МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

КАТЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Дискретна математика

Лабораторна робота №3

«Графи. Способи представлення графів. Остовні дерева. Пошук найкоротших шляхів»

Виконав:
студент групи ІО-32
Крадожон М. Р.
Номер у списку групи: 16
Перевірив:
Пономаренко А. М.

Київ – 2023

Лабораторна робота №3

Тема: «Графи. Способи представлення графів. Остовні дерева. Пошук найкоротших шляхів».

Мета: вивчення властивостей графів, способів їх представлення та основних алгоритмів на графах.

Загальне завдання: Створити програму, яка реалізує один з алгоритмів на графах.

Теоретичні основи:

Останнім часом теорія графів стала простим, доступним і потужним засобом вирішення завдань, що відносяться до широкого кола проблем. Це проблеми проектування інтегральних схем і схем управління, дослідження автоматів, логічних ланцюгів, блок-схем програм, економіки та статистики, хімії та біології, теорії розкладів і дискретної оптимізації.

Граф *G* задають множиною точок або вершин $x_1, x_2, ..., x_n$ (яку позначають через *X*) і множиною ліній або ребер $a_1, a_2, ...$, (яку позначають символом *A*), що з'єднують між собою всі або частину цих точок. Таким чином, граф *G* повністю задають (і позначають) парою (*X, A*).

Якщо ребра з множини *A* орієнтовані, що зазвичай показується стрілкою, то їх називають дугами, і граф з такими ребрами називають орієнтованим графом (рис. 3.1 (а)). Якщо ребра не мають орієнтації, то граф називають неорієнтованим (рис. 3.1 (б)).

