МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

АЛГОРИТМИ ТА СТРУКТУРИ ВПОРЯДКУВАННЯ ДАНИХ. ЧАСТИНА 1. АЛГОРИТМИ ДЛЯ РОБОТИ З ГРАФАМИ

до виконання самостійних робіт
з дисциплін «Прикладне програмування на Python»,
«Мультипарадигмальні мови програмування»
для студентів усіх форм навчання за спеціальностями
G7 «Автоматизація, комп'ютерно-інтегровані технології та робототехніка» та
F3 «Комп'ютерні науки»

Затверджено
редакційно-видавничою
радою університету,
протокол № 1 від 19.02.2026 р.

Харків
НТУ «ХПІ»
2026

Методичні вказівки «Алгоритми та структури впорядкування даних. Частина 1. Алгоритми для роботи з графами» для виконання самостійних робіт з дисциплін «Прикладне програмування на Python», «Мультипарадигмальні мови програмування» для студентів усіх форм навчання за спеціальностями G7 «Автоматизація, комп'ютерно-інтегровані технології та робототехніка» та F3 «Комп’ютерні науки» / уклад. Д. Г. Караман, А. О. Зуєв, О. А. Татарінова, Д. В. Сальніков – Харків : НТУ «ХПІ», 2026. – 77 с.

Укладачі: Д. Г. Караман
А. О. Зуєв
О. А. Татарінова
Д. В. Сальніков

Рецензент: Д. А. Гапон

Кафедра автоматизації та кібербезпеки енергосистем

ЗМІСТ

Вступ 4
1 Загальні відомості про графи 5
2 Різновиди графів та принципи їх побудови 9
3 Способи представлення графів 12
4 Пошук шляху між заданими вершинами 25
5 Пошук найкоротших та оптимальних шляхів 36
6 Ієрархічні деревоподібні структури.. 50
7 Пошук мінімального кістякового дерева.. 55
8 Методи топологічного сортування 65
Список рекомендованої літератури 76

ВСТУП

Алгоритми та структури впорядкування даних є фундаментальною складовою сучасної інформатики та програмної інженерії. Вони лежать в основі побудови ефективних програмних систем, обробки великих обсягів інформації, моделювання складних процесів та прийняття рішень у різних галузях науки і техніки. Особливе місце серед таких алгоритмів займають алгоритми роботи з графами, методи кластерного аналізу та алгоритми сортування, які забезпечують формалізацію зв'язків між об'єктами, структуризацію даних і оптимізацію обчислювальних процесів.

Теорія графів, що бере свій початок з робіт Леонарда Ейлера, на сьогодні є одним із найрозвиненіших розділів дискретної математики та алгоритмічної теорії. Графові моделі широко застосовуються при аналізі транспортних і комунікаційних мереж, соціальних структур, ієрархічних систем, баз даних, комп'ютерних мереж і задач оптимізації. Алгоритми пошуку шляхів, побудови мінімальних кістякових дерев, топологічного сортування та роботи з деревоподібними структурами дозволяють ефективно розв'язувати широкий клас прикладних задач.

Методичні вказівки призначені для студентів технічних спеціальностей, які вивчають дисципліни з алгоритмів, структур даних і програмування. У вказівках систематизовано теоретичні відомості та розглянуто основні алгоритми роботи з графами відповідно до окремих розділів дисциплін «Прикладне програмування на Python» та «Мультипарадигмальні мови програмування». Зазначені дисципліни входять до навчальних програм спеціальностей G7 «Автоматизація, комп'ютерно-інтегровані технології та робототехніка», F3 «Комп'ютерні науки», а також міждисциплінарної освітньої програми «Інтелектуальні кіберфізичні та робототехнічні системи». Матеріал може бути використаний під час виконання лабораторних і практичних робіт, а також для самостійної підготовки й поглиблення знань з алгоритмічних методів обробки даних.

1 ЗАГАЛЬНІ ВІДОМОСТІ ПРО ГРАФИ

Теорія графів як самостійний розділ математики сформувалася у XVIII столітті. Її виникнення пов'язують з іменем Леонарда Ейлера, який у 1736 році розв'язав задачу про сім кенігсберзьких мостів, запропонувавши новий абстрактний підхід до опису об'єктів та зв'язків між ними. Подальший розвиток теорії графів був продовжений у працях А. Келі, Г. Кірхгофа, В. Вітні та інших науковців, які заклали основи теорії дерев, дослідження електричних кіл і структурних властивостей графів. У XX столітті теорія графів набула інтенсивного розвитку завдяки появі обчислювальної техніки та необхідності алгоритмічного аналізу складних систем, що сприяло її широкому впровадженню в інформатиці, програмуванні та прикладних науках.

Теорія графів є фундаментальною складовою сучасної прикладної математики та алгоритмічної теорії. Завдяки своїй універсальності графи широко застосовуються для моделювання й аналізу різноманітних систем із розгалуженою структурою взаємозв'язків, зокрема комунікаційних, транспортних, інформаційних і соціальних мереж. Методи теорії графів використовуються як у прикладних дисциплінах (теорія інформації, теорія систем, проектування мереж), так і в абстрактних математичних дослідженнях.

У різних галузях знань поняття графа може набувати різних інтерпретацій і виступати у вигляді структури, мережі, соціограми, молекулярної моделі, навігаційної карти або розподільної системи. Така багатозначність пояснюється здатністю графів наочно й формалізовано описувати об'єкти та зв'язки між ними. Перевагами теорії графів як інструменту дослідження є геометрична наочність, математична змістовність і відсутність громіздкого математичного апарату. Графові моделі широко застосовуються при побудові географічних карт, гіпертекстових систем, електронних мікросхем, розкладів руху транспорту, аналізі транзакцій телекомунікаційних компаній, обчислювальних мереж і структур програм.

Граф — це сукупність об’єктів і зв’язків між ними. Об’єкти представляються у вигляді *вершин* (або вузлів), а зв’язки — у вигляді *ребер*.

Формально граф визначається як група з двох множин

$$G = (V, E),$$

де V — скінченна непорожня множина вершин, а E — множина ребер, кожне з яких з’єднує пару вершин. Одне ребро може відповідати не більше ніж одній парі вершин.

Залежно від призначення та області застосування, графи можуть мати різні властивості. Ребра графа можуть бути *орієнтованими* або *неорієнтованими* — в залежності від того, важливий або ні порядок переходу між вершинами. Також вони можуть характеризуватися *вагами*, що використовуються для задання відстаней, вартості або інших числових параметрів. Вершини й ребра можуть містити додаткові атрибути, що розширюють можливості моделювання реальних систем.

Графічно граф подається як фігура (рис. 1.1), яка складається з множини вершин у вигляді певних графічних елементів — кружків, прямокутників, овалів, і множини ребер — ліній, що з’єднують деякі пари вершин, зі стрілками або без них, в залежності від того, чи є ребра графа орієнтованими, чи ні. Таке подання забезпечує зручність візуального аналізу структури графа.

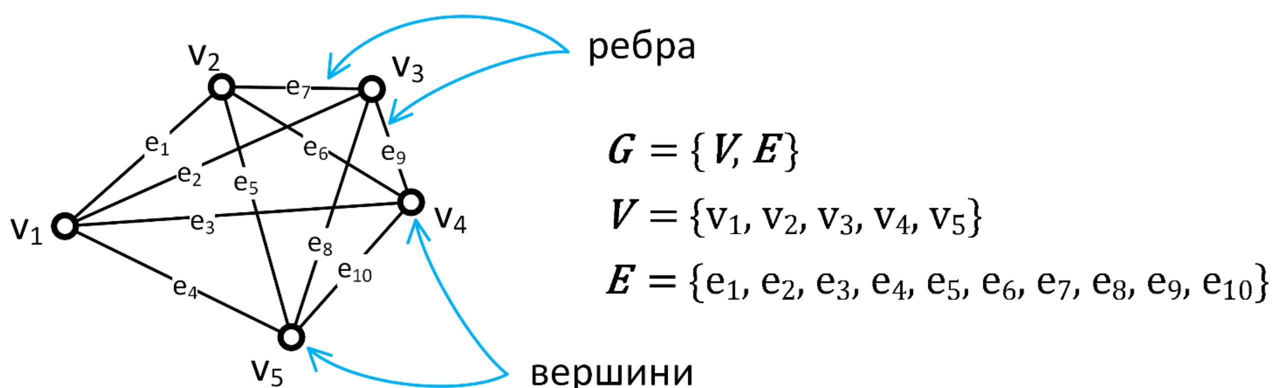

Рисунок 1.1 – Графічне представлення графа

Вершини графа називаються *суміжними*, якщо вони з’єднані одним ребром. *Ступінь* вершини визначається кількістю ребер, *інцидентних* цій вершині, тобто

таких, що виходять з неї або входять до неї (для орієнтованих графів розрізняють вхідний і вихідний ступені). Також є поняття *інцидентності вершин*. Вершина a називається інцидентною вершині b, якщо між ними існує ребро і можливий перехід з вершини a до вершини b. Таким чином, дві вершини є інцидентними, якщо вони мають спільне ребро.

Шляхом у графі називається послідовність суміжних вершин, у якій усі вершини є різними. *Цикл* — це шлях, у якого початкова та кінцева вершини співпадають. Граф, що не містить циклів, називається *ациклічним*. У випадку орієнтованих графів розрізняють також *контури*, під якими розуміють цикли з урахуванням напрямків ребер; *контуром* також може називатися цикл, що включає всі вершини графа.

Довжина шляху або циклу визначається кількістю ребер, з яких він складається, або, у випадку зваженого графа, сумою ваг цих ребер. *Непересічними* шляхами називаються такі шляхи, які не мають спільних вершин, за винятком, можливо, їх початкових або кінцевих вершин.

Контрольні запитання

1. Розв’язання якої задачі пов’язують з започаткуванням теорії графів? Який відомий вчений займався її розв’язанням?

2. Назвіть принаймні три сфери людської діяльності або типи систем, де, на вашу думку застосовуються графові моделі.

3. Дайте визначення графа. З яких основних частин він складається?

4. Зобразіть графічно довільний неорієнтований граф з п’ятьма вершинами. Поясніть, яка саме ознака визначає його неорієнтованість? Що треба зробити, щоб перетворити граф на орієнтований?

5. Покажіть на графі інцидентні вершини. Розрахуйте значення ступенів для декількох вершин.

6. Оберіть дві несуміжні вершини. Чи існує шлях між ними? Покажіть його на графі. Розрахуйте довжину цього шляху.

7. Чи містить ваш граф цикли? Покажіть їх. Якщо не містить, покажіть, як треба модифікувати граф, щоб з'явився цикл. Чи є цей цикл контуром? Зробіть модифікації у графі (зробіть копію графа з модифікаціями), щоб він містив контур.

8. Що таке зважений граф? Модифікуйте ваші орієнтований та неорієнтований графи, щоб вони стали зваженими. Чи відрізняються довжини шляхів у зваженому та незваженому графах?

2 РІЗНОВИДИ ГРАФІВ ТА ПРИНЦИПИ ЇХ ПОБУДОВИ

Графи можуть суттєво відрізнятися за своєю структурою та властивостями. Вибір конкретного виду графа визначається характером зв’язків між об’єктами, вимогами до моделі та типом алгоритмів, що над ним виконуються. Класифікація графів дозволяє впорядкувати різноманіття моделей і спростити їх подальший аналіз.

За наявністю петель і кратних ребер розрізняють *прості* графи та *мультиграфи*. Мультиграфом називається граф, у якому допускаються дублюючі ребра, а також ребра, що з’єднують вершину саму з собою (*петлі*). Якщо ж граф не містить петель і кратних ребер, такий граф називається простим. Для простого графа, що складається з V вершин, максимальна кількість ребер обмежується співвідношенням

$$E \leq \frac{V(V-1)}{2}.$$

У складі графа можуть виділятись *підграфи*. Підграфом називається граф, який утворений з деякої підмножини вершин і ребер заданого графа та сам задовольняє означенню графа. Підграфи використовуються для аналізу окремих частин складних структур і побудови допоміжних моделей.

Планарним називається граф, який може бути зображений на площині таким чином, що його ребра не перетинаються. Окремим випадком є *евклідові* графи, у яких кожній вершині відповідає точка на площині або в просторі, а відстані між вершинами мають фізичний або геометричний зміст. Прикладом евклідового графа є географічна карта, де збереження масштабу є обов’язковим.

За властивістю досяжності вершин графи поділяються на *пов’язані* та *непов’язані*. Пов’язаним називається граф, у якому для будь-якої пари вершин існує шлях, що з’єднує їх. Якщо граф не є пов’язаним, він складається з кількох компонент зв’язності (кількох ізольованих фрагментів).

Особливим класом пов’язаних графів є *дерева*. Деревом називається ациклічний пов’язаний граф. Якщо ж граф складається з кількох непов’язаних

дерев, така структура називається *лісом*. Деревоподібні структури широко застосовуються для подання ієрархій та оптимальних зв’язків між елементами.

Для пов’язаного графа важливим поняттям є *кістяк*. Кістяком називається підграф, який містить усі вершини початкового графа та є єдиним деревом. Якщо ж граф є непов’язаним, відповідною узагальненою структурою є *кістяковий ліс* — підграф, що містить усі вершини графа і є лісом. Кістякові структури відіграють важливу роль у задачах оптимізації та аналізу мереж.

За кількістю ребер графи можуть бути *повними* або *неповними*. Повним називається граф, у якому кожна пара різних вершин з’єднана ребром. *Кліка* — це повний підграф довільного графа. Кліки використовуються для виявлення щільно пов’язаних підмножин вершин у складних мережах.

Для кількісної оцінки щільності зв’язків у графі використовується *показник насиченості графа*, який визначається співвідношенням

$$a = \frac{2E}{V},$$

де E — кількість ребер, а V — кількість вершин графа. Насиченість дозволяє порівнювати графи різних розмірів за інтенсивністю зв’язків між вершинами.

Розглянуті різновиди графів створюють основу для подальшого вивчення алгоритмів пошуку шляхів, побудови кістякових дерев, топологічного сортування та інших методів роботи з графовими структурами.

Контрольні запитання

1. У чому полягає головна відмінність простого графа від мультиграфа? Що таке петля? На рисунку 2.1 зображені графи. Чи є вони мультиграфами? Чи Містять вони петлі?

2. Що таке простий граф? Яка максимальна кількість ребер може бути у простому графі, що має п’ять вершин?

3. Що таке підграф? Покажіть на графах рисунку 2.1 можливі підграфи.

4. Що таке планарний граф? Чи графи на рисунку 2.1 планарними?

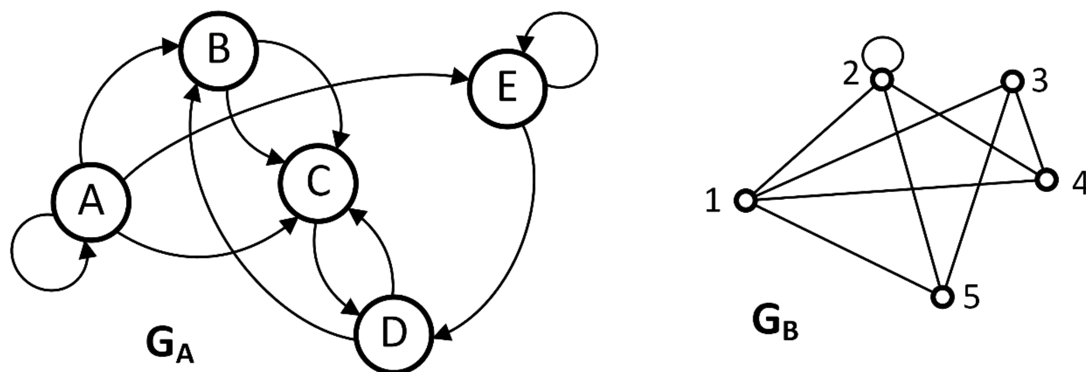

Рисунок 2.1 – Графи для завдань

5. Що таке евклідовий граф? Наведіть приклади.

6. Що таке пов’язані та непов’язані графи? До якого типу відносяться графи, зображені на рисунку 2.1? Поясніть, чому.

7. Які графи називають деревами? У якому разі граф називають лісом?

8. Чи є графи, зображені на рисунку 2.1, повними? Чи містить якийсь з цих графів кліку?

9. Розрахуйте показники насиченості для кожного з графів, зображених на рисунку 2.1. Який з них є більш насиченим?

3 СПОСОБИ ПРЕДСТАВЛЕННЯ ГРАФІВ

Графи дуже широко застосовуються у програмуванні, тому важливим є питання ефективного подання графових структур у пам’яті комп’ютера. Вибір способу представлення графа безпосередньо впливає на обсяг необхідного об’єму пам’яті, складність реалізації алгоритмів та швидкодію операцій над графом, зокрема пошуку суміжних вершин, перевірки наявності ребра та обходу графа.

У практиці програмування, зокрема при використанні мови Python, найчастіше застосовуються такі способи представлення графів:

– матриця суміжності;

– матриця інцидентності;

– список суміжності;

– список ребер.

