

Київський національний університет імені Тараса Шевченка
Географічний факультет
Кафедра економічної та соціальної географії

Олексій ГНАТЮК
Олена КОНОНЕНКО

**РУТНОН ДЛЯ СУСПІЛЬНИХ ГЕОГРАФІВ ТА УРБАНІСТІВ:
МЕТОДИ ТЕОРІЇ ГРАФІВ**

Навчально-методичне видання
для здобувачів освіти, що навчаються за освітньо-професійними програмами
«Урбаністика та міське планування» та «Економічна географія»
спеціальностей 106 Географія / С6 Географія та регіональні студії
першого (бакалаврського) освітнього рівня

Київ – 2026

УДК 004.438.3:911.3:711:37.091.3

Рецензенти:

Костянтин МЕЗЕНЦЕВ, доктор географічних наук, професор, завідувач кафедри економічної та соціальної географії географічного факультету, Київський національний університет імені Тараса Шевченка

Тетяна КУПАЧ, кандидат географічних наук, доцент, доцент кафедри географії України географічного факультету, Київський національний університет імені Тараса Шевченка

Рекомендовано:

Вченою радою географічного факультету Київського національного університету імені Тараса Шевченка (протокол № 13 від 30.04.2026 р.)

Гнатюк О., Кононенко О. Python для суспільних географів та урбаністів: Методи теорії графів. 2026. 42 с.

У навчально-методичному виданні наведено концепцію, код та приклади застосування програми, написаною мовою програмування Python та призначеної для визначення характеристик плоских графів, що використовуються для моделювання просторових зв'язків між елементами природних та/або суспільних територіальних систем. Також видання містить стислий конспект лекції про методи теорії графів у геопросторовому аналізі, завдання для самостійної роботи студентів та список рекомендованої літератури. Програма використовується для виконання окремих практичних робіт з навчальних дисциплін студентами освітніх програм «Урбаністика та міське планування» та «Економічна географія» ОР «Бакалавр» у Київському національному університеті імені Тараса Шевченка.

Навчально-методичне видання призначене для здобувачів вищої освіти, які навчаються за освітніми програмами за спеціальностями 106 Географія / С6 Географія та регіональні студії, а також всіх зацікавлених у використанні Python для аналізу географічної інформації.

ЗМІСТ

Передмова	4
Методи теорії графів: конспект лекції	5
Концептуалізація програми	16
Код програми	18
Приклад застосування програми	37
Завдання для самостійної роботи.	41
Список рекомендованої літератури	42

ПЕРЕДМОВА

Це навчально-методичне видання доповнює і розширює зміст попереднього, базового навчально-методичного видання на тему застосування мови програмування Python у суспільній географії та урбаністиці (Гнатюк О., Кононенко О. Python для суспільних географів: виконання практичних робіт з освітніх компонентів. Київ: Printto, 2025. 123 с.), фокусуючись на теорії графів.

Теорія графів є одним із фундаментальних інструментів сучасного просторового аналізу, який активно використовується у географії та урбаністиці. Вона дозволяє формалізувати складні територіальні системи, зокрема транспортні мережі, системи розселення, інженерну інфраструктуру та соціально-економічні зв'язки.

На відміну від класичної геометрії, яка оперує метричними величинами, теорія графів належить до так званої «якісної геометрії». Вона зосереджується не на точних вимірах відстаней, а на структурі зв'язків між об'єктами. Саме це робить її надзвичайно ефективною для аналізу просторових систем, де важливішими є відносини між елементами, ніж їхні абсолютні координати.

Ефективне навчання та практика аналізу геопросторових мереж за допомогою теорії графів потребує не лише теоретичного розуміння, але й інструментів для обчислень і моделювання. Програмування на Python відіграє дедалі важливішу роль у підготовці студентів-географів та урбаністів, особливо в контексті опанування методів теорії графів. Python є оптимальним середовищем для цього завдяки своїй простоті, гнучкості та наявності великого набору спеціалізованих бібліотек. Використання Python дозволяє студентам переходити від абстрактних понять, таких як зв'язність, центральність чи найкоротший шлях, до їх практичної реалізації. Наприклад, алгоритми на кшталт Флойда–Воршела або Дейкстри можуть бути безпосередньо реалізовані найпростішими засобами Python та застосовані до реальних даних, що суттєво поглиблює розуміння матеріалу.

Наведена та детально проаналізована у цьому виданні програма на Python буде корисною для студентів, оскільки вона, будучи досить простою з точки зору реалізації алгоритму та його розуміння здобувачами освіти, поєднує теоретичні основи теорії графів із їх практичною реалізацією. Вона дозволяє застосовувати знання до реальних даних (транспортні чи міські мережі), що підвищує прикладну цінність навчання і формує навички аналізу складних систем. Знайомство з цією програмою спростить опанування студентами складнішого професійного програмного забезпечення у сфері аналізу геопросторових мереж, а також створює основу для розробки власних подібних програм для власних аналітичних задач. Важливо, що програмування сприяє формуванню аналітичного мислення. Студенти вчаться формалізувати просторові задачі, структурувати дані та інтерпретувати отримані результати. Крім того, Python забезпечує інтеграцію з іншими інструментами просторового аналізу, зокрема геоінформаційними системами та бібліотеками для роботи з просторовими даними. Це дозволяє поєднувати аналіз графів із картографуванням та візуалізацією, що значно розширює можливості досліджень. Таким чином, дослідження графів як моделей реальних геопросторових мереж за допомогою Python не лише підвищує технічні компетенції студентів, але й робить їхню підготовку більш прикладною, сучасною та релевантною до реальних задач міського планування та регіональних досліджень.

Матеріали навчально-методичного видання можуть використовуватись при викладанні таких освітніх компонентів за освітніми програмами ОР Бакалавр на географічному факультеті Київського національного університету імені Тараса Шевченка: «Методи просторового аналізу», «Урболабораторія 1: Просторовий аналіз», «Моделювання в географії», «Міський транспорт і комунікації».

МЕТОДИ ТЕОРІЇ ГРАФІВ: КОНСПЕКТ ЛЕКЦІЇ

1. Поняття про граф. Елементи графа

Граф – це математична структура, яка складається з двох множин:

- множини вершин (V , від англійського *vertice* – вершина), що представляють об'єкти;
- множини ребер (E , від англійського *edge* – ребро), що представляють зв'язки між цими об'єктами.

У географічних дослідженнях вершинами можуть виступати міста, населені пункти, райони або вузли транспортної мережі, тоді як ребра відображають дороги, транспортні маршрути або функціональні зв'язки. Графічно граф подається у вигляді точок (вершин), з'єднаних лініями (ребрами).

Довільним чином вибрана частина вершин та ребер графа формують підграф.

Якщо пара вершин з'єднується одночасно кількома ребрами, то ці ребра називають кратними (паралельними). Якщо зв'язок має напрямок, відповідне ребро називається дугою.

Між елементами графа можуть існувати відношення інцидентності та суміжності:

- Інцидентність – це відношення між вершиною та ребром: ребро є інцидентним вершинам, які воно з'єднує.
- Суміжність – це відношення між однотипними елементами: вершини суміжні, якщо вони з'єднані ребром; ребра суміжні, якщо мають спільну вершину.

Степінь вершини (d) – це кількість ребер, які до неї інцидентні. У транспортній мережі це може інтерпретуватися як кількість напрямків, що виходять з вузла.

Окремо виділяють:

- висячі вершини (ступінь 1);
- ізольовані вершини (ступінь 0);
- петлі (ребра, що з'єднують вершину саму з собою).

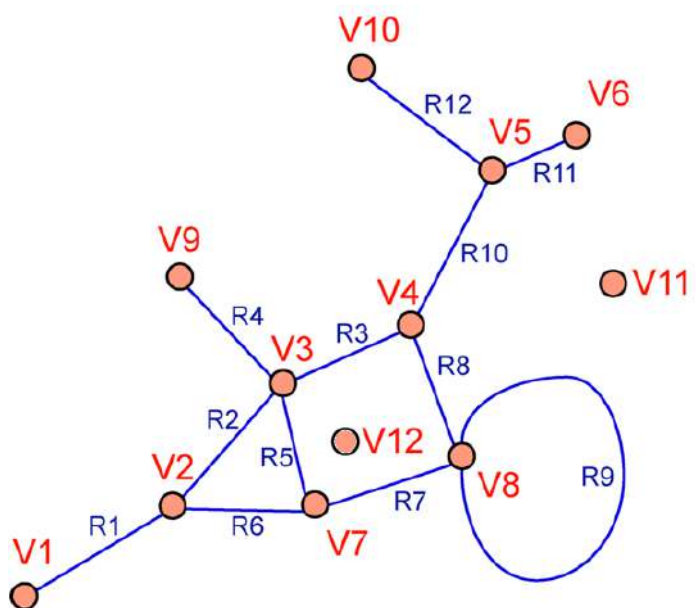
Наприклад, граф на ілюстрації справа має 12 вершин (позначених $V1$ - $V12$) та 12 ребер (позначених $R1$ - $R12$).

Приклади суміжних ребер: $R1$ та $R2$, $R1$ та $R6$. А от ребра $R1$ та $R3$ не є суміжними.

Приклади суміжних вершин: $V1$ та $V2$, $V2$ та $V7$. А от вершини $V1$ та $V3$ не є суміжними.

Вершина $V2$ інцидентна ребрам $R1$, $R2$ та $R6$.

Вершини $V11$ та $V12$ – ізольовані, $V1$? $V6$, $V9$ та $V10$ – висячі. Максимальний ступінь (4) має вершина $V3$, оскільки їй інцидентні одразу 4 ребра – $R2$, $R3$, $R4$ та $R5$.



2. Класифікація графів

Графи класифікуються за різними ознаками.

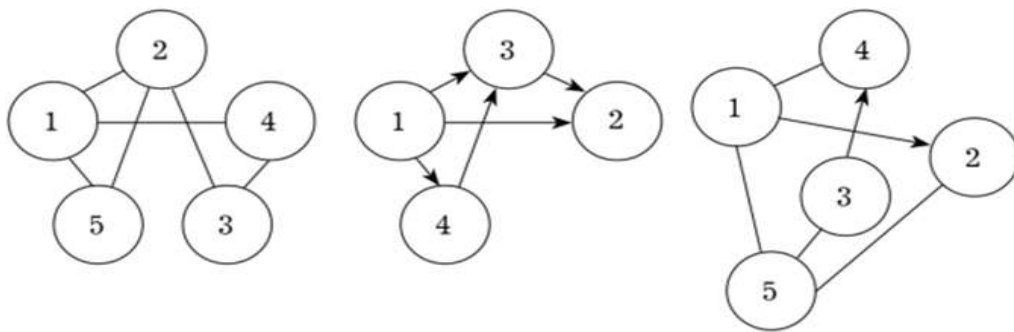
За структурою

- Простий граф – не має кратних ребер;
- Мультиграф – може містити кратні ребра;
- Нуль-граф – не містить жодного ребра.

За напрямком зв'язків

- Неорієнтований граф – всі зв'язки двосторонні;
- Орієнтований граф – всі зв'язки мають напрям;
- Змішаний граф – містить обидва типи зв'язків.

На рисунку нижче граф зліва – неорієнтований, граф посередині – орієнтований, граф справа – змішаний.

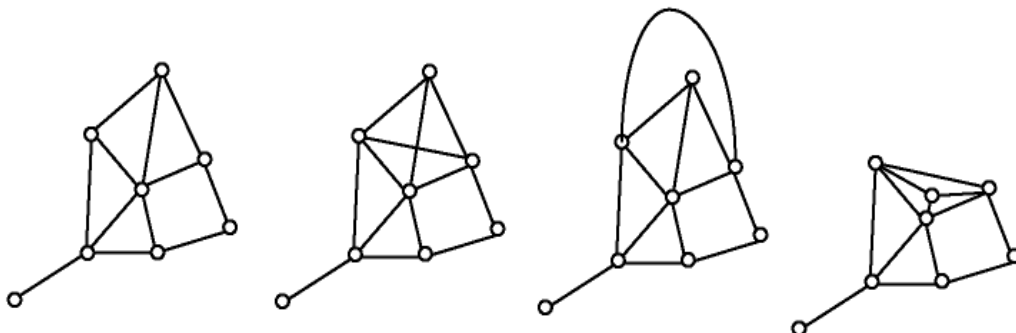


У міських системах орієнтовані та змішані графи використовуються для моделювання транспортних потоків.

За топологічними властивостями

- Повний граф – граф, кожна пара вершин з'єднана одним ребром;
- Доповнення до графа – граф, ребра якого разом з основним графом утворюють повний граф;
- Плоский граф – граф, ребра якого при зображенні на площині не перетинаються;
- Планарний граф – може бути зображений на площині таким чином, щоб ребра не перетинались.
- Грані – частини площини, обмежені ребрами плоского графа (в тому числі зовнішня грань – частина площини навколо графа).

На рисунку нижче зліва направо зображено: (1) плоский граф; (2) не плоский, але планарний граф; (3) та (4) попередній планарний граф, розгорнутий таким чином, щоб його ребра не перетинались.



