**Міністерство освіти і науки України**

**Національний університет «Запорізька Політехніка»**

Кафедра програмних засобів

**ЗВІТ**

з лабораторної роботи №6

з дисципліни «Алгоритми та Структури Даних» на тему:

«Алгоритми пошуку найкоротших шляхів»

**Виконав:**

Студент групи КНТ-122 О. А. Онищенко

**Прийняли:**

Старший викладач: Л. Ю. Дейнега

2023

**Алгоритми пошуку найкоротших шляхів**

**Мета роботи**

Вивчити основні алгоритми пошуку найкоротших шляхів в графах: алгоритми Дейкстри, Флойда-Воршелла та Беллмана-Форда. Навчитися застосовувати алгоритми пошуку найкоротшого шляху в графі для розв'язання практичних задач.

**Завдання до роботи**

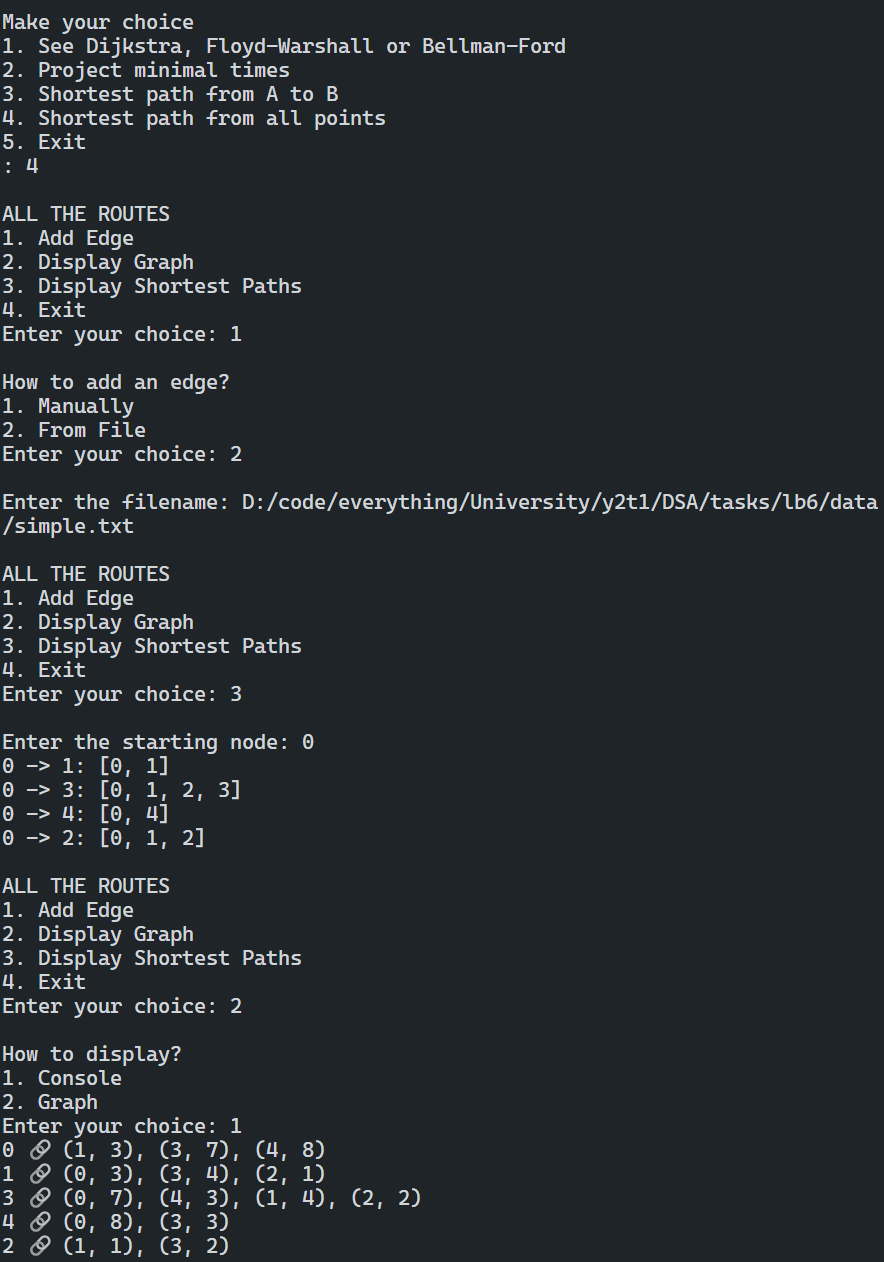
- Розробити програмне забезпечення, в якому реалізується алгоритми Дейкстри, Флойда-Уоршелла та Беллмана-Форда на основі створеного відповідного класу для виконання всіх необхідних обчислень, визначення параметрів (в тому числі безпосередньо визначення графа) та отримання та виведення результатів.

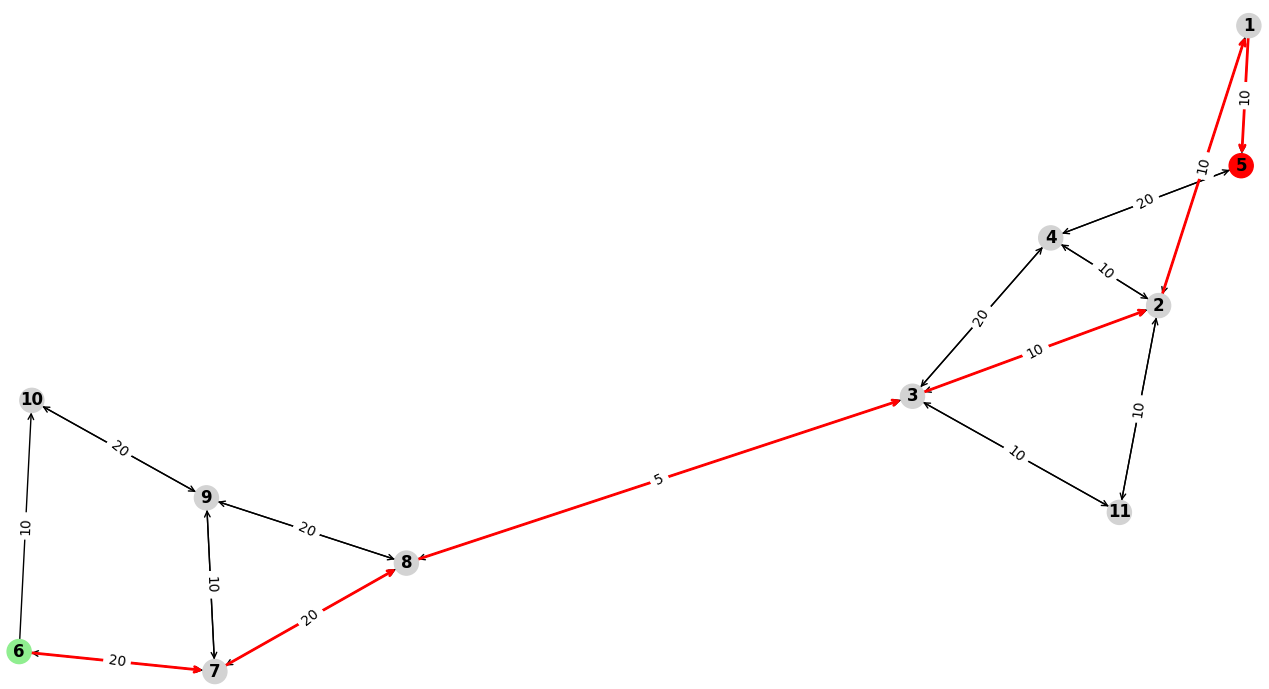
- Перелік завдань, які мають бути виконані в процесі роботи над деяким проєктом групою спеціалістів, та їх запланована тривалість визначаються користувачем. Після цього користувач визначає зв’язок між даними завданнями, визначаючи завдання, які мають завершитися до початку кожного завдання. Визначити мінімальний період часу, який знадобиться на виконання проєкту.

