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 В.А.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучение алгоритма поиска с возвратом и реализация с его помощи программы, которая решает задачу размещения квадратов на столе.

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 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.

Для алгоритма A*:

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.

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 (англ. Greedy algorithm) — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Поиск A* (произносится «А звезда» или «А стар», от англ. A star) — 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.

Классы, методы классов и функции, используемые в программном коде для решения поставленной задачи:

1) Класс *OrientedGraph()* – класс, хранящий в себе информацию об ориентированном графе, а именно начальная вершина *start_node*, конечная вершина *end_node*, само представление графа *graph_paths*, также методы работы с графом. Переменная *graph_paths* представляет собой словарь, где ключами являются вершины, из которых можно добраться до других. По ключу хранится список списков, где элемент нулевого индекса – название вершины, в которую можно попасть, элемент с индексом единица – расстояние между этой вершиной и вершиной «ключом».

2) Метод класса *def __init__(self, start_node, end_node, graph_paths)* – конструктор класса. Этот метод вызывается при создании экземпляра класса. Он принимает ключевое слово *self* в качестве первого аргумента, который позволяет получить доступ к атрибутам или методу класса. На вход методу также подаются основные атрибуты класса – стартовая вершина, конечная вершина и представление графа. Используется для инициализации атрибутов класса.

3) Метод класса *def sorted_graph(self)* – метод сортировки вершин по их отдаленности от исходной. Принимает в качестве аргумента экземпляр класса. Так как в жадном алгоритме при переходе из одной вершины в другую всегда будет выбираться та, до которой расстояние меньше, для ускорения работы алгоритма необходима сортировка. По каждому ключу список списков в словаре-графе располагается по увеличению значения расстояния, которое хранится под первым индексом *self.graph_paths[key].sort(key=lambda x:x[1])*.

4) Метод класса *def greedy_algorithm(self)* – реализация жадного алгоритма. Принимает в качестве аргумента экземпляр класса. Перед началом работы основного алгоритма для ускорения применяется описанный выше метод *self.sorted_graph()*. Переменной *start* присваивается значение *self.start_node* – начальная вершина, из которой ищется путь. Переменная *answer* предназначена для вывода корректного ответа, значение данной переменной изначально равно *start*. Основной алгоритм: запускается цикл *while*, который работает до тех пор, пока *start* не станет равной *self.end_node* – вершине, в которую необходимо попасть. Так как суть жадного алгоритма в данном случае заключается в том, чтобы из последней посещенной вершины переходить в ту, путь до которой является самым дешевым, переменной *next_node* присваивается значение *self.graph_paths[start][0][0]*. Так как по каждому ключу список списков отсортирован в порядке увеличения длины, соответственно вершина, в которую надо попасть, располагаются по значению ключа *start*, в списке под индексом *0*, а в самом внутреннем списке название вершины также хранится под нулевым индексом. В случае если данная вершина не находится в ключах словаря, реализующего граф, *next_node not in self.graph_paths.keys()* и при этом она не является конечной *next_node != self.end_node*, то есть вершина является тупиковой, то ее надо удалить *self.graph_paths[start].pop(0)*. Иначе переменная *start* теперь становится *next_node*, а к переменной *answer* добавляется название вершины, из которой пришли *answer += start*. Метод возвращает переменную *answer*, в которой

хранятся вершины, по которым необходимо пройти от начальной вершины до конечной, то есть путь.

5) Метод класса *def heuristic_function(self, node)* – эвристическая функция для работы алгоритма A*. Принимает в качестве аргументов экземпляр класса и вершину. Возвращает близость символов, обозначающих вершины графа, а именно конечную и любую другую, в таблице ASCII.

