

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: Алгоритмы на графах

Студентка гр. 9382

Балаева М.О

Преподаватель

Фирсов М.А

Санкт-Петербург

2021

Цель работы.

Изучение, сравнение и реализация алгоритмов поиска пути во взвешенных графах.

Задание Вар. 9. Вывод графического представления графа.

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.

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

2) 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
```

Описание алгоритма.

1. Жадный алгоритм.

Изначально имеется список не посещенных вершин. Алгоритм находит минимальное по весу и не посещенную смежную для текущей вершины. Если в процессе выполнения алгоритма из рассматриваемой вершины нет никаких ребер, или все вершины отмечены посещенными, или она не конечная, данная вершина отмечается посещенной и удаляется от текущего пути. Когда алгоритм доходит до конечной вершины, она записывается в ответ и алгоритм завершается.

2.Алгоритм A^* .

Имеется список всех вершин графа. Выполнение алгоритма начинается с записи начальной вершины в путь, последующие вершины выбираются в зависимости от значения функции: $f(v)=g(v)+h(v)$, где v – текущая вершина, $g(v)$ - наименьшая стоимость пути в v из стартовой вершины, $h(v)$ — в данном случае: близость символов, обозначающих вершины графа, в таблице ASCII(см. Задание). Для смежных вершин текущей выбираем вершину наименьшим значением $f(v)$. Если в процессе выполнения алгоритма будет найден более короткий путь, текущий путь будет заменен на него. Если текущая вершина является конечной, она записывается в путь и алгоритм завершается.

Сложность алгоритма.

1.Сложность жадного алгоритма:

1)Сложность алгоритма по времени:

Сложность данного алгоритма $O(V+E)$, где V - количество вершин, а E -ребра графа, т.к в худшем случае придется пройти весь граф.

2)Сложность алгоритма по памяти:

Сложность алгоритма по памяти $O(E)$, где E — ребра графа, т.к хранятся ребра графа.

2.Сложность алгоритма A^* :

1)Сложность алгоритма по времени:

В лучшем случае эвристическая функция позволяет совершать шаг в верном направлении, то есть будет $O(V+E)$, где E — количество ребер графа, а V — количество вершин графа.

2)Сложность алгоритма по памяти:

В худшем случае придется пройти через все вершины графа, то есть число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, то есть $O(V*(E+V))$, где E — количество ребер графа, а V — количество вершин графа.

Описание функций и СД.

1. Жадный алгоритм.

class Edge — класс, который хранит в себе информацию о ребрах графа

src: str — начальная вершина

dest: str — конечная вершина

weight: float — вес ребра

dead: bool — посещена ли конечная вершина (не конечная, заданная по условию, а конечная для текущего ребра). Т.к все вершины в графе, кроме начальной (заданной по условию) являются конечной для определенных ребер, данная переменная однозначно отображает посещена ли вершина графа.

