МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Алгоритмы поиска пути в графах

Студент гр. 9382 _____ Герасев Г.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Ознакомиться с Жадным алгоритмом и алгоритмом A* и научиться применять их на практике. Написать программу реализовывающую поиск пути в графе Жадным алгоритмом и алгоритмом A*.

Постановка задачи.

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.

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

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.

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

ade

Индивидуальное задание.

Вариант 9. Вывод графического представления графа.

Описание алгоритма.

Описание Жадного алгоритма:

Жадный алгоритм начинает работу в указанной вершине и переходит з нее на вершину с минимальным весом ребра, если возможно. Этот процесс продолжается до тех пор, пока не будет найдена вершина указанная как терминальная, или до тех пор, пока не станет невозможен переход. Во втором случае последний переход отменяется и выбирается следующая по минимальности веса вершина. Отмечаются все пройденные вершины для избежания циклов. Параллельно записывается выбранный путь и его вес.

Сложность жадного алгоритма.

В худшем случае сложность алгоритма будет равна O(N*M), где N – количество вершин, M – количество ребер. Объясняется это тем, что необходимо будет пройти все вершины и все ребра графа.

Так как для хранения графа используется список смежности, то сложность по памяти будет равна:

$$O(N+M),$$

где N – количество вершин, M – количество рёбер.

Описание алгоритма A*:

Создается упорядоченная очередь, в которую записываются возможные продолжения имеющихся путей с учетом эвристической функции и весом пути до вершины. В очередь добавляются вершины при переходе в новую, если возможно сокращение получившегося пути. При переходе в новую вершину выбирается вершина из верха очереди и продолжается соответствующий путь. Алгоритм продолжает работу до тех пор, пока не будет произведен переход в терминальную вершину.

Сложность алгоритма A*.

В лучшем случае сложность алгоритма будет равно O((N+M), где N – количество вершин, M – количество ребер. Объясняется это тем, что подходяще заданная эвристическая функция будет правильно выбирать путь до следующей вершины.

Временная сложность алгоритма A* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$\left|h(x) - h^{\square}(x)\right| \leq O\left(\log\left(h^{\square}(x)\right)\right);$$

где h* — оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка h(x) не должна расти быстрее, чем логарифм от оптимальной эвристики.

В худшем случае сложность по памяти будет экспоненциальная, так как все пути будут храниться в очереди. В лучшем случае будет храниться только путь от начала до текущей вершины. Из этого оценка по памяти будет равна:

$$O\left(N * (N+M)\right),$$

где N – количество вершин, M – количество ребер.

Описание структур.

Таблица 1 – Описание структур данных Жадного и A* алгоритмов

Название структуры	Поля класса	Описание
class	self.graph	Данное поле хранит список смежности графа. Создание происходит с помощью метода addEdge()

Graph		(описан в методах класса). Также используется в жадном алгоритме для нахождения расстояния от одной заданной вершины, до другой заданной вершины. (описан в методах класса). Тип: dict

Описание функций.

Описание функций Жадного алгоритма:

Таблица 3 – Описание методов Жадного алгоритма

Сигнатура	Параметры	Описание
def addEdge(self, head, leave, value)	self – экземпляр класса head – вершина из которой идет ребро (str/char/int/float – типы с которыми может работать) leave – вершина к которой идет ребро (str/char/int/float – типы с которыми	Метод проверяет наличие вершины, из которой идет ребро. Если вершина уже есть в словаре, то просто добавляет в словарь данной вершины новое поле: ключ со значением вершины, к которой идет ребро и со значением по ключу равным длине ребра. Если вершины нет, то в

	может работать) value – длина ребра (float/int – типы с которыми может работать)	изначальный словарь добавляет ключ со значением вершины, из которой идет ребро и со значением по ключу равным новому словарю. Внутрь этого словаря добавляется вершина и длина ребра как описывалось выше. Возвращаемого значения нет
def greedy(self, start, end)	self – экземпляр класса start – начальная вершина (str/char/int/float – типы с которыми может работать) end – конечная вершина (str/char/int/float – типы с которыми может работать)	Метод жадного алгоритма Возвращаемое значение – list - список вершин, показывающий пусть в графе от начальной вершины до конечной.
def __init__(self)	self – экземпляр класса	Конструктор класса Graph, инициализирует

		поля представленные в табл. 1

Описание функций алгоритма A*:

Таблица 4 – Описание функций алгоритма A*

Сигнатура	Параметры	Описание
def addEdge(self, head, leave, value)	self – экземпляр класса head – вершина из которой идет ребро (str/char/int/float – типы с которыми может работать) leave – вершина к которой идет ребро (str/char/int/float – типы с которыми может работать) value – вес ребра (float/int – типы с которыми может работать)	Метод проверяет наличие вершины, из которой идет ребро. Если вершина уже есть в словаре, то просто добавляет в словарь данной вершины новое поле: ключ со значением вершины, к которой идет ребро и со значением по ключу равным длине ребра. Если вершины нет, то в изначальный словарь добавляет ключ со значением вершины из которой идет ребро и со значением по ключу равным новому словарю. Внутрь этого словаря добавляется вершина и длина

		ребра как описывалось выше. Возвращаемого значения нет
def draw(self)	self – экземпляр класса	Метод отрисовки графа Возвращаемого значения нет
def __init__(self)	self – экземпляр класса	Конструктор класса Graph, инициализирует поля представленные в табл. 1
def a_star(self, start, end)	self – экземпляр класса start – начальная вершина (str/char/int/float – типы с которыми может работать) end – конечная вершина (str/char/int/float – типы с которыми может работать)	Метод реализации алгоритма A* Возвращаемое значение – list - список вершин, показывающий пусть в графе от начальной вершины до конечной.

Тестирование.

Входные данные	Ответ	Тест для алгоритма…
a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 8.0 f g 1.0	abdeag	Жадного алгоритма
a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0	abdefg	Жадного алгоритма
a g a b 3.0 a c 1.0 b d 2.0	abdefg	Жадного алгоритма

b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0 c m 1.0 m n 1.0		
g j a b 1 a f 3 b c 5 b g 3 f g 4 c d 6 d m 1 g e 4 e h 1 e n 1 n m 2 g i 5 i j 6 i k 1 j l 5 m j 3	genmj	A*
a j a b 1	afghij	A*

b c 1 c d 1 d e 1 e j 1 a f 1 f g 1 g h 1 h i 1 i j 1		
a z a x 5.0 x y 1.0 x z 1.0 a b 4.0 b z 2.0	axz	A*
a z a w 2.0 a b 1.0 b y 1.0 y z 1.0 w z 3.0	awz	A*

Вывод.

В ходе выполнения данной лабораторной работы были изучены и реализованы два алгоритма. Первый – жадный алгоритм поиска пути в ориентированном графе. Этот алгоритм выбирает наименьший путь на каждом шаге в надежде на то, что выбирая локальный минимум будет достигнут глобальный. Второй – алгоритм поиска минимального пути в

ориентированном графе A*, который является модификацией алгоритма Дейкстры. Модификация состоит в том, что A* находит минимальные пути не до каждой вершины в графе, а для заданной, а также используется эвристическая функция для выбора продолжения для вершин.

A*, в отличие от жадного алгоритма, гарантирует, что найденный путь будет минимальным возможным.

Также был реализован графический вывод графа.

ПРИЛОЖЕНИЕ А
Исходный код программы

Название файла: greedy.py

```
import networkx as nx
from pylab import show as pylabShow
from math import inf as mathInf



class Graph:
    def __init__(self):
        self.graph = {}

    def addEdge(self, root, leaf, value):
        if root not in self.graph:
            self.graph[root] = {}
        self.graph[root][leaf] = value
        print("Adding {} and {} with weight {}".format(root,
leaf, value))
        print("Current graph state: {}".format(self.graph))

    def greedy(self, start, end):
        handeledNodes = []
        while True:
            key = start
            ans = []
            while key in self.graph and any(self.graph[key]):
                ans.append(key)
                print("Current node: {}".format(key))
                min = mathInf
                next = None
                print("Checking all children of:
{}".format(key))
                for i in self.graph[key]:
```

```
                    print("Handeling node: {} with weight: {},
child of: {}".format(i, self.graph[key][i], key))
                    if self.graph[key][i] < min and i not in
handeledNodes:
                        if i in self.graph:
                            print("The weight is smaller\nGoing
to new node: {}".format(i))
                            next = i
                            min = self.graph[key][i]
                        elif i == end:
                            next = i
                            min = self.graph[key][i]
                key = next
                print("List of handeled nodes:
{}".format(handeledNodes))
                handeledNodes.append(key)
                if key is not None:
                    print("Adding node {} to the list of
handeled ones".format(key))
                print()
                if key == end:
                    ans.append(key)
                if ans[-1] == end:
                    return ans


    def draw(self):
        print("Initialisation of graph drawing procces...")
        g = nx.DiGraph()


        for i in self.graph:
            for j in self.graph[i]:
                print("Adding nodes {} {} with weight
{}".format(i, j, self.graph[i][j]))
                g.add_edges_from([(i, j)], weight=self.graph[i]
[j])
```

```
        edge_labels = dict([((u, v,), d['weight'])
        for u, v, d in g.edges(data=True)])
        pos = nx.spring_layout(g, scale=100, k=10)
        nx.draw_networkx_edge_labels(g, pos,
edge_labels=edge_labels)
        print("Drawing graph")
        nx.draw(g, pos, node_size=500, with_labels=True )
        pylabShow()



def inputHandler():
    inputList = []
    line = input()
    while line:
        inputList.append(line.strip())
        line = input()
    graph = Graph()

    for i in range(len(inputList)):
        inputList[i] = inputList[i].split(" ")
        if i > 0:
            graph.addEdge(inputList[i][0], inputList[i][1],
float(inputList[i][2]))

    start = inputList[0][0]
    end = inputList[0][1]

    return (graph, start, end)

if __name__ == '__main__':
    proccesedInput = inputHandler()
    graph = proccesedInput[0]
    start = proccesedInput[1]
    end = proccesedInput[2]
```

```
ans = graph.greedy(start, end)

graph.draw()
print("\nAnswer: ", end='')
for i in ans:
    print(i, end='')
```

