Міністерство освіти та науки України

Національний технічний університет України

«Київський Політехнічний Інститут ім. Ігоря Сікорського»

ФТІ

Кафедра ФТЗЗІ

**Лабораторна робота №10**

з дисципліни: «Програмування 4»

# на тему:

# «Індивідуальні завдання»

# 

Виконав:

Студент групи ФЕ-81

Кривчук Назарій

Перевірив:

Доцент Прогонов Д.О.

Київ 2020

**Мета роботи:** Використати добуті знання протягом курсу “Програмування 4” .

**Завдання:** В неориентированном графе найти наибольший и наименьший путь между вершинами (топология графа задается пользователем).

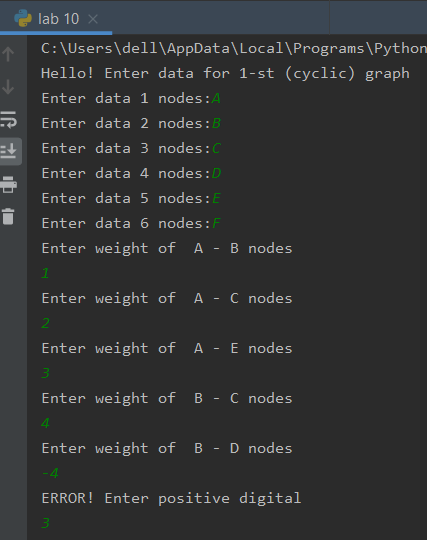
**Пояснення:** Для знаходження найкоротшого щляху я використав Алгоритм Дейкстри, який може бути застосований до будь якого графа (циклічного,ациклічного) з невід’ємними значеннями ваги ребра. В нашому випадку перший граф буде циклічним , так як для знаходження найдовшого цей граф не підходить створимо ще один другий граф тільки вже ациклічний.

Для знаходження найдовшого шляху я використав Алгоритм BFS(пошук в ширину) який дієвий тільки для ациклічних графів.

**Код реалізації:**

class Node:  
  
 def \_\_init\_\_(self, data, indexloc=None):  
 self.data = data  
 self.index = indexloc  
  
  
class Graph:  
  
 @classmethod  
 def create\_from\_nodes(self, nodes):  
 return Graph(len(nodes), len(nodes), nodes)  
  
 def \_\_init\_\_(self, row, col, nodes=None):  
 # установка матрица смежности  
 self.adj\_mat = [[0] \* col for \_ in range(row)]  
 self.nodes = nodes  
 for i in range(len(self.nodes)):  
 self.nodes[i].index = i  
  
 # Связывает node1 с node2  
 # Обратите внимание, что ряд - источник, а столбец - назначение  
 def connect\_dir(self, node1, node2, weight=1):  
 node1, node2 = self.get\_index\_from\_node(node1), self.get\_index\_from\_node(node2)  
 self.adj\_mat[node1][node2] = weight  
  
 # Опциональный весовой аргумент для поддержки алгоритма Дейкстры  
 def connect(self, node1, node2, weight=1):  
 self.connect\_dir(node1, node2, weight)  
 self.connect\_dir(node2, node1, weight)  
  
 # Получает ряд узла, отметить ненулевые объекты с их узлами в массиве self.nodes  
 # Выбирает любые ненулевые элементы, оставляя массив узлов  
 # которые являются connections\_to (для ориентированного графа)  
 # Возвращает значение: массив кортежей (узел, вес)  
 def connections\_from(self, node):  
 node = self.get\_index\_from\_node(node)  
 return [(self.nodes[col\_num], self.adj\_mat[node][col\_num]) for col\_num in range(len(self.adj\_mat[node])) if  
 self.adj\_mat[node][col\_num] != 0]  
  
 # Проводит матрицу к столбцу узлов  
 # Проводит любые ненулевые элементы узлу данного индекса ряда  
 # Выбирает только ненулевые элементы  
 # Обратите внимание, что для неориентированного графа  
 # используется connections\_to ИЛИ connections\_from  
 # Возвращает значение: массив кортежей (узел, вес)  
 def connections\_to(self, node):  
 node = self.get\_index\_from\_node(node)  
 column = [row[node] for row in self.adj\_mat]  
 return [(self.nodes[row\_num], column[row\_num]) for row\_num in range(len(column)) if column[row\_num] != 0]  
  
 def print\_adj\_mat(self):  
 for row in self.adj\_mat:  
 print(row)  
  
 def node(self, index):  
 return self.nodes[index]  
  
 def remove\_conn(self, node1, node2):  
 self.remove\_conn\_dir(node1, node2)  
 self.remove\_conn\_dir(node2, node1)  
  