6) Метод класса *def Astar_answer(self, node_from)* – вывод корректного ответа для работы алгоритма A* в соответствии с заданием. Принимает в качестве аргументов экземпляр класса и словарь ребер, использованных в пути от начальной до конечной вершины. Переменной *answer* присваивается значение вершины, в которую необходимо прийти, *answer = self.end_node*. Так как *node_from* – словарь, в котором хранятся по очереди вершины, проходимые в ориентированном графе, запущен цикл *while* который работает до тех пор, пока не будут пройдены все значения по следующему ключу словаря *node_from[answer[0]]*. К переменной *answer* прибавляется значение, хранящееся по данному ключу. Метод возвращает переменную *answer*, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

7) Метод класса *def Astar_algorithm(self)* – реализация алгоритма A* для поиска пути в ориентированном графе. Принимает на вход экземпляр класса. Перед началом работы алгоритма необходимо обработать граф. Создается словарь *node_cost*, в котором будет хранится оценка, «стоимость» для каждой вершины. Для начальной вершины *self.start_node* эта оценка равно нулю. Для реализации алгоритма был подключен модуль *heapq,* который обеспечивает реализацию очереди кучи, также известного как алгоритм очереди приоритетов. Создается список, в котором будет хранится куча с приоритетом, *pqueue = []*. Функция *heappush()* модуля *heapq* добавляет значение элемента – кортежа *(0, self.start_node)* в кучу *pqueue*, сохраняя инвариант кучи. Словарь *node_from* реализован следующим образом: по ключу, которым является вершина, хранится та вершина, из которой в вершину-ключ произошел переход. Основная

работа алгоритма – цикл *while*, работающий до тех пор, пока куча *pqueue* не пуста. Переменной *current* присваивается значение вершины, хранящейся в *heapq.heappop(pqueue)[1]*, так как она находится на вершине кучи, значит, приоритет у нее самый высокий. Осуществляется проверка *current == self.end_node*, не произошло ли попадание в конечную вершину, если да, цикл завершает работу с помощью *break*. Иначе алгоритм продолжает работу. Так как из текущей вершины необходимо идти дальше, проверка: находится ли она в ключах словаря ребер графа *if current in self.graph_paths.keys()*. В таком случае просматриваются все элементы *node*, хранящиеся по данному ключу в графе *self.graph_paths[current]*. *Node* – список, где по нулевому индексу хранится название вершины, по первому индексу расстояние до вершины. В переменной *cost* рассчитывается стоимость данной вершины, которая складывается из стоимости вершины, из которой в нее перешли, *node_cost[current]* и расстояния до этой вершины, которое хранится соответственно в *node[1]*. Далее осуществляется проверка: нет ли данной вершина в словаре стоимости вершин *node[0] not in node_cost* или же новая рассчитанная стоимость меньше, чем уже находится в словаре *cost < node_cost[node[0]]*. При выполнении одного из условий по ключу *node[0]* в словаре *node_cost* присваивается значение *cost*. В переменной *priority* рассчитывается приоритет для данной вершины, он вычисляется как сумма стоимости *cost* и значение эвристической функции *self.heuristic_function()* описанной выше, ей в качестве аргументов передается соответственно *node[0]*. Функция *heappush()* модуля *heapq* добавляет значение элемента-кортежа *(priority, node[0])* в кучу *pqueue*, сохраняя инвариант кучи. Словарю *node_from*, в котором хранятся вершины, из которых был осуществлен переход, по ключу *node[0]* значение становится *current*. Метод возвращает результат работы еще одного метода класса *self.Astar_answer(node_from)*, а именно ответ на задачу – путь из начальной вершины в конечную. В качестве аргумента методу передается словарь, по которому составляется цепочка перехода вершин из одной в другую.

8) Функция *def reading_edges()* – чтение ребер графа и их веса. Создается словарь *graph_paths = dict()*, где ключ вершина, по ней хранится список списков. Внутри каждого списка по нулевому индексу вершина, в которую можно попасть, по первому индексу – расстояние. В процессе считывания с клавиатуры словарь обновляется. К вершине либо дописываются имеющиеся ребра, либо вершинка как новая появляется в словаре. Метод возвращает словарь ребер и их веса.

9) Функция *def solution()* – запуск решения поставленной задачи. Считываются начальная и конечная вершины *start_node, end_node = input().split(),* переменной *graph_paths* присваивается результат работы функции *reading_edges()* – словарь ребер с весами. Создается переменная *graph* – объект класса *OrientedGraph* от параметров *start_node, end_node, graph_paths*. Для вывода ответа с помощью функции *print()* при работе жадного алгоритма вызывается метод класса *graph.greedy_algorithm()*, для алгоритма A* *graph.Astar_algorithm()*.

10) Условие *if __name__ == "__main__"* – проверка, был ли файл запущен напрямую. Запуск программы.

Разработанный программный код смотреть в приложении А.

Выводы.