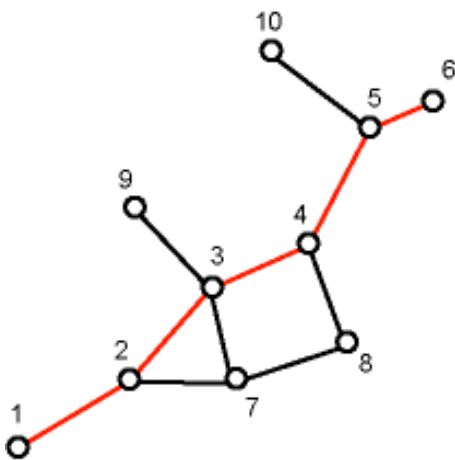
3. Шляхи, відстані та зв'язність

Шлях у графі – послідовність інцидентних елементів (вершин та ребер) графа. Різновиди шляху: маршрут, ланцюг, простий ланцюг.

- Маршрут – будь-яка послідовність інцидентних вершин і ребер.
- Ланцюг – маршрут без повторень ребер (кожне ребро може бути використане лише один раз).
- Простий ланцюг – без повторення вершин (кожна вершина може бути використана лише один раз).

Відстань між двома вершинами – довжина найкоротшого шляху між ними.

Діаметр графа – максимальна відстань між будь-якою парою вершин. У географії це може відповідати максимальному часу або відстані між елементами мережі.



Приклад маршруту: 1-2-3-4-5-4-8-7-3-9

Приклад ланцюга: 1-2-3-4-8-7-3-9

Приклад простого ланцюга: 1-2-3-4-5-10

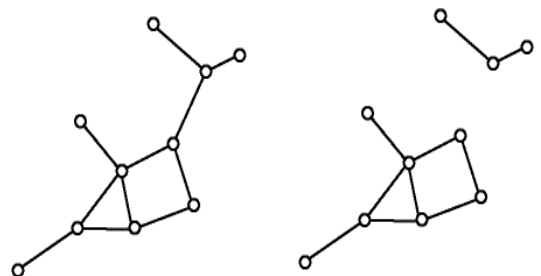
Діаметр графа $D = 5$:

1-2-3-4-5-6 або 1-2-3-4-5-10

Важливою характеристикою графа (та модельованої ним мережі) є його зв'язність. Граф є зв'язним, якщо між будь-якими двома вершинами існує шлях.

Компонента зв'язності – підграф, у якому всі вершини взаємно досяжні. Незв'язні графи складаються з кількох таких компонент.

Міст – ребро, видалення якого порушує зв'язність. На рисунку показано, як після видалення ребра-мосту зв'язний граф перетворився на незв'язний, а кількість компонент зв'язності зросла з 1 до 2.



4. Цикли та деревоподібні структури

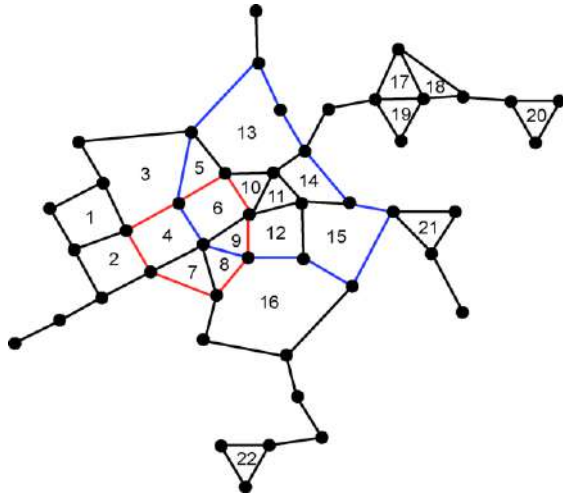
Цикл – замкнений шлях у графі. В орієнтованому графі цикли називають також контурами.

Дендрит (дерево) – зв'язний граф без циклів. Ліс – сукупність кількох дендритів (дерев).

Елементарний цикл – не включає інших циклів. Складний цикл – включає інші цикли.

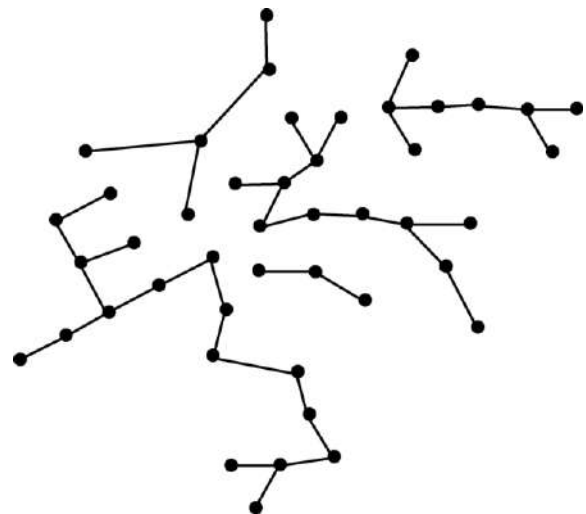
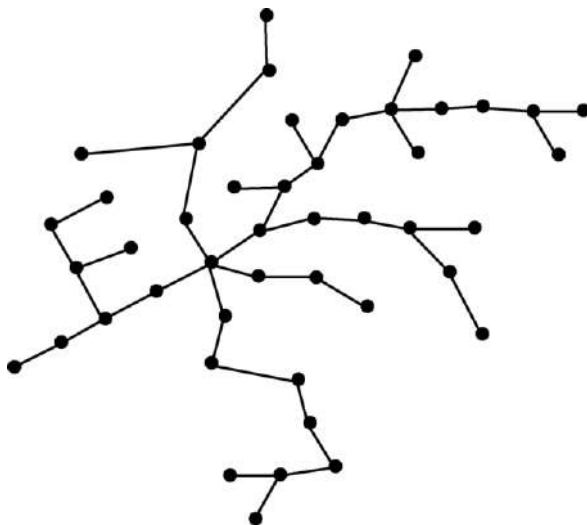
Цикломатичне число графа (C) – це кількість елементарних циклів графа.

- Цикломатичне число можна також визначити як мінімальну кількість ребер, які треба прибрати, щоб граф став дендритом
- Цикломатичне число – важливий показник складності мережі. Чим більшу кількість циклів має граф при фіксованій кількості вершин – тим складнішою є мережа, тим більше альтернативних маршрутів можна побудувати між парами вершин графа, тим стійкішою є мережа до розриву зв'язків між елементами. У транспортних системах дендритна структура характерна для слаборозвинених територій.



- Граф на рисунку має 22 елементарних цикли. Відповідно, його цикломатичне число $C = 22$.
- Шляхи, позначені синім та червоним кольорами – також цикли, але складні, оскільки включають інші цикли.

На зображенні нижче зліва – приклад графа-дендриту, справа – графа-лісу.



5. Поняття про остов графа

Остов графа – скупчення його елементарних циклів, що мають хоча б одне спільне ребро.

В одному графі можуть бути декілька остовів:

- головний остов (з найбільшою кількістю циклів)
- побічні остови (з меншою кількістю циклів)

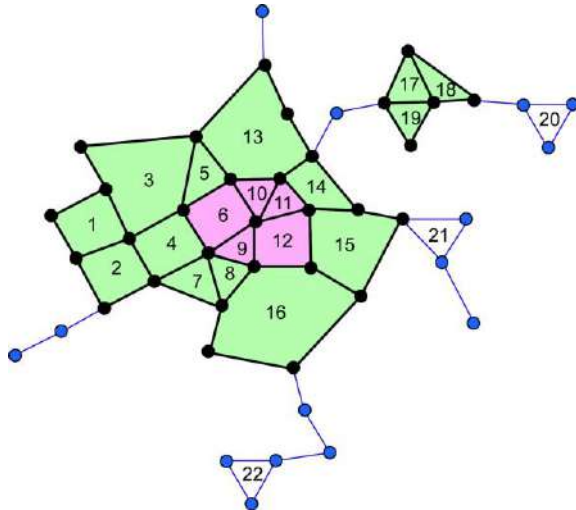
Решта циклів, що не входять до остова – позаостові.

Топологічні яруси – концентричні кільцеподібні смуги циклів остова.

Топологічні яруси нумеруються від зовнішньої межі остова до внутрішньої його частини:

- Перший топологічний ярус включає цикли, щонайменше одне ребро або одна вершина яких є спільними із зовнішньою границею остова.
- Другий топологічний ярус включає цикли, хоча б одне ребро або одна вершина яких є спільними із внутрішньою границею першого яруса...
- І так далі ... поки не скінчаться всі цикли остова

Кількість топологічних ярусів графа – клас топологічної складності мережі – важливий показник рівня її розвитку.



- Граф на рисунку має два остови
- Головний остов більший і включає 16 елементарних циклів (1-16)
- Побічний остов менший і включає 3 елементарні цикли (17-19)
- Цикли 20-22 – позаостові
- Граф має два топологічні яруси остова: перший ярус позначено зеленим кольором, другий – рожевим

6. Топологічні показники мереж

Топологічні характеристики поділяються на три групи.

I. Міри розміру – визначають масштаб мережі:

- кількість вершин (V);
- кількість ребер (E);
- кількість граней (G);
- кількість компонент зв'язності (K);
- цикломатичне число (C);
- клас топологічної складності (T)
- діаметр графа (D).

Між деякими цими величинами для плоских графів існують аналітичні співвідношення:

$$\begin{aligned} E - V + K &= C \\ C_{\max} &= 2 \cdot V - 5 \\ E_{\max} &= 2 \cdot V - 2 \end{aligned}$$

$$\begin{aligned} V - E + G &= 2 \\ G - C + K &= 2 \\ G &= C + 1 \end{aligned}$$

$$\Sigma d = 2 \cdot E$$

Варто згадати також про хроматичне число графа – мінімальна кількість кольорів, якими можна зафарбувати вершини графа так, щоб ніякі суміжні вершини не були пофарбовані в один колір («проблема політичної карти»)

- Існує так звана «теорема про 4 фарби»: максимально можливе хроматичне число плоского графа = 4. Іншими словами: для розфарбування політичної карти завжди достатньо 4 фарб.

II. Міри зв'язності – визначають, наскільки зв'язані між собою елементи мережі:

- Альфа-індекс – показує рівень насиченості графа циклами через відношення їх фактичної кількості до максимально можливої при даній кількості вершин:

$$\alpha = \frac{C}{C_{max}} = \frac{E - V + K}{2 * V - 5} \quad \alpha \in [0; 1]$$

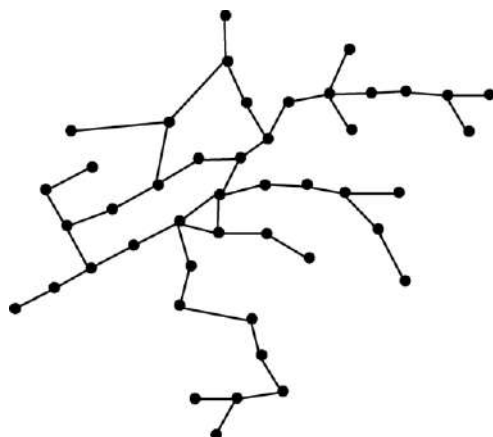
- Бета-індекс – показує рівень насиченості графа ребрами через відношення кількості ребер до кількості вершин:

$$\beta = \frac{E}{V} \quad \beta \in [0; 3)$$

- Фі-індекс – показує рівень насиченості графа ребрами через відношення їх фактичної кількості до максимально можливої при даній кількості вершин.

$$\varphi = \frac{E}{E_{max}} = \frac{E}{3 * (V - 2)} \quad \varphi \in [0; 1]$$

На рисунку нижче наведено порівняння індексів зв'язності для двох графів. Граф справа істотно сильніше насичений ребрами та циклами.



$$\alpha = 0,04 \quad \beta = 1,04 \quad \varphi = 0,36$$



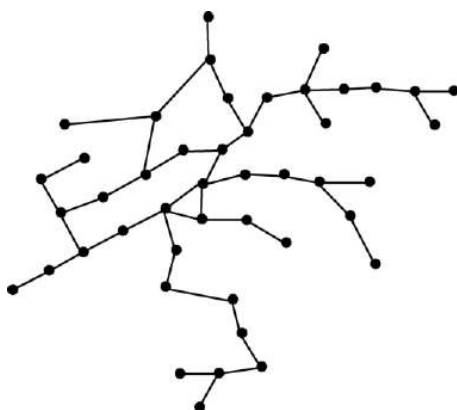
$$\alpha = 0,26 \quad \beta = 1,49 \quad \varphi = 0,51$$

III. Міри компактності – характеризують міру симетрії розвитку мережі у різних напрямках:

- Пі-індекс – характеризує компактність мережі відносно її діаметра.

$$\pi = \frac{E}{D} \quad \pi \in [0; E]$$

На рисунку нижче наведено порівняння індексів компактності для двох графів. Граф справа компактніший ніж граф зліва.



$$\pi = 3,0$$



$$\pi = 4,7$$

Показники зв'язності та компактності широко застосовуються для аналізу транспортних систем міст, регіонів та країн.

7. Матричне представлення графів

Для комп'ютерного аналізу графи подаються у вигляді матриць:

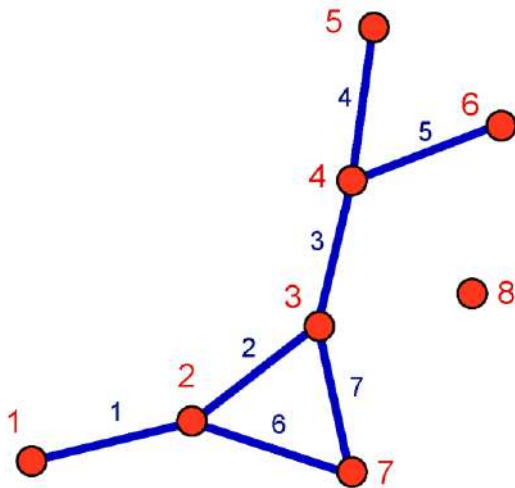
Базові матриці:

- матриця суміжності вершин
- матриця суміжності ребер
- матриця інцидентій

Ці матриці представляють граф з точністю до ізоморфізму – маючи матрицю, можна однозначно відновити вихідний граф.

