М. В. ПЕТРИЧКО
С. Д. ШТОВБА
О. М. КОЗАЧКО

ДИСКРЕТНА МАТЕМАТИКА

ДЛЯ ПРОГРАМІСТІВ

Міністерство освіти і науки України
Вінницький національний технічний університет

Петричко М. В., Штовба С. Д., Козачко О. М.

ДИСКРЕТНА МАТЕМАТИКА ДЛЯ ПРОГРАМІСТІВ

Електронний навчальний посібник

Вінниця
ВНТУ
2026

УДК 519.854
П30

Рекомендовано до видання Вченою Радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 5 від 27.11.2025).

Рецензенти:
Квєтний Р. Н., член-кор. НАПН України, професор, д-р техн. наук;
Лужецький В. А., професор, д-р техн. наук;
Бойко О. Р., канд. техн. наук.

Петричко, М. В.

П30 Дискретна математика для програмістів : навчальний посібник [Електронний ресурс] / Петричко М. В., Штовба С. Д., Козачко О. М. – Вінниця : ВНТУ, 2026. – (PDF, 120 с.)
ISBN 978-617-8163-78-5 (PDF)

Наведено теоретичні відомості, практичні поради та завдання для самостійного дослідження із дисципліни «Дискретна математика» для здобувачів вищої освіти, які навчаються за усіма спеціальностями галузі знань «Інформаційні технології». Усі дослідницькі завдання передбачають виконання обчислювальних експериментів. Більшість експериментів проводиться або на реальних даних з інформаційних мереж, або на загальноприйнятих в предметній області тестових задачах (benchmark problems). Особливістю посібника є використання завдань на двох рівнях – базовому та поглибленому. Це дозволяє здобувачам вищої освіти реалізовувати індивідуальну траєкторію навчання в межах однієї дисципліни шляхом самостійного вибору додаткових завдань. Дослідницькі завдання поглибленого рівня можуть використовуватися під час викладання споріднених дисциплін, пов’язаних з комбінаторною оптимізацією, дослідженням операцій та моделюванням інформаційних систем.

УДК 519.854

ISBN 978-617-8163-78-5 5 (PDF)

ЗМІСТ

ПЕРЕДМОВА

Навчальний посібник призначено для здобувачів вищої освіти перших курсів бакалаврату усіх спеціальностей галузі знань «Інформаційні технології» для вивчення дискретної математики та споріднених дисциплін. Посібник створено з урахуванням актуальних потреб майбутніх фахівців з комп'ютерних наук, інформаційних технологій, системного аналізу та розробки програмного забезпечення. Завдання для самостійного виконання закріплюють теоретичні знання через практичну реалізацію задач, що мають як навчальне, так і фахове наповнення. Особлива увага приділяється моментам, які важливі в інформатиці, теорії алгоритмів, штучному інтелекті, логістиці та дискретній оптимізації. Усі завдання передбачають виконання обчислювальних експериментів. Більшість експериментів проводиться або на реальних даних з інформаційних мереж, або на загальноприйнятих в предметній області тестових задачах (benchmark problems). За дидактичної доцільності наведені у посібнику завдання можна легко трансформувати у лабораторні роботи.

Особливістю посібника є подання завдань на двох рівнях – базовому та поглибленому. Це дозволяє здобувачам вищої освіти реалізовувати індивідуальну траєкторію навчання в межах однієї дисципліни шляхом самостійного вибору додаткових завдань. Трудомісткість ґрунтовного виконання усієї сукупності завдань базового рівня в межах одного розділу становить 7–15 академічних годин. Залежно від особливостей освітньої програми за різними спеціальностями галузі знань «Інформаційні технології», гарант може вибрати ту чи іншу підмножину завдань. Завдання поглибленого рівня можна використовувати під час викладання споріднених дисциплін, пов'язаних з комбінаторною оптимізацією, дослідженням операцій та моделюванням інформаційних систем на старших курсах бакалаврату або в магістратурі. Таку структуру посібника сформовано авторами з метою формування у здобувачів вищої освіти цілісного уявлення про міждисциплінарні зв'язки дискретної математики та її ролі в майбутній фаховій діяльності.

Посібник містить як теоретичні відомості, так і перевірені на практиці поради, рекомендації та покрокові інструкції виконання завдань, що надає змогу здобувачам вищої освіти сформувати навички осмисленого застосування та програмної реалізації дискретних структур і алгоритмів.

За результатами виконання завдань здобувачі вищої освіти мають підготувати звіт. Обов'язковими структурними елементами звіту є постановка задачі, протокол її вирішення та висновки. Протокол виконання завдання має містити викладені у логічній послідовності власні інформаційні

матеріали – лістинги програм, результати моделювання у формі графіків, діаграм та таблиць. Кожен результат моделювання має бути прокоментований або за ним має бути сформульований проміжний висновок. Загальні висновки мають надавати вичерпну інформацію щодо аналізованого дослідницького питання.

Більшість матеріалу посібника апробовано авторами під час багаторічного викладання дискретної математики, дослідження операцій, дискретної оптимізації та моделювання інформаційних систем у Вінницькому національному технічному університеті та у Донецькому національному університеті імені Василя Стуса. Водночас, деякі завдання не оприлюднювалися; вони створені авторами з нуля під час роботи над цим посібником. Формуючи перелік завдань, автори також спиралися на власний досвід наукових досліджень та практичної розробки комерційних інформаційних систем. Автори будуть вдячні за зворотний зв'язок – знайдені помилки та мимодруки, зауваження, поради та рекомендації щодо змісту посібника та їх релевантності сучасним вимогам до підготовки айтішників.

Автори намагалися викладати матеріал посібника просто і доступно з урахуванням особливостей сприйняття інформації цільовою аудиторією – другорядні математичні викладки мінімізовано, водночас акцентовано увагу на прикладах з інформаційних систем. Під час підготовки матеріалу посібника використовувалася така література:

для розділу 1 – [1-4, 26];

для розділу 2 – [5, 13, 19, 23-26];

для розділу 3 – [5-8, 13, 15-17, 19-22];

для розділу 4 – [9-12, 18];

для розділу 5 – [14, 25, 27-34].

Ця література рекомендована для поглибленого вивчення курсу.

Автори над усіма розділами посібника працювали спільно. Загальний обсяг посібника становить 6.4 авторських аркушів. Розподіл внеску такий: Петричко М.В. – 2.2 авторських аркушів; Штовба С.Д. – 2.2 авторських аркушів; Козачко О.М. – 2 авторські аркуші.

РОЗДІЛ 1 МНОЖИНИ, ВІДНОШЕННЯ ТА ФУНКЦІЇ

1.1 Множини

Множина (set) – це сукупність об'єктів або елементів, які мають спільну властивість. Елементи множини є унікальними, тобто один і той самий елемент присутній у множині лише в одному екземплярі. У дискретній математиці множини широко використовуються для подання груп об'єктів, над якими виконуються операції як-от об'єднання, перетин, різниця та доповнення. Множини зазвичай позначають великими літерами латинського алфавіту. Сукупність елементів множини записують у фігурних дужках. Наприклад, множина простих чисел від 0 до 9 є такою: $A = \{1, 2, 3, 5, 7\}$. Порядок запису елементів у множині неважливий.

Над множинами можна виконувати різні операції. Найбільш поширені з них – об'єднання (union), перетин (intersection) та різниця (differenece).

Об'єднання двох множин A та B – це бінарна операція, яка утворює нову множину, що містить усі елементи кожної із них:

$$A \cup B = \{x \mid x \in A \text{ або } x \in B\}.$$

Примітка: символ | вказує на те, що за ним буде опис властивості елемента, який передує цьому символу.

Перетин двох множин A та B – це бінарна операція, яка утворює нову множину, що містить лише ті елементи, які належать і множині A, і множині B одночасно:

$$A \cap B = \{x \mid x \in A \text{ та } x \in B\}.$$

Різниця двох множин – це бінарна операція, яка утворює нову множину, що містить лише ті елементи першої множини, яких немає у другій множині:

$$A \setminus B = \{x \mid x \in A \text{ та } x \notin B\}.$$

Приклад. Нехай $A = \{1, 2, 4, 6, 7\}$ та $B = \{1, 3, 4, 8\}$. Тоді:

$A \cup B = \{1, 2, 4, 6, 7, 3, 8\}$;

$A \cap B = \{1, 4\}$;

$A \setminus B = \{2, 6, 7\}$.

Найбільш поширені множини, що використовуються в математиці:

$Z = \{..., -3, -2, -1, 0, 1, 2, 3, ...\}$ – множина цілих чисел;

$N = \{0, 1, 2, 3, ...\}$ – множина натуральних чисел;

R – множина дійсних чисел;

Q – множина раціональних чисел;

$\varnothing$ – пуста множина.

Кількість елементів множини називають її *потужністю* (cardinality), яку позначають $|A|$.

1.2 Відношення

Відношення (relation) $\mathbf{R}$ між множинами A і B – це будь-яка підмножина декартового добутку $A \times B$. Тобто:

$$\mathbf{R} \subseteq A \times B.$$

Декартовий добуток (Cartesian product) $A \times B$ являє собою сукупність усіх пар, перший елемент яких взято з множини *A*, а другий – з множини *B*.

Відношення позначає певний зв'язок між об'єктами. Наприклад, відношення $<$ на множині натуральних чисел містить у собі усі пари чисел, для яких: $\mathbf{R} = \{(a,b) \mid a<b \text{ та } a \in N \text{ та } b \in N\}$. Відношення може бути задано як між елементами однієї множини – $\mathbf{R} \subseteq A \times A$, так і між елементами різних множин $\mathbf{R} \subseteq A \times B$. Відношення, що задано на декартовому добутку двох множин, називається бінарним. Відношення на декартовому добутку трьох множин називається тернарним. Відношення на декартовому добутку *n* множин називається *n*-арним. Далі розглядатимємо лише бінарні відношення.

Відношення можуть мати такі властивості як-от: рефлексивність, антирефлексивність, симетричність, антисиметричність, асиметричність, транзитивність тощо.

Відношення $\mathbf{R}$ є *рефлексивним*, якщо $\forall a \in A \rightarrow (a,a) \in \mathbf{R}$. Наприклад, відношення $\geq$ є рефлексивним, а відношення $>$ не є рефлексивним.

Відношення $\mathbf{R}$ є *антирефлексивним*, якщо $\forall a \in A \rightarrow (a,a) \notin \mathbf{R}$. Наприклад, відношення $>$ є антирефлексивним, а відношення $\geq$ не є антирефлексивним.

Відношення $\mathbf{R}$ є *симетричним*, якщо $\forall (a,b) \in \mathbf{R} \rightarrow (b,a) \in \mathbf{R}$. Наприклад, відношення «родичі» є симетричним, оскільки a є родичем b тоді і тільки тоді, коли b є родичем a.

Відношення $\mathbf{R}$ є *антисиметричним*, якщо $(a,b) \in \mathbf{R}$ та $(b,a) \in \mathbf{R}$ лише тоді коли $a = b$. Наприклад, відношення $\geq$ є антисиметричним.

Відношення $\mathbf{R}$ є *асиметричним*, якщо $(a,b) \in \mathbf{R}$ та $(b,a) \notin \mathbf{R}$. Наприклад, відношення $>$ є асиметричним.

Відношення $\mathbf{R}$ є *транзитивним*, якщо для усіх трійок $a, b, c,$ з належності двох пар до відношення – $(a,b)\in\mathbf{R}$ та $(b,c)\in\mathbf{R}$ випливає належність до відношення і пари (a, b):

$$\big((a,b)\in\mathbf{R} \text{ та } (a,b)\in\mathbf{R}\big)\to(a,c)\in\mathbf{R}.$$

Прикладом транзитивного відношення є «вище», «холодніше», «швидше». В той самий час, відношення «сильніше» стосовно спортивних гравців не є транзитивним: a може перемогти b, b – перемогти c, а c – перемогти a.

Одне відношення може мати кілька згаданих властивостей. Наприклад, відношення еквівалентності є симетричним, рефлексивним та транзитивним.

Над відношеннями виконують теоретико-множинні операції, зокрема об'єднання, перетин та доповнення, аналогічно до того, як це робиться для множин. Але для відношень є і специфічні операції. Однією із найбільш важливих з них є композиція відношень, яка описує транзитивні зв'язки. Композиція позначається символом $\circ$. Розглянемо відношення $\mathbf{R}_1\subseteq A\times B$ та $\mathbf{R}_2\subseteq B\times C$. Тоді композиція цих відношень $\mathbf{R}=\mathbf{R}_1\circ\mathbf{R}_2$ буде задана на декартовому добутку $A\times C$. Отже, множина B є своєрідним посередником, через елементи якої поширюються зв'язки множини A на множину B.

Формально, *композиція відношень* $\mathbf{R}_1\subseteq A\times B$ та $\mathbf{R}_2\subseteq B\times C$ записується так:

$$\mathbf{R}_1\circ\mathbf{R}_2=\big\{(a,\ c)\ \big|\ (a,\ b)\in\mathbf{R}_1 \text{ та } (b,\ c)\in\mathbf{R}_2\big\}.$$

Приклад. Нехай: $\mathbf{R}_1=\{(a_1,b_1),(a_1,b_2),(a_2,b_2),(a_3,b_1),(a_4,b_3)\}$ та $\mathbf{R}_2=\{(b_1,c_1),(b_2,c_2),(b_2,c_3)\}$.

Тоді, $\mathbf{R}=\mathbf{R}_1\circ\mathbf{R}_2=\{(a_1,c_1),(a_1,c_2),(a_1,c_3),(a_2,c_2),(a_2,c_3),(a_3,c_1)\}$.

Графічне зображення композиції відношень за цим прикладом наведено на рис. 1.1. З нього видно, що множина $\{b_1,b_2,b_3\}$ виступила проміжним шаром між множинами $\{a_1,a_2,a_3,a_4\}$ та $\{c_1,c_2,c_3\}$.

Над відношеннями зручно виконувати операції, якщо їх подавати в матричному вигляді. Розміри таких матриць визначаються декартовим добутком множин, на яких задано відношення. В комірці матриці записується 1, якщо між відповідними елементами множин відношення виконується, і 0, якщо відношення не виконується. Наприклад, матриці відношень з рис. 1.1 є такими:

$$\mathbf{R}_1 = \begin{array}{c} \begin{matrix} b_1 & b_2 & b_3 \end{matrix} \\ \begin{Vmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{Vmatrix} \end{array} \begin{matrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{matrix};$$

$$\mathbf{R}_2 = \begin{array}{c} \begin{matrix} c_1 & c_2 & c_3 \end{matrix} \\ \begin{Vmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{Vmatrix} \end{array} \begin{matrix} b_1 \\ b_2 \\ b_3 \end{matrix};$$

$$\mathbf{R} = \begin{array}{c} \begin{matrix} c_1 & c_2 & c_3 \end{matrix} \\ \begin{Vmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{Vmatrix} \end{array} \begin{matrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{matrix}.$$

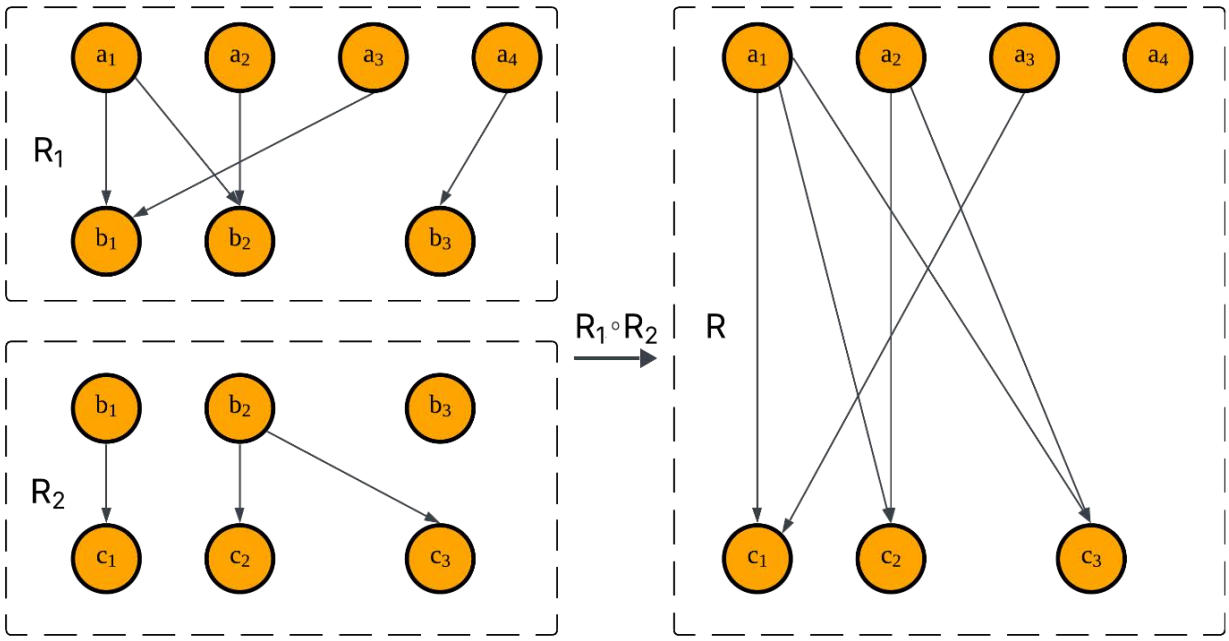


Рисунок 1.1 – До прикладу виконання композиції відношень

Композиція відношень в матричній формі реалізується за схемою множення матриць, у цьому разі добуток пар елементів замінюється на операцію мінімума, а додавання – на операцію максимума. Таку композицію називають макс-мінним добутком відношень.

Якщо бінарне відношення задано на декартовому добутку двох однакових множин – $\mathbf{R} \subseteq A \times A$, тоді можна зробити композицію $\mathbf{R} \circ \mathbf{R}$, тобто піднести відношення до другого степеня: $\mathbf{R}^2 = \mathbf{R} \circ \mathbf{R}$. Відношення $\mathbf{R}^2$

описує зв'язок між елементами множини A через один транзитний елемент. Якщо A – це множина людей, а $\mathbf{R}$ – відношення знайомства, тоді $\mathbf{R}^2$ – це знайомство через одного посередника, через одне рукостискання. Відношення $\mathbf{R}^3 = \mathbf{R}^2 \circ \mathbf{R}$ – це зв'язок через двох посередників, через два рукостискання. Щоб виявити усі можливі зв'язки між елементами множини, потрібно об'єднати початкове відношення з відношеннями через одного, двох, трьох … посередників. Така операція називається *транзитивним замиканням відношення*. Вона записується так:

$$\mathbf{T} = \mathbf{R} \cup \mathbf{R}^2 \cup \mathbf{R}^3 \cup \mathbf{R}^4 \cup \ldots.$$

1.3 Функції

Функцією (function) f між множинами X та Y називається правило, за яким кожному елементу $x \in X$ ставиться у відповідність єдиний елемент $y \in Y$. Функція може бути визначена у термінах відношення, і насправді є бінарним відношенням з певними умовами.

Формально функція записується так: $f : X \to Y$. Якщо елемент $y \in Y$ відповідає елементу $x \in X$, то говорять, що функція f відображає x в y: $y = f(x)$. В такому разі x називають аргументом функції *f*, а *y* – значенням функції *f*.

Область визначення (домен) функції f – множина всіх елементів X, для яких функція *f* є визначеною.

Область значень (кодомен) функції *f* – це множина *Y*, до якої належать усі можливі значення функції *f*.

Множина значень функції f (або образ X) – множина всіх можливих значень $y \in Y$, коли $x \in X$.

Якщо $A \subseteq X$, то образ множини A для функції f – це множина всіх значень функції для елементів множини A:

$$f(A) = \{f(x) \mid x \in A\} \subseteq Y.$$

Нехай $f(x) = x^2, X = R$. Розглянемо множину $A = \{-2, 1, 3\}$. Її образ становить $f(A) = \{4, 1, 9\}$.

Прообраз множини $B \subseteq Y$ – це множина всіх елементів X, які відображаються в B функцією f: $f^{-1}(B) = \{x \in X \mid f(x) \in B\}$. Нехай $f(x) = x^2$ і $B = \{4\}$. Тоді прообраз $f^{-1}(B) = \{-2, 2\}$, оскільки $(-2)^2 = 4$ та $2^2 = 4$.

Функції за результатом перетворення вхідних аргументів у образ класифікують на ін'єктивні та сюр'єктивні.

Функція $f: X \to Y$ є *ін'єктивною*, якщо різним елементам області визначення X відповідають різні елементи множини визначення Y: $f(a) = f(b) \to a = b$.

Функція $f: X \to Y$ є *сюр'єктивною*, якщо кожен елемент множини визначення Y має хоча б один прообраз у X: $\forall y \in Y,\ \exists x \in X,\ y = f(x)$.

Функція може бути одночасно ін'єктивною та сюр'єктивною, неін'єктивною та несюр'єктивною, ін'єктивною та несюр'єктивною а також неін'єктивною та сюр'єктивною.

Функція може мати як один аргумент, так і багато аргументів. Функції можуть утворювати інші функції, тобто значення однієї функції можуть бути аргументами для іншої функції. Така комбінація функцій називається *суперпозицією*.

Елементарною функцією називають функції, які утворені скінченною кількістю суперпозицій арифметичних операцій (додавання, віднімання, множення та ділення), експонети, логарифму, синусу та арксинусу. З елементарними функціями пов'язана важлива для комп'ютерних наук *XIII проблема Гільберта*. Суть цієї проблеми полягає у встановленні факту чи можна за скінченного числа елементарних функцій від двох аргументів реалізувати довільну неперервну функцію від багатьох аргументів. В середині минулого сторіччя уродженець Одеси Володимир Арнольд в двадцятирічному віці довів, що так, це можна зробити. Трохи згодом було встановлено, що для цього потрібна лише операція додавання. Таке вирішення XIII проблеми Гільберта можна розглядати як теоретичне обгрунтування реалізації двохоперандної архітектури комп'ютерного процесора. Також деякою мірою вирішення XIII проблеми Гільберта є теоретичним підгрунтям і МГУА – методу групового урахування аргументів Олексія Івахненка та дотичної до нього архітектури нейронних мереж глибокого навчання.

1.4 Хеш-функції

Хеш-функція – це функція, яка перетворює вхідне значення у числовий хеш-код. *Хешування* (hashing) – це перетворення вхідного масиву даних довільної довжини у вихідний бітовий рядок фіксованої довжини. Хешування є ефективним з погляду обчислень. Замість того, щоб переглядати кожен елемент списку (як у випадку з лінійним пошуком) або переходити по вузлах дерева (як за пошуку по бінарному дереву), хеш-функція миттєво обчислює індекс, за яким має зберігатися або знаходитися елемент. Це забезпечує середню обчислювальну складність доступу $O(1)$, тобто постійний час незалежно від кількості елементів.

Математично хеш-функція позначається так:

$$h : U \to \{0,\ 1,\ 2,\ 3, ...,\ m-1\},$$

де U – універсальна множина можливих вхідних значень (ключів);

m – розмір хеш-таблиці (кількість слотів);

$h(x)$ – хеш-функція, яка відображає вхідний ключ $x \in U$ у певний індекс в межах множини $\{0,\ 1,\ 2,\ 3, ...,\ m-1\}$.

Оскільки область визначення хеш-функції значно більша за область значень, то виникають моменти, коли різні вхідні значення хешуються в один і той самий індекс. Така ситуація називається *колізією*. Для пояснення механізму колізій часто використовують принцип Діріхле, який також називають *принципом голубів і кліток* (pigeonhole principle). Якщо голубів більше ніж кліток, а кожного голуба потрібно помістити у клітку, тоді в деяких клітках буде більше одного голуба. Математично це позначається таким чином. Якщо задана деяка функція $f : X \to Y$ та $|X| > |Y|$, тоді існують елементи $x_1, x_2 \in X$ такі, що $f(x_1) = f(x_2)$, коли $x_1 \neq x_2$ (рис. 1.2).

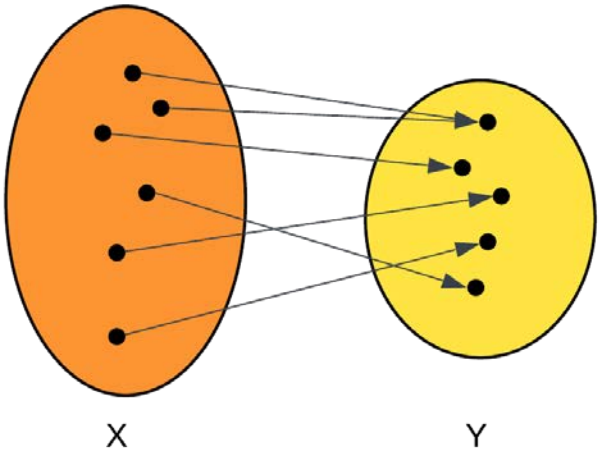


Рисунок 1.2 – Ілюстрація принципу Діріхле

За $|X| < |Y|$ може здаватись, що колізія неможлива. Проте це не так. Для прикладу розглянемо парадокс днів народження. Цей парадокс стосується оцінювання ймовірності того, що у випадковій групі людей дні народження деяких людей будуть повторюватись. Виявляється, що якщо у групи є понад 22 людини, тоді ймовірність повторення днів народження перевищує 0.5. Якщо є 57 людей, то така ймовірність становить понад 0.99. Отже, навіть якщо область значення хеш-функції значно більша за область визначення, це не гарантує відсутність колізій.

Різновиди хеш-функцій

Розглянемо такі найпростіші хеш-функції, як-от: просте модульне хешування, хешування за методом множення та хеш-функція Бернштейна.

Просте модульне хешування записується так:

$$h(x) = x \bmod m,$$

де m – розмір хеш-таблиці, $x \in N$;

mod – остача від ділення.

Модальне хешування є швидким, але може створювати багато колізій.

Хешування за методом множення використовує множення числа на деяку константу A, яка зазвичай дорівнює золотій пропорції 0.6180339887, що дозволяє розподілити значення рівномірніше:

$$h(x) = \lfloor m((x \cdot A) \bmod 1) \rfloor,$$

де $\lfloor\ \rfloor$ – округлення до найближчого меншого цілого – відкидання дробової частки.

Хеш-функція Бернштейна:

$$h(-1,\ c) = 5381;$$

$$h(x,\ c) = (33h(x-1,\ c) + c_x) \bmod m,$$

де c – вхідний текстовий рядок для якого необхідно знайти хеш-код;

c_x – числове подання символу текстового рядка за номером x, $x \in \{0,\ 1,\ 2,\ ...,\ n\}$;

n – кількість символів у c.

1.5 Програмна реалізація множини

Множина в програмуванні – це невпорядкована структура даних, яка містить лише унікальні елементи. У програмуванні множини застосовуються для швидкого пошуку, фільтрації унікальних значень та оптимізації алгоритмів.

У різних мовах програмування множини реалізують різними способами, зокрема з використанням:

1) масивів або списків із перевіркою на унікальність;

2) деревоподібних структур, наприклад, бінарного дерева пошуку;

3) хеш-таблиць для швидкого доступу до елементів.

У більшості мов програмування множина подана як вбудований тип, який використовує хеш-таблицю для ефективного зберігання та доступу до елементів.

Хеш-таблиця – це структура даних, що використовує хеш-функцію для швидкого доступу до значень. Схематично хеш-таблицю подано на рис. 1.3. У ній за назвою предмета зберігається відповідна кількість здобувачів. Хеш-функція перетворює ключ, пов'язаний з даними, у хеш-код для індексації хеш-таблиці. Коли елемент потрібно додати до таблиці, хеш-код може індексувати порожній слот, і в цьому випадку елемент просто додається. Якщо хеш-код індексує заповнений слот, тоді потрібне певне вирішення колізій: новий елемент може бути пропущено (не додано до таблиці) або замінено старим елементом, або додано до таблиці в іншому місці за допомогою визначеної певної процедури. Ця процедура залежить від структури хеш-таблиці.

