**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 А\*

|  |  |  |
| --- | --- | --- |
| Студент гр. 1304 |  | Поршнев Р.А. |
| Преподаватель |  | Шевелева А.М. |

Санкт-Петербург

2023

## Цель работы.

Написать программу, которая решает задачу построения пути в ориентированном графе с помощью жадного алгоритма и А\*.

## 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 А\*. 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.

## Основные теоретические положения.

1. Жадный алгоритм реализован с помощью рекурсии. На каждом шаге рекурсии происходит проверка: если путь найден, то нужно выйти из текущего состояния рекурсии. Гарантируется, что первый найденный путь будет самым оптимальным согласно алгоритму в задании, ведь среди всех не посещённых узлов на каждом шаге выбираются в первую очередь узлы с минимальным весом. Также происходит следующая проверка: если текущий узел равен конечному, то следует зафиксировать путь, отметить, что ответ найден, и выйти из текущего состояния рекурсии. Флаг о том, что ответ найден, потребуется для проверки выше. Реализована ещё одна проверка: если из текущего узла не выходят дуги, то следует выйти из текущего состояния рекурсии. После данных проверок всех смежных узлов для текущего узла выбираются не посещённые узлы, а затем сортируются по неубыванию весов. Следующим шагом текущий узел отмечается посещённым и для каждого отсортированного узла-соседа вызывается рекурсивная функция.
2. Начало работы алгоритма A\* выглядит следующим образом: в очередь с приоритетом добавляется старт-узел и его приоритет, равный нулю. Далее инициализируется словарь, в котором в качестве ключа выступает узел, а в качестве значения – минимальный вес рёбер, ведущих к данному узлу. Также инициализируется словарь, в котором в качестве ключа выступает узел, а значение – узел, из которого можно попасть в ключ-узел. Данный словарь нужен для восстановления пути. Далее начинает работу цикл, который работает до тех пор, пока очередь с приоритетом не станет пуста. Внутри цикла происходит извлечение из очереди элемента с наименьшим приоритетом. Затем следует проверка: если текущий узел равен конечному, то цикл останавливается. Затем для каждого узла, смежного с текущим, пересчитывается новое значение веса рёбер, ведущих к смежному узлу. Если смежный узел не находится в очереди или новое значение веса лучше текущего для данного смежного узла, то ему присваивается значение новое значение веса, в словарь для восстановления пути в ключ записывается название смежного узла, а в значение – текущий узел. Затем происходит расчёт приоритета, который равен новому значению веса и эвристической функции. Последним шагом итерации в данном блоке является добавление данного смежного узла и его приоритета в очередь узлов, которые следует рассмотреть. После окончания работы алгоритма остаётся восстановить путь, опираясь на записи в словаре.

## Выполнение работы.

1. Для решения поиска пути в орграфе с помощью жадного алгоритма создан класс *Graph*. В конструкторе происходит объявление переменной для сохранения пути, стартового и конечного узла, словаря для хранения посещённых узлов и самого графа, инициализация флага о том, что путь не найден. Реализованы следующие методы:

* *def solution(self)* – данный метод вызывает функционал для считывания исходного графа, поиска пути с помощью жадного алгоритма и печати пути в консоль;
* *def \_\_print\_path(self)* – метод для вывода ответа на исходную задачу в консоль;
* *def \_\_read\_graph(self)* – данный метод считывает стартовый и конечный узел, а так же весь граф. Граф сохраняется следующим образом: в словарь в качестве ключа записываются все узлы, из которых выходят дуги, а значение для каждого такого ключа – список кортежей, где первый элемент кортежа – название узла, в которую входит дуга из ключа, а второй – вес ребра между ними. Все узлы в контексте посещения устанавливаются в значение *false*;
* *def \_\_get\_unexplored\_neighbors(self, neighbors\_info)* – данный метод предназначен для получения не посещённых узлов, которые смежны рассматриваемому узлу. Входные данные: смежные узлы. Выходные данные: смежные не посещённые узлы;
* *def \_\_greedy\_algorithm(self, current\_node, path)* – данный метод является реализацией жадного алгоритма. Принцип его работы описан в разделе [Основные теоретические положения, п. 1](#жадник). Входные данные: текущий узел и путь.

[Код представлен в Приложении А, greedy.py](#greedy).