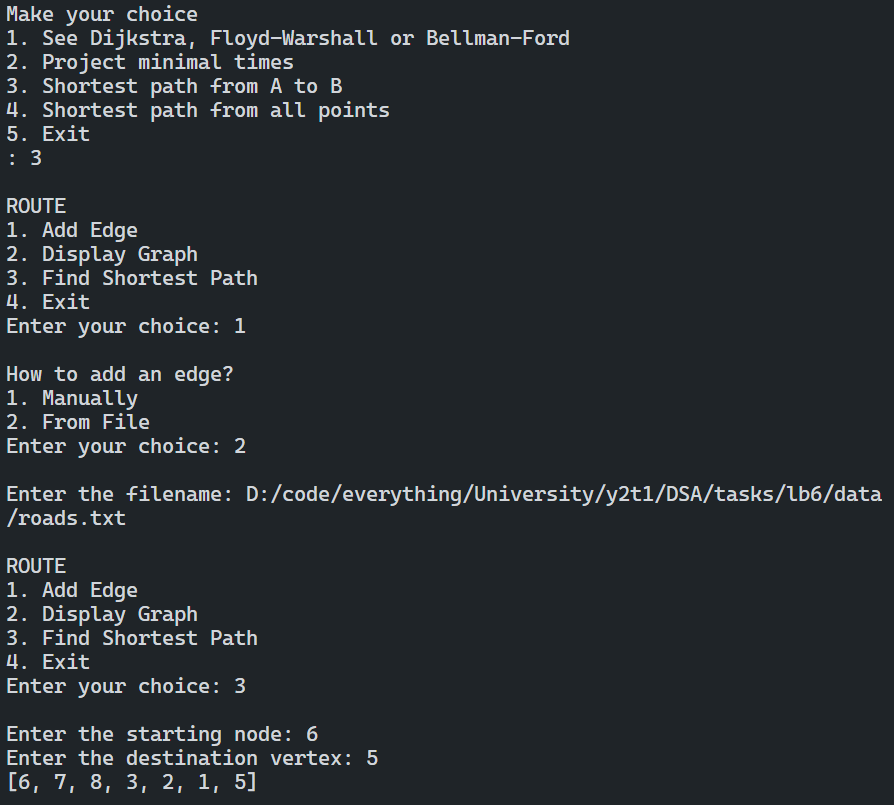
- Мапа визначає автомобільні шляхи деякої частини міста Запоріжжя. Деякі вулиці мають односторонній рух, а на деяких можуть зустрічатися затори. Використовуючи дану інформацію та враховуючи обмеження швидкості на вулицях, визначити найкоротший шлях, яким можна дістатися з однієї заданої точки у Запоріжжі до іншої в заданий момент часу.

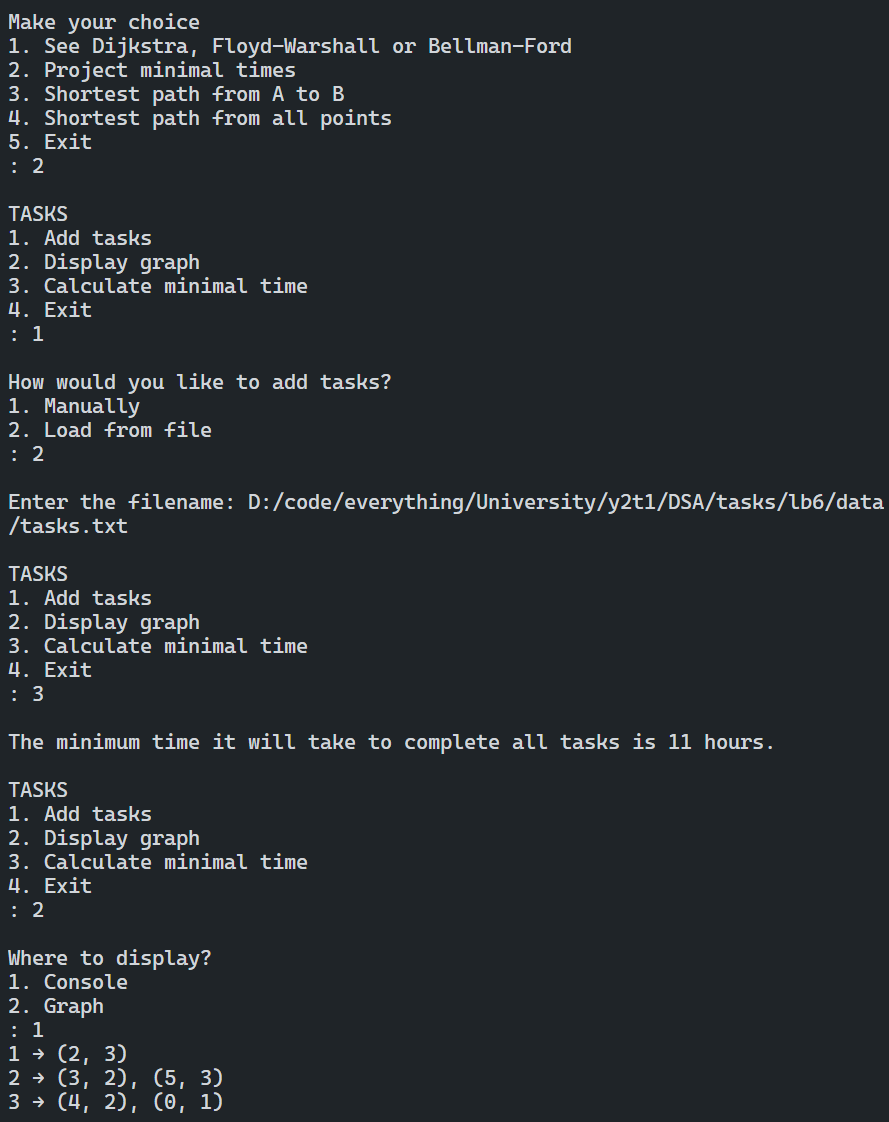
- Визначити найкоротші шляхи між всіма точками на мапі міста Запоріжжя, використовуючи обмеження попереднього завдання.