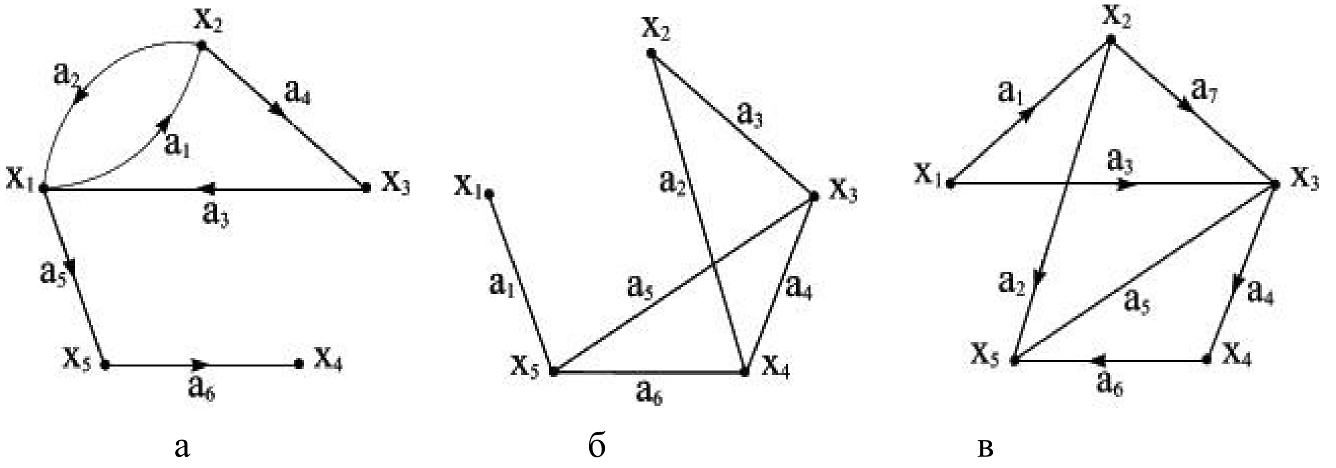


Рис. 3.1 (а) – орієнтований граф; (б) – неорієнтований граф; (в) – змішаний граф

Якщо дугу позначають впорядкованою парою, що складається з початкової та кінцевої вершин (тобто двома кінцевими вершинами дуги), її напрямок передбачається заданим від першої вершини до другої. Так, наприклад, на рис. 3.1 (а) позначення (x_1, x_2) відноситься до дуги a_1, а (x_2, x_1) – до дуги a_2.
Інший, вживаний частіше, опис орієнтованого графа G полягає у задаванні множини вершин X і *відповідності* Г, яка показує, як між собою пов'язані вершини. Відповідність Г називають відображенням множини X в X, а граф в цьому випадку позначають парою $G = (X, \Gamma)$.
Для графа на рис. 4.1 (а) маємо $\Gamma(x_1) = \{x_2, x_5\}$, тобто вершини x_2 і x_5 є кінцевими вершинами дуг, у яких початковою вершиною є x_1.

$\Gamma(x_2) = \{x_1, x_3\}$, $\Gamma(x_3) = \{x_1\}$, $\Gamma(x_4) =$ - порожня множина, $\Gamma(x_5) = \{x_4\}$

У разі неорієнтованого графа або графа, що містить і дуги, і неорієнтовані ребра (див., наприклад, графи, зображені на рис. 3.1 (б) і рис. 3.1 (в)), передбачається, що відповідність Г задає такий еквівалентний орієнтований граф, який отримуємо з початкового графа заміною кожного неорієнтованого ребра двома протилежно спрямованими дугами, що з'єднують ті ж самі вершини. Так, наприклад, для графа, наведеного на рис. 3.1 (б), маємо $\Gamma(x_5) = \{x_1, x_3, x_4\}$, $\Gamma(x_1) = \{x_5\}$ і ін.
Оскільки пряма відповідність або образ вершини $\Gamma(x_i)$ є множиною таких x_j X, для яких в графі G існує дуга (x_i,x_j), то через $\Gamma^{-1}(x_i)$ природно позначити множину вершин x_k, для яких в G існує дуга (x_k,x_j). Таку відповідність прийнято називати *зворотною відповідністю* або *прообразом* вершини. Для графа, зображеного на рис. 3.1(а), маємо

$\Gamma^{-1}(x_1)=\{x_2,x_3\}$, $\Gamma^{-1}(x_2)=\{x_1\}$ і т. д.

Цілком очевидно, що для неорієнтованого графа $\Gamma^{-1}(x_i)=\Gamma(x_i)$ для всіх x_i X.
Коли відображення Г діє не на одну вершину, а на множину вершин $X_q=\{x_1,x_2,...,x_q\}$, то під $\Gamma(X_q)$ розуміють об'єднання $\Gamma(x_1)$ $\Gamma(x_2)$ ... $\Gamma(x_q)$, тобто $\Gamma(X_q)$ є множиною таких вершин x_j X, що для кожної з них існує дуга (x_i,x_j) в G, де x_i X_q. Для графа, наведеного на рис. 3.1(а),

$\Gamma(\{x_2,x_5\})=\{x_1,x_3,x_4\}$ і $\Gamma(\{x_1,x_3\})=\{x_2,x_5,x_1\}$.

Відображення $\Gamma(\Gamma(x_i))$ записують як $\Gamma^2(x_i)$. Аналогічно "потрійне" відображення $\Gamma(\Gamma(\Gamma(x_i)))$ записують як $\Gamma^3(x_i)$ і т. д. Для графа, показаного на рис. 3.1(а), маємо:

$\Gamma^2(x_1)=\Gamma(\Gamma(x_1))=\Gamma(\{x_2,x_5\})=\{x_1,x_3,x_4\}$;

$\Gamma^3(x_1)=\Gamma(\Gamma^2(x_1))=\Gamma(\{x_1,x_3,x_4\})=\{x_2,x_5,x_1\}$ і т. д

Аналогічно потрібно розуміти позначення $\Gamma^{-2}(x_i)$, $\Gamma^{-3}(x_i)$ і т. д.

