**Лабораторная работа №3.**

**Код**

**1.**Алгоритм Дейкстры

def dejkstra(graph, start):

- определяю функцию dejkstra, которая принимает два параметра: graph (граф в виде словаря) и start (начальная вершина).

Инициализация

distances = {vertex: math.inf for vertex in graph}  
distances[start] = 0  
visited = set()  
priority\_queue = [(0, start)] # (расстояние, вершина)

- создаю словарь distances, где каждому ключу (вершине) соответствует значение бесконечности. Для стартовой вершины устанавливаю расстояние равное 0.

- создаю множество visited для отслеживания посещенных вершин.

- инициализирую приоритетную очередь priority\_queue, которая будет хранить кортежи (расстояние, вершина).

Основной цикл

while priority\_queue:  
 current\_distance, current\_vertex = min(priority\_queue) # получаю вершину с минимальным расстоянием  
 priority\_queue.remove((current\_distance, current\_vertex)) # удаляю её из очереди  
  
 if current\_vertex in visited:  
 continue  
  
 visited.add(current\_vertex)

- на каждой итерации извлекаю вершину с минимальным расстоянием и удаляю её из очереди.

- если текущая вершина уже была посещена, пропускаю её, чтобы избежать повторных вычислений.

Обновление расстояний

for neighbor, weight in graph[current\_vertex]:  
 distance = current\_distance + weight  
  
 if distance < distances[neighbor]:  
 distances[neighbor] = distance  
 priority\_queue.append((distance, neighbor)) # добавляю соседей в очередь

- прохожу по всем соседям текущей вершины и вычисляю новое расстояние до каждого соседа.

- если новое расстояние меньше ранее известного, обновляю его и добавляю соседа в приоритетную очередь для дальнейшего анализа.

Возврат результата

return distances

- после завершения работы алгоритма возвращаем словарь distances, который содержит кратчайшие расстояния от стартовой вершины до всех других вершин графа.

**2.**Пример графа для алгоритма Дейкстры

graph = {  
 0: [(1, 4), (2, 1)],  
 1: [(3, 1)],  
 2: [(1, 2), (3, 5)],  
 3: []  
}  
  
start\_vertex = 0  
distances = dejkstra(graph, start\_vertex)  
  
print("Кратчайшие расстояния от вершины", start\_vertex)  
for vertex, distance in distances.items():  
 print(f"Вершина {vertex}: {distance}")

- создаю пример графа в виде словаря смежности. Каждая вершина связана с соседями и весами рёбер.

- запускаю алгоритм Дейкстры и вывожу кратчайшие расстояния от стартовой вершины.

**3.**Алгоритм Беллмана-Форда

def bellman\_ford(graph, start):

- определяю функцию bellman\_ford, которая также принимает два параметра: graph и start.

Инициализация

distances = {vertex: math.inf for vertex in graph}  
distances[start] = 0

Основной цикл

for \_ in range(len(graph) - 1):  
 for vertex in graph:  
 for neighbor, weight in graph[vertex]:  
 if distances[vertex] + weight < distances[neighbor]:  
 distances[neighbor] = distances[vertex] + weight

- выполняю внешний цикл |V| - 1 раз (где |V| — количество вершин), чтобы обновить расстояния.

- во вложенных циклах прохожу по всем вершинам и их соседям. Если нахожу более короткий путь к соседу, обновляю его расстояние.

Проверка на отрицательные циклы

for vertex in graph:  
 for neighbor, weight in graph[vertex]:  
 if distances[vertex] + weight < distances[neighbor]:  
 raise ValueError("Граф содержит отрицательный цикл")

- после основного цикла проверяю наличие отрицательных циклов. Если можно улучшить расстояние к любой из вершин, это означает наличие отрицательного цикла.

Возврат результата

return distances

- возвращаю словарь с кратчайшими расстояниями от стартовой вершины.

**4.**Пример графа для алгоритма Беллмана-Форда

graph\_bf = {  
 0: [(1, 4), (2, 1)],  
 1: [(3, 1)],  
 2: [(1, 2), (3, 5)],  
 3: []  
}  
  
start\_vertex\_bf = 0  
distances\_bf = bellman\_ford(graph\_bf, start\_vertex\_bf)  
  
print("Кратчайшие расстояния от вершины", start\_vertex\_bf)  
for vertex, distance in distances\_bf.items():  
 print(f"Вершина {vertex}: {distance}")

- создаю пример графа для алгоритма Беллмана-Форда и вывожу результаты.

**5.**Визуализация графа

def visualize\_graph(graph, shortest\_paths, start):  
 with open("graph.dot", "w") as f:  
 f.write("digraph G {\n")  
 for vertex in graph:  
 for neighbor, weight in graph[vertex]:  
 f.write(f" {vertex} -> {neighbor} [label={weight}];\n")  
  
 # Выделяем кратчайшие пути  
 for vertex in shortest\_paths:  
 if shortest\_paths[vertex] != math.inf and vertex != start:  
 f.write(f" {start} -> {vertex} [color=red];\n")  
  
 f.write("}\n")

- определяю функцию visualize\_graph, которая принимает граф, кратчайшие пути и стартовую вершину.

- создаю файл graph.dot для записи графа в формате DOT. Записываю рёбра графа и выделяю кратчайшие пути красным цветом.

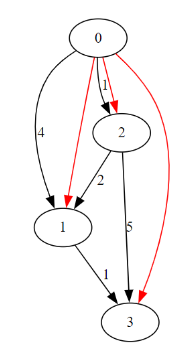
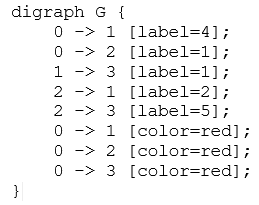
Визуализация графа с использованием алгоритма Дейкстры

visualize\_graph(graph, distances, start\_vertex)

- вызываю функцию visualize\_graph, чтобы создать файл graph.dot, который можно использовать для визуализации графа и кратчайших путей.

**Результаты**

Алгоритм Дейкстры



Алгоритм Беллмана-Форда