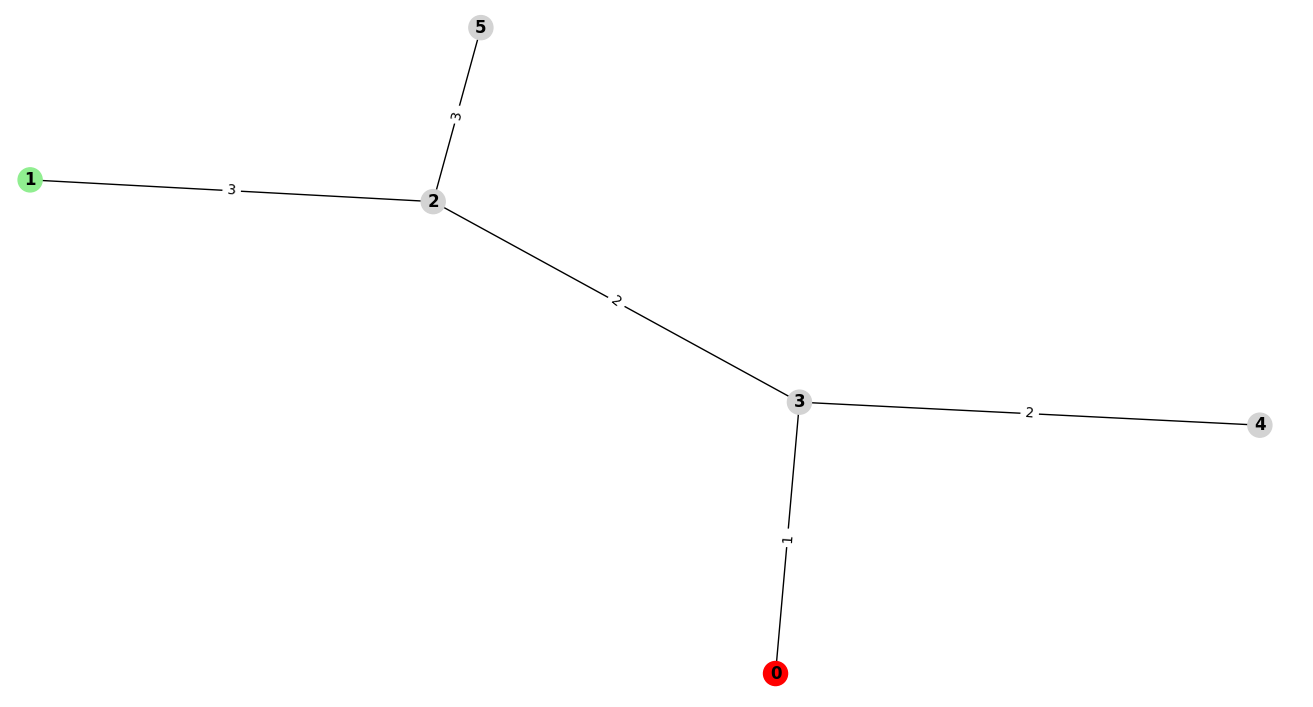
**Результати виконання роботи**

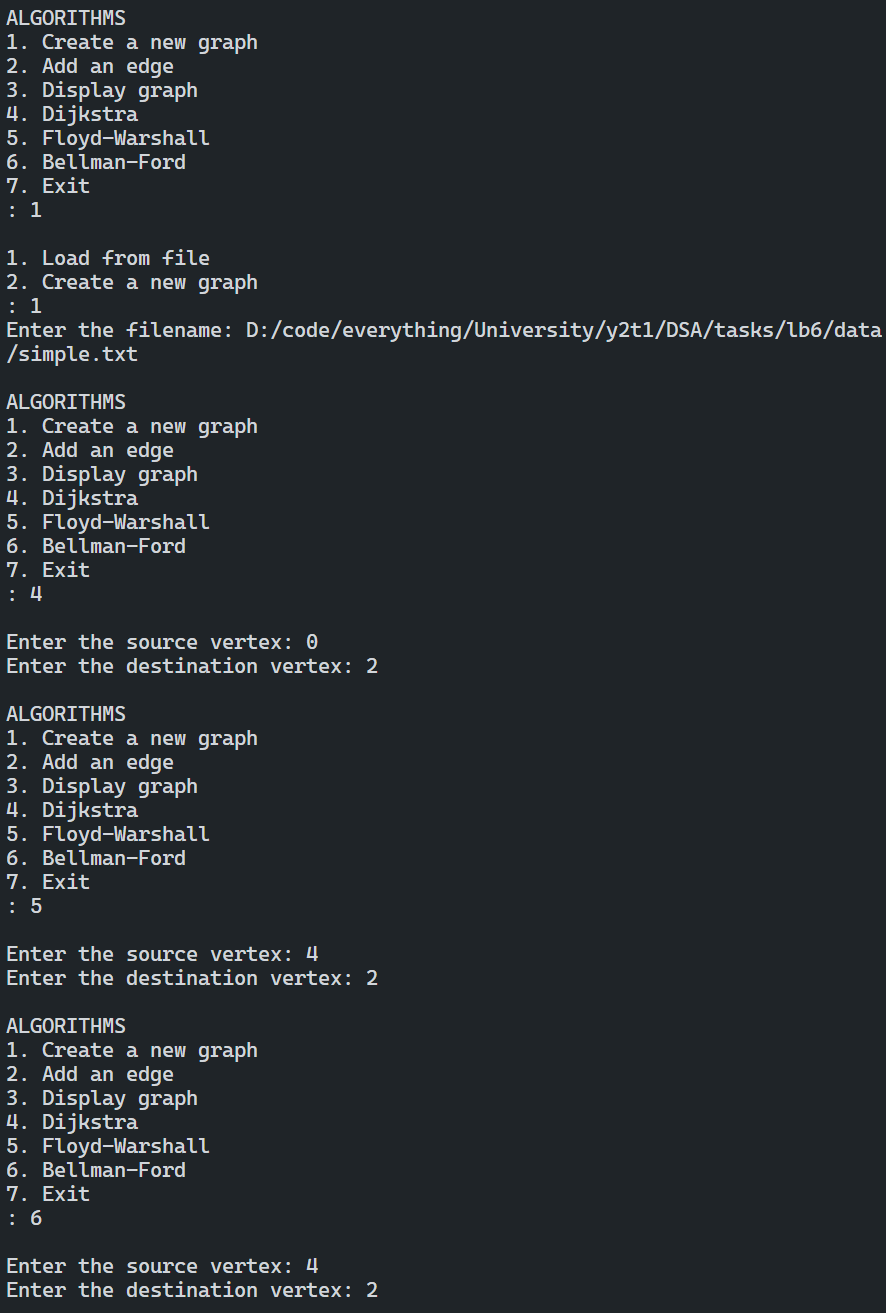


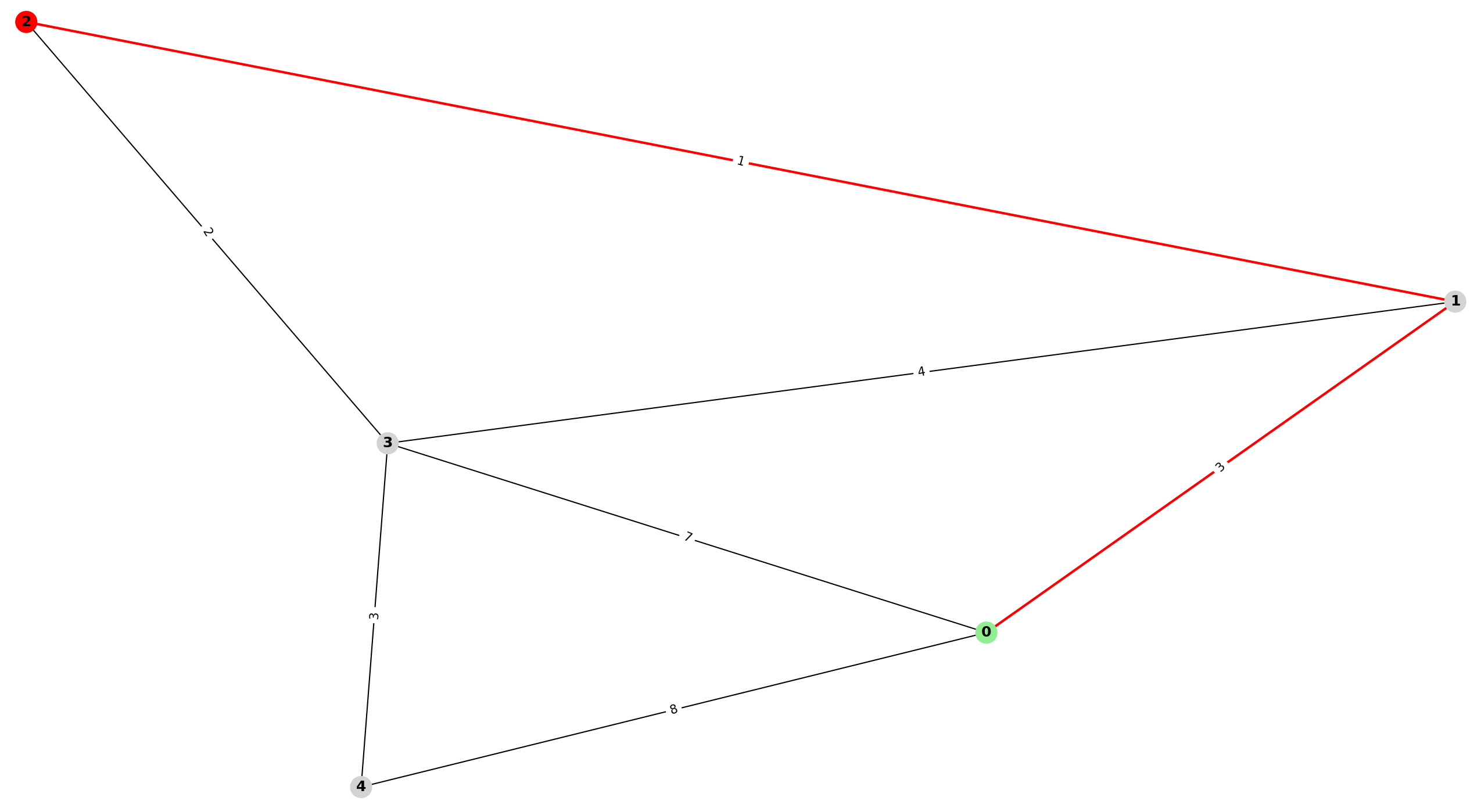