Дуги (x_i,x_j), (x_i, x_j), що мають спільні кінцеві вершини, називають суміжними. Дві вершини x_i і x_j називають суміжними, якщо яка-небудь з двох дуг (x_i,x_j) і (x_j,x_i) або обидві одночасно присутні в графі. Так, наприклад, на рис. 3.2 дуги a_1, a_{10}, a_3 і a_6, як і вершини x_5 і x_3, є суміжними, у той час як дуги a_1 і a_5 або вершини x_1 і x_4 не є суміжними. Число дуг, які мають вершину x_i своєю початковою вершиною, називають **напівстепенем виходу** вершини x_i, і, аналогічно, число дуг, які мають x_i своєю кінцевою вершиною, називають напівстепенем входу вершини x_i.

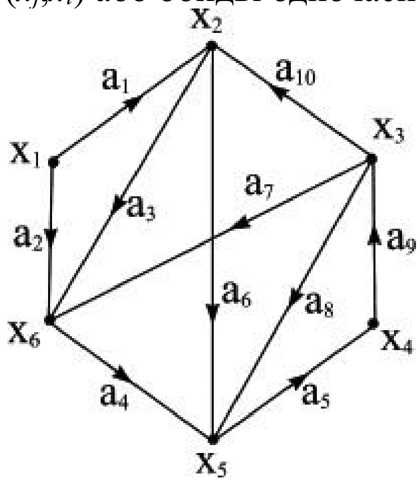


Рис. 3.2.

Таким чином, на рис. 3.2 напівстепінь виходу вершини x_3, позначена через $deg^+(x_3)$, дорівнює $|\Gamma(x_3)|=3$, і напівстепінь входу вершини x_3, позначена через $deg^-(x_3)$, дорівнює $|\Gamma^{-1}(x_3)|=1$.

Очевидно, що сума напівстепенів входу всіх вершин графа, а також сума напівстепенів виходу всіх вершин дорівнюють загальному числу дуг графа G, тобто

$$\sum_{i=1}^{n} \deg^{+}(x_i) = \sum_{i=1}^{n} \deg^{-}(x_i) = m \qquad (3.1)$$

де n – число вершин і m – число дуг графа G. Для неорієнтованого графа $G=(X, Г)$ степінь вершини x_i визначають аналогічно – за допомогою співвідношення $deg(x_i) = |\Gamma(x_i)|=|\Gamma^{-1}(x_i)|$.

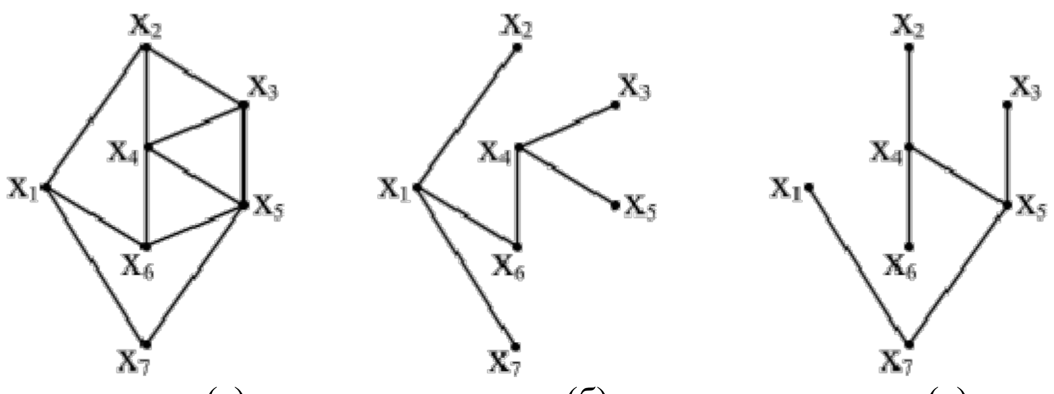
Петлею (x_i,x_i) називають дугу, початкова та кінцева вершини якої збігаються.

Одним з найбільш важливих понять теорії графів є дерево. *Неорієнтованим деревом* називають зв'язний граф, що не має циклів.