1. Для решения поиска пути в орграфе с помощью жадного алгоритма создан класс *Graph*. В конструкторе происходит объявление переменной для сохранения пути, стартового и конечного узла, словаря для хранения графа. Метод для [вывода ответа](#print_path) на исходную задачу такой же, как и в реализации задачи поиска пути жадным алгоритмом, считывание графа в алгоритме А\* отличается от реализации в [жадном](#read_graph) лишь тем, что все узлы в контексте посещения никак не помечаются.

* *def solution(self)* – данный метод вызывает функционал для считывания исходного графа, поиска пути алгоритмом А\* и печати пути в консоль;
* *def \_\_heuristic(self, current\_node)* – данный метод предназначен для расчёта эвристической функции, которая равна модулю разности кодов символов в таблице ASCII, где в качестве символов выступают буквенные обозначения текущего узла и конечного. Входные данные: буквенное обозначение текущего узла;
* *def \_\_a\_star(self)* – данный метод реализует алгоритм A\*. Принцип его работы описан в разделе [Основные теоретические положения, п. 2](#a_star);
* *def \_\_recover\_path(self)* – данный метод предназначен для восстановления пути по записям из словаря *self.\_\_came\_from*.

[Код представлен в Приложении А, a\_star.py](#жопа).

## Выводы.

В ходе выполнения работы было реализовано два алгоритма, которые ищут минимальный путь в графе между заданными вершинами.

Было реализовано две программы на языке *Python*, каждая из которых ищет минимальный путь в орграфе между заданными узлами. Первая программа ищет минимальный путь с помощью жадного алгоритма, а вторая с помощью алгоритма A\*. Жадный алгоритм является рекурсивным и в данной задаче зачастую уступает в оптимальности найденного пути алгоритму А\*.

# Приложение А Исходный код программы

Название файла: greedy.py

class Graph:

# инициализация необходимых переменных и структур данных

def \_\_init\_\_(self):

self.\_\_path = '' # для сохранения ответа на задачу

self.\_\_graph\_storage = {} # структура для хранения графа

self.\_\_start\_node = '' # старт-узел

self.\_\_end\_node = '' # финиш-узел

self.\_\_path\_was\_found = False

self.\_\_visited\_nodes = {}

# данный метод запуск логику решения задачи

def solution(self):

self.\_\_read\_graph()

self.\_\_greedy\_algorithm(self.\_\_start\_node, self.\_\_start\_node)

self.\_\_print\_path()

# выход: путь из стартового узла в финишный

# данцный метод предназачен для вывода ответа на исходную задачу

def \_\_print\_path(self):

print(self.\_\_path)

# данный метод предназначен для считывания графа

def \_\_read\_graph(self):

self.\_\_start\_node, self.\_\_end\_node = list(input().split()) # считывание стартового и финишного узла

while True:

try:

node\_pair\_data = input().split() # первый узел, второй и вес дуги между ними

if node\_pair\_data[0] not in self.\_\_graph\_storage.keys(): # если для первого узла это первая дуга

self.\_\_graph\_storage[node\_pair\_data[0]] = [(node\_pair\_data[1], float(node\_pair\_data[2]))]

else: # если для первого узла это не первая дуга

self.\_\_graph\_storage[node\_pair\_data[0]].append((node\_pair\_data[1], float(node\_pair\_data[2])))

self.\_\_visited\_nodes[node\_pair\_data[0]] = False # первый узел ещё не посещён

self.\_\_visited\_nodes[node\_pair\_data[1]] = False # как и второй, это нужно для жадного алгоритма

except:

break # закончить считывание

# данный метод предназначен для отбора непросмотренных вершин в текущем состоянии рекурсии

# входные данные: список смежных вершин с данной

# выходные данные: список непросмотренных смежных вершин с данной

def \_\_get\_unexplored\_neighbors(self, neighbors\_info):

unexplored\_neighbors = []

for node\_info in neighbors\_info:

if not self.\_\_visited\_nodes[node\_info[0]]: # если вершина не посещалась ранее

unexplored\_neighbors.append(

node\_info) # то добавить её и вес входящего в неё ребра из данной вершины в список

return unexplored\_neighbors

# данный рекурсивный метод предназначен для поиска пути в графе по заданному жадному алгоритму

def \_\_greedy\_algorithm(self, current\_node, path):

if self.\_\_path\_was\_found:

return

elif current\_node == self.\_\_end\_node: # если финишный узел достигнут

self.\_\_path = path # записать полученный путь

self.\_\_path\_was\_found = True # отметить, что ответ найдён

return

elif current\_node not in self.\_\_graph\_storage: # если узел висячий и не финишный узел

return # то из него точно не выходят дуги, можно остановить поиск

unexplored\_neighbors = self.\_\_get\_unexplored\_neighbors(self.\_\_graph\_storage[current\_node])

sorted\_neighbors\_weight = sorted(unexplored\_neighbors, key=lambda node\_data: node\_data[