Кожен із цих підходів має свої переваги та обмеження і використовується залежно від розміру графа, його щільності та поставленої задачі. Крім того, існує безліч варіацій реалізації цих методів, які використовують особливості мов програмування, наприклад, у Python опис графа можна реалізувати не просто як комбінацію наявних структур даних, але й у вигляді об’єктної моделі як найбільш характерної конструкції для цієї мови програмування.

Для кращого розуміння особливостей різних способів подання графів доцільно розглядати їх не лише теоретично, а й на конкретних практичних прикладах. У подальшому викладі у якості базових прикладів будуть використовуватись два графи: неорієнтований граф G_1 та орієнтований граф (орграф) G_2, зображені на рисунку 3.1.

Матриця суміжності є одним з найпростіших та найбільш наочних способів представлення графа. Вона являє собою квадратну матрицю

$$S = \left|S_{ij}\right|$$

розміром $p \times p$, де p — кількість вершин графа. У кожну комірку матриці записується значення, що визначає наявність або відсутність ребра між вершинами i та j.

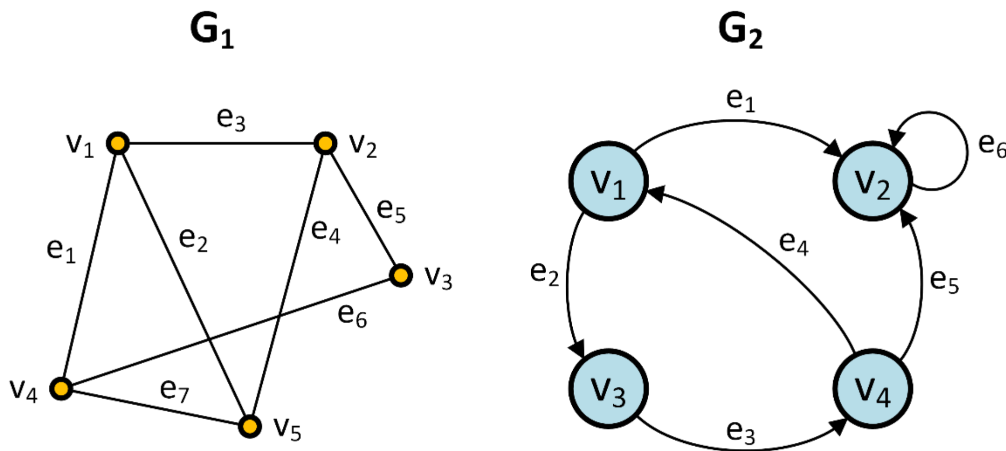

Рисунок 3.1 – Приклади графів для практичного опису

Для неорієнтованого графа елемент матриці суміжності визначається так:

$S_{ij} = 1$, якщо вершини i та j є суміжними;

$S_{ij} = 0$, якщо вершини i та j не суміжні.

У цьому випадку матриця є симетричною відносно головної діагоналі. Якщо якийсь елемент в основній діагоналі $S_{ii} = 1$, то це означає, що вершина має петлю.

Для орієнтованого графа матриця суміжності дозволяє враховувати напрямок ребер:

$S_{ij} = 1$, якщо є ребро, що переходить з вершини i у вершину j;

$S_{ij} = 0$, якщо немає ребра, що переходить з вершини i у вершину j.

Наявність одиниці на головній діагоналі також означає петлю. Ступінь вершини у неорієнтованому графі визначається як сума одиниць у відповідному рядку матриці суміжності, а для орієнтованого графа можуть окремо визначатися вхідний і вихідний ступені.

Матриці суміжності графів на рисунку 3.1 будуть виглядати наступним чином:

$$S_{\text{см}}(G_1) = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix} \qquad S_{\text{см}}(G_2) = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Матриця інцидентності є альтернативним способом подання графа, який встановлює зв’язок між вершинами та ребрами. На відміну від матриці суміжності, вона має розмір $p \times n$, де p — кількість вершин, а n — кількість ребер графа:

$$S = |S_{ij}|, \qquad i = [1..p],\ j = [1..n]$$

Для неорієнтованого графа елементи матриці інцидентності приймають такі значення:

$S_{ij} = 1$ — вершина інцидентна відповідному ребру;

$S_{ij} = 0$ — вершина не інцидентна ребру.

Для орієнтованого графа використовується розширене кодування:

$S_{ij} = 1$ — вершина є початком ребра (ребро виходить з вершини);

$S_{ij} = -1$ — вершина є кінцем ребра (ребро входить у вершину);

$S_{ij} = 0$ — вершина не інцидентна ребру.

Матриця інцидентності дозволяє зручно аналізувати структуру зв’язків між вершинами та ребрами, однак у практичному програмуванні використовується рідше через складність побудови та вищі витрати пам’яті.

Матриці інцидентності графів, зображених на рисунку 3.1, будуть виглядати наступним чином:

$$S_{\text{інц}}(G_1) = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \qquad S_{\text{інц}}(G_2) = \begin{pmatrix} 1 & 1 & 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 1 & 0 \end{pmatrix}$$

Список суміжності — це спосіб подання графа, у якому для кожної вершини зберігається перелік усіх вершин, суміжних з нею безпосередньо.

У такому поданні граф розглядається як відображення множини вершин у множину їх сусідів:

$$Adj(G): V \to 2^V$$

де V — множина вершин графа G.

Для неорієнтованого графа кожне ребро фіксується у списках обох вершин, які воно з'єднує. Для орієнтованого графа в список суміжності вершини заносяться лише ті вершини, до яких ведуть дуги з даної вершини. Тобто, таким чином, загальне визначення списку суміжності виглядає як:

$$Adj(G) = \left(Adj(v_1), Adj(v_2), \dots, Adj(v_n)\right)$$

де суміжні вершини для кожної вершини неорієнтованого графа визначаються як:

$$Adj(v) = \{u \in V \mid \{v, u\} \in E\},$$

а суміжні вершини для орієнтованого графа визначаються як:

$$Adj(v) = \{u \in V \mid (v \to u) \in E\}$$

Списки суміжності графів, зображених на рисунку 3.1, будуть виглядати наступним чином:

$$Adj(G_1) = \begin{pmatrix} v_1: (v_2, v_4, v_5), \\ v_2: (v_1, v_3, v_5), \\ v_3: (v_2, v_4), \\ v_4: (v_1, v_3, v_5), \\ v_5: (v_1, v_2, v_4) \end{pmatrix} \qquad Adj(G_2) = \begin{pmatrix} v_1: (v_2, v_3), \\ v_2: (v_2), \\ v_3: (v_4), \\ v_4: (v_1, v_2) \end{pmatrix}$$

Подання графа у вигляді списку суміжності потребує пам'яті пропорційно загальній кількості вершин та ребер, тому цей спосіб є ефективним для подання розріджених графів. Воно дозволяє швидко отримувати всі суміжні вершини, є зручним для реалізації обходів вершин графа та природно реалізується в Python за допомогою словників, списків або множин. Список суміжності є основним способом подання графів у практичному програмуванні, особливо при реалізації алгоритмів обходу та пошуку шляхів.

Список ребер — це спосіб подання графа у вигляді набору всіх його ребер без явного збереження інформації про суміжність вершин.

Кожне ребро подається у вигляді пари вершин:

– для неорієнтованого графа — (u, v);

– для орієнтованого графа — (u, v), де u — початок дуги, v — її кінець;

– для зваженого графа — (u, v, w), де w — вага ребра.

Списки ребер графів, зображених на рисунку 3.1, будуть виглядати наступним чином:

$$Edg(G_1) = (e_1, e_2, e_3, e_4, e_5, e_6, e_7) = \\ = \begin{pmatrix} (v_1, v_4), (v_1, v_5), (v_1, v_2), \\ (v_2, v_5), (v_2, v_3), (v_3, v_4), \\ (v_4, v_5) \end{pmatrix}$$

$$Edg(G_2) = (e_1, e_2, e_3, e_4, e_5, e_6) = \\ = \begin{pmatrix} (v_1, v_2), (v_1, v_3), \\ (v_3, v_4), (v_4, v_1), \\ (v_4, v_2), (v_2, v_2) \end{pmatrix}$$

Список ребер використовується у задачах, де ключову роль відіграють саме ребра графа, а також коли необхідні найбільш компактне збереження та передача структури графа.

У мові Python немає вбудованого типу даних для безпосереднього подання графів, тому вони задаються за допомогою комбінації стандартних структур даних. Нижче наведені найбільш притаманні для Python способи подання графів із прикладами, які реально використовуються на практиці.

Матриця суміжності графа найбільш природньо задається через вкладені списки — найбільш поширений спосіб задавання квадратних матриць з однорідними елементами.

Реалізації матриць суміжності у мові Python для графів, зображених на рисунку 3.1 наведені нижче.

```
# Неорієнтований граф G1 (матриця суміжності)
G1 = [
    [0, 1, 0, 1, 1],
    [1, 0, 1, 0, 1],
    [0, 1, 0, 1, 0],
    [1, 0, 1, 0, 1],
    [1, 1, 0, 1, 0]
]
```

```
# Орієнтований граф G2 (матриця суміжності)
G2 = [
    [0, 1, 1, 0],  # v1
    [0, 1, 0, 0],  # v2 (петля)
    [0, 0, 0, 1],  # v3
    [1, 1, 0, 0]   # v4
]
```

У випадку, коли необхідно задати зважений граф, замість бінарних ознак наявності чи відсутності ребер (0 та 1) у відповідних позиціях задається числове значення ваги відповідного ребра:

```
adj_matrix_nd = [
    [0, 5, 2],
    [5, 0, 1],
    [2, 1, 0]
]
```

```
INF = float('inf')
adj_matrix_dir = [
    [0,   5,   2],
    [INF, 0,   1],
    [INF, INF, 0]
]
```

Подання графа у вигляді матриці суміжності є інтуїтивно зрозумілим і простим для реалізації, що робить його зручним при розробці та аналізі алгоритмів, у яких часто виконується перевірка наявності ребра між заданими вершинами. Водночас такий спосіб представлення потребує обсягу пам’яті, пропорційного квадрату кількості вершин, незалежно від фактичної кількості ребер у графі. Тому використання матриці суміжності є малоефективним для великих розріджених графів, які широко застосовуються на практиці.

Матриця інцидентності графа задається за схожим принципом. Вона не є квадратною, оскільки окремі рядки відповідають вершинам, а стовпчики — ребрам графа.

Реалізації матриць інцидентності у мові Python для графів, зображених на рисунку 3.1, виглядають наступним чином.

```
# Матриця інцидентності неорієнтованого графа G1
# Рядки — вершини v1..v5, стовпці — ребра e1..e7
I_G1 = [
    [1, 1, 1, 0, 0, 0, 0],  # v1
    [0, 0, 1, 1, 1, 0, 0],  # v2
    [0, 0, 0, 0, 1, 1, 0],  # v3
    [1, 0, 0, 0, 0, 1, 1],  # v4
    [0, 1, 0, 1, 0, 0, 1]   # v5
]
```

```
# Матриця інцидентності орієнтованого графа G2
#  1 — дуга виходить
# -1 — дуга входить
#  0 — немає інцидентності
I_G2 = [
    [ 1,  1,  0, -1,  0,  0],  # v1
    [-1,  0,  0,  0, -1,  0],  # v2
    [ 0, -1,  1,  0,  0,  0],  # v3
    [ 0,  0, -1,  1,  1,  0]   # v4
]
```

Матриця інцидентності дозволяє зручно аналізувати структуру зв'язків між вершинами та ребрами, однак у практичному програмуванні використовується рідше через складність побудови та вищі витрати пам'яті.

Список суміжності є рекомендованим і найбільш часто вживаним способом задавання графів у мові Python.

Не зважаючи на те, що цей метод має у назві слово «список», найбільш природною структурою даних для його реалізації у мові Python є словник (dict). Є декілька узагальнених способів задавання графу за допомогою списку суміжності:

– словник зі списком (list) суміжних вершин — dict(list);

– словник з множиною (set) суміжних вершин — dict(set), використання множини замість списку дозволяє автоматично уникнути дублювання;

– спеціальний тип словника зі списком (list) суміжних вершин — defaultdict(list), в цьому типі словника не потрібно перевіряти, чи є вже вершина, що ми хочемо додати, серед ключів у словнику: звернення до вершини, якої ще немає у словнику автоматично додає її у словник з пустим списком значень.

Варіанти реалізації списків суміжності мовою Python для графів, зображених на рисунку 3.1, виглядають так.

```
# список суміжності як звичайний словник зі списками dict(list)
G1_adj_list = {
    'v1': ['v2', 'v4', 'v5'],
    'v2': ['v1', 'v3', 'v5'],
    'v3': ['v2', 'v4'],
    'v4': ['v1', 'v3', 'v5'],
    'v5': ['v1', 'v2', 'v4']
}
```

```
G2_adj_list = {
    'v1': ['v2', 'v3'],
    'v2': ['v2'],        # петля
    'v3': ['v4'],
    'v4': ['v1', 'v2']
}


# список суміжності як словник множин dict(set)
G1_adj_set = {
    'v1': {'v2', 'v4', 'v5'},
    'v2': {'v1', 'v3', 'v5'},
    'v3': {'v2', 'v4'},
    'v4': {'v1', 'v3', 'v5'},
    'v5': {'v1', 'v2', 'v4'}
}

G2_adj_set = {
    'v1': {'v2', 'v3'},
    'v2': {'v2'},
    'v3': {'v4'},
    'v4': {'v1', 'v2'}
}


# список суміжності як спеціальний словник defaultdict(list)
from collections import defaultdict
G1_dd = defaultdict(list)
G1_dd['v1'] = ['v2', 'v4', 'v5']
G1_dd['v2'] = ['v1', 'v3', 'v5']
G1_dd['v3'] = ['v2', 'v4']
G1_dd['v4'] = ['v1', 'v3', 'v5']
G1_dd['v5'] = ['v1', 'v2', 'v4']

from collections import defaultdict
G2_dd = defaultdict(list)
G2_dd[1] = ['v2', 'v3']
G2_dd[2] = ['v2']
G2_dd[3] = ['v4']
G2_dd[4] = ['v1', 'v2']
```

Список суміжності з використанням defaultdict(list) зручний для автоматичної побудови графа з інших подань, наприклад, набору ребер:

```
G1_from_edges = defaultdict(list)
G1_edges = [
    ('v1', 'v2'), ('v1', 'v4'), ('v1', 'v5'),
    ('v2', 'v3'), ('v2', 'v5'), ('v3', 'v4'), ('v4', 'v5')
]
```

```
for u, v in G1_edges:
    G1_from_edges[u].append(v)
    G1_from_edges[v].append(u)
```

Основною перевагою цього способу є економне використання пам’яті, оскільки зберігаються лише наявні ребра, що робить його особливо ефективним для великих розріджених графів. Такий формат подання забезпечує швидкий доступ до всіх вершин, суміжних із заданою, і добре узгоджується з реалізацією базових алгоритмів обходу та пошуку шляхів. Список суміжності природно реалізується засобами Python, зокрема за допомогою словників і динамічних списків, що спрощує програмну реалізацію та підвищує читабельність коду.

Водночас цей спосіб має і певні обмеження. Перевірка наявності ребра між двома довільними вершинами в загальному випадку потребує перегляду списку суміжності, що є менш ефективним порівняно з матрицею суміжності. Для щільних графів, де кількість ребер наближається до максимально можливої, список суміжності може втрачати свої переваги щодо компактності.

Задавання графу у вигляді *списку ребер* виглядає як список кортежів, кожен з яких складається з двох або трьох елементів.

Варіанти реалізації списків ребер мовою Python для графів, зображених на рисунку 3.1, виглядають наступним чином.

```
# список ребер неорієнтованого графа G1
G1_edges = [
    ('v1', 'v2'), ('v1', 'v4'),
    ('v1', 'v5'), ('v2', 'v3'),
    ('v2', 'v5'), ('v3', 'v4'),
    ('v4', 'v5')
]


# список ребер орієнтованого графа G2
G2_edges = [
    ('v1', 'v2'), ('v1', 'v3'), ('v2', 'v2'),
    ('v3', 'v4'), ('v4', 'v1'), ('v4', 'v2')
]
```

Для зваженого графа до кожного кортежу, що описує ребро графа, додається чисельне значення ваги, наприклад:

```
weighted_graph = [
    ('A', 'B', 5),
    ('A', 'C', 2),
    ('B', 'C', 1)
]
```

Цей формат задання графа є досить простим та інтуїтивно зрозумілим, він може використовуватись при зчитуванні чи зберіганні інформації про граф. Проте він є незручним при виконанні пошуку та обходу графа.

При описі графів із використанням мови Python доцільно звернути увагу на об’єктно-орієнтований підхід, оскільки класи та об’єкти є природним і базовим механізмом роботи з даними в цій мові програмування. Застосування класів дозволяє інкапсулювати структуру графа та пов’язані з ним операції, такі як додавання вершин і ребер, виконання обходів та аналіз властивостей графа. Такий підхід забезпечує наочність, зручність розширення та повний контроль над реалізацією алгоритмів, що робить його особливо доцільним у навчальних і дослідницьких цілях.

