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 _____ Сорокумов С.В.

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:

Изначально вызывается метод preparing, который принимает вершину, с которой необходимо начать поиск и вершину, к которой необходимо дойти. Далее создается пустой список пройденных точек и переменная, предназначенная для проверки, что жадный алгоритм закончил на необходимой вершине. После чего запускается цикл пока переменная check не станет равной true и вызывается жадный алгоритм.

Жадный алгоритм выставляет переменной key значение переданного элемента start. После чего начинается цикл до того момента пока key есть в словаре и по ключу key есть элементы. Ключ заносится в список результата и алгоритм начинает проходить по всем значениям ключа и выбирать вершину, чьё ребро имеет меньший вес. После проверки всех значений по ключу, ключ добавляется в список пройденных вершин. И в переменную key заносится следующая вершина, которая была выбрана ранее. Это всё происходит до момента выхода из цикла, после чего алгоритм возвращает кортеж из списков – ответ (вершины) и список пройденных вершин.

Метод preparing распаковывает кортеж и проверяет если последняя вершина списка ответа является необходимой, то возвращает список с ответом, в ином случае запускает жадный алгоритм, передавая ему список пройденных вершин.

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.

Описание алгоритма A*:

В самом начале алгоритм создает экземпляр очереди с приоритетом и словарь, куда будут записываться пути. После чего запускается цикл пока очередь не пуста. В теле цикла алгоритм проверяет если в верхнем элементе

очереди есть значение вершины равной конечной, то он возвращает короткий путь из словаря по ключу конечной вершины. В ином случае в переменную temp передает текущую вершину и удаляет верхний элемент очереди. После чего проходится по словарю графа с ключом temp. Далее находит текущую длину и проверяет есть ли данная вершина в словаре коротких путей или сравнивает длины (текущую и из словаря). Если условие равно True, то добавляет вершину в словарь и добавляет расстояние, или только перезаписывает расстояние. После чего считает эвристическую функцию и добавляет в очередь новое поле, в котором находится вершина и расстояние плюс эвристическая функция, так как алгоритм использует очередь с приоритетом, то все элементы в ней отсортированы. Если алгоритм вышел из цикла, то он возвращает список вершин согласно условию.

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.

В худшем случае – $O(N^2)$ - где N – количество вершин. Из-за плохо подобранной эвристической функции придётся перебрать все возможные пути.

Описание структур.

Таблица 1 – Описание структур данных Жадного алгоритма

Название структуры	Поля класса	Описание
	self.graph	Хранит граф для вычислений Тип: dict

class Graph		
	self.g	Хранит граф для отрисовки Тип: nx.DiGraph

Таблица 2 – Описание структур данных алгоритма A*

Название структуры	Поля класса	Описание
class Graph	self.graph	Хранит граф для вычислений
	self.g	Хранит граф для отрисовки Тип: dict
class Queue	self.__data	Хранит элементы очереди Тип: list

Описание функций.

Описание функций Жадного алгоритма:

Таблица 3 – Описание методов Жадного алгоритма

Сигнатура	Параметры	Описание
def add_adge(self, head, leave, value)	self – экземпляр класса head – родитель (str/char/int/float – типы с которыми может работать)	Метод проверки если экземпляр родителя есть в словаре, то в словарь родителя добавляет новое поле с ключом потомка и значением value. Если

	leave – потомок (str/char/int/float – типы с которыми может работать) value – вес ребра (float/int – типы с которыми может работать)	значения родителя нет, то изначально добавляет его в словарь, повторяя алгоритм описанный выше Возвращаемого значения нет
def print_graph(self)	self – экземпляр класса	Печатает текстовое представление графа Возвращаемого значения нет
def preparing(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 подготовки к жадному алгоритму проверяет правильно ли выполнился жадный алгоритм Возвращаемое значение – list - список с вершинами
def greedy(self, start, end, done)	self – экземпляр класса start – начальная вершина	Метод жадного алгоритма Возвращаемое значение – tuple(list, list) – кортеж из списка вершин и списка

	(str/char/int/float – типы с которыми может работать) end – конечная вершина (str/char/int/float – типы с которыми может работать) done – список пройденных вершин (list)	пройденных вершин
def get_graph(self)	self – экземпляр класса	Метод для получения графа Возвращаемое значение словарь dict
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