1]) # сортировка непосещённых узлов по весам входящих в них дуг из текущей вершины

self.\_\_visited\_nodes[current\_node] = True # пометить текущую вершину, что она посещена

for node\_info in sorted\_neighbors\_weight:

self.\_\_greedy\_algorithm(node\_info[0], path + node\_info[0])

if \_\_name\_\_ == "\_\_main\_\_":

graph = Graph()

graph.solution()

Название файла: a\_star.py

from queue import PriorityQueue

class Graph:

# инициализация необходимых переменных и структур данных

def \_\_init\_\_(self):

self.\_\_path = '' # для сохранения ответа на задачу

self.\_\_graph\_storage = {} # структура для хранения графа

self.\_\_start\_node = '' # старт-узел

self.\_\_end\_node = '' # финиш-узел

# данный метод запуск логику решения задачи

def solution(self):

self.\_\_read\_graph()

self.\_\_a\_star()

self.\_\_print\_path()

# данный метод предназачен для вывода ответа на исходную задачу

def \_\_print\_path(self):

print(self.\_\_path)

# данный метод предназначен для считывания графа

def \_\_read\_graph(self):

self.\_\_start\_node, self.\_\_end\_node = list(input().split()) # считывание стартового и финишного узла

while True:

try:

node\_pair\_data = input().split() # первый узел, второй и вес дуги между ними

if node\_pair\_data[0] not in self.\_\_graph\_storage.keys(): # если для первого узла это первая дуга

self.\_\_graph\_storage[node\_pair\_data[0]] = [(node\_pair\_data[1], float(node\_pair\_data[2]))]

else: # если для первого узла это не первая дуга

self.\_\_graph\_storage[node\_pair\_data[0]].append((node\_pair\_data[1], float(node\_pair\_data[2])))

except:

break # закончить считывание

# входные данные: текущая вершина

# выходные данные: значение эвристической функции

# данный метод предназначен для нахождения значения эвристической функции, которая представляет собой

# модуль разности ASCII-кодов текущего узла и конечного

def \_\_heuristic(self, current\_node):

return abs(ord(current\_node) - ord(self.\_\_end\_node))

# данный метод предназначен для нахождения минимального по стоимости пути между старовым и конечным узлом

# с помощью алгоритма А\*

def \_\_a\_star(self):

open\_nodes = PriorityQueue() # узлы, которые следует рассмотреть

open\_nodes.put((0, self.\_\_start\_node)) # начиная с начального

weight = {self.\_\_start\_node: 0} # ключ - узел, значение - суммарный вес рёбер, необходимый для достижения узла

self.\_\_came\_from = {

self.\_\_start\_node: None} # ключ - узел, значение - из какой вершины можно попасть в ключ - узел

while not open\_nodes.empty():

current\_node\_name = open\_nodes.get()[1] # извлечь из очереди имя самого приоритетного узла

if current\_node\_name == self.\_\_end\_node:

break # конечный узел достигнут

if current\_node\_name not in self.\_\_graph\_storage:

self.\_\_graph\_storage[current\_node\_name] = [] # узел без исходящих из него дуг

for neighbor\_node in self.\_\_graph\_storage[current\_node\_name]: # для каждого узла смежного с текущим узлом

neighbor\_name = neighbor\_node[0]

weight\_between = neighbor\_node[1]

new\_weight = weight[current\_node\_name] + weight\_between # новый текущий вес для соседнего узла

if neighbor\_name not in weight or new\_weight < weight[neighbor\_name]:

# если соседний узел ещё не находится в открытом списке

# или вес узла, смежного с текущим, можно уменьшить

weight[neighbor\_name] = new\_weight

self.\_\_came\_from[neighbor\_name] = current\_node\_name

priority = new\_weight + self.\_\_heuristic(neighbor\_name) # приоритетность соседнего узла

open\_nodes.put((priority, neighbor\_name))

self.\_\_recover\_path()

# данный метод предназначен для восстановления пути по записям из словаря между стартовым и конечным узлом

def \_\_recover\_path(self):

current\_node = self.\_\_end\_node

while current\_node:

self.\_\_path += current\_node

current\_node = self.\_\_came\_from[current\_node]

self.\_\_path = self.\_\_path[::-1]

if \_\_name\_\_ == "\_\_main\_\_":

graph = Graph()

graph.solution()