Приклад простого класу графа, який використовує опис графа за допомогою списку суміжності:

```
from collections import defaultdict

class Graph:
    def __init__(self, directed=False):
        self.directed = directed
        self.adj = defaultdict(list)

    def add_vertex(self, v):
        _ = self.adj[v]

    def add_edge(self, u, v):
        self.adj[u].append(v)
        if not self.directed:
            self.adj[v].append(u)

    def vertices(self):
        return list(self.adj.keys())

    def neighbors(self, v):
        return self.adj[v]
```

Цей досить простий клас дозволяє задавати вершини та ребра графа, а також отримувати список вершин та усі суміжні вершини для вказаної. Він є універсальним: за його допомогою можна задавати орієнтовані та неорієнтовані графи. Приклад використання цього класу:

```
# створення та робота з неорієнтованим графом G1
G1_custom = Graph(directed=False)

for v in [1, 2, 3, 4, 5]:
    G1_custom.add_vertex(v)

G1_custom.add_edge(1, 2)
G1_custom.add_edge(1, 3)
G1_custom.add_edge(2, 3)
G1_custom.add_edge(2, 4)
G1_custom.add_edge(3, 5)
G1_custom.add_edge(4, 5)

print(G1_custom.vertices())   # виведе [1, 2, 3, 4, 5]
print(G1_custom.neighbors(2)) # виведе [1, 3, 4]

# створення та робота з орієнтованим графом G2
for v in [1, 2, 3, 4]:
    G2_custom.add_vertex(v)

G2_custom.add_edge(1, 2)
G2_custom.add_edge(1, 3)
G2_custom.add_edge(3, 2)
G2_custom.add_edge(2, 4)
G2_custom.add_edge(3, 4)

print(G2_custom.vertices())   # виведе [1, 2, 3, 4]
print(G2_custom.neighbors(2)) # виведе [4]
```

Використання класів є гнучким і концептуально завершеним підходом до роботи з графами. Такий спосіб надає розробнику повний контроль над структурою графа та доступними над ним операціями, дозволяючи поступово переходити від реалізації базових дій із вершинами та ребрами до побудови складних алгоритмів пошуку і обходу, а також до аналізу властивостей графів. Водночас цей підхід потребує від розробника глибшого розуміння теорії графів і передбачає самостійну реалізацію відповідних алгоритмів та механізмів перевірки коректності.

Іншим підходом до роботи з графами є використання сторонніх спеціалізованих бібліотек. Зокрема, бібліотека **NumPy** орієнтована на ефективну та високопродуктивну обробку матричних структур даних, що робить її зручною для представлення графів у вигляді матриць суміжності або інцидентності. Бібліотека **NetworkX** є спеціалізованим інструментом для роботи з графами та містить готові реалізації основних алгоритмів, зокрема алгоритмів обходу вершин і ребер, пошуку найкоротших шляхів, аналізу зв'язності, а також засоби візуалізації графів.

Застосування таких бібліотек є доцільним для розв'язання прикладних задач, виконання досліджень і швидкого прототипування, оскільки вони надають широкий набір готових структур і алгоритмів. Водночас використання спеціалізованих бібліотек обмежує можливості детального вивчення принципів роботи алгоритмів, тому в навчальних цілях доцільніше надавати перевагу самостійній реалізації базових структур і методів опрацювання графів.

Контрольні запитання

1. Назвіть способи представлення графів, які найчастіше застосовуються у практиці програмування.

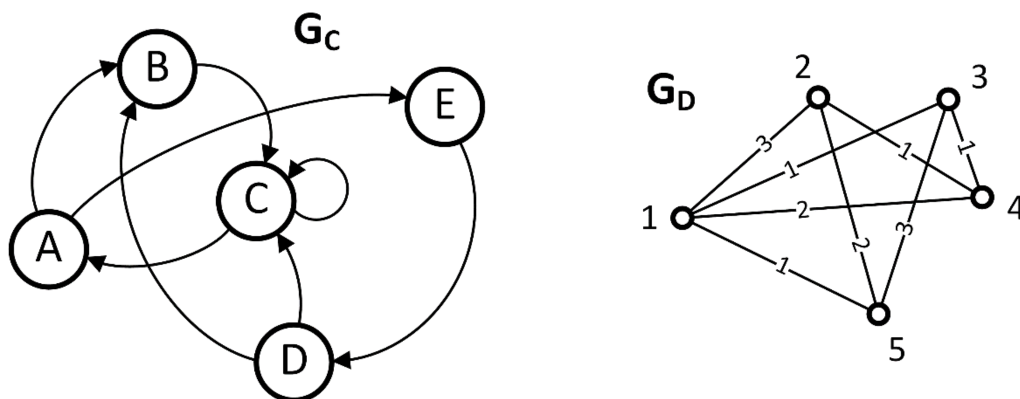

Рисунок 3.2 – Графи для завдань

2. Складіть загальний опис у вигляді матриці суміжності для графів, зображених на рисунку 3.2. В яких рисах вони є спільними, і чим відрізняються?

Чи є якісь особливі випадки? Створіть опис матриць суміжності за допомогою структур даних мови Python за прикладом, поданим у розділі.

3. Складіть у загальній формі матриці інцидентності для графів, зображених на рисунку 3.2. Поясніть, у чому відмінність між матрицями суміжності та інцидентності на прикладі створених описів графів. Складіть опис матриць інцидентності за допомогою конструкцій мови Python.

4. Складіть списки суміжності для графів, зображених на рисунку 3.2, у загальній формі. Складіть опис списків суміжності на мові Python у всіх варіантах, що наведені у цьому розділі методичних вказівок. Поясніть, у яких випадках буде більш зручним той чи інший спосіб подання списку суміжності.

5. Складіть списки ребер графів, зображених на рисунку 3.2, у загальній формі. Складіть опис списків ребер цих графів за допомогою конструкцій мови Python.

6. Спробуйте створити опис графів, зображених на рисунку 3.2, у вигляді класу у Python. Поясність сильні сторони та недоліки такого підходу.

7. Дослідіть можливості сторонньої бібліотеки NetworkX. Задайте графи, зображені на рисунку 3.2 за допомогою можливостей цієї бібліотеки. Спробуйте створити візуалізацію заданих графів (для цього додатково знадобиться стороння бібліотека matplotlib, також рекомендується для спрощення виконання завдання скористатись можливостями платформи Google Colab).

4 ПОШУК ШЛЯХУ МІЖ ЗАДАНИМИ ВЕРШИНАМИ

Однією з базових задач теорії графів є пошук шляху між двома заданими вершинами. Така задача виникає при аналізі зв'язності графа, маршрутизації в мережах, побудові послідовностей переходів між станами та в багатьох прикладних задачах. У найпростішій постановці необхідно визначити, чи існує шлях між початковою вершиною s та цільовою вершиною t, а у разі його існування — побудувати сам шлях.

Для пошуку простого шляху між двома вершинами зазвичай використовується алгоритм *обходу графа в глибину* (Depth-First Search, DFS).

Основні кроки алгоритму:

1) почати з початкової вершини та додати її до поточного шляху;
2) якщо поточна вершина збігається з цільовою, шлях вважається знайденим;
3) переглянути всі суміжні вершини, які ще не входять до поточного шляху;
4) рекурсивно виконати пошук для кожної з таких вершин;
5) якщо шлях не знайдено, вилучити поточну вершину зі шляху та повернутися назад.

Алгоритм є концептуально простим і наочним, легко реалізується рекурсивно та добре узгоджується з поданням графа у вигляді списку суміжності. Він дозволяє не лише перевірити існування шляху, а й отримати конкретну послідовність вершин.

Основним недоліком є висока обчислювальна складність у найгіршому випадку, оскільки алгоритм може перебирати велику кількість можливих шляхів. Для великих або щільних графів такий підхід стає малоефективним. Крім того, стандартний алгоритм DFS не гарантує знаходження найкоротшого шляху.

Реалізація пошуку простого шляху між двома вершинами за допомогою алгоритму DFS у вигляді функції на мові Python:

```
def find_path(graph, start, goal, path=None):
    if path is None:
        path = []
```

```
    path.append(start)

    if start == goal:
        return path

    for neighbor in graph.get(start, []):
        if neighbor not in path:
            result = find_path(graph, neighbor, goal, path)
            if result is not None:
                return result

    path.pop()
    return None
```

Приклад використання функції пошуку шляху. Для демонстрації роботи функції пошуку розглянемо неорієнтований граф з наступною конфігурацією вершин та ребер (рис. 4.1):

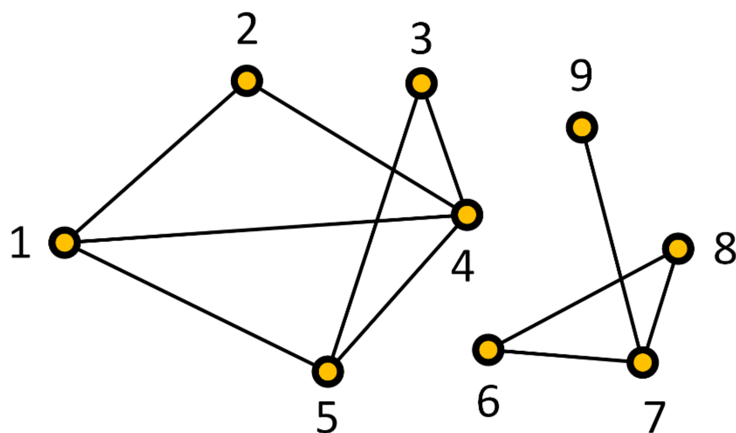

Рисунок 4.1 – Приклад неорієнтованого графу

Виконаємо пошук шляхів між вершиною 1 та вершинами 3, 5 та 9:

```
# приклад використання функції пошуку шляху для заданого графу g1
g1 = {
    1: [2, 4, 5],
    2: [1, 4],
    3: [4, 5],
    4: [1, 2, 3, 5],
    5: [1, 3, 4],
    6: [7, 8],
    7: [6, 8, 9],
    8: [6, 7],
    9: [7]
}
```

```
p_1_3 = find_path(g1, 1, 3)
print("path between 1 and 3:", p_1_3)
p_1_5 = find_path(g1, 1, 5)
print("path between 1 and 5:", p_1_5)
p_1_9 = find_path(g1, 1, 9)
print("path between 1 and 9:", p_1_9)
```

Отримаємо наступні результати:

```
path between 1 and 3: [1, 2, 4, 3]
path between 1 and 5: [1, 2, 4, 3, 5]
path between 1 and 9: None
```

Між зазначеними вершинами може існувати декілька шляхів, проте функція повертає перший знайдений шлях. Він не обов'язково може бути найбільш оптимальним. Оскільки граф є непов'язаним, між деякими парами вершин не існує шляхів (наприклад між 1 та 9). У таких випадках функція не формує послідовність вершин шляху, а повертає значення None, що свідчить про неможливість переходу між заданими вершинами.

Гамільтонів шлях — це простий шлях у графі, який проходить через усі вершини рівно один раз. Якщо такий шлях починається і закінчується в одній і тій самій вершині, він називається *гамільтоновим циклом*. На відміну від задачі пошуку довільного шляху, у цьому випадку накладається жорстка вимога відвідати кожну вершину графа без повторень.

Гамільтонів шлях існує не в кожному графі. Низка властивостей і структурних обмежень графа можуть унеможливлювати його побудову:

– граф має бути пов'язаним, у непов'язаному графі гамільтонів шлях не може існувати;

– наявність вершини зі ступенем 0 повністю виключає можливість побудови гамільтонового шляху, а вершини зі ступенем 1 унеможливлюють існування гамільтонового циклу;

– у графах із «вузькими місцями» (вершинами-розрізами) шлях може бути заблокований;

– не існує простого універсального критерію існування гамільтонового шляху для довільного графа.

Вершина-розріз — це така вершина графа, видалення якої (разом з усіма інцидентними ребрами) робить граф непов'язаним, тобто розвалює його на дві або більше компонент пов'язаності.

Загальна задача визначення наявності гамільтонового шляху належить до класу NP-повних задач, що означає відсутність відомих ефективних алгоритмів для її розв'язання у загальному випадку.

Алгоритм пошуку гамільтонового шляху. Найбільш поширеним підходом є повний перебір із поверненням (backtracking), який базується на обході графа в глибину. Алгоритм послідовно будує шлях, додаючи до нього суміжні вершини, які ще не були відвідані. Якщо на певному етапі подальше розширення шляху неможливе, виконується повернення до попереднього кроку та перевірка альтернативних варіантів.

Основні кроки алгоритму:

1) обрати початкову вершину;

2) додати її до поточного шляху;

3) послідовно переходити до суміжних вершин, які ще не входять до шляху;

4) якщо довжина шляху дорівнює кількості вершин графа — гамільтонів шлях знайдено;

5) якщо продовження неможливе, виконати повернення та спробувати інший варіант.

Алгоритм є універсальним і дозволяє знайти гамільтонів шлях, якщо він існує. Його реалізація добре поєднується зі списком суміжності та рекурсивним DFS. Водночас алгоритм має експоненційну складність і швидко стає непридатним для великих графів.

Оцінка обчислювальної складності. У найгіршому випадку часові витрати становлять $O(V!)$, де V — кількість вершин графа. Просторова складність становить $O(V)$ для збереження поточного шляху та множини відвіданих вершин.

Розглянемо реалізацію методу погуку гамільтонового шляху мовою Python на прикладі графу, зображеного на рисунку 4.2. Цей граф схожий на граф на

рисунку 4.1, але він є пов'язаним за рахунок утворення переходу між вершинами 2 та 6. Граф має у своєму складі вершину з ступенем 1 (вершина 9), а отже це означає, що граф не може мати гамільтонів цикл. Проте, цей граф відповідає усім умовам для наявності гамільтонового шляху.

Представлення цього графу у вигляді списку суміжності мовою Python у форматі dict(list) виглядає наступним чином:

```
g2 = {
    1: [2, 4, 5],
    2: [1, 4, 5, 6],
    3: [4, 5],
    4: [1, 2, 3, 5],
    5: [1, 2, 3, 4],
    6: [2, 7, 8],
    7: [6, 8, 9],
    8: [6, 7],
    9: [7]
}
```

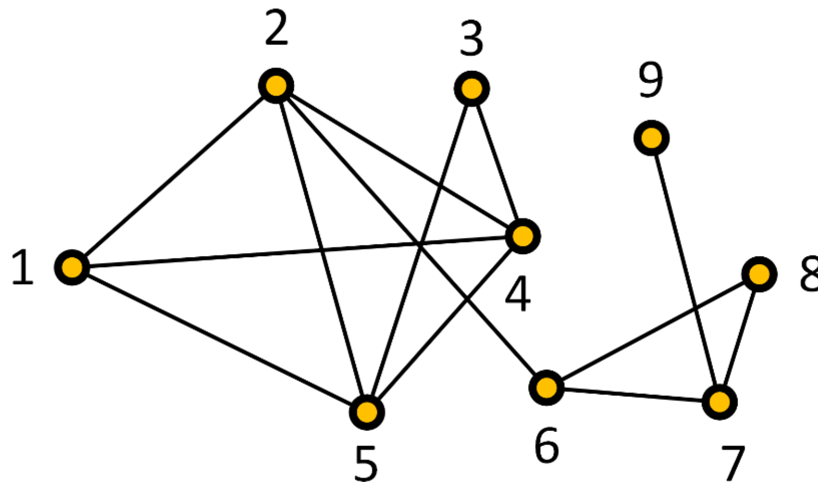

Рисунок 4.2 – Приклад графу для пошуку гамільтонового шляху

Програма на Python, що реалізує алгоритм пошуку гамільтонового шляху:

```
def hamiltonian_path(graph):
    V = len(graph)

    def backtrack(path, visited):
        if len(path) == V:
            return path
```

```
        current = path[-1]
        for neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                path.append(neighbor)

                result = backtrack(path, visited)
                if result is not None:
                    return result

                path.pop()
                visited.remove(neighbor)
        return None

    for start in graph:
        result = backtrack([start], {start})
        if result is not None:
            return result

    return None
```

Якщо гамільтонів шлях існує, функція повертає список вершин у порядку їх проходження. Якщо жоден із можливих варіантів не задовольняє умову, повертається значення None.

