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 _____ Кривоченко Д.И.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Разработать программу, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма, при помощи алгоритма 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 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.
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.

Рассмотрим структуру программы. Всё, что требуется для решения задачи находится в классе *Solve*. Рассмотрим его методы.

1) В конструкторе находятся следующие переменные: булевая переменная *path_found*, используемая в алгоритме жадного перебора; переменная *path*, хранящая построенный путь; *graph*, хранящая граф в виде словаря *{узел: список дочерних узлов}*; *start_node* и *end_node*, хранящие начальный и конечный узел соответственно.
2) Метод *printSolution()* выводит в консоль найденный путь
3) Функция *getDictGraphFromInput()* считывает с консоли, составляет граф и записывает его в переменную *graph*, формат которой описан выше

4) Метод *sortKeysInGraphDict()* сортирует список дочерних узлов при каждом узле-родителе графа по возрастанию веса рёбер между узлом-родителем и дочерним узлом

5) Метод *heuristic()* возвращает эвристическое значения для данного узла

6) Метод *startSolutionGreedy()* инициализирует решение жадным алгоритмом.

7) Метод *startSolutionAStar()* инициализирует решение алгоритмом *A**

8) Метод *iterateGreedy()* – рекурсивен, является реализацией жадного алгоритма. Получает на вход текущий узел и текущий путь. Если текущий узел совпадает с текущим путём, то флаг *path_found* обращается в *True* и все ветки перебора прерываются, а путь записывается в *path*. Иначе – в эту же функцию подаются дочерние узлы (текущий путь расширяется), при том первыми подаются узлы, инцидентные рёбра которых меньше. Таким образом, если самые «дешёвые» грани заведут нас в тупик, переберутся другие и мы получим ответ.

9) Метод *iterateAStar()* является реализацией алгоритма *A**. В алгоритме используется встроенный модуль с приоритетной очередью. В начале инициализируется эта самая очередь и два словаря (*parents* и *cost_sheet*): *parents* хранит *{дочерний узел: его родитель}* на данный момент, *cost_sheet* хранит *{узел: вес пути до этого узла}*. Пока очередь не пуста или полученный из очереди элемент не равен конечному, из очереди вынимается элемент и происходит проверка на то, что он является более выгодным, как промежуточный, для перехода к следующему узлу (является ли вес ребра, инцидентного этому элементу и дочернему ему меньше, чем записанный вес в *cost_sheet*). Если это так – записывается новый, меньший вес в *cost_sheet*, вычисляется приоритет, включающий в себя эвристическое значение, и в очередь помещается дочерний элемент с рассчитанным раннее приоритетом, а в словаре parents обновляется родитель дочернего элемента. После этого, происходит генерация строки-пути по словарю parents, она записывается в переменную path.

Исходный код приведён в приложении А.

Выводы.

Разработана программа, решающая задачу построения пути в ориентированном графе при помощи жадного алгоритма, построения кратчайшего пути при помощи алгоритма *A**. Успешно пройдены тесты на платформе *Stepik* для обоих алгоритмов. Реализация обоих алгоритмов заключена в одном классе. В ходе написания решающую роль играл тип данных словарь и абстрактная структура данных очередь с приоритетом. Жадный алгоритм реализован рекурсивно, чтобы сократить число строк кода и получить легко читаемый метод. В алгоритме A* использована эвристика, приведённая в *Stepik,* а именно – разность символов, характеризующих узлы по модулю в таблице *ASCII*.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from queue import PriorityQueue

class Solve :

    def __init__(self)->None :
    self.path_found = False
    self.path = None
    self.graph = None
    self.start_node = None
    self.end_node = None

    #    Выводит ответ в требуемом на Stepik формате(строка, содержащая
путь)

    def printSolution(self)->None:
print(self.path)

#    Составляет граф по входным данным, записывает граф в виде словаря,
где
#   ключ - узел - родитель, а значение - список дочерних узлов
def getDictGraphFromInput(self)->None:
graph = {}
self.start_node, self.end_node = input().split()
while True :
    try :
    for elem in input().split('\n') :
        cur_start_node, cur_end_node, cur_length = el - em.split()
        if graph.get(cur_start_node) is None :
graph[cur_start_node] = [(cur_end_node,
    float(cur_length))]
        else :
            graph[cur_start_node] += [(cur_end_node,
                float(cur_length))]
            except :
            break
            self.graph = graph

            #   Сортирует список дочерних узлов по весу рёбер
            def sortKeysInGraphDict(self) :
            for key in self.graph :
                self.graph.update(
                    { key: sorted(self.graph[key], key = lambda x : x[1])
})

                #   Вычисляет  эвристическое значения для данного узла
                def heuristic(self, node_to_check: str) :
                return abs(ord(self.end_node) - ord(node_to_check))

                #   Инициализирует решение жадным алгоритмом
                def startSolutionGreedy(self) :
```

```
                self.getDictGraphFromInput()
                self.sortKeysInGraphDict()
                self.iterateGreedy(self.start_node, f"{self.start_node}")

                #   Инициализирует решениеалгоритмом A *
                def startSolutionAStar(self)->None:
self.getDictGraphFromInput()
self.iterateAStar()

#   Строит путь жадным алгоритмом
def iterateGreedy(self, cur_node: str, cur_path : str)->None:
if self.path_found is True :
return
if cur_node == self.end_node :
    self.path = cur_path
    self.path_found = True
    return
    if self.graph.get(cur_node) is not None :
        for elem in self.graph[cur_node] :
            self.iterateGreedy(elem[0], cur_path + f"{elem[0]}")

            #   Строит путь алгоритмом A*
            def iterateAStar(self) :
            weighted_node_queue = PriorityQueue()
            parents = {}
            cost_sheet = {}
            weighted_node_queue.put((0, self.start_node))
            parents[self.start_node] = None
            cost_sheet[self.start_node] = 0
            while not weighted_node_queue.empty() :
                cur_node = weighted_node_queue.get()[1]
                if cur_node == self.end_node :
                    break
                    if self.graph.get(cur_node) != None :
                        for       next_node,       next_weight       in
self.graph.get(cur_node) :
                            new_cost = cost_sheet[cur_node] + next_weight
                            if next_node not in cost_sheet or new_cost <
cost_sheet[
                                next_node] :
                                cost_sheet[next_node] = new_cost
                                    new_priority     =     new_cost     +
self.heuristic(next_node)

weighted_node_queue.put((new_priority, next_node))
                                    parents[next_node] = cur_node
                                    cur_node = self.end_node
                                    self.path = f"{self.end_node}"
                                    while cur_node != self.start_node:
                                cur_node = parents[cur_node]
                                    self.path += cur_node
                                    self.path = self.path[:: - 1]

                                    if __name__ == "__main__" :
                                        solution = Solve()
                                        solution.startSolutionGreedy()
                                        solution.printSolution()
```