Матриця суміжності вершин:

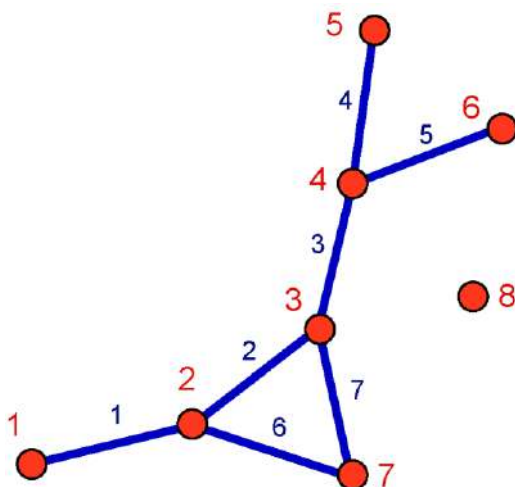
- елемент матриці дорівнює одиниці, якщо відповідні вершини суміжні



	1	2	3	4	5	6	7	8
1		1	0	0	0	0	0	0
2	1		1	0	0	0	1	0
3	0	1		1	0	0	1	0
4	0	0	1		1	1	0	0
5	0	0	0	1		0	0	0
6	0	0	0	1	0		0	0
7	0	1	1	0	0	0		0
8	0	0	0	0	0	0	0	

Матриця суміжності ребер:

- елемент матриці дорівнює одиниці, якщо відповідні ребра суміжні



	1	2	3	4	5	6	7
1		1	0	0	0	1	0
2	1		1	0	0	1	1
3	0	1		1	1	0	1
4	0	0	1		1	0	0
5	0	0	1	1		0	0
6	1	1	0	0	0		1
7	0	1	1	0	0	1	

Матричні методи дозволяють застосовувати алгоритми для обчислення найкоротших шляхів, центральності та інших характеристик.

8. Центральність вершин графа

Центральність відображає важливість вершини у структурі мережі – симетричність та оптимальність її розміщення. У міському та регіональному аналізі це дає можливість визначати центри активності та функціональну периферію, локації з найкращою та найгіршою доступністю.

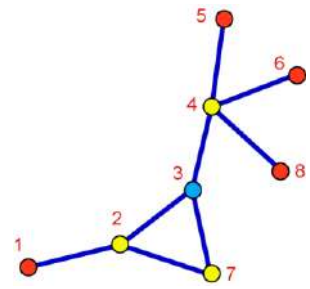
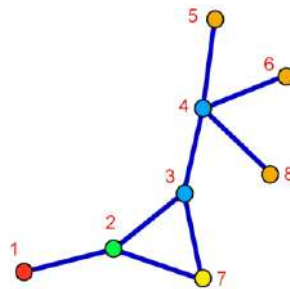
Для оцінки рівня центральності вершин графа використовуються абсолютні та відносні показники центральності.

Абсолютні показники центральності дають можливість порівнювати рівень центральності вершин в межах одного графа:

- Індекс доступності – сума відстаней від даної вершини до інших.
- Число Кеніга – максимальна відстань від даної вершини до іншої.

На рисунку нижче показано матрицю відстаней графа (зліва) та визначено індекси доступності (по центру) і числа Кеніга (справа) для його вершин.

	1	2	3	4	5	6	7	8	Si	Ki
1	0	1	2	3	4	4	2	4	20	4
2	1	0	1	2	3	3	1	3	14	3
3	2	1	0	1	2	2	1	2	11	2
4	3	2	1	0	1	1	2	1	11	3
5	4	3	2	1	0	2	3	2	17	4
6	4	3	2	1	2	0	3	2	17	4
7	2	1	1	2	3	3	0	3	15	3
8	4	3	2	1	2	2	3	0	17	4



Відносні показники центральності дають можливість порівнювати рівень центральності вершин різних графів

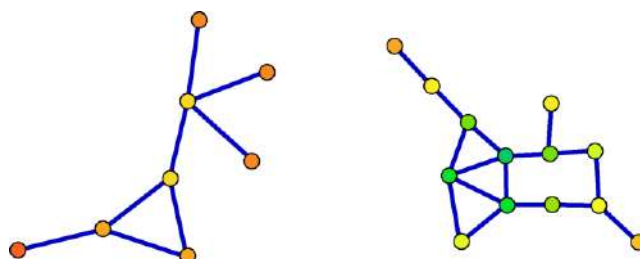
- Індекс Бавелаша
- Індекс Бічема
- Індекс відносної доступності

Індекс Бавелаша – відношення суми індексів доступності всіх вершин графа до індексу доступності даної вершини:

$$I_{Bav} = \frac{\sum_1^V S}{S_i}$$

- Чим більший індекс Бавелаша – тим більш центральне положення вершини

На рисунку нижче кольорами показано градацію індексів Бавелаша для двох графів (зелений – висока центральність, червоний – низька центральність). Помітно, що індекс Бавелаша дещо переоцінює центральність найбільш центральних вершин великих мереж.

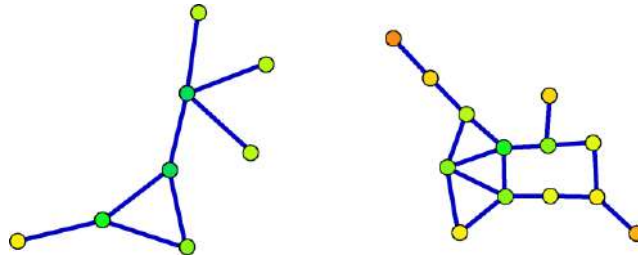


Індекс Бічема – відношення зменшеної на одиницю кількості вершин графа до індексу доступності даної вершини:

$$I_{Bich} = \frac{V - 1}{S_i}$$

- Чим більший індекс Бічема – тим більш центральне положення вершини

На рисунку нижче кольорами показано градацію індексів Бічема для двох графів (зелений – висока центральність, червоний – низька центральність). Помітно, що індекс Бічема дещо недооцінює центральність найбільш центральних вершин великих мереж.

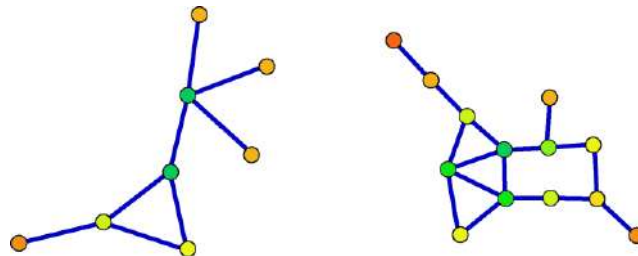


Індекс відносної доступності – відношення індексу доступності найбільш центральної вершини до індексу доступності даної вершини.

$$I_{RA} = \frac{S_{min}}{S_i}$$

- Чим більшим є значення індексу, тим більшим центральним є положення вершини

На рисунку нижче кольорами показано градацію індексів відносної доступності для двох графів (зелений – висока центральність, червоний – низька центральність).



9. Закономірності просторового розвитку мереж

Мережі (що моделюються графами) розвиваються за певними закономірностями.

Ріст мереж: відбувається перехід від простих деревоподібних структур до складних циклічних систем. Характерні процеси:

- зміна фаз дендритоутворення та циклоутворення
- формування остова
- поліцентризація

Розпад мереж: супроводжується спрощенням структури та зменшенням кількості циклів. Може бути повним або частковим. Характерні процеси:

- спрощення структури мережі – циклоруйнування, остоворуйнування, дендриторуйнування
- як наслідок – перехід мережі до нижчого класу топологічної складності

10. Топологічні дефекти структури мереж

В результаті швидкого росту або внаслідок розпаду в топологічних структурах мережі можуть виникати дефекти:

- незв'язаність частин графу
- підвищений рівень дендритності (вітвистості)
- утворення великих циклів
- багатоостовість
- глибокі клини та виступи циклічного остову

11. Зважені графи

У зважених графах вершини або ребра мають числові характеристики. Зокрема, в географії:

- вершини можуть мати вагу у вигляді чисельності населення;
- ребра – у вигляді відстані, часу або інтенсивності потоків.

Зважені графи є основою транспортного моделювання.

- Матриця суміжності вершин зваженого графа містить ваги ребер між парами вершин.
- Матриця відстаней між вершинами зваженого графа містить найменші суми ваг ребер шляхів між кожною парою вершин.

12. Застосування теорії графів у географії та урбаністиці

Основні напрями використання:

- аналіз транспортних мереж
 - оцінка доступності територій
 - геологістика
 - моделювання систем розселення
 - виявлення центрів і периферії
 - аналіз просторової нерівності
- Теорія графів також використовується у геоінформаційних системах для вирішення прикладних задач просторового планування.

Висновки

Теорія графів є універсальним інструментом аналізу просторових систем. Вона дозволяє досліджувати структуру мереж, визначати їх ефективність, виявляти слабкі місця та прогнозувати розвиток. У сучасній урбаністиці та географії графові моделі є основою для прийняття управлінських рішень, планування інфраструктури та аналізу територіального розвитку. Таким чином, опанування методів теорії графів є необхідною складовою підготовки фахівців у сфері урбаністики та просторового планування.

КОНЦЕПТУАЛІЗАЦІЯ ПРОГРАМИ

Перед написанням коду програми доцільно спочатку здійснити концептуалізацію майбутнього програмного продукту:

- для кого він буде призначений?
- які задачі вирішуватиме?
- у якому форматі прийматиме вхідні дані?
- у якому форматі відбуватиметься виведення результатів?

У нашому випадку загальна концепція програми виглядатиме так:

Програма GRAPH буде призначена для аналізу графів, що моделюють просторові мережеві системи (зокрема транспортні, інфраструктурні, територіально-адміністративні мережі), на основі матриці суміжності, заданої у форматі Excel. Це має бути універсальний інструмент, який зможе автоматично обчислювати ключові топологічні характеристики графа, оцінювати зв'язність та компактність мережі, визначати показники центральності та доступності вершин графа та використовувати результати для подальшого аналізу. Користувач отримує інструмент, який на основі матриці суміжності графа визначає повний набір його характеристик. Граф може бути як з незваженими, так і зі зваженими ребрами. Програма запускається у консолі операційної системи та керується користувачем з клавіатури комп'ютера.

На основі концепції програмного продукту можна сформулювати технічне завдання:

1. Основний функціонал – що повинна «вміти» програма:

- Визначати базові характеристики графа: кількість вершин, ребер, максимально можливе число ребер, цикломатичне число, максимально можливе цикломатичне число;
- Обчислювати матрицю відстаней між вершинами графа, визначати його діаметр, кількість компонент зв'язності;
- Визначати індекси зв'язності (альфа-індекс, бета-індекс, фі-індекс) та компактності графа (π-індекс);
- Визначати міри центральності окремих вершин графа: абсолютні (числа Кеніга, індекси доступності) та відносні (індекси Бавелаша, індекси Бічема, індекси відносної доступності);
- Визначати степені вершин графа.

2. Формат представлення вхідних даних

Граф малого та середнього розміру – до декількох сотень вершин (цього цілком достатньо для аналізу більшості міських та регіональних просторових мереж) зручно представити для подальшої автоматизованої обробки у вигляді матриці суміжності вершин. Отже, програма повинна приймати файл формату *.xlsx, у якому присутній аркуш, що містить вхідні дані: перший стовпець – назви вершин, наступні стовпці – матриця суміжності. Матриця суміжності вершин – симетрична, квадратна матриця, що містить тільки два можливі значення – '1' (якщо між двома вершинами є ребро) або '0' (якщо між двома вершинами немає ребра). Також проставимо '0' по головній діагоналі матриці – для відношень однойменних вершин. Матриця буде симетричною відносно головної діагоналі. Програма повинна обробляти найпростіші помилки введення: перевіряти існування файлу та аркуша зі вказаними користувачем іменами.

3. Формат виведення результатів

Для показників, що характеризують параметри графа одним числом, передбачимо форматований вивід тексту у консолі. Для об'ємних даних, таких як матриця відстаней між вершинами графа та показники центральності його вершин, передбачимо можливість збереження результатів до файлу MS Excel. Це дасть можливість користувачеві комфортно працювати з цими результатами після завершення роботи програми

4. Логіка виконання та внутрішня структура програми

Для уникнення дублювання коду оформимо розрахунок окремих параметрів графа у вигляді окремих функцій. У основній програмі будемо викликати ці функції за необхідністю. При цьому одні функції можуть викликатись всередині інших функцій. Наша програма буде цілісним програмним продуктом, однак варто передбачити можливість зберігання цих спеціалізованих функцій у окремому модулі та їх використання окремо одна від одної - якщо користувачу в майбутньому знадобиться визначення лише якоїсь конкретної характеристики графа. Тому бажано уніфікувати вхідний параметр всіх таких функцій – завжди використовувати матрицю суміжності вершин графа. Водночас, це зробить програму гнучкою за рахунок можливості розширення – додавання нових функцій для визначення інших характеристик графа.

Програма буде мати два блоки результатів. Перший блок – обов'язкові результати. Це ті характеристики, які будуть визначатись для будь-якого графа. Другий блок – опційні результати – характеристики, які визначаються тільки для зв'язаних графів. Відповідно, потрібно передбачити умовні обчислення та вивід результатів: якщо граф незв'язний – частина показників не визначається та не виводиться.

