**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: Кратчайшие пути в графе: жадный алгоритм и**

**алгоритм А\***

| Студент гр. 1303 |  | Чубан Д.В. |
| --- | --- | --- |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2023

* 1. **Цель работы.**
  2. Написать программу, осуществляющую поиск и построение кратчайшего пути между двумя вершинами в графе при помощи жадного алгоритма и алгоритма А\*.
  3. **Задание:**
  4. 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.
  5. 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

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.  
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

6. Реализация очереди с приоритетами, используемой в А\*, через двоичную кучу.

* 1. **Выполнение работы.**

Для решения задачи были использованы два разных алгоритма для поиска кратчайшего пути в графе:

* Жадный алгоритм:

Данный алгоритм основан на поиске в глубину на каждом шаге выбирается смежное ребро с наименьшим весом, после чего добавляем вершину, в которую перешли, к ответу и проверяем, есть ли из неё рёбра в другие узлы. Если же рёбер нет, мы поднимаемся в пройденные вершины до тех пор, пока не найдётся та, у которой есть не исследованные рёбра. Как только была достигнута конечная вершина, функция, реализующая алгоритм, возвращает список вершин, формирующих путь.

Т.к. алгоритм основан на поиске в глубину, его сложность по времени можно оценить как , где - число вершин графа. Затраты памяти можно оценить как , где |E| и |V| - число рёбер и вершин графа соответственно, потому что для хранения рёбер графа используется словарь, ключами которого являются названия вершин, а значениями для ключей – пары: вторая вершина смежного ребра и вес соответственного ребра.

* Алгоритм А\*:

В реализованную очередь с приоритетом добавляем стартовую вершину. До тех пор, пока очередь не пуста, достаём из неё вершину с наименьшей оценкой пути и рассчитываем оценку пути для смежных вершин. Если очередная вершина ещё не была посещена, или существующая оценка пути больше только что вычисленного, значение цены посещения для данной вершины обновляется. После этого вершина и её стоимость помещаются в очередь с учётом эвристической функции. Если вершина в текущей итерации совпадает с конечной, то поиск прекращается и алгоритм возвращает ответ. Если вершина является листом, то она пропускается.

Временная сложность алгоритма А\* зависит от выбранной эвристики и от разветвлённости графа. В худшем случае количество рассматриваемых путей на каждом шаге растёт экспоненциально по сравнению с длиной оптимального пути. То есть сложность будет оцениваться как , где b – коэффициент разветвлённости графа, а d – длина кратчайшего пути. Однако сложность становится [полиномиальной](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BD%D0%BE%D0%BC%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C), когда пространство поиска является деревом, а эвристика удовлетворяет следующему условию: |h(x) – h\*(x)| ≤ O(log h\*(x)), где h\* - оптимальная эвристика, а h(x) – величина ошибки. Таким образом, чем лучше будет выбранная эвристика, тем меньше получится коэффициент ветвления, и, следовательно, меньше временная сложность. Затраты по памяти в худшем случае будут оцениваться экспоненциально, так как нам придётся хранить примерно узлов, где |V| - число вершин в графе. В лучшем случае сложность по памяти равна O(|V|+|E|), где |V| - число вершин, а |E| - число рёбер.

В качестве индивидуализации для 6 варианта лабораторной работы было дано задание реализовать очередь с приоритетом с использованием минимальной кучи. Оценка основных операций – нахождение минимального элемента за O(1), удаление и вставка за O(log *n*), где *n* – количество элементов кучи.

Для работы с графом и алгоритмами по поиску кратчайших путей был реализован класс Graph, у которого есть единственное поле:

* *graph* – словарь, содержащий в качестве ключей названия вершин, а в качестве значений словари, содержащие пары смежная вершина – вес ребра до этой вершины.