def __init__(self, src: str, dest: str, weight: float, dead: bool) — конструктор класса Edge.

def get_minimal(graph: typing.List, src: str) — функция, находящая индекс минимальной смежной непосещенной вершины для данной (src).

Аргументы:

src: str — текущая вершина графа

graph: typing.List — граф (массив данных типа Edge).

def print_recursion(message, count) — вспомогательная функция для промежуточных данных.

Аргументы:

message — текст

count — количество отступов

def get_way(graph: typing.List, qwert: typing.List[str], src: str, dest: str, pr) -> bool — основная функция для выполнения жадного алгоритма.

Аргументы:

graph: typing.List — граф (массив данных типа Edge).

qwert: typing.List[str] — путь (ответ)

src: str — начальная вершина ребра

dest: str — конечная вершина ребра (В программе идет проверка является ли эта вершина конечной, если да : функция возвращает True)

pr — счетчик отступов

`__main__`

`src, dest = input().split()` - функция получает несколько входных данных
разбивает данный ввод по пробелу.

`graph = []` - граф (массив данных типа Edge)

`qwert = []` - путь

`edges = []` - ребра графа

Представление графа:

`G = nx.DiGraph()`- Создание пустой структуры графа («нулевой граф») без узлов и ребер.

`G.add_edge(graph[i].src, graph[i].dest, weight=graph[i].weight)` — для всех ребер происходит добавление в граф(названия, вес , направление)

`pos =nx.spring_layout(G)` — Словарь с узлами в качестве ключей и позициями в качестве значений.

`nx.draw_networkx_nodes(G, pos, cmap=plt.get_cmap('jet'), node_color='r', node_size=500)` - рисует только узлы графа G. `node_color='r'` — цвет, `node_size=500` — размер узлов.

`nx.draw_networkx_labels(G, pos)` — функция рисует метки узлов на графе

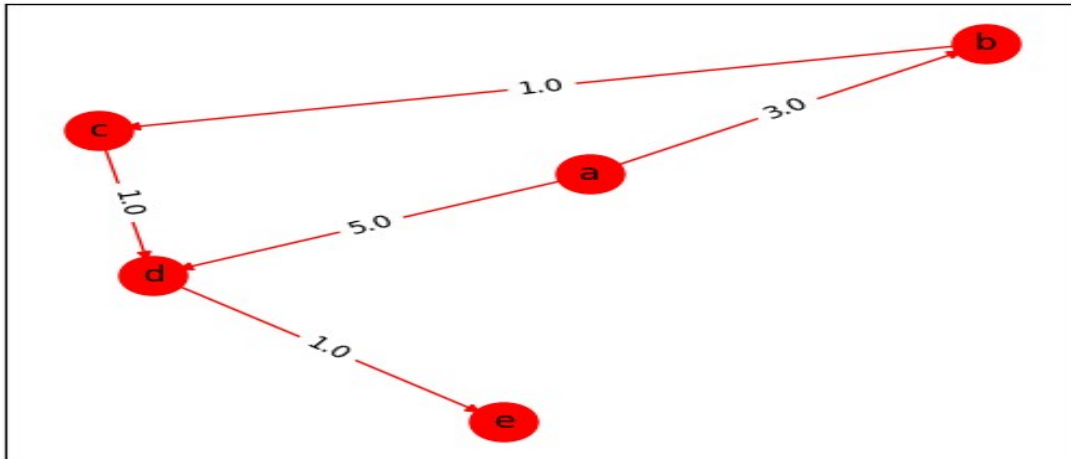
`nx.draw_networkx_edges(G, pos, edgelist=edges, edge_color='r', arrows=True)` — метод, рисующий только ребра графа

`labels = nx.get_edge_attributes(G, 'weight')` —Словарь атрибутов с ключом по краю.

`nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)`- рисует метки на ребрах

`plt.show()`-показывает граф

Пример вывода графа:



Для ДАННЫХ:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

Алгоритм A*:

class Edge - Класс ребра графа. Содержит в себе всю необходимую информацию о ребрах графа

def __init__(self, src: str, dest: str, weight: int): - конструктор класса Edge

src — начальная вершина ребра

dest — конечная вершина ребра

weight — вес ребра

class Vertex — класс вершины графа

def __init__(self, name: str, src: typing.Any, f: int, g: int):

name- наименование вершины

src — наименование смежной ей. Необходимо для дальнейшего выполнения алгоритма

self.f = f — $f(v)$ (см. Описание алгоритма A*)

self.g = g — $g(v)$ (см. Описание алгоритма A*)