Суграфом графа G є підграф G_p, що містить всі вершини початкового графа. Якщо $G=(X,A)$ – неорієнтований граф з n вершинами, то зв'язний суграф G_p, який не має циклів, називають *остовним деревом (остовом) графа G*.

Для остовного дерева справедливе співвідношення:

$$G_p=(X_p,A_p) \subseteq G, \quad \text{де } X_p = X,\ A_p \subseteq A \qquad (3.2)$$



(а) (б) (в)

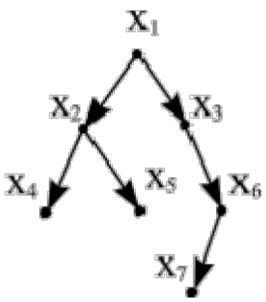
Рис. 3.3. Представлення графа у вигляді остовних дерев

Легко довести, що остовне дерево має наступні властивості:
1) Остовне дерево графа з n вершинами має n-1 ребро (|X_p|=| $_p$|-1); 2) Існує єдиний шлях, що з'єднує будь-які дві вершини остова графа:
x_i,x_j X_p (i j) ! (x_i,x_j).

Наприклад, якщо G – граф, показаний на рис. 3.3(а), то графи на рис. 3.3(б, в) є остовами графа G. Зі сформульованих вище визначень випливає, що остов графа G можна розглядати як мінімальний зв'язний остовний підграф графа G.

Поняття дерева як математичного об'єкта було вперше запропоновано Кірхгофом у зв'язку з визначенням фундаментальних циклів, застосовуваних при аналізі електричних ланцюгів. Приблизно десятьма роками пізніше Келі знову (незалежно від Кірхгофа) ввів поняття дерева і отримав більшу частину перших результатів у галузі дослідження властивостей дерев. Велику популярність здобула його знаменита теорема:

Теорема Келі. На графі з n вершинами можна побудувати n^{n-2} остови дерев.



Орієнтоване дерево є орієнтованим графом без циклів, в якому напівстепінь входу кожної вершини, за винятком однієї (вершини r), дорівнює одиниці, а напівстепінь входу вершини r (названої коренем цього дерева) дорівнює нулю.
На рис. 3.4 показаний граф, який є орієнтованим деревом з коренем у вершині x_1. З наведеного визначення випливає, що орієнтоване дерево з n вершинами має n-1 дуг і є зв'язним.

Рис. 3.4.

Неорієнтоване дерево можна перетворити в орієнтоване: треба взяти його довільну вершину як корінь і ребрам приписати таку орієнтацію, щоб кожна вершина з'єднувалася з коренем простим ланцюгом.

"Генеалогічне дерево", в якому вершини відповідають особам чоловічої статі, а дуги орієнтовані від батьків до дітей, є добре відомим прикладом орієнтованого дерева. Корінь в цьому дереві відповідає "засновнику" роду (особі, народженій раніше за інших).
Шляхом (або *орієнтованим маршрутом*) орієнтованого графа називають послідовність дуг, в якій кінцева вершина будь-якої дуги, відмінної від останньої, є початковою вершиною наступної. Так, на рис. 4.5 послідовності дуг $\mu_1=\{a_6, a_5, a_9, a_8, a_4\}$, $_2=\{a_1, a_6, a_5, a_9\}$, $\mu_3=\{a_1, a_6, a_5, a_9, a_{10}, a_6, a_4\}$ є шляхами.
Орієнтованим ланцюгом називають такий шлях, в якому кожна дуга використовується не більше одного разу. Так, наприклад, наведені вище шляхи μ_1 і μ_2 є орієнтованими ланцюгами, а шлях μ_3 не є таким, оскільки дуга $_6$ в ньому використовується двічі. a
Маршрут є неорієнтованим "двійником" шляху, і це поняття розглядається в тих випадках, коли можна знехтувати спрямованістю дуг у графі.