Програма буде мати простий текстовий інтерфейс (робота в режимі консолі), однак повинна містити інтерактивні підказки та можливість обрання користувачем опцій щодо формату виводу/збереження результатів.

Для обчислення матриці відстаней між вершинами графа використаємо алгоритм Флойда–Воршелла.

Оскільки програма повинна коректно працювати з незв'язаними графами, де виникає нескінченно велика відстань між деякими вершинами, вона повинна коректно обробляти нескінченність (∞).

5. Мова програмування

Будемо писати програму на Python. Також нам знадобляться зовнішні бібліотеки (модулі) Python: NumPy – для зручної роботи з матрицями (зокрема такими, що містять нескінченно великі значення), Pandas – для роботи з файлами MS Excel.

КОД ПРОГРАМИ

Тепер, коли ми сформулювали загальну концепцію програми, можна приступити до написання її коду. Як і було задумано, напишемо спочатку ряд спеціалізованих функцій, а вже потім будемо викликати їх у головній програмі у потрібній послідовності. Єдиним параметром для більшості спеціалізованих функцій буде матриця суміжності вершин графа. За потреби будемо викликати одні функції всередині інших для уникнення дублювання коду.

Розпочнемо з написання функції `get_vertices`, яка визначатиме кількість вершин графа – адже це базова його характеристика, яка неодноразово знадобиться для визначення інших його параметрів. Функція ця досить проста – вона приймає як єдиний свій параметр `amatrix` матрицю суміжності вершин графа, визначає її довжину (кількість рядків матриці – це і буде кількість вершин графа) і повертає одержане значення. Однак зручніше викликати у потрібних місцях цю функцію, ніж щоразу розраховувати довжину матриці.

```
def get_vertices(amatrix):
    """Розраховує кількість вершин графа.
    Параметр - матриця суміжності вершин графа"""
    vertices = len(amatrix)
    return vertices
```

Наступною напишемо функцію `get_edges`, що розраховує кількість ребер графа. Параметром функції `amatrix` знову виступає матриця суміжності вершин графа. Для цього проходимо циклом `for` всіма елементами матриці суміжності вершин графа. Ідея тут така: якщо значення елемента не дорівнює нулю – це означає, що між відповідними вершинами наявне ребро. Загальна кількість ненульових елементів матриці суміжності вершин дорівнює подвоєній кількості ребер графа – адже кожне ребро при цьому враховується двічі. Тому одержаний результат потрібно поділити на два.

```
def get_edges(amatrix):
    """Розраховує кількість ребер графа.
    Параметр - матриця суміжності вершин графа"""
    edges = 0
    for i in range(get_vertices(amatrix)):
        for j in range(get_vertices(amatrix)):
            if amatrix[i][j] != 0:
                edges += 1
    edges /= 2
    return edges
```

Далі визначимо максимально можливу кількість ребер графа. Для цього напишемо функцію `get_edges_max`. Максимально можлива кількість ребер плоского графа визначається за формулою:

Відповідно, функція `get_edges_max` повинна по-різному спрацьовувати в залежності від кількості вершин матриці, для чого вводимо ряд умовних операторів `if` та `elif`: на кроці ❶ відпрацьовуємо базовий випадок, коли кількість вершин матриці більша двох, на кроці ❷ – випадок, коли граф має 2 вершини (можливе лише 1 ребро), на кроці ❸ – випадок, коли граф має лише 1 вершину (ребер при цьому немає взагалі). При цьому всередині функції на кожному з кроків ми викликаємо раніше написану функцію `get_vertices`, яка повертає кількість вершин графа. Зовнішня функція `get_edges_max` повертає максимально можливу кількість ребер графа як значення змінної `edges_max`.

```
def get_edges_max(amatrix):
    """Розраховує максимально можливу кількість ребер графа.
    Параметр - матриця суміжності вершин графа"""
    ❶ if get_vertices(amatrix) > 2:
        edges_max = 3*(get_vertices(amatrix) - 2)
    ❷ elif get_vertices(amatrix) == 2:
        edges_max = 1
    ❸ elif get_vertices(amatrix) == 1:
        edges_max = 0
    return edges_max
```

Наступний крок – визначення функції `get_distances`. Її задача – приймати як параметр матрицю суміжності вершин графа та повертати матрицю відстаней (за визначенням – найкоротших можливих відстаней) між вершинами графа. Мати цю матрицю конче важливо, оскільки надалі вона використовується для визначення кількості компонент зв'язності графа, діаметра графа, а також абсолютних та відносних мір компактності вершин графа. Код цієї функції складніший, тож розберемо його детальніше.

```
def get_distances(amatrix):
    """Розраховує матрицю відстаней між вершинами графа.
    Параметр - матриця суміжності вершин графа"""
    ❶ dist = []
    for row in amatrix:
        dist.append(row[:])
    ❷ d = np.zeros((get_vertices(amatrix), get_vertices(amatrix)))
    ❸ for i in range(get_vertices(amatrix)):
        for j in range(get_vertices(amatrix)):
            if (i != j) and (dist[i][j] == 0):
                d[i, j] = INF
            else:
                d[i, j] = dist[i][j]
    ❹ for k in range(get_vertices(amatrix)):
        for i in range(get_vertices(amatrix)):
            for j in range(get_vertices(amatrix)):
                if d[i, k] + d[k, j] < d[i, j]:
                    d[i, j] = d[i, k] + d[k, j]
    ❺ distances = matrix_transform(d)
    ❻ return distances
```

На кроці ❶ створюється новий список `dist` і кожен рядок матриці відстаней `amatrix` копіюється до нього за допомогою оператора `row[:]`. Тепер для виконання подальших операцій будемо використовувати `dist` – копію оригінальної матриці. Це робиться для того, щоб не змінювати оригінальну матрицю – адже вона буде потрібна для виконання інших обчислень.

На кроці ❷ за допомогою функції `np.zeros` з пакета NumPy створюється квадратна матриця `d`, заповнена нулями. Її розмір – кількість вершин графа (`get_vertices(amatrix)`). Це – заготовка матриці відстаней.

На кроці ❸ відбувається початкове заповнення матриці відстаней. Якщо вершини різні ($i \neq j$) і між ними немає ребра ($dist[i][j] == 0$), то відстань між ними дорівнює `INF` (нескінченність). У

іншому випадку відстань береться з матриці суміжності: для незважених графів це завжди буде 1, для зважених – вага ребра графа.

На кроці ④ ініціюється класичний *алгоритм Флойда–Воршелла*: для кожної трійки вершин (i, j, k) перевіряється, чи вигідніше пройти з i до j через k , ніж напряму. Якщо $d[i, k] + d[k, j] < d[i, j]$ – це означає, що шлях через вершину k коротший, ніж напряму. У такому разі відстань між вершинами i та j оновлюється: $d[i, j] = d[i, k] + d[k, j]$. Після завершення виконання цього коду матриця d містить найкоротші відстані між усіма парами вершин.

На кроці ⑤ викликається функція `matrix_transform`, яка перетворює матрицю d у форматі NumPy у звичайну матрицю `distances`, представлену у вигляді списку. Цю допоміжну функцію поки ще не написано – відкладемо цей крок до завершення написання функцій, що визначають безпосередньо шукані характеристики графа та його окремих вершин.

На кроці ⑥ функція повертає у місці свого виклику матрицю відстаней між вершинами графа.

Далі створюємо функцію `get_components` для визначення кількості компонент зв'язності графа. Єдиним параметром функції вже традиційно визначаємо матрицю суміжності вершин. Також нам потрібно знати кількість вершин графа, а також мати матрицю відстаней між його вершинами – тут зручно використати вже готові, раніше написані функції.

```
def get_components(amatrix):
    """Розраховує кількість компонент зв'язності графа. Параметр - матриця
    суміжності вершин графа"""
    unvisited = [False] * get_vertices(amatrix)
    ① components = 0
    ② d = get_distances(amatrix)
    ③ for i in range(get_vertices(amatrix)):
    ④     if unvisited[i] == False:
        components += 1
        unvisited[i] = True
        for j in range(get_vertices(amatrix)):
            if (d[i][j] != 0) and (d[i][j] != "+inf"):
                unvisited[j] = True
    ⑤ return components
```

На кроці ① створюється список `unvisited`. Його довжина дорівнює кількості вершин графа, всі значення спочатку дорівнюють логічному значенню `False` (хиба). Логіка тут така: вершини, позначені як `False`, не віднесені до жодної компоненти зв'язності графа.

На кроці ② вводимо змінну для підрахунку кількості компонент зв'язності та встановлюємо її початкове значення 0.

На кроці ③ викликається функція `get_distances`, яка повертає матрицю відстаней між вершинами графа і зберігає її у змінній `d`.

На кроці ④ проходимо циклом вершинами графа. Перевіряємо, чи оброблено вже цю вершину. (На початку роботи циклу всі вершини необроблені). Якщо вершину i не оброблено – збільшуємо кількість компонент на 1 та позначаємо вершину i як оброблену – присвоюємо їй логічне значення `True` (істина). Після цього знаходимо всі вершини j , зв'язані з поточною вершиною. Для цього між двома вершинами повинен існувати шлях: $d[i][j] \neq "+inf"$. Також це не повинна бути одна й та сама вершина: $d[i][j] \neq 0$. Якщо умова виконується, вершина j належить до того ж самого компонента зв'язності. Позначаємо вершину j як оброблену – присвоюємо їй логічне значення `True`. Таким чином, кількість компонент зв'язності визначається як кількість зв'язних груп оброблених вершин.

На кроці ⑤ функція повертає число – кількість компонент зв'язності графа.

Далі напишемо функцію `get_cyclomatic_act` для визначення фактичного цикломатичного числа графа, яке розраховується за формулою:

```
def get_cyclomatic_act(amatrix):
    """Розраховує фактичне цикломатичне число графа.
    Параметр - матриця суміжності вершин графа"""
    ❶ cyclomatic_act = get_edges(amatrix) - get_vertices(amatrix) +
        get_components(amatrix)
    return cyclomatic_act
```

Зверніть увагу: на кроці ❶ всередині функції `get_cyclomatic_act` було використано одразу три інші, раніше написані функції: `get_edges` – для визначення кількості ребер графа, `get_vertices` – для визначення кількості вершин графа, `get_components` – для визначення кількості компонент зв'язності графа. У свою чергу, функція `get_edges` містить вкладену функцію `get_vertices`, а функція `get_components` – вкладені функції `get_vertices` та `get_distances`, друга з яких також має вкладену функцію `get_vertices`. Таким чином, маємо три рівні вкладеності функцій, причому всі вони – і внутрішні, і зовнішні – використовують один і той самий вхідний параметр – матрицю суміжності вершин графа.

Наступна функція функцію `get_cyclomatic_max (amatrix)` визначатиме максимально можливе цикломатичне число графа. Оскільки формула для обчислення максимально можливого цикломатичного числа плоского графа (X) працює лише тоді, коли кількість вершин графа більша двох, у коді функції з'являються умовні оператори `if` та `else`: на кроці ❶ ми обробляємо стандартні ситуації, а на кроці ❷ обробляємо винятки, коли кількість вершин матриці становить 1 або 2, і циклів при цьому бути не може в принципі.

```
def get_cyclomatic_max(amatrix):
    """Розраховує максимально можливе цикломатичне число графа.
    Параметр - матриця суміжності вершин графа"""
    ❶ if get_vertices(amatrix) > 2:
        cyclomatic_max = 2*get_vertices(amatrix) - 5
    ❷ else:
        cyclomatic_max = 0
    return cyclomatic_max
```

Маючи значення кількості вершин та ребер графа, його фактичне та максимально можливе цикломатичні числа, можна визначити міри зв'язності графа: альфа-індекс, бета-індекс, фі-індекс. Ці розрахунки також реалізуємо у вигляді окремих функцій.

Для початку напишемо функцію `get_alpha` для визначення альфа-індексу зв'язності графа.

```
def get_alpha(amatrix):
    """Розраховує альфа-індекс зв'язності графа.
    Параметр - матриця суміжності вершин графа"""
    ❶ if get_cyclomatic_max(amatrix) != 0:
    ❷     alpha = get_cyclomatic_act(amatrix)/get_cyclomatic_max(amatrix)
    ❸     alpha = round(alpha, 2)
    ❹ else:
        alpha = "не визначений"
    return alpha
```

Альфа-індекс зв'язності обчислюється лише для графів, які в принципі можуть мати цикли – тобто для таких, які мають щонайменше 3 вершини. Тому на кроці ❶ ми перевіряємо, чи часом максимально можливе цикломатичне число аналізованого графа не дорівнює 0. Якщо така халепа не трапилась – проводимо розрахунок за формулою (крок ❷). Для того, щоб вивід результату мав зручний вигляд, за допомогою функції `round` обмежуємо дробову частину числа двома знаками після коми (крок ❸). Такої точності цілком достатньо для аналізу зв'язності графа, в тому числі порівняння з іншими графами. Якщо ж максимально можливе цикломатичне число графа дорівнює 0 – замість конкретного значення альфа-індексу зв'язності виводимо повідомлення про те, що цей індекс для даного графа не визначений (крок ❹).

Тепер напишемо функцію `get_beta` для визначення бета-індексу зв'язності графа.

```
def get_beta(amatrix):
    """Розраховує бета-індекс зв'язності графа.
    Параметр - матриця суміжності вершин графа"""
    beta = get_edges(amatrix)/get_vertices(amatrix)
    ❶ beta = round(beta, 2)
    return beta
```

У цьому випадку додаткова перевірка не знадобилась, так як кількість вершин графа (у знаменнику) не може дорівнювати 0 – завжди буде щонайменше 1 вершина. На кроці ❶ значення бета-індексу округлюємо до двох десяткових знаків.