def minimal_f(q: typing.List[Vertex]) -> int: функция для нахождения индекса минимального ребра. Изначально проверяется равенство $f(v)$ текущей минимальной вершины с каждой вершиной графа: $q[i].f == \text{minimal.f}$, если это оказывается так , сравниваются их $h(v)$. Функция вернет значение индекса.

def A1(graph: typing.List[Edge], start: str, end: str) — основная функция для выполнения алгоритма.

Аргументы:

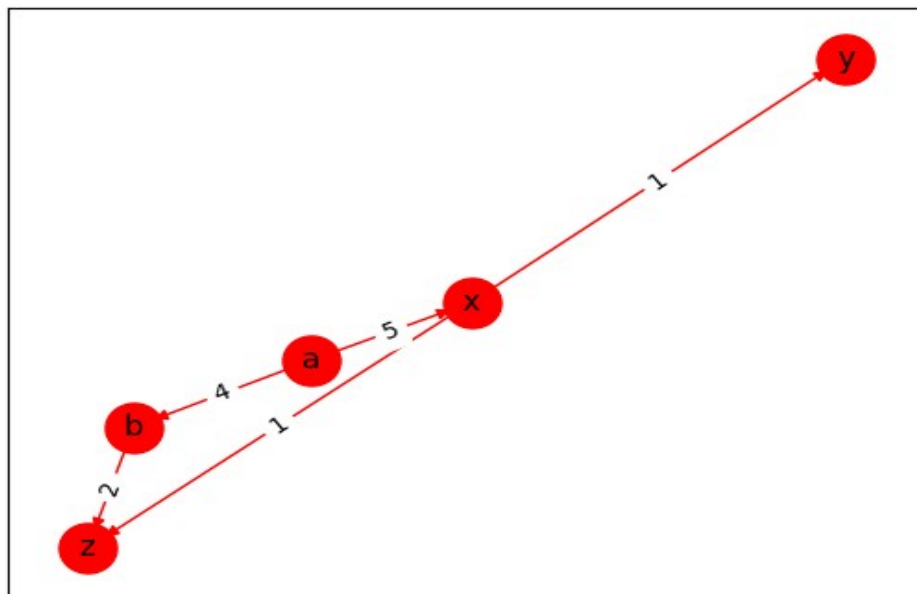
graph: typing.List[Edge] — граф (массив элементов класса Edge.

start: str- начальная вершина графа

end: str- конечная вершина графа

__main__ - ввод данны, отрисовка графа такие же, как и в жадном алгоритме.

Пример вывода графа:



Для

данных:

a z

a x 5.0

x y 1.0

x z 1.0

a b 4.0

b z 2.0

Тестирование

Входные данные	Выходные данные
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	A*: ag Жадный алгоритм: abdefg
a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0	A*: ad Жадный алгоритм: abcd

e d 3.0	
a h a b 1 a c 2 b d 5 b g 10 b e 4 c e 2 c f 1 d g 2 e d 1 e g 7 f e 3 f h 8 g h 1	A*: acedgh Жадный алгоритм: abedgh
a b a b 1	A*: ab Жадный алгоритм: ab
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	A*: ade Жадный алгоритм: abcde

Вывод.

Были изучены жадный алгоритм и алгоритм A^* . Реализована программа для поиска пути в ориентированном графе.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
import sys
```

```
import typing
```

```
import networkx as nx
```

```
import numpy.random as rnd
```

```
import matplotlib.pyplot as plt
```

```
import pylab
```

```
class Edge:
```

```
    src: str
```

```
    dest: str
```

```
    weight: float
```

```
    dead: bool
```

```
    def __init__(self, src: str, dest: str, weight: float, dead: bool):
```

```
        self.src = src
```

```
        self.dest = dest
```

```
        self.weight = weight
```

```
        self.dead = dead
```

```
def get_minimal(graph: typing.List, src: str) -> int:
```

```
    k = -1
```

```
    minimal = float("+inf")
```

```
    for i, node in enumerate(graph):
```

```
        if not node.dead and node.src == src and node.weight < minimal:
```

```

        minimal = graph[i].weight
        k = i

    return k

def print_recursion(message, count):
    msg = "
    for val in range(0, count):
        msg += " "
    print(msg + message)

def get_way(graph: typing.List, qwert: typing.List[str], src: str, dest: str, pr) -> bool:
    print_recursion('Вход в рекурсию', pr)
    print_recursion("Добавляем " + src + " в массив пути(в ответ)", pr)
    qwert.append(src)

    if src == dest:
        print_recursion('Выход из рекурсии', pr)
        return True

    print_recursion("Находим ребро с минимальным весом для данной вершины"
+ src + "", pr)
    k = get_minimal(graph, src)
    m = str(graph[k].weight)
    print_recursion("Минимальное ребро:" + src + graph[k].dest + " с весом: " + m,
pr)
    while k != -1:
        if get_way(graph, qwert, graph[k].dest, dest, pr + 1):
            print_recursion('Выход из рекурсии', pr)