Таким чином, маршрут є послідовністю ребер $a_1, a_2,...,a_q$, в якій кожне ребро a_i, за винятком, можливо, першого і останнього ребра, пов'язане з ребрами a_{i-1} і a_{i+1} своїми двома кінцевими вершинами.
Послідовності дуг на рис. 3.6 $\mu_4=\{a_2, a_4, a_8, a_{10}\}$, $\mu_5=\{a_2, a_7, a_8, a_4, a_3\}$ і $\mu_6=\{a_{10}, a_7, a_4, a_8, a_7, a_2\}$ є маршрутами. а
Контуром (*простим ланцюгом*) називають такий шлях (маршрут), в якому кожна вершина використовується не більше одного разу. Наприклад, шлях μ_2 є контуром, а шляхи μ_1 і μ_3 – ні. Очевидно, що контур є також ланцюгом, але зворотне твердження невірне. Наприклад, шлях μ_1 є ланцюгом, але не контуром, шлях μ_2 є ланцюгом і контуром, а шлях μ_3 не є ні ланцюгом, ні контуром.

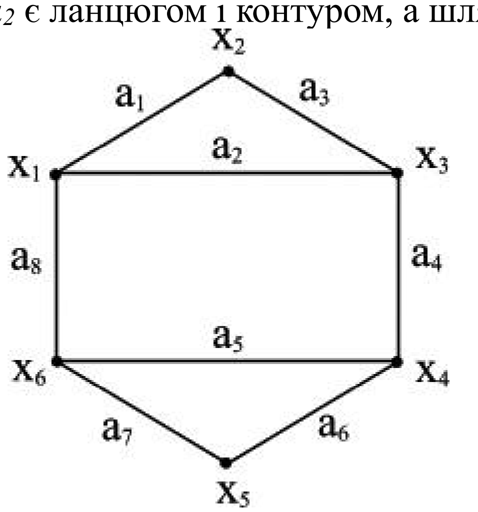


Рис. 3.5

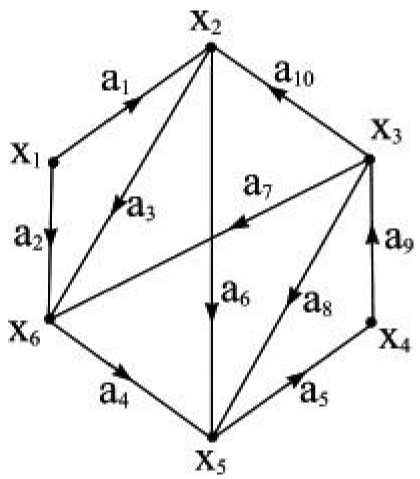


Рис. 3.6

Аналогічно визначають простий ланцюг у неорієнтованих графах. Так, наприклад, маршрут μ_4 є простим ланцюгом, маршрут μ_5 – ланцюг, а маршрут μ_6 не є ланцюгом.

Шлях або маршрут можна зображати також послідовністю вершин. Наприклад, шлях μ_1 можна представити так: $\mu_1=\{x_2, x_5, x_4, x_3, x_5, x_6\}$ і таке представлення часто виявляється більш корисним у тих випадках, коли здійснюється пошук контурів або простих ланцюгів.
Іноді дугам графа G співставляють (приписують) числа – дузі (x_i,x_j) ставлять у відповідність деяке число c_{ij}, назване **вагою**, або довжиною, або **вартістю** (ціною) дуги. Тоді граф G називають зваженим. Іноді ваги (числа v_i) приписують вершинам x_i графа.
При розгляді шляху μ, представленого послідовністю дуг $(a_1, a_2,...,a_q)$, за його вагу (або довжину, або вартість) приймають число $L(\mu)$, яке дорівнює сумі ваг всіх дуг, що входять до μ, тобто

$$L(\mu)=\sum_{(x_i,x_j)\in\mu} c_{ij} \qquad (3.3)$$

Таким чином, коли слова "довжина", "вартість", "ціна" і "вага" застосовують до дуг, то вони еквівалентні за змістом, і в кожному конкретному випадку вибирають таке слово, яке краще відповідає змісту завдання. **Довжиною** (або потужністю) шляху називають кількість (число) дуг, що входять до нього.