Описание функций алгоритма A*:

Таблица 4 – Описание функций алгоритма A*

Сигнатура	Параметры	Описание
def add_adge(self, head, leave, value)	self – экземпляр класса head – родитель (str/char/int/float –	Метод проверки если экземпляр родителя есть в словаре, то в словарь родителя добавляет новое

	типы с которыми может работать) leave – потомок (str/char/int/float – типы с которыми может работать) value – вес ребра (float/int – типы с которыми может работать)	поле с ключом потомка и значением value. Если значения родителя нет, то изначально добавляет его в словарь, повторяя алгоритм описанный выше Возвращаемого значения нет
def print_graph(self)	self – экземпляр класса	Печатает текстовое представление графа Возвращаемого значения нет
def get_graph(self)	self – экземпляр класса	Метод для получения графа Возвращаемое значение словарь dict
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 __init__(self)	self – экземпляр класса	Конструктор класса Queue, инициализирует поля представленные в табл. 2
def __compare(self, a, b)	self – экземпляр класса a – первое	Приватный метод для сравнения двух элементов Возвращает true/false

	сравниваемое значение (принимает tuple) b – второе сравниваемое значение (принимает tuple)	
def top(self)	self – экземпляр класса	Возвращает верхний элемент в очереди Возвращает верхний элемент очереди
def push(self, el)	self – экземпляр класса el – элемент (принимает tuple)	Метод добавления в конец очереди el
def __sort(self)	self – экземпляр класса	Приватный метод для сортировки очереди
def pop(self)	self – экземпляр класса	Удаляет верхний элемент очереди
def a_star(self, start, end)	self – экземпляр класса start – начальная вершина (str/char/int/float – типы с которыми может работать)	Метод реализации алгоритма A* Возвращаемое значение – list - список вершин

	end – конечная вершина (str/char/int/float – типы с которыми может работать)	

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	abdefg	Жадного алгоритма

a g 8.0 f g 1.0		
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 c m 1.0 m n 1.0	abdefg	Жадного алгоритма
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	genmj	A*

i j 6 i k 1 j l 5 m j 3		
a j a b 1 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	afghij	A*
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 этом заключается жадность, алгоритм достаточно прост, но за это платит своей надежностью, так как он не гарантирует, что найденный путь будет минимальным возможным. 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. В ходе его работы при выборе пути учитывается не только вес ребра, но и эвристическая близость вершины к искомой. A* гарантирует, что найденный путь будет минимальным возможным. Также был реализован графический вывод графа.