Результат перевірки наявності гамільтонового шляху у графах g2 та g1, заданих на рисунках 4.1 та 4.2:

```
h_path_g2 = hamiltonian_path(g2)
print("H-path for g2:", h_path_g2)

h_path_g1 = hamiltonian_path(g1)
print("H-path for g1:", h_path_g1)
```

Результат виводу:

```
H-path for g2: [1, 4, 3, 5, 2, 6, 8, 7, 9]
H-path for g1: None
```

Таким чином ми можемо переконатись, що у графі g1 неможливо знайти гамільтонів шлях, оскільки він не відповідає умовам, а у графі g2 знайдено один з двох можливих шляхів від вершини 1:

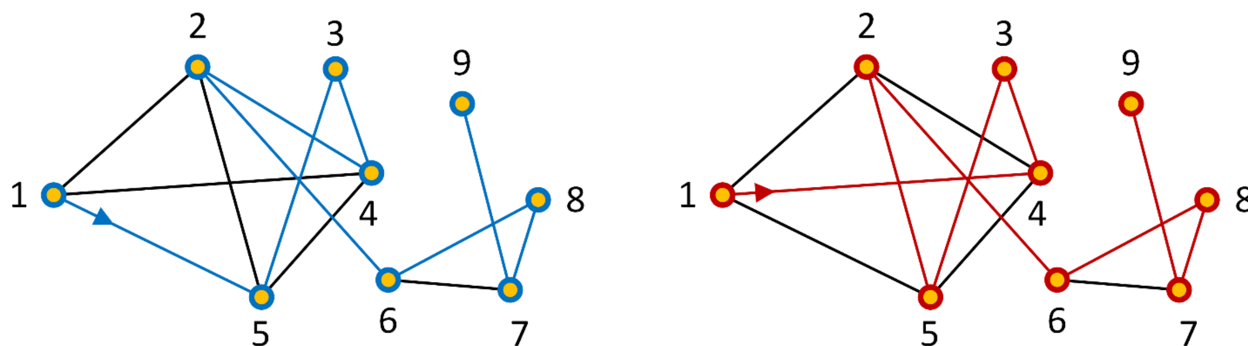

Рисунок 4.3 – Можливі гамільтонові шляхи у графі g2

Шлях Ейлера — це шлях у графі, який проходить через кожне ребро рівно один раз. *Цикл Ейлера* — це ейлерів шлях, у якого початкова і кінцева вершини збігаються. На відміну від гамільтонових шляхів, де ключову роль відіграють вершини, у задачі Ейлера основна увага приділяється ребрам графа.

Для неорієнтованого графа мають виконуватись такі необхідні та достатні умови, щоб стверджувати, що він має шлях або цикл Ейлера:

– граф має бути зв’язним (якщо не враховувати ізольовані вершини);

– ейлерів цикл існує тоді і тільки тоді, коли ступені всіх вершин є парними;

– ейлерів шлях (але не цикл) існує, якщо рівно дві вершини мають непарний ступінь, а всі інші — парний;

– якщо кількість вершин із непарним ступенем більша за дві, ейлерів шлях не існує.

Для орієнтованого графа умови існування ейлерового шляху та ейлерового циклу формулюються через вхідні та вихідні ступені вершин і дещо відрізняються від випадку неорієнтованого графа.

Цикл Ейлера існує тоді і тільки тоді, коли виконуються обидві умови:

– для кожної вершини $in_degree(v) = out_degree(v)$;

– граф є слабко зв’язним (без урахування ізольованих вершин).

Шлях Ейлера існує, якщо виконуються такі умови:

– рівно одна вершина має $out_degree(v) = in_degree(v) + 1$ — це початкова вершина шляху;

– рівно одна вершина має $in_degree(v) = out_degree(v) + 1$ — це кінцева вершина шляху;

– для всіх інших вершин $in_degree(v) = out_degree(v)$;

– граф є слабко зв’язним (без ізольованих вершин).

Якщо кількість вершин, для яких $in_degree(v) \neq out_degree(v)$ перевищує дві, або різниця між вхідним і вихідним ступенями не дорівнює ±1 для «крайніх» вершин, то для такого графу шлях Ейлера не існує.

Для пошуку ейлерового шляху або циклу в теорії графів застосовують кілька підходів, серед яких найбільш ефективним і поширеним є алгоритм Гієргольцера. Концептуально близьким до нього є метод послідовного видалення та злиття циклів, що часто використовується для теоретичного обґрунтування алгоритму. Інші підходи, зокрема перебірні методи або обходи з контролем використаних ребер, мають обмежене практичне застосування через високу обчислювальну складність.

Основна ідея алгоритму полягає у послідовному проходженні ребер графа з їх одночасним видаленням, доки всі ребра не будуть використані рівно один раз.

Загальна схема алгоритму:

1) обрати стартову вершину (для циклу — будь-яку, для шляху — одну з вершин з непарним ступенем).

2) рухатися вздовж довільного доступного ребра, видаляючи його з графа.

3) якщо поточна вершина не має невикористаних ребер, додати її до шляху.

4) повторювати кроки, доки всі ребра не буде пройдено.

На відміну від гамільтонової задачі, пошук ейлерового шляху не є обчислювально складним. Оцінка обчислювальної складності алгоритму Гієргольцера становить O(E), де E — кількість ребер графа, оскільки кожне ребро обробляється рівно один раз. Просторова складність також дорівнює O(E).

Розглянемо у якості прикладу для пошуку циклу Ейлера наступний граф, зображений на рисунку 4.4. Він повністю задовольняє умовам існування циклу Ейлера.

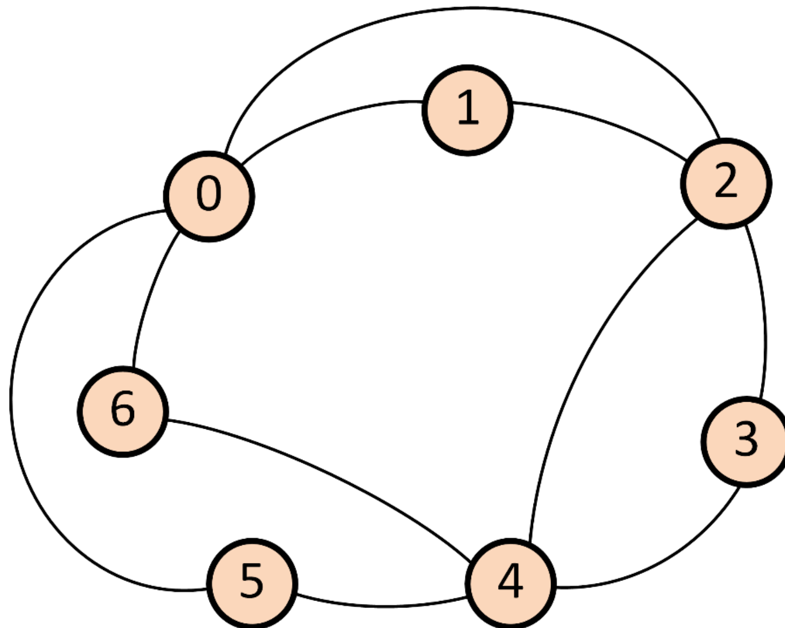

Рисунок 4.4 – Граф для пошуку циклу Ейлера

Цей граф задається списком суміжності у форматі dict(list) наступним чином:

```
g3 = {
    0: [1, 2, 5, 6],
    1: [0, 2],
    2: [0, 1, 3, 4],
    3: [2, 4],
    4: [2, 3, 5, 6],
    5: [0, 4],
    6: [0, 4]
}
```

Реалізація пошуку ейлерового шляху або циклу (алгоритм Гієргольцера):

```
# алгоритм пошуку циклу Ейлера (алгоритм Гієргольцера)
def euler_path(graph):
    # копія графа, щоб не змінювати початкову структуру
    g = {v: adj.copy() for v, adj in graph.items()}

    # перевірка умов відповідності - цикл або шлях
    odd_vertices = [v for v in g if len(g[v]) % 2 != 0]
    if len(odd_vertices) not in (0, 2):
        return None
    # визначаємо стартову вершину
    start = odd_vertices[0] if odd_vertices else next(iter(g))
    stack = [start]
    path = []
    # основна частина алгоритму
    while stack:
```

```
        v = stack[-1]
        if g[v]:
            u = g[v].pop()
            g[u].remove(v)
            stack.append(u)
        else:
            path.append(stack.pop())

    return path[::-1]
```

У разі існування ейлерового циклу або шляху функція повертає список вершин у порядку їх проходження. Якщо умови існування не виконуються, повертається значення None.

Приклад використання. Для реалізації прикладу виконаємо пошук циклу Ейлера в графах, зображених на рис. 4.4 та 4.1:

```
# приклад використання функції пошуку шляху Ейлера
ep_g3 = euler_path(g3)
print("Euler path for g3:", ep_g3)
ep_g2 = euler_path(g2)
print("Euler path for g2:", ep_g2)
```

Результати виводу:

```
Euler path for g3: [0, 6, 4, 5, 0, 2, 4, 3, 2, 1, 0]
Euler path for g2: None
```

Контрольні запитання

1. Що таке шлях у графі? Які алгоритми пошуку шляхів у графі ви можете назвати? Сформулюйте кроки алгоритму пошуку шляху при обході графа в глибину. Які є недоліки у алгоритму?

2. Скористайтесь реалізацією алгоритму пошуку шляху між довільно взятими вершинами на мові Python, наведеною у розділі та відшукайте шлях між вершинами графів, наведених на рисунку 3.2: між A та D для G_C, між D та E для G_C, між 2 та 3 для G_D, між 4 та 5 для G_D.

3. Які шляхи с особливими властивостями можуть існувати у графі? Що таке гамільтонів шлях? Які його особливості? Чи у будь-якому графі може

існувати гамільтонів шлях? Опишіть алгоритм його пошуку. Спробуйте знайти гамільтонові шляхи в алгоритмах на рисунку 3.2.

4. Що таке цикл і в чому його відмінність від шляху? Чи існують гамільтонові цикли у графах, зображених на рисунку 3.2? Виконайте пошук за допомогою описаної програми на Python (треба адаптувати програму для пошуку саме циклу, а не шляху).

5. Що таке шлях та цикл Ейлера? Яким умовам має відповідати граф, щоб стверджувати, що він має шлях/цикл Ейлера? Які відмінності в цих вимогах до орієнтованих та неорієнтованих графів? Виконайте пошук шляху Ейлера у графах, зображених на рисунку 3.2 за допомогою програми, наведеної у цьому розділі.

5 ПОШУК НАЙКОРОТШИХ ТА ОПТИМАЛЬНИХ ШЛЯХІВ

У попередньому розділі було розглянуто алгоритм пошуку довільних шляхів у графах. Такий алгоритм дозволяє встановити факт існування шляху між заданими вершинами та отримати один із можливих варіантів проходження, однак не гарантується, що знайдений шлях є найкоротшим або оптимальним за певним критерієм. На практиці ж у більшості прикладних задач виникає потреба знайти саме найвигідніший шлях — з мінімальною кількістю переходів або з найменшою сумарною вагою ребер.

Задача пошуку найкоротших або оптимальних шляхів між вершинами графа дуже часто потребує вирішення у таких галузях як маршрутизація, аналіз мереж, транспортне планування, логістика, телекомунікаційні системи та багатьох інших прикладних областях.

Поняття довжини шляху залежить від типу графа та формалізації задачі.

У незважених графах довжина шляху визначається кількістю ребер (переходів) між вершинами. Найкоротшим вважається шлях, що містить мінімальну кількість ребер.

У зважених графах кожному ребру поставлена у відповідність числова вага, яка може інтерпретуватися як відстань, вартість, час або інший ресурс. Довжина шляху визначається як сума ваг усіх ребер, що входять до цього шляху. Оптимальним є шлях із мінімальною сумарною вагою.

У загальному випадку під оптимальним шляхом розуміють шлях, який мінімізує задану цільову функцію (кількість переходів, суму ваг, час проходження тощо).

Алгоритми пошуку найкоротших шляхів можна умовно поділити за кількома ознаками:

- пошук шляху між однією парою вершин;
- пошук шляхів від однієї вершини до всіх інших;
- пошук найкоротших шляхів між усіма парами вершин;
- робота з незваженими або зваженими графами;

– можливість виявлення від’ємних ваг ребер.

Для розв’язання цих задач використовуються різні алгоритми, які відрізняються за умовами застосування, обчислювальною складністю та форматом результату.

Алгоритм Флойда-Уоршелла (Floyd-Warshall) — це класичний алгоритм динамічного програмування, призначений для знаходження найкоротших шляхів між усіма парами вершин у зваженому графі. Він може працювати як з орієнтованими, так і з неорієнтованими графами та допускає наявність від’ємних ваг ребер, за умови відсутності від’ємних циклів.

Алгоритм послідовно розглядає кожну вершину графа як проміжну і перевіряє, чи можна покращити поточну відстань між будь-якою парою вершин i та j, дозволивши проходження через вершину k.

Ключова рекурентна формула має вигляд:

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

де $d_{ij}^{(k)}$ — найкоротша відстань між вершинами i та j, якщо дозволено використовувати лише вершини з множини $\{1,2,\dots,k\}$ як проміжні; k — номер поточної проміжної вершини.

Початково вважається, що жодні проміжні вершини не використовуються, а відстані задаються безпосередньо з матриці ваг.

Найбільш природним і зручним способом подання графа для алгоритму Флойда-Уоршелла є матриця суміжності з вагами, оскільки алгоритм безпосередньо працює з усіма парами вершин.

Часова складність алгоритму оцінюється як $O(V^3)$, просторова складність — $O(V^2)$. Через кубічну складність алгоритм доцільно застосовувати лише для графів малого та середнього розміру.

Нижче наведено приклад реалізації алгоритму Флойда-Уоршелла для орієнтованого зваженого графа, заданого матрицею суміжності.

```
def floyd_warshall(adj_matrix):
    V = len(adj_matrix)
```

```
    dist = [row[:] for row in adj_matrix]  # копія матриці

    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

Функція повертатиме матрицю, яка міститиме довжини найкоротших шляхів між усіма парами вершин графа.

Розглянемо використання цієї функції для наступних графів:

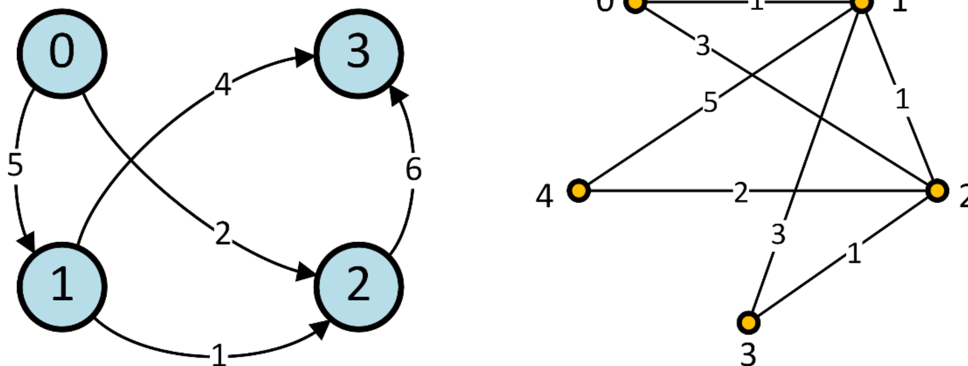

Рисунок 5.1 – Орієнтований та неорієнтований графи

Опис цих графів мовою Python у вигляді матриць суміжності:

```
# орієнтований граф g4
N = float('inf')

g4 = [
    [0, 5, 2, N],
    [N, 0, 1, 4],
    [N, N, 0, 6],
    [N, N, N, 0]
]
```

```
# неорієнтований граф g5
N = float('inf')

g5 = [
    [0, 1, 3, N, N],
    [1, 0, 1, 3, 5],
    [3, 1, 0, 1, 2],
    [N, 3, 1, 0, N],
    [N, 5, 2, N, 0]
]
```

Приклад виконання функції `floyd_warshall()` для графу g4:

```
shortest_paths_g4 = floyd_warshall(g4)
print("shortest path between 0 and 3:", shortest_paths_g4[0][3])

print("full matrix of shortest paths:")
```

```
for row in shortest_paths_g4:
    print(row)
```

Результати виводу:

```
shortest path between 0 and 3: 8
full matrix of shortest paths:
[0, 5, 2, 8]
[inf, 0, 1, 4]
[inf, inf, 0, 6]
[inf, inf, inf, 0]
```

Приклад виконання функції `floyd_warshall()` для графу g5:

```
shortest_paths_g5 = floyd_warshall(g5)
print("shortest path between 0 and 4:", shortest_paths_g5[0][4])

print("full matrix of shortest paths:")
for row in shortest_paths_g5:
    print(row)
```

Результати виводу:

```
shortest path between 0 and 4: 4
full matrix of shortest paths:
[0, 1, 2, 3, 4]
[1, 0, 1, 2, 3]
[2, 1, 0, 1, 2]
[3, 2, 1, 0, 3]
[4, 3, 2, 3, 0]
```

Алгоритм Флойда-Уоршелла є універсальним і зручним для аналізу повної картини шляхів у графі, проте його висока обчислювальна складність обмежує практичне використання великими або розрідженими графами. Тому на практиці він часто застосовується у навчальних задачах, аналізі невеликих мереж або як еталонний алгоритм для порівняння з більш ефективними методами.

Алгоритм Форда-Беллмана (Bellman-Ford) — це класичний алгоритм пошуку найкоротших шляхів від однієї заданої вершини до всіх інших у зваженому графі. Його ключова особливість полягає в тому, що він коректно працює з від'ємними вагами ребер та дозволяє виявляти від'ємні цикли.

Алгоритм базується на послідовній *релаксації* ребер. Релаксація означає перевірку, чи можна покращити поточну оцінку відстані до вершини v через вершину u:

$$d(v) > d(u) + w(u, v)$$

Якщо нерівність виконується, відстань до вершини v оновлюється.

Основні кроки алгоритму:

1) ініціалізувати відстані: відстань до стартової вершини — 0; до всіх інших — ∞;

2) повторити $v - 1$ разів: виконати релаксацію всіх ребер графа;

3) для перевірки наявності від’ємного циклу:

– виконати ще одну ітерацію релаксації;

– якщо хоча б одна відстань зменшується — у графі існує від’ємний цикл.

Обчислювальна складність оцінюється як $O(V \times E)$, де V — загальна кількість вершин, E — загальна кількість ребер графа, просторова складність оцінюється як $O(V)$.

Для алгоритму Форда-Беллмана найбільш зручним є подання графа у вигляді списку ребер, оскільки алгоритм багаторазово проходить по всіх ребрах.

Подання графів, зображених на рисунку 5.1, у вигляді списку ребер засобами мови Python виглядають наступним чином:

```
g4_loe_edges = [
    (0, 1, 5),
    (0, 2, 2),
    (1, 3, 4),
    (1, 2, 1),
    (2, 3, 6)
]
```

```
g5_loe_edges = [
    (0, 1, 1), (0, 2, 3), (1, 0, 1),
    (1, 2, 1), (1, 3, 3), (1, 4, 5),
    (2, 0, 3), (2, 1, 1), (2, 3, 1),
    (2, 4, 2), (3, 1, 3), (3, 2, 1),
    (4, 1, 5), (4, 2, 2)
]
```

Реалізація алгоритму пошуку найкоротших шляхів Форда–Беллмана у вигляді функції на Python:

```
def bellman_ford(vertices, edges, source):
    # ініціалізація відстаней
    dist = {v: float('inf') for v in vertices}
    dist[source] = 0

    # основні ітерації релаксації
    for _ in range(len(vertices) - 1):
        for u, v, w in edges:
```

```
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # перевірка наявності від’ємних циклів
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            raise ValueError("ERROR: There is negative cycle!")

    return dist
```

Функція повертає словник, який містить найкоротші відстані від стартової вершини до всіх інших вершин графа або повідомляє про наявність від’ємного циклу. Результати роботи функції для графів g4 та g5 подані нижче. Для роботи алгоритму необхідний також перелік вершин, для кожного з графів вони визначаються за допомогою додаткового обчислення.