Індивідуальне завдання: **Варіант 7:**

7	1. Вивчити принципи застосування алгоритму топологічного сортування до пошуку найкоротших шляхів у графі. Написати програму пошуку найкоротших шляхів у графі з застосуванням алгоритму топологічного сортування. 2. За правилом, наданим викладачем, сформувати матрицю ваг $C = \left[c_{i,j}\right]$ графа. 3. Представити граф, заданий матрицею ваг C, у графічній формі та виділити найкоротший шлях між заданими викладачем вершинами, сформований за алгоритмом топологічного сортування.

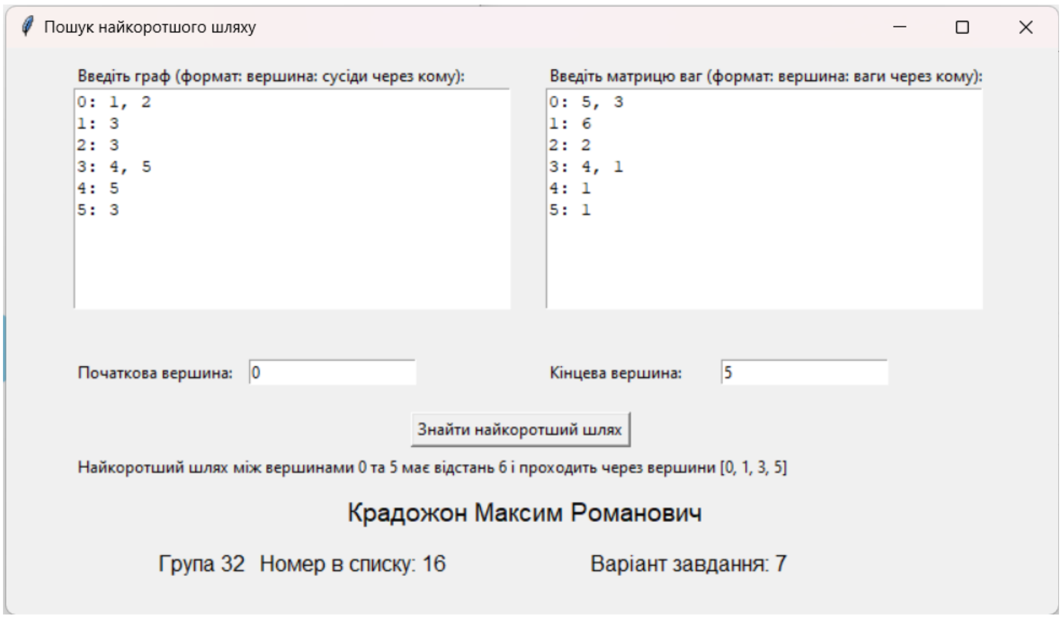
Роздруківка коду:

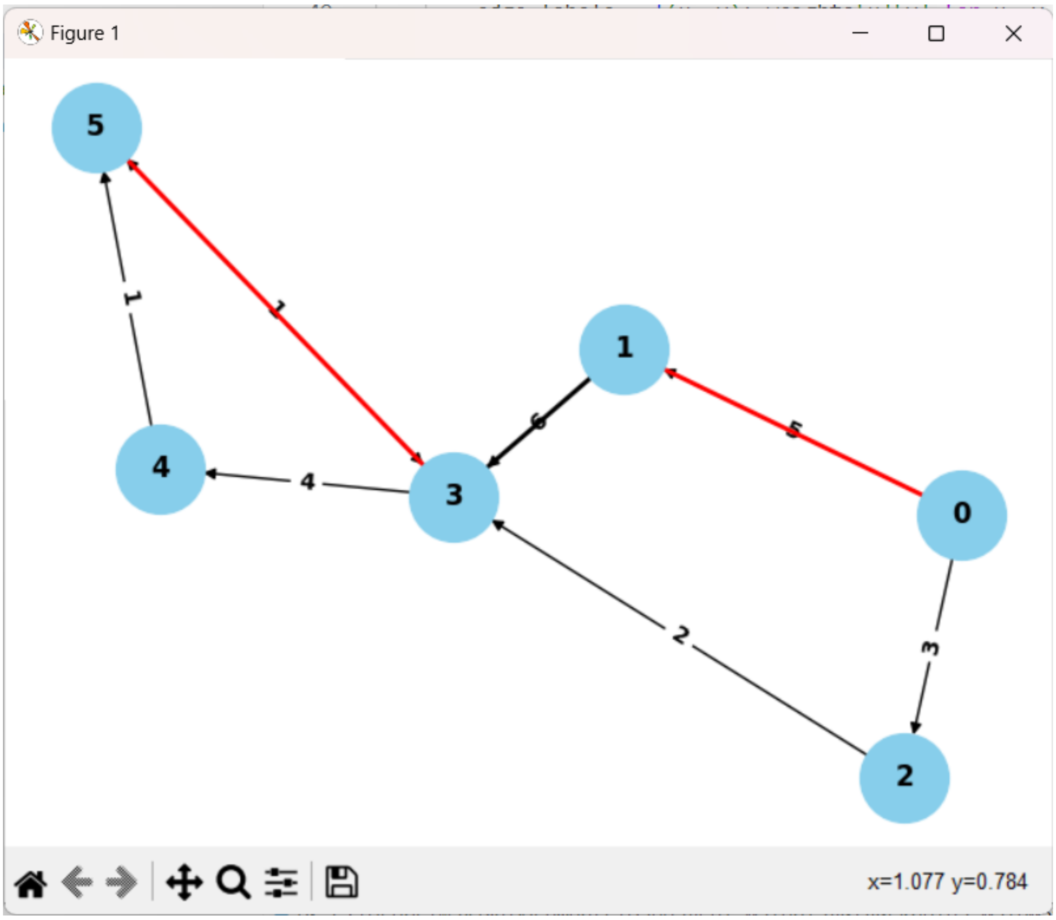
```
from tkinter import *
from sencitive_data import *

import networkx as nx
import matplotlib.pyplot as plt


class GraphSolver:
    def __init__(self):
        pass

    def topological_sort(self, graph):
        visited, sorted_vertices = set(), []
        def visit(vertex):
            if vertex not in visited:
                visited.add(vertex)
                for neighbor in graph[vertex]:
                    visit(neighbor)
                sorted_vertices.append(vertex)
        for vertex in graph:
            visit(vertex)
        return sorted_vertices[::-1]

    def shortest_path(self, graph, weights, start, end):
        sorted_vertices = self.topological_sort(graph)
        distances = {vertex: float('inf') for vertex in graph}
        distances[start] = 0
        for vertex in sorted_vertices:
            for neighbor in graph[vertex]:
                new_distance = distances[vertex] + weights[vertex][neighbor]
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance
        path = []
        current_vertex = end
        while current_vertex != start:
            path.append(current_vertex)
            current_vertex = min((distances[current_vertex] - weights[v][current_vertex], v) for v in graph if
current_vertex in graph[v])[1]
        path.append(start)
        return distances[end], path[::-1]
```

```
    def visualize_graph(self, graph, weights, shortest_path_edges):
        G = nx.DiGraph()
        G.add_nodes_from(graph)
        for node, neighbors in graph.items():
            for neighbor in neighbors:
                edge_weight = weights[node][neighbor]
                G.add_edge(node, neighbor, weight=edge_weight)
        pos = nx.spring_layout(G)
        edge_labels = {(u, v): weights[u][v] for u, v in G.edges()}
        nx.draw(G, pos, with_labels=True, node_size=1500, node_color='skyblue', font_weight='bold')
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_weight='bold')
        red_edges = [(shortest_path_edges[i], shortest_path_edges[i + 1]) for i in range(len(shortest_path_edges) -
1)]
        edge_colors = ["red" if edge in red_edges else "black" for edge in G.edges()]
        nx.draw_networkx_edges(G, pos, edgelist=red_edges, edge_color=edge_colors, width=2.0)
        plt.show()

    def parse_input(self, graph_input, weights_input, start_vertex_input, end_vertex_input):
        graph = {}
        weights = {}
        for line in graph_input.split("\n"):
            if line.strip():
                vertex, neighbors = line.split(":")
                vertex = int(vertex.strip()) if vertex.strip() else None
                neighbors = list(map(int, neighbors.split(",")))
                if vertex is not None:
                    graph[vertex] = neighbors
        for line in weights_input.split("\n"):
            if line.strip():
                vertex, weight_list = line.split(":")
                vertex = int(vertex.strip())
                weight_list = weight_list.split(",")
                weights[vertex] = {graph[vertex][i]: int(weight_list[i].strip()) for i in range(len(graph[vertex]))}
