**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: Алгоритм Дейкстры.

|  |  |  |
| --- | --- | --- |
| Студент гр. 8304 |  | Мухин А. М. |
| Преподаватель |  | Размочаева Т. |

Санкт-Петербург

2020

## Цель работы.

Вар. 5. Реализовать алгоритм Дейкстры поиска пути в графе (на основе кода А\*).

## Задание.

## Разработать программу, которая решает задачу поиска кратчайшего пути в ориентированном графе до каждой вершины от текущей с помощью алгоритма Дейкстры. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

**Описание алгоритма.**

Был реализован класс приоритетной очереди, которая хранила объекты в списке и при каждом новом добавлении сортировала его. Для корректной работы алгоритмы были созданы следующие методы: push(item, value), pop(), empty().

Так как необходимо реализовать алгоритм на основе А\*, то необходимо просто в цикле вызвать А\* для каждой конечной вершины. Основная идея алгоритма – это поиск в ширину с приоритетом обрабатываемой вершины. Также во время работы алгоритмы запоминается путь до каждой вершины, правда в обратном порядке. После того, как основной костяк программы завершил работу, восстанавливается путь до текущей вершины и создаётся строка из вершин, по которым надо пройти чтобы достичь необходимую. Вершина и результат записываются в словарь, который в конечном итоге выводится программой.

Также стоит упомянуть о двух вспомогательных функциях. Это same\_value\_in(reverse\_dict), которая проверяет, есть ли в словаре коллизии. И del\_same\_value(reverse\_dict, end\_pos), которая удаляет ключ, которого нет в значениях словаря, притом, что этот ключ не должен равняться значению последнего элемента. Переходя от логики функций с русскому языку, эти функции избавляют нас от путей, которые никуда не ведут. Вначале мы проверяем на наличие таких “отростков”, а потом удаляем их.

**Пример входных данных.**

a

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

В первой строке через указывается начальная вершина. Далее в каждой строке указываются ребра графа и их вес

**Выходные данные.**

В качестве выходных данных представлен словарь, с ключами в качестве вершин, до которых найден кратчайший путь и значениями в качестве кратчайшего пути. В случае, если к какой-то вершине графа нет пути, выведется сообщение об этом.

**Сложность алгоритма.**

Сложность алгоритма по памяти: О(V+E)

Сложность алгоритма по времени: О(n2)

## Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
|  | a  a b 3.0  b c 1.0  c d 1.0  a d 5.0  d e 1.0 | {'a': 0, 'b': 3.0, 'd': 5.0, 'c': 4.0, 'e': 6.0} |
|  | a  a b 1.0  a f 3.0  b c 5.0  b g 3.0  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 j 6.0  i k 1.0  j l 5.0  m j 3.0 | {'a': 0, 'b': 1.0, 'f': 3.0, 'c': 6.0, 'g': 4.0, 'e': 8.0, 'i': 9.0, 'd': 12.0, 'h': 9.0, 'n': 9.0, 'm': 11.0, 'j': 14.0, 'k': 10.0, 'l': 19.0} |
|  | j  b a 1.0  f a 3.0  c b 5.0  g b 3.0  g f 4.0  d c 6.0  m d 1.0  e g 4.0  i g 5.0  h e 1.0  n e 1.0  m n 2.0  j i 6.0  k i 1.0  l j 5.0  j l 5.0  j m 3.0 | you can't achieve this vertex: h  you can't achieve this vertex: k  {'b': 'jmnegb', 'f': 'jmnegf', 'c': 'jmdc', 'g': 'jmneg', 'd': 'jmd', 'm': 'jm', 'e': 'jmne', 'i': 'ji', 'h': 'jmneg', 'n': 'jmn', 'k': 'jmneg', 'l': 'jl', 'a': 'jmnegba', 'j': 'j'} |

## Вывод.

В ходе подготовки в данной лабораторной работе и её выполнение мы узнали, что такое жадные алгоритмы. Реализовали такие алгоритмы как А\*, алгоритм Дейкстры и жадный алгоритм.

# Приложение А Исходный код программы

Название файла: main.cpp

from sys import stdin

class PriorityQueue:

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

self.repository = []

def push(self, item, value):

self.repository.append((item, value))

self.repository.sort(key=lambda pair: pair[1])

def pop(self):

return self.repository.pop(0)

def empty(self):

return len(self.repository) == 0

def del\_same\_value(reverse\_dict, end\_pos):

for key in list(reverse\_dict):

if key not in reverse\_dict.values() and key != end\_pos:

del reverse\_dict[key]

def same\_value\_in(reverse\_dict):

for value in list(reverse\_dict.values()):

if list(reverse\_dict.values()).count(value) > 1:

return True

return False

start\_position = input()

graph = {}

result = {}

for line in stdin:

source, destination, value = line.split()

if source not in graph:

graph[source] = {destination: float(value)}

else:

graph[source].update({destination: float(value)})

vertexes = [vertex for vertex in graph if vertex != start\_position]

for key in graph:

for vertex in graph[key]:

if vertex not in vertexes:

vertexes.append(vertex)

for end\_position in vertexes:

queue = PriorityQueue()

queue.push(start\_position, 0)

path\_to = {start\_position: 0}

came\_from = {start\_position: None}

while not queue.empty():

current\_element = queue.pop()

if current\_element[0] == end\_position:

break

if current\_element[0] in graph:

for neighbour in graph[current\_element[0]]:

path\_length = path\_to[current\_element[0]] + graph[current\_element[0]][neighbour]

if neighbour not in path\_to or path\_length <= path\_to[neighbour]:

path\_to[neighbour] = path\_length

queue.push(neighbour, path\_length)

came\_from[neighbour] = current\_element[0]

while same\_value\_in(came\_from):

del\_same\_value(came\_from, end\_position)

reverse\_list = {}

for key in came\_from:

reverse\_list[came\_from[key]] = key

path = start\_position

while path[-1] != end\_position:

try:

path += reverse\_list[path[-1]]

except KeyError:

print(f'you can\'t achieve this vertex: {end\_position}')

break

result[end\_position] = path

print(result)