Зрештою, потрібна ще функція `get_phi` для визначення фі-індексу зв'язності графа.

```
def get_phi(amatrix):
    """Розраховує фі-індекс зв'язності графа.
    Параметр - матриця суміжності вершин графа"""
    ❶ if get_edges_max(amatrix) != 0:
    ❷     phi = get_edges(amatrix)/get_edges_max(amatrix)
    ❸     phi = round(phi, 2)
    ❹ else:
        phi = "не визначений"
    return phi
```

У цьому випадку може виникнути ділення на 0, так як у випадку графа з однією вершиною (рідкісна, але можлива ситуація) граф не матиме жодного ребра. Тому, за аналогією до альфа-індексу, вводимо додаткову перевірку (крок ❶), і в залежності від її результату обраховуємо за формулою (крок ❷) і округлюємо отримане значення до двох десяткових знаків (крок ❸), або ж виводимо повідомлення про те, що для цього графа фі-індекс зв'язності не визначений (крок ❹).

Окрім індексів зв'язності графа, нас цікавить також пі-індекс компактності графа. Однак його розрахунок вимагає знання діаметра графа. У свою чергу, діаметр графа дорівнює найбільшому з чисел Кеніга для вершин графа – відстаней від даної вершини графа до найбільш віддаленої від неї. Отже, є сенс спочатку розрахувати числа Кеніга для вершин графа, тоді визначити діаметр графа, після чого, знаючи діаметр, розрахувати пі-індекс.

Числа Кеніга легко визначити, маючи матрицю відстаней між вершинами графа – це будуть найбільші значення в кожному рядку цієї матриці. Функція `get_koenigs` виглядатиме так:

```

❶ def get_koenigs(amatrix):
    """Розраховує числа Кеніга для вершин графа. Параметр - матриця суміжності
    вершин графа"""
❷     koenigs = []
❸     if get_components(amatrix) == 1:
❹         d = get_distances(amatrix)
❺         for row in d:
             koenigs.append(max(row))
    return koenigs

```

Вхідний параметр цієї функції – це матриця суміжності вершин графа (Крок **❶**). Далі створюємо порожній список **koenigs** для збереження чисел Кеніга (Крок **❷**). Потім потрібно перевірити, чи є граф зв'язаним, адже числа Кеніга можна визначити лише для зв'язаного графа. Граф буде зв'язаним, коли кількість його компонент зв'язності дорівнює 1 (Крок **❸**). Якщо граф зв'язаний – розраховуємо матрицю відстаней між вершинами графа (Крок **❹**), шукаємо найбільше значення у кожному рядку і додаємо його до списку **koenigs** (Крок **❺**). У підсумку функція повертає цей список. Що відбудеться у випадку, якщо граф не є зв'язаним, тобто має більше одного компонента зв'язності? Тоді список **koenigs** залишиться порожнім, тобто не міститиме жодного елемента.

Маючи числа Кеніга для вершин графа, можна визначити його діаметр за допомогою функції **get_diameter**, яка міститиме вкладену функцію **get_koenigs**.

```

def get_diameter(amatrix):
    """Розраховує діаметр графа. Параметр - матриця суміжності вершин графа"""
❶     k = get_koenigs(amatrix)
❷     if k:
❸         diameter = max(k)
❹     else:
         diameter = "не визначений"
    return diameter

```

На кроці **❶** визначаємо числа Кеніга для вершин графа, після чого на кроці **❷** перевіряємо, чи був одержаний список чисел Кеніга не порожнім. Якщо він не порожній (а це автоматично означає, що граф зв'язаний) – на кроці **❸** визначаємо діаметр як найбільше з чисел Кеніга та присвоюємо одержане значення змінній **diameter**. У іншому випадку змінна **diameter** отримує текстове значення "не визначений" (крок **❹**).

Зрештою, напишемо функцію **get_pi** для розрахунку пі-індекса компактності графа.

```

def get_pi(amatrix):
    """Розраховує пі-індекс компактності графа. Параметр - матриця суміжності
    вершин графа"""
❶     d = get_diameter(amatrix)
❷     if d != "не визначений":
❸         pi = get_edges(amatrix)/get_diameter(amatrix)
❹         pi = round(pi, 2)
❺     else:
         pi = "не визначений"
    return pi

```

Розраховуємо діаметр графа за допомогою написаної перед цим функції `get_diameter` (крок ❶), після чого перевіряємо одержане значення (Крок ❷). Якщо діаметр графа визначений (граф, відповідно, зв'язаний) – розраховуємо пі-індекс компактності графа, присвоюємо його змінній `pi` (крок ❸) та округлюємо одержане значення до двох десяткових знаків (крок ❹). Якщо ж діаметр графа не визначений – присвоюємо змінній `pi` текстове значення "не визначений" (Крок ❹).

Тепер можна обрахувати абсолютні показники центральності окремих вершин графа. Для розрахунку першого такого показника, чисел Кеніґа, функцію вже було створено. Друга абсолютна міра центральності – індекс доступності вершин графа, тобто сума відстаней від даної вершини графа до всіх інших його вершин. Для розрахунку індексів центральності напишемо функцію `get_accessibility_indices`. Функція повертатиме список індексів центральності для зв'язаних графів.

```
def get_accessibility_indices(amatrix):
    """Розраховує індекси доступності вершин графа. Параметр - матриця суміжності
    вершин графа"""
    ❶ accessibility_indices = []
    ❷ if get_components(amatrix) == 1:
    ❸     d = get_distances(amatrix)
    ❹     for row in d:
        accessibility_indices.append(sum(row))
    return accessibility_indices
```

Спочатку створюємо список для зберігання індексів центральності (крок ❶). Потім перевіряємо, чи граф зв'язаний (крок ❷). Якщо так – розраховуємо матрицю відстаней (крок ❸), після чого для кожного рядка матриці відстаней визначаємо суму його елементів та додаємо одержане значення до списку індексів доступності (крок ❹).

До відносних показників центральності належать індекси Бавелаша, Бічема та відносної доступності. Оформимо розрахунок кожного з них у вигляді окремої функції. Розпочнемо з індексу Бавелаша – напишемо функцію `get_bavelashes`.

```
def get_bavelashes(amatrix):
    """Розраховує індекси Бавелаша вершин графа. Параметр - матриця суміжності
    вершин графа"""
    ❶ bavelashes = []
    ❷ if get_components(amatrix) == 1:
    ❸     indices = get_accessibility_indices(amatrix)
    ❹     for index in indices:
        bavelashes.append(sum(indices)/index)
    return bavelashes
```

Спочатку створюємо список для зберігання індексів Бавелаша (крок ❶). Потім перевіряємо, чи граф зв'язаний (крок ❷). Якщо так – розраховуємо індекси доступності за допомогою функції `get_accessibility_indices` (крок ❸), після чого проходимо циклом індексами доступності, щоразу обраховуючи відношення їх загальної суми до поточного індексу доступності – таким чином отримуємо значення індексів Бавелаша для вершин графа (крок ❹).

Для обрахунку індексів Бічема напишемо функцію `get_beachems`.

```

def get_beachems(amatrix):
    """Розраховує індекси Бічема вершин графа. Параметр - матриця суміжності
    вершин графа"""
    ❶ beachems = []
    ❷ if get_components(amatrix) == 1:
    ❸     indices = get_accessibility_indices(amatrix)
    ❹     for index in indices:
        beachems.append((get_vertices(amatrix) - 1)/index)
    return beachems

```

За своїм призначенням кроки ❶, ❷, ❸ ідентичні попередній функції, а на кроці ❹ відбувається розрахунок індексів Бічема для вершин графа як відношень зменшеної на одиницю кількості вершин графа до індексів доступності вершин.

Зрештою, розрахунок індексів відносної доступності для вершин графа буде виконувати функція `get_reaccessibility_indices`.

```

def get_reaccessibility_indices(amatrix):
    """Розраховує індекси відносної доступності вершин графа. Параметр - матриця
    суміжності вершин графа"""
    ❶ relaccessibility_indices = []
    ❷ if get_components(amatrix) == 1:
    ❸     indices = get_accessibility_indices(amatrix)
    ❹     for index in indices:
        relaccessibility_indices.append(min(indices)/index)
    return relaccessibility_indices

```

Тут кроки ❶, ❷, ❸ ідентичні двом попереднім функціям, а на кроці ❹ відбувається розрахунок індексів відносної доступності для вершин графа як відношень найменшого з індексів доступності вершин графа до індексу доступності поточної вершини.

Із заявленого функціоналу наша програма досі «не вміє» робити ще одну річ – визначати степені вершин графа. Це завдання буде вирішувати окрема функція `get_degrees`.

```

def get_degrees(amatrix):
    """Розраховує степені вершин графа. Параметр - матриця суміжності вершин
    графа"""
    ❶ degrees = []
    ❷ for row in amatrix:
    ❸     degree = 0
    ❹     for element in row:
        if element != 0:
            degree += 1
    ❺     degrees.append(degree)
    return degrees

```

На кроці ❶ створюємо порожній список для зберігання степенів вершин графа. На кроці ❷ проходимо циклом рядками матриці суміжності вершин – фактично – вершинами графа. Встановлюємо початкове значення степеня вершини 0 (крок ❸). Далі перевіряємо кожен елемент рядка матриці: якщо він не дорівнює 0 (є відношення суміжності з іншою

вершиною) – збільшуємо значення степеня вершини на 1 (крок ④). Вираз `degree += 1` – це скорочена форма запису `degree = degree + 1`. На кроці ⑤ додаємо степінь поточної вершини до загального списку, який і повертає функція.

Тепер маємо в наявності всі ключові функції, які визначають параметри графа в цілому або окремих його вершин відповідно до визначеного технічного завдання. Однак, для роботи програми не вистачає ще кількох функцій.

По-перше, у функції `get_distances` ми використали внутрішню функцію `matrix_transform`, яка повинна трансформувати матрицю у форматі NumPy у звичайний формат представлення матриці в Python у вигляді списку. Таке перетворення потрібне, оскільки низка функцій використовують матрицю відстаней між вершинами графа, і написані вони таким чином, що ця матриця повинна бути саме у звичайному форматі. Ця функція має один вхідний параметр `matrix` – матрицю у форматі NumPy.

```
def matrix_transform(matrix):
    """Трансформує матрицю з формату numpy у звичайний формат"""
    ① count = matrix.shape[0]
    ② transformed = []
    ③ for i in range(count):
    ④     row = []
    ⑤     for j in range(count):
        if matrix[i,j] == INF:
            row.append("+inf")
        else:
            row.append(float(matrix[i,j]))
    ⑥ transformed.append(row)
    return transformed
```

На кроці ① визначаємо кількість рядків у вхідній матриці. На кроці ② створюємо порожній список `transformed` – заготовку для трансформованої матриці. На кроці ③ проходимо циклом всіма рядками вхідної матриці. На кроці ④ створюємо порожній список `row` елементів рядка трансформованої матриці. На кроці ⑤ перевіряємо значення вхідної матриці у відповідному рядку: якщо воно дорівнює `INF`, додаємо до списку `row` рядкове значення `"inf"`, у іншому разі – якщо це конкретне скінченне число – трансформуємо його у звичайний формат дійсних чисел з плаваючою комою за допомогою вбудованої функції `float` та додаємо до списку `row`. На кроці ⑥ додаємо список `row` як черговий рядок до трансформованої матриці.

По-друге, зручно виділити як окрему функцію `read_file_xlsx` блок коду, відповідальний за зчитування вхідних даних. Як було визначено на етапі концептуалізації програми, вхідними даними слугуватиме матриця суміжності вершин графа. Цю матрицю зручно створювати в табличному редакторі MS Excel. Це буде квадратна матриця (з однаковою кількістю рядків та стовпців). Перший рядок та перший стовпець матриці міститимуть підписи – номери вершин графа. Значення самої матриці будуть дорівнювати 1, якщо дві відповідні вершини суміжні, та 0, якщо ці вершини не є суміжними. Також проставимо нулі по головній діагоналі матриці – для значень, що знаходяться на перетині однойменних рядків та стовпців. Це необхідно для коректної роботи деяких функцій програми.