 # Убирает связь в направленной манере (nod1 к node2)  
 # Может принять номер индекса ИЛИ объект узла  
 def remove\_conn\_dir(self, node1, node2):  
 node1, node2 = self.get\_index\_from\_node(node1), self.get\_index\_from\_node(node2)  
 self.adj\_mat[node1][node2] = 0  
  
 # Может пройти от node1 к node2  
  
 def can\_traverse\_dir(self, node1, node2):  
 node1, node2 = self.get\_index\_from\_node(node1), self.get\_index\_from\_node(node2)  
 return self.adj\_mat[node1][node2] != 0  
  
 def has\_conn(self, node1, node2):  
 return self.can\_traverse\_dir(node1, node2) or self.can\_traverse\_dir(node2, node1)  
  
 def add\_node(self, node):  
 self.nodes.append(node)  
 node.index = len(self.nodes) - 1  
 for row in self.adj\_mat:  
 row.append(0)  
 self.adj\_mat.append([0] \* (len(self.adj\_mat) + 1))  
  
 # Получает вес, представленный перемещением от n1  
 # к n2. Принимает номера индексов ИЛИ объекты узлов  
 def get\_weight(self, n1, n2):  
 node1, node2 = self.get\_index\_from\_node(n1), self.get\_index\_from\_node(n2)  
 return self.adj\_mat[node1][node2]  
  
 # Разрешает проводить узлы ИЛИ индексы узлов  
 def get\_index\_from\_node(self, node):  
 if not isinstance(node, Node) and not isinstance(node, int):  
 raise ValueError("node must be an integer or a Node object")  
 if isinstance(node, int):  
 return node  
 else:  
 return node.index  
  
 def dijkstra(self, node):  
 # Получает индекс узла (или поддерживает передачу int)  
 nodenum = self.get\_index\_from\_node(node)  
 # Заставляет массив отслеживать расстояние от одного до любого узла  
 # в self.nodes. Инициализирует до бесконечности для всех узлов, кроме  
 # начального узла, сохраняет "путь", связанный с расстоянием.  
 # Индекс 0 = расстояние, индекс 1 = перескоки узла  
 dist = [None] \* len(self.nodes)  
 for i in range(len(dist)):  
 dist[i] = [float("inf")]  
 dist[i].append([self.nodes[nodenum]])  
  
 dist[nodenum][0] = 0  
 # Добавляет в очередь все узлы графа  
 # Отмечает целые числа в очереди, соответствующие индексам узла  
 # локаций в массиве self.nodes  
 queue = [i for i in range(len(self.nodes))]  
 # Набор увиденных на данный момент номеров  
 seen = set()  
 while len(queue) > 0:  
 # Получает узел в очереди, который еще не был рассмотрен  
 # и который находится на кратчайшем расстоянии от источника  
 min\_dist = float("inf")  
 min\_node = None  
 for n in queue:  
 if dist[n][0] < min\_dist and n not in seen:  
 min\_dist = dist[n][0]  
 min\_node = n  
  
 # Добавляет мин. расстояние узла до увиденного, убирает очередь  
 queue.remove(min\_node)  
 seen.add(min\_node)  
 # Получает все следующие перескоки  
 connections = self.connections\_from(min\_node)  
 # Для каждой связи обновляет путь и полное расстояние от  
 # исходного узла, если полное расстояние меньше  
 # чем текущее расстояние в массиве dist  
 for (node, weight) in connections:  
 tot\_dist = weight + min\_dist  
 if tot\_dist < dist[node.index][0]:  
 dist[node.index][0] = tot\_dist  
 dist[node.index][1] = list(dist[min\_node][1])  
 dist[node.index][1].append(node)  
 return dist  
  
 def bfs(self, node):  
 dist = [[] for i in range(len(self.nodes))]  
 for it in dist:  
 it = []  
 visited = set()  
 queue = [node]  
 while True:  
 visited.add(node)  
  
 nodenum = self.get\_index\_from\_node(node)  
  
 conn = self.connections\_from(node)  
 connections = []  
 for elem in conn:  
 if not visited.\_\_contains\_\_(elem[0]):  
 connections.append(elem)  
 if len(connections) == 0:  
 if len(dist[nodenum]) == 0:  
 for i in queue:  
 dist[nodenum].append(i)  
 queue.pop()  
 if len(queue) == 0:  
 return dist  
 node = queue[-1]  
 continue  
  