В ходе лабораторной работы были изучены жадный алгоритм и алгоритм A* для нахождения пути в графе. На языке программирования *Python* реализован класс, представляющий собой ориентированный граф. Внутри класса реализованы методы, описывающие каждый из алгоритмов.

Для ускорения работы жадного алгоритма словарь, в котором хранились ребра графа и их вес, был отсортирован по увеличению веса. Это позволило всегда выбирать самый первый элемент.

Для реализации алгоритма A* использовался модуль *heapq*, для создания очереди приоритетов. Приоритет для каждой конкретной вершины получался следующим образом: сумма расстояния до вершины, стоимость вершины, из

которой в нее перешли и эвристическая функция. Эвристическая функция оценивала близость символов текущей вершины и конечной в таблице ASCII.

Разработанный программный код для решения поставленной задачи успешно прошел тестирование на онлайн платформе *Stepik*.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
# Модуль для создания очереди приоритетов
import heapq

# Класс, описывающий ориентированный граф
class OrientedGraph():

    # Конструктор класса от значения стартовой и конечной вершины,
    # словаря ребер и их веса
    def __init__(self, start_node, end_node, graph_paths):
        self.start_node = start_node
        self.end_node = end_node
        self.graph_paths = graph_paths

    # Сортировка вершин по увеличению расстояния от исходной
    # Принимает на вход экземпляр класса
    def sorted_graph(self):
        for key in self.graph_paths:
            self.graph_paths[key].sort(key=lambda x:x[1])

    # Реализация жадного алгоритма
    # Принимает на вход экземпляр класса
    # Возвращает путь от начальной до конечной вершины
    def greedy_algorithm(self):
        self.sorted_graph()
        start = self.start_node
        answer = start
        while start != self.end_node:
            next_node = self.graph_paths[start][0][0]
            if next_node not in self.graph_paths.keys() and
next_node != self.end_node:
                self.graph_paths[start].pop(0)
            else:
                start = next_node
                answer += start
        return answer

    # Эвристическая функция для алгоритма A*
    # Принимает на вход экзмепляр класса и вершину
    # Возвращает близость символов, обозначающих
    # переданную и конечную вершину, в таблице ASCII
    def heuristic_function(self, node):
        return abs(ord(node)-ord(self.end_node))

    # Составление корректного ответа для алгоритма A*
    # Принимает на вход экземпляр класса и словарь ребер
    # Возвращает отформатированный ответ в соответствии с заданеим
    def Astar_answer(self, node_from):
        answer = self.end_node
        while node_from[answer[0]]:
            answer = node_from[answer[0]] + answer
```

```
            return answer

        # Реализация алгоритма A*
        # Принимает на вход экземпляр класса
        # Возвращает путь от начальной до конечной вершины
        def Astar_algorithm(self):
            node_cost = {self.start_node:0}
            pqueue = []
            heapq.heappush(pqueue, (0, self.start_node))
            node_from = {self.start_node: None}

            while len(pqueue):
                current = heapq.heappop(pqueue)[1]

                if current == self.end_node:
                    break

                if current in self.graph_paths.keys():
                    for node in self.graph_paths[current]:
                        cost = node_cost[current] + node[1]
                        if node[0] not in node_cost or cost <
node_cost[node[0]]:
                            node_cost[node[0]] = cost
                            priority = cost +
self.heuristic_function(node[0])
                            heapq.heappush(pqueue, (priority, node[0]))
                            node_from[node[0]] = current

            return self.Astar_answer(node_from)

    # Функция, считывающая ребра графа и их вес
    # Возвращает словарь ребер графа и их веса
    def reading_edges():
        graph_paths = dict()
        while True:
            try:
                for elements in input().split('\n'):
                    node1, node2, length = elements.split()
                    if node1 not in graph_paths.keys():
                        graph_paths[node1] = [[node2, float(length)]]
                    else:
                        graph_paths[node1] += [[node2, float(length)]]
            except:
                break
        return graph_paths

    # Функция, запускающая решение поставленной задачи
    def solution():
        start_node, end_node = input().split()
        graph_paths = reading_edges()
        graph = OrientedGraph(start_node, end_node, graph_paths)
        print(graph.greedy_algorithm())
        print(graph.Astar_algorithm())

    # Условие для запуска программы
    if __name__ == "__main__":
        solution()
```