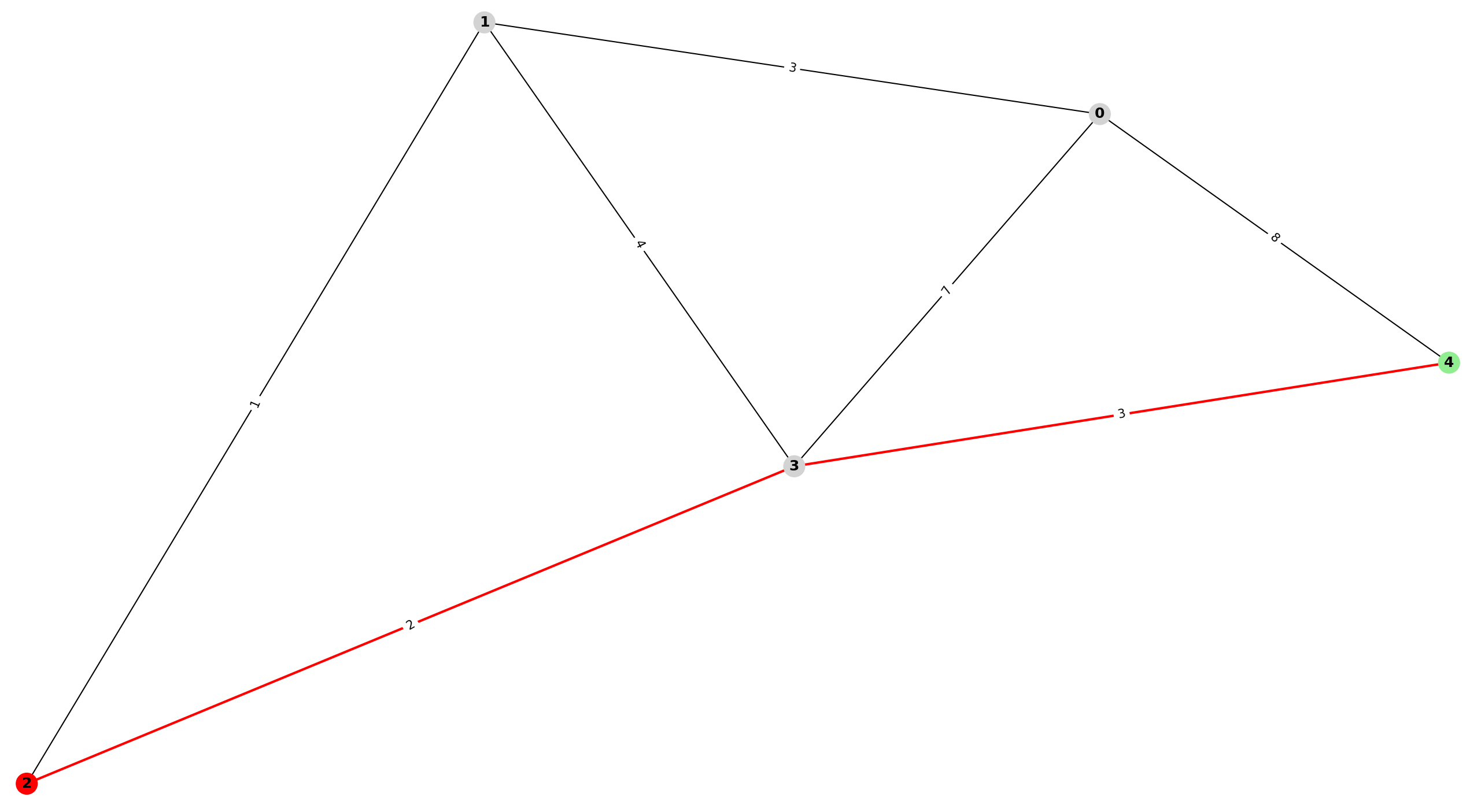












**Код**

from collections import defaultdict

from heapq import heappop, heappush

import matplotlib.pyplot as plt

import networkx as nx

def algorithms():

