**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ЛАБОРАТОРНАЯ РАБОТА №4

по дисциплине: **«Оптимизация проектных решений»**

на тему: **«Поиск кратчайшего пути. Алгоритм Дейкстры»**

Выполнил: студент гр. ИТП-41

Керпеченко Н.В.

Принял: преподаватель

Мурашко И. А.

Гомель 2020

**Цель**: знакомство с методами решения оптимизационных задач на графах.

**Ход работы**

1. Изучить изложенный метод поиска кратчайшего пути (Алгоритм Дейкстры).

2. Составить программу для поиска кратчайшего пути на основе алгоритма Дейкстры.

3. Оформить отчет о выполнении задания с приведением условия задачи, алгоритма и программного кода, результаты работы программы и выводы по результатам.

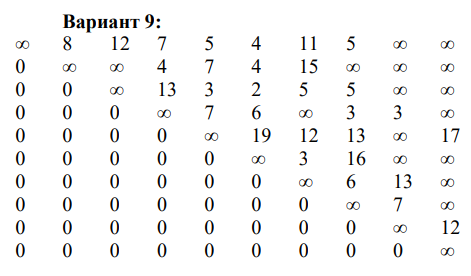


Рисунок 1 – Индивидуальное задание

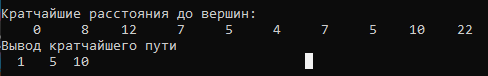


Рисунок 2 – Результат нахождения кратчайшего пути

**Вывод:** в ходе выполнения лабораторной работы 4 были изучены и разработаны алгоритмы для решения задачи Дейкстры. Освоены основные навыки применения алгоритмов для данного типа задач.

**ПРИЛОЖЕНИЕ А**

**Листинг программы**

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

# Обновлен для поддержки взвешенных ребер (поддержка алгоритма Дейкстры)

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

nodes = [Node(str(i)) for i in range(1, 11)]

graph = Graph.create\_from\_nodes(nodes)

positions = [(1, 9), (3, 6), (3, 9), (4, 10), (5, 8), (6, 10), (7, 9), (8, 9), (9, 10)]

for i in range(9):

s, e = positions[i]

for j in range(s, e):

if s == 4 and e == 10:

if (j == 6):

continue

graph.connect(nodes[i], nodes[j])

graph.print\_adj\_mat()

w\_graph = Graph.create\_from\_nodes(nodes)

mt = [

[8, 12, 7, 5, 4, 1, 5, 13],

[4, 7, 4],

[13, 3, 2, 5, 9, 5],

[7, 6, 0, 3, 3, 8],

[19, 12, 13],

[3, 16, 12, 11],

[3, 13],

[7],

[12]

]

for i in range(9):

s, e = positions[i]

for index, m in enumerate(mt[i], start=s):

w\_graph.connect(nodes[i], nodes[index], m)

w\_graph.print\_adj\_mat()

print([(weight, [n.data for n in node]) for (weight, node) in w\_graph.dijkstra(nodes[1])])