```
# обчислення найкоротших шляхів від 0 для g4 - алгоритм Форда–Беллмана
# визначення списку вершин графу g4
vo_g4 = {v[0] for v in g4_loe_edges}
vi_g4 = {v[1] for v in g4_loe_edges}
g4_loe_vertices = sorted(vo_g4 | vi_g4)
# визначення відстаней
g4_distances = bellman_ford(g4_loe_vertices, g4_loe_edges, 0)
for v in g4_distances:
    print(f"g4: distance between 0 and {v}: {g4_distances[v]}")

# обчислення найкоротших шляхів від 0 для g5 - алгоритм Форда–Беллмана
# визначення списку вершин графу g5
vo_g5 = {v[0] for v in g5_loe_edges}
vi_g5 = {v[1] for v in g5_loe_edges}
g5_loe_vertices = sorted(vo_g5 | vi_g5)
# визначення відстаней
g5_distances = bellman_ford(g5_loe_vertices, g5_loe_edges, 0)
for v in g5_distances:
    print(f"g5: distance between 0 and {v}: {g5_distances[v]}")
```

Результати виводу:

```
g4: distance between 0 and 0: 0
g4: distance between 0 and 1: 5
g4: distance between 0 and 2: 2
g4: distance between 0 and 3: 8

g5: distance between 0 and 0: 0
g5: distance between 0 and 1: 1
g5: distance between 0 and 2: 2
g5: distance between 0 and 3: 3
g5: distance between 0 and 4: 4
```

Алгоритм Форда-Беллмана є надійним і універсальним способом пошуку найкоротших шляхів у зважених графах з від’ємними ребрами. Його основним недоліком є висока обчислювальна складність, що робить його малопридатним для великих графів. Тому на практиці його часто застосовують у задачах аналізу мереж, перевірки коректності графових моделей та як еталонний алгоритм у навчальних цілях.

Алгоритм Дейкстри також призначений для знаходження найкоротших шляхів від однієї початкової вершини до всіх інших вершин графа. Він застосовується до зважених графів без від’ємних ваг ребер, однаково, як до орієнтованих, так і неорієнтованих. Основна ідея алгоритму полягає в поступовому розширенні множини вершин з уже відомими найкоротшими відстанями, вибираючи на кожному кроці вершину з мінімальною поточною оцінкою відстані.

Основні кроки алгоритму в його базовій версії є наступними:

1) ініціалізувати відстані до всіх вершин значенням ∞.

2) відстань до початкової вершини встановити рівною 0.

3) поки є необроблені вершини:

– вибрати вершину з мінімальною поточною відстанню;

– виконати релаксацію всіх ребер, що з неї виходять.

Обчислювальна складність цього алгоритму оцінюється як $O(V^2)$ де V — кількість вершин графа. Цей варіант є прийнятним для щільних графів, але неефективним для великих розріджених. Для розріджених графів є окрема модифікація цього алгоритму, яка буде розглянута далі.

Найбільш доцільним способом подання графа для застосування алгоритму Дейкстри є список суміжності з вагами ребер, наприклад у вигляді словника. Нижче подано описи графів, зображених на рисунку 5.1 в такому форматі:

```
g4_ajlw = {
    0: [(1, 1), (2, 3)],
    1: [(0, 1), (2, 1), (3, 3), (4, 5)],
    2: [(0, 3), (1, 1), (3, 1), (4, 2)],
    3: [(1, 3), (2, 1)],
```

```
    4: [(1, 5), (2, 2)]
}

g5_ajlw = {
    0: [(1, 5), (2, 2)],
    1: [(2, 1), (3, 4)],
    2: [(3, 6)],
    3: []
}
```

Приклад реалізації алгоритму Дейкстри на Python

```
def dijkstra(graph, start):
    distances = {v: float('inf') for v in graph}
    distances[start] = 0
    visited = set()

    while len(visited) < len(graph):
        current = min(
            (v for v in graph if v not in visited),
            key=lambda v: distances[v],
            default=None
        )

        if current is None:
            break

        visited.add(current)

        for neighbor, weight in graph[current]:
            if distances[current] + weight < distances[neighbor]:
                distances[neighbor] = distances[current] + weight

    return distances
```

У розріджених графах кількість ребер E значно менша за V^2. Використання базовоі версії алгоритму Дейкстри стає неефективним, оскільки пошук мінімальної відстані серед усіх вершин займає лінійний час на кожному кроці. Для оптимізації роботи алгоритму застосовується пріоритетна черга (бінарна купа). Обчислювальна складність алгоритму після такої оптимізації оцінюється як $O((V+E)\log V)$. Це робить алгоритм придатним для ефективної обробки графів із тисячами й мільйонами вершин.

Приклад реалізації алгоритму Дейкстри з пріоритетною чергою на мові Python зображена нижче:

```
import heapq

def dijkstra_sparse(graph, start):
    distances = {v: float('inf') for v in graph}
    distances[start] = 0

    pq = [(0, start)]

    while pq:
        current_dist, current = heapq.heappop(pq)

        if current_dist > distances[current]:
            continue

        for neighbor, weight in graph[current]:
            new_dist = current_dist + weight
            if new_dist < distances[neighbor]:
                distances[neighbor] = new_dist
                heapq.heappush(pq, (new_dist, neighbor))

    return distances
```

Використання пріоритетної черги не змінює логіку алгоритму Дейкстри, але істотно підвищує його ефективність. Саме ця варіація алгоритму застосовується в більшості сучасних програмних реалізацій і бібліотек.

Алгоритм Шимбелла є одним із матричних методів пошуку найкоротших шляхів у графах. Він ґрунтується на використанні матриці суміжності з вагами ребер і дозволяє визначити найкоротші відстані між усіма парами вершин графа.

Ідея алгоритму полягає в послідовному уточненні оцінок довжин шляхів між вершинами шляхом багаторазового перерахунку матриці відстаней. Кожна ітерація враховує можливість проходження шляхів із більшою кількістю ребер. Після виконання достатньої кількості ітерацій отримуються найкоротші відстані між усіма вершинами графа.

Для застосування алгоритму Шимбелла граф задається матрицею ваг (матрицею суміжності з вагами):

$$A_{ij} = \begin{cases} 0, & i = j \\ w_{ij}, & \text{якщо існує ребро } i \to j \\ \infty, & \text{якщо ребро відсутнє.} \end{cases}$$

Алгоритм базується на ідеї, що кожен елемент матриці відстаней, зведеної у ступінь k — $d_{ij}^{(k)}$ містить інформацію про найкоротший шлях з вершини i у вершину j, який складається не більше ніж з k ребер.

Згідно цього алгоритму виконується послідовне оновлення матриці відстаней за правилом:

$$d_{ij}^{(k)} = \min\left(\left(d_{i1}^{(k-1)} + A_{1j}\right), \left(d_{i2}^{(k-1)} + A_{2j}\right), \ldots \left(d_{im}^{(k-1)} + A_{mj}\right)\right)$$
$$= \min_{m}\left(d_{im}^{(k-1)} + A_{mj}\right)$$

де $d_{ij}^{(k-1)}$– попереднє значення довжини, перед початком обчислень усі елементи матриці довжин $d_{ij}^{(0)}$ визначаються як ∞.

Процедура повторюється $|V|-1$ разів, оскільки у графі без від’ємних циклів найкоротший шлях не може містити більше ніж $|V|-1$ ребер.

Обчислювальна складність алгоритму становить $O(V^4)$, що зумовлено багаторазовим перерахунком матриці відстаней.

Приклад реалізації алгоритму Шимбелла у вигляді функції мовою Python:

```
def shimbel_algorithm(graph):
    vert_quantity = len(graph)
    dims = [row[:] for row in graph]

    for _ in range(vert_quantity - 1):
        new_dims = [[float('inf')] * vert_quantity for _ in
range(vert_quantity)]
        for i in range(vert_quantity):
            for j in range(vert_quantity):
                for k in range(vert_quantity):
                    new_dims[i][j] = min(new_dims[i][j], dims[i][k] +
graph[k][j])
        dims = new_dims

    return dims
```

Результати виконання алгоритму для графів, зображених на рисунку 5.1 (можна використати подання матриць суміжності для цих графів, що слідують одразу після рисунку):

```
distances = shimbel_algorithm(g4)
print("Calculated distances for g4:")
for row in distances:
    print(row)

distances = shimbel_algorithm(g5)
print("Calculated distances for g5:")
for row in distances:
    print(row)
```

Результати виводу:

```
Calculated distances for g4:
[0, 5, 2, 8]
[inf, 0, 1, 4]
[inf, inf, 0, 6]
[inf, inf, inf, 0]

Calculated distances for g5:
[0, 1, 2, 3, 4]
[1, 0, 1, 2, 3]
[2, 1, 0, 1, 2]
[3, 2, 1, 0, 3]
[4, 3, 2, 3, 0]
```

Алгоритм Шимбелла має переважно навчальне та теоретичне значення, оскільки наочно демонструє матричний підхід до задачі пошуку найкоротших шляхів. У практичних задачах для розв'язання аналогічних проблем зазвичай використовуються більш ефективні алгоритми, зокрема алгоритм Флойда-Уоршелла або алгоритм Дейкстри.

Алгоритм Оттермана є матричним методом пошуку найкоротших шляхів між усіма парами вершин графа. Він належить до класу алгоритмів динамічного програмування та є розвитком ідеї послідовного уточнення матриці відстаней, подібно до алгоритмів Шимбелла та Флойда-Уоршелла.

Основною особливістю алгоритму Оттермана є те, що на кожному кроці перерахунку матриці враховується можливість проходження шляхів через одну додаткову проміжну вершину, а обчислення припиняється тоді, коли матриця відстаней перестає змінюватися.

Для застосування алгоритму Оттермана граф задається матрицею ваг (матрицею суміжності з вагами), так само, як і для алгоритму Шимбелла.

Нехай D — поточна матриця найкоротших відстаней. На кожній ітерації виконується оновлення за правилом:

$$D_{ij} = \min\left(D_{ij},\ D_{ik} + D_{kj}\right),$$

де k — фіксована проміжна вершина.

Процес повторюється для всіх вершин графа, а обчислення завершуються, коли на черговій ітерації жодне значення матриці не змінюється. Таким чином досягається стабільний стан, що відповідає найкоротшим шляхам між усіма парами вершин.

У найгіршому випадку обчислювальна складність становить $O(V^3)$, де V — кількість вершин графа. За складністю алгоритм Оттермана співставний з алгоритмом Флойда-Уоршелла, однак реалізує ідею поступового збіжного уточнення матриці відстаней.

Приклад реалізації алгоритм Оттермана у вигляді функції на Python:

```
def otterman_algorithm(D):
    n = len(D)
    dist = [row[:] for row in D]
    changed = True