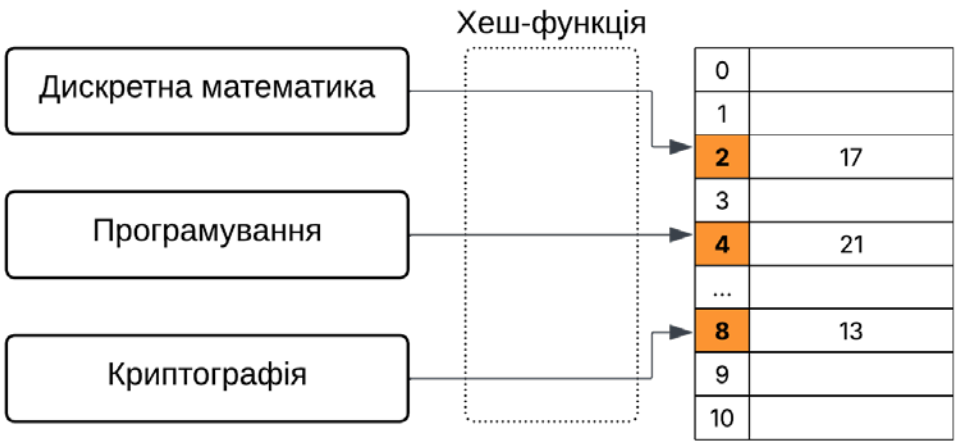


Рисунок 1.3 – Хеш-таблиця, в якій за назвою предмета зберігається кількість здобувачів, що його вивчають

Хеш-таблиця складається з таких трьох компонентів:

1) масиву або списку комірок (buckets), де зберігаються значення;

2) хеш-функції, що визначає, в яку комірку потрапить значення;

3) методу розв'язання колізій, наприклад, ланцюгове хешування або відкрита адресація.

Хеш-функція під час використання у хеш-таблиці виконує 3 таких завдання:

1) перетворення ключів змінної довжини на значення фіксованої довжини, зазвичай, довжиною не більше машинного слова;

2) перемішування бітів ключа так, щоб отримані значення були рівномірно розподілені по простору ключів;

3) відображення значення ключів в значення, що не перевищують розмір таблиці.

Для ефективної реалізації хеш-таблиці до хеш-функції висуваються такі вимоги:

1) швидкість обчислень;
2) мінімізація колізій;
3) рівномірний розподіл значень.

Розмір хеш-таблиці сильно впливає на швидкість роботи з хеш-таблицею та на ефективне використання ресурсів. Якщо в хеш-таблицю додається велика кількість елементів, тоді потрібно збільшити розмір хеш-таблиці, щоб уникнути колізій і таким чином пришвидшити операції над хеш-таблицею. Якщо кількість елементів зменшується, тоді необхідно зменшити розмір хеш-таблиці для економії пам'яті. Ця процедура називається *динамічним розширенням та стисненням хеш-таблиці*. Процедура відносно проста. Коли кількість елементів перевищує певний поріг, наприклад, 75% заповненості, розмір таблиці збільшується, зазвичай, удвічі, і всі елементи повторно хешуються. Якщо використовується менше певного відсотка комірок, наприклад, менше 25%, тоді таблицю можна зменшити для економії пам'яті. Для оцінювання рівня заповненості хеш-таблиці використовують *коефіцієнт заповнення* (load factor):

$$\alpha = \frac{n}{m},$$

де n – кількість доданих елементів в таблицю;

m – кількість комірок в хеш-таблиці.

Продуктивність хеш-таблиці погіршується залежно від коефіцієнта заповнення α. Його значення зазвичай тримають нижче певного $\alpha_{\max}$. Це гарантує продуктивність. Загальний підхід містить у собі повторне хешування хеш-таблиці, коли коефіцієнт заповнення α досягає $\alpha_{\max}$. Аналогічно, таблиця зменшується, коли коефіцієнт заповнення спадає до рівня $0.25\alpha_{\max}$.

Важливим моментом під час хешування є *обробка колізій*. Її можна реалізувати за допомогою ланцюгового хешування. В цьому випадку, кожна комірка хеш-таблиці є вказівником на ланцюжок пар «ключ – значення», відповідних одному і тому самому хеш-значенню ключа. Колізії призводять до того, що з'являються ланцюжки довжиною понад 1 елемент. Операції пошуку або видалення елемента потребують перегляду всіх елементів відповідного ланцюжка – потрібно знайти в ньому елемент з заданим ключем. Під час додавання нового елемента необхідно додати елемент в кінець або у початок відповідного ланцюжка. Якщо коефіцієнт заповнення стає занадто великим, тоді необхідно збільшити розмір ланцюжка комірок і перебудувати таблицю. Схематично це зображено на рис. 1.4, де утворилася колізія з хеш-кодом 2 – відповідна комірка складається зі зв'язаного списку з двох елементів, які і утворили колізію.

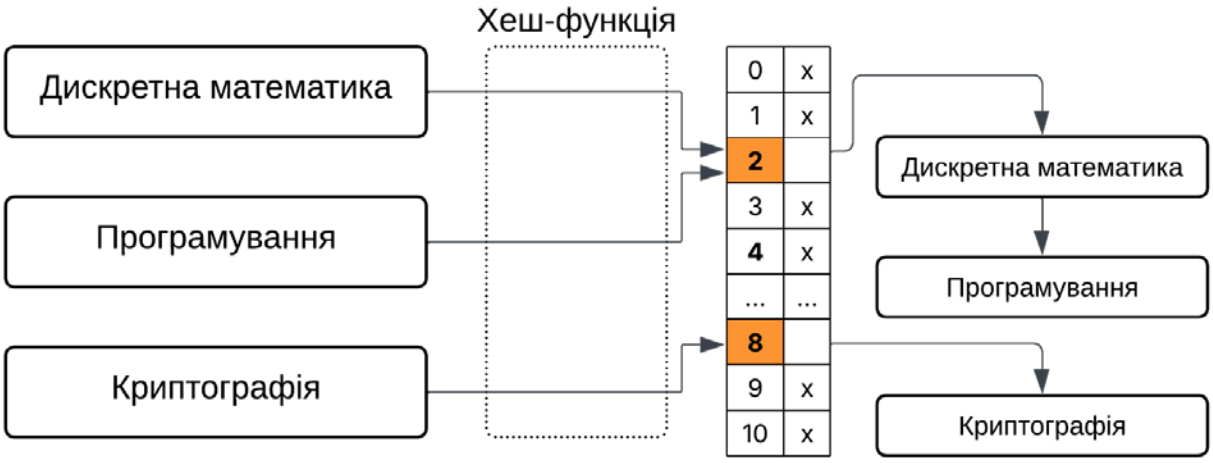


Рисунок 1.4 – Ілюстрація ланцюгового хешування

1.6 Завдання для самостійного дослідження

Базовий рівень

1. Реалізувати власний клас множини, використовуючи хеш-таблицю для збереження елементів. Тип даних, який можна зберігати в множині – текстовий рядок. Як хеш-функцію вибрати просте модульне хешування. Для динамічного розширення та стискання використати ланцюгове хешування. Розмір після розширення та стискання – вдвічі більший та вдвічі менший за поточний.

2. Реалізувати операції об’єднання, перетину, різниці множин, видалення елемента з множини та перевірки належності елемента множині.

3. Виконати тестування реалізованих операцій. Перевірити коректність роботи множини на тестових прикладах. Переконатись, що під час додавання в множину додаються лише унікальні елементи і повторів немає.

4. Порахувати кількість колізій хешування за різної кількості доданих елементів. Експерименти провести для 10, 100 та 1000 елементів.

Поглиблений рівень

1. Встановити, як залежить кількість колізій від типу хеш-функції. Для цього необхідно провести серію експериментів для таких хеш-функцій, як-от: просте модульне хешування, хешування за методом множення та з використанням хеш-функції Бернштейна. В кожному експерименті використовується одна і та сама множина. До цієї множини додаються різні значення; в кожному експерименті кількість елементів різна. Кожен із трьох експериментів провести для 10, 100, 1000, 2000 та 3000 елементів. В кожному експерименті необхідно порахувати середню кількість колізій, медіанну кількість колізій, максимальну кількість колізій в одній комірці, кількість

заповнених комірок. Результати звести у табл. 1.1. Зобразити на графіку дві з цих величин на вибір. Зробити висновки.

Таблиця 1.1 – Шаблон оформлення результатів експериментів

Показник	Просте хешування	Хешування методом множення	Хеш-функція Бернштейна
Кількість елементів 10			
Середня кількість колізій			
Медіанна кількість колізій			
Максимальна кількість колізій в одній комірці			
Кількість заповнених комірок			
Кількість елементів – 100			
Середня кількість колізій			
Медіанна кількість колізій			
Максимальна кількість колізій в одній комірці			
Кількість заповнених комірок			
⋮			
Кількість елементів – 3000			
Середня кількість колізій			
Медіанна кількість колізій			
Максимальна кількість колізій в одній комірці			
Кількість заповнених комірок			

Для прикладу на рис. 1.5 подано залежність середньої кількості елементів в списку комірок від кількості доданих в множину елементів. Видно, що просте хешування і хешування методом множення є значно гіршими за хеш-функцію Бернштейна. Це свідчить про те, що ці хеш-функції не можуть рівномірно розподілити всі значення в хеш-таблиці, що спричиняє багато колізій. На рис. 1.6 зображено залежність кількості заповнених комірок від кількості доданих елементів в множину. Тут також хеш-функція

Бернштейна є кращою – спостерігається значна кількість заповнених комірок, що є показником рівномірності розподілу елементів по хеш-таблиці.

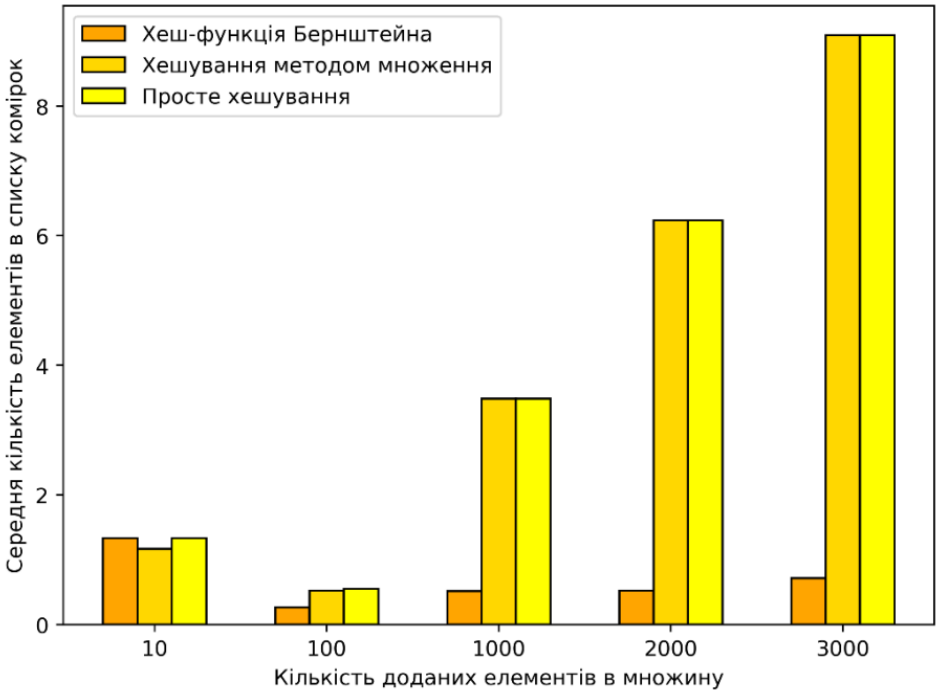


Рисунок 1.5 – Залежності середньої кількості елементів в списку комірок

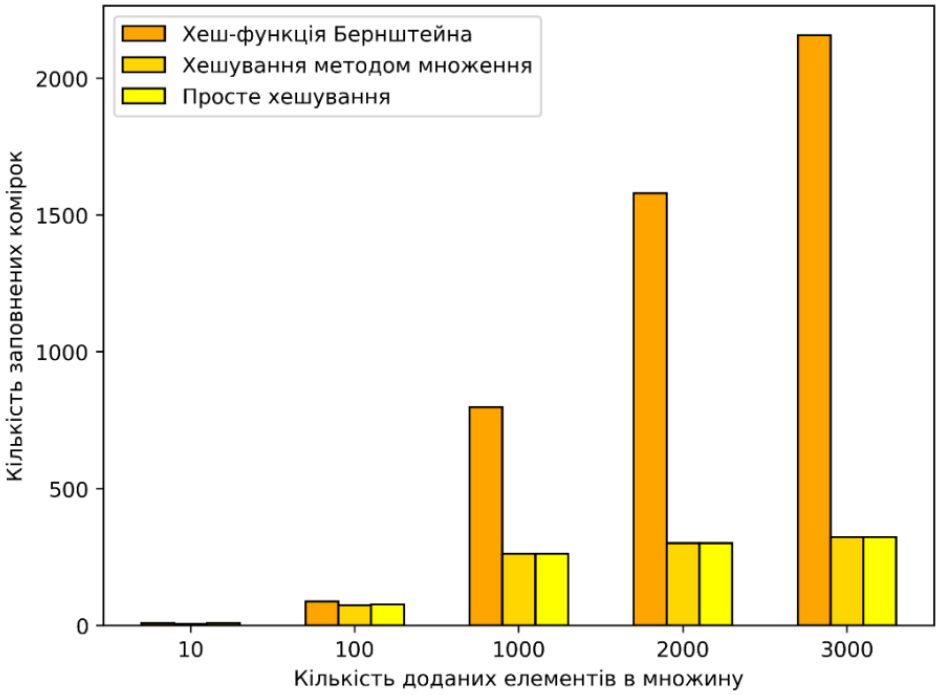


Рисунок 1.6 – Порівняння хеш-функцій за кількістю заповнених комірок

1.7 Поради та рекомендації

Для реалізації множини можна використати такий шаблон коду на Python:

```
class HashSet:
    def __init__(self, initial_capacity=8, load_factor=0.75,
shrink_factor=0.25):
        # розмір хеш-таблиці
        self.capacity = initial_capacity
        # кількість елементів
        self.size = 0
        self.load_factor = load_factor
        self.shrink_factor = shrink_factor
        # хеш-таблиця
        self.buckets = [[] for _ in range(self.capacity)]
    def _hash(self, key: str):
        pass
    def union(self, other):
        pass
    def intersection(self, other):
        pass
    def difference(self, other):
        pass
    def add(self, key: str):
        pass
    def remove(self, key: str):
        pass
    def contains(self, key: str):
        pass
    def get_collisions(self):
        pass
    def _resize(self, new_capacity):
        pass
    def __str__(self):
        return "{" + ", ".join(str(item) for bucket in
self.buckets for item in bucket) + "}"
```

За простого хешування необхідно вхідні текстові дані перетворити на число. Для цього можна використовувати суму числових подань кожного символа окремо, наприклад, `k = sum(ord(c) for c in key)`.

Для тестування розробленої множини та дослідженя колізій необхідно мати достатню кількість даних. Їх можна ввести вручну або згенерувати випадковим чином. Для прикладу нижче наведено код, який генерує 10 випадкових текстових рядків із восьми латинських літер та чисел:

```
import random
import string
random_strings = [‘‘.join(random.choices(string.ascii_letters +
string.digits, k=8)) for _ in range(10)]
```