В качестве реализации исследуемых алгоритмов написаны следующие методы:

* *greedy(start, end)* – метод, реализующий жадный алгоритм, принимает на вход 2 вершины – узел, из которого начинаем поиск (*start*) и конечный узел (*end*). Сначала создаётся список *res*, в который помещается стартовая вершина, затем в переменную *tmp* записывается значение переменной *start* для дальнейшего использования. Далее начинается цикл, в теле которого производится выбор смежной вершины, вес ребра до которого минимален. Полученная вершина добавляется в список *res*, и ребро до неё удаляется. Цикл будет работать до тех пор, пока переменная *tmp* не станет равна переменной *end*.
* *a\_star(start, end)* – метод, реализующий алгоритм А\*, принимает на вход 2 вершины – узел, из которого начинаем поиск (*start*) и конечный узел (*end*). Сначала создаются словарь *distances* длин путей до вершин, словарь *roots*, содержащий в качестве ключей вершины, а в качестве значений соответствующие им родительские вершины, а также очередь с приоритетом *queue*. Из очереди извлекается минимальный элемент (оценка минимальности производится по сумме пути до вершины и значения эвристической функции). Далее начинается цикл, который будет продолжаться до тех пор, пока очередь не опустеет. В теле цикла сначала проводится проверка текущей вершины на равенство с конечной вершиной *end* и проверка на наличие у очередной вершины рёбер, ведущих к другим вершинам. После этого для каждой смежной с текущей вершины производится вычисление суммарного пути и добавление её в словари *distances* и *roots.* После этого в очередь добавляется кортеж из названия вершины и суммы её эвристики и пути. Для полноценной работы необходимы функции, описанные ниже. Метод возвращает словарь *roots*, который в дальнейшем будет использован для формирования пройденного пути.

*heuristic(cur\_vert, end\_vert) –* функция принимает на вход названия двух вершин и считает расстояние между ними в таблице ASCII.

*recover\_path(map, end) –* функция принимает на вход словарь *map* и конечную вершину *end*, после чего формирует пройденный до вершины *end* путь (*map* – словарь, который вернул метод *a\_star(start, end)*).

Для реализации очереди с приоритетом, представленной в виде минимальной двоичной кучи, был написан класс *Heap.* Данный класс имеет одно поле:

* *\_\_heap* – список, содержащий в себе все элементы кучи.

Методы класса *Heap*:

* *sift\_up(index) –* метод, осуществляющий просеивание элемента с индексом *index* вверх. Если элемент больше своего отца, то условие кучи соблюдается и дальнейшее просеивание не требуется. Иначе мы меняем элемент местами со своим родителем и выполняем *sift\_up(index)* для родителя.
* *sift\_down(index) -* метод, осуществляющий просеивание элемента с индексом *index* вниз. Если элемент index меньше, чем его сыновья, всё поддерево уже является кучей, и делать ничего не надо. В противном случае меняем местами элемент index с наименьшим из его сыновей, после чего выполняем *sift\_down(index)* для этого сына.
* *extract\_min() –* метод, извлекающий минимальный элемент из кучи. Первый и последний элементы меняются местами, после чего последний (бывший первый) удаляется из кучи, а первый (бывший последний) просеивается вниз.
* *put(element) –* метод, помещающий элемент в кучу. Изначально элемент добавляется в конец, после чего просеивается вверх.
* *size() –* метод возвращает длину словаря, формирующего кучу.

Код программы см. в приложении А.

**Тестирование.**

Ниже представлено тестирование программы.

Результаты тестирования жадного алгоритма представлены в табл. 1.

Таблица 1 – Результаты прогона программы с разными входными данными.

| Входные данные | Выходные данные | Комментарии |
| --- | --- | --- |
| a e  a b 3.0  b c 1.0  c d 1.0  a d 5.0  d e 1.0 | abcde | Верно |
| 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 | Верно |
| b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0 | bge | Верно |

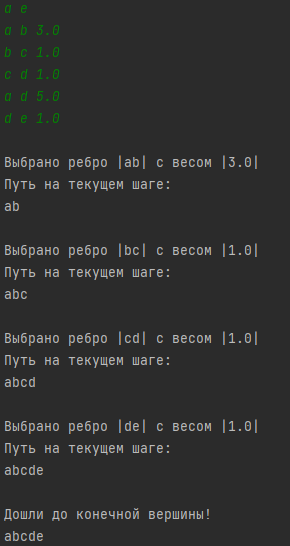
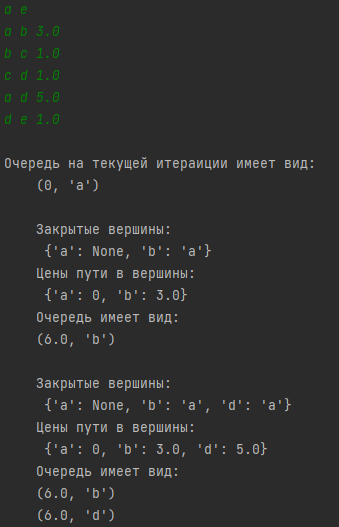