Для зчитування вхідних даних з файлу формату MS Excel нам знадобиться модуль `Pandas`. Користувачеві потрібно буде ввести назви файлу MS Excel та аркуша з вхідними даними. Крім того, потрібно продумати, яким чином програма реагуватиме на помилки користувача при введенні назв файлу та аркуша.

```

❶ def read_file_excel():
❷     """Зчитує дані з файлу *.xlsx"""
❸     while True:
❹         try:
                file_name = input("\nВведіть ім'я файлу *.xlsx з даними: ")
                sheet_name = input("\nВведіть назву аркуша з матрицею
                суміжності вершин: ")
                file_path = f'{file_name}.xlsx'
                df = pd.read_excel(file_path, sheet_name=sheet_name)
                return df
                break
❺         except FileNotFoundError:
                print(f"\nПомилка: файл '{file_name}.xlsx' не існує.")
                repeat = input("Повторне введення назви файлу? Так - 1,
                Скасувати та вийти - 0: ")
                if repeat == "0":
                    print("Завершення роботи програми.")
                    sys.exit()
❻         except ValueError:
                print(f"\nПомилка: аркуш '{sheet_name}.xlsx' не існує або містить
                некоректні дані.")
                repeat = input("Повторне введення назви файлу? Так - 1,
                Скасувати та вийти - 0: ")
                if repeat == "0":
                    print("Завершення роботи програми.")
                    sys.exit()

```

На кроці ❶ оголошуємо функцію `read_file_excel`. Рядок ❷ – це стандартний docstring, стислий опис завдання, яке виконує функція. Як і в усіх раніше написаних функціях, він потрібен для того, щоби інші читачі цього коду (а також і ми самі через деякий проміжок часу) могли зрозуміти (пригадати) функціонал відповідної функції, а також – за потреби – певні особливості її внутрішньої логіки. На кроці ❸ ініціюємо цикл `while`, який працюватиме доти, доки не будуть коректно зчитані дані з файлу MS Excel, або поки користувач не вирішить відмовитись від чергової спроби введення даних і примусово завершити роботу програми. На кроці ❹ ініціюємо конструкцію `try` → `except`. Блок `try` приймає від користувача назву файлу MS Excel та назву аркуша, на якому розташовані дані. Якщо спроба успішна, зчитані дані зберігаються як об'єкт Pandas dataframe у вигляді змінної `df`; значення цієї змінної повертається функцією `read_file_excel` у головну програму, а цикл `while` переривається оператором `break`. Якщо назву файлу введено невірно (програма не знаходить файл з такою назвою), на кроці ❺ спрацьовує блок `except FileNotFoundError`; якщо невірно введено назву аркуша, на кроці ❻ спрацьовує блок `except ValueError`. Ці блоки ідентифікують відповідні винятки і пропонують користувачеві повторно ввести вхідні дані, або завершити роботу програми, для чого використовується стандартна функція `sys.exit`.

Оскільки написання інших функцій не передбачається, можна перейти до роботи над кодом основної програми.

Розпочнемо зі зчитування вхідних даних.

```

❶ df = read_file_xlsx()
❷ vernames = df.iloc[:, 0].tolist()
❸ adjacency = [df.iloc[:, column].tolist() for column in range(1, len(values)+1)]
❹ INF = float('inf')

```

На кроці ❶ зберігаємо дані з файлу MS Excel у вигляді Pandas Data Frame (змінна `df`), викликаючи функцію `df = read_file_xlsx()`. На кроці ❷ значення першого стовпця (з індексом 0 !) Pandas Data Frame зберігаємо у вигляді списку (змінна `values`). Цей список буде містити номери (або імена) вершин і знадобиться нам при виводі даних. На кроці ❸ за допомогою генераторного списку формуємо власне матрицю суміжності вершин, послідовно зчитуючи дані стовпців Pandas dataframe, починаючи з другого (індекс 1) до останнього (індекс `len(values)+1`). Ця матриця буде зберігатись у змінній `adjacency`.

Також одразу робимо ще одну необхідну операцію: на кроці ❹ за допомогою функції `float` рядок `'inf'` трансформуємо у спеціальне значення «нескінченність» формату NumPy – `INF`. Тепер при роботі з матрицями формату NumPy ми зможемо використовувати змінну `INF` для позначення нескінченно великого числа. А така матриця використовується у функції `get_distances` для побудови матриці відстаней між вершинами графа.

Тепер будемо викликати раніше написані спеціалізовані функції та послідовно визначати характеристики графа та його вершин. Одночасно потрібно буде поміркувати над оптимальним форматом виведення та/або зберігання результатів.

Розпочнемо з визначення кількості вершин. Кількість вершин – це одне число, тож його можна просто вивести на екран.

```

--вирізано--
INF = float('inf')
❶ vertices = get_vertices(adjacency)
❷ print(f"\nКількість вершин графа: {vertices}")

```

На кроці ❶ викликаємо функцію `get_vertices` та передаємо їй як аргумент матрицю суміжності вершин графа `adjacency`. На кроці ❷ виводимо на екран повідомлення для користувача про кількість вершин графа.

Далі визначаємо кількість ребер графа.

```

--вирізано--
print(f"\nКількість вершин графа: {vertices}")

❶ edges = get_edges(adjacency)
❷ print(f"\nКількість ребер графа: {edges: .0f}")

```

На кроці ❶ викликаємо функцію `get_edges` та передаємо їй як аргумент матрицю суміжності вершин графа `adjacency`. На кроці ❷ виводимо на екран повідомлення для користувача про кількість ребер графа. Відмінність від аналогічного коду для виведення кількості вершин – округлення отриманого значення при виведенні на екран до цілого числа: `edges: .0f`. Справа в тому, що функція `get_edges` повертає дійсне число з плаваючою комою, однак кількість ребер завжди буде цілим числом. Щоб вивід був зручнішим для користувача, позбавимо його необхідності спостерігати зайві нулі після крапки.

Визначаємо максимально можливу кількість ребер графа.

--вирізано--

```
print(f"\nКількість ребер графа: {edges:.0f}")
```

- ❶ `edges_max = get_edges_max(adjacency)`
 - ❷ `print(f"\nМаксимально можлива кількість ребер графа: {edges_max}")`
-

На кроці ❶ викликаємо функцію `get_edges_max` та передаємо їй як аргумент матрицю суміжності вершин графа `adjacency`. На кроці ❷ виводимо на екран повідомлення для користувача про максимально можливу кількість вершин графа. Оскільки функція `edges_max` працює лише з натуральними числами і не використовує процедуру ділення – округлювати результат при виведенні на екран не потрібно.

Наступний етап – а побудова матриці відстаней між вершинами графа. На відміну від попередніх характеристик графа, матриця відстаней – це не одне число, а двовимірний таблиця. Простий вивід її на екран, особливо у випадку графа з великою кількістю вершин, буде незручним для користувача. Крім того, користувач може захотіти використати матрицю відстаней пізніше для інших обчислень. Тому ми передбачимо виведення цієї матриці на екран терміналу, однак запропонуємо користувачеві додаткову опцію – збереження матриці відстаней у файл MS Excel.

--вирізано--

```
print(f"\nМаксимально можлива кількість ребер графа: {edges_max}")
```

- ❶ `distances = get_distances(adjacency)`
 - ❷ `print("\nМатриця відстаней між вершинами графа:")`
`print(distances)`
 - ❸ `writing = input("\nЗберегти матрицю відстаней між вершинами графа у файл *.xlsx?`
`Так - '1', Ні - 'Enter': ")`
 - ❹ `if writing == "1":`
`distance_matrix = pd.DataFrame(distances)`
`distance_matrix.to_excel('distance_matrix.xlsx', index=False)`
`print("Матрицю відстаней між вершинами графа збережено у файлі`
`distance_matrix.xlsx")`
-

На кроці ❶ викликаємо функцію `get_distances`, яка обраховує та повертає матрицю відстаней. На кроці ❷ виводимо цю матрицю на екран. На кроці ❸ запитуємо користувача, чи потрібно зберегти матрицю відстаней також у файл MS Excel. Якщо відповідь користувача – цифра ‘1’, введена з клавіатури – на кроці ❹ зберігаємо матрицю відстаней як об’єкт `Pandas Dataframe distance_matrix`, записуємо його у файл MS Excel (цей файл матиме ім’я `distance_matrix.xlsx` та буде створений у тій самій директорії, де знаходиться файл самої програми), та інформуємо про це користувача, виводячи на екран повідомлення. Якщо користувач натисне ‘Enter’ або здійснить будь-який інший ввід з клавіатури окрім ‘1’ – запис у файл здійснюватись не буде і програма перейде до виконання наступного коду.

Далі визначаємо кількість компонент зв’язності графа, фактичне та максимально можливе цикломатичні числа графа. Ці три операції зводяться до виклику відповідних функцій та виводу на екран повернутих ними значень.

--вирізано--

```
print("Матрицю відстаней між вершинами графа збережено у файлі
```

```
distance_matrix.xlsx")
```

- ❶ components = get_components(adjacency)
print(f"\nКількість компонент зв'язності графа: {components}")
 - ❷ cyclomatic_act = get_cyclomatic_act(adjacency)
print(f"\nФактичне цикломатичне число графа: {cyclomatic_act:.0f}")
 - ❸ cyclomatic_max = get_cyclomatic_max(adjacency)
print(f"\nМаксимально можливе цикломатичне число графа: {cyclomatic_max}")
-

Відповідно, на кроці ❶ викликаємо функцію `get_components` і виводимо на екран кількість компонент зв'язності графа; на кроці ❷ викликаємо функцію `get_cyclomatic_act` і виводимо на екран фактичне цикломатичне число графа; на кроці ❸ викликаємо функцію `get_cyclomatic_max` і виводимо на екран фактичне цикломатичне число графа.

Наступний етап – розрахунок індексів зв'язності графа. Розпочнемо з альфа-індексу. Для зручності користувача було б добре округлювати вивід до двох десяткових знаків. Однак, якщо альфа-індекс не визначений (кількість вершин графа менша трьох), виникне помилка – округлити рядок тексту значення неможливо. Тому обробку виводу потрібно розділити в залежності від одержаного значення.

```
--вирізано--
```

```
print(f"\nМаксимально можливе цикломатичне число графа: {cyclomatic_max}")
```

- ❶ alpha = get_alpha(adjacency)
 - ❷ if alpha == "не визначений":
 print(f"\nАльфа-індекс зв'язності графа: {alpha}")
 - ❸ else:
 print(f"\nАльфа-індекс зв'язності графа: {alpha:.2f}")
-

Визначаємо альфа-індекс за допомогою функції `get_alpha` (крок ❶). Якщо функція повертає рядок "не визначений" – виводимо на екран інформацію, що альфа-індекс для цього графа не визначений (крок ❷). В усіх інших випадках виводимо значення альфа-індексу, округлене до двох десяткових знаків.

Далі визначаємо бета-індекс зв'язності. У цьому випадку виняткових значень не виникає, одержаний індекс завжди є дійсним числом. Тому розраховуємо його викликом функції `get_beta` (крок ❶) та одразу виводимо на екран, округлюючи одержане значення для зручності (крок ❷).

```
--вирізано--
```

```
else:
```

```
    print(f"\nАльфа-індекс зв'язності графа: {alpha:.2f}")
```

- ❶ beta = get_beta(adjacency)
 - ❷ print(f"\nБета-індекс зв'язності графа: {beta:.2f}")
-

Логіка обчислення та вивід фі-індексу зв'язності така сама, як і для альфа-індексу – всі три кроки аналогічні.

```
--вирізано--
```

```
print(f"\nБета-індекс зв'язності графа: {beta: .2f}")
```

- ❶ `phi = get_phi(adjacency)` # розрахунок фі-індексу зв'язності графа
 - ❷ `if phi == "не визначений":`
`print(f"\nФі-індекс зв'язності графа: {phi}")`
 - ❸ `else:`
`print(f"\nФі-індекс зв'язності графа: {phi: .2f}")`
-

Тепер виводимо числа Кеніґа. Потрібно передбачити різні варіанти виводу для зв'язаних графів (числа Кеніґа визначені) та незв'язаних графів (числа Кеніґа не визначені). Якщо числа Кеніґа визначені – виводити їх для зручності користувача будемо не списком у квадратних дужках (а саме такий результат вийде, якщо просто написати `print(koenigs)`), а окремо для кожної вершини графа. Для цього знадобиться пройти циклом `for` списком чисел Кеніґа, який поверне функція `get_koenigs`.

```
--вирізано--
```

```
else:
```

```
print(f"\nФі-індекс зв'язності графа: {phi: .2f}")
```

- ❶ `koenigs = get_koenigs(adjacency)`
 - ❷ `if koenigs:`
 - ❸ `print("\nЗначення чисел Кеніґа вершин графа:")`
`for i in range(len(koenigs)):`
`print(f"\tЧисло Кеніґа вершини {vernames[i]} = {koenigs[i]}")`
 - ❹ `else:`
`print("\nЗначення чисел Кеніґа вершин графа: не визначені")`
-

На кроці ❶ викликаємо функцію `get_koenigs`, яка повертає числа Кеніґа для вершин графа у вигляді списку `koenigs`. Далі на кроці ❷ перевіряємо, чи цей список не порожній. Якщо результат перевірки позитивний – проходимо циклом `for` вершинами графа та виводимо для кожної вершини її число Кеніґа (крок ❸). Якщо список порожній – виводимо інформацію про те, що значення чисел Кеніґа для вершин цього графа не визначені.

Визначаємо діаметр графа. Тут досить викликати функцію (крок ❶) та вивести повернуте нею значення на екран (крок ❷). Якщо діаметр графа визначити не можливо, функція вже повертає рядок `"не визначений"`, а якщо діаметр обчислюється, то він завжди буде натуральним числом – немає потреби округлювати вивід.

```
--вирізано--
```

```
print("\nЗначення чисел Кеніґа вершин графа: не визначені")
```

- ❶ `diameter = get_diameter(adjacency)`
 - ❷ `print(f"\nДіаметр графа: {diameter}")`
-

Далі визначаємо пі-індекс компактності графа. Викликаємо функцію `get_pi` (крок ❶); якщо вона повертає рядок `"не визначений"` – виводимо його на екран (крок ❷), у іншому випадку виводимо значення індексу компактності, округливши вивід до двох десяткових знаків (крок ❸).

```
--вирізано--
```

```
print(f"\nДіаметр графа: {diameter}")
```

- ❶ pi = get_pi(adjacency)
 - ❷ if pi == "не визначений":
 print(f"\nПі-індекс компактності графа: {pi}")
 - ❸ else:
 print(f"\nПі-індекс компактності графа: {pi: .2f}")
-

На черзі абсолютні та відносні індекси центральності вершин графа, окрім вже обрахованих чисел Кеніга. Почнемо з індексів доступності. Так само як і числа Кеніга, їх будемо виводити у відформатованому вигляді, окремо для кожної вершини графа.

```
--вирізано--
```

```
else:
```

```
    print(f"\nПі-індекс компактності графа: {pi: .2f}")
```

- ❶ accessibility_indices = get_accessibility_indices(adjacency)
 - ❷ if accessibility_indices:
 print("\nЗначення індексів доступності вершин графа:")
 for i in range(len(accessibility_indices)):
 print(f"\tІндекс доступності вершини {vernamentes[i]} =
❸ {accessibility_indices[i]}")
 else:
 print("\nЗначення індексів доступності вершин графа: не визначені")
-

На кроці ❶ викликаємо функцію `get_accessibility_indices`, яка повертає індекси доступності для вершин графа у вигляді списку `accessibility_indices`. Далі на кроці ❷ перевіряємо, чи цей список не порожній. Якщо результат перевірки позитивний – проходимо циклом `for` вершинами графа та виводимо для кожної вершини її індекс доступності (крок ❸). Якщо список порожній – виводимо інформацію про те, що значення індексів доступності для вершин цього графа не визначені.