Для побудови графіків можна використовувати бібліотеку matplotlib. Наприклад, наведений нижче код створить гістограму для трьох різних змінних:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(5)
plt.bar(x - 0.2, [1, 2, 3, 4, 5], width=0.2, color="orange",
edgecolor="black")
plt.bar(x, [6, 7, 8, 9, 10], width=0.2, color="gold",
edgecolor="black")
plt.bar(x + 0.2, [11, 12, 13, 14, 15], color="yellow",
edgecolor="black")
plt.xticks(x, [‘10’, ‘100’, ‘1000’, ‘2000’, ‘3000’])
plt.xlabel("Кількість доданих елементів в множину")
plt.ylabel("Кількість заповнених слотів")
plt.legend(["Хеш-функція Бернштейна", "Хешування методом
множення", "Просте хешування"])
plt.tight_layout()
plt.show()
```

1.8 Питання для самоконтролю та професійного розвитку

1. Що таке множина у дискретній математиці?
2. Як позначаються найбільш поширені множини в математиці? Як ці множини взаємопов’язані?
3. Які основні операції можна виконувати над множинами?
4. Як реалізувати операції перетину та об’єднання відношень за матричної форми подання відношень?
5. Яким буде результат транзитивного замикання відношення «сусіди по поверху»?
6. У чому полягає різниця між множиною в математиці та множиною як структурою даних у програмуванні?
7. Як множини реалізуються у різних мовах програмування?
8. Що таке хеш-функція?
9. У чому полягає принцип Діріхле?
10. У чому полягає парадокс днів народження?
11. У чому полягає принцип дії простого модульного хешування? Який його недолік?
12. Які основні вимоги до програмної реалізації хеш-функцій?
13. Які 3 основні вимоги висуваються до ефективної хеш-функції?
14. Що таке колізія хешування і чому вони виникають?
15. Що таке хеш-таблиця?

16. Що таке коефіцієнтт завантаженості хеш-таблиці і як він характеризує продуктивність хеш-таблиці?

17. Які існують методи обробки колізій у хеш-таблицях?

18. Як працює метод ланцюгового хешування?

19. Як працює метод відкритої адресації?

20. Чому розмір хеш-таблиці бажано задавати простим числом?

21. Як здійснити динамічне розширення та стиснення хеш-таблиці? Коли доцільно це робити? Навіщо це робити, який ефект очікується від таких дій?

22. Як впливає тип хеш-функції на рівномірність розподілу елементів у хеш-таблиці?

23. Що швидше працюватиме: пошук елемента в списку чи в множині?

РОЗДІЛ 2 БАЗОВІ МАНІПУЛЯЦІЇ НА ГРАФАХ

2.1 Поняття графу

Граф (graph) – це сукупність точок, які з'єднані між собою відрізками. Точки графа називають *вершинами* (vertices), а відрізки – *ребрами* (edges). На рис. 2.1 наведено приклад графу, який має 4 вершини та 5 ребер.

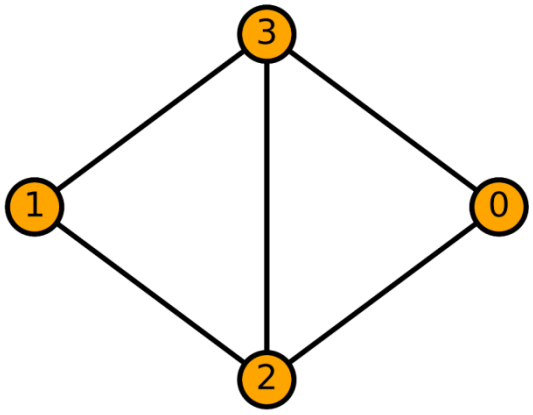


Рисунок 2.1 – Приклад неорієнтованого графу

Графи є важливим інструментом моделювання та аналізу складних систем у різних сферах науки і техніки. Наприклад, у комп'ютерних науках графи використовуються для подання мереж, де вершини відповідають серверам та комп'ютерам, а з'єднання між ними моделюють канали передачі даних. Інший приклад – алгоритм PageRank, який лежить в основі Google Search. Він аналізує граф посилань між вебсторінками для визначення їх важливості.

В соціальних мережах також широко використовують графові структури: акаунти користувачів відповідають вершинам, а зв'язки між ними – ребрам. Це дозволяє виокремлювати спільноти, аналізувати вплив окремих користувачів та прогнозувати поведінку мережі. Подібні методи застосовуються й у логістиці та в транспортних технологіях, де графи допомагають знаходити найкоротші маршрути, оптимізувати перевезення та прогнозувати затори. У навігаційних системах, таких як Google Maps чи Waze, міста та перехрестя є вершинами графу, а дороги – ребрами з відповідними вагами, що можуть відображати відстань або час у дорозі.

У біоінформатиці графи використовуються для аналізу послідовностей ДНК, моделювання білкових взаємодій та побудови еволюційних дерев. У хімії графи застосовуються для подання молекулярних структур, де атоми є вершинами, а хімічні зв'язки – ребрами. В електроніці та комунікаціях

графові моделі допомагають аналізувати складні електричні схеми та телекомунікаційні мережі, визначаючи оптимальні шляхи передачі сигналів та енергії.

В термінах дискретної математики, граф – це сукупність двох множин:

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

де V – множина вершин;

E – множина двоелементних підмножин множини V, тобто множина ребер, які з'єднують вершини з множини V.

Кількість вершин графу G називається *розміром графу*: $n = |V|$. Ребра графу зазвичай позначають $e_1, e_2, ..., e_m$ або з використанням номерів з'єднуваних вершин – e_{ij}. Кількість ребер графу G називають *потужністю графу*: $|E| = m$. Наприклад, граф з рис. 2.1 задається двома множинами: $V = \{0,\ 1,\ 2,\ 3\}$ та $E = \{(0, 2), (0, 3), (1, 3), (1, 2), (2, 3)\}$. Для цього графу $n = 4$ та $m = 5$. Порядок нумерації вершин не має значення. Так само не має значення і порядок нумерації ребер.

Графи бувають неорієнтованими та орієнтованими. *Неорієнтований граф* – це граф, у якому ребра не мають напрямку. В *орієнтованому графі* деякі або усі ребра мають напрямок. Такі ребра називаються *дугами* (arcs). Орієнтований граф скорочено називають *орграфом* (digraph – directed graph). Приклад орграфу подано на рис. 2.2. Для орграфу ребро може мати напрямок, що позначає односторонній зв'язок між двома вершинами. Такий односторонній зв'язок на рисунку позначається стрілкою.

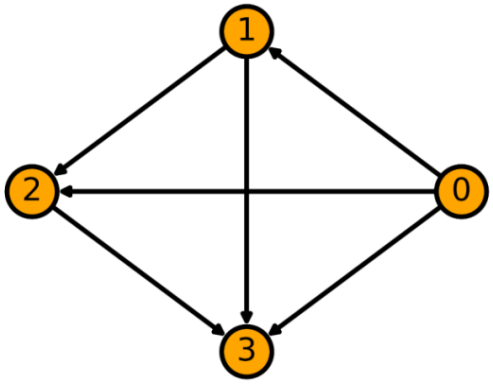


Рисунок 2.2 – Приклад орграфу

Іноді є потреба пару вершин з'єднати більше, ніж одним ребром. Ребра чи дуги одного напряму, які з'єднують ту саму пару вершин, називаються *кратними* або *паралельними*. Дуга, що сполучає вершину саму з собою називається *петлею*. Граф без кратних дуг і петель називається *простим*.

Вершина і ребро називаються *інцидентними*, якщо ребро з'єднує цю вершину з іншою. Якщо v_1, v_2 – вершини, а $e = \{v_1, v_2\}$ – ребро, що їх з'єднує, тоді вершина v_1 і ребро e інцидентні. Вершина v_2 і ребро e також є інцидентними.

Два ребра, що інцидентні одній вершині, називаються *суміжними ребрами*. Дві вершини, інцидентні одному ребру, також називаються *суміжними*.

Неорієнтований граф називається *повним*, якщо кожну пару його вершин поєднує ребро. Повний граф також називають *повнозв'язним* графом або *клікою*. Повний граф має максимально можливу кількість ребер (не враховуючи петлі), яка становить: $m = \frac{n(n-1)}{2}$, де n – число вершин. Кожна з n вершин з'єднується з $(n-1)$ іншими вершинами, тому загальна кількість кінців ребер дорівнює $n(n-1)$. Але, у кожного ребра є 2 кінці, тому отримане число ділиться на 2.

Граф називається *щільним* (dense), якщо кількість його ребер близька до максимальної. Граф з малою кількістю ребер називається *розрідженим* (sparse). Різниця між щільним і розрідженим графом розмита, і залежить від контексту. Повний граф є щільним графом, оскільки він має максимальну кількість ребер. Формально, щільним є граф, кількість ребер якого наближається до $|V|^2$. У розрідженого графу кількість ребер значно менша за $|V|^2$. Для опису щільних графів потрібно набагато більше пам'яті, ніж для розріджених.

Для оцінювання щільності неорієнтованого графу можна використовувати такий показник:

$$D = \frac{|E|}{0.5n(n-1)} = \frac{2|E|}{n(n-1)},$$

де $|E|$ – кількість ребер у графі;

$0.5n(n–1)$ – максимально можлива кількість ребер у неорієнтованому графі.

Приклад повного графа, графа з максимальною щільністю, подано на рис. 2.3.

У ньому є n=7 вершин, кожна з яких з'єднана з усіма іншими вершинами, відповідно кількість ребер становить m=21. Для цього графа D=1.

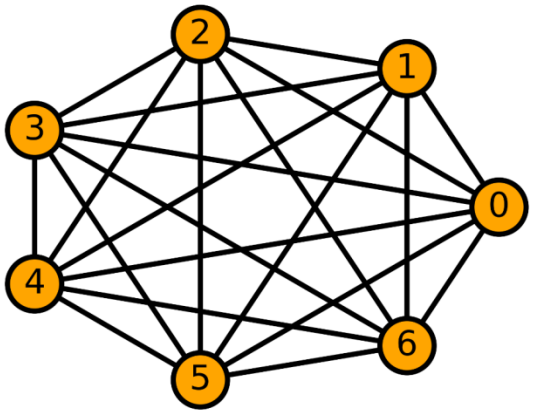


Рисунок 2.3 – Приклад повного графу

Приклад графу з середнім рівнем щільності подано на рис. 2.4. У ньому є $n=7$ вершин, і лише мала частка з них з'єднана ребрами. На цьому графі є $m=10$ ребер, тому $D=\frac{2\cdot 10}{7(7-1)}=0.48$.

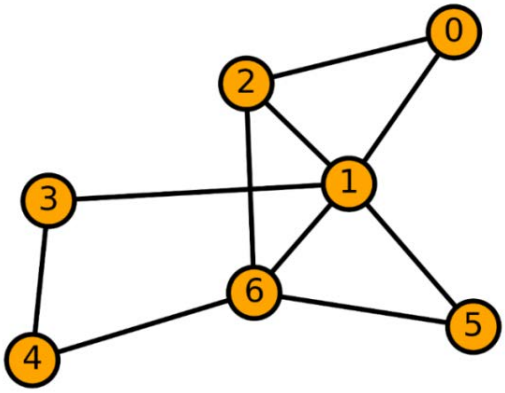


Рисунок 2.4 – Приклад графу з середнім рівнем щільності

2.2 Способи подання графів

Графи можна задати різними структурами даних, що дозволяє ефективно знаходити найкоротші шляхи та компоненти зв'язності, аналізувати графові властивості та виконувати інші процедури. Вибір способу подання залежить від типу графу, його розмірності та частоти виконання тих чи інших операцій. Найчастіше граф задають списком ребер, матрицею суміжності, матрицею інцидентності та списком суміжності.

Списком ребер графу G називається матриця **A** розміром $m\times 2$, в якій кожен рядок – ребро. Значення в рядку відповідають вершинам, які з'єднує ребро.

В i-му рядку $a_{i1} = v_j$ і $a_{i1} = v_k$, якщо вершини v_j та v_k з'єднані ребром. Для розрідженого графу з рис. 2.4 список ребер подано на рис. 2.5.

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

Рисунок 2.5 – Список ребер графу з рис. 2.4

Список ребер легко зчитувати з файлів чи баз даних, особливо якщо граф подано списком зв'язків. Цей список підходить для алгоритмів, які працюють безпосередньо з ребрами, наприклад, для пошуку мінімального кістякового дерева чи обчислення найкоротших шляхів. Але, якщо потрібно швидко знайти всі вершини, суміжні з аналізованою вершиною, то доводиться переглядати весь список – а це нераціонально. За такого подання неможливо реалізувати швидкий доступ до всіх ребер, що інцидентні вибраній вершині. Якщо граф динамічно змінюється, то такі зміни легко реалізуються лише у випадку додавання нового ребра. В цьому випадку нове ребро дописується в кінець матриці додатковим рядком. Якщо потрібно видалити ребро, тоді потрібно не лише знайти відповідний рядок матриці, але й після видалення стиснути матрицю. Як варіант, можна у матрицю ввести додатковий стовпець зі статусом ребра – наявне чи видалене, але це погіршить компактність подання.

Матрицею суміжності (adjacency matrix) простого графу G називається бінарна матриця $\mathbf{A}$ розміром $n \times n$, в якій елемент $a_{ij} = 1$, коли вершини v_i та v_j є суміжними, і $a_{ij} = 0$, коли вершини v_i та v_j не суміжні. Для неорієнтованого графу матриця суміжності є симетричною. Матрицею суміжності можна задавати і граф з петлями, тоді на головній діагоналі будуть не нулі, а одиниці.

Матрицю суміжності графу з рис. 2.4 показано на рис. 2.6. Для наочності рядки і стовпці матриці позначено відповідними номерами вершин. Як бачимо, для розрідженого графу матриця суміжності містить багато нулів. Її розмір не залежить від кількості ребер графу. На матриці суміжності можна

просто реалізовувати маніпуляцію з ребрами – додавати чи видаляти їх. Матриця суміжності може задаватися і як бульова матриця. Тоді $a_{ij} = \text{true}$, коли вершини v_i та v_j є суміжними, і $a_{ij} = \text{false}$ в інших випадках.

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

Рисунок 2.6 – Матриця суміжності графу з рис. 2.4

Матрицею інцидентності (incident matrix) графу G називається матриця **A** розміром $n \times m$, кожен елемент якої дорівнює 1 ($a_{ij} = 1$) тоді і тільки тоді, коли вершина v_i інцидентна ребру e_j. Матрицю інцидентності графу з рис. 2.4 показано на рис. 2.7. У ній рядки відповідають номерам вершин, а стовпці – ребрам. Інколи матрицю інцидентності задають навпаки, коли рядки відповідають ребрам, а стовпці – вершинам. Тому, перед використанням «чужих», зокрема і згенерованих штучним інтелектом, алгоритмів чи програмних функцій, потрібно перевірити спосіб подання матриці інцидентності.

	(0, 1)	(0, 2)	(1, 2)	(1, 3)	(1, 5)	(1, 6)	(2, 6)	(3, 4)	(4, 6)	(5, 6)
0	1	1	0	0	0	0	0	0	0	0
1	1	0	1	1	1	1	0	0	0	0
2	0	1	1	0	0	0	1	0	0	0
3	0	0	0	1	0	0	0	1	0	0
4	0	0	0	0	0	0	0	1	1	0
5	0	0	0	0	1	0	0	0	0	1
6	0	0	0	0	0	1	1	0	1	1

Рисунок 2.7 – Матриця інцидентності графу з рис. 2.4

Для простого графу у кожному стовпці матриці інцидентності буде рівно дві одиниці, а решта – нулі. Якщо граф має петлю, тоді у відповідному стовпці буде лише одна одиниця. Якщо граф має кратні ребра, тоді в матриці інцидентності будуть однакові стовпці. Матрицею інцидентності можна

задати і орієнтовані графи. В цьому випадку, $a_{ij} = 1$, якщо j-та дуга входить у вершину v_i, та $a_{ij} = -1$, якщо j-та дуга виходить з вершини v_i.

У випадку подання графу матрицею інцидентності доволі просто додавати нову вершину чи ребро. Матриця тоді збільшується на 1 рядок чи 1 стовпець, а решта матриці змін не зазнає. Якщо видаляти некрайні вершину чи ребро, тоді потрібно буде стискати матрицю.

Список суміжності (adjacency list) – це набір невпорядкованих списків, які використовуються для подання графу. Кожен невпорядкований список у ньому описує набір сусідів певної вершини графу. Список суміжності графу пов'язує кожну вершину в графі з набором її сусідніх вершин або ребер. Для графу з рис. 2.4 список суміжності наведено на рис. 2.8. Для цього графу список суміжності значно менший за відповідні матрицю суміжності та матрицю інцидентності. Це насамперед зумовлено тим, що в списку суміжності інформація про ребра зберігається без надлишковості, фіксується лише факт наявності ребра. На відміну від матриць суміжності та інцидентності, відсутність ребра спеціально не зазначається. Це робить список суміжності ефективним у задачах, де важливо працювати з великими графами, з малою кількість зв'язків між вершинами. Прикладами таких об'єктів є дорожні карти міста, кожне перехрестя на яких можна подати вершиною, а відповідні дороги – ребрами. Оскільки дорожні мережі зазвичай мають невелику кількість з'єднань між кожною парою точок, список суміжності дозволяє зберігати таку інформацію раціонально.

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

Рисунок 2.8 – Список суміжності графу з рис. 2.4

Якщо сформувати список суміжності для повного графу з рис. 2.3, тоді розмір отриманого списку наближатиметься до розміру матриці суміжності. Звідси випливає, що список суміжності економить пам'ять лише для розріджених графів. Якщо граф щільний, тоді розмір списку суміжності наближається до розміру матриці суміжності і його основна перевага зникає. Це наочно показано на рис. 2.9. Для розріджених графів розмір списку суміжності значно менший за розмір матриці суміжності. Розмір списку ребер

пропорційний $2|E|$. Розмір матриці суміжності пропорційний $|V|^2$. Розмір списку суміжності пропорційний $|V|+|E|$. Щодо матриці інцидентності, то її розмір пропорційний $|V|\cdot|E|$. Якщо граф повний, то розмір матриці інцидентності перевищує розмір матриці суміжності, бо в цьому випадку $|E|>|V|$.

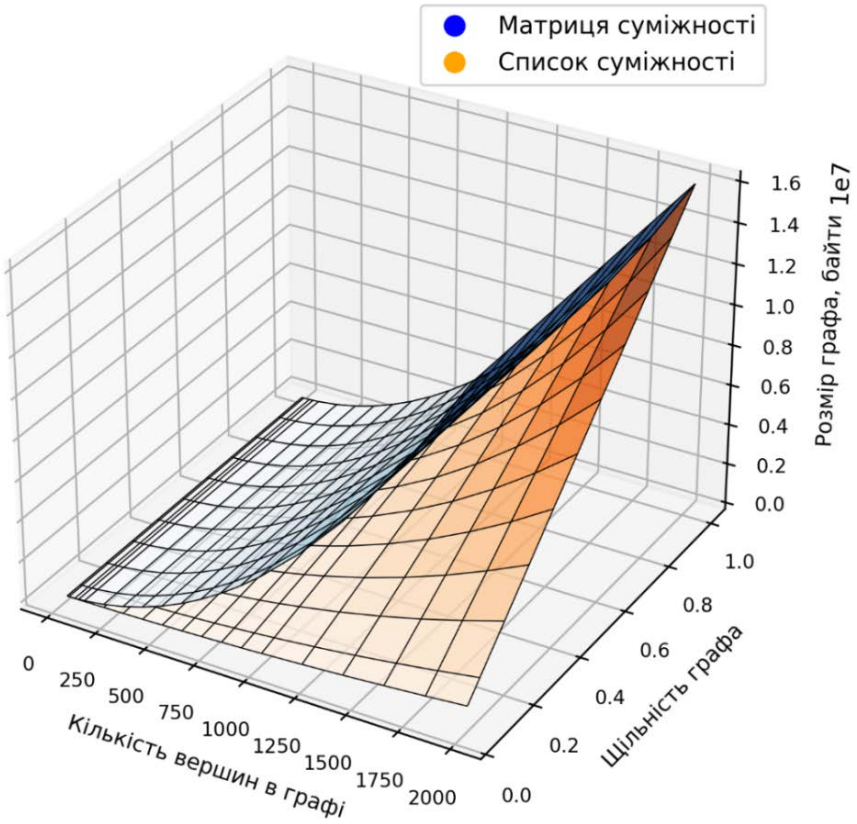


Рисунок 2.9 – Залежність обсягу пам’яті для зберігання графу від його щільності та кількості вершин

2.3 Типові задачі обходу графів

2.3.1 Шлях на графі

Шлях або *маршрут* (path) на графі $G=(V,E)$ – це послідовність вершин та ребер виду $v_1-e_1-v_2-e_2-\ldots-v_k$, у якій сусідні елементи є інцидентними. Шлях також можна позначати у вигляді послідовності вершин $v_1-v_2-v_3-\ldots-v_k$ за умови, що кожна сусідня пара вершин утворює ребро та між ними відсутні кратні ребра. Шлях є *простим*, якщо кожна вершина зустрічається в ньому лише один раз. На рис. 2.10 на неорієнтованому графі виділено простий шлях 0–1–4–3–2. Шлях позначено червоним кольором.

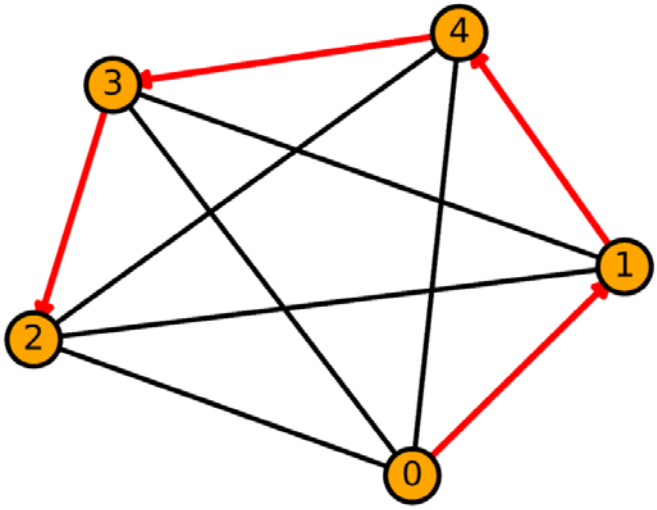


Рисунок 2.10 – Приклад простого шляху через 5 вершин

Шлях $v_1 - e_1 - v_2 - e_2 - \ldots - v_k$ називається *циклом*, якщо в ньому початкова та кінцева вершини збігаються: $v_1 = v_k$. Цикл називається *простим*, якщо будь-яка вершина зустрічається в ньому лише один раз (рис. 2.11). Простий цикл – це цикл без самоперетинів, тобто вершини у ньому не повторюються. Першу та останню вершину в циклі можна вважати за одну, тому в простому циклі немає вершин, що повторюються. Відповідно в простому циклі відсутні повторювані ребра. З будь-якого непростого цикла можна виділити принаймні 2 різні прості цикли. Якщо граф не містить циклів, то він називається *ациклічним*.

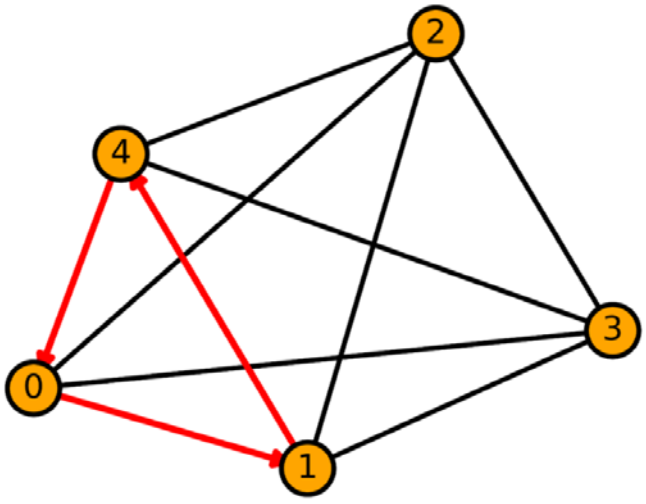


Рисунок 2.11 – Приклад простого циклу на неорієнтованому графі

2.3.2 Зв'язність

Неорієнтований граф $G = (V, E)$ називається *зв'язним*, якщо існує шлях із будь-якої його вершини у будь-яку іншу. Це поняття застосовується для

неорієнтованих графів. У незв'язних графах можна виділити окремі компоненти зв'язності. Компонента зв'язності – це підграф, в якому між будь-якою парою вершин є шлях, тобто це зв'язний підграф. У граничному випадку, коли граф взагалі не має ребер ($E = \varnothing$), у нього буде $m = n$ компонентів зв'язності – по одній компоненті на кожну вершину. У зв'язному графі виконується умова: $m \geq n - 1$. Тобто, мінімально можлива кількість ребер зв'язного графу дорівнює $n - 1$. Неорієнтований зв'язний ациклічний граф, де кожні дві вершини з'єднані єдиним шляхом називають *деревом* (tree). Якщо у графі є декілька компонент зв'язності і вони ациклічні, тобто є деревами, то такий граф називається лісом. *Ліс* (forest) – це деяка сукупність дерев. На рис. 2.12 зображено ліс із двох дерев – граф з двома компонентами зв'язності.

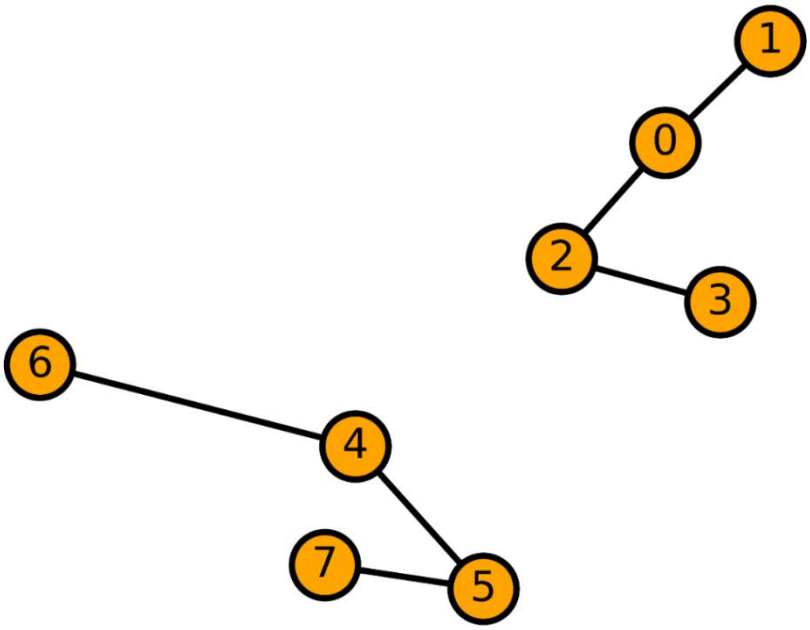


Рисунок 2.12 – Ліс із двох дерев

Орієнтований граф називається *сильно зв'язним*, якщо існує шлях з будь-якої вершини графу до будь-якої іншої вершини. Зокрема, це означає наявність шляхів в обох напрямках. *Компонента сильної зв'язності* орієнтованого графу G – це зв'язний підграф, в якому з будь-якої вершини можна дійти до будь-якої іншої вершини. На рис. 2.13 виділено 3 компоненти зв'язності, усередині яких існує шлях з будь-якої вершини до кожної з інших вершин. На цьому графі є 3 компоненти сильної зв'язності.

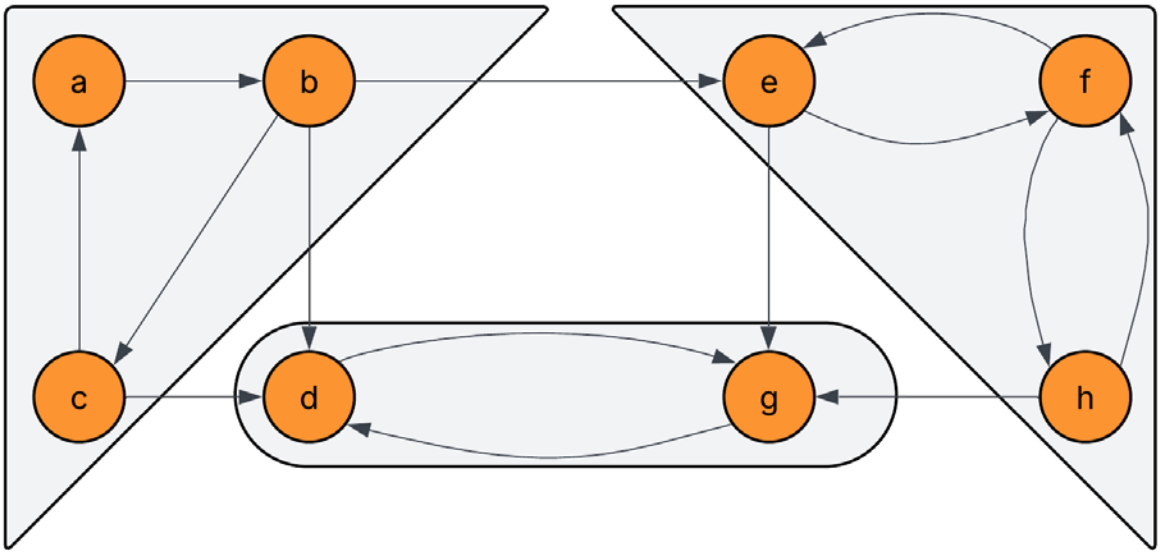


Рисунок 2.13 – Компоненти сильної зв’язності орієнтованого графу

2.3.3 Обхід графу

Обхід графу – це процес відвідування його вершин за певним правилом. Алгоритми обходу графу використовуються для вирішення різних задач, як-от перевірка зв’язності, пошук шляхів, виявлення циклів, визначення найкоротших маршрутів тощо. Обхід графу важливий елемент розв’язання широкого спектра задач.

Розглянемо задачі аналізу структури графу – перевірка зв’язності графу, пошук компонент зв’язності, знаходження найкоротшого шляху на графі та виявлення циклів на графі. Найпростішими алгоритмами обходу графу є пошук в глибину та пошук в ширину.

2.3.4 Пошук в глибину

Пошук в глибину (depth first search) – це алгоритм обходу дерева або структури, подібної до дерева або графу. Для дерева робота алгоритму починається з кореня, а для іншого типу графу початковою може бути будь-яка вершина. Вибір наступної вершини здійснюється так, щоб забезпечити максимально можливу глибину обходу. Під глибиною обходу розуміється кількість вершин в ланцюгу від початкової вершини до кінцевої.

На початку пошуку початкова вершина заноситься до списку відвіданих. Потім алгоритм переходить до першої невідвіданої суміжної вершини та повторює цей процес, занурюючись дедалі глибше в граф. Якщо поточна вершина не має невідвіданих сусідів, алгоритм повертається до попередньої вершини, звідки був здійснений перехід, і шукає інший шлях. Такий процес триває доти, доки не будуть розглянуті всі можливі шляхи.

Пошук в глибину застосовується у багатьох практичних задачах, пов’язаних із графами та деревами. Один із найпоширеніших випадків – це

пошук шляхів у лабіринті або в мережах, де алгоритм допомагає знайти можливі маршрути від початкової точки до кінцевої. Також пошук в глибину застосовується для перевірки зв'язності графу, тобто виявлення шляху між двома довільними вершинами. Це важливо під час аналізу соціальних мереж, де алгоритм допомагає знаходити групи взаємопов'язаних користувачів. У задачах складання розкладу завдань пошук в глибину використовується для топологічного сортування, коли потрібно визначити порядок виконання дій без порушення логічних зв'язків між ними. Ще одним важливим застосуванням є виявлення циклів на графах, що може використовуватися в аналізі маршрутизації або в компіляторах для перевірки наявності циклічних залежностей.

Псевдокод алгоритму пошуку у глибину, який використовує список суміжності, подано нижче. Складність цього алгоритму становить $O(|V|+|E|)$. Ілюстрацію пошуку у глибину подано на рис. 2.14. Порядок обходу показано паралельними до дуг стрілками. Початкова вершина пошуку – *A*.

```
function DFS(adjacency_list, node, visited):
    if node not in visited:
        visited.add(node)
        for neighbor in adjacency_list[node]:
            DFS(adjacency_list, neighbor, visited)
```

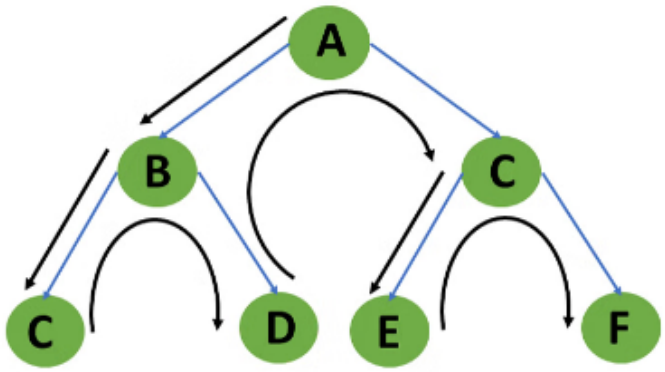


Рисунок 2.14 – Ілюстрація пошуку в глибину

2.3.5 Пошук в ширину

Пошук у ширину (breadth first search) – це алгоритм обходу графу, який працює за принципом порівневого перегляду вершин. На відміну від пошуку в глибину, який заглиблюється у граф, пошук у ширину спершу досліджує всі вершини на одному рівні, а вже потім аналізує вершини наступного рівня. Якщо обхід починається з деякої вершини *A*, то на першому рівні знаходяться всі суміжні вершини до *A*, на другому рівні знаходяться всі вершини суміжні до вершин з першого рівня тощо. Пошук розпочинається з додавання

вибраної початкової вершини в чергу. Вершина позначається як відвідана, щоб уникнути повторних візитів. Поки черга не порожня, здійснюємо такі операції: 1) беремо вершину з початку черги; 2) обробляємо цю вершину (наприклад, виводимо її в консоль); 3) додаємо у кінець черги всіх її невідвіданих сусідів і позначаємо її як відвідану. Повторюємо цей процес, доки не будуть оброблені всі досяжні вершини.

Псевдокод алгоритму пошуку у ширину, що використовує список суміжності, подано нижче. Складність такого алгоритму становить $O(|V|+|E|)$. Ілюстрацію пошуку у ширину подано на рис. 2.15. Порядок обходу показано стрілками. Початкова вершина, з якої розпочинається пошук – *A*:

```
function BFS(graph, start):
    queue = Queue()
    visited = Set()

    queue.enqueue(start)
    visited.add(start)

    while not queue.isEmpty():
        node = queue.dequeue()
        process(node)

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.enqueue(neighbor)
```

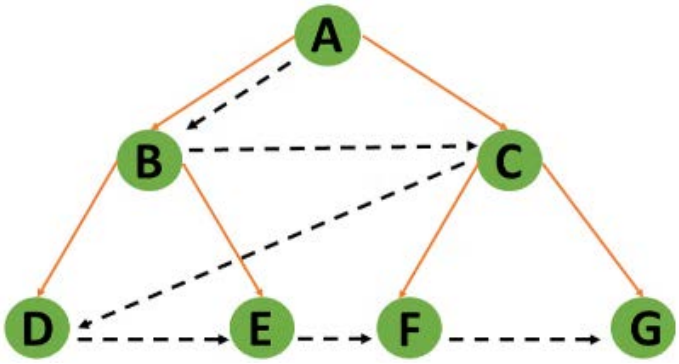


Рисунок 2.15 – Приклад пошуку в ширину, починаючи з вершини A

2.3.6 Пошук найкоротшого маршруту на неорієнтованому графі

Пошук найкоротшого шляху між двома вершинами в неорієнтованому графі означає знаходження найменшої кількості ребер, необхідних для переходу від початкової вершини до цільової. Для цієї задачі найефективнішим є алгоритм пошуку у ширину. Псевдокод *виявлення найкоротшого шляху за схемою пошуку у ширину* є таким:

```
function BFS_ShortestPath(graph, start, target):
    distance = {v: ∞ for v in graph}
    parent = {v: NULL for v in graph}
    queue = [start]
    distance[start] = 0
    while queue.isNotEmpty():
        current = queue.pop(0)
        if current == target: break
        for neighbor in graph[current]:
            if distance[neighbor] == ∞:
                distance[neighbor] = distance[current] + 1
                parent[neighbor] = current
                queue.append(neighbor)
    if distance[target] == ∞: return "Немає шляху"
    path = [], step = []
    while step is not NULL:
        path.insert(0, step)
        step = parent[step]
    return path
```

2.3.7 Виявлення зв’язності графу

Для неорінтованого графу доцільно знати, чи є він зв’язним. Зв’язність означає, що з будь-якої вершини можна дістатися до будь-якої іншої вершини. *Алгоритм встановлення зв’язності* є таким:

1. Вибрати довільну вершину як стартову.
2. Виконати обхід графу за допомогою пошуку в глибину або пошуку в ширину.
3. Позначати відвідані вершини під час обходу.
4. Після завершення обходу перевірити, чи усі вершини відвідано. Якщо так, тоді граф є зв’язним, інакше граф є незв’язним.

Використовуючи обхід графу за пошуком у глибину або за пошуком в ширину, можна визначити всі вершини, що належать до однієї компоненти, а потім перейти до наступної.

Алгоритм визначення компонент зв’язності є таким:

1. Створити масив відвіданих вершин.
2. Перебрати всі вершини графу. Якщо вершина ще не відвідана, то виконати пошук в глибину або пошук в ширину від цієї вершини та позначити всі досяжні вершини як частину однієї компоненти зв’язності.
3. Повторити попередній крок, поки всі вершини графу не будуть перевірені.
4. Кількість запусків пошуку в глибину або пошуку в ширину дорівнює кількості компонент зв’язності.

2.3.8 Перевірка існування циклу

Способи перевірки наявності циклів залежать від типу графу. *Алгоритм виявлення циклу на неорієнтованому графі* є таким:

1. Запустити пошук у глибину від довільної вершини, відзначаючи всі відвідані вершини.
2. Для кожної сусідньої вершини перевірити, чи сусід ще невідвіданий. Якщо невідвіданий, то продовжити пошук в глибину, інакше, якщо сусід не є батьківською вершиною, то зафіксувати виявлений цикл.
3. Повторити, поки не буде перевірено всі вершини.

У орієнтованому графі цикл означає, що існує шлях, за яким можна повернутися у початкову вершину, дотримуючись напрямку ребер.

Алгоритм виявлення циклу на орієнтованому графі є таким:

1. Кожну вершину позначити такими кольорами:
   - білим, якщо вершина ще не розглянута;
   - сірим, якщо вершина перебуває в обробці (відкритий рекурсивний стек);
   - чорним, якщо вершина повністю оброблена.
2. Виконати пошук в глибину для кожної вершини, враховуючи таке:
   - якщо натрапляємо на вершину сірого кольору, це означає повернення у вже відкритий рекурсивний стек, тобто знайдено цикл;
   - якщо вершина стала чорного кольору, то вона вже перевірена і не внесена до циклу.
3. Повторити процес для всіх вершин.

2.4 Завдання для самостійного дослідження

Базовий рівень

1. Випадковим чином згенерувати граф, кількість вершин і щільність заповнення якого наведено в табл. 2.1.

2. Подати згенерований граф матрицею суміжності, матрицею інцидентності та списком суміжності. Написати всі можливі функції перетворення: матриці суміжності в матрицю інцидентності, матриці інцидентності в список суміжності, матриці суміжності в список суміжності тощо. Перевірити коректність згенерованих функцій-конверторів.

3. За характеристиками графа з табл. 2.1 згенерувати граф з щільностями $\underline{D} = 0.1$ та $\overline{D} = 0.9$. Порівняти розмір матриці суміжності з розміром списку суміжності для трьох щільностей: $\underline{D} = 0.1$, $\overline{D} = 0.9$ та щільність D^*, яка задана у варіанті. Зробити проміжний висновок.

Таблиця 2.1 – Варіанти завдань

Варіант	Кількість вершин, n^*	Щільність, D^*	Варіант	Кількість вершин, n^*	Щільність, D^*
1	30	0.4	33	22	0.61
2	20	0.5	34	27	0.39
3	25	0.6	35	38	0.37
4	35	0.7	36	15	0.35
5	15	0.41	37	41	0.33
6	40	0.43	38	44	0.31
7	45	0.45	39	33	0.25
8	33	0.47	40	22	0.77
9	21	0.49	41	26	0.67
10	26	0.51	42	39	0.88
11	31	0.53	43	47	0.51
12	35	0.55	44	49	0.53
13	37	0.57	45	51	0.55
14	39	0.59	46	52	0.57
15	41	0.61	47	53	0.59
16	43	0.39	48	33	0.37
17	45	0.37	49	28	0.35
18	47	0.35	50	25	0.33
19	49	0.33	51	35	0.31
20	51	0.31	52	16	0.25
21	52	0.25	53	26	0.37
22	53	0.77	54	31	0.35
23	54	0.67	55	35	0.33
24	55	0.88	56	37	0.43
25	56	0.79	57	39	0.45
26	57	0.44	58	64	0.33
27	58	0.45	59	65	0.41
28	59	0.38	60	66	0.72
29	60	0.48	61	67	0.64
30	61	0.57	62	68	0.62
31	62	0.53	63	69	0.54
32	63	0.59	64	70	0.77

4. Провести серію експериментів для графів з різною кількістю вершин та ребер, і з'ясувати, який обсяг пам'яті потрібен для запису матриці суміжності та списку суміжності графу. Розмір графу в кожній серії експериментів змінювати за таким планом: [10, 20, 30, 40, 50, 100, 200, 300]. Для кожного експерименту розрахувати обсяг пам'яті для розрідженого графу, коли $\underline{D} = 0.1,$ та для щільного графу, коли $\overline{D} = 0.9$.

Результати експериментів візуалізувати. Рекомендований формат візуалізації наведено на рис. 2.16 та 2.17.

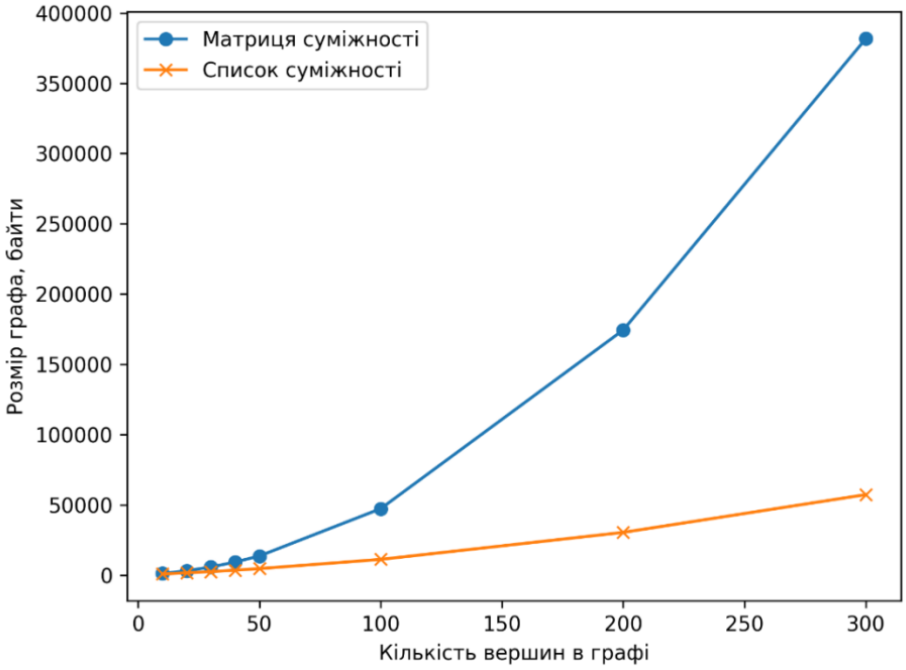


Рисунок 2.16 – Залежність розміру матриць розрідженого графу від кількості вершин за $\underline{D} = 0.1$

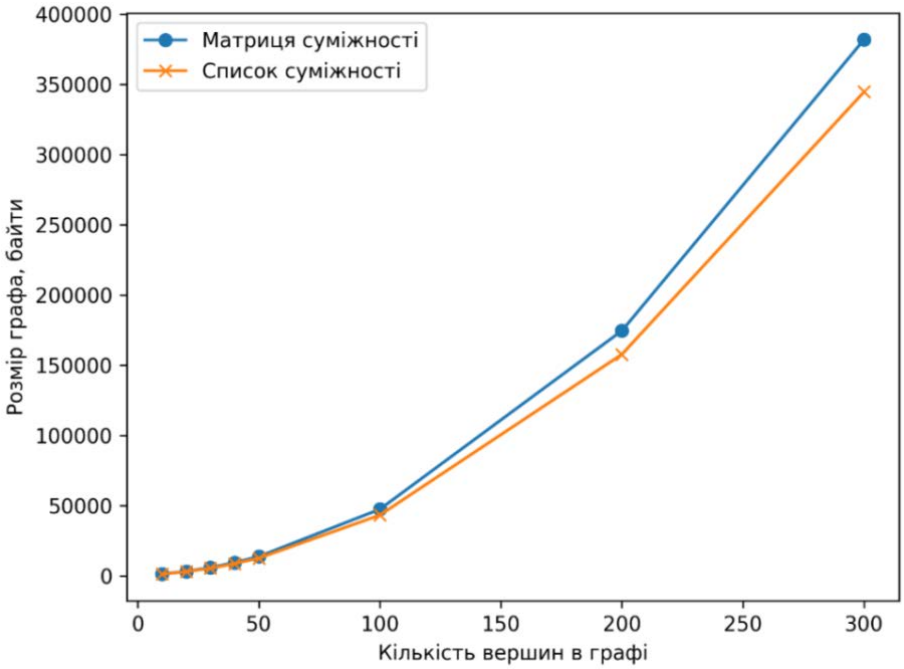


Рисунок 2.17 – Залежність розміру матриць щільного графу від кількості вершин за $\overline{D} = 0.9$

Поглиблений рівень

1. Реалізувати програмно задачу обходу графу відповідно до варіанта. Ребра або дуги графу згенерувати на власний розсуд, проте граф має бути придатним для ілюстрації особливостей роботи програми. Для кожної задачі обходу має бути щонайменше 2 графи для перевірки коректності роботи програми:

- для пошуку компонент зв'язності – один граф лише з однією компонентою зв'язності та один граф принаймні з двома компонентами зв'язності;
- для перевірки зв'язності – один зв'язний граф та один незв'язний граф;
- для виявлення циклу – один граф з циклом та один без циклу;
- для пошуку найкоротшого шляху – один граф, де існує найкоротший шлях між двома вершинами та один граф, де немає шляху між двома вершинами (неможливо знайти найкоротший шлях).

Вимоги до результатів роботи програми:

- *для пошуку компонент зв'язності* – вивести всі компоненти зв'язності;
- *для перевірки зв'язності* – встановити з відповідним обґрунтуванням, чи є граф є зв'язним;
- *для виявлення циклу* – вивести цикл або повідомлення, що циклу немає;
- *для пошуку найкоротшого шляху* – вивести шлях або повідомлення, що найкоротшого шляху немає.

Таблиця 2.2 – Варіанти завдань

Варіант	Кількість вершин у графі	Тип графу	Задача обходу
1	10	Неорієнтований	Пошук компонент зв'язності
2	11		
3	12		
4	13		
5	14		
6	15		
7	16		
8	17		
9	18		
10	19		
11	20		
12	21		
13	22		

Продовження таблиці 2.2

Варіант	Кількість вершин у графі	Тип графу	Задача обходу
14	12	Неорієнтований	Перевірка зв'язності
15	11		
16	10		
17	13		
18	15		
19	8		
20	16		
21	14		
22	17		
23	18		
24	11	Неорієнтований	Виявлення циклу
25	12		
26	9		
27	8		
28	13		
29	15		
30	17		
31	14	Орієнтований	
32	11		
33	10		
34	12		
35	13		
36	18		
37	16		
38	14	Неорієнтований	Пошук найкоротшого шляху
39	11		
40	10		
41	12		
42	13		
43	15		
44	17		
45	18		
46	19		
47	20		
48	11	Орієнтований	
49	12		
50	16		
51	17		
52	13		
53	18		
54	19		
55	14		
56	20		

2.5 Поради та рекомендації

Для створення матриці суміжності та списку суміжності в Python можна використати структуру даних «Список». Наприклад, така функція створює матрицю суміжності розміром $n \times n$:

```
def create_matrix_graph(n: int):
    return [[0] * n for _ in range(n)]

adjacency_matrix = create_matrix_graph(3)
adjacency_matrix[0][1] = 1 # ребро між вершиною 0 та 1
```

А такий код створює список суміжності розміром n:

```
def create_adjacency_list_graph(n: int):
    return [[] for _ in range(n)]

adjacency_list = create_adjacency_list_graph(3)
adjacency_list[0] = [1, 2] # ребро між вершиною 0 і вершинами 1,2
```

В Python структура даних «Список» є динамічною, її фактичний розмір може значно перевищувати кількість доданих елементів. Це може спотворити процес визначення фактичного розміру матриці суміжності і списку суміжності. Тому для коректного підрахунку після заповнення даними необхідно конвертувати список у статичний масив Python:

```
import array

adjacency_matrix = [array.array('I', l) for l in
adjacency_matrix]
adjacency_list = [array.array('I', l) for l in adjacency_list]
```

Щоб згенерувати граф з певним рівнем щільності можна використовувати 2 підходи. Перший підхід – це додавання ребра до кожної пари вершин послідовно до досягнення заданого рівня щільності. Другий підхід – додавати ребро лише з певним рівнем ймовірності, що дорівнює рівню щільності. Другий підхід реалізувати простіше, але він не гарантує, що щільність графа буде точно такою, як задана. За ймовірнісної реалізації отримаємо граф, рівень щільності якого буде близьким до вказаної. В контексті мети самостійної роботи таке відхилення є другорядним. Для реалізації ймовірносного підходу можна використати такий код:

```
import random

sparsity = 0.5 # щільність
for i in range(n):
    for j in range(n):
        if random.random() < sparsity:
            # між i та j є ребро
```

Для визначення розміру матриці суміжності та списку суміжності можна використовувати таку функцію:

```
def total_size(obj, seen=None):
    size = 0

    if isinstance(obj, (list, tuple, set, array.array)):
        size += sum(total_size(i, seen) for i in obj)
        for e in obj:
            if isinstance(e, (list, tuple, set, array.array)):
                for i in e:
                    size += sys.getsizeof(i)
            else:
                size += sys.getsizeof(e)

    return size
```

2.6 Питання для самоконтролю та професійного розвитку

1. Що таке граф?
2. Чим відрізняються орієнтований граф від неорієнтованого?
3. Чи може дуга бути і петлею?
4. Що таке інцидентність?
5. Що таке суміжність?
6. Що таке потужність графу?
7. Як порахувати максимально можливу кількість ребер на неорієнтованому графі?
8. Яка максимальна кількість ребер може бути у простому графі з *n* вершинами?
9. Як можна кількісно оцінити щільність графу?
10. Чим відрізняється розріджений граф від щільного?
11. Яка різниця між простим графом, мультиграфом та графом з петлями?
12. Які існують способи подання графів?
13. Чим відрізняються між собою список ребер, матриця суміжності, матриця інцидентності та список суміжності? Які їх основні переваги та недоліки?
14. Як впливає тип задачі на вибір структури зберігання графа?
15. У яких випадках матриця суміжності буде симетричною?
16. Яка обчислювальна складність конвертації одного типу матриці графу в інший?
17. Що таке обхід графу? Які є основні алгоритми обходу графів? Яка мета обходу графу?
18. Чим відрязняється шлях і маршрут на графі?
19. У чому полягає різниця між пошуком у глибину і пошуком у ширину?

20. Як реалізується пошук у глибину? Яка його складність?

21. Як реалізується пошук у ширину? Яка його складність?

22. Чи можна комбінувати пошук у глибину з пошуком у ширину і навпаки?

23. Чи може пошук у глибину знайти найкоротший шлях у неорієнтованому графі? Чому?

24. Як можна використовувати пошук у глибину для перевірки зв’язності графу?

25. Що таке зв’язний граф? Як знайти всі компоненти зв’язності у графі?

26. Скільки компонент зв’язності буде у графі без ребер?

27. Як знайти сильні компоненти зв’язності у орієнтованому графі?

28. Як можна визначити наявність циклу у графі?

29. Як за допомогою алгоритмів обходу графів визначити кількість компонент зв’язності?

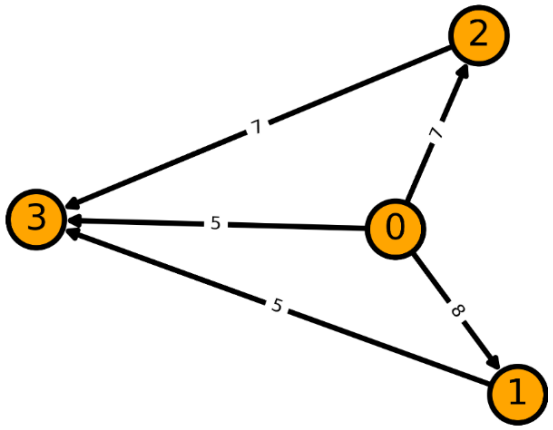
РОЗДІЛ 3 ПОШУК НАЙКОРОТШИХ МАРШРУТІВ МІЖ ВЕРШИНАМИ ГРАФУ

3.1 Зважений граф

Граф G називається *зваженим* або *графом зі зваженими ребрами*, якщо задане відображення множини E на множину дійсних чисел:

$$\omega : E \to R.$$

Дійсні числа $c_k = \omega(e_k)$, що характеризують кожне ребро, називаються *вагами ребер*. Отже, кожне ребро має деяке числове значення, яке характеризує перехід між вершинами, наприклад, тривалість, вартість, ймовірність тощо. Якщо ваги ребер задано натуральними числами (або елементами будь-якої скінченної множини), тоді можна розфарбувати граф – взяти набір фарб та перенумерувати їх відповідно до цих чисел. Граф зі зваженими ребрами може бути як орієнтованим, так і неорієнтованим. Для прикладу, на рис. 3.1 подано орієнтований зважений граф.



	0	1	2	3
0	∞	8	7	5
1	∞	∞	∞	5
2	∞	∞	∞	7
3	∞	∞	∞	∞

Рисунок 3.1 – Зважений орієнтований граф та його матриця відстаней

Зважені графи широко застосовуються в різних сферах, де необхідно моделювати системи з вагою зв’язків. Наприклад, у логістиці та транспортних мережах такий граф використовується для пошуку найкоротшого маршруту з урахуванням відстані, часу або вартості перевезення. В комп’ютерних мережах такими графами моделюють маршрутизацію пакетів між вузлами з урахуванням затримок або пропускної здатності каналів. У штучному інтелекті та теорії ігор їх застосовують для моделювання випадковості переходів між станами системи.

Для подання зваженого графу можна модифікувати типові способи подання графів – список ребер, список суміжності, матрицю суміжності та матрицю інцидентності. Для матриці суміжності замість бінарних значень 1 та 0, що позначають наявність або відсутність ребра, необхідно вказати вагу ребра. Таку модифіковану матрицю суміжності називають *матрицею відстаней* графу. Якщо між двома вершинами немає ребра, то зазвичай вказується дуже велике число (умовна нескінченність) або спеціальне значення `NaN` (Not a Number), що в подальшому буде коректно оброблятись алгоритмами обходу графів. Приклад такого подання зображено на рис. 3.1. Якщо зважений граф задавати списком суміжних вершин, тоді необхідно зберігати і ваги ребер. Це можна реалізувати структурою даних «пара», що поєднує в собі 2 елементи – вершину і вагу ребра, яке входить у вершину. Приклад такого списку суміжності наведено на рис. 3.2.

0	(1, 8)	(2, 7)	(3, 5)
1	(3, 5)		
2	(3, 7)		
3			

Рисунок 3.2 – Список суміжності зваженого графу з рис. 3.1

3.2 Знаходження найкоротших маршрутів за алгоритмом Дейкстри

Задача знаходження найкоротшого маршруту на зваженому графі від однієї вершини до всіх інших розв'язується алгоритмом Дейкстри. Алгоритм названо за прізвищем його автора – Едсгера Дейкстри (Edsger Wybe Dijkstra). Один із практичних прикладів цієї задачі – пошук найкоротшого маршруту пересування містом на велосипеді. У алгоритмі Дейкстри задається стартова вершина v_s і знаходяться довжини найкоротших маршрутів від неї до усіх інших вершин.

Алгоритм Дейкстри (Dijkstra's algorithm) полягає у поступовій побудові маршрутів від v_s до всіх інших вершин по ребрах або дугах мінімальної ваги. Для цього подамо граф матрицею відстаней **A** розміром $n \times n$. Якщо якесь ребро або якась дуга відсутні, значення відповідного елемента матриці **A** встановимо нескінченним. На виході алгоритму отримуємо масив *d* розміром n, координатами якого є довжини найкоротших маршрутів від вершини v_s до усіх інших. Для відслідковування відвіданих вершин використаємо булевий масив *u* довжини n, в якому будемо помічати, які вершини відвідано.

На початку роботи алгоритма знаходимось у вершині v_s; жодна вершина ще не переглянута, тому $u_i = 0, i = \overline{1, n}$. У масиві *d* тільки одне s-те

значення дорівнює 0 (це найкоротший маршрут від s-ї вершини до самої себе) – $d_s = 0$, а всі інші координати встановлено у нескінченність: $d_i = \infty,\ i = 1, 2, ..., s-1, s+1, ..., n$. Сам алгоритм складається з n ітерацій. На кожній ітерації виконуються такі кроки.

Крок 1. Серед ще не переглянутих вершин (тобто поміж тих, для яких $u_i = 0$), вибираємо вершину, для якої відповідне значення в масиві d є найменшим. Нехай її номер i, тобто аналізується вершина v_i. Якщо таких вершин декілька, вибираємо будь-яку. На першій ітерації завжди вибираємо стартову вершину v_s. Помічаємо вершину v_i як переглянуту: $u_i = 1$.

Крок 2. Для всіх непереглянутих вершин перераховуємо довжину найкоротшого маршруту від v_s до кожної з них. Наприклад, вершина v_t ще непереглянута. У матриці **A** елемент a_{it} – це довжина шляху з v_i до v_t. У масиві d значення d_i та d_t – це поточні довжини найкоротших маршрутів $v_s \to v_i$ та $v_s \to v_t$. Якщо маршрут від v_s до v_t через v_i є коротшим за поточний, тоді знайдено коротший маршрут і масив d потрібно оновити: $d_t = \min(d_t,\ d_i + a_{it}),\ \forall t : u_t = 0$. Така операція називається релаксацією непереглянутих вершин. На цьому поточна ітерація закінчується.

Крок 3. Якщо є ще непереглянуті вершини, повертаємося до *Кроку 2*. Якщо усі n вершин переглянуто, закінчуємо роботу алгоритму.

Алгоритм Дейкстри працює коректно лише коли ваги ребер задано додатними числами. Обчислювальна складність базового алгоритму Дейкстри становить $O\left(|V|^2\right)$. Реалізації алгоритму з використанням черги з пріоритетом мають нижчу складність – $O\left(|V| \cdot \log|V| + |E|\right)$.

Внаслідок роботи алгоритму Дейкстри отримуємо масив d з довжинами найкоротших маршрутів з v_s до кожної із вершин графу. Для практичного застосування результатів важливо також мати і самі найкоротші маршрути у формі послідовності вершин від v_s до усіх інших. Для цього необхідно відслідковувати вершини, за якими сформовано найкоротші маршрути з поточної вершини v_i. Сформуємо масив предків p довжини n. Кожен його елемент p_i зберігає номер вершини, яка передує вершині v_i у найкоротшому маршруті з v_s. Для стартової вершини v_s: $p_s = 0$, бо у неї немає предків. За такого масиву предків найкоротший маршрут до вершини v_t можна побудувати з кінця, додаючи у шлях наступну вершину – предка попередньої, допоки предком не стане початкова вершина: $v_t \leftarrow p(v_t) \leftarrow p(p(v_t)) \leftarrow p(p(p(v_t))) \leftarrow ... \leftarrow v_s$. Елементи в масиві предків P оновлюються, коли за умови $d_i + a_{it} < d_t$ виконується релаксація вершини.

Саме тоді і встановлюємо нового предка вершини v_t: $p_t = i$. Формування та модифікацію масиву предків необхідно додати у відповідні кроки описаного вище алгоритма Дейкстри.

Розглянемо роботу алгоритма Дейкстри – на графі з рис. 3.3 знайдемо довжини найкоротших маршрутів від вершини *D* до усіх інших вершин.

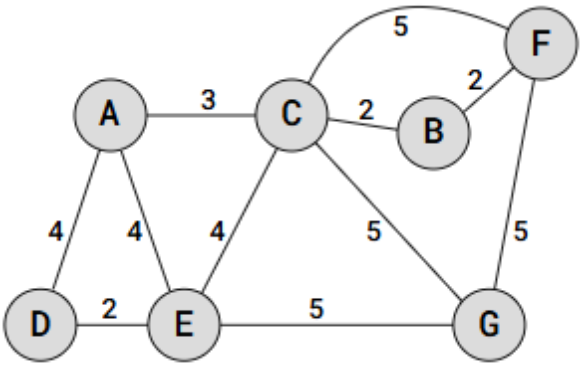


Рисунок 3.3 – Граф для ілюстрації роботи алгоритма Дейкстри

Спершу встановимо початкові відстані від вершини *D* до всіх інших. Усі вони дорівнюють нескінченності, окрім, звичайно, вершини *D* (рис. 3.4). Відстань від вершини *D* до кожної вершини вказана всередині цієї вершини. Ці значення запишемо в масив *d*.

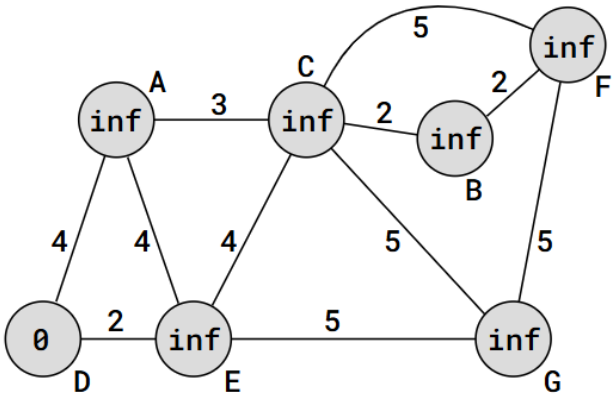


Рисунок 3.4 – Початкова розмітка графу з рис. 3.3

На першій ітерації вибирається перша невідвідана вершина з найменшим значенням в масиві *d*. Нею буде стартова вершина $v_s = D$. Для усіх вершин, що ще не переглянуті та суміжні з v_s, перераховуємо довжину найкоротшого маршруту від v_s до кожної з них. Такими вершинами є *A* та *E*. Результат релаксації запишемо всередину вершин *A* та *E* (рис. 3.5); червоним

кольором запишемо значення в масиві предків. Оскільки вершина D напряму з'єднана з вершинами A та E, то відстань дорівнює вагам ребер між ними.

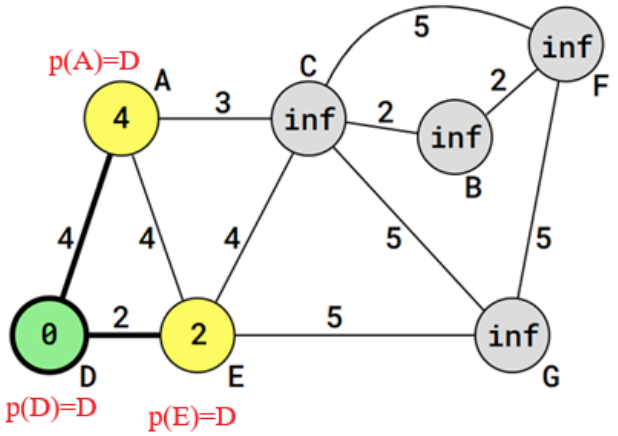


Рисунок 3.5 – Розмітка графу з рис. 3.3 після першої ітерації алгоритму Дейкстри

Після релаксації вершин A та E, вершина D вважається відвіданою, і повертатися в неї не будемо. Наступною вершиною для релаксації, вибирається та, що має найменшу відстань в масиві d. Оскільки вершина D вже відвідана, то такою вершиною буде E. Відстань до усіх суміжних і раніше невідвіданих вершин від вершини E тепер має бути обчислена і якщо необхідно – оновлена. Відстань від D до вершини A через E дорівнює 2+4=6. Але поточна відстань до вершини A дорівнює 4, що менше, тому відстань до вершини A не оновлюємо. Відстань до вершини C обчислюється як 2+4=6, що менше нескінченності, тому відстань до вершини C оновлюємо. Аналогічно, відстань до вершини G обчислюється та оновлюється до 2+5=7. Результат подано на рис. 3.6.

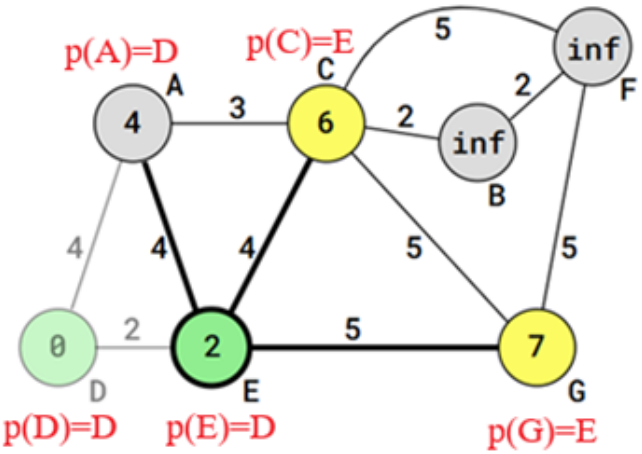


Рисунок 3.6 – Розмітка графу з рис. 3.3 після другої ітерації алгоритму

Повторимо ітерації, поки не будуть відвідані всі вершини. Остаточний результат роботи алгоритму Дейкстри подано на рис. 3.7.

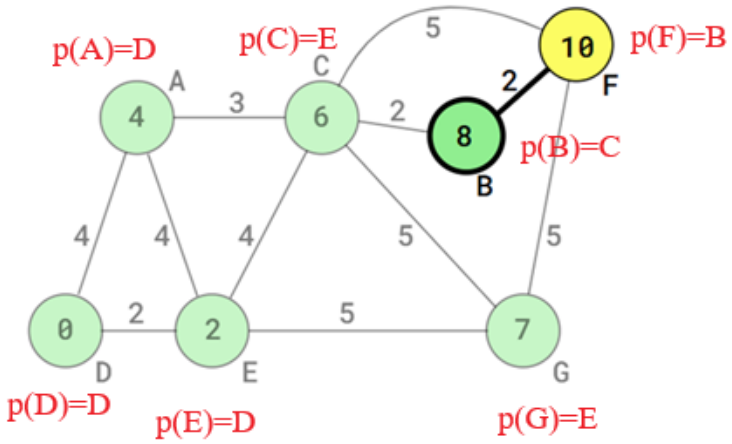


Рисунок 3.7 – Остаточна розмітка графу з рис. 3.3 після виконання алгоритму Дейкстри

Для відновлення найкоротшого маршруту скористаємося масивом предків p. Для прикладу відновимо найкоротший шлях до вершини F. В масиві p предком вершини F є $p(F) = B$, для вершини B предком є $p(B) = C$ тощо. В такий спосіб добираємося до початкової вершини D. Найкоротший маршрут від D до F є таким $F \leftarrow B \leftarrow C \leftarrow E \leftarrow D$. Його довжина дорівнює 10. За аналогією отримуємо такі найкоротші маршрути до інших вершин:

$B \leftarrow C \leftarrow E \leftarrow D$, довжиною 8;

$G \leftarrow E \leftarrow D$ довжиною 7;

$C \leftarrow E \leftarrow D$ довжиною 6;

$E \leftarrow D$ довжиною 2;

$A \leftarrow D$ довжиною 8.

Псевдокод алгоритму Дейкстри подано нижче. Для швидкого пошуку наступної вершини з найменшою відстанню в масиві d використовується черга з пріоритетом.

