# 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 Сулименко М.А.

Преподаватель Шевелева А.М.

Санкт-Петербург 2023

# Цель работы.

Изучение алгоритмов на графах. Изучение жадных алгоритмов, их сравнение с эвристическими алгоритмами, а также решение задачи поиска кратчайшего пути между 2 вершинами графа.

# 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.

# Выполнение работы.

Выполнение работы было разбито на 2 части: жадный алгоритм и алгоритм А\*.

Для решения первой задачи был разработан класс *Solve*, со следующим списком методов.

1. Метод *\_\_init\_\_* — конструктор класса, в котором инициализируются необходимые переменные. *Graph* – граф в виде списка, *startNode –* начальный узел пути, *endNode –* конечный узел.*isPathFound –* булевая переменная, отвечающая за то, найден ли путь(изначально инициализирована False), *path –* строка для записи пути, необходимого для ответа.
2. Метод *print\_final\_path* — функция, которая выводит финальный путь. Ничего не принимает и не возвращает.
3. Метод *read\_input —* функция, которая считывает данные из потока ввода и преобразует их в словарь, в котором ключом являются вершины-родители, а значениями – вершины дети и расстояние между ними.
4. Метод *sort\_keys\_in\_dict —* функция, которая сортирует все значения каждого ключа в словаре по расстоянию между ними.
5. Метод *start\_greedy\_algorithm —* функция, которая инициализирует решение для жадного алгоритма путем вызова необходимых методов.
6. Метод *iterate\_greedy\_algorithm —* рекурсивная функция, которая непосредственно решает поставленную задачу. Принцип действия алгоритма следующий: из очередной вершины происходит переход к вершине по ребру с минимальным весом, после этого текущей вершиной назначается это ребро и тоже самое повторяется для новой полученной вершины. Функция принимает и возвращает текущую вершину и промежуточный путь.

Помимо этого, была разработана функция *get\_solution* для инициализации элемента класса Slove и старта и вывода решения.

Для решения первой задачи был разработан класс *Solve*, со следующим списком методов.

1. Метод *\_\_init\_\_* — конструктор класса, в котором инициализируются необходимые переменные. *Graph* – граф в виде списка, *startNode –* начальный узел пути, *endNode –* конечный узел. *path –* строка для записи пути, необходимого для дальнейшего вывода правильного ответа.
2. Метод *read\_input —* функция, которая считывает данные из потока ввода и преобразует их в словарь, в котором ключом являются вершины-родители, а значениями – вершины дети и расстояние между ними.
3. Метод *get\_heuristics —* функция, которая определяет эвристику двух вершин с помощью близости их символов в таблице ASCII.
4. Метод *start\_a\_star\_algorithm —* Пока очередь не пуста в эту очередь с приоритетом добавляется очередное ребро. Если вершина уже была просмотрена, или по эвристике минимальный путь через данную вершину точно не проходит, данная вершина не рассматривается. Стоит отметить, что очередь с приоритетом формируется с учетом эвристики.
5. Метод *build\_path —* функция, которая рассчитывает необходимый для ответа путь, с помощью обратного следования по словарю *previousNode*. Полученная строка записывается в path и инвертируется.

Помимо этого, была разработана функция *get\_solution* для инициализации элемента класса Slove и старта и вывода решения. Для удобной реализации очереди с приоритетом был импортирован класс *PriorityQueue*.

Исходный код программы представлен в [Приложение А Исходный код](#_bookmark0) [программы](#_bookmark0).

# Выводы.

Были изучены основные алгоритмы на графах, такие как A\* и жадный алгоритм. При сравнении двух алгоритмов было получено, что жадный алгоритм, выбирая локально лучший результат не всегда вычисляет глобально лучшее решения. Также был изучен эвристический подход к решению задач. С помощью алгоритма A\* был найден кратчайший путь между 2 вершинами в ориентированном графе. На платформе *Stepik* были успешно пройдены проверки и обе программы оказались верными.

# ПРИЛОЖЕНИЕ А ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: greedy.py

class Solve:

def \_\_init\_\_(self) -> None:

self.graph = dict()

self.startNode = None

self.endNode = None

self.isPathFound = False

self.path = None

# Выводит ответ в нужном формате

def print\_final\_path(self):

print(self.path)

# Считывает входные данные и составляет граф в виде словаря

def read\_input(self):

self.startNode, self.endNode = input().split()

while True:

try:

from\_node, in\_node, weight = input().split()

if from\_node in self.graph:

self.graph[from\_node].append([in\_node, float(weight)])

else:

self.graph[from\_node] = [[in\_node, float(weight)]]

except:

break

# Сортирует значения по весу рёбер

def sort\_keys\_in\_dict(self):

for key in self.graph:

self.graph[key].sort(key=lambda elem: elem[1])

# Инициализирует решение жадным алгоритмом

def start\_greedy\_algorithm(self):

self.read\_input()

self.sort\_keys\_in\_dict()

self.iterate\_greedy\_algorithm(self.startNode, self.startNode)

# Строит путь жадным алгоритмом

def iterate\_greedy\_algorithm(self, currentNode, currentPath):

if self.isPathFound:

return

if currentNode == self.endNode:

self.path = currentPath

self.isPathFound = True

return

if self.graph.get(currentNode):

for nodes in self.graph[currentNode]:

nextNode = nodes[0]

nextPath = currentPath + nextNode

self.iterate\_greedy\_algorithm(nextNode, nextPath)

# Функция, которая создаёт экземпляр класса solve

# и запускает необходимые для решения функции

def get\_solution():

solution = Solve()

solution.start\_greedy\_algorithm()

solution.print\_final\_path()

get\_solution()

Название файла: aStar.py

from queue import PriorityQueue

class Solve:

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

self.graph = dict()

self.startNode = None

self.endNode = None

self.path = ''

# Считывает входные данные и составляет граф в виде словаря

def read\_input(self):

self.startNode, self.endNode = input().split()

while True:

try:

parent, child, cost = input().split()

if parent in self.graph:

self.graph[parent].append([child, float(cost)])

else:

self.graph[parent] = [[child, float(cost)]]

except:

break

# Метод, реализующий эвристическую функцию

def get\_heuristics(self, node):

return abs(ord(node) - ord(self.endNode))

# Метод, который строит путь в необходимом формате

def build\_path(self, previousNode):

currentNode = self.endNode

while currentNode is not None:

self.path += currentNode

currentNode = previousNode[currentNode]

self.path = self.path[::-1]

print(self.path)

# Метод, который реализует алгорит A\*

def start\_a\_star\_algorithm(self):

graphQueue = PriorityQueue()

graphQueue.put((0, self.startNode))

previousNode = {self.startNode: None}

intermediateCost = {self.startNode: 0}

while not graphQueue.empty():

currentNode = graphQueue.get()[1]

if currentNode == self.endNode:

break

if currentNode in self.graph:

adjacentNodes = self.graph[currentNode]

for nextNode, costOfNextNode in adjacentNodes:

newCost = intermediateCost[currentNode] + costOfNextNode

if nextNode not in intermediateCost or newCost < intermediateCost[nextNode]:

intermediateCost[nextNode] = newCost

priority = newCost + self.get\_heuristics(nextNode)

graphQueue.put((priority, nextNode))

previousNode[nextNode] = currentNode

self.build\_path(previousNode)

# Функция, которая создаёт экземпляр класса solve

# и запускает необходимые для решения функции

def get\_solution():

solution = Solve()

solution.read\_input()

solution.start\_a\_star\_algorithm()

get\_solution()