Розрахунок та виведення на екран індексів Бавелаша, Бічема та відносної доступності відбувається за аналогічною схемою, тому не будемо зосереджуватись на детальному описі – лише позначимо аналогічні кроки ❶, ❷ та ❸ у фрагментах коду. Єдина відмінність – для цих показників вивід округлюємо до двох десяткових знаків.

```
--вирізано--
```

```
    print("\nЗначення індексів доступності вершин графа: не визначені")
```

```
# визначення індексів Бавелаша
```

- ❶ bavelashes = get_bavelashes(adjacency)
 - ❷ if bavelashes:
 print("\nЗначення індексів Бавелаша вершин графа:")
 for i in range(len(bavelashes)):
 print(f"\tІндекс Бавелаша вершини {vernamentes[i]} = {bavelashes[i]: .2f}")
 - ❸ else:
 print("\nЗначення індексів Бавелаша вершин графа: не визначені")
-

```

# визначення індексів Бічема
❶ beachems = get_beachems(adjacency)
❷ if beachems:
    print("\nЗначення індексів Бічема вершин графа:")
    for i in range(len(beachems)):
        print(f"\tІндекс Бічема вершини {vernemes[i]} = {beachems[i]: .2f}")
❸ else:
    print("\nЗначення індексів Бічема вершин графа: не визначені")

# визначення індексів Бічема
❶ relaccessibility_indices = get_relaccessibility_indices(adjacency)
❷ if relaccessibility_indices:
    print("\nЗначення індексів відносної доступності вершин графа:")
    for i in range(len(relaccessibility_indices)):
        print(f"\tІндекс відносної доступності вершини {vernemes[i]} =
        {relaccessibility_indices[i]: .2f}")
    else:
❸ print("\nЗначення індексів відносної доступності вершин графа: не визначені")

```

Звертаємо лише увагу на використання спеціальних символів для форматування виводу: символ `\n` виводить наступний текст з нового рядка, а символ `\t` виводить наступний текст з нового рядка з абзацним відступом (табуляція).

Ще потрібно розрахувати і вивести на екран степені вершин графа. Схема коду тут аналогічна, однак степені вершини визначається для будь-якого графа (в тому числі незв'язаного) та завжди є цілим невід'ємним числом, тому залишаються лише кроки ❶ та ❷

```

--вирізано--
else:
    print("\nЗначення індексів відносної доступності вершин графа: не визначені")

❶ degrees = get_degrees(adjacency)
❷ print(f"\nЗначення степенів вершин графа:")
    for i in range(len(degrees)):
        print(f"\tСтепінь вершини {i + 1} = {degrees[i]}")

```

Таким чином, визначено та виведено на екран всі характеристики графа та його окремих вершин, які визначаються цією програмою. Крім того, користувач за бажання може зберегти матрицю відстаней між вершинами графа у файл MS Excel. Однак, логічно припустити, що користувач також захоче зберегти у табличному вигляді показники центральності та степені вершин графа. Запропонуємо йому таку можливість.

```

--вирізано--
else:
    print("\nЗначення індексів відносної доступності вершин графа: не визначені")

❶ if get_components(adjacency) == 1:
❷     writing = input("\nЗберегти показники центральності та степені вершин графа у
    файл *.xlsx? Так - '1', Ні - 'Enter': ")
❸     if writing == "1":
❹         centrality_indices = pd.DataFrame({

```

```

        'Назва вершини': vernames,
        'Число Кеніґа': koenigs,
        'Індекс доступності': accessibility_indices,
        'Індекс Бавелаша': bavelashes,
        'Індекс Бічема': beachems,
        'Індекс відносної доступності': relaccessibility_indices,
        'Степінь вершини': degrees,
    })
5    centrality_indices.to_excel('centrality_indices.xlsx', index=False)
6    print("Показники центральності та степені вершин графа збережено у
      файлі centrality_indices.xlsx")

```

Оскільки всі ці показники визначаються лише для зв'язаних графів, на кроці **1** перевіряємо кількість компонент зв'язності графа. Якщо така компоненти лише одна (граф зв'язаний), на кроці **2** запитуємо користувача про потребу збереження показників центральності та степенів вершин графа у файл MS Excel. Якщо від користувача отримано ствердну відповідь (з клавіатури введено '1') (крок **3**) – формуємо словник та записуємо його у Pandas Dataframe `centrality_indices` (крок **4**). Ключами якого є назви вершин та показників ('Номер вершини', 'Число Кеніґа', 'Індекс доступності', 'Індекс Бавелаша', 'Індекс Бічема', 'Індекс відносної доступності' та 'Степінь вершини'), а значеннями – відповідні списки (`vernames`, `koenigs`, `accessibility_indices`, `bavelashes`, `beachems`, `relaccessibility_indices`, `degrees`), що були повернуті відповідними функціями (окрім першого списку `vernames`, який було зчитано з файлу MS Excel з вхідними даними). Зверніть увагу на форматування словника за допомогою відступів – це зручний і стандартний спосіб запису довгих словників (інакше довелось би записувати словник у одному довжелезному рядку коду). На кроці **5** зберігаємо Pandas Dataframe `centrality_indices` до файлу MS Excel з назвою `centrality_indices.xlsx`. Цей файл буде збережено у тій самій директорії, що і файл самої програми. На кроці **6** інформуємо про це користувача програми.

Тепер програма «вміє» отримувати дані від користувача, обробляти їх, визначаючи задані характеристики графа, та повідомляти результати користувачу, за потреби зберігаючи окремі категорії вихідних даних у файли MS Excel.

Після того, як підготовлені окремі фрагменти програми, необхідно скомпонувати їх в цілісний робочий код, додавши певні необхідні або бажані його елементи.

По-перше, потрібно імпортувати зовнішні модулі Pandas (**1**) та NumPy (**2**), а також вбудований модуль системних функцій `sys` (**3**). Ми використовували функції та методи з цих модулів, тому якщо забути їх імпортувати, програма не буде працювати і поверне повідомлення про помилку. Це потрібно зробити у самому початку коду програми.

```

1  import numpy as np
2  import pandas as pd
3  import sys

```

Далі визначаємо наші функції, зі створення яких ми і починали роботу над програмою. Всі власні функції необхідно визначити до початку коду основної програми, де вони будуть викликатись за необхідністю. Код усіх функцій було детально розібрано вище, тож наведемо тут лише перші рядки (оголошення) кожної функції.

```

import numpy as np
import pandas as pd
import sys

def matrix_transform(matrix):
    --вирізано--
def read_file_xlsx():
    --вирізано--
def get_vertices(amatrix):
    --вирізано--
def get_edges(amatrix):
    --вирізано--
def get_edges_max(amatrix):
    --вирізано--
def get_distances(amatrix):
    --вирізано--
def get_components(amatrix):
    --вирізано--
def get_cyclomatic_act(amatrix):
    --вирізано--
def get_cyclomatic_max(amatrix):
    --вирізано--
def get_alpha(amatrix):
    --вирізано--
def get_beta(amatrix):
    --вирізано--
def get_phi(amatrix):
    --вирізано--
def get_koenigs(amatrix):
    --вирізано--
def get_diameter(amatrix):
    --вирізано--
def get_pi(amatrix):
    --вирізано--
def get_accessibility_indices(amatrix):
    --вирізано--
def get_bavelashes(amatrix):
    --вирізано--
def get_beachems(amatrix):
    --вирізано--
def get_relaccessibility_indices(amatrix):
    --вирізано--
def get_degrees(amatrix):
    --вирізано--

```

Тепер, коли всі функції визначені, переходимо до коду основної частини програми.

Було б добре, якби програма «вітала» користувача при запуску. Додаємо для цього один рядок коду (❶) – вітання користувача та інформування про назву програми ('GRAPH').

--вирізано--

```
def get_degrees(amatrix):
    """Розраховує степені вершин графа. Параметр - матриця суміжності вершин графа"""
    degrees = []
    for row in amatrix:
        degree = 0
        for element in row:
            if element != 0:
                degree += 1
        degrees.append(degree)
    return degrees
```

❶ print("\nЦе програма GRAPH")

Далі розмістимо код основної частини програми – від зчитування вхідних даних з файлу MS Excel у Pandas DataFrame (**❶**) до збереження показників центральності та степенів вершин графа у вихідний файл MS Excel (**❷**).

--вирізано--

```
print("\nЦе програма GRAPH")
```

❶ df = read_file_xlsx()
 vernames = df.iloc[:, 0].tolist()

--вирізано--

❷ centrality_indices.to_excel('centrality_indices.xlsx', index=False)
 print("Показники центральності та степені вершин графа збережено у файлі centrality_indices.xlsx")

Зрештою, виведемо повідомлення про завершення роботи програми (**❶**).

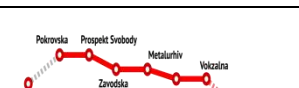
--вирізано--

```
centrality_indices.to_excel('centrality_indices.xlsx', index=False)
print("Показники центральності та степені вершин графа збережено у файлі centrality_indices.xlsx")
```

❶ print("\nРоботу завершено. Дякуємо за користування програмою!")

Збережемо файл програми GRAPH під іменем 'graph.py'.

Все зроблено – у такому вигляді програма відповідає нашому задуму та готова до роботи.

Схеми ліній метрополітену (2026)			
			
Відень	Київ	Харків	Дніпро

На рисунку представлено скріншот вікна програми MS Excel з матрицею суміжності станцій Харківського метрополітену.

[illegible]

Для максимально повного та ефективного аналізу мереж метрополітену проведемо його для двох різних графів – повного та генералізованого. У випадку повного графа будемо вважати вершинами всі станції та пересадкові вузли системи метрополітену. У випадку генералізованого графа вершинами визначаємо лише кінцеві станції ліній метрополітену та пересадкові вузли. Сенс такої генералізації в тому, щоб акцентувати найбільш загальні, однак функціонально важливі риси топології мережі, адже просте нарощування кількості станцій (наприклад, за рахунок зменшення відстаней між ними) збільшує, але принципово не ускладнює мережу з точки зору її функціонування та стійкості до потенційних збоїв.

Збережемо матриці суміжності вершин у файлі MS Excel ‘metro.xlsx’ на аркушах ‘adjvienna’, ‘adjkyiv’, ‘adjkharkiv’ та ‘adjdnipro’, відповідно, для повного графа, та ‘adjvienna_gen’, ‘adjkyiv_gen’, ‘adjkharkiv_gen’ та ‘adjdnipro_gen’ для генералізованого графа. ‘adj’ тут походить від англійського слова ‘adjacency’ – суміжність. Файл ‘metro.xlsx’ повинен знаходитись у тій самій директорії (теці) на комп’ютері, що і файл програми GRAPH ‘graph.py’.

Для порівняння нас цікавитимуть такі характеристики графів мереж метрополітену, як кількість вершин та ребер, кількість компонент зв’язності, діаметр, цикломатичні числа, міри зв’язності (α -індекс, β -індекс, ϕ -індекс), міра компактності (π -індекс). Також для кожної мережі метрополітену визначимо найбільш центральні та найбільш периферійні станції / пересадкові вузли, використовуючи сукупність показників центральності вершин.

Запустимо програму GRAPH та по черзі обробимо за її допомогою всі вісім матриць суміжності вершин – для метрополітенів Відня, Києва, Харкова та Дніпра, обираючи щоразу потрібну назву аркуша MS Excel. Матрицю відстаней та показники центральності вершин для зручності будемо зберігати у файли MS Excel лише для повних графів. Результати систематизовано у таблиці.