```
function Dijkstra(Graph, source):
    PQ = PriorityQueue()   # Min-Heap (distance, node)
    d = {v: ∞ for v in Graph}
    p = {v: NULL for v in Graph}

    d[source] = 0
    p[source] = 0
    PQ.insert(0, source)
    VISITED = set()
```

```
while PQ is not empty:
    dist, u = PQ.extractMin()
    VISITED.add(u)
    for v, w in Graph[u]:
        if v not in VISITED and d[u] + w < d[v]:
            d[v] = d[u] + w
            p[v] = u
            PQ.insert(d[v], v)

return d, p
```

3.3 Знаходження найкоротших маршрутів за алгоритмом Флойда–Воршелла

В попередньому підрозділі розглянуто пошук найкоротшого маршруту від однієї вершини до всіх інших. На практиці також є потреба пакетного знаходження найкоротших маршрутів між усіма парами вершин. Зокрема, це може використовуватись у транспортних графах зі стабільним трафіком, коли оптимальні маршрути знаходяться один раз, зберігаються в пам'яті і активуються за запитом без застосування трудомістких процедур їх синтезу. Найпростіший спосіб знайти всі маршрути – застосувати алгоритм Дейкстри послідовно до кожної вершини графу. Але він працюватиме ефективно лише для графів з невисокою щільністю. Адже для великих графів кількість повторних знаходжень найкоротших фрагментів маршрутів стає великою. Тому на практиці є зацікавленість у алгоритмах маршрутизації, які не використовують надлишкові процедури, як це має місце під час ітераційного запуску алгоритму Дейкстри з різних початкових вершин. Таким алгоритмом є алгоритм Флойда–Воршелла, який інколи називають алгоритомом Флойда, алгоритмом Роя–Флойда або алгоритмом Флойда–Роя–Воршелла. Назва алгоритму утворена за прізвищами його авторів – Роберта Флойда (Robert Floyd), Стівена Воршелла (Stephen Warshall) та Бернарда Роя (Bernard Roy).

Алгоритм Флойда–Воршелла (Floyd–Warshall algorithm) знаходить найкоротші маршрути та їх довжини між усіма парами вершин графу одночасно. Він базується на тій самій операції релаксації, що і алгоритм Дейкстри. Проте, на відміну від нього, застосовує релаксацію до всіх можливих проміжних вершин. Для роботи цього алгоритму необхідно мати матрицю відстаней **A** розміром $n \times n$ (так само, як і у випадку з алгоритмом Дейкстри). Суть алгоритму полягає у перевірці існування коротшого маршруту від вершини v_i до вершини v_j через деяку транзитну вершину v_k. Така перевірка називається застосуванням трикутного оператора, що може призвести до релаксації дуги $v_i \to v_j$ (рис. 3.8). Якщо маршрут $v_i \to v_k \to v_j$

коротший за маршрут $v_i \to v_j$, тобто $a_{ij} > a_{ik} + a_{kj}$, тоді оновлюється довжина поточного найкоротшого маршруту $v_i \to v_j$. Трикутний оператор застосовується для усіх можливих трійок різних вершин v_i, v_j та v_k. Назагал, є $n \cdot (n-1)$ різних пар вершин v_i та v_j, для яких є $(n-2)$ транзитних вершин v_k, тому обчислювальна складність алгоритму Флойда–Воршелла становить $O(n^3)$.

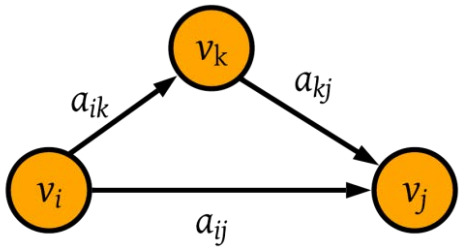


Рисунок 3.8 – Трикутний оператор для вершин v_i, v_j та v_k

Внаслідок роботи алгоритму Флойда–Воршелла початкова матриця відстаней **A** перетворюється на матрицю найкоротших відстаней. В алгоритмі Дейкстри для відновлення найкоротших маршрутів використовувався вектор предків p довжиною n. У випадку алгоритму Флойда–Воршелла для знаходження найкоротших маршрутів необхідно використовувати матрицю предків **P** розміром $n \times n$, оскільки відновлюються маршрути між усіма можливими парами вершин. Кожний елемент p_{ij} в матриці **P** – це предок вершини v_j у найкоротшому маршруті $v_i \to ... \to v_j$. Якщо $p_{ij} = v_j$ – це означає, що більше проміжних вершин немає, і вершині v_j передує лише початкова вершина v_i.

Матриця **P** формується таким чином. Якщо на графі є дуга $v_i \to v_j$, тоді $p_{ij} = v_j$. Якщо дуга $v_i \to v_j$ відсутня, тоді p_{ij}=NaN. Якщо відбувається релаксація дуги $v_i \to v_j$ за допомогою транзитної вершини v_k, тоді в комірку p_{ij} записується v_k – новий предок вершини v_j.

Псевдокод алгоритму Флойда–Воршелла подано нижче.

```
function FloydWarshall(graph):
    A = nxn matrix // заповнена значеннями ∞
    P = nxn matrix // заповнена значеннями NaN
```

```
    for (u, v, w) in graph:
        A[u][v] = w
        P[u][v] = v
    for v in graph:
        A[v][v] = 0
        P[v][v] = v
    for k from 1 to n:
        for i from 1 to n:
            for j from 1 to n:
                if i == k or j == k:
                   continue
                if A[i][j] > A[i][k] + A[k][j]:
                    A[i][j] = A[i][k] + A[k][j]
                    P[i][j] = k
    return d, p
```

Для відновлення маршруту між вершинами u та v необхідно обійти матрицю предків $\mathbf{P}$ за таким алгоритмом