```

```

        return True

    print_recursion("Удаляем вершину"+ qwert[len(qwert)-1] + " из массива
ответа( из ответа) вершин", pr)

    qwert.pop(len(qwert) - 1)

    print_recursion("Отмечаем вершину" + graph[k].dest + " посещенной", pr)
    graph[k].dead = True

    k = get_minimal(graph, src)

print_recursion('Выход из рекурсии', pr)
return False


if __name__ == "__main__":
    src, dest = input().split()

    graph = []
    qwert = []
    edges = []

    data = sys.stdin.readline()
    while data.strip():
        _src, _dest, _weight = data.split()
        _weight = float(_weight)
        edge = Edge(_src, _dest, _weight, False)
        graph.append(edge)
        edges.append((_src, _dest))
        data = sys.stdin.readline()

    way = get_way(graph, qwert, src, dest, 0)
    print("Ответ:" + ".join(qwert))

```

```
# РИСОВАНИЕ ГРАФА
```

```
G = nx.DiGraph()
for i in range(0, len(graph)):
    G.add_edge(graph[i].src, graph[i].dest, weight=graph[i].weight)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, cmap=plt.get_cmap('jet'),
                        node_color='r', node_size=500)
nx.draw_networkx_labels(G, pos)
nx.draw_networkx_edges(G, pos, edgelist=edges, edge_color='r', arrows=True)
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.show()
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: A1.py

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

```
class Edge:
```

```
    def __init__(self, src: str, dest: str, weight: int):
        self.src = src
        self.dest = dest
        self.weight = weight
```

```
class Vertex:
```

```
    def __init__(self, name: str, src: typing.Any, f: int, g: int):
        self.name = name
        self.src = src
        self.f = f
        self.g = g

    def __str__(self) -> str:
        if self.src:
```



```
        return str(self.src) + self.name
    return self.name
```

```
def minimal_f(q: typing.List[Vertex]) -> int:
```

```
    k = 0
```

```
    minimal = q[k]
```

```
    for i in range(1, len(q)):
```

```
        if q[i].f == minimal.f:
```

```
            if minimal.f - minimal.g > q[i].f - q[i].g:
```

```
                minimal = q[i]
```

```
                k = i
```

```
        elif q[i].f < minimal.f:
```

```
            minimal = q[i]
```

```
            k = i
```

```
    return k
```

```
def A1(graph: typing.List[Edge], start: str, end: str):
```

```
    q = []
```

```
    u = []
```

```
    begin = Vertex(start, None, abs(ord(end) - ord(start)), 0)
```

```
    q.append(begin)
```

```
    print("Записываем " + begin.name + " в список рассматриваемых  
вершин")
```

```

while q:
    index = minimal_f(q)
    current = q[index]
    if current.name == end:
        print("Ответ: " + str(current))
        return True
    print("Удаляем " + "" + q[index].name + "" + " из списка рассматриваемых
вершин")
    q.pop(index)
    print("Записываем " + "" + current.name + "" + " в список посещенных
вершин")
    u.append(current)

for i in range(len(graph)):
    if current.name == graph[i].src:
        print("Находим смежную вершину для " + "" + current.name + "" + " с
наименьшей эвристикой и наименьшим весом")
        neighbor = Vertex(graph[i].dest, current,
                           abs(ord(end) - ord(graph[i].dest)) + graph[i].weight + current.g,
                           graph[i].weight + current.g)
        print("Выбрана вершина: " + "" + neighbor.name + "")
        l = str(graph[i].weight)
        print("Вес текущего ребра " + "" + current.name + neighbor.name + "" +
" равен: " + l)
        if neighbor in u:
            continue

        if neighbor not in q:
            q.append(neighbor)

```

```

        print("Записываем " + "" + neighbor.name + "" + " в список
рассматриваемых вершин")
    else:
        index_neighbor = q.index(neighbor)

        if neighbor.g < q[index_neighbor].g:
            print("Заменяем " + "" + q[index_neighbor].name + " на " + current
+ "")

            q[index_neighbor].src = current
            q[index_neighbor].g = neighbor.g
            q[index_neighbor].f = neighbor.f

    return False

```

```

if __name__ == "__main__":
    begin, last = input().split()
    graph = []
    edges = []
    data = sys.stdin.readline()
    while data.strip():
        src, dest, weight = data.split()
        weight = int(float(weight))
        graph.append(Edge(src, dest, weight))
        edges.append((src, dest))
        data = sys.stdin.readline()
    G = nx.DiGraph()
    for i in range(0, len(graph)):
        G.add_edge(graph[i].src, graph[i].dest, weight=graph[i].weight)
    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, cmap=plt.get_cmap('jet'),

```

```
        node_color='r', node_size=500)
nx.draw_networkx_labels(G, pos)
nx.draw_networkx_edges(G, pos, edgelist=edges, edge_color='r', arrows=True)
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.show()
```

```
A1(graph, begin, last)
```