    class Graph:

        def \_\_init\_\_(self, directed=False):

            self.edges = defaultdict(list)

            self.directed = directed

        def addEdge(self, u, v, weight):

            self.edges[u].append((v, weight))

            if not self.directed:

                self.edges[v].append((u, weight))

        def displayGraph(self):

            for key, value in self.edges.items():

                if self.directed:

                    print(f"{key} → {', '.join(map(str, value))}")

                else:

                    print(f"{key} 🔗 {', '.join(map(str, value))}")

        def drawGraph(self, shortest\_path=None):

            G = nx.Graph()

            for u, edges in self.edges.items():

                for v, weight in edges:

                    G.add\_edge(u, v, weight=weight)

            pos = nx.spring\_layout(G, k=0.15)

            if shortest\_path:

                node\_colors = [

                    "red"

                    if node == shortest\_path[-1]

                    else "lightgreen"

                    if node == shortest\_path[0]

                    else "lightgray"

                    for node in G.nodes()

                ]

                nx.draw\_networkx\_nodes(G, pos, node\_color=node\_colors)

                plt.text(

                    0.00,

                    1.13,

                    "Red: End",

                    transform=plt.gca().transAxes,

                    fontsize=10,

                    verticalalignment="top",

                    bbox=dict(boxstyle="round", facecolor="red", alpha=0.5),

                )

                plt.text(

                    0.00,

                    1.06,

                    "Green: Start",

                    transform=plt.gca().transAxes,

                    fontsize=10,

                    verticalalignment="top",

                    bbox=dict(boxstyle="round", facecolor="lightgreen", alpha=0.5),

                )

            else:

                nx.draw\_networkx\_nodes(G, pos, node\_color="lightblue")

            nx.draw\_networkx\_edges(G, pos)

            nx.draw\_networkx\_labels(G, pos, font\_weight="bold")

            if shortest\_path:

                edges = [

                    (shortest\_path[i], shortest\_path[i + 1])

                    for i in range(len(shortest\_path) - 1)

                ]

                nx.draw\_networkx\_edges(

                    G, pos, edgelist=edges, edge\_color="red", width=2

                )

            labels = nx.get\_edge\_attributes(G, "weight")

            nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels)

            plt.show()

        def loadFromFile(self, filename):

            with open(filename, "r") as file:

                for line in file:

                    try:

                        u, v, w = line.strip().split()

                        self.addEdge(int(u), int(v), int(w))

                    except ValueError:

                        print(f"Skipping line {line}")

        def floydWarshall(self):

            # Initialize a defaultdict to store the shortest distances between nodes

            dist = defaultdict(lambda: defaultdict(lambda: float("inf")))

            # Initialize a defaultdict to store the next node in the shortest path

            nextNode = defaultdict(dict)

            # Iterate over each node in the graph

            for u in self.edges:

                # Set the distance from a node to itself as 0

                dist[u][u] = 0

                # Iterate over each neighbor of the current node

                for v, weight in self.edges[u]:

                    # Set the distance to the neighbor node

                    dist[u][v] = weight

                    # Set the next node in the shortest path to be the neighbor node

                    nextNode[u][v] = v

            # Iterate over each intermediate node

            for k in self.edges:

                # Iterate over each source node

                for i in self.edges:

                    # Iterate over each destination node

                    for j in self.edges:

                        # If the distance from i to k plus the distance from k to j is smaller than the current distance from i to j

                        if dist[i][k] + dist[k][j] < dist[i][j]:

                            # Update the distance from i to j

                            dist[i][j] = dist[i][k] + dist[k][j]

                            # Update the next node in the shortest path from i to j

                            nextNode[i][j] = nextNode[i][k]

            # Return the shortest distances and next nodes

            return dist, nextNode

        def shortestPath(self, start, end):

            # Run the Floyd-Warshall algorithm to get the nextNode matrix

            distances, nextNode = self.floydWarshall()

            # If there is no path from start to end, return None

            if nextNode[start][end] is None:

                return None

            # Create a list to store the path

            path = [start]

            # Traverse the nextNode matrix to find the shortest path from start to end

            while start != end:

                start = nextNode[start][

                    end

                ]  # Update the current node to the next node in the path

                path.append(start)  # Add the current node to the path

            # Return the shortest path

            return path

        def dijkstra(self, start, end):

            # Initialize a priority queue with a tuple containing the cost, the current node, and the path taken so far

            queue = [(0, start, [])]

            # Create an empty set to keep track of visited nodes

            seen = set()

            # Continue looping while there are nodes in the queue

            while queue:

                # Pop the node with the lowest cost from the queue

                (cost, node, path) = heappop(queue)

                # Check if the node has been visited before

                if node not in seen:

                    # Mark the node as visited

                    seen.add(node)

                    # Append the current node to the path taken so far

                    path = path + [node]

                    # Check if the current node is the destination node

                    if node == end:

                        # Return the path taken

                        return path

                    # Iterate over the edges of the current node

                    for nextNode, c in self.edges[node]:

                        # Check if the next node has not been visited

                        if nextNode not in seen:

                            # Calculate the new cost by adding the cost of the current node and the cost of the edge to the next node

                            newCost = cost + c

                            # Add the next node to the queue with the updated cost and path

                            heappush(queue, (newCost, nextNode, path))

            # If the destination node cannot be reached, return an empty path

            return []

        def bellmanFord(self, start, end):

            # Initialize distance dictionary with all nodes set to infinity

            distance = {node: float("infinity") for node in self.edges}

            # Set the distance of the starting node to 0

            distance[start] = 0

            # Initialize predecessor dictionary with all nodes set to None

            predecessor = {node: None for node in self.edges}

            # Iterate (number of nodes - 1) times

            for \_ in range(len(self.edges) - 1):

                # Iterate over each node in the graph

                for node in self.edges:

                    # Iterate over each neighbor and weight of the current node

                    for neighbour, weight in self.edges[node]:

                        # If the distance from the starting node to the current node plus the weight of the edge is smaller than the current distance to the neighbor node, update the distance and predecessor dictionaries

                        if distance[node] + weight < distance[neighbour]:

                            distance[neighbour] = distance[node] + weight

                            predecessor[neighbour] = node

            # Check for negative-weight cycles by iterating over each node and its neighbors

            for node in self.edges:

                for neighbour, weight in self.edges[node]:

                    # If the distance from the starting node to the current node plus the weight of the edge is smaller than the current distance to the neighbor node, it means there is a negative-weight cycle in the graph

                    assert (

                        distance[node] + weight >= distance[neighbour]

                    ), "Graph contains a negative-weight cycle"

            # Build the path from the end node to the start node by following the predecessors

            path = []

            current\_node = end

            while current\_node is not None:

                path.append(current\_node)

                current\_node = predecessor[current\_node]

            # Reverse the path to get the correct order

            path.reverse()

            # Return the shortest path from start to end

            return path

    def main():

        g = None

        while True:

            print("\nALGORITHMS")

            print("1. Create a new graph")

            print("2. Add an edge")

            print("3. Display graph")

            print("4. Dijkstra")

            print("5. Floyd-Warshall")

            print("6. Bellman-Ford")

            print("7. Exit")

            choice = int(input(": "))

            print()

            if choice == 1:

                print("1. Load from file")

                print("2. Create a new graph")

                choice = int(input(": "))

                if choice == 1:

                    filename = input("Enter the filename: ")

                    g = Graph()

                    g.loadFromFile(filename)

                elif choice == 2:

                    directed = input("Is the graph directed? (y/n): ") == "y"

                    g = Graph(directed)

            elif choice == 2:

                u, v, weight = map(

                    int, input("Enter the vertices and weight: ").split()

                )

                g.addEdge(u, v, weight)

            elif choice == 3:

                g.displayGraph()

            elif choice == 4:

                start = int(input("Enter the source vertex: "))

                end = int(input("Enter the destination vertex: "))

                res = g.dijkstra(start, end)

                print(res)

                g.drawGraph(res)

            elif choice == 5:

                start = int(input("Enter the source vertex: "))

                end = int(input("Enter the destination vertex: "))

                res = g.shortestPath(start, end)

                print(res)

                g.drawGraph(res)

            elif choice == 6:

                start = int(input("Enter the source vertex: "))

                end = int(input("Enter the destination vertex: "))

                res = g.bellmanFord(start, end)

                print(res)

                g.drawGraph(res)

            else:

                break

    main()

def project\_minimal\_times():

    class Graph:

        def \_\_init\_\_(self, directed=False):

            self.edges = defaultdict(list)

            self.directed = directed

        def addEdge(self, u, v, weight):

            self.edges[u].append((v, weight))

            if not self.directed:

                self.edges[v].append((u, weight))

        def displayGraph(self):

            for key, value in self.edges.items():

                if self.directed:

                    print(f"{key} → {', '.join(map(str, value))}")

                else:

                    print(f"{key} 🔗 {', '.join(map(str, value))}")

        def loadFromFile(self, filename):

            with open(filename, "r") as file:

                for line in file:

                    try:

                        u, v, w = line.strip().split()

                        self.addEdge(int(u), int(v), int(w))

                    except ValueError:

                        print(f"Skipping line {line}")

        def drawGraph(self, shortest\_path=None):

            G = nx.Graph()

            for u, edges in self.edges.items():

                for v, weight in edges:

                    G.add\_edge(u, v, weight=weight)

            pos = nx.spring\_layout(G, k=0.15)

            if shortest\_path:

                node\_colors = [

                    "red"

                    if node == shortest\_path[-1]

                    else "lightgreen"

                    if node == shortest\_path[0]

                    else "lightgray"

                    for node in G.nodes()

                ]

                nx.draw\_networkx\_nodes(G, pos, node\_color=node\_colors)

                plt.text(

                    0.00,

                    1.13,

                    "Red: Last Task",

                    transform=plt.gca().transAxes,

                    fontsize=10,

                    verticalalignment="top",

                    bbox=dict(boxstyle="round", facecolor="red", alpha=0.5),

                )

                plt.text(

                    0.00,

                    1.06,

                    "Green: First Task",

                    transform=plt.gca().transAxes,

                    fontsize=10,

                    verticalalignment="top",

                    bbox=dict(boxstyle="round", facecolor="lightgreen", alpha=0.5),

                )

            else:

                nx.draw\_networkx\_nodes(G, pos, node\_color="lightblue")

            nx.draw\_networkx\_edges(G, pos)

            nx.draw\_networkx\_labels(G, pos, font\_weight="bold")

            labels = nx.get\_edge\_attributes(G, "weight")

            nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels)

            plt.show()

        def calculateMinimumTime(self):

            totalTime = 0

            for edges in self.edges.values():

                for edge in edges:

                    totalTime += edge[1]

            return totalTime

    def main():

        ERROR\_ADD\_TASKS = "\nPlease add tasks first. The graph is empty now."

        g = Graph(directed=True)

        while True:

            print("\nTASKS")

            print("1. Add tasks")

            print("2. Display graph")

            print("3. Calculate minimal time")

            print("4. Exit")

            choice = int(input(": "))

            print()

            if choice == 1:

                print("How would you like to add tasks?")

                print("1. Manually")

                print("2. Load from file")

                choice = int(input(": "))

                print()

                if choice == 1:

                    print("Enter the total number of tasks the team has to complete")

                    numberOfTasks = int(input(": "))

                    for i in range(numberOfTasks):

                        taskNumber = i + 1

                        timeToComplete = int(

                            input(

                                f"Enter time it takes to complete task {taskNumber}: "

                            )

                        )

                        g.addEdge(taskNumber, taskNumber + 1, timeToComplete) if i != (

                            numberOfTasks - 1

                        ) else g.addEdge(taskNumber, 0, timeToComplete)

                elif choice == 2:

                    filename = input("Enter the filename: ")

                    g.loadFromFile(filename)

            elif choice == 2:

                if not g.edges:

                    print(ERROR\_ADD\_TASKS)

                    continue

                print("Where to display?")

                print("1. Console")

                print("2. Graph")

                choice = int(input(": "))

                print()

                if choice == 1:

                    g.displayGraph()

                elif choice == 2:

                    g.drawGraph([1, 0])

            elif choice == 3:

                if not g.edges:

                    print(ERROR\_ADD\_TASKS)

                    continue

                res = g.calculateMinimumTime()

                print(

                    f"The minimum time it will take to complete all tasks is {res} hours."