```
function Path(u, v, P) is
    if P[u][v] = NaN:
        return []
    path = [v]
    prev_v = -1
    while prev_v != v
        prev_v = v
        v = P[u][v]
        path.prepend(v)
    path.prepend(u)
    return path
```

Розглянемо приклад роботи алгоритму Флойда–Воршелла для графу з рис. 3.9.

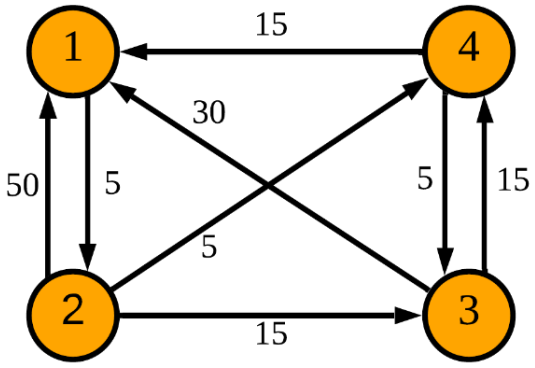


Рисунок 3.9 – Тестовий граф для ілюстрації роботи алгоритму

Матриця відстаней **A** та матриця предків **P** на початку роботи алгоритму є такими:

$$\mathbf{A}^{<0>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array};$$

$$\mathbf{P}^{<0>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 1 & 2 & \mathtt{NaN} & \mathtt{NaN} \\ 1 & 2 & 3 & 4 \\ 1 & \mathtt{NaN} & 3 & 4 \\ 1 & \mathtt{NaN} & 3 & 4 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array}.$$

На першій ітерації $k=1$. Тому застосуємо трикутний оператор з транзитною вершиною v_1 для усіх можливих пар вершин v_i та v_j. Елементи матриці відстаней перерахуємо таким чином:

$a_{23} = \min(a_{23}, a_{21} + a_{13}) = \min(15, 50 + \infty) = 15;$

$a_{24} = \min(a_{24}, a_{21} + a_{14}) = \min(5, 50 + \infty) = 5;$

$a_{32} = \min(a_{32}, a_{31} + a_{12}) = \min(\infty, 30 + 5) = 35;$

$a_{34} = \min(a_{34}, a_{31} + a_{14}) = \min(15, 30 + \infty) = 15;$

$a_{42} = \min(a_{42}, a_{41} + a_{12}) = \min(\infty, 15 + 5) = 20;$

$a_{43} = \min(a_{43}, a_{41} + a_{13}) = \min(5, 15 + \infty) = 5.$

На першій ітерації знайдено коротші маршрути від вершини v_3 до вершини v_2 та від вершини v_4 до вершини v_2. Матриці відстаней та матриця предків стали такими:

$$\mathbf{A}^{<0>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array} \Rightarrow \mathbf{A}^{<1>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array};$$

$$\mathbf{P}^{<0>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 1 & 2 & \mathtt{NaN} & \mathtt{NaN} \\ 1 & 2 & 3 & 4 \\ 1 & \mathtt{NaN} & 3 & 4 \\ 1 & \mathtt{NaN} & 3 & 4 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array} \Rightarrow \mathbf{P}^{<1>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 1 & 2 & \mathtt{NaN} & \mathtt{NaN} \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 3 & 4 \\ 1 & 1 & 3 & 4 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array};$$

На другій ітерації транзитною буде вершина v_2. Матриця відстаней та матриця предків стануть такими:

$$\mathbf{A}^{<1>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array} \Rightarrow \mathbf{A}^{<2>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array};$$

$$\mathbf{P}^{<1>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 1 & 2 & \mathtt{NaN} & \mathtt{NaN} \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 3 & 4 \\ 1 & 1 & 3 & 4 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array} \Rightarrow \mathbf{P}^{<2>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 1 & 2 & 2 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 3 & 4 \\ 1 & 1 & 3 & 4 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array}.$$

На третій ітерації транзитною буде вершина v_3. Матриця відстаней та матриця предків стануть такими:

$$\mathbf{A}^{<2>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array} \Rightarrow \mathbf{A}^{<3>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array};$$

$$\mathbf{P}^{<2>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 1 & 2 & 2 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 3 & 4 \\ 1 & 1 & 3 & 4 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array} \Rightarrow \mathbf{P}^{<3>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 1 & 2 & 2 & 2 \\ 3 & 2 & 3 & 4 \\ 1 & 1 & 3 & 4 \\ 1 & 1 & 3 & 4 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array}.$$

На четвертій ітерації транзитною буде вершина v_4. В результаті отримуємо такі матрицю найкоротших відстаней та матрицю предків:

$$\mathbf{A}^{<3>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array} \Rightarrow \mathbf{A}^{<4>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array};$$

$$\mathbf{P}^{<3>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 1 & 2 & 2 & 2 \\ 3 & 2 & 3 & 4 \\ 1 & 1 & 3 & 4 \\ 1 & 1 & 3 & 4 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array} \Rightarrow \mathbf{P}^{<4>} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \left\| \begin{matrix} 1 & 2 & 4 & 2 \\ 4 & 2 & 4 & 4 \\ 1 & 1 & 3 & 4 \\ 1 & 1 & 3 & 4 \end{matrix} \right\| \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array}.$$

Розглянемо приклад відновлення найкоротшого маршруту з вершини v_1 до вершини v_3 за матрицею предків $\mathbf{P}^{<4>}$. В тій матриці $p_{13} = 4$, отже вершина v_4 передує вершині v_3: $v_1 \to ? \to v_4 \to v_3$.

Тепер подивимося, яка вершина передує v_4 в фрагменті маршрута $v_1 \to ? \to v_4$. В матриці передків $p_{14} = 2$, тобто вершина v_2 передує вершині v_4: $v_1 \to ? \to v_2 \to v_4 \to v_3$.

Подивимося, яка вершина передує v_2 в фрагменті маршруту $v_1 \to ? \to v_2$. В матриці передків $p_{12} = 2$, а це означає, що проміжних вершин немає.

Отже, найкоротший маршрут з вершини v_1 до вершини v_3 є таким: $v_1 \to v_2 \to v_4 \to v_3$.

3.4 Завдання для самостійного дослідження

Базовий рівень

1. Дослідити алгоритм Дейкстри для знаходження найкоротшого маршруту між двома вершинами *A* та *B*. Дослідження провести на карті доріг деякого міста відповідно до індивідуального варіанта. Вершини *A* та *B* – це географічні точки в місті, між якими необхідно знайти найкоротший маршрут для проїзду на транспорті.

Карту доріг пропонується отримати з відкритих даних сайту OpenStreetMap за посиланням – https://www.openstreetmap.org/. Рекомендується використати бібліотеку Python – osmnx.

Для реалізації іншими мовами програмування можна використати API OpenStreetMap – https://wiki.openstreetmap.org/wiki/API.

Програмно реалізувати алгоритм Дейкстри для знаходження найкоротшого маршруту між двома вершинами *A* та *B*. Відповідно до індивідуального варіанта та відкритих даних карт доріг OpenStreetMap візуалізувати граф. Випадковим чином вибрати вершини *A* та *B* для подальших досліджень. Вершини вибрати таким чином, щоб кількість ребер на найкоротшому маршруті між вершинами *A* та *B* була щонайменше 15. Для вибраних вершин знайти найкоротший маршрут за алгоритмом Дейкстри. Результат оформити аналогічно до рис. 3.10, на якому показано найкоротший маршрут між двома вершинами. Рисунок створено за допомогою бібліотеки osmnx.

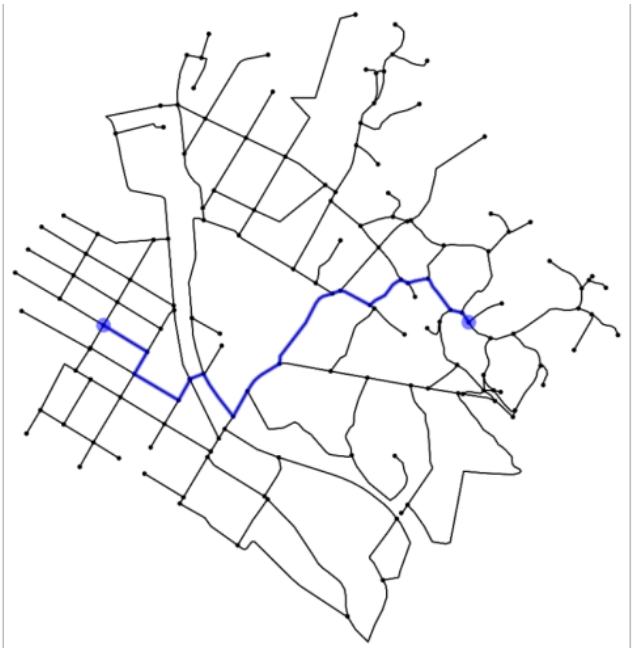


Рисунок 3.10 – Найкоротший маршруту між двома вершинами

2. Дослідити динаміку зміни маршруту залежно від зміни дорожнього трафіка. Уявімо, що водій хоче проїхати від точки *A* до точки *B* за найкоротшим маршрутом, який знайдено алгоритмом Дейкстри. Проте, окрім нього ще кілька водіїв за підказкою навігатора також виберуть цей найкоротший маршрут. Відповідно, змінюється трафік, зокрема збільшується вага ребер графа. І через деякий час нові водії виберуть інший маршрут від *A* до *B*, бо він буде коротшим за нової дорожньої ситуації. Необхідно промоделювати зміни трафіка та його вплив на синтез найкоротшого маршруту. Пропонується моделювання реалізувати за таким сценарієм:

- за алгоритмом Дейкстри знаходимо найкоротший маршрут від вершини *A* до вершини *B*;
- імітуємо зміну трафіка – збільшуємо вагу випадкового ребра поточного найкоротшого маршруту на 15–25%, а вагу ребер, які не увійшли до найкоротшого маршруту, зменшуємо на 1%;
- повторюємо ці кроки 1000 разів.

Результати моделювання доцільно подати у вигляді графіків, аналогічних до рис. 3.11–3.13. Часова діаграма на рис. 3.11 показує зміни довжини найкоротшого маршруту залежно від зміни дорожнього трафіка. Видно, що з часом довжина найкоротшого маршруту може як збільшуватися, так і зменшуватися. Проте, результат суттєво залежатиме від структури початкового графу та ваг ребер. З гістограми на рис. 3.12 видно, що приблизно половина маршрутів в умовах зміни трафіка мають не більше п’яти спільних ребер з початковим маршрутом, тобто суттєво відрізняються

від першого розв’язку задачі. Під час експерименту початковий найкоротший маршрут із 18 ребер залишився без змін лише 6 разів, це означає що початковий маршрут за вибраних параметрів моделювання є дуже чутливим до зміни трафіка.

На рис. 3.13 подано карту доріг з найпопулярнішими маршрутами, знайденими під час моделювання зміни умов трафіка. Видно, що початковий маршрут сюди не входить. Всі інші маршрути в більшості випадків містять у собі ребра початкового найкоротшого маршруту.

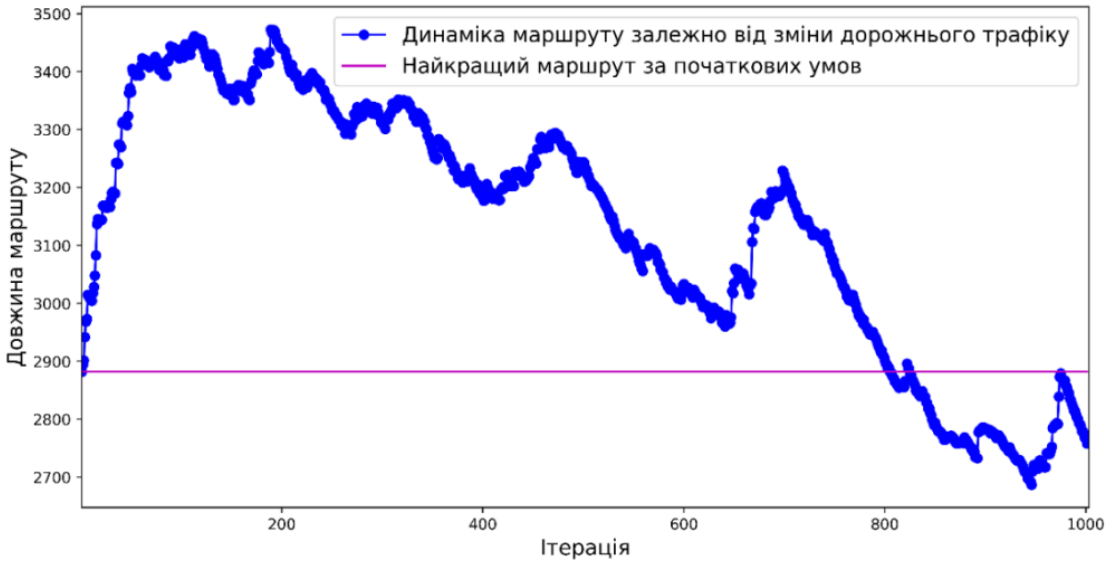


Рисунок 3.11 – Динаміка довжин найкоротших маршрутів

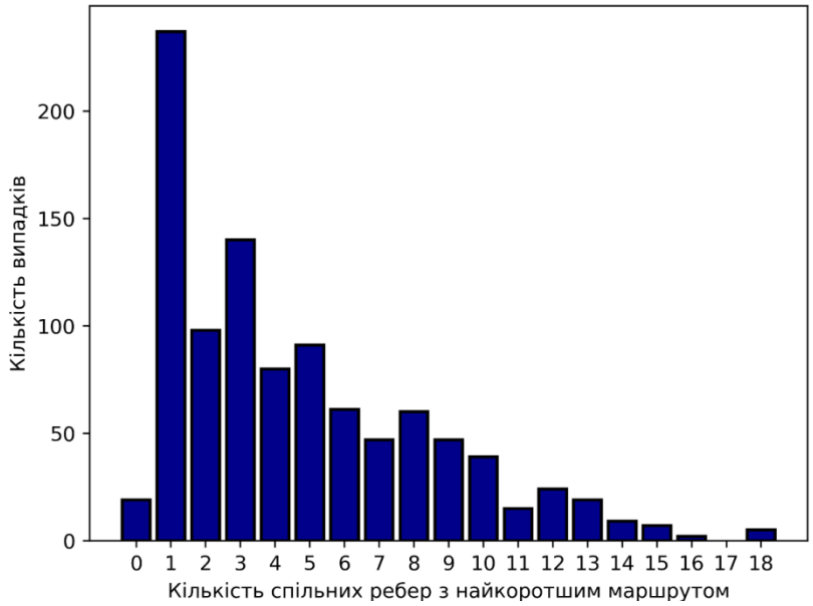


Рисунок 3.12 – Диверсифікація найкоротших маршрутів

Рисунок 3.13 – Топ-10 найпопулярніших маршрутів

За результатами експериментів необхідно також виявити: 1) момент переходу на суттєво інший найкоротший маршрут, в якому від початкового маршруту залишилося менше 1/3 ребер; 2) момент переходу на абсолютно інший найкоротший маршрут, в якому від початкового не залишилося жодного ребра.

3. Програмно реалізувати алгоритм Флойда–Воршелла. Граф для дослідження вибирається відповідно до варіанта. Для кожного знайденого найкоротшого маршруту порахувати кількість ребер. Результати подати у вигляді графіків, аналогічних до рис. 3.14 та 3.15. Виявити, якому теоретичному розподілу відповідають експериментальні дані.

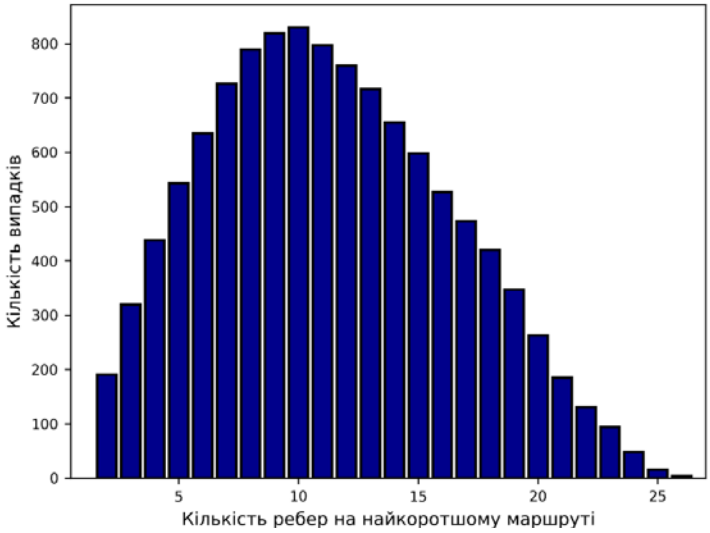


Рисунок 3.14 – Розподіл кількості ребер на найкоротших маршрутах

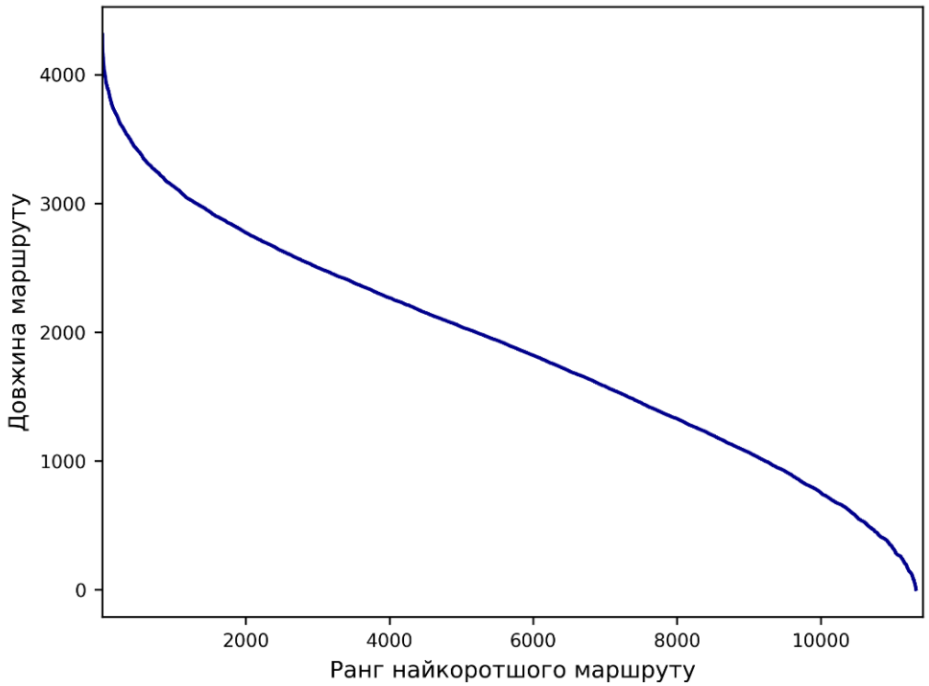


Рисунок 3.15 – Розподіл довжин найкоротших маршрутів

4. Дослідити важливість вершин в знайдених найкоротших маршрутах. Вершину вважатимемо важливою, якщо вона зустрічається в багатьох найкоротших маршрутах. В транспортних графах таку вершину можна назвати критичною, оскільки через неї проходить багато найкоротших шляхів. Для оцінювання важливості вершини v необхідно знати кількість найкоротших маршрутів між будь-якими вершинами, що проходять через v. Кількісно важливість вершини v оцінимо так:

$$g(v) = \frac{N_v}{N},$$

де N_v – кількість найкоротших маршрутів, в яких v є однією із проміжних вершин;

N – кількість найкоротших маршрутів, в яких потенційно може фігурувати v як проміжна вершина.

Для орієнтованих графів $N = (n-1)(n-2)$, а для неорієнтованих – $N = \frac{(n-1)(n-2)}{2}$.

Результати дослідження необхідно подати у вигляді графіків, аналогічних до рис. 3.16–3.17. На рис. 3.16 подано розподіл важливостей вершин, а на рис. 3.17 позначено розташування топ-10 найбільш важливих вершин.

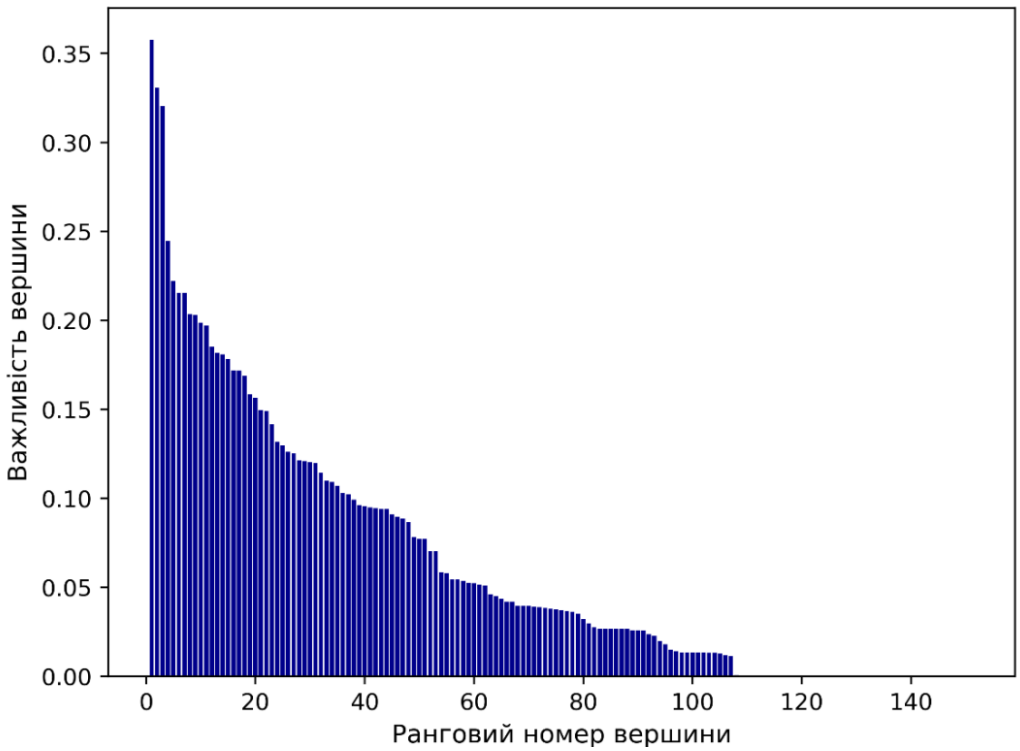


Рисунок 3.16 – Ранговий розподіл важливості вершин за частотою їх появи у найкоротших маршрутах

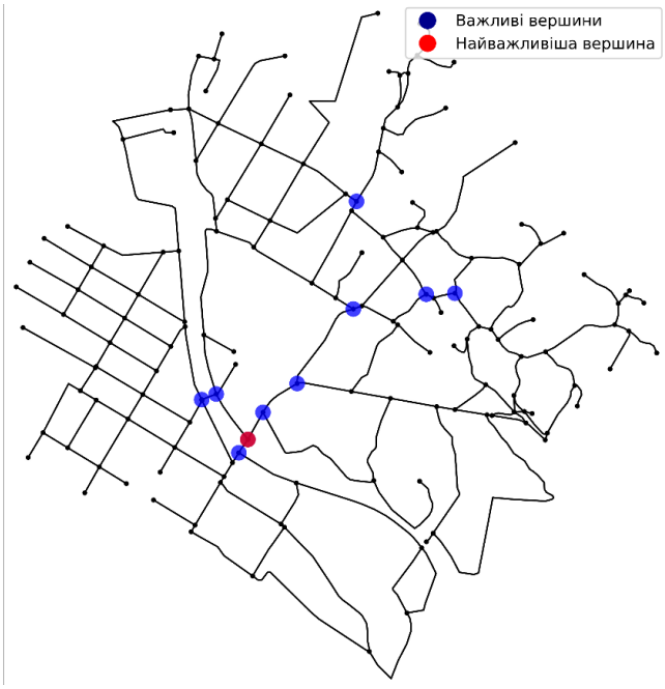


Рисунок 3.17 – Топ-10 вершин за важливістю

Поглиблений рівень

1. Видалити всі дуги найважливішої вершини та перевірити як це вплине на найкоротші маршрути та важливість інших вершин. Найважливіша вершина зазвичай є критичною, її руйнування значно погіршує характеристики мережі. Такі вершини часто вибирають для атак, а з іншого боку докладають зусиль, витрачають додаткові ресурси для її захисту. Під час прийняття рішення про виділення ресурсів для захисту чи для атаки варто знати наслідки руйнування найважливішої вершини. Результати моделювання потрібно подати графіками, аналогічними до рис. 3.18–3.20. На рис. 3.18 подано нові топ-10 найважливіших вершин; порівнюючи з рис. 3.17 бачимо суттєві зміни.

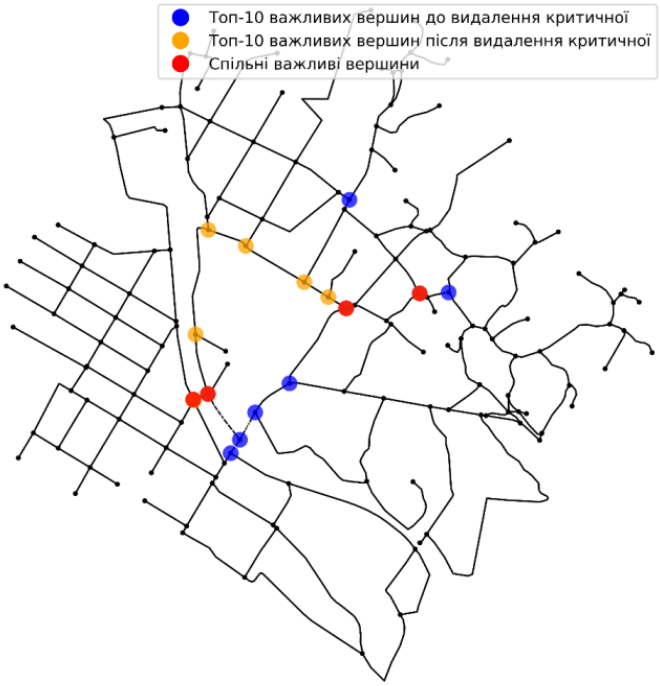


Рисунок 3.18 – Зіставлення найбільш важливих вершин *до* та *після* атаки

На рис. 3.19 подано гістограму частот різниць рангів важливостей вершин до видалення та після видалення найважливішої вершини. Видно, що більшість вершин свої позиції майже не змінили, проте деякі вершини стрімко опустилися в рейтингу, а деякі стрімко піднялися. Саме вершини, які стрімко піднялися у рейтингу потребують деталізованого аналізу, саме на них зав’язуються нові маршрути після руйнування лідера. На рис. 3.20 подано гістограму частот різниці довжин найкоротших маршрутів до і після видалення найважливішої вершини. Довжини багатьох маршрутів суттєво збільшились.

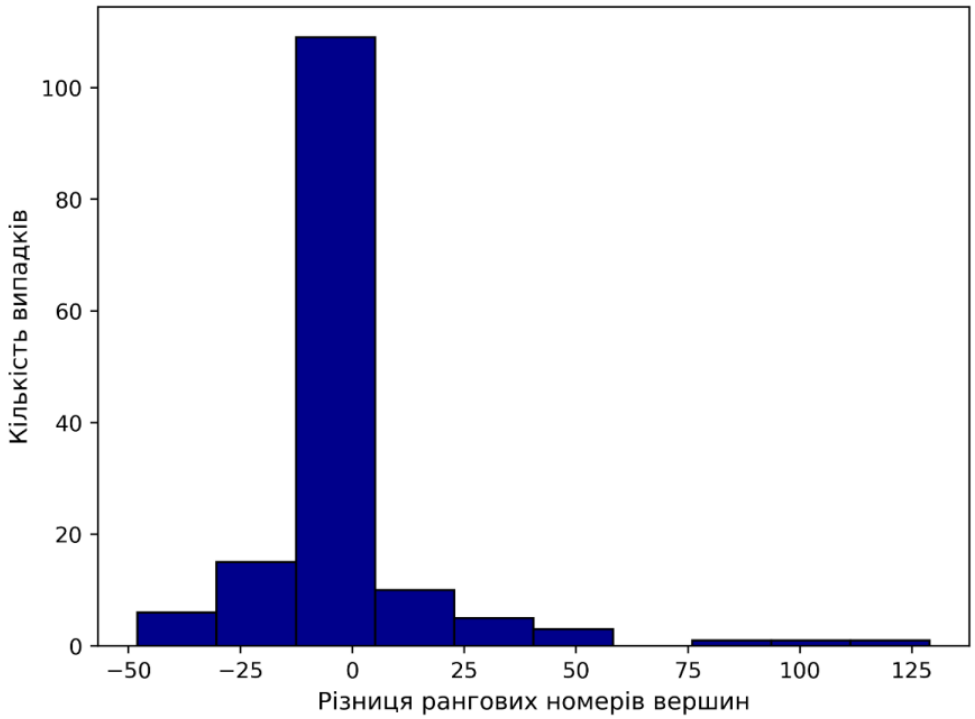


Рисунок 3.19 – Зіставлення важливості вершин *до* та *після* видалення найважливішої вершини

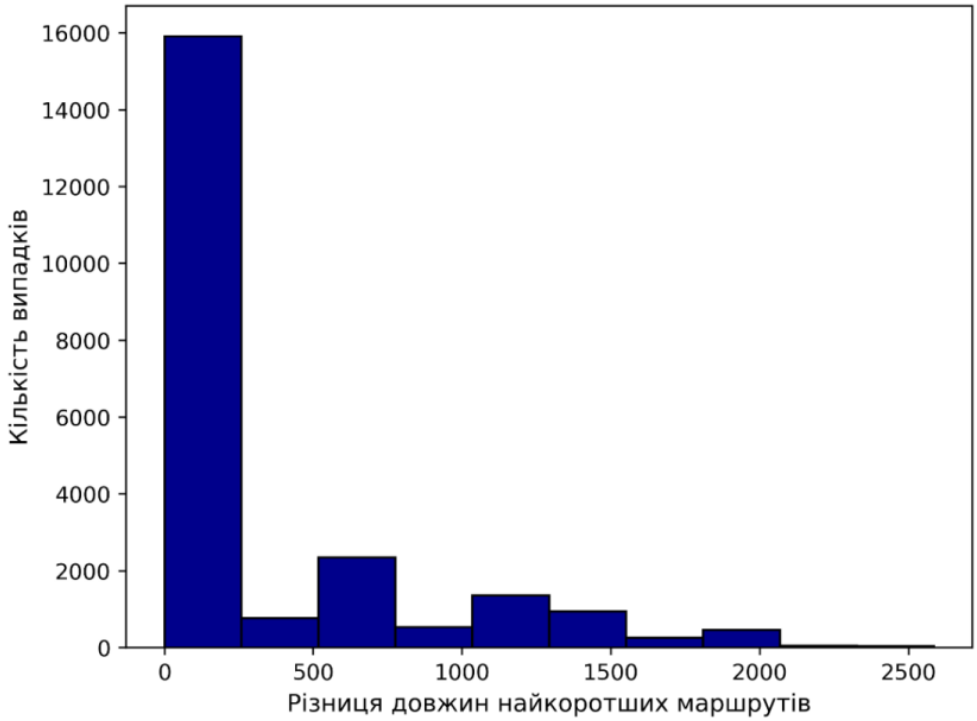


Рисунок 3.20 – Зіставлення довжин найкоротших маршрутів *до* та *після* видалення найважливішої вершини

Таблиця 3.1 – Варіанти завдань

Варіант	Місто	Варіант	Місто
1	Вінниця	26	Кам’янець-Подільський
2	Київ	27	Мукачево
3	Одеса	28	Конотоп
4	Харків	29	Умань
5	Львів	30	Олександрія
6	Ужгород	31	Дрогобич
7	Івано-Франківськ	32	Бердичів
8	Чернівці	33	Шостка
9	Хмельницький	34	Ізмаїл
10	Рівне	35	Ковель
11	Луцьк	36	Ніжин
12	Тернопіль	37	Сміла
13	Суми	38	Ірпінь
14	Полтава	39	Калуш
15	Кропивницький	40	Бориспіль
16	Дніпро	41	Коростень
17	Кривий Ріг	42	Коломия
18	Миколаїв	43	Стрий
19	Херсон	44	Чорноморськ
20	Черкаси	45	Лозова
21	Кам’янське	46	Прилуки
22	Краматорськ	47	Нововолинськ
23	Нікополь	48	Охтирка
24	Слов’янськ	49	Марганець
25	Павлоград	50	Ізюм

3.5 Питання для самоконтролю та професійного розвитку

1. Що таке зважений граф? У чому його особливість порівняно з незваженим графом?
2. Для чого використовуються ваги ребер у графі?
3. Для яких реальних задач доцільно використовувати зважені графи?
4. Яка основна ідея алгоритму Дейкстри?
5. Яка обчислювальна складність алгоритму Дейкстри?
6. Які структури даних зазвичай використовуються для реалізації алгоритму Дейкстри?
7. Чому алгоритм Дейкстри не працює з графами, що мають ребра з від’ємною вагою? Які є альтернативні алгоритми для знаходження найкоротших маршрутів на таких графах?
8. Чи можна знайти за алгоритмом Дейкстри найкоротші шляхи за неадитивних ваг ребер, наприклад, для ймовірносних графів?

9. Як впливає топологія графу (щільність, розмірність) на ефективність алгоритму Дейкстри?

10. Як подати зважений граф програмно?

11. Яким чином відбувається оновлення мінімальних відстаней до сусідніх вершин у алгоритмі Дейкстри?

12. Як відновити найкоротші маршрути після виконання алгоритму Дейкстри?

13. Яка складність алгоритму Дейкстри за використання пріоритетної черги?

14. Наскільки ефективно використовувати алгоритм Дейкстри для пошуку найкоротшого маршруту лише між однією парою вершин графа?

15. Яку задачу розв’язує алгоритм Флойда–Воршелла?

16. Що спільного та відмінного у матриці відстаней та матриці предків?

17. Як формується матриця предків на початку виконання алгоритму?

18. Чи можливо за алгоритмом Флойда–Воршелла обробляти неорієнтовані графи?

19. Чи можливо за алгоритмом Флойда–Воршелла обробляти графи з кратними ребрами?

20. Чи можливо за алгоритмом Флойда–Воршелла обробляти графи, що мають цикли, усі дуги яких мають від’ємні ваги?

21. Яка обчислювальна складність алгоритму Флойда–Воршелла?

22. Як впливає щільність графа на кількість ітерацій алгоритму Флойда–Воршелла?

23. У чому полягає основна відмінність алгоритму Флойда–Воршелла від алгоритму Дейкстри?

24. Як впливає топологія графу (щільність, розмірність) на ефективність алгоритму Флойда–Воршелла?

25. Яким чином відбувається оновлення найкоротших маршрутів між вершинами у алгоритмі Флойда–Воршелла?

26. Чи можна розпаралелити алгоритм Флойда–Воршелла?

27. Яким чином ініціалізується матриця відстаней?

28. Яким чином ініціалізується матриця предків?

29. Як відновити найкоротші маршрути після виконання алгоритму Флойда–Воршелла?

30. Чи можливо за алгоритмом Флойда–Воршелла ідентифікувати, є граф зв’язним чи ні?

31. Чи можливо за алгоритмом Флойда–Воршелла розрахувати транзитивне замикання бінарного відношення?

32. Як знайти важливість вершини використовуючи найкоротші маршрути між усіма парами вершинами?

РОЗДІЛ 4 ЗАДАЧА КОМІВОЯЖЕРА

4.1 Постановка задачі комівояжера

Задача комівояжера (Travelling Salesman Problem, TSP) – це одна з класичних NP-складних задач комбінаторної оптимізації. Формулюється вона таким чином. Є комівояжер – бродячий торгівець, який має відвідати *n* міст, кожне рівно один раз, і повернутися у початкове місто. Такий цикл називається гамільтоновим; їх на графі може бути багато. За відомою матрицею відстаней можна сформувати задачу пошуку найкоротшого гамільтонового циклу – гамільтонового циклу з мінімальною сумарною вагою ребер. Знаходження такого гамільтонового циклу і є задачею комівояжера.

Задача комівояжера походить з дитячої головоломки Icosian Game, яку придумав Вільям Гамільтон (William Hamilton) в 1857 р. Задачу комівояжера насамперед вивчали в контексті організації поштових маршрутів. У 1930-х роках Карл Менгер (Karl Menger) розглядав її в рамках теорії графів. Значний інтерес до задачі комівояжера виник у 1950-х роках. В США навіть організували змагання про обхід 33 міст з великим призовим фондом (рис. 4.1). Сьогодні різноманітні варіанти задачі комівояжера застосовують в логістиці, робототехніці, плануванні виробництва тощо.

Рисунок 4.1 – Оголошення про змагання із вирішення задачі комівояжера

В термінах дискретної математики задачу комівояжера сформулюємо таким чином. Відомо: зважений граф $G=(V,E)$, де V – множина вершин потужністю $n=|V|$, а E – множина ребер, кожне з яких має вагу. Необхідно знайти цикл мінімальної ваги, що проходить через кожну вершину рівно один раз $v_1 \to v_2 \to v_3 \to ... \to v_n \to v_1$:

$$d(v_n, v_1)+\sum_{i=1}^{n-1} d(v_i,\ v_{i+1}) \to \min,$$

де $d(v_i,\ v_{i+1})$ – відстань між вершинами, що мають в маршруті номера i та $i+1$;

v_1 – початкова вершина маршруту.

Якщо відстань між будь-якими двома вершинами v_i та v_j та між вершинами v_j та v_i однакові – $d(v_i,\ v_j)=d(v_j, v_i)$, то така задача комівояжера називається симетричною. У іншому випадку – вона асиметрична. Симетрична задача комівояжера розглядається на неорієнтованому графі. Приклад такого графу з чотирма вершинами подано на рис. 4.2.

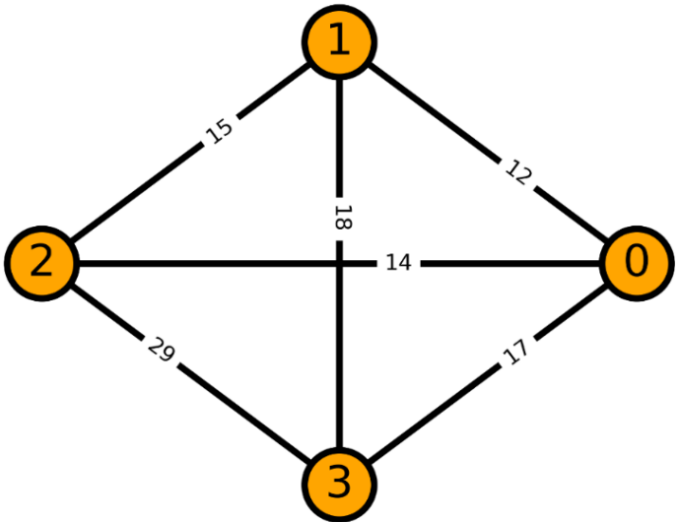


Рисунок 4.2 – Приклад неорієнтованого графу для задачі комівояжера

Усі можливі гамільтонові цикли на графі з рис. 4.2 наведено в табл. 4.1. Для поданого графу є декілька циклів мінімальної ваги, один з них – 0–2–1–3–0 має сумарну вагу ребер у 64. Як бачимо з таблиці, більшість гамільтонових циклів мають однакову вагу, проте це лише для поданого графу, для реальних задач таке спостерігається рідко. Всього для неорієнтованого повного графу з 4 вершинами є 24 гамільтонових цикли.

Граф із 5 вершин має вже 120 гамільтонових циклів Для графу з n вершин загальна кількість гамільтонових циклів становить $n!$. Для симетричної задачі комівояжера кількість унікальних гамільтонових циклів менша, бо деякі із них за різного порядку вершин мають однаковий набір ребер, наприклад, 0–1–3–2–0 та 0–2–3–1–0. Ці 2 цикли є дзеркальними, вони містять одні і ті самі ребра: 0–1 та 1–0, 1–3 та 3–1 тощо. Тому фактична кількість унікальних гамільтонових циклів для неорієнтованого графу становить $0.5n!$.

Таблиця 4.1 – Перелік усіх можливих гамільтонових циклів на графі з рис. 4.2

Гамільтонів цикл	Вага
0–1–2–3–0	73
0–1–3–2–0	73
0–2–1–3–0	64
0–2–3–1–0	73
0–3–1–2–0	64
0–3–2–1–0	73
1–0–2–3–1	73
1–0–3–2–1	73
1–2–0–3–1	64
1–2–3–0–1	73
1–3–0–2–1	64
1–3–2–0–1	73
2–0–1–3–2	73
2–0–3–1–2	64
2–1–0–3–2	73
2–1–3–0–2	64
2–3–0–1–2	73
2–3–1–0–2	73
3–0–1–2–3	73
3–0–2–1–3	64
3–1–0–2–3	73
3–1–2–0–3	64
3–2–0–1–3	73
3–2–1–0–3	73

4.2 Жадібний алгоритм розв’язання задачі комівояжера

Точні методи розв’язання задачі комівояжера працюють лише за малої розмірності графу. Метод повного перебору має факторіальну складність, динамічне програмування – експоненційну складність. Задача комівояжера є NP-складною, тому для вирішення практичних задач середньої та великої розмірності (рис. 4.3) застосовують евристики та наближені алгоритми. Одним із найпростіших наближених алгоритмів розв’язання задачі комівояжера є жадібний алгоритм. *Жадібний алгоритм* – це однопрохідний евристичний алгоритм, який на кожній ітерації вибирає локально найкраще продовження маршруту – додає до поточного фрагменту найближчу невідвідану вершину. Це швидкий алгоритм, який формує більш-менш раціональний розв’язок без гарантії оптимальності.

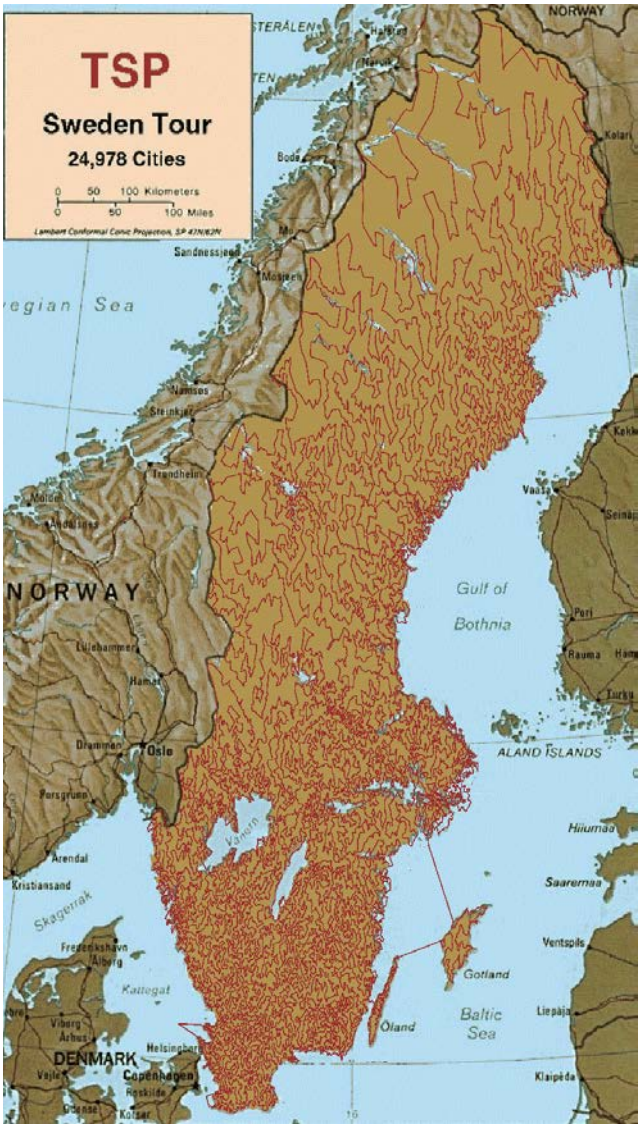


Рисунок 4.3 – Найкращий на сьогодні розв’язок задачі комівояжера на графі із 24 978 вершин

Жадібний алгоритм для задачі комівояжера складається з таких кроків:

1) стартуємо з довільної початкової вершини;
2) за рядком матриці відстаней відшукаємо найближчу невідвідану вершину і переходимо в неї;
3) повторюємо другий крок, поки не пройдемо за усіма вершинами;
4) повертаємося у початкову вершину, щоб сформувати цикл.

Жадібний алгоритм знаходить локальний розв'язок, хоча інколи може потрапити і на глобальний мінімум. Розглянемо роботу жадібного алгоритму для графа з рис. 4.2. Матрицю суміжності цього графу наведено на рис. 4.4. Як стартову виберемо вершину 0. Спочатку знайдемо найближчу вершину до вершини 0 – це вершина 1 з вагою переходу у 12. Далі шукаємо найближчу вершину до вершини 1 – це вершина 2 з вагою переходу у 15. Тепер йдемо у вершину 3 – це єдина вершина, яку ще не відвідали, і повертаємося у вершину 0. Загальна вага знайдено гамільтонового циклу 0–1–2–3–0 становить 12+15+29+17=73. Як бачимо з табл. 4.1 цей цикл не є найкращим. Отже, жадібний алгоритм не гарантує знаходження глобального мінімуму.

	0	1	2	3
0	0	12	14	17
1	12	0	15	18
2	14	15	0	29
3	17	18	29	0

Рисунок 4.4 – Матриця суміжності для графа з рис. 4.2

Якщо стартувати з інших вершин, тоді для аналізованого графу жадібний алгоритм знаходить такі розв'язки: 1–0–2–3–1 з вагою 72, 2–0–1–3–2 з вагою 73 та 3–0–1–2–3 з вагою 73. Усі вони різні, і усі вони гірші за глобальний мінімум (див. табл. 4.1). Проте на деяких графах жадібний алгоритм може знайти і глобальний розв'зок. Дослідження таких процесів пропонується для самостійного поглибленого вивчення.

4.3 Рандомізований жадібний пошук

В попередньому підрозділі розглянуто питання про наближене розв'язання задачі комівояжера за допомогою жадібного алгоритму. Для забезпечення диверсифікації наближених розв'язків використано мультистарти – прогін жадібного алгоритму з різних початкових вершин. Це доволі ефективний прийом, але кількість синтезованих за ним маршрутів невелика. Є кілька інших варіацій жадібного алгоритму, які дають змогу швидко згенерувати більшу кількість наближених розв'язків. Однією із таких варіацій є *рандомізований жадібний пошук*.

Рандомізація жадібного алгоритму полягає у тому, що продовженням маршруту вибирається не найближча невідвідана вершина, а одна із множини перспективних вершин. Множину перспективних вершин також називають *списком кандидатів* (candidate list). Якщо на деякій ітерації поточний список кандидатів виявляється порожнім, тоді маршрут продовжується за звичайним жадібним алгоритмом.

Список кандидатів доцільно сформувати для кожної вершини на етапі препроцесингу – до початку роботи алгоритму оптимізації. У список кандидатів вносять сусідні вершини. Кількість вершин у списку кандидатів обрізають директивно або на основі порогового обмеження на відстань. За директивним підходом довжину списку кандидатів обмежують якимось числом, наприклад, вносять у список кандидатів кожної вершини лише 15 найближчих сусідів. Підхід на основі порогового обмеження на відстань полягає у тому, щоб внести у список кандидатів лише ті вершини, відстань до яких не перевищує деяке граничне значення. Тоді довжина списку кандидатів для різних вершин може бути різною. Можливий також варіант, коли з матриці відстаней вибирають деякий відсоток найменших значень, і відповідні вершини вносять у списки кандидатів.

Під час формування маршруту комівояжера відбір вершини зі списку кандидатів може відбуватися за такими способами:

1) випадковий вибір, коли усі кандидати мають однакові шанси;

2) вибір за схемою колеса рулетки з нерівномірними секторами, коли кращі кандидати мають більші шанси. Зазвичай площа сектора рулетки, яка визначає шанси вершини, є обернено пропорційною до відстані;

3) вибір за турнірною селекцією, коли випадково відбираємо зі списку кандидатів N учасників турніру і поміж них знаходимо переможця за меншою відстаню;

4) елітарний відбір за ε-правилом, коли з ймовірністю ε продовження маршруту відбувається у найближчу сусідню вершину, а з ймовірністю $(1-\varepsilon)$ – у випадкову вершину з поточного списку кандидатів.

Розглянемо приклад роботи рандомізованого жадібного алгоритму на графі з матрицею відстаней з рис. 4.5. Сформуємо список кандидатів для кожної вершини на основі порогового обмеження – відстань до сусідньої вершини не має перевищувати 6. За таких умов списки кандидатів є таким:

для вершини A – $\{D, C, E\}$;

для вершини B – $\{E, C, A\}$;

для вершини C – $\{E, D, A\}$;

для вершини D – $\{B, C\}$;

для вершини E – $\{B, C\}$.

	A	*B*	*C*	*D*	*E*
A	0	6	2	1	4
B	5	0	3	6	2
C	5	8	0	4	3
D	7	4	5	0	7
E	9	1	6	8	0

Рисунок 4.5 – Матриця відстаней тестового графу

Для прикладу як стартову виберемо вершину *A* і запустимо рандомізований жадібний пошук. Один з можливих варіантів роботи цього алгоритму є таким. На першому кроці зі списку кандидатів вершини *A* випадковим чином вибираємо вершину, нехай це буде *C*. Додаємо її в маршрут, і на наступному кроці вже розглядаємо список кандидатів вершини *C*. Вершина *A* вже є у маршруті, тому список кандидатів звужується до $\{E, D\}$. З цієї пари випадковим чином вибираємо одну, нехай це буде вершина *E*. Повторимо кроки, поки не сформуємо гамільтонів цикл. Схематично цей процес зобразимо таким чином:

$$A-\{D,C,E\} \quad \to A-C;$$
$$A-C-\{E,D\} \quad \to A-C-E;$$
$$A-C-E-\{B\} \quad \to A-C-E-B;$$
$$A-C-E-B-\{\ \} \quad \to A-C-E-B-D;$$
$$length(A-C-E-B-D-A)=2+3+1+6+7=19.$$

Розглянемо інший прогон рандомізованого жадібного алгоритму:

$$A-\{D,C,E\} \quad \to A-D;$$
$$A-D-\{B,C\} \quad \to A-D-C;$$
$$A-D-C-\{E\} \quad \to A-D-C-E;$$
$$A-D-C-E-\{B\} \quad \to A-D-C-E-B;$$
$$length(A-D-C-E-B-A)=1+5+3+1+5=15.$$

Довжина першого маршруту становить 19, а другого – 15. Отже, нові запуски рандомізованого жадібного алгоритму можуть знаходити кращі розв’язки задачі комівояжера.

4.4 Методи локального покращення

Синтез наближених розв’язків складних задач дискретної оптимізації за деяким швидким евристичним алгоритом є доволі поширеною практикою. Два швидких евристичних алгоритми для задачі комівояжера розглянуто в попередніх підрозділах – мова йде про жадібний та рандомізований жадібний

пошуки. Отримані за подібними алгоритмами розв'язки інколи доцільно відправити на постпроцесінг – запустити для них процедури локального покращення маршрутів комівояжера. Суть цих процедур полягає в ітераційній перестановці деяких фрагментів маршрутів. Процедури локального покращення базуються або на перестановці двох вершин маршрута, або на модифікації кількох ребер. На перший погляд здається, що такі процедури є змістовно еквівалентними, бо перестановка вершин змінює і послідовність ребер в маршруті, а модифікація ребер змінює і порядок вершин в маршруті, але принципова різниця між цими процедурами існує. Вона обумовлена можливістю реверсного обходу великої ділянки маршруту коли застосовується процедура модифікації ребер. Реверсний обхід надає змогу усунути проблеми перехресної топології маршруту комівояжера (рис. 4.6), тоді як перестановка вершин краще справляється з проблемою невдалого сусідства (рис. 4.7). Розглянемо процедури локального покращення детальніше.

