges.sort()
            self.nodes[elems].heuristic = abs(ord(elems) - ord(self.finish_node))
        self.nodes[self.start_node].cost_path = 0
        self.__astar_seach()

    #Поиск пути алгоритмом A*
    def __astar_seach(self):
        #поместить start в очередь
        h = self.nodes[self.start_node].heuristic
        g = self.nodes[self.start_node].cost_path
        current_node_elem = [h+g,self.start_node]
        self.queue.put(current_node_elem)
        #обход графа
        while not self.queue.empty():
            if current_node_elem[1] == self.finish_node:
                break
            current_node_elem = self.queue.get() #извлечение элемента с самым высоким
приоритетом
            self.viewed.append(current_node_elem)
            #обход по смежным с текущей извлеченной вершиной
            for way in self.nodes[current_node_elem[1]].out_edges:
```

```
                if self.nodes[current_node_elem[1]].cost_path + way[1] < self.nodes[way[0]].cost_path:
                    self.nodes[way[0]].prev_node = current_node_elem[1]
                    self.nodes[way[0]].cost_path = self.nodes[current_node_elem[1]].cost_path + way[1]
                    h = self.nodes[way[0]].heuristic
                    g = self.nodes[way[0]].cost_path
                    self.queue.put([h+g,way[0]])

    #Формат для вывода
    def __str__(self):
        result = ""
        current_node = self.nodes[self.finish_node].node_name
        while current_node!='':
            result+=current_node
            current_node = self.nodes[current_node].prev_node
        return result[::-1]

#ввод данных для построения графа
def input_data():
    start_node_name, finish_node_name = input().split()
    graph = Solve(start_node_name, finish_node_name)
    input_edges = ' '
    for input_edges in stdin:
        input_edges = input_edges.split()
        if input_edges == []:
            break
        input_edges[2] = float(input_edges[2])
        graph.insert_way(input_edges)
    return graph

if __name__ == '__main__':
    graph = input_data()
    graph.solve()
    print(graph)
    pass
```