        start_vertex = int(start_vertex_input)
        end_vertex = int(end_vertex_input)
        return graph, weights, start_vertex, end_vertex


class GraphSolverGUI:
    def __init__(self, root):
        self.root = root
        self.solver = GraphSolver()

        self.graph_label = self.create_and_place_widget(Label, 50, 10, text="Введіть граф (формат: вершина: сусіди
через кому):")
        self.graph_text = self.create_and_place_widget(Text, 50, 30, height=10, width=40)

        self.weights_label = self.create_and_place_widget(Label, 400, 10, text="Введіть матрицю ваг (формат:
вершина: ваги через кому):")
        self.weights_text = self.create_and_place_widget(Text, 400, 30, height=10, width=40)
```

```
        self.start_vertex_label = self.create_and_place_widget(Label, 50, 230, text="Початкова вершина:")
        self.start_vertex_entry = self.create_and_place_widget(Entry, 180, 231)

        self.end_vertex_label = self.create_and_place_widget(Label, 400, 230, text="Кінцева вершина:")
        self.end_vertex_entry = self.create_and_place_widget(Entry, 530, 231)

        self.submit_button = self.create_and_place_widget(Button, 300, 270, text="Знайти найкоротший шлях",
command=self.on_submit)
        self.result_label = self.create_and_place_widget(Label, 50, 300, text="")

    def create_and_place_widget(self, widget_type, x, y, **kwargs):
        widget = widget_type(self.root, **kwargs)
        widget.place(x=x, y=y)
        return widget

    def on_submit(self):
        graph_input = self.graph_text.get("1.0", END)
        weights_input = self.weights_text.get("1.0", END)
        start_vertex_input = self.start_vertex_entry.get()
        end_vertex_input = self.end_vertex_entry.get()

        graph, weights, start_vertex, end_vertex = self.solver.parse_input(graph_input, weights_input,
start_vertex_input, end_vertex_input)
        shortest_path_length, shortest_path_edges = self.solver.shortest_path(graph, weights, start_vertex,
end_vertex)

        self.result_label.config(text=f"Найкоротший шлях між вершинами {start_vertex} та {end_vertex} має
відстань {shortest_path_length} і проходить через вершини {shortest_path_edges}")
        self.solver.visualize_graph(graph, weights, shortest_path_edges)


root = Tk()
root.title("Пошук найкоротшого шляху")
root.geometry("780x420")

Label(root, text=senc_data['name'], font='Arial 14').place(x=250, y=310+20)
Label(root, text=f"Група {senc_data['group']}", font='Arial 12').place(x=10+100, y=350+20)
Label(root, text=f"Номер в списку: {senc_data['number_in_list']}", font='Arial 12').place(x=85+100, y=350+20)
Label(root, text=f'Варіант завдання: 7', font='Arial 12').place(x=330+100, y=350+20)

gui = GraphSolverGUI(root)
root.mainloop()
```

Скриншоти:





<u>Висновок:</u> Виконавши цю лабораторну роботу, я зміг здобути відповідні навички в графах, способи представлення графів, остовні дерева та пошук найкоротших шляхів. Під час виконання лабораторної роботи проблем не виникало, а складність була в структуруванні коду та приведенні його до більш гарного вигляду.