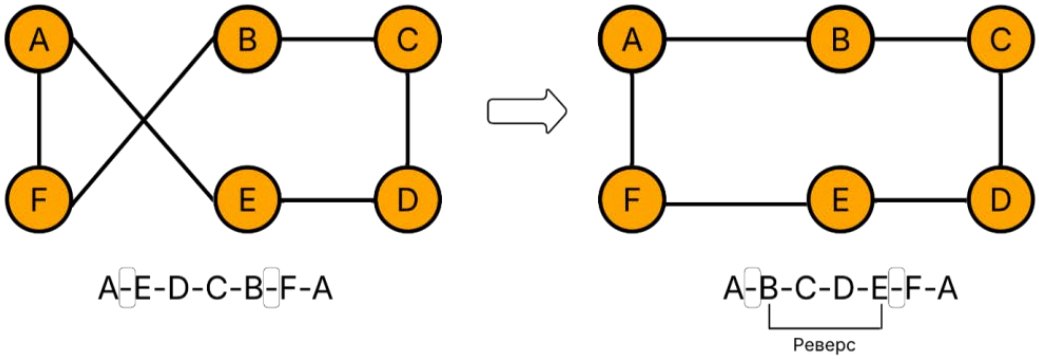


Рисунок 4.6 – Усування перехресної топології маршруту шляхом модифікації двох ребер

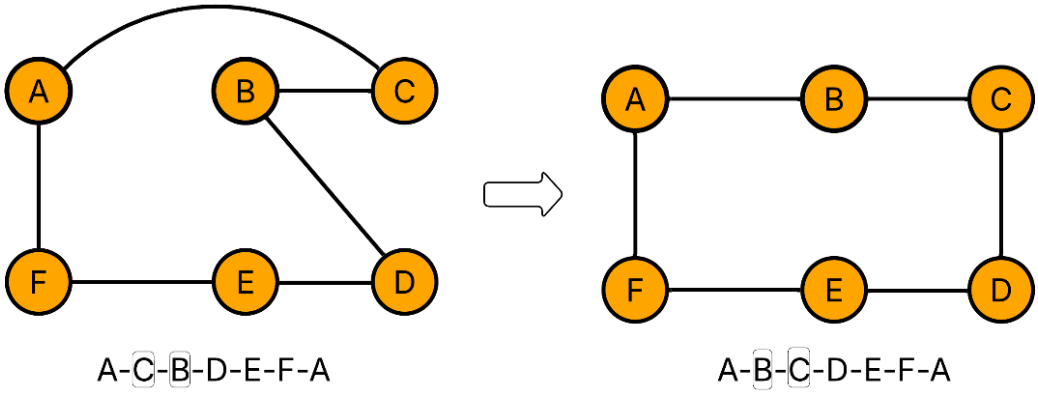


Рисунок 4.7 – Усуванання невдалого сусідства шляхом перестановки двох вершин маршрута

Перестановка вершин

Суть процедури – вибираємо дві вершини в маршруті і переставляємо їх місцями. Якщо перестановка є вдалою і модифікований маршрут став коротшим, тоді фіксуємо її та пробуємо вже для нового маршруту переставити інші вершини. Для приладу розглянемо граф, матриця відстаней якого задана на рис. 4.1. Для цього графу жадібний маршрут комівояжера $A \rightarrow D \rightarrow B \rightarrow E \rightarrow C \rightarrow A$ має довжину 1+4+2+6+5=18. Якщо в маршруті переставити місцями вершини B і E, тоді утворюється маршрут $A \rightarrow D \rightarrow E \rightarrow B \rightarrow C \rightarrow A$ з довжиною 1+7+1+3+5=17. Вона менша, отже перестановка вершин B і E покращила жадібний маршрут комівояжера.

Розглянемо обчислювальну ефективність алгоритму перестановки вершин. Якщо маршрут комівояжера короткий – до кількох десятків вершин, тоді довжину нового маршруту можна перераховувати за повною процедурою, коли за деякою функцією чи фрагментом коду в основній програмі розраховується довжина маршруту за його описом та матрицею відстаней. Якщо граф має n вершин, тоді потрібно з матриці відстаней взяти n+1 значення і виконати n операцій додавання. Якщо n – велике, тоді краще від довжини поточного маршруту відняти ваги дуг, вибраних для перестановки вершин, тобто ваги тих дуг, які видаляються з маршруту. Після цього додати ваги нових дуг. Для розглянутого прикладу, в якому переставлено сусідні вершини, потрібно від довжини поточного маршруту відняти ваги трьох старих дуг $D \rightarrow B$, $B \rightarrow E$ та $E \rightarrow C$ і додати ваги трьох нових дуг $D \rightarrow E$, $E \rightarrow B$ та $B \rightarrow C$. Назагал виконуємо 6 арифметичних операцій, тобто таку саму кількість, як і за повного перерахунку маршруту, бо n+1=6. Якщо задача комівояжера симетрична, тоді арифметичних операцій буде 4 – довжини дуг $B \rightarrow E$ та $E \rightarrow B$ однакові, тому немає сенсу їх віднімати та додавати. Але навіть і для симетричної задачі за n=5 процедура такого перерахунку може бути і невигідною через ускладнення структури програми. Натомість за великого розміру графа така процедура дозволяє суттєво скоротити тривалість роботи з урахуванням того, що доводиться реалізовувати багато варіантів перестановок. У випадку перестановки несусідніх вершин потрібно відняти ваги вже чотирьох старих дуг та додати ваги чотирьох нових дуг. Це збільшує обчислювальну складність на третину, але інколи дозволяє суттєво покращити маршрут комівояжера (рис. 4.8).

У більшості випадків перестановка вершин не покращує жадібний маршрут. Щоб зрозуміти буде покращення маршруту чи ні, можна не розраховувати його довжину, а лише розрахувати її приріст – від ваг нових дуг відняти ваги старих дуг. Тоді кількість арифметичних операцій скоротиться на 1. Це видається дрібницею, але вона може прискорити пошук на кілька відсотків, що за великих графів може мати сенс.

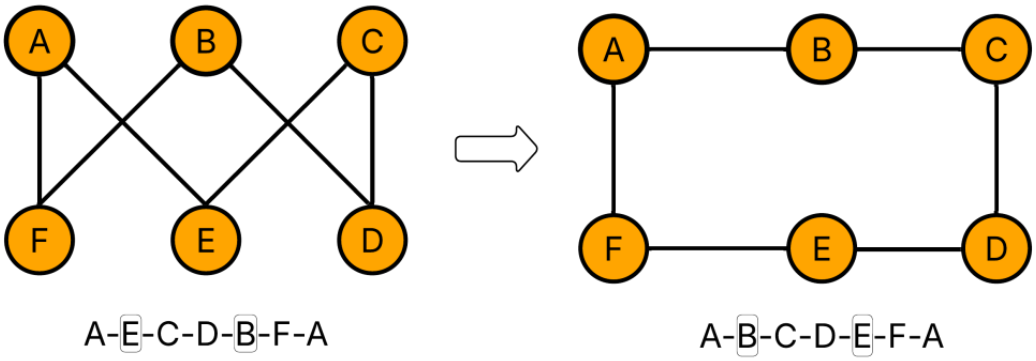


Рисунок 4.8 – Ефект від перестановки двох несусідніх вершин маршруту

Модифікація ребер

Для модифікації ребер розроблено кілька алгоритмів, зокрема: *2-opt*, який застосовується до двох ребер, *3-opt*, який застосовується до трьох ребер та алгоритм *Лін-Кернінгана* (Lin-Kernighan) для варіативної кількості ребер. Мова йде саме про ребра, тобто про симетричну задачу комівояжера. Розглянемо найпростіший та найпопулярнішиий із названих алгоритмів – 2-opt [11].

Алгоритм 2-opt полягає в ітеративних спробах поліпшення поточного маршруту шляхом заміни двох ребер маршруту на інші два ребра. Спроби продовжують, допоки новий маршрут не стане кращим. Алгоритм 2-opt полягає у виконанні таких кроків:

Крок 1. Починаємо з якогось початкового маршруту, наприклад, з маршруту, отриманого за жадібним.

Крок 2. Вибираємо з маршруту пару ребер, наприклад, *A–B* та *C–D*. Усі вершини в ребрах мають бути різними.

Крок 3. Перевіряємо, чи вигідно в маршруті переставити місцями вибрані ребра. Якщо довжина маршруту після перестановки зменшується, то видаляємо старі ребра *A–B* та *C–D* і на їх місця додаємо нові *A–C* та *B–D*.

Крок 4. Повторюємо кроки 2–3, поки не спрацює критерій зупинки.

Критеріями зупинки алгоритму 2-opt, як і алгоритму перестановки вершин, можуть бути такі: 1) завершення перевірки для усіх можливих пар ребер або вершин; 2) досягнення ліміту часу для покращення маршруту; 3) досягнення ліміту спроб; 4) відсутність покращення маршруту протягом значної кількості спроб.

Розглянемо приклад роботи алгоритму 2-opt. Для цього використаємо граф, матриця відстаней якого задана рис. 4.9. Нехай початковий маршрут є таким: *A–D–E–B–C–A*, довжина якого становить 17.

	A	*B*	*C*	*D*	*E*
A	0	6	2	1	4
B	6	0	3	6	2
C	2	3	0	4	3
D	1	6	4	0	7
E	4	2	3	7	0

Рисунок 4.9 – Матриця відстаней тестового графу

Випадковим чином виберемо пару ребер *A–D* та *E–B* та видалимо їх з маршруту:

A D–E B–C–A.

Додавши нові ребра *A–E* та *D–B,* отримаємо такий новий маршрут: *A–E–D–B–C–A,* довжина якого становить 4+7+7+3+3=22.

Маршрут вийшов довшим, тому така перестановка невигідна. Спробуємо переставити іншу пару ребер: *D–E* та *B–C*. Після їх видалення початковий маршрут стає таким:

A–D E–B C–A.

Додавши нові ребра *D–B* та *E–C*, отримуємо такий новий маршрут: *A–D–B–E–C–A* з довжиною 1+6+2+3+2=14. Маршрут скоротився, тому зафіксуємо цю перестановку і продовжимо ітерації алгоритму 2-opt.

Для розуміння усіх особливостей алгоритму 2-opt розглянемо випадок, коли між вибраними ребрами є більше двох вершин. Якщо з початкового маршруту видалити ребра *A–D* та *B–C,* тоді між ними буде ланцюг із 3-ма вершинами: *D–E–B.* Маршрут після видалення цих двох ребер є таким:

A D–E–B C–A.

Тепер необхідно вставити нові ребра *A–B* та *D–C*. Після переходу з вершини *A* у вершину *B* рух по ланцюгу *D–E–B* відбувається у зворотньому (реверсному) порядку: *B–E–D*. А після вершини *D* по новому ребру *D–C* відбувається перехід на вершину *C* і рух продовжується вже у прямому порядку. Отже, маршрут між ребрами, що видаляються, перевертається і новий маршрут має такий вигляд:

A–B–E–D–C–A.

Для розрахунку довжини нового маршруту достатньо до довжини поточного маршруту додати різницю ваг нових та старих ребер, тобто виконати лише 4 арифметичні операції. Наприклад, довжина синтезованого маршруту розраховується так: 17+(6+4-1-3)=23. Отже, пропонована заміна є недоцільною. Для того, щоб встановити чи заміна є доцільною, достатньо виконати лише 3 арифметичні операції за будь-якого *n*. Такі малі обчислювальні витрати на одну ітерацію обумовлюють ефективність

застосування алгоритму 2-opt для локального покращення маршрутів для симетричної задачі комівояжера.

4.5 Модифікації задачі комівояжера

Для врахування особливостей предметних областей розроблено кілька модифікацій задачі комівояжера. Найбільшим популярними поміж них є такі:

1) незамкнута задача, коли потрібно знайти не гамільтонів цикл, а гамільтоновий ланцюг, тобто не потрібно повертатися в початкову вершину;

2) узагальнена задача комівояжера (Generalized TSP), коли вершини графу розбито на кластери, і потрібно з кожного кластера обійти лише по одній вершині. Практичним прикладом такої задачі є маршрутизація покупця в гіпермаркеті, коли один і той самий товар може знаходитися на різних полицях. В цій задачі полиці відповідають вершинам графу, а товари – кластерам. Іншим прикладом є задача міжнародного мандрівника, якому потрібно відвідати кожну країну лише один раз, і неважливо через який аеропорт прибути в країну. В цій задачі аеропорти є вершинами графу, а країни – кластерами. Аналогічно можна розглядати задачу маршрутизації диверсійної групи, коли в кожному кластері потрібно вивести із ладу лише один елемент, наприклад, фрагмент лінії постачання ресурсів чи комунікаційний канал. За фіксованого порядку обходу кластерів узагальнена задача комівояжера перетворюється на специфічну задачу про прокладення магістралі, яка має важливе прикладне значення;

3) неповна задача комівояжера (Orienteering Problem), коли потрібно відвідати не усі вершини графу, а лише деяку їх частку. Кількість вершин в обході відповідає ємності комівояжера або визначається за досягнутим ефектом. Для цих задач окрім матриці відстаней потрібно знати і прибуток від відвідування кожної вершини. Задача комівояжера ставиться як задача максимізація прибутку за обмеження на довжину маршруту або як мінімізація маршруту, який дозволяє отримати бажаний розмір прибутку. Такі задачі комівояжера можуть інтерпретуватися як нелінфйні задачі про рюкзак (Knapsack Problem);

4) задачі планування пов’язанних обходів комівояжера, наприклад, коли потрібно сформувати маршрути інспекторів протягом кількох діб. У цьому разі штат доступних інспекторів може змінюватися від доби до доби, так само як і перелік об’єктів та їх важливість;

5) кур’єрська задача, коли потрібно з одних вершин забрати посилку і передати її на замовлену вершину, причому обсяг рюкзака у кур’єра обмежений. Якщо обсяг рюкзака дорівнює одній посилці, тоді кур’єр працює в режимі взяв посилку – передав посилку;

6) задачі з часовими вікнами (time windows), коли для кожної вершини задано часовий інтервал, протягом якого комівояжер має її відвідати;

7) задачі з багатьма одночасно працюючими комівояжерами – задачі маршрутизації вантажівок (Vechile Routing Problem) та задачі маршрутизації шкільних автобусів (School Bus Scheduling).

На практиці для кожної реальної задачі враховуються багато специфічних чинників, як-от тривалість перебування школярів в автобусі, обсяг чи площа кузова вантажівки, товарна сумісність, часові вікна, обмеження на безперевну роботу комівояжера, комфортність та естетична привабливість маршруту тощо. Хоча специфіка предметних областей зазначених задач різна, але підходи до їх вирішення спираються на базові алгоритми розв'язання класичної задачі комівояжера.

4.6 Завдання для самостійного дослідження

Базовий рівень

1. Дослідити простий жадібний алгоритм вирішення задачі комівояжера. Завдання передбачає програмну реалізацію жадібного алгоритму, вирішення однієї із задач комівояжера та аналіз результатів. Задача для дослідження вибирається відповідно до варіанта з табл. 4.2. Варіанти 1–69 стосуються симетричної задачі комівояжера, а варіанти 70–78 – асиметричної. Матриці відстаней та поточні світові рекорди з розв'язання цих задач наведено на сайті http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html. Для деяких симетричних задач комівояжера задана не уся матриця відстаней, а її унікальний трикутник – усі елементи вище або нижче головної діагоналі. Для деяких варіантів задано не матриці відстаней, а координати вершин на площині (x, y) або географічні координати вершин. Перед пошуком маршрутів комівояжера потрібно перетворити координати вершин на матрицю відстаней відповідно до метрики, яка також зазначена у описі задачі. Граф вважається повнозв'язним. В таблиці варіантів вказано назву набору даних зі згаданого сайту. Наприклад, для першого варіанта назва – a280. На сайті можна знайти архів цього набору даних у форматі .tsp: a280.tsp.gz (рис. 4.10). Там також є і файл найкращого маршруту: a280.opt.tour.gz. Набори даних для варіантів знаходяться на сайті в розділах Symmetric traveling salesman problem та Asymmetric traveling salesman problem.

Одним із пропонованих досліджень є експериментальне встановлення залежності довжини маршруту комівояжера, який знайдено жадібним алгоритмом, від кількості мультистартів. Необхідно проаналізувати динаміку

оптимізації та порівняти знайдений найкращий розв’язок з поточним світовим рекордом.

Таблиця 4.2 – Варіанти завдань

Варіант	Задача	Варіант	Задача
1	a280	16	linhp318
2	ali535	17	p654
3	att48	18	pa561
4	bayg29	19	pcb442
5	bays29	20	pr76
6	berlin52	21	pr107
7	bier127	22	pr124
8	brazil58	23	pr136
9	brg180	24	pr144
10	ch130	25	pr152
11	ch150	26	pr226
12	d198	27	pr264
13	d493	28	pr299
14	d657	29	pr439
15	dantzig42	30	rat99
31	eil101	55	rat195
32	eil76	56	rat575
33	fl417	57	rat783
34	fri26	58	rd100
35	gr24	59	rd400
36	gr48	60	si175
37	gr96	61	si535
38	gr120	62	st70
39	gr137	63	swiss42
40	gr202	64	ts225
41	gr229	65	tsp225
42	gr431	66	u159
43	gr666	67	u574
44	hk48	68	u724
45	kroA100	69	ulysses22
46	kroB100	70	ft70
47	kroC100	71	ftv33
48	kroD100	72	ftv38
49	kroE100	73	ftv47
50	kroA150	74	ftv64
51	kroB150	75	kro124
52	kroB200	76	rbg323
53	lin105	77	rbg403
54	lin318	78	ry48p

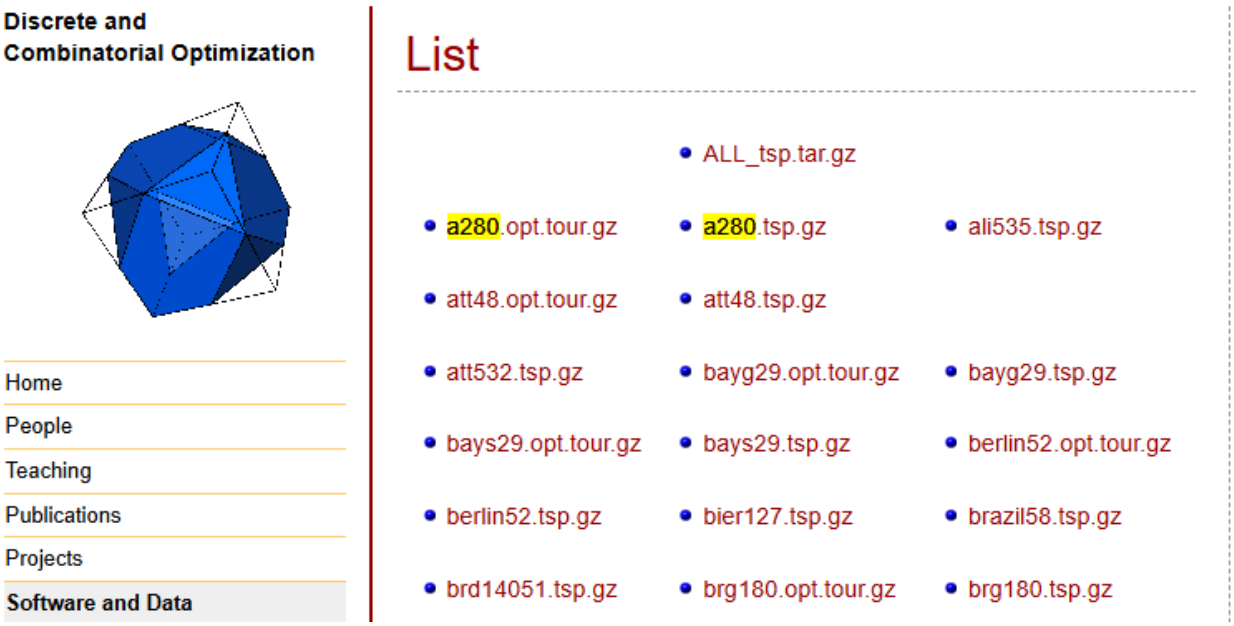


Рисунок 4.10 – Фрагмент набору даних задач комівояжера

Під мультистартом розуміється запуск жадібного алгоритму з нової вершини графу. Якщо граф має 20 вершин, то можливі 20 мультистартів, і, відповідно, 20 жадібних маршрутів комівояжера. Деякі із знайдених маршрутів можуть бути ідентичними. Якщо задача комівояжера асиметрична, тоді можна формувати маршрути не з початкової вершини, а з кінцевої, тобто зробити жадібним алгоритмом зворотний прохід. Тоді для графу із 20 вершин можна синтезувати 20+20=40 жадібних маршрутів. Зворотний прохід можна реалізувати тим самим жадібним алгоритмом прямого проходу, потрібно лише транспонувати матрицю відстаней.

Очікується, що результати виконання дослідження буде візуалізовано у формі рис. 4.11. Рисунок побудовано для випадку, коли для графу із 20 вершин довжини маршрутів, знайдених жадібним алгоритмом із різних початкових точок, дорівнють: 40, 35, 33, 33, 33, 33, 32, 32, 32, 32, 32, 32, 32, 31, 30, 30, 30, 29, 28 та 25. Найкращий знайдений розв’язок має довжину 25. Бузкова лінія на рис. 4.11 відповідає найкращому розв’язку. Інші криві показують середню динаміку та динаміку у найгіршому випадку.

Розглянемо побудову кривої динаміки за найкращого випадку. Першу вершину для побудови маршруту вибираємо випадково, тому є шанс, що відразу попадемо на вдалу вершину і знайдемо найкоротший маршрут. У випадку двох мультистартів ці шанси подвоюються. У випадку 20 мультистартів для досліджуваної задачі завжди знайдемо найкоротший жадібний маршрут, маршрут довжиною 31. Найкращий розв’язок можна знайти за будь-якої кількості мультистартів, але з різними шансами на успіх, тому бузкова лінія на рис. 4.11 є горизонтальною.

Найгірший розв’язок має довжину 40. За одного мультистарту є шанси потрапити саме на нього. Тому, чорна суцільна лінія з квадратами починається з точки (1, 40). Якщо робимо 2 мультистарти, тоді у найгіршому випадку отримаємо розв’язки з довжинами 40 та 35. Серед них кращим буде розв’язок з довжиною 35, відповідно, чорна суцільна лінія з квадратами надалі проходитиме через точку (2, 35). Далі з трійки найгірших варіантів вибираємо найкращий, потім з четвірки тощо. Фактично, для побудови найгіршої динаміки достатньо відсортувати знайдені за мультистартами розв’язки від більшої до меншої довжини маршруту.

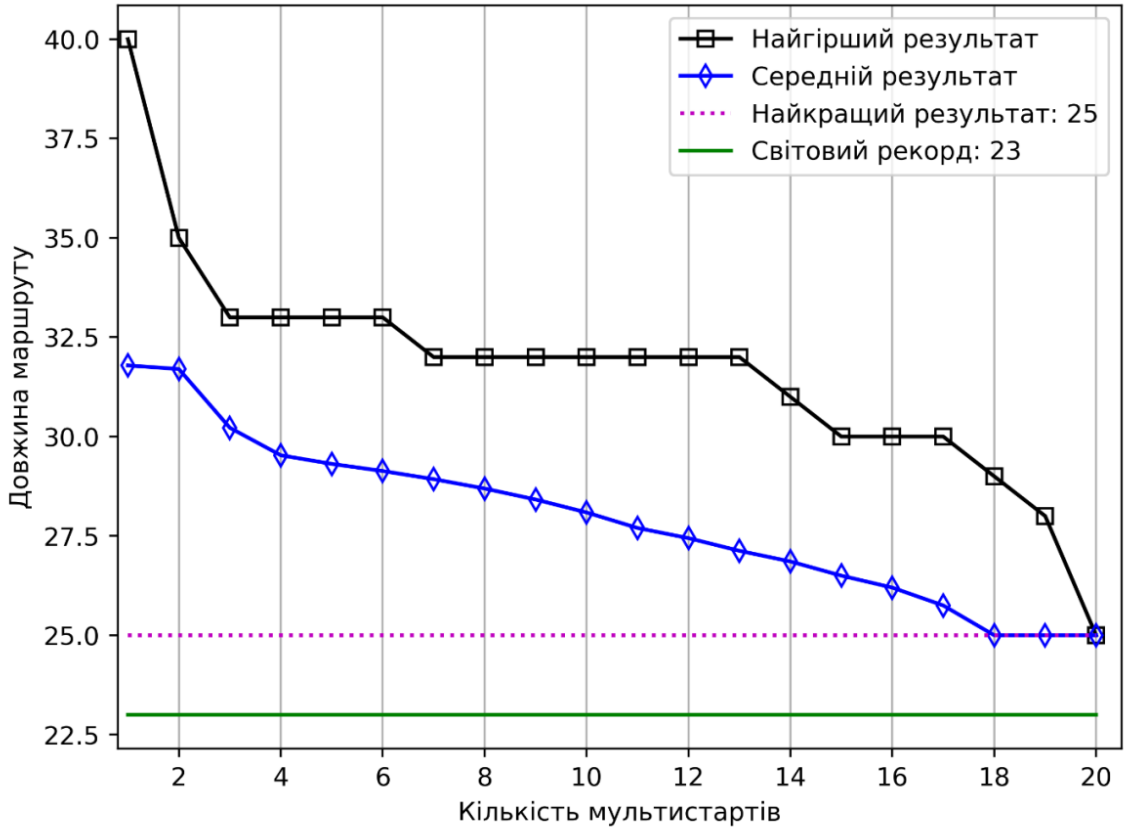


Рисунок 4.11 – Залежність якості жадібного розв’язку від числа мультистартів

Синя лінія з ромбами на рис. 4.11 – це усереднена динаміка оптимізації. Вона показує середнє значення довжин знайдених маршрутів комівояжера за відповідної кількості мультистартів. Якщо робимо 2 мультистарти, то отримуємо пару маршрутів, які починаються, наприклад, з вершини 1 та з вершини 2, з вершини 1 та з вершини 3, з вершини 1 та з вершини 4, ... , з вершини 19 та з вершини 20. Якщо робимо 3 мультистарти, то матимемо трійку маршрутів, які починаються з вершин 1, 2 та 3, або з вершин 1, 2 та 5, або, …, з вершин 18, 19 та 20. З кожної комбінації – з кожної пари, трійки,

четвірки, п’ятірки тощо, потрібно вибрати кращий маршрут, маршрут з мінімальною довжиною, а потім усереднити вибрані довжини за усіма парами, за усіма трійками, за усіма четвірками, за усіма п’ятірками тощо. Нехай маршрути першої трійки мають довжину 41, 40 та 39. Тоді з цієї трійки вибираємо маршрут довжиною **39**. Якщо маршрути другої трійки мають довжину 41, 37 та 35, тоді з цієї трійки вибираємо маршрут довжиною **35.** Якщо маршрути третьої трійки мають довжину 37, 32 та 35, тоді вибираємо маршрут з довжиною **32**. Якщо є лише 3 названі трійки, тоді відповідне значення на кривій усередненої динаміки становитеме (39+35+32)/3=35.3. Для графа із 20 вершин кількість трійок жадібних маршрутів становить 1140. Після усереднення довжин кращих маршрутів усіх трійок отримаємо ординату 30.2 кривої середньої динаміки, яка відповідає абсцисі 3 (див. рис. 4.11).

Для розрахунку усередненої динаміки потрібно перебрати усі підмножини з множини жадібних маршрутів. Вище була мова про трійки із множини у 20 елементів. Кількість таких підмножин розраховується за формулою: $\frac{20!}{3!\cdot(20-3)!}=\frac{20\cdot 19\cdot 18}{3\cdot 2}=1140$. П’ятірок вже буде $\frac{20!}{5!\cdot(20-5)!}=\frac{20\cdot 19\cdot 18\cdot 17\cdot 16}{120}=15504$, а десяток – $\frac{20!}{10!\cdot(20-10)!}=184756$. За великої потужності початкової множини та середньої потужності аналізованої підмножини, варіантів буде дуже багато, тому їх перебрати неможливо. В цьому випадку, щоб оцінити усереднену динаміку варто скористатися методом Монте-Карло. Ідея полягає в тому, щоб випадковим чином згенерувати деяку кількість пар, трійок, четвірок, п’ятірок тощо, і за кожним набором знайти середнє значення. Якщо таких пар буде доволі багато, тоді розраховані значення будуть близькими до точних. Ознакою близькості буде гладкий вигляд кривої усередненої динаміки – вона має бути монотонно спадною. Пропонується, вивести такі криві динаміки за різного розміру монте-карловських вибірок, наприклад, 100, 500, 1000.

Якщо в умові завдання задано не матрицю відстаней, а координати вершин графу, то необхідно вивести та проаналізувати найкращий знайдений маршрут комівояжера. Приклад такого маршруту наведено на рис. 4.12.

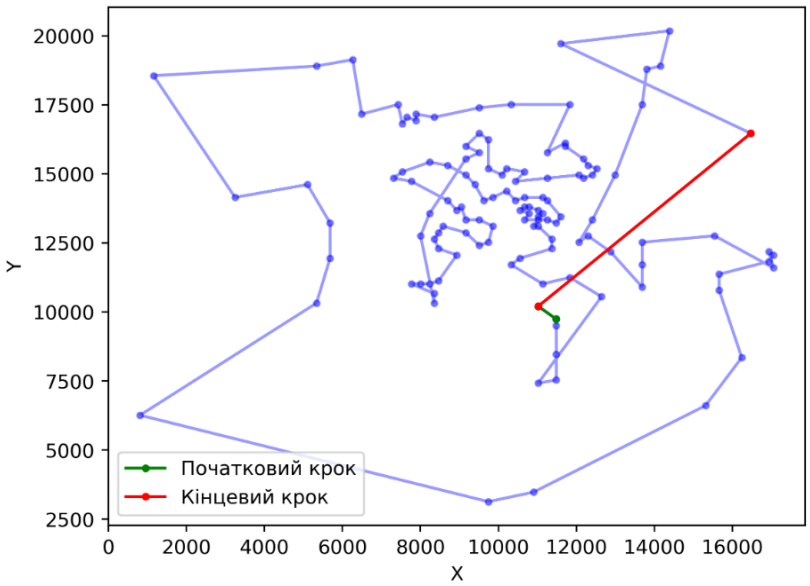


Рисунок 4.12 – Найкращий жадібний маршрут комівояжера

2. Випадковим чином згенерувати $20 \cdot n$ гамільтонових циклів та розрахувати їх довжину. Таку генерацію зручно виконувати випадковою перестановкою вершин в маршруті. За аналогією з рис. 4.11 за отриманими даними побудувати статистичні залежності якості розв'язку задачі комівояжера від кількості згенерованих варіантів. Порівнюючи усереднені динаміки за мультистартами та за випадковими гамільтоновими циклами, побудувати графік, аналогічний до рис. 4.13. Засікти час пошуку одного маршруту комівояжера за жадібним алгоритмом та за випадковим пошуком, і порівняти усереднені динаміки в часовому просторі.

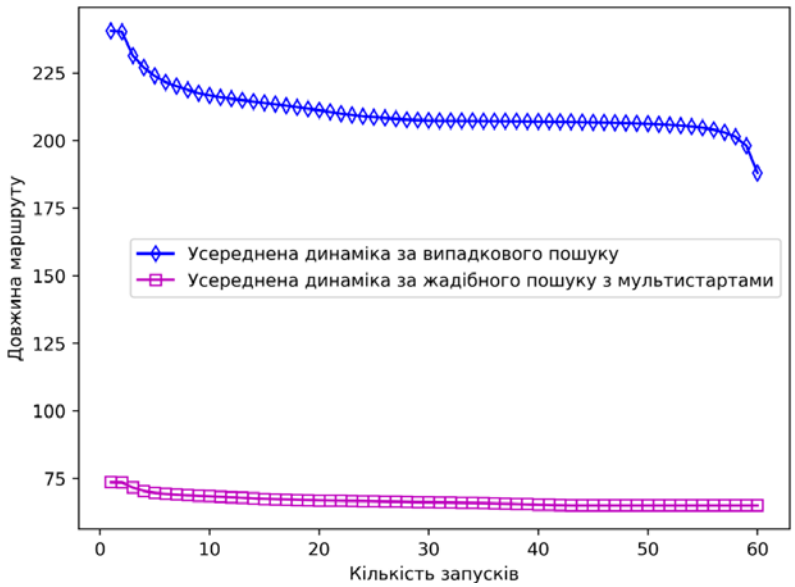


Рисунок 4.13 – Порівняння усереднених динамік якості розв'язку

3. Дослідити рандомізований жадібний алгоритм вирішення задачі комівояжера. Задача для дослідження вибирається відповідно до варіанта з табл. 4.2. Експериментально оцінити як залежить якість розв'язку від кількості синтезованих рандомізованим жадібним алгоритмом маршрутів комівояжера за різної довжини списку кандидатів. Можливі довжини списку кандидатів однієї вершини – 2, 3, 5 та 10. Кількість прогонів рандомізованого жадібного алгоритму – 5000. Під час експериментів для кожного прогону початкову вершину для синтезу маршруту вибирати випадковим чином. Порівняти найкращі розв'язки, отримані за кожним варіантом рандомізованого жадібного алгоритму з класичним жадібним алгоритмом з мультистартами та зі світовим рекордом. Результати експериментів зобразити графіками, аналогічними до рис. 4.14. Порівняти усереднені динаміки оптимізації за класичним жадібним алгоритмом та рандомізованим жадібним алгоритмом.

Для кожного варіанта рандомізованого жадібного алгоритму відобразити довжину найкращих маршрутів. Наприклад, для даних з рис. 4.14 найкращі довжини маршрутів мають такий вигляд: за списку кандидатів із 2 вершин – 127337; за списку кандидатів із 5 вершин – 137267; за списку кандидатів із 15 вершин – 154246; за класичного жадібного алгоритму – 133970. Якщо в умовах задачі комівояжера наведено координати вершин графа, то вивести найкращий маршрут комівояжера, який занайдено за класичними та за рандомізованим жадібними алгоритмами (рис. 4.12 та 4.15).

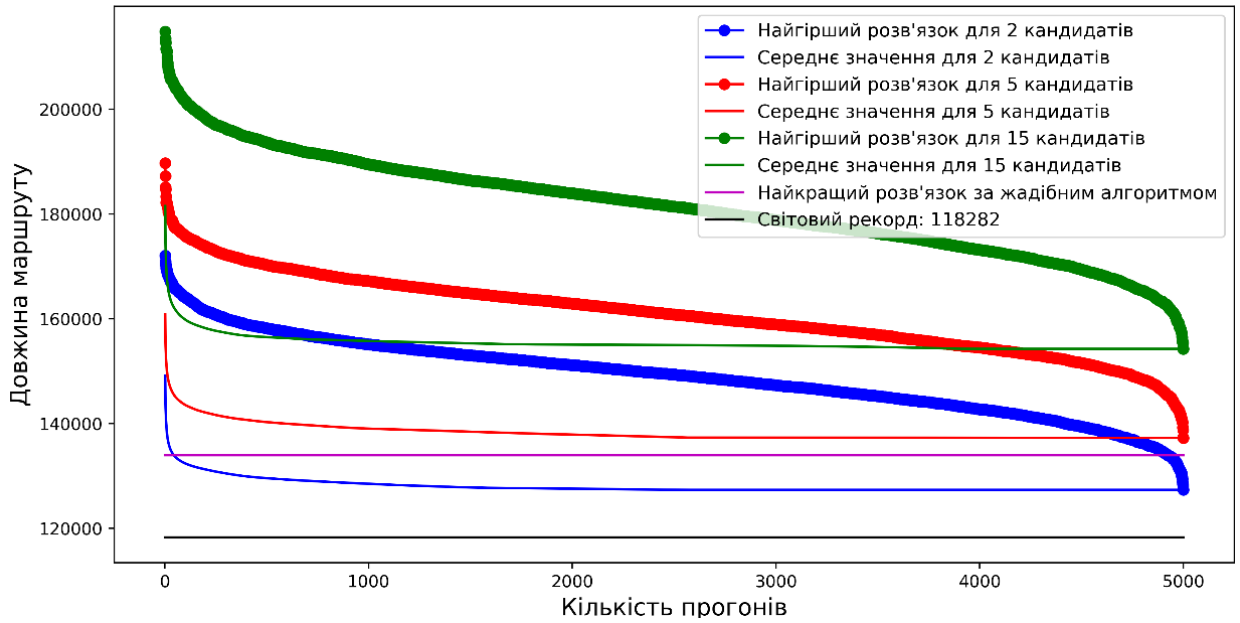


Рисунок 4.14 – Динаміка довжини маршруту комівояжера

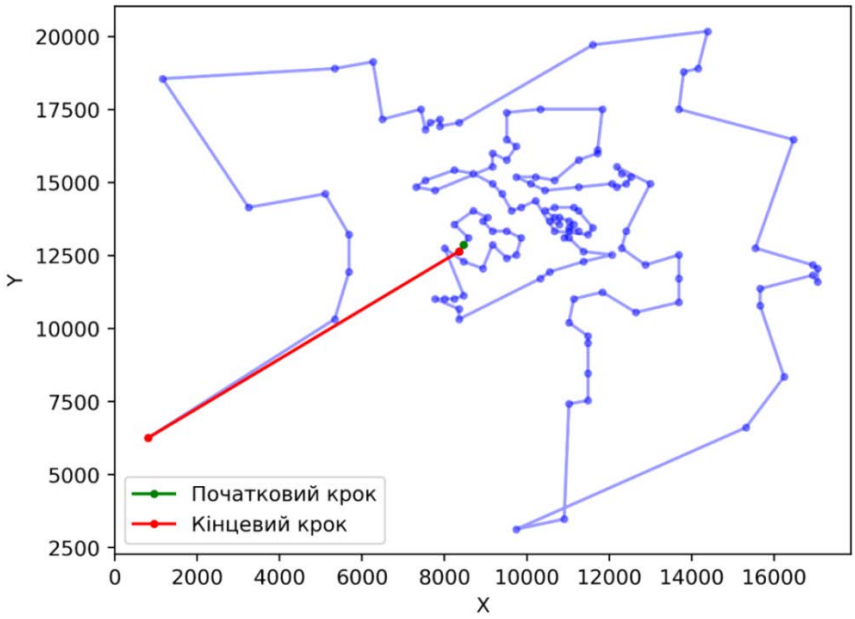


Рисунок 4.15 – Найкращий маршрут, знайдений рандомізованим жадібним алгоритмом з довжиною списку кандидатів у дві вершини

Поглиблений рівень

1. Експериментально дослідити, як впливає на усереднену динаміку якості розв'язку задачі комівояжера тип селекції кандидатів: рівномірна випадкова, колесо рулетки, турнірний вибір та елітарний відбір за ε-правилом. Кількість прогонів рандомізованого жадібного алгоритму для кожного типу селекції – 5000.

2. Експериментально дослідити, як впливає на усереднену динаміку якості розв'язку задачі комівояжера адаптація пошуку. Під адаптацією пошуку розуміється вибір стартової вершини з урахуванням поточної статистики успішності її використання. Найпростіший варіант адаптації – це повторний запуск з вибраної вершини, якщо поточний рандомізований жадібний пошук з неї був успішним. Критерієм успішності можна вибрати потрапляння поточного розв'язку у 2% найкоротших маршрутів, знайдених від початку експериментів. Для перших 100 прогонів статистики ще замало, тому пошук з тієї самої вершини повторюється лише, якщо знайдений розв'язок виявився найкращим. Необхідно дослідити, як впливає на усереднену динаміку якості розв'язку збільшення планки успішності з 2% до 3%, 5% та 10%.

3. Експериментально дослідити, як впливає на усереднену динаміку якості розв'язку задачі комівояжера попередній відбір перспективних початкових вершин. Попередній відбір полягає в тому, що після мультистартів жадібного алгоритму виявляються вершини, з яких отримано найкоротші маршрути комівояжера за жадібним алгоритмом. Саме на цих

початкових вершинах і проганяється рандомізований жадібний алгоритм. Необхідно дослідити, як впливає на усереднену динаміку якості розв'язку кількість відібраних вершин. Експерименти провести, коли відібрано 2%, 3%, 5% та 10% від усіх вершин графа. За результатами експериментів зробити висновок про обчислювальну ефективність такого гібридного вирішення задачі комівояжера.

4. Дослідити методи локального покращення розв'язків симетричної задачі комівояжера. Задача для дослідження вибирається відповідно до варіанта з табл. 4.2. Експериментально оцінити, як залежить якість розв'язку від кількості переставлених пар ребер. Провести два прогони алгоритму 2-opt: один з найкращого жадібного маршруту та один з найгіршого жадібного маршруту. Під час виконання алгоритму 2-opt необхідно зберігати довжини усіх маршрутів, навіть якщо перестановка не покращує початковий маршрут. Також потрібно засікти тривалість перебору варіантів. Порівняти найкращі розв'язки, які отримано за алгоритмом 2-opt та за класичним жадібним алгоритмом з мультистартом, а також зі світовим рекордом. Результати експериментів зобразити графіками, аналогічними до рис. 4.16. Зробити висновки. Для кожного прогону у звіті необхідно відобразити довжину найкращих маршрутів, а також найкращий маршрут за жадібним алгоритмом. Наприклад, для даних з рис. 4.16 найкращі довжини маршрутів мають такий вигляд:

- найкращий результат 2-opt, починаючи з найкращого жадібного маршруту – 129804;
- найкращий результат 2-opt, починаючи з найгіршого жадібного маршруту – 139923;
- найкращий результат за жадібним алгоритмом – 133970.

На рис. 4.16 подано динаміку довжини маршруту залежно від кількості перестановок за алгоритмом 2-opt. Алгоритм застосовано до двох маршрутів: найкращого жадібного та найгіршого жадібного. Вісь абсцис відповідає кількості перебраних пар ребер. Бачимо, що у випадку застосування алгоритму 2-opt до найкращого жадібного маршруту за порівняно невеликої кількості перестановок виходимо на максимально можливе покращення маршруту. Якщо запустити 2-opt для найгіршого жадібного маршруту, тоді кількість перестановок зростає і в кінцевому результаті знаходимо довший маршрут порівняно з розв'язками за мультистартами жадібного алгоритму.

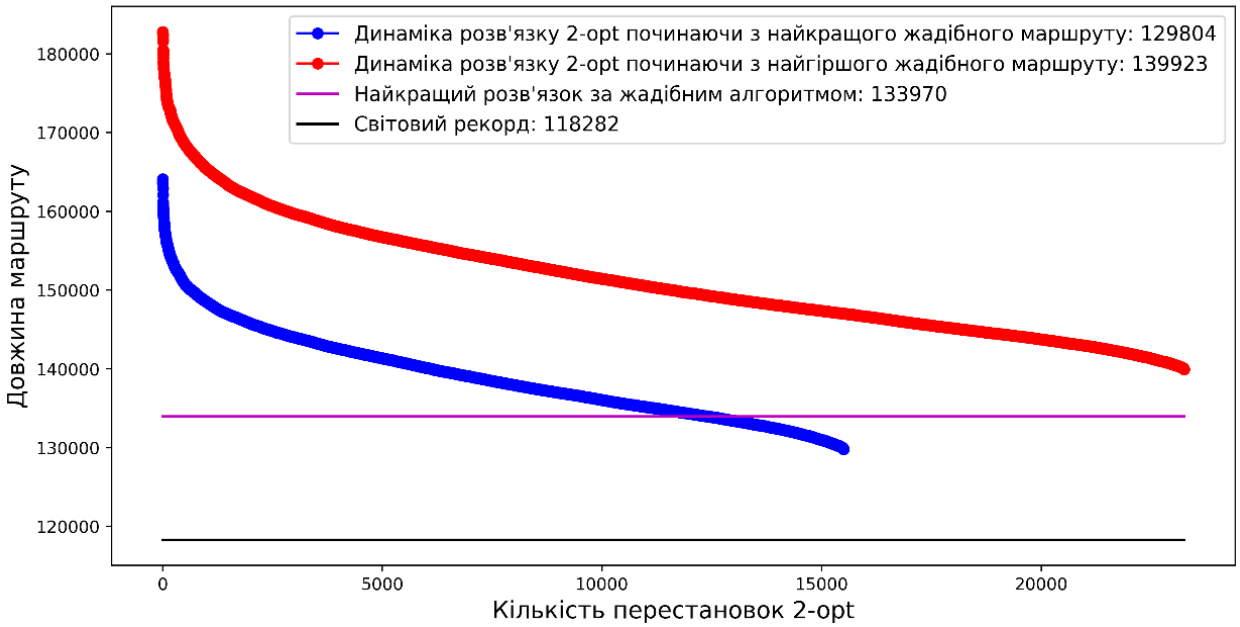


Рисунок 4.16 – Динаміка довжини маршруту залежно від кількості перестановок за алгоритмом 2-opt

5. Експериментально оцінити, як залежить якість розв’язку від кількості переставлених пар вершин. Провести 2 прогони алгоритму локального покращення: один з найкращого жадібного маршруту та один з найгіршого жадібного маршруту. Під час виконання алгоритму необхідно зберігати довжини усіх маршрутів, навіть якщо перестановка не покращує початковий маршрут. Також потрібно засікти тривалість перебору варіантів. Порівняти найкращі розв’язки, які отримано за перестановкою вершин та за класичним жадібним алгоритмом з мультистартом, а також зі світовим рекордом. Результати експериментів зобразити графіками, аналогічним до рис. 4.16. Зробити висновки. Для кожного прогону у звіті необхідно відобразити довжину найкращих маршрутів, а також найкращий маршрут за жадібним алгоритмом.

6. Порівняти ефективність покращення маршрутів на основі модифікації ребер за алгоритмом 2-opt та на основі перестановки вершин. Порівнювати потрібно з урахуванням тривалості перебору за кожним алгоритмом. Якщо в завданні зазначено координати вершин графу, тоді необхідно вивести початкові маршрут комівояжера та найкращі результати модифікації за алгоритмами локального пошуку. Приклад такого маршруту наведено на рис. 4.17.

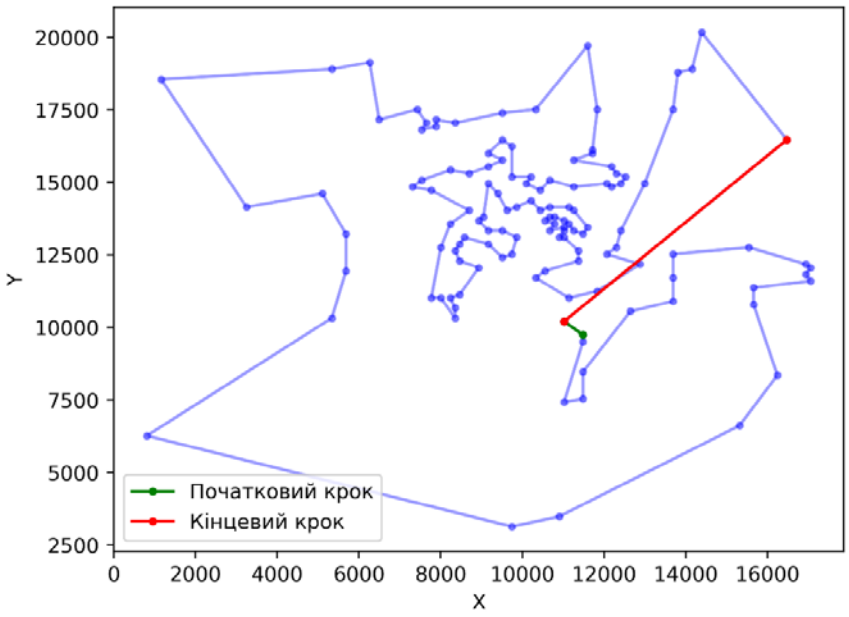


Рисунок 4.17 – Найкращий маршрут, знайдений за методом 2-opt

7. Експериментально оцінити ефект від застосування алгоритму 2-opt для різних початкових маршрутів. Множиною початкових маршрутів вибрати всі можливі жадібні маршрути, знайдені з кожної вершини графу. Наприклад, якщо задача комівояжера полягає в обході 100 вершин, то початкових маршрутів буде 100. Результати експериментів подати за аналогією з рис. 4.18 та 4.19. Зробити висновок, чи доцільно покращувати лише найкоротші жадібні маршрути.

Рисунок 4.18 – Ефект застосування 2-opt за різних початкових маршрутів

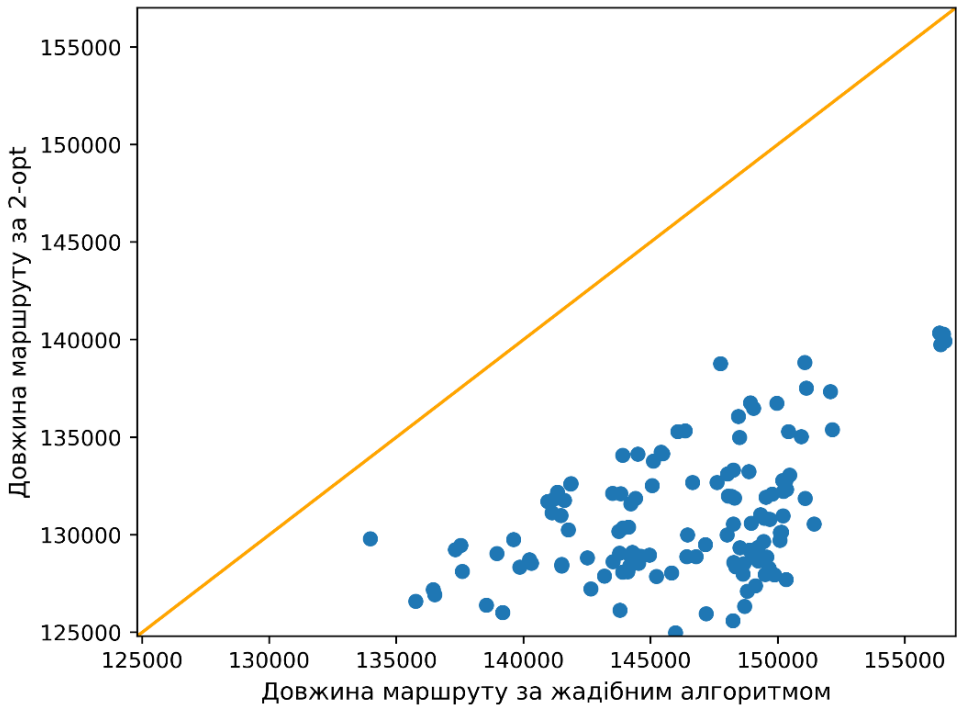


Рисунок 4.19 – Порівняння довжин жадібних маршрутів до та після 2-opt

8. Експериментально оцінити ефект від перестановки вершин для різних початкових маршрутів. Множиною початкових маршрутів вибрати всі можливі жадібні маршрути, знайдені з кожної вершини графу. Результати експериментів подати у вигляді точкової діаграми. Зробити висновок, чи доцільно покращувати лише найкоротші жадібні маршрути. Якщо в завданні наведено не матрицю відстаней, а координати вершин графа, тоді необхідно вивести найкращі маршрути комівояжера, які знайдено під час цього експерименту.

4.7 Поради та рекомендації

Для парсингу файлів у форматі `.tsp` можна використати Python бібліотеку `tsplib95`. Для іншої мови програмування необхідно шукати альтернативи або ж парсити вручну.