    while changed:
        changed = False
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    if dist[i][k] + dist[k][j] < dist[i][j]:
                        dist[i][j] = dist[i][k] + dist[k][j]
                        changed = True
    return dist
```

У результаті функція повертає матрицю найкоротших відстаней між усіма парами вершин графа.

Якщо порівнювати з іншими алгоритмами, то коли у алгоритмі Шимбелла послідовно враховуються шляхи зі зростаючою кількістю ребер, а у алгоритмі Флойда-Уоршелла використовується чітко структурований перебір проміжних вершин, в алгоритмі Оттермана виконується ітеративне уточнення до стабілізації результату.

Алгоритм Оттермана доцільно розглядати у навчальному курсі як приклад матричного підходу до задачі пошуку найкоротших шляхів. Він дозволяє краще зрозуміти ідею динамічного програмування та взаємозв'язок між різними алгоритмами аналізу графів. У практичних задачах зазвичай застосовують більш оптимізовані алгоритми.

Контрольні запитання

1. В яких галузях вирішення задачі пошуку найкоротшого шляху буде дуже актуальною?
2. Чим відрізняється поняття «найкоротший шлях» у незваженому графі від аналогічного поняття у зваженому графі?
3. Які варіанти вирішення має задача пошуку найкоротших шляхів у графі?
4. Розкажіть про алгоритм Флойда-Уоршелла. Який тип задачі пошуку вирішує цей алгоритм? З якими типами графів він може працювати? Чи має він якісь обмеження? Спробуйте застосувати алгоритм Флойда-Уоршелла для графів, зображених на рисунку 3.2.
5. Який різновид задачі пошуку найкоротших шляхів вирішує алгоритм Форда-Беллмана? В чому полягає його ключова особливість? Поясніть механізм роботи алгоритму, зокрема принцип релаксації ребер. Спробуйте застосувати цей алгоритм для пошуку найкоротших шляхів в графах, зображених на рисунку 3.2.
6. Опишіть алгоритм Дейкстри. У чому полягає його особливість? За якої умови використання алгоритму Дейкстри є неможливим або дасть некоректний результат? Спробуйте застосувати цей алгоритм для знаходження найкоротших шляхів у графах, зображених на рис. 3.2.
7. Як використання пріоритетної черги (бінарної купи) змінює обчислювальну складність алгоритму Дейкстри? Для яких графів ця оптимізація є найбільш відчутною?
8. У чому полягає основна ідея алгоритму Шимбелла? Поясніть, які дії виконуються у кроках цього алгоритму. Використайте запропоновану реалізацію

алгоритму на мові Python для знаходження найкоротших шляхів в графах на рисунку 3.2.

9. До якої групи відноситься алгоритм Оттермана? Поясніть принцип дії цього алгоритму. Який критерій зупинки обчислень використовується в алгоритмі Оттермана і чим він концептуально відрізняється від способу зупинки в інших алгоритмах? В якій формі найбільш зручно подавати граф для роботи цього алгоритму? Скористайтесь наданою реалізацією, щоб знайти найкоротші шляхи в графах, зображених на рисунку 3.2.

10. Наведіть оцінки складності усіх алгоритмів, описаних у розділі. Надайте порівняльну характеристику, сформулюйте висновки, який алгоритм є більш оптимальним і для яких випадків.

6 ІЄРАРХІЧНІ ДЕРЕВОПОДІБНІ СТРУКТУРИ

Ієрархічні деревоподібні структури є одними з найважливіших об'єктів вивчення в теорії графів і широко застосовуються в інформатиці для організації та впорядкування даних. Вони лежать в основі багатьох класичних способів подання інформації, зокрема файлових систем, синтаксичних дерев, діаграм рішень, індексних структур баз даних, мережевих структур та алгоритмів пошуку.

З математичної точки зору такі структури описуються спеціальним класом графів, які називають *деревами*.

Дерево — це зв'язний неорієнтований граф, який не містить циклів.

Для того, щоб граф $G = (V, E)$ з кількістю вершин $|V| = n$ та кількістю ребер $|E| = m$ вважався деревом, він має відповідати наступним критеріям (ознакам):

1) *зв'язність* — між будь-якими двома вершинами графа існує шлях; водночас, ця зв'язність має бути *мінімальною*: видалення будь-якого ребра робить граф незв'язним;

2) *відсутність циклів* — у графі немає замкнених маршрутів;

3) *єдиність шляху* — між будь-якою парою вершин існує один і тільки один простий шлях;

4) *співвідношення вершин і ребер* — кількість ребер завжди на одиницю менша за кількість вершин: $m = n - 1$.

Дерева являють собою ієрархічні структури, у яких одна з вершин виділяється як *корінь*, вершини нижчих рівнів (більш віддалених від кореня відносно вершини, що розглядається) називають *нащадками*, а вершини вищих рівнів (менш віддалених від корня) — *предками*. Вершини, що мають хоч одного нащадка, називають *внутрішніми*, а ті, що не мають нащадків, називають *листковими*. Висота дерева визначається найдовшим шляхом від кореня до листка. Часто безпосередній вузол-предок називають *батьком* (*батьківським вузлом*), а безпосередніх нащадків — в*узлами-дітьми* або *дочірніми вузлами*. Кожна вершина (окрім кореня) має рівно одного *батька*.

Приклади графів, що являють собою дерева, зображено на рисунку 6.1.

Висота дерева T_1 визначається переходами від вершини 0 до вершини 17 (або 16) і складає 4, висота дерева T_2 складає 2, вона є однаковою для будь-якої листкової вершини цього графа.

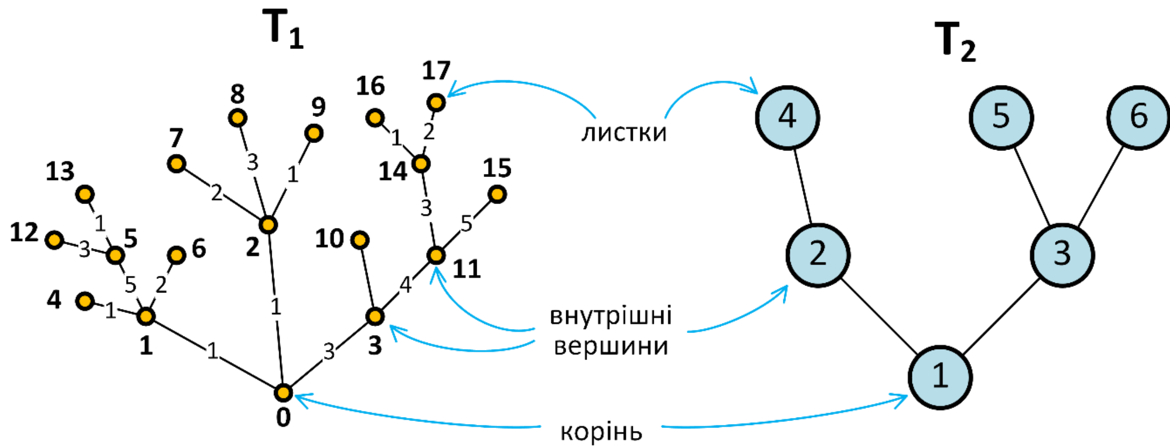

Рисунок 6.1 – Приклади дерев

У комп'ютерних науках дерева класифікують за структурою та способом організації зв'язків:

– *кореневі дерева* — дерева, в яких виділено одну особливу вершину — корінь, від неї відраховується рівень (глибина) усіх інших вузлів;

– *впорядковані дерева* — дерева, у яких порядок слідування нащадкових вузлів для кожного вузла є фіксованим;

– *бінарні* (*двійкові*) *дерева* — дерева, у яких кожен вузол має не більше двох нащадків (лівого та правого).

– *збалансовані дерева* — дерева, у яких висота лівого та правого піддерев для кожного вузла відрізняється не більше ніж на одиницю (наприклад, AVL-дерева).

У класичній теорії графів дерево визначається як неорієнтований зв’язний ациклічний граф. Проте в інформатиці та прикладних задачах дуже часто використовують орієнтовані кореневі дерева, у яких ребра мають напрямок і відображають ієрархічні відношення.

Орієнтоване кореневе дерево — це дерево, у якому є чітко виділений корінь, а кожне ребро має напрямок: або від предка до нащадка (найпоширеніший варіант), або від нащадка до предка.

Таку структуру коректніше називати деревоподібним орієнтованим ациклічним графом (Directed Acyclic Graph — DAG), але на практиці її часто просто називають орієнтованим деревом. Кожне орієнтоване дерево є DAG (орієнтованим ациклічним графом), але не кожен DAG є деревом — у DAG можуть існувати кілька коренів, вершини з кількома батьками, а також кілька різних шляхів між вершинами (рис. 6.2).

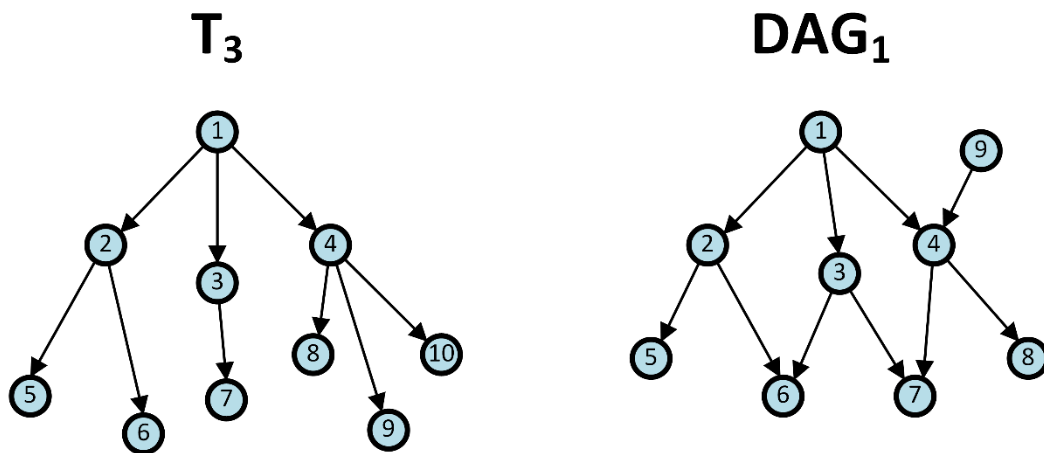

Рисунок 6.2 – Різниця між деревом та DAG

Деревоподібні структури широко застосовуються в прикладних задачах для організації даних у сучасних системах:

Файлові системи — один з найбільш наглядних прикладів використання кореневого дерева. Ієрархічна структура папок та файлів на диску виглядає як типове дерево: кореневий каталог файлової системи відповідає кореню дерева, каталоги — внутрішнім вершинам, а файли — листковим вершинам. Шлях до файлу є унікальним шляхом у дереві.

Структуровані документи, наприклад сторінки, оформлені за допомогою мов розмітки HTML/XML. Такі документи мають чітку структуру, стандартизовані теги (у випадку HTML), теги можуть бути вкладені один в одного, можуть повторюватись, як на одному рівні вкладеності, так і рекурсивно.

Саме така організація дозволяє ефективно виконувати аналіз та обхід структури документа при відображенні сторінки у браузері.

Бази даних. Для організації індексів і забезпечення швидкого доступу до даних у них широко застосовуються бінарні дерева (B-дерева та B+-дерева). Ці структури дозволяють зберігати дані у впорядкованому вигляді та забезпечують логарифмічну складність операцій пошуку, вставки та видалення навіть для дуже великих обсягів інформації.

Алгоритми стиснення. Класичним прикладом є дерево Хаффмана, яке використовується для побудови оптимальних префіксних кодів символів на основі частоти їх появи: символи, які трапляються частіше, отримають коротший код, тоді як рідкісні символи отримують довший код. Це дозволяє мінімізувати середню довжину коду та зменшити обсяг даних без втрати інформації.

Теорія прийняття рішень. Дерева використовуються у машинному навчанні для класифікації та прогнозування. Вузли дерева відповідають перевіркам умов або ознак, а гілки — можливим результатам цих перевірок, що дозволяє наочно моделювати процес ухвалення рішень.

Контрольні запитання

1. Сформулюйте математичне визначення дерева. Які ключові властивості графа роблять його деревом? Надайте перелік критеріїв, за якими можна визначити, що заданий граф є деревом.

2. Згадайте загальну формулу зв'язку між кількістю вершин та ребер. Якщо дерево має 100 вершин, скільки ребер воно повинно містити?

3. Поясніть різницю між корінням, внутрішніми вершинами та листками дерева. Хто такий «предок», а хто — «нащадок»?

4. Як визначається висота дерева? Що є точкою відліку для вимірювання глибини вузлів?

5. Чим бінарне дерево відрізняється від звичайного впорядкованого дерева? Яка умова має виконуватися, щоб дерево вважалося збалансованим?

6. У чому полягає принципова відмінність між орієнтованим деревом та загальним орієнтованим ациклічним графом (DAG)? Чи може у дерева бути кілька «батьків» в одного вузла?

7. Чому файлову систему вважають класичним прикладом дерева? Що в цій структурі відповідає листкам, а що — внутрішнім вузлам?

8. Яку роль відіграють дерева в алгоритмі Хаффмана? За яким принципом символи отримують коротші або довші коди?

9. Які типи дерев (наприклад, B-дерева) використовуються для побудови індексів у базах даних і яку перевагу вони надають при пошуку інформації?

7 ПОШУК МІНІМАЛЬНОГО КІСТЯКОВОГО ДЕРЕВА

При роботі з пов'язаними неорієнтованими графами часто виникає потреба знайти такий підграф, який охоплював би всі вузли графа з мінімальними «витратами»: мінімальною кількістю існуючих ребер, або таким чином, щоб сума ваг усіх обраних ребер була мінімальною. Такий підграф, зазвичай, має вигляд дерева, і таке дерево називають *кістяком* (або *каркасом*).

Підграф T неорієнтованого графа $G = (V, E)$ називається *кістяковим* або *каркасним деревом* (*spanning tree*), якщо він задовольняє двом умовам:

1) підграф T є деревом (зв'язним графом без циклів);

2) множина вершин підграфа T збігається з множиною вершин V вихідного графа G.

Якщо кожному ребру графа $e \in E$ поставлено у відповідність певне число $w(e)$ — вага ребра, то вагою кістякового дерева називають суму ваг усіх його ребер:

$$W(T) = \sum_{e \in T} w(e)$$

Приклад кістякового дерева для довільного графа G_6 зображений на рисунку 7.1.

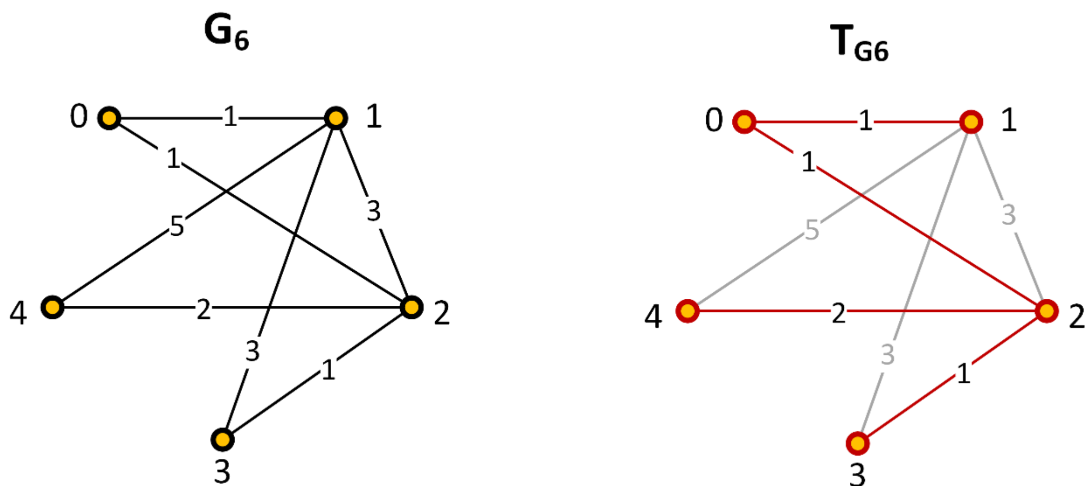

Рисунок 7.1 – Граф G_6 та його кістякове дерево T_{G6}.

Кістякове дерево має кілька ключових властивостей, які відрізняють його від довільних підграфів:

– між будь-якими двома вершинами кістяка існує рівно один простий шлях;

– якщо граф має n вершин, то будь-яке його кістякове дерево містить рівно $n - 1$ ребер;

– видалення будь-якого ребра з кістяка робить його незв’язним;

– додавання будь-якого ребра (що належить вихідному графу, але не кістяку) до кістякового дерева обов’язково утворює рівно один цикл.

Особливий інтерес становить задача **пошуку мінімального кістякового дерева** (Minimum Spanning Tree — MST) — такого кістяка T, вага якого $W(T)$ є мінімальною серед усіх можливих кістякових дерев даного графа, тобто такого, для якого сума ваг ребер є найменшою.

Важливо відмітити, якщо ваги всіх ребер графа різні, то мінімальне кістякове дерево є унікальним. Якщо ж є ребра з однаковими вагами, мінімальних кістяків може бути декілька, але їхня сумарна вага завжди буде однаковою.

Задача пошуку мінімального кістякового дерева має велике практичне значення. Вона природно виникає в задачах оптимального проєктування певної інфраструктури, де необхідно поєднати множину об'єктів комунікаційними лініями з мінімальною вартістю:

– створення комунікацій — побудова доріг між містами, прокладання ліній електропередачі, трубопроводів або телефонних мереж, де вершини — це об'єкти, а ваги ребер — вартість будівництва чи відстань;

– оптимізація комп’ютерних та телекомунікаційних мереж, зокрема, наприклад, протоколи STP (Spanning Tree Protocol) використовуються для усунення петель у локальних мережах Ethernet;

– кластеризація даних — пошук мінімального кістякового дерева використовується в аналізі даних для виявлення груп схожих об'єктів;

– наближені алгоритми — пошук MST є допоміжним етапом у розв’язанні складніших задач, наприклад, задачі комівояжера.

Алгоритм Крускала є одним із класичних, так званих «жадібних» алгоритмів для пошуку мінімального кістякового дерева в неорієнтованому зваженому графі. Він був вперше опублікований Джозефом Крускалом у 1956 році. Його стратегія полягає в тому, щоб на кожному кроці обирати ребро з найменшою вагою серед тих, що ще не були розглянуті, за умови, що воно не утворює цикл із вже обраними ребрами.

Алгоритм працює з ребрами графа, поступово об'єднуючи вершини в компоненти зв'язності. На кожному кроці обирається найлегше ребро, яке з'єднує дві різні компоненти. Процес триває доти, доки не буде побудовано кістякове дерево, що містить $|V| - 1$ ребер.

Цей алгоритм особливо ефективний для розріджених графів (де кількість ребер не набагато більша за кількість вершин), оскільки його складність значною мірою залежить від швидкості сортування ребер.

Алгоритм Крускала є особливо ефективним для розріджених графів та має просту реалізацію у поєднанні зі структурою даних, яку називають *системою непересічних множин*.

Система непересічних множин (Disjoint Set Union — DSU) — це структура даних, призначена для підтримки набору множин, які не перетинаються між собою. Вона дозволяє ефективно визначати, до якої множини належить певний елемент, а також об'єднувати дві множини в одну.

При роботі з графом вона дозволяє перевірити, чи утворює додавання ребра цикл, і таким чином підтримувати інформацію про компоненти зв'язності графа на поточному етапі побудови кістякового дерева.

Основна ідея DSU полягає в тому, що кожна вершина належить певній множині, яка відповідає компоненті зв'язності. На початку роботи алгоритму кожна вершина утворює окрему множину. Під час розгляду чергового ребра перевіряється, чи належать його кінці до різних множин. Якщо так — ребро не створює цикл, множини об'єднуються, а ребро додається до кістякового дерева. Якщо ж обидві вершини вже належать до однієї множини, додавання ребра призведе до циклу, тому таке ребро відкидається.

Основними операціями системи непересічних множин є:

– створення нової множини, що містить один елемент;

– визначення представника (кореня) множини, до якої належить елемент;

– об'єднання двох множин в одну.

Зазвичай кожна множина реалізується у вигляді дерева, де кожен елемент містить посилання на свого батька. Кореневий елемент дерева є представником відповідної множини.

Для підвищення ефективності роботи DSU використовують дві оптимізації:

1) стиснення шляху — під час виконання операції визначення представника множини усі вершини на шляху до кореня починають безпосередньо посилатися на нього;

2) об'єднання за рангом або розміром — корінь меншого дерева приєднується до кореня більшого, що зменшує висоту дерева.

Завдяки цим оптимізаціям амортизована складність основних операцій DSU є близькою до сталої та оцінюється як $O(\alpha(n))$, де $\alpha(n)$ — обернена функція Акермана, яка зростає надзвичайно повільно і на практиці вважається майже константною.

Повертаючись до алгоритму Крускала розглянемо його основні кроки:

1) ініціалізувати порожню множину ребер мінімального кістякового дерева;

2) відсортувати всі ребра графа за зростанням їх ваг;

3) послідовно переглядати відсортовані ребра:

– якщо ребро з'єднує дві різні компоненти зв'язності, додати його до кістякового дерева;

– якщо додавання ребра призводить до утворення циклу, пропустити його;

4) завершити роботу алгоритму, коли кількість вибраних ребер дорівнює $|V| - 1$.

У результаті формується мінімальне кістякове дерево з найменшою можливою сумарною вагою.

Часова складність алгоритму Крускала визначається сортуванням ребер та операціями об’єднання множин і зазвичай становить $O(E \log E)$, де E — кількість ребер графа.

Нижче наведено приклад реалізації алгоритму Крускала для неорієнтованого зваженого графа мовою Python. Для спрощення використовується базова реалізація системи непересічних множин.