 for (i, weight) in connections:  
 node = i  
 queue.append(node)  
 break  
  
print("Hello! Enter data for 1-st (cyclic) graph ")  
n1 = Node(input("Enter data 1 nodes:"))  
n2 = Node(input("Enter data 2 nodes:"))  
n3 = Node(input("Enter data 3 nodes:"))  
n4 = Node(input("Enter data 4 nodes:"))  
n5 = Node(input("Enter data 5 nodes:"))  
n6 = Node(input("Enter data 6 nodes:"))  
  
w\_graph = Graph.create\_from\_nodes([n1, n2, n3, n4, n5, n6])  
  
print("Enter weight of ", n1.data, "-", n2.data, "nodes")  
x = int(input())  
if x < 0:  
 print("ERROR! Enter positive digital")  
 x = int(input())  
else:  
 w\_graph.connect(n1, n2, x)  
  
print("Enter weight of ", n1.data, "-", n3.data, "nodes")  
x = int(input())  
if x < 0:  
 print("ERROR! Enter positive digital")  
 x = int(input())  
else:  
 w\_graph.connect(n1, n3, x)  
  
print("Enter weight of ", n1.data, "-", n5.data, "nodes")  
x = int(input())  
if x < 0:  
 print("ERROR! Enter positive digital")  
 x = int(input())  
else:  
 w\_graph.connect(n1, n5, x)  
  
print("Enter weight of ", n2.data, "-", n3.data, "nodes")  
x = int(input())  
if x < 0:  
 print("ERROR! Enter positive digital")  
 x = int(input())  
else:  
 w\_graph.connect(n2, n3, x)  
  
print("Enter weight of ", n2.data, "-", n4.data, "nodes")  
x = int(input())  
if x < 0:  
 print("ERROR! Enter positive digital")  
 x = int(input())  
else:  
 w\_graph.connect(n2, n4, x)  
  
print("Enter weight of ", n3.data, "-", n4.data, "nodes")  
x = int(input())  
if x < 0:  
 print("ERROR! Enter positive digital")  
 x = int(input())  
else:  
 w\_graph.connect(n3, n4, x)  
  
print("Enter weight of ", n3.data, "-", n6.data, "nodes")  
x = int(input())  
if x < 0:  
 print("ERROR! Enter positive digital")  
 x = int(input())  
else:  
 w\_graph.connect(n3, n6, x)  
  
print("Enter weight of ", n4.data, "-", n5.data, "nodes")  
x = int(input())  
if x < 0:  
 print("ERROR! Enter positive digital")  
 x = int(input())  
else:  
 w\_graph.connect(n4, n5, x)  
  
print("Our graph:")  
w\_graph.print\_adj\_mat()  
  
print("The shortest way:")  
print([(weight, [n.data for n in node]) for (weight, node) in w\_graph.dijkstra(n1)])  
  
print("Hello! Enter data for 2-st (acyclic) graph ")  
c1 = Node(input("Enter data 1 nodes:"))  
c2 = Node(input("Enter data 2 nodes:"))  
c3 = Node(input("Enter data 3 nodes:"))  
c4 = Node(input("Enter data 4 nodes:"))  
c5 = Node(input("Enter data 5 nodes:"))  
c6 = Node(input("Enter data 6 nodes:"))  
c7 = Node(input("Enter data 7 nodes:"))  
c8 = Node(input("Enter data 8 nodes:"))  
  
c\_graph = Graph.create\_from\_nodes([c1, c2, c3, c4, c5, c6,c7,c8])  
  
print("Enter weight of ", c1.data, "-", c2.data, "nodes")  
c\_graph.connect(c1, c2, int(input()))  
  
print("Enter weight of ", c1.data, "-", c5.data, "nodes")  
c\_graph.connect(c1, c5, int(input()))  
  
print("Enter weight of ", c2.data, "-", c3.data, "nodes")  
c\_graph.connect(c2, c3, int(input()))  
  
print("Enter weight of ", c2.data, "-", c4.data, "nodes")  
c\_graph.connect(c2, c4, int(input()))  
  
print("Enter weight of ", c4.data, "-", c8.data, "nodes")  
c\_graph.connect(c3, c8, int(input()))  
  
print("Enter weight of ", c5.data, "-", c6.data, "nodes")  
c\_graph.connect(c5, c6, int(input()))  
  
print("Enter weight of ", c5.data, "-", c7.data, "nodes")  
c\_graph.connect(c5, c7, int(input()))  
  
print("Our 2-nd acyclic graph:")  
c\_graph.print\_adj\_mat()  
  
print("The longest way:")  
  
dest = c\_graph.bfs(c1)  
print([[n.data for n in node] for node in dest])

**Приклад виконання програми:**

****