Показник	Мережа метрополітену			
	Відень	Київ	Харків	Дніпро
Характеристики повного графа				
Кількість вершин	98	48	27	6
Кількість ребер	104	48	27	5
Компоненти зв’язності	1	1	1	1
Діаметр	28	21	16	5
Цикломатичне число	7	1	1	0
α -індекс	0,04	0,01	0,02	0,0
β -індекс	1,06	1,00	1,00	0,83
ϕ -індекс	0,36	0,35	0,36	0,42
π -індекс	3,71	2,29	1,69	1,0
Найбільш центральні станції/ пересадкові вузли (вказано значення індексу відносної доступності)	Шведенплац (1,00) Карлсплац (0,99) Штефансплац (0,99)	вузол Площа Українських Героїв - Палац Спорту (1,00) вузол Хрещатик - Майдан Незалежності (0,98) вузол Театральна - Золоті Ворота (0,97)	вузол «Держпром-Університет» (1,00) вузол Майдан Конституції-Історичний музей (0,99) Архітектора Бекетова (0,94)	Заводська (1,00) Металургів (1,00)
Найбільш периферійні станції / пересадкові вузли (вказано значення індексу відносної доступності)	Зеештадт (0,38) Асперн Норд (0,40) Леопольдау (0,41)	Червоний Хутір (0,37) Бориспільська (0,40) Академмістечко (0,40)	Індустріальна (0,41) Тракторний Завод (0,46) Салтівська (0,46)	Покровська (0,60) Вокзальна (0,60)

Показник	Мережа метрополітену			
	Відень	Київ	Харків	Дніпро
Характеристики генералізованого графа				
Кількість вершин	19	9	6	2
Кількість ребер	24	9	7	1
Компоненти зв'язності	1	1	1	1
Діаметр	6	3	3	1
Цикломатичне число	7	1	1	0
α -індекс	0,21	0,08	0,14	0,0
β -індекс	1,26	1,00	1,17	0,5
ϕ -індекс	0,47	0,43	0,58	1,00
π -індекс	3,43	3,00	2,00	1,0

У повному представленні чітко простежується ієрархія систем за розміром: Відень (98 вершин) – найбільша та найбільш розгалужена мережа; Київ (48) і Харків (27) – середні за масштабом; Дніпро (6) – найменша система. У генералізованій моделі абсолютна кількість вершин зменшується, при цьому пропорції між числами загалом зберігаються: 19–9–6–2 вершин відповідно. Всі системи зв'язані – кількість компонент зв'язності в усіх випадках становить 1.

Водночас, проаналізовані системи метрополітену відрізняються за зв'язністю більш суттєво, ніж за розміром. Особливо це помітно стосовно насиченості мереж циклічними структурами. Цикломатичне число для всіх мереж однакове як для повної, так і генералізованої моделі: у Відні наявні 7 циклів, у Києві та Харкові – по 1 циклу, у Дніпрі – жодного циклу. Таким чином, метрополітен Відня пропонує велику кількість альтернативних маршрутів між парами віддалених станцій, у метрополітенах Києва та Харкові має місце обмежена варіативність руху, а мережа метрополітену Дніпра являє собою лінійну дендритну структуру без альтернативних маршрутів. Це підтверджується значеннями α -індексу, причому найвищий він у Відні, найменший у Дніпрі, а у Харкові більший ніж у Києві за рахунок меншої розгалуженості (дендритності) ліній метрополітену. Характерно, що після генералізації спостерігаємо різке зростання значень α -індексу.

Що стосується насиченості мереж метрополітену ребрами (ϕ -індекс), то тут, навпаки, на перше місце виходить метрополітен Дніпра за рахунок свого малого розміру, тоді як інші три метрополітени істотно відстають. Цікаво, що найкраща ситуація спостерігається у Харкові. Однак, за співвідношенням кількості ребер та вершин (β -індекс) для повного графа лідером стає метрополітен Відня, Київ та Харків мають тотожну ситуацію, а аутсайдером є метрополітен Дніпра; у випадку генералізованого графа попереду метрополітен Відня, метрополітен Харкова дещо випереджає метрополітен Києва, метрополітен Дніпра залишається на останній позиції.

Значення діаметра в цілому віддзеркалює розмір мережі, але не лінійно. Якщо за кількістю станцій система метрополітену Відня удвічі перевищує систему метрополітену Києва, то співвідношення діаметрів цих мереж (28 до 21) не настільки велике. Це опосередковано вказує на більшу компактність системи Віденського метрополітену, що підтверджується значенням π -індексу як для повних, так і генералізованих графів. За компактністю мережі метрополітену лідирує Відень, за ним слідує Київ та Харків, і замикає рейтинг Дніпро з найменш компактною мережею. Показовий момент: при переході від повного до генералізованого графа π -індекс для метрополітенів Києва та Харкова зростає, а для Відня – зменшується. Це говорить про те, що у Києві та Харкові загальна просторова структура пересадкових вузлів організована ефективніше, ніж лінійна структура станцій, а у Відні – навпаки.

Роль центральних вузлів у аналізованих системах метрополітену відіграють переважно пересадкові вузли. Зокрема, у Києві найбільш центральні вершини графа - це всі три наявні станом на 2026 рік пересадкові вузли між лініями. У Харкові це вузли «Держпром – Університет» та «Майдан Конституції – Історичний музей», а також станція Архітектора Бекетова, тоді як третій пересадковий вузол «Метробудівників – Спортивна» має більш периферійне положення. У метрополітені Відня найбільш центральне положення займають станції Шведенплац, Карлсплац та Штефансплац – усі є пересадковими вузлами між двома а то й трьома (Карлсплац) лініями, і всі розташовані на найстарішій лінії U1. Однак, у Відні, через значну насиченість мережі циклами, існують і периферійні пересадкові вузли. Дніпро з його лінійною системою метрополітену не має пересадкових вузлів, тож тут дві найбільш центральні станції (Заводська та Металургів) розташовані посередині єдиної лінії.

Що стосується найбільш периферійних станцій та ліній, то в Києві це південно-східний кінець зеленої лінії (Червоний Хутір та Бориспільська) та західний кінець червоної лінії (Академмістечко). У випадку Харківського метрополітену найбільш периферійні станції зосереджені на східних околицях міста – Індустріальна, Тракторний Завод (червона лінія), Салтівська (синя лінія). У Відні найбільш периферійні станції знаходяться на протилежному від центру міста березі Дунаю – Зеештадт, Асперн Норд (лінія U2) та Леопольдау (лінія U1).

Узагальнення з проведеного аналізу такі:

- Відень має найбільшу та найскладнішу мережу, що обумовлено як великою кількістю ліній та станцій, так і наявністю альтернативних маршрутів. Крім того, це найбільш компактна система. Це робить цю систему метрополітену найстійкішою з чотирьох проаналізованих.
- Київ та Харків мають помірно розвинені мережі метрополітену. Вони схожі як за розміром (кількістю ліній та станцій), так і за параметрами зв'язності (один цикл у центральній частині мережі, сформований трьома пересадковими вузлами). Система Київського метрополітену більш компактна (симетрична) порівняно з системою Харківського метрополітену. Концентрація пересадкових вузлів у центральній частині мережі посилює навантаження на них, а відсутність альтернативних циклів, окрім єдиного «трикутника» між лініями, створює підвищені ризики збоїв та втрати зв'язності мережі під час аварій.
- Дніпро має не просто найменшу за кількістю станцій, але й примітивну лінійну систему метрополітену. Завдяки своїм невеликим розмірам система формально демонструє непогану зв'язність, однак повна відсутність циклів зумовлює неможливість прокладання альтернативних маршрутів та мінімальну стійкість до збоїв функціонування. Оскільки плани розвитку Дніпровського метрополітену передбачають нарощування довжини єдиної наявної лінії, ця вразливість в середньостроковій перспективі збережеться.
- Порівняння двох моделей мережі показує: повний граф (всі станції – вершини графа) відображає фізичну структуру мережі, тоді як генералізований граф (лише кінцеві станції та пересадкові вузли – вершини графа) – її функціональну логіку. Аналіз генералізованого графа краще виявляє роль вузлів у функціонуванні мережі та дозволяє оцінити ефективність її організації. Водночас, аналіз повного графа більш інформативний в контексті оцінки центральності / периферійності окремих станцій, вузлів та ліній мережі.

ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

1. Змініть програму таким чином, щоб при завершенні всіх операцій вона не закривалась, а пропонувала користувачеві обробити наступний граф.
2. Якщо користувач збереже матрицю відстаней та міри центральності і степені вершин графа у файли MS Excel, при наступному такому збереженні ці файли перезапишуться, і попередні результати буде втрачено. Для уникнення такого сценарію потрібно не забувати одразу перейменовувати вихідні файли або переміщувати їх в іншу директорію. Відредагуйте код програми таким чином, щоб користувач міг одразу у консолі обирати власні унікальні імена для кожного вихідного файлу.
3. Користувач може помилитись при заповненні матриці суміжності вершин графа. Додайте в програму можливість найпростішої перевірки правильності заповнення матриці? Які параметри при цьому потрібно тестувати?
4. Користувач може захотіти зберегти характеристики графа, такі як кількість вершин, ребер, цикломатичне число, діаметр тощо. Вдоскональте програму таким чином, щоб за бажанням користувача відповідні характеристики можна було зберегти у зрозумілому та зручному для користування вигляді у текстовий файл (*.txt).
5. Збережена у файл MS Excel матриця відстаней між вершинами графа містить у першому рядку індекси стовпців, а стовпця з назвами чи номерами вершин графа у ній немає взагалі. Доопрацюйте код програми, щоб збережена матриця одразу мала більш зручний для користувача формат?
6. Перемістіть спеціалізовані функції, що визначають окремі параметри графа чи його вершин, у окремий модуль (файл *.py). Імпортуйте цей модуль або окремі його функції у файл програми graph.py. Перевірте коректність роботи програми в цьому випадку.
- 7*. Як потрібно змінити програму, щоб вона визначала (а) перелік вершин, які належать до кожного компонента зв'язності, та (б) діаметр та показники центральності вершин для кожного окремого компонента зв'язності графа? Спробуйте реалізувати таку вдосконалену версію програми.
- 8*. Змініть код програми, щоб за її допомогою можна було формувати перелік елементарних циклів графа з назвами вершин, які входять до складу кожного такого циклу.
- 9*. Доопрацюйте програму таким чином, щоб за її допомогою можна було визначати кількість остовів графа, кількість елементарних циклів у кожному остові, а також клас топологічної складності мережі – найбільшу кількість топологічних ярусів остова серед наявних остовів.
- 10*. Якщо вмієте працювати з об'єктно-орієнтованою парадигмою програмування на Python – спробуйте написати аналогічну програму з використанням власних класів об'єктів.

Примітка: зірочкою (*) позначено завдання підвищеної складності

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

Наведений перелік літератури буде корисним для як для бажаючих вивчати програмування на Python з нуля, так і для тих, хто хоче істотно поглибити володіння Python з метою вирішення прикладних задач за професійною діяльністю.

1. Беррі П. Python. Легкий для сприйняття довідник. Харків: Фабула 2021. 624 с.
2. Васильєв О. М. Програмування в PYTHON. Теорія і практика. Ліра-К, 2023. 462 с.
3. Зеленьяк О. Програмування мовою Python. Алгоритмічні структури і стратегії. Теорія та практика. Київ: Ліра-К, 2025. 338 с.
4. Копей В. Б. Мова програмування Python для інженерів і науковців: навчальний посібник. Івано-Франківськ: ІФНТУНГ, 2019. 272 с.
5. Лутц М. Python. Довідник програміста. Київ: Науковий світ, 2024. 294 с.
6. Маттес Е. Пришвидшений курс Python. Практичний, проєктно-орієнтований вступ до програмування. Перекл. з англ. Ольги Белової. Львів: Видавництво Старого Лева. 2021. 600 с.
7. Яковенко А. В. Основи програмування. Python. Частина 1: підручник для студ. спеціальності 122 «Комп'ютерні науки», спеціалізації «Інформаційні технології в біології та медицині». [Електронний ресурс]. Київ: КПІ ім. Ігоря Сікорського, 2018. 195 с.
8. McClain B. P. Python for Geospatial Data Analysis. Sebastopol, CA: Gravenstein Highway North, O'Reilly Media, Inc., 2023. 262 p.
9. Norton D. Python for Data Science: Master Python Programming for Effective Data Analysis, Visualization, and Machine Learning. Independently published (March 26, 2024). 147 p.

Джерела по застосуванню теорії графів в суспільній географії та урбаністиці

1. Байтеряков О., Воронова В. Топологічний аналіз автомобільної транспортної мережі Запорізької області. Вісник Харківського національного університету імені В. Н. Каразіна, серія «Геологія. Географія. Екологія». 2024. Вип. 60. С. 137-158.
2. Грицевич В. С., Ванда І. В. Математичні методи в суспільній географії: навчальний посібник-практикум для самостійної роботи студентів. Львів: Видавничий центр ЛНУ імені Івана Франка, 2015. 96 с.
3. Колотуха І. О. Історико-географічний аналіз зв'язності житлових районів міста Києва легкорейковим транспортом у XX-XXI сторіччі. Економічна та соціальна географія. 2016. Вип. 76. С. 27-31.
4. Колотуха О., Колотуха І. Еволюція функціонально-територіальної структури рейкового громадського транспорту Києва. Економічна та соціальна географія. 2023. Вип. 90. С. 47-60.
5. Методи суспільно-географічних досліджень: навчально-методичне видання / за ред. К. Мезенцева. Київ: Видавництво «Фенікс», 2021. С. 80-82.
6. Сергєєв І., Гнатюк О. Оцінка доступності території міста Чернігова громадським транспортом. Соціально-економічні особливості та проблеми сучасного розвитку Чернігівської області: матеріали третьої Всеукраїнської науково-практичної конференції (м. Ніжин, 13-14 лютого 2025 року). Ніжин: НДУ імені Миколи Гоголя, 2025. С. 10-13.
7. Yang Z., Zhou H., Gao P., Chen H., Zhang N. The topological analysis of urban transit system as a small-world network. International Journal of Computational Intelligence Systems. 2011. Vol. 4. No. 6. P. 1216-1223.
8. Żochowska R., Soczówka P. Analysis of selected transportation network structures based on graph measures. Scientific Journal of Silesian University of Technology. Series Transport. 2018. Vol. 98. P. 223-233.