```
# структура даних - система непересічних множин (Disjoint Set Union, DSU)
# використовується для роботи алгоритму Крускала
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))

    def find(self, v):
        if self.parent[v] != v:
            self.parent[v] = self.find(self.parent[v])
        return self.parent[v]

    def union(self, a, b):
        a = self.find(a)
        b = self.find(b)
        if a != b:
            self.parent[b] = a
            return True
        return False

# реалізація алгоритму Крускала
# vertices_count: кількість вершин графа
# edges: список ребер у форматі кортежів (вершина1, вершина2, вага)
def kruskal(vertices_count, edges):
    # сортування ребер за вагою
    edges_sorted = sorted(edges, key=lambda x: x[2])
    # створення DSU для заданого графа
    dsu = DSU(vertices_count)
    mst = []
    total_weight = 0

    # наповнення списку ребер кістякового дерева
    for u, v, weight in edges_sorted:
        if dsu.union(u, v):
            mst.append((u, v, weight))
            total_weight += weight

    return mst, total_weight
```

У наведеному прикладі граф задається списком зважених ребер. Алгоритм Крускала формує мінімальне кістякове дерево, додаючи ребра з мінімальною вагою та уникаючи утворення циклів за допомогою структури DSU. В кінці функція повертає список ребер, що формують мінімальний кістяк графа, а також загальну вагу знайденого кістяка.

Розглянемо роботу цієї реалізації алгоритму на прикладі графа, зображеного на рисунку 7.1.

Опис цього графу у вигляді списку ребер з вагами виглядає наступним чином:

```
# граф у форматі списку ребер
# кожне ребро задається кортежем (вершина1, вершина2, вага)
g6_edges = [
    (0, 1, 1),
    (0, 2, 1),
    (1, 2, 3),
    (1, 3, 3),
    (1, 4, 5),
    (2, 3, 1),
    (2, 4, 2)
]
```

Код, в якому буде виконано пошук та вивід результату:

```
mst, weight = kruskal(5, g6_edges)

print("Minimum spanning tree edges:")
for u, v, w in mst:
    print(f"{u} -- {v}, weight = {w}")

print("Total weight:", weight)
```

Результати виводу:

```
Minimum spanning tree edges:
0 -- 1, weight = 1
0 -- 2, weight = 1
2 -- 3, weight = 1
2 -- 4, weight = 2
Total weight: 5
```

Як бачимо, результати виводу описують кістяк графа, зображений на рисунку 7.1.

Алгоритм Прима є ще одним «жадібним» алгоритмом пошуку мінімального кістякового дерева в неорієнтованому зваженому графі. На відміну від алгоритму Крускала, який працює з множиною ребер, алгоритм Прима поступово розширює кістякове дерево, починаючи з довільної вершини та на кожному кроці додаючи одне ребро мінімальної ваги, що з'єднує вже включену до дерева частину графа з рештою вершин.

Алгоритм Прима ґрунтується на ідеї поетапного нарощування зв'язної підмножини вершин. На кожному кроці обирається ребро з найменшою вагою серед усіх ребер, що мають рівно одну вершину в поточному кістяковому дереві. Додавання такого ребра не призводить до утворення циклів і гарантує мінімальність сумарної ваги результату.

Алгоритм Прима коректно працює для зв'язних неорієнтованих зважених графів. Якщо граф є незв'язним, алгоритм побудує мінімальне кістякове дерево лише для тієї компоненти зв'язності, з якої було розпочато роботу.

Основні кроки алгоритму Прима:

1) обрати довільну початкову вершину та включити її до кістякового дерева;

2) ініціалізувати множину відвіданих вершин;

3) серед усіх ребер, що з'єднують відвідані вершини з невідвіданими, обрати ребро з мінімальною вагою;

4) додати вибране ребро та нову вершину до кістякового дерева;

5) повторювати кроки 3–4, доки до кістякового дерева не будуть включені всі вершини графа.

У результаті буде побудовано мінімальне кістякове дерево, що містить $|V| - 1$ ребер.

Часова складність алгоритму Прима залежить від способу його реалізації:

– $O(V^2)$ — при використанні матриці суміжності;

– $O(E \log V)$ — при використанні списків суміжності та черги з пріоритетами.

Нижче наведено приклад реалізації алгоритму Прима мовою Python з використанням черги з пріоритетами.

```
import heapq

# функція знаходження MST - алгоритм Прима
# graph: список суміжності з вагами, словник вигляду:
#   {вершина: [(сусід, вага), ...]}
# start: початкова вершина - корінь дерева
def prim(graph, start):
    visited = set()
    min_heap = [(0, start, None)]  # (вага, вершина, батько)
    mst = []
    total_weight = 0

    while min_heap and len(visited) < len(graph):
        weight, v, parent = heapq.heappop(min_heap)

        if v in visited:
            continue

        visited.add(v)
        total_weight += weight

        if parent is not None:
            mst.append((parent, v, weight))

        for to, w in graph[v]:
            if to not in visited:
                heapq.heappush(min_heap, (w, to, v))

    return mst, total_weight
```

Результатом роботи алгоритму Прима є множина ребер, що утворюють мінімальне кістякове дерево, та сумарна вага цих ребер. Кожне ребро задається парою вершин і відповідною вагою, що дозволяє однозначно відновити структуру побудованого кістякового дерева.

Розглянемо роботу цієї реалізації алгоритму на прикладі графа, зображеного на рисунку 7.1.

Опис цього графу у вигляді списку суміжності з вагами виглядає наступним чином:

```
# граф у форматі списку суміжності
# формат: {вершина: [(сусід, вага), ...]}
```

```
g6_ajlist = {
    0: [(1, 1), (2, 1)],
    1: [(0, 1), (2, 3), (3, 3), (4, 5)],
    2: [(0, 1), (1, 3), (3, 1), (4, 2)],
    3: [(1, 3), (2, 1)],
    4: [(1, 5), (2, 2)]
}
```

Код, в якому буде виконано пошук та вивід результату:

```
mst, weight = prim(g6_ajlist, 0)

print("Edges of MST:")
for u, v, w in mst:
    print(f"{u} -- {v}, weight = {w}")
print("Total MST weight:", weight)
```

Результати виводу:

```
Edges of MST:
0 -- 1, weight = 1
0 -- 2, weight = 1
2 -- 3, weight = 1
2 -- 4, weight = 2
Total MST weight: 5
```

Алгоритми Крускала та Прима є базовими «жадібними» методами побудови мінімального кістякового дерева. Перший алгоритм орієнтований на обробку ребер і використовує систему непересічних множин, тоді як другий нарощує дерево від початкової вершини. Обидва алгоритми забезпечують мінімальну сумарну вагу кістякового дерева та широко застосовуються в практичних задачах оптимізації мереж.

Контрольні запитання

1. Що називається кістяковим (каркасним) деревом неорієнтованого графа і які дві основні умови воно повинно задовольняти?

2. Які ключові властивості має кістякове дерево та чим воно відрізняється від довільного підграфа?

3. Як визначається вага кістякового дерева та що означає поняття мінімального кістякового дерева (MST)?

4. За яких умов мінімальне кістякове дерево є унікальним, а коли можливе існування кількох мінімальних кістяків?

5. У яких практичних задачах виникає потреба пошуку мінімального кістякового дерева та чому ця задача є важливою?

6. У чому полягає основна ідея алгоритму Крускала та чому його відносять до «жадібних» алгоритмів? Розгляньте граф, поданий на рисунку 7.2. Скористайтесь наданою у розділі реалізацією алгоритму Крускала для визначення мінімального кістякового дерева цього графу.

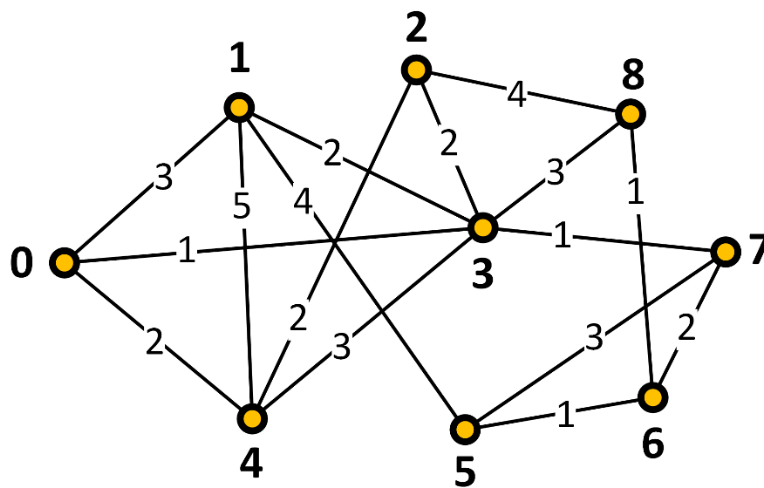

Рисунок 7.2 – Граф для індивідуальних завдань

7. Яку роль у алгоритмі Крускала відіграє система непересічних множин (DSU) і які її базові операції?

8. Опишіть алгоритм Прима. У чому полягає принципова відмінність між алгоритмами Крускала та Прима з точки зору побудови мінімального кістякового дерева? Скористайтесь наданою у розділі реалізацією алгоритму Прима для визначення мінімального кістякового дерева графу, зображеного на рисунку 7.2.

9. Від чого залежить часова складність алгоритмів Крускала і Прима та для яких типів графів кожен із них є більш ефективним?

8 МЕТОДИ ТОПОЛОГІЧНОГО СОРТУВАННЯ

Топологічне сортування — це процес впорядкування вершин орієнтованого ациклічного графа (Directed Acyclic Graph — DAG) таким чином, що для кожного орієнтованого ребра (u, v) вершина u передує вершині v у впорядкованій послідовності. Топологічне сортування застосовується для аналізу залежностей між об'єктами, зокрема під час планування задач, компіляції програм, аналізу передумов курсів та обробки графів залежностей.

Топологічне сортування можливе лише для ациклічних орієнтованих графів. Наявність циклу означає неможливість побудови коректного порядку виконання.

Існує два базові алгоритмічні підходи до топологічного сортування:

– алгоритм на основі пошуку в глибину (DFS);

– алгоритми, засновані на аналізі вхідних степенів вершин.

Кожен з підходів має однакову асимптотичну складність, але відрізняється способом реалізації та інтерпретації результатів.

Алгоритм топологічного сортування на основі DFS базується на виконанні пошуку в глибину для кожної неперевіреної вершини графа. Вершина додається до результатного списку після завершення обходу всіх її нащадків. Після завершення DFS для всіх вершин отриманий список інвертується, утворюючи топологічний порядок.

Основні кроки алгоритму:

1) позначити всі вершини як неперевірені;

2) для кожної вершини виконати DFS, якщо вона ще не була відвідана;

3) після завершення рекурсивного виклику для вершини додати її до списку результату;

4) інвертувати список результату.

У методі топологічного сортування на основі пошуку у глибину побудова топологічного порядку відбувається безпосередньо в процесі обходу графа.

Відстеження станів вершин дозволяє виявляти цикли, а реалізація методу зазвичай ґрунтується на рекурсивному підході або використанні стека.

Обчислювальна складність цього алгоритму оцінюється як $O(|V| + |E|)$, де $|V|$ — кількість вершин, $|E|$ — кількість ребер.

Розглянемо практичну реалізацію методів топологічного сортування на прикладі графа, зображеного на рисунку 8.1.

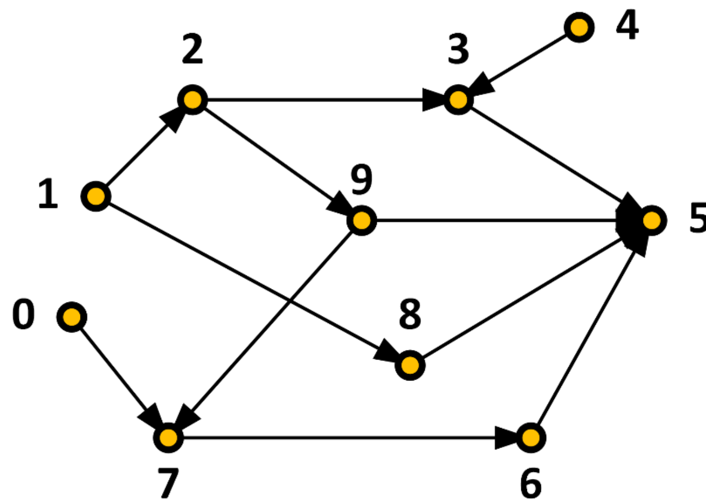

Рисунок 8.1 – Граф для дослідження роботи методів топологічного сортування

Реалізація алгоритму топологічного сортування на основі DFS мовою Python наведена нижче.

```
def dfs_tpl_sort(adj):
    memo = {}
    # Вершини, які зараз обробляються (для виявлення циклів)
    visited_in_stack = set()
    # Вершини, для яких рівень уже обчислено
    visited_fully = set()

    # виявлення циклів
    def get_level(u):
        if u in visited_in_stack:
            raise ValueError(f"ERROR: There is cycle on vertex: {u}")

        if u in memo:
            return memo[u]

        # Позначаємо вершину як таку, що в процесі обробки
        visited_in_stack.add(u)
```

```
        if not adj[u]:
            level = 0
        else:
            # Рекурсивно шукаємо рівні сусідів
            level = 1 + max(get_level(v) for v in adj[u])

        # Прибираємо зі стеку та зберігаємо результат
        visited_in_stack.remove(u)
        memo[u] = level
        return level

    # Обробляємо кожну вершину
    try:
        for node in adj:
            get_level(node)
    except ValueError as e:
        print(f"Помилка: {e}")
        return None

    # Групування результатів
    levels_dict = {}
    for node, lvl in memo.items():
        levels_dict.setdefault(lvl, []).append(node)

    return dict(sorted(levels_dict.items()))
```

Програма приймає опис графа у вигляді списку суміжності (словника зі списком суміжних вершин для кожної вершини графу) і повертає словник, в якому вершини графа відсортовані по топологічних рівнях. Програма додатково містить перевірку на наявність циклів у графі.

Опис графа, зображеного на рисунку 8.1, у вигляді списку суміжності:

```
# граф для перевірки роботи алгоритму
g8_adj = {
    0: [7], 1: [2, 8], 2: [3, 9],
    3: [5], 4: [3], 5: [], 6: [5],
    7: [6], 8: [5], 9: [5, 7]
}
```

Код програми, яка перевіряє роботу алгоритму:

```
levels = dfs_tpl_sort(g8_adj)

for lvl, verts in levels.items():
    print(f"Level {lvl}: {sorted(verts)}")
```

Результати виводу:

```
Level 0: [5]
Level 1: [3, 6, 8]
Level 2: [4, 7]
Level 3: [0, 9]
Level 4: [2]
Level 5: [1]
```

В цьому алгоритмі використовується обхід вершин графа «від кінця до початку»: вершина 5 отримує рівень 0, оскільки це єдина вершина, що не має нащадків, вершини, що ведуть у 5 (тобто, 3, 6 і 8), отримують рівень 1, вершина 7, яка веде до 6 (на рівні 1), отримує рівень 2, і так далі. Візуалізація рівнів сортування вершин графа продемонстрована на рисунку 8.2.

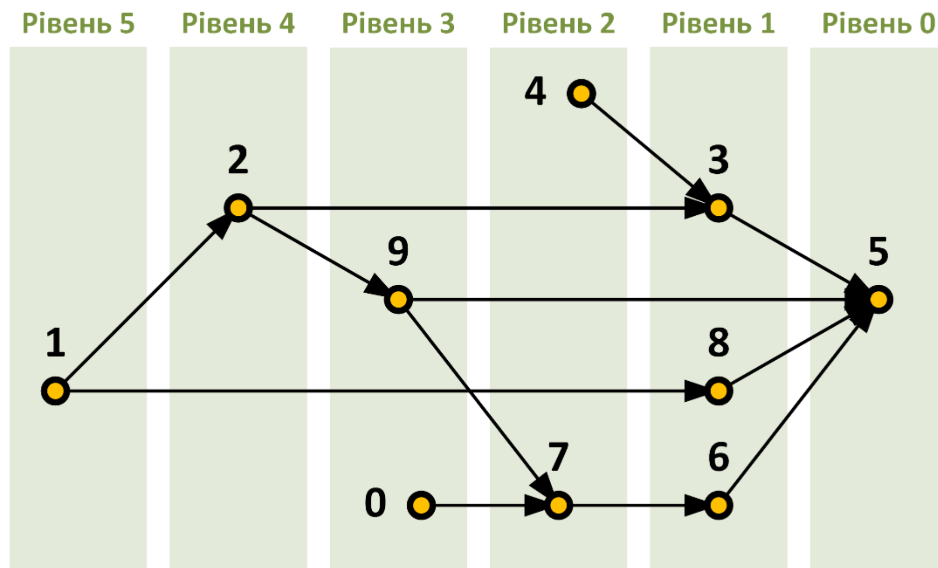

Рисунок 8.2 – Візуалізація рівнів сортування графу за допомогою алгоритму топологічного сортування на базі DFS

Серед алгоритмів для топологічного сортування орієнтованих ациклічних графів, що використовують аналіз вхідних степенів вершин, найбільш відомими є алгоритми Демукрона і Кана.

Алгоритм Кана ґрунтується на ідеї послідовного видалення вершин з нульовим вхідним степенем. Вхідний степінь вершини — це кількість ребер, що входять до неї. В процесі сортування виконується лінійне впорядкування вершин, де для кожного ребра $u \rightarrow v$ вершина u передує v у списку. Якщо після

виконання алгоритму в список сортування потрапили не всі вершини графа — у графі є цикл.

Основні кроки алгоритму:

1) обчислити вхідний степінь для кожної вершини графа;

2) додати до черги всі вершини з нульовим вхідним степенем;

3) поки черга не порожня:

– видалити вершину u з черги та додати її до списку сортування;

– зменшити вхідний степінь усіх її сусідів v_i;

– якщо вхідний степінь сусідньої вершини v_i стає нульовим — додати її до черги.

4) перевірити, чи всі вершини були включені до результату.

Серед особливостей цього методу варто відзначити, що він не використовує рекурсію, наочно демонструє залежності між вершинами та дозволяє легко визначити наявність циклів.

Обчислювальна складність алгоритму оцінюється як $O(|V| + |E|)$.

Реалізація алгоритму топологічного сортування на основі алгоритму Кана мовою Python наведена нижче. Програма приймає опис графа у вигляді списку суміжності в такому ж самому форматі, як і попередній алгоритм, але повертає результат у вигляді лінійного списку вершин, де по суті, кожна вершина має свій окремий рівень.