ПРИЛОЖЕНИЕ А
Исходный код программы

Название файла: greedy.py

```
import sys
import networkx as nx
import numpy.random as rnd
import matplotlib.pyplot as plt
import pylab



# Класс графа
class Graph:
    def __init__(self):
        """
        Конструктор класса, не принимает ничего, создает поля: где хранится граф и необходимые для него.
        """
        self.graph = {}
        self.g = nx.DiGraph()
        self.node_in_graph = []


    def add_adge(self, head, leave, value):
        """
        Функция добавления вершины с ребром
        :param head: вершина из которой будет проведено ребро
        :param leave: вершина к которой будет проведено ребро
        :param value: вес ребра
        :return: None
        """
        if head not in self.graph:
            self.graph[head] = {}
        self.graph[head][leave] = value
        print('Добавляем вершины {} и {} и расстояние между ними равное {}'.format(head, leave, value))
```

```
        print("текущее значение графа: {}".format(self.graph))

    def print_graph(self):
        """
        Печатает словарь графа
        :return: None
        """
        print(self.graph)

    def preparing(self, start, end):
        """
        Метод подготовк для жирного алгоритма
        :param start: вершина с которой начинается алгоритм
        :param end: вершина на которой закончится алгоритм
        :return: массив вершин
        """
        done = []
        check = False
        ans = []

        while not check:  # Будет выполняться до тех пор пока не
получит необходимый ответ
            ans, done = self.greedy(start, end, done)  #
Вызывается жирный алгоритм
            if ans[-1] == end:
                check = True
        return ans

    def greedy(self, start, end, done):
        """
        Метод класса, отвечающий за жирный алгоритм
        :param start: вершина с которой начинается алгоритм
        :param end: вершина на которой закончится алгоритм
        :param done: массив вершин
```

```
        :return:
        """
        key = start
        ans = []
        while key in self.graph and any(self.graph[key]):


            ans.append(key)
            print("Текущая вершина: {}".format(key))
            min = 999999999999
            next = None
            print("Проходимся по всем вершинам от вершины:
{}".format(key))
            for i in self.graph[key]:
                print("Рассматриваем вершину: {} с расстоянием:
{} от вершины {}".format(i, self.graph[key][i], key))
                # если длина меньше чем минимальное значение и
ключ не находится в пройденных вершинах
                if min > self.graph[key][i] and i not in done:
                    # если ключ в графе следующее значение ключа
                    if i in self.graph:
                        print("Расстояние меньше\nПереходим на
вершину {}".format(i))
                        next = i # задаём следующее значение
ключа
                        min = self.graph[key][i] # задаём
минимальное расстояние
                    # если ключ равен конечной вершине
                    elif i == end:
                        next = i # задаём следующее значение
ключа
                        min = self.graph[key][i] # задаём
минимальное расстояние
            key = next # задаём следующее значение ключа
            print("Список пройденных вершин равен:
{}".format(done))
            done.append(key) # добавляем вершину в список
пройденных
```

```
            print("Добавляем вершину {} в список
пройденных".format(key))
            print()
            if key == end:
                ans.append(key)
                return ans, done
        return ans, done # возвращаем ответ


    def get_graph(self):
        return self.graph


    def draw(self):
        # метод для отрисовки графа
        print("Начинаем построение графа для отрисовки")
        # Инициализируем вершины и ребра графа
        for i in self.graph:
            for j in self.graph[i]:
                print("Добавляем вершины {} {} с расстоянием
{}".format(i, j, self.graph[i][j]))
                self.g.add_edges_from([(i, j)],
weight=self.graph[i][j])


        edge_labels = dict([((u, v,), d['weight'])
                            for u, v, d in
self.g.edges(data=True)])
        pos = nx.spring_layout(self.g, scale=100, k=10)  #
вычисление позиций
        nx.draw_networkx_edge_labels(self.g, pos,
edge_labels=edge_labels)  # отрисовывем граф
        print("Рисуем граф")
        nx.draw(self.g, pos, node_size=500, with_labels=True )
        pylab.show()



if __name__ == '__main__':
    a_lst = []
```

```
    # считываем данные
    line = input()
    while line:
        a_lst.append(line.strip())
        line = input()
    # создание объекта графа
    tree = Graph()
    # заполнение графа
    for i in range(len(a_lst)):
        a_lst[i] = a_lst[i].split(" ")
        if i > 0:
            tree.add_adge(a_lst[i][0], a_lst[i][1],
float(a_lst[i][2]))
    print("\nПостроение графа завершено, переходим к жадному
алгоритму\n")
    ans = tree.preparing(a_lst[0][0], a_lst[0][1])
    # Вывод ответа


    # отрисовка графа
    tree.draw()
    print("\nОтвет на задание:", end='')
    for i in ans:
        print(i, end='')
```