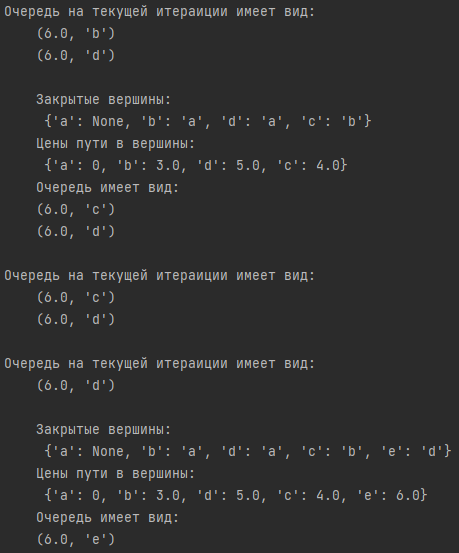
Рисунок 1 – Демонстрация промежуточных выводов для жадного алгоритма.

Результаты тестирования алгоритма А\* представлены в табл. 2.

Таблица 2 – Результаты прогона программы с разными входными данными.

| Входные данные | Выходные данные | Комментарии |
| --- | --- | --- |
| a e  a b 3.0  b c 1.0  c d 1.0  a d 5.0  d e 1.0 | ade | Верно |
| g j  a b 1.0  a f 3.0  b g 3.0  b c 5.0  f g 4.0  c d 6.0  d m 1.0  g e 4.0  g i 5.0  e h 1.0  e n 1.0  n m 2.0  i k 1.0  i j 6.0  j l 5.0  m j 3.0 | genmj | Верно |





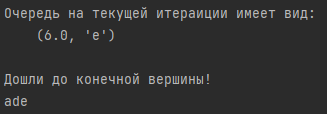


Рисунок 2 – Демонстрация промежуточных выводов для алгоритма А\*.

* 1. **Выводы.**

Написана программа, которая решает задачу построения кратчайшего пути в ориентированном графе при помощи жадного алгоритма и алгоритма А\*. Также была оценена сложность каждого алгоритма.

**ПРИЛОЖЕНИЕ А  
ИСХОДНЫЙ КОД ПРОГРАММЫ**

Название файла: main.py

from Heap import \*

def heuristic(cur\_vert, end\_vert):

return abs(ord(end\_vert) - ord(cur\_vert)) # Расстояние между символами в таблице ASCII

def recover\_path(map, end):

path = ''

current = end

while current: # пока существует вершина из которой пришли

path += current # к пути добавляем текущую

current = map[current] # берем вершину откуда пришли в текущую

return path[::-1] # переворачиваем путь

class Graph:

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

self.graph = dictionary

def greedy(self, start, end):

res = [start] # Список, в который будет записываться решение

tmp = start # Текущая вершина

while tmp != end: # До тех пор, пока текущая вершина не равна конечной

while not self.graph[tmp].keys():

print(f"Из вершины |{tmp}| нет доступных для перехода рёбер. Поднимаемся на одну вершину выше\n")

res.pop(-1) # Удаление вершина без рёбер из ответа

tmp = res[-1] # "Откат" к родительской вершине

minimal = 10000

way = None

for node, val in self.graph[tmp].items():

if minimal > val: # Выбор ребра наименьшего веса

minimal = val

way = node

print(f"Выбрано ребро |{tmp}{way}| с весом |{minimal}|")

self.graph[tmp].pop(way)

res.append(way)

print("Путь на текущем шаге:")

print("".join(res), '\n')

tmp = way

print("Дошли до конечной вершины!")