```
from collections import deque

def kahn_tpl_sort(adj):
    # взначення вхідних ступенів всіх вершин
    indegree = {u: 0 for u in adj}
    for u in adj:
        for v in adj[u]:
            indegree[v] = indegree.get(v, 0) + 1

    # знаходження вершини з нульовим вхідним ступенем
    queue = deque([u for u in adj if indegree[u] == 0])

    topo_order = []

    # основний процес сортування
    while queue:
```

```
        u = queue.popleft()
        topo_order.append(u)

        for v in adj[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)

    # перевірка на цикли
    if len(topo_order) != len(adj):
        raise ValueError("ERROR: There are cycles in the graph!")

    return topo_order
```

Код програми, яка перевіряє роботу алгоритму:

```
result = kahn_tpl_sort(g8_adj)
print(f"Sorted list of vertices: {result}")
```

Результати виводу:

```
Sorted list of vertices: [0, 1, 4, 2, 8, 3, 9, 7, 6, 5]
```

Графічно результати сортування представлені на рисунку 8.3.

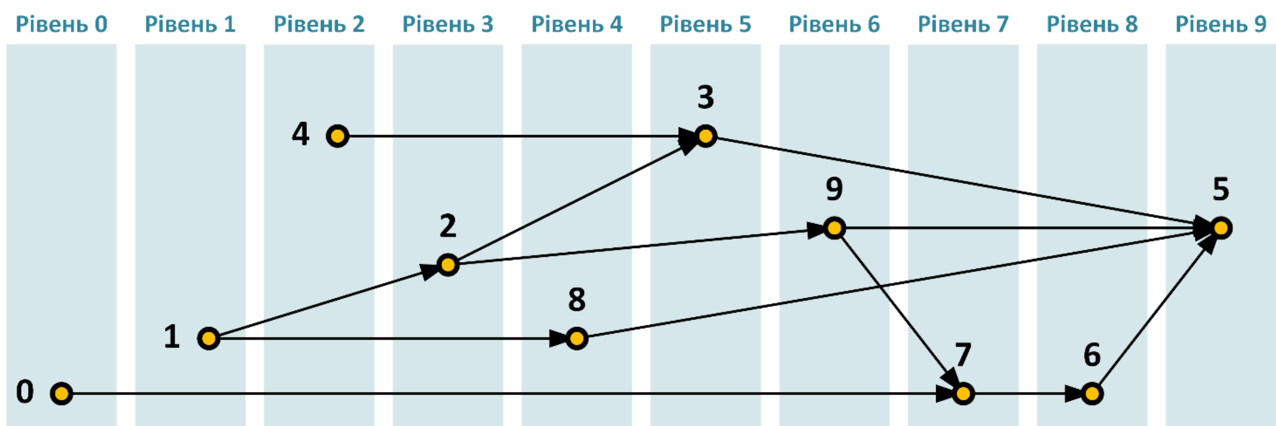

Рисунок 8.3 – Результати сортування за допомогою алгоритму Кана

Алгоритм Демукрона — це метод топологічного сортування орієнтованого ациклічного графа, що базується на поетапному виділенні рівнів вершин із нульовим вхідним степенем. На кожному кроці формується множина вершин, для яких відсутні вхідні ребра, після чого ці вершини вилучаються з графа разом із відповідними вихідними ребрами.

Алгоритм Демукрона широко застосовується для задач планування та аналізу залежностей, оскільки результатом його роботи є не лише лінійний порядок, а й шарова (рівнева) структура графа.

Основна ідея алгоритму полягає у наступному. Кожна вершина графа характеризується вхідним степенем — кількістю дуг, що входять до неї. Вершини з нульовим вхідним степенем не залежать від інших і можуть бути розміщені на першому рівні. Після їх вилучення деякі інші вершини можуть отримати нульовий вхідний степінь і бути віднесеними до наступного рівня.

Основні кроки алгоритму Демукрона:

1) для кожної вершини графа обчислити вхідний степінь;

2) сформувати перший рівень з усіх вершин, що мають нульовий вхідний степінь;

3) видалити вершини поточного рівня з графа разом з усіма їх вихідними ребрами;

4) перерахувати вхідні степені для решти вершин;

5) повторювати кроки 2-4 до тих пір, поки всі вершини не будуть оброблені.

Якщо на певному циклі обходу не існує вершин з нульовим вхідним степенем, граф містить цикл.

Результатом алгоритму Демукрона є упорядкована послідовність рівнів $L_0, L_1, \ldots, L_k$, де кожен рівень містить список вершин, що можуть бути реалізовані паралельно (наприклад, якщо граф описує набір взаємозалежних дисциплін, які вивчаються за навчальним планом, то на одному рівні будуть знаходитись дисципліни, які можна вивчати одночасно). Будь-яке лінійне злиття рівнів утворює коректний топологічний порядок.

Алгоритм Демукрона має обчислювальну складність $O(|V| + |E|)$.

Реалізація алгоритму Демукрона у вигляді функції на Python:

```
from collections import defaultdict

def demukron_tpl_sort(vertices, edges):
    # ініціалізація вхідних степенів
    indegree = {v: 0 for v in vertices}
```

```
    graph = defaultdict(list)

    for u, v in edges:
        graph[u].append(v)
        indegree[v] += 1

    levels = []
    remaining_vertices = set(vertices)

    while remaining_vertices:
        # поточний рівень — вершини з нульовим вхідним степенем
        level = [v for v in remaining_vertices if indegree[v] == 0]

        if not level:
            raise ValueError("Граф містить цикл")

        levels.append(level)

        # видалення вершин поточного рівня
        for v in level:
            remaining_vertices.remove(v)
            for neighbor in graph[v]:
                indegree[neighbor] -= 1

    return levels
```

Для роботи функції граф треба подати як два списки: список вершин та список ребер:

```
# опис графа
g8_vertices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
g8_edges = [
    (0, 7), (1, 2), (2, 3), (4, 3),
    (3, 5), (6, 5), (7, 6), (1, 8),
    (2, 9), (9, 5), (9, 7), (8, 5)
]
```

Функція поверне вкладений список, де в основному списку, що являє собою перелік сформованих рівнів містяться списки з вершинами графу, що знаходяться на відповідному рівні.

Програма, яка використовує функцію топологічного сортування за алгоритмом Демукрона наведена нижче:

```
levels = demukron_tpl_sort(g8_vertices, g8_edges)

for item, level in enumerate(levels):
    print(f"Рівень {item}: {level}")
```

Результат роботи програми:

```
Рівень 0: [0, 1, 4]
Рівень 1: [2, 8]
Рівень 2: [3, 9]
Рівень 3: [7]
Рівень 4: [6]
Рівень 5: [5]
```

Результат роботи можна візуалізувати так, як показано на рисунку 8.4.

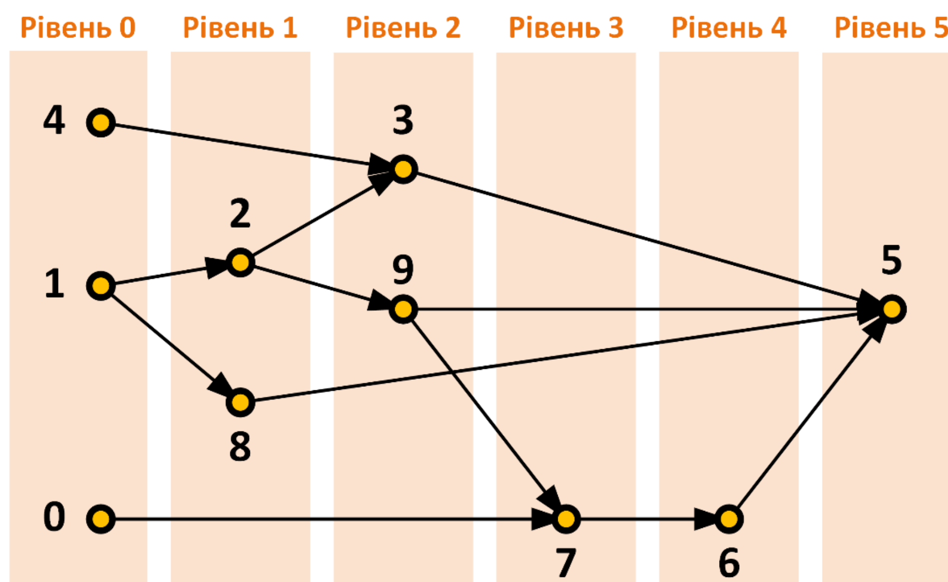

Рисунок 8.4 – Візуалізація результатів топологічного сортування за допомогою алгоритму Демукрона

Методи топологічного сортування є універсальним інструментом аналізу орієнтованих ациклічних графів. Вони широко застосовуються в різних галузях людської діяльності — від освіти, програмної інженерії та обробки даних до проєктування складних систем та виконання виробничих процесів. Застосування методів топологічного сортування дозволяє встановити коректний порядок виконання залежних дій, виявляти циклічні залежності, у розгалужених процесах та визначити можливості паралельного виконання задач.

У сфері освіти та управління проєктами топологічне сортування використовується для впорядкування об’єктів, між якими існують чітко визначені залежності. У навчальних планах це дозволяє формалізувати передумови дисциплін і визначити коректну послідовність їх вивчення, а також

виокремити рівні курсів, які можуть опановуватися паралельно. Аналогічним чином у системах керування проєктами та автоматизації бізнес-процесів завдання або операції подаються у вигляді орієнтованого графа залежностей, де топологічне сортування забезпечує допустимий порядок виконання робіт і дозволяє уникнути логічних суперечностей під час планування.

У програмній інженерії методи топологічного сортування широко застосовуються для аналізу залежностей між програмними модулями, бібліотеками, класами або функціями. Під час компіляції програмного забезпечення топологічне впорядкування дає змогу визначити коректний порядок обробки модулів з урахуванням імпортів і виключити ситуації використання ще не скомпільованих компонентів. Подібний підхід використовується в системах керування пакетами та засобах DevOps, де необхідно встановлювати або оновлювати програмні пакети у відповідності до їхніх залежностей.

У задачах аналізу даних, машинного навчання та побудови пайплайнів обробки інформації окремі етапи обчислень мають виконуватися в суворо визначеній послідовності. Топологічне сортування дозволяє формалізувати такі пайплайни, визначити коректний порядок етапів та виявити можливості для їх паралельного виконання.

Контрольні запитання

1. Що таке топологічне сортування та для яких типів графів воно є можливим?
2. Чому наявність циклів у графі унеможливлює побудову топологічного порядку?
3. У чому полягає основна ідея топологічного сортування на основі пошуку в глибину (DFS)? Назвіть основні кроки цього алгоритму. Як за допомогою DFS виявляються цикли в орієнтованому графі під час топологічного сортування?

4. У чому полягає принцип роботи алгоритму Кана та яку характеристику вершин він використовує? Який критерій дозволяє визначити наявність циклу в графі після виконання алгоритму Кана?

5. У чому полягає основна ідея алгоритму Демукрона та яке представлення результату він формує? Назвіть основні кроки цього алгоритму.

6. Чим відрізняється результат алгоритму Кана від результату алгоритму Демукрона? У чому різниця між результатами алгоритму Демукрона та топологічного сортування за допомогою пошуку в глибину (DFS)?

7. Назвіть основні сфери практичного застосування топологічного сортування та поясніть, яку задачу воно там вирішує.

8. Наведіть оцінки обчислювальної складності для усіх розглянутих у розділі алгоритмів. Чим вони відрізняються?

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Крива Н. Р., Блащак Н. І. Курс лекцій з дисципліни «Дискретна математика»: розділ «Теорія графів» / упоряд. Н. Р. Крива, Н. І. Блащак ; Тернопільський нац. техн. ун-т ім. Івана Пулюя. – Тернопіль : ТНТУ, 2023. – 40 с.

2. Іглін С. П. Теорія графів на базі MATLAB : навч. посібник / С. П. Іглін, Ю. І. Зайцев, Ю. Б. Решетняк ; Нац. техн. ун-т "Харків. політехн. ін-т". – Харків : НТМТ, 2023. – 236 с.

3. Кузьменко І. М. Теорія графів [Електронний ресурс] : навч. посібник / І. М. Кузьменко. – Київ : КПІ ім. Ігоря Сікорського, 2020. – 71 с.

4. Kubica J. Graph Algorithms the Fun Way: Powerful Algorithms Decoded, Not Oversimplified / Jeremy Kubica — No Starch Press, 2025. — 416 с. — ISBN 978-1718503861.

5. Farrelly C. M., Mutombo F. K. Modern Graph Theory Algorithms with Python: Harness the Power of Graph Algorithms and Real-World Network Applications Using Python / Colleen M. Farrelly, Franck Kalala Mutombo — Packt Publishing Limited, 2024. — 290 с. — ISBN 978-1-80512-017-9.

6. Valiente G. Algorithms on Trees and Graphs: With Python Code / Gabriel Valiente — Springer Cham, 2021 (2-е вид.). — 387 с. — ISBN 978-3-030-81885-2.

7. NetworkX : офіційний веб-сайт документації та інформації про бібліотеку для аналізу та роботи з графами у Python / веб-ресурс — Режим доступу: https://networkx.org/ (дата звернення: 13.01.2026).

8. Mermaid Live Editor: інтерактивний онлайн-редактор для створення діаграм та графів / веб-ресурс — Режим доступу: https://mermaid.live/ (дата звернення: 13.01.2026).

9. Graphviz: офіц. сайт / графічний інструментарій для візуалізації графів — Режим доступу: https://www.graphviz.org/ (дата звернення: 13.01.2026).

10. graphviz : Python-пакет для візуалізації графів з використанням Graphviz / веб-ресурс PyPI — Режим доступу: https://pypi.org/project/graphviz/ (дата звернення: 13.01.2026)

Навчальне видання

Методичні вказівки

Алгоритми та структури впорядкування даних.
Частина 1. Алгоритми для роботи з графами

для виконання самостійних робіт з дисциплін «Прикладне програмування на Python», «Мультипарадигмальні мови програмування» для студентів усіх форм навчання за спеціальностями G7 «Автоматизація, комп'ютерно-інтегровані технології та робототехніка» та F3 «Комп’ютерні науки»

Укладачі:

КАРАМАН Дмитро Григорович
ЗУЄВ Андрій Олександрович
ТАТАРІНОВА Оксана Андріївна
САЛЬНІКОВ Дмитро Валентинович

Відповідальний за випуск Зуєв А.О.

Роботу до видання рекомендував Пугановський О.В.

В авторській редакції

План 2026 р. поз .159

Гарнітура Times New Roman. Умов. друк. арк.

Видавничий центр НТУ «ХПІ».
Свідоцтво суб’єкта видавничої справи ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2.

Електронне видання