ПРИЛОЖЕНИЕ Б
Исходный код программы

Название файла: A*.py

```
import sys
import networkx as nx
import numpy.random as rnd
import matplotlib.pyplot as plt
import pylab



class Queue:
    def __init__(self):
        print("Инициализация очереди выполнена успешно")
        self.__data = []

    def __compare(self, a, b):
        if a[1] == b[1]:
            return a[0] < b[0]
        else:
            return a[1] > b[1]

    def top(self):
        # Возврат верхнего элемента очереди
        return self.__data[-1]

    def push(self, el):
        # добавление элемента в очердь
        print("Добавление элемента {} в очерредь".format(el))
        self.__data.append(el)
        self.__sort()

    def __sort(self):
        # Сортировка очереди
```

```
        print("Сортировка очереди")
        for i in range(len(self.__data) - 1):
            for j in range(len(self.__data) - i - 1):
                if not self.__compare(self.__data[j],
self.__data[j + 1]):
                    self.__data[j], self.__data[j + 1] =
self.__data[j + 1], self.__data[j]


    def pop(self):
        # удаление верхнего элемента из очереди
        print("Удаляем элемент {} из
очереди".format(self.__data[-1]))
        self.__data.pop()


    def empty(self):
        # проверка пустая ли очередь
        if len(self.__data) == 0:
            print("Очередь пуста")
            return True
        else:
            print("Очередь не пуста")
            return False



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


    def add_adge(self, head, leave, value):
        """
        Функция добавления вершины с ребром
        :param head: вершина из которой будет проведено ребро
        :param leave: вершина к которой будет проведено ребро
        :param value: вес ребра
```

```
        :return: None
        """
        if head not in self.graph:
            self.graph[head] = {}
        self.graph[head][leave] = value
        print('Добавляем вершины {} и {} и расстояние между ними равное {}'.format(head, leave, value))
        print("текущее значение графа: {}".format(self.graph))


    def print_graph(self):
        """
        Печатает словарь графа
        :return: None
        """
        print(self.graph)



    def a_star(self, start, end):
        shortPath = {}
        # Инициализации очереди с приоритетом
        queue = Queue()
        queue.push((start, 0))
        vector = [start]
        shortPath[start] = (vector, 0)
        # Пока очередь не пуста
        while not queue.empty():
            if queue.top()[0] == end:
                # ]если верхний элемент очереди равен итоговой вершине возвращаем результат
                return shortPath[end][0]
            temp = queue.top()
            print("Верхний элемент очереди равен {}".format(queue.top()))
            print("Текущая вершина {}".format(temp[0]))
            queue.pop()
```

```
            # Проходимся по всем вершинам от текущей
            if temp[0] in self.graph:
                for i in list(self.graph[temp[0]].keys()):
                    # Считаем текущее рассстояние
                    currentPathLength = shortPath[temp[0]][1] +
self.graph[temp[0]][i]
                    print("Текущий путь равен
{}".format(currentPathLength))
                    # Если текущее расстояние меньше
                    if i not in shortPath or shortPath[i][1] >
currentPathLength:
                        # Изменяем данный
                        path = []
                        for j in shortPath[temp[0]][0]:
                            path.append(j)
                        path.append(i)
                        print("Текущий путь оказался короче")
                        print("Текущий путь из вершин
{}".format(path))
                        shortPath[i] = (path, currentPathLength)
                        # Подсчет эвристической функции
                        evristic = abs(ord(end) - ord(i))
                        print("Считаем эвристическую функцию,
которая равна {}".format(evristic))
                        queue.push((i, evristic +
shortPath[i][1]))
                print()
        return shortPath[end][0]


    def draw(self):
        # метод для отрисовки графа
        print("\nНачинаем построение графа для отрисовки")
        # Инициализируем вершины и ребра графа
        for i in self.graph:
            for j in self.graph[i]:
```

```
                print("Добавляем вершины {} {} с расстоянием
{}".format(i, j, self.graph[i][j]))
                self.g.add_edges_from([(i, j)],
weight=self.graph[i][j])


        edge_labels = dict([((u, v,), d['weight'])
                            for u, v, d in
self.g.edges(data=True)])
        pos = nx.spring_layout(self.g, scale=100, k=10)  #
вычисление позиций
        nx.draw_networkx_edge_labels(self.g, pos,
edge_labels=edge_labels)  # отрисовывем граф
        print("Рисуем граф")
        nx.draw(self.g, pos, node_size=500, with_labels=True)
        pylab.show()



if __name__ == '__main__':
    a_lst = []
    # считываем данные
    line = input()
    while line:
        a_lst.append(line.strip())
        line = input()
    # создание объекта графа
    tree = Graph()
    # заполнение графа
    for i in range(len(a_lst)):
        a_lst[i] = a_lst[i].split(" ")
        if i > 0:
            tree.add_adge(a_lst[i][0], a_lst[i][1],
float(a_lst[i][2]))
    print("\nПостроение графа завершено, переходим к A*
алгоритму\n")
    ans = tree.a_star(a_lst[0][0], a_lst[0][1])
    # Вывод ответа
```

```
# отрисовка графа
tree.draw()
print("\nОтвет на задание:", end='')
for i in ans:
    print(i, end='')
```