#1

import heapq

def dijkstra(graph, start):

distances = {node: float('inf') for node in graph}

distances[start] = 0

heap = [(0, start)]

while heap:

current\_distance, current\_node = heapq.heappop(heap)

if current\_distance > distances[current\_node]:

continue

for neighbor, weight in graph[current\_node].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(heap, (distance, neighbor))

return distances

# Пример графа в виде словаря с весами ребер

graph = {

'A': {'B': 3, 'C': 1},

'B': {'A': 3, 'C': 7, 'D': 5},

'C': {'A': 1, 'B': 7, 'D': 2},

'D': {'B': 5, 'C': 2}

}

start\_node = 'A'

shortest\_paths = dijkstra(graph, start\_node)

print("Кратчайшие пути от вершины", start\_node, ":")

for node, distance in shortest\_paths.items():

print(f"До вершины {node} расстояние: {distance}")

#2

import heapq

def dijkstra\_shortest\_path(graph, start, end):

# Инициализация расстояний

distances = {vertex: float('inf') for vertex in graph}

distances[start] = 0

# Очередь с приоритетом для обработки вершин

priority\_queue = [(0, start)]

while priority\_queue:

current\_distance, current\_vertex = heapq.heappop(priority\_queue)

# Если достигли конечной вершины

if current\_vertex == end:

return distances[end]

# Просмотр соседей текущей вершины

for neighbor, weight in graph[current\_vertex].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

return -1 # Если путь не найден

# Пример взвешенного графа в виде матрицы смежности

graph = {

'A': {'B': 3, 'C': 2},

'B': {'A': 3, 'C': 7, 'D': 4},

'C': {'A': 2, 'B': 7, 'D': 1},

'D': {'B': 4, 'C': 1}

}

start\_vertex = 'A'

end\_vertex = 'D'

shortest\_path = dijkstra\_shortest\_path(graph, start\_vertex, end\_vertex)

if shortest\_path != -1:

print(f"Кратчайший путь между вершинами {start\_vertex} и {end\_vertex}: {shortest\_path}")

else:

print(f"Путь между вершинами {start\_vertex} и {end\_vertex} не найден.")

#3

def floyd\_warshall(graph):

num\_vertices = len(graph)

distances = [[float('inf') for \_ in range(num\_vertices)] for \_ in range(num\_vertices)]

# Инициализация матрицы кратчайших путей

for i in range(num\_vertices):

for j in range(num\_vertices):

if i == j:

distances[i][j] = 0

elif graph[i][j] != 0:

distances[i][j] = graph[i][j]

# Алгоритм Флойда-Уоршелла

for k in range(num\_vertices):

for i in range(num\_vertices):

for j in range(num\_vertices):

distances[i][j] = min(distances[i][j], distances[i][k] + distances[k][j])

return distances

# Пример взвешенного графа в виде матрицы смежности (0 означает отсутствие ребра)

graph\_matrix = [

[0, 3, 0, 0],

[0, 0, 7, 4],

[2, 0, 0, 1],

[0, 0, 0, 0]

]

all\_shortest\_paths = floyd\_warshall(graph\_matrix)

print("Матрица кратчайших путей между всеми парами вершин:")

for row in all\_shortest\_paths:

print(row)

#4

def floyd\_warshall(graph):

num\_vertices = len(graph)

# Инициализация матрицы кратчайших путей и предшественников

distances = [[float('inf') for \_ in range(num\_vertices)] for \_ in range(num\_vertices)]

for i in range(num\_vertices):

for j in range(num\_vertices):

if i == j:

distances[i][j] = 0

elif graph[i][j] != 0:

distances[i][j] = graph[i][j]

# Алгоритм Флойда-Уоршелла

for k in range(num\_vertices):

for i in range(num\_vertices):

for j in range(num\_vertices):

distances[i][j] = min(distances[i][j], distances[i][k] + distances[k][j])

return distances

# Пример взвешенного графа в виде матрицы смежности (0 означает отсутствие ребра)

graph\_matrix = [

[0, 3, 0, 8],

[0, 0, 2, 0],

[0, 0, 0, 4],

[0, 0, 0, 0]