                )

            else:

                break

    main()

def shortest\_path\_from\_a\_to\_b():

    class Graph:

        def \_\_init\_\_(self, directed=False):

            self.edges = defaultdict(list)

            self.directed = directed

        def addEdge(self, u, v, weight):

            self.edges[u].append((v, weight))

            if not self.directed:

                self.edges[v].append((u, weight))

        def displayGraph(self):

            for key, value in self.edges.items():

                if self.directed:

                    print(f"{key} → {', '.join(map(str, value))}")

                else:

                    print(f"{key} 🔗 {', '.join(map(str, value))}")

        def drawGraph(self, shortest\_path=None):

            G = nx.DiGraph() if self.directed else nx.Graph()

            for u, edges in self.edges.items():

                for v, weight in edges:

                    G.add\_edge(u, v, weight=weight)

            pos = nx.spring\_layout(G, k=0.5)

            if shortest\_path:

                node\_colors = [

                    "red"

                    if node == shortest\_path[-1]

                    else "lightgreen"

                    if node == shortest\_path[0]

                    else "lightgray"

                    for node in G.nodes()

                ]

                nx.draw\_networkx\_nodes(G, pos, node\_color=node\_colors)

                plt.text(

                    0.00,

                    1.13,

                    "Red: End",

                    transform=plt.gca().transAxes,

                    fontsize=10,

                    verticalalignment="top",

                    bbox=dict(boxstyle="round", facecolor="red", alpha=0.5),

                )

                plt.text(

                    0.00,

                    1.06,

                    "Green: Start",

                    transform=plt.gca().transAxes,

                    fontsize=10,

                    verticalalignment="top",

                    bbox=dict(boxstyle="round", facecolor="lightgreen", alpha=0.5),

                )

            else:

                nx.draw\_networkx\_nodes(G, pos, node\_color="lightblue")

            nx.draw\_networkx\_edges(

                G, pos, arrowstyle="->"

            ) if self.directed else nx.draw\_networkx\_edges(G, pos)

            nx.draw\_networkx\_labels(G, pos, font\_weight="bold")

            if shortest\_path:

                edges = [

                    (shortest\_path[i], shortest\_path[i + 1])

                    for i in range(len(shortest\_path) - 1)

                ]

                nx.draw\_networkx\_edges(

                    G, pos, edgelist=edges, edge\_color="red", arrowstyle="->", width=2

                ) if self.directed else nx.draw\_networkx\_edges(

                    G, pos, edgelist=edges, edge\_color="red", width=2

                )

            labels = nx.get\_edge\_attributes(G, "weight")

            nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels)

            plt.show()

        def loadFromFile(self, filename):

            with open(filename, "r") as file:

                for line in file:

                    try:

                        u, v, w = line.strip().split()

                        self.addEdge(int(u), int(v), int(w))

                    except ValueError:

                        pass

        def dijkstra(self, start, end):

            # Initialize a priority queue with a tuple containing the cost, the current node, and the path taken so far

            queue = [(0, start, [])]

            # Create an empty set to keep track of visited nodes

            seen = set()

            # Continue looping while there are nodes in the queue

            while queue:

                # Pop the node with the lowest cost from the queue

                (cost, node, path) = heappop(queue)

                # Check if the node has been visited before

                if node not in seen:

                    # Mark the node as visited

                    seen.add(node)

                    # Append the current node to the path taken so far

                    path = path + [node]

                    # Check if the current node is the destination node

                    if node == end:

                        # Return the path taken

                        return path

                    # Iterate over the edges of the current node

                    for nextNode, c in self.edges[node]:

                        # Check if the next node has not been visited

                        if nextNode not in seen:

                            # Calculate the new cost by adding the cost of the current node and the cost of the edge to the next node

                            newCost = cost + c

                            # Add the next node to the queue with the updated cost and path

                            heappush(queue, (newCost, nextNode, path))

            # If the destination node cannot be reached, return an empty path

            return []

    def main():

        g = Graph(directed=True)

        while True:

            print("\nROUTE")

            print("1. Add Edge")

            print("2. Display Graph")

            print("3. Find Shortest Path")

            print("4. Exit")

            choice = int(input("Enter your choice: "))

            print()

            if choice == 1:

                print("How to add an edge?")

                print("1. Manually")

                print("2. From File")

                choice = int(input("Enter your choice: "))

                print()

                if choice == 1:

                    print(

                        "Enter the starting node, ending node and weight. Separate them with a space"

                    )

                    u, v, weight = map(

                        int,

                        input(": ").split(),

                    )

                    isOneWay = input("Is this a one-way road? (y/n): ") == "y"

                    doesHaveTraffic = (

                        input("Does this road have a traffic jam? (y/n): ") == "y"

                    )

                    weight += 5 if doesHaveTraffic else 0

                    if isOneWay:

                        g.addEdge(u, v, weight)

                    else:

                        g.addEdge(u, v, weight)

                        g.addEdge(v, u, weight)

                elif choice == 2:

                    filename = input("Enter the filename: ")

                    g.loadFromFile(filename)

            elif choice == 2:

                print("How to display?")

                print("1. Console")

                print("2. Graph")

                choice = int(input("Enter your choice: "))

                print()

                if choice == 1:

                    g.displayGraph()

                elif choice == 2:

                    g.drawGraph()

            elif choice == 3:

                start = int(input("Enter the starting node: "))

                end = int(input("Enter the destination vertex: "))

                res = g.dijkstra(start, end)

                print(res)

                g.drawGraph(res)

            else:

                break

    main()

def shortest\_path\_from\_all\_points():

    class Graph:

        def \_\_init\_\_(self, directed=False):

            self.edges = defaultdict(list)

            self.directed = directed

        def addEdge(self, u, v, weight):

            self.edges[u].append((v, weight))

            if not self.directed:

                self.edges[v].append((u, weight))

        def displayGraph(self):

            for key, value in self.edges.items():

                if self.directed:

                    print(f"{key} → {', '.join(map(str, value))}")

                else:

                    print(f"{key} 🔗 {', '.join(map(str, value))}")

        def drawGraph(self, shortest\_path=None):

            G = nx.DiGraph() if self.directed else nx.Graph()

            for u, edges in self.edges.items():

                for v, weight in edges:

                    G.add\_edge(u, v, weight=weight)

            pos = nx.spring\_layout(G, k=0.5)

            if shortest\_path:

                node\_colors = [

                    "red"

                    if node == shortest\_path[-1]

                    else "lightgreen"

                    if node == shortest\_path[0]

                    else "lightgray"

                    for node in G.nodes()

                ]

                nx.draw\_networkx\_nodes(G, pos, node\_color=node\_colors)

                plt.text(

                    0.00,

                    1.13,

                    "Red: End",

                    transform=plt.gca().transAxes,

                    fontsize=10,

                    verticalalignment="top",

                    bbox=dict(boxstyle="round", facecolor="red", alpha=0.5),

                )

                plt.text(

                    0.00,

                    1.06,

                    "Green: Start",

                    transform=plt.gca().transAxes,

                    fontsize=10,

                    verticalalignment="top",

                    bbox=dict(boxstyle="round", facecolor="lightgreen", alpha=0.5),

                )

            else:

                nx.draw\_networkx\_nodes(G, pos, node\_color="lightblue")

            nx.draw\_networkx\_edges(

                G, pos, arrowstyle="->"

            ) if self.directed else nx.draw\_networkx\_edges(G, pos)

            nx.draw\_networkx\_labels(G, pos, font\_weight="bold")

            if shortest\_path:

                edges = [

                    (shortest\_path[i], shortest\_path[i + 1])

                    for i in range(len(shortest\_path) - 1)

                ]

                nx.draw\_networkx\_edges(

                    G, pos, edgelist=edges, edge\_color="red", arrowstyle="->", width=2

                ) if self.directed else nx.draw\_networkx\_edges(

                    G, pos, edgelist=edges, edge\_color="red", width=2

                )

            labels = nx.get\_edge\_attributes(G, "weight")

            nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels)

            plt.show()

        def loadFromFile(self, filename):

            with open(filename, "r") as file:

                for line in file:

                    try:

                        u, v, w = line.strip().split()

                        self.addEdge(int(u), int(v), int(w))

                    except ValueError:

                        pass

        def floydWarshall(self):

            # Initialize a defaultdict to store the shortest distances between nodes

            dist = defaultdict(lambda: defaultdict(lambda: float("inf")))

            # Initialize a defaultdict to store the next node in the shortest path

            nextNode = defaultdict(dict)

            # Iterate over each node in the graph

            for u in self.edges:

                # Set the distance from a node to itself as 0

                dist[u][u] = 0

                # Iterate over each neighbor of the current node

                for v, weight in self.edges[u]:

                    # Set the distance to the neighbor node

                    dist[u][v] = weight

                    # Set the next node in the shortest path to be the neighbor node

                    nextNode[u][v] = v

            # Iterate over each intermediate node

            for k in self.edges:

                # Iterate over each source node

                for i in self.edges:

                    # Iterate over each destination node

                    for j in self.edges:

                        # If the distance from i to k plus the distance from k to j is smaller than the current distance from i to j

                        if dist[i][k] + dist[k][j] < dist[i][j]:

                            # Update the distance from i to j

                            dist[i][j] = dist[i][k] + dist[k][j]

                            # Update the next node in the shortest path from i to j

                            nextNode[i][j] = nextNode[i][k]

            # Return the shortest distances and next nodes

            return dist, nextNode

        def shortestPath(self, start, end):

            # Run the Floyd-Warshall algorithm to get the nextNode matrix

            distances, nextNode = self.floydWarshall()

            # If there is no path from start to end, return None

            if nextNode[start][end] is None:

                return None

            # Create a list to store the path

            path = [start]

            # Traverse the nextNode matrix to find the shortest path from start to end

            while start != end:

                start = nextNode[start][

                    end

                ]  # Update the current node to the next node in the path

                path.append(start)  # Add the current node to the path

            # Return the shortest path

            return path

    def main():

        g = Graph(directed=False)

        while True:

            print("\nALL THE ROUTES")

            print("1. Add Edge")

            print("2. Display Graph")

            print("3. Display Shortest Paths")

            print("4. Exit")

            choice = int(input("Enter your choice: "))

            print()

            if choice == 1:

                print("How to add an edge?")

                print("1. Manually")

                print("2. From File")

                choice = int(input("Enter your choice: "))

                print()

                if choice == 1:

                    print(

                        "Enter the starting node, ending node and weight. Separate them with a space"

                    )

                    u, v, weight = map(

                        int,

                        input(": ").split(),

                    )

                    isOneWay = input("Is this a one-way road? (y/n): ") == "y"

                    doesHaveTraffic = (

                        input("Does this road have a traffic jam? (y/n): ") == "y"

                    )

                    weight += 5 if doesHaveTraffic else 0

                    if isOneWay:

                        g.addEdge(u, v, weight)

                    else:

                        g.addEdge(u, v, weight)

                        g.addEdge(v, u, weight)

                elif choice == 2:

                    filename = input("Enter the filename: ")

                    g.loadFromFile(filename)

            elif choice == 2:

                print("How to display?")

                print("1. Console")

                print("2. Graph")

                choice = int(input("Enter your choice: "))

                print()

                if choice == 1:

                    g.displayGraph()

                elif choice == 2:

                    g.drawGraph()

            elif choice == 3:

                start = int(input("Enter the starting node: "))

                for currentNode in g.edges:

                    if start != currentNode:

                        res = g.shortestPath(start, currentNode)

                        print(f"{start} -> {currentNode}: {res}")

                        g.drawGraph(res)

            else:

                break

    main()

def menu():

    while True:

        print("\nMake your choice")

        print("1. See Dijkstra, Floyd-Warshall or Bellman-Ford")

        print("2. Project minimal times")

        print("3. Shortest path from A to B")

        print("4. Shortest path from all points")

        print("5. Exit")

        choice = int(input(": "))

        if choice == 1:

            algorithms()

        elif choice == 2:

            project\_minimal\_times()

        elif choice == 3:

            shortest\_path\_from\_a\_to\_b()

        elif choice == 4:

            shortest\_path\_from\_all\_points()

        else:

            break

if \_\_name\_\_ == "\_\_main\_\_":

    menu()

// roads.txt

1 2 10

2 3 10

3 4 20

4 5 20

2 1 10

3 2 10

4 3 20

5 4 20

4 2 10

2 4 10

3 11 10

11 2 10

11 3 10

2 11 10

6 7 20

7 8 20

8 9 20

9 10 20

7 6 20

8 7 20

9 8 20

10 9 20

7 9 10

9 7 10

3 8 5

8 3 5

6 10 10

1 5 10

// simple.txt

0 1 3

0 3 7

0 4 8

4 3 3

1 3 4

1 2 1

3 2 2

// tasks.txt

1 2 3

2 3 2

2 5 3

3 4 2

3 0 1

**Висновки**

Таким чином, ми вивчили основні алгоритми пошуку найкоротших шляхів в графах: алгоритми Дейкстри, Флойда-Воршелла та Беллмана-Форда, а також навчилися застосовувати алгоритми пошуку найкоротшого шляху в графі для розв'язання практичних задач.

**Контрольні питання**

**Для роботи з якими видами графів призначені алгоритми пошуку найкоротших шляхів?**

Алгоритми найкоротшого шляху призначені для роботи зі зваженими графами. У цих графах кожному ребру присвоюється вага або вартість, що відображає вартість проходження цього ребра. Вагою може бути що завгодно, залежно від контексту, наприклад, відстань між двома містами, час, необхідний для подорожі між двома точками, або вартість певної операції в комп'ютерній програмі. Мета алгоритму - знайти шлях з найменшою сумарною вагою від початкового вузла до кінцевого.

**За яких умов може використовуватися алгоритм Дейкстри?**

Алгоритм Дейкстри можна використовувати, коли граф зв'язний, тобто між кожною парою вершин існує шлях, а всі ваги ребер невід'ємні. Це пов'язано з тим, що алгоритм припускає, що найкоротший шлях до вершини є сумою ваг ребер на цьому шляху, і ця сума завжди невід'ємна. Якщо є від'ємні ваги, алгоритм може працювати некоректно, оскільки він може вважати шляхи з від'ємними вагами найкоротшими.

**У чому полягає алгоритм Дейкстри?**

Алгоритм Дейкстри - це популярний алгоритм, який використовується для пошуку найкоротшого шляху в графі від початкової вершини до всіх інших вершин. Він працює шляхом збереження набору невідвіданих вершин і безперервного вибору вершини з найменшою орієнтовною відстанню від початку, а потім оновлення відстаней до її сусідніх вершин. Цей процес повторюється до тих пір, поки не будуть відвідані всі вершини. Алгоритм було названо на честь його винахідника, Едсгера В. Дейкстри, який представив його у 1956 році.