return res

def a\_star(self, start, end):

distances = dict({start: 0}) # Словарь длин путей до закрытых вершин

roots = dict({start: None}) # Словарь вершин и их родителей

queue = Heap([(0, start)]) # Очередь с приоритетом

while queue.size() != 0: # Пока очередь не пуста

print("Очередь на текущей итераиции имеет вид:")

print(queue)

cur = queue.extract\_min()[1] # Извлечение минимального элемента из кучи

print(f"Сейчас обрабатывается вершина |{cur}|\n")

if cur == end: # Если теущая вершина равна конечной

print('Дошли до конечной вершины!')

break

if not self.graph[cur].keys(): # Если у текущей вершины нет исходящих рёбер

print(f"Вершина |{cur}| является листом графа")

continue

for node, val in self.graph[cur].items():

temp\_dist = distances[cur] + val # Вычисление полного пути до вершины

print(f"Сейчас рассматривается ребро |{cur}{node}|")

if node not in distances or temp\_dist < distances[node]:

roots[node] = cur

distances[node] = temp\_dist

print(f"Антиприоритет вершины |{node}|: {temp\_dist} + {heuristic(node, end)} = {temp\_dist + heuristic(node, end)}\n")

queue.put((temp\_dist + heuristic(node, end), node)) # Помещение вершины в очередь

print("\tЗакрытые вершины:")

print("\t", roots)

print("\tЦены пути в вершины:")

print("\t", distances)

print("\tОчередь имеет вид:")

print(queue)

else:

print(f"Нет необходимости в замене антиприоритета для ребра |{cur}{node}|\n")

return roots

begin, destination = input().split(' ') # Считывание начальной и конечной вершин

config = {}

while True:

try: # Проверка на то, что из входного потока была подана строка

line = input()

except EOFError:

break

if not line:

break

vert, dest, length = line.split(' ') # Считывание тройки: первая вершина ребра, вторая вершина, вес ребра

length = float(length)

if vert not in config.keys():

config[vert] = {}

config[vert][dest] = length # Рразмещение длины ребра в словарь config

config[vert] = dict(sorted(config[vert].items())) # Сортировка рёбер в лексикографическом порядке

if dest not in config.keys():

config[dest] = {}

ex = Graph(config)

# lst = ex.greedy(begin, destination)

# result = "".join(lst)

lst = ex.a\_star(begin, destination)

result = recover\_path(lst, destination)

print(result)

Название файла: Heap.py

class Heap:

def \_\_init\_\_(self, arr=None):

# Инициализация объекта класса

if arr is None:

arr = []

self.\_\_heap = []

for el in arr:

self.put(el)

def sift\_up(self, index):

# Просеивает узел с индексом index вверх

if index < 0 or index >= len(self.\_\_heap):

return

parent = (index - 1) // 2

while index and self.\_\_heap[parent] >= self.\_\_heap[index]:

self.\_\_heap[parent], self.\_\_heap[index] = self.\_\_heap[index], self.\_\_heap[parent]

index, parent = parent, (index - 1) // 2

def sift\_down(self, index):

# Просеивает узел с индексом index вниз

if index < 0 or index >= len(self.\_\_heap):

return None

min\_index = index

while True:

left, right = 2\*index + 1, 2\*index + 2

if right < len(self.\_\_heap) and self.\_\_heap[right] < self.\_\_heap[min\_index]:

min\_index = right

if left < len(self.\_\_heap) and self.\_\_heap[left] < self.\_\_heap[min\_index]:

min\_index = left

if min\_index == index:

return None

else:

self.\_\_heap[index], self.\_\_heap[min\_index] = self.\_\_heap[min\_index], self.\_\_heap[index]

index = min\_index

def extract\_min(self):

# Метод достаёт минимальный элемент из кучи, меняет его местами с максимальным и потом удаляет

if not self.\_\_heap:

return None

min\_element = self.\_\_heap[0]

self.\_\_heap[0] = self.\_\_heap[-1]

del self.\_\_heap[-1]

self.sift\_down(0)

return min\_element

def put(self, element):

# метод добавляет в конец кучи новый элемент, после чего просеивает его вверх

self.\_\_heap.append(element)

self.sift\_up(self.size() - 1)

def size(self):

# Метод возвращает размер кучи

return len(self.\_\_heap)

def \_\_repr\_\_(self):

representation = ""

for item in self.\_\_heap:

representation += '\t' + str(item) + "\n"

return representation