```
import tsplib95

with open('./a280.tsp') as file:
    coordinates = tsplib95.read(file)
    node_coordinates = coordinates.node_coords
    matrix = get_adjacency_matrix(node_coordinates)
    # розв'язок задачі жадібним алгоритмом
```

Якщо вершини графа задано географічними координатами, то для розрахунку матриці відстаней потрібно використовувати формулу гаверсинуса:

```
def euclidian_distance(x1, y1, x2, y2):
    return math.sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2))

def geo_distance(lat1, lon1, lat2, lon2):
    lat1 = math.radians(lat1)
    lon1 = math.radians(lon1)
    lat2 = math.radians(lat2)
    lon2 = math.radians(lon2)
    R = 6371.0
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = math.sin(dlat / 2)**2 + math.cos(lat1) * math.cos(lat2) *
math.sin(dlon / 2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    return R * c
```

Для виконання другого завдання доцільно використовувати функції `random.shuffle()` для випадкової перестановки вершин в гамільтоновому циклі та `time.time()` для фіксації тривалості роботи модулів програми.

4.8 Питання для самоконтролю та професійного розвитку

1. В чому суть задачі комівояжера?
2. Наведіть реальні приклади симетричних та асиметричних задач комівояжера.
3. Що таке гамільтонів цикл і як він пов’язаний із задачею комівояжера?
4. Скільки може бути гамільтонових циклів у повному графі з n вершинами для симетричного та асиметричного випадків?
5. Які існують підходи до розв’язання задачі комівояжера?
6. Яка складність повного перебору варіантів маршруту комівояжера?
7. Чи є задача комівояжера NP-складною? Чому?
8. В чому полягає ідея жадібного алгоритму розв’язання задачі комівояжера?
9. Як працює жадібний алгоритм?
10. Яка складність жадібного алгоритму вирішення задачі комівояжера?
11. Чи гарантує жадібний алгоритм знаходження оптимального розв’язку?
12. Як початкова вершина впливає на результат роботи жадібного алгоритму?

13. Чому зворотний прохід жадібним алгоритмом має сенс лише для асиметричних задач комівояжера?

14. Як інтерпретувати горизонтальну ділянку на графіку залежності довжини найкращого розв’язку задачі комівояжера від кількості мультистартів жадібного алгоритму? Чи може бути горизонтальною ділянка на графіку усередненої динамики?

15. Чи можливо вирішити задачу комівояжера шляхом багатократного застосування алгоритму Дейкстри?

16. Як працює рандомізований жадібний алгоритм і чим він відрізняється від класичного жадібного алгоритму?

17. Чому багаторазове виконання рандомізованого жадібного алгоритму може покращити результат?

18. Що таке детерміновані та недетерміновані алгоритми? До якої категорії належить рандомізований жадібний алгоритм?

19. Як залежить середня довжина маршруту від кількості прогонів алгоритму?

20. Навіщо у рандомізованому жадібному алгоритмі використовувати списки кандидатів?

21. Яким чином можна вибирати наступну вершину зі