ПРИЛОЖЕНИЕ Б
Исходный код программы

Название файла: A_star.py

```
import networkx as nx
from operator import itemgetter
from pylab import show as pylabShow



class Graph:
    def __init__(self):
        self.graph = {}

    def addEdge(self, root, leaf, value):
        if root not in self.graph:
            self.graph[root] = {}
        self.graph[root][leaf] = value
        print("Adding {} and {} with weight {}".format(root,
leaf, value))
        print("Current graph state: {}".format(self.graph))

    def a_star(self, start, end):
        shortPath = {}
        queue = []
        queue.append((start, 0))
        queue.sort(key=itemgetter(1), reverse=True)
        vector = [start]
        shortPath[start] = (vector, 0)
        while not queue == []:
            if queue[-1][0] == end:
                return shortPath[end][0]
            topOfQueue = queue[-1]
            print("Top of the queue is {}".format(queue[-1]))
            print("Current node {}".format(topOfQueue[0]))
            queue.pop()
```

```
            if topOfQueue[0] in self.graph:
                for i in list(self.graph[topOfQueue[0]].keys()):
                    currentPathLength = shortPath[topOfQueue[0]]
[1] + self.graph[topOfQueue[0]][i]
                    print("Current path from {} to {} =
{}".format(start, i, currentPathLength))
                    if i not in shortPath or shortPath[i][1] >
currentPathLength:
                        path = []
                        for j in shortPath[topOfQueue[0]][0]:
                            path.append(j)
                        path.append(i)
                        print("Current path is shorter")
                        shortPath[i] = (path, currentPathLength)
                        evristic = abs(ord(end) - ord(i))
                        print("Computing evristic function
{}".format(evristic))
                        queue.append((i, evristic + shortPath[i]
[1]))
                        queue.sort(key=itemgetter(1),
reverse=True)
                print()
        return shortPath[end][0]

    def draw(self):
        print("Initialisation of graph drawing procces...")
        g = nx.DiGraph()

        for i in self.graph:
            for j in self.graph[i]:
                print("Adding nodes {} {} with weight
{}".format(i, j, self.graph[i][j]))
                g.add_edges_from([(i, j)], weight=self.graph[i]
[j])

        edge_labels = dict([((u, v,), d['weight'])
        for u, v, d in g.edges(data=True)])
```

```
        pos = nx.spring_layout(g, scale=100, k=10)
        nx.draw_networkx_edge_labels(g, pos,
edge_labels=edge_labels)
        print("Drawing graph")
        nx.draw(g, pos, node_size=500, with_labels=True )
        pylabShow()



def inputHandler():
    inputList = []
    line = input()
    while line:
        inputList.append(line.strip())
        line = input()
    graph = Graph()

    for i in range(len(inputList)):
        inputList[i] = inputList[i].split(" ")
        if i > 0:
            graph.addEdge(inputList[i][0], inputList[i][1],
float(inputList[i][2]))

    start = inputList[0][0]
    end = inputList[0][1]

    return (graph, start, end)

if __name__ == '__main__':
    proccesedInput = inputHandler()
    graph = proccesedInput[0]
    start = proccesedInput[1]
    end = proccesedInput[2]

    ans = graph.a_star(start, end)
```

```
print("\nAnswer: ", end='')
for i in ans:
    print(i, end='')
print()
```