]

all\_shortest\_paths = floyd\_warshall(graph\_matrix)

for row in all\_shortest\_paths:

print(row)

#5

import sys

def prim\_mst(graph):

num\_vertices = len(graph)

mst = [None] \* num\_vertices # Каркас минимального веса

key = [sys.maxsize] \* num\_vertices

visited = [False] \* num\_vertices

key[0] = 0 # Начинаем с первой вершины

for \_ in range(num\_vertices):

min\_key = sys.maxsize

min\_index = -1

# Находим вершину с минимальным значением ключа, еще не включенную в каркас

for v in range(num\_vertices):

if key[v] < min\_key and not visited[v]:

min\_key = key[v]

min\_index = v

visited[min\_index] = True

# Добавляем вершину к каркасу

if mst[min\_index] is not None:

# Ваша логика здесь для добавления ребра в каркас минимального веса

pass

# Обновляем значения ключей для смежных вершин

for v in range(num\_vertices):

if graph[min\_index][v] > 0 and not visited[v] and graph[min\_index][v] < key[v]:

key[v] = graph[min\_index][v]

mst[v] = min\_index

return mst

# Пример взвешенного графа в виде матрицы смежности

graph\_matrix = [

[0, 2, 0, 6, 0],

[2, 0, 3, 8, 5],

[0, 3, 0, 0, 7],

[6, 8, 0, 0, 9],

[0, 5, 7, 9, 0]

]

minimum\_spanning\_tree = prim\_mst(graph\_matrix)

# В minimum\_spanning\_tree будет содержаться каркас минимального веса

#6

def complement\_graph(graph):

num\_vertices = len(graph)

complement = [[0 for \_ in range(num\_vertices)] for \_ in range(num\_vertices)]

for i in range(num\_vertices):

for j in range(num\_vertices):

if i != j and graph[i][j] == 0:

complement[i][j] = 1

elif i != j and graph[i][j] == 1:

complement[i][j] = 0

return complement

# Пример исходного графа в виде матрицы смежности

graph = [

[0, 1, 0, 1],

[1, 0, 1, 0],

[0, 1, 0, 1],

[1, 0, 1, 0]

]

complement = complement\_graph(graph)

print("Дополнение графа:")

for row in complement:

print(row)

#7

def is\_complete\_graph(graph):

num\_vertices = len(graph)

for i in range(num\_vertices):

for j in range(num\_vertices):

if i != j and graph[i][j] != 1:

return False # Нашли несоединенные вершины, граф не полный

return True # Все вершины соединены, граф полный

# Пример графа в виде матрицы смежности

graph = [

[0, 1, 1, 1],

[1, 0, 1, 1],

[1, 1, 0, 1],

[1, 1, 1, 0]

]

result = is\_complete\_graph(graph)

if result:

print("Граф является полным.")

else:

print("Граф не является полным.")

#8

from collections import deque

def find\_sources(graph):

sources = []

num\_vertices = len(graph)

for i in range(num\_vertices):

is\_source = True

for j in range(num\_vertices):

if graph[i][j] == 0 and i != j:

is\_source = False

break

if is\_source:

sources.append(i)

return sources

def find\_sinks(graph):

sinks = []

num\_vertices = len(graph)

for i in range(num\_vertices):

is\_sink = True

for j in range(num\_vertices):

if graph[j][i] == 0 and i != j:

is\_sink = False

break

if is\_sink:

sinks.append(i)

return sinks

# Пример графа в виде матрицы смежности

graph = [

[0, 0, 1, 0],

[1, 0, 0, 0],

[0, 0, 0, 0],

[0, 1, 1, 0]

]

sources = find\_sources(graph)

sinks = find\_sinks(graph)

print("Источники графа:", sources)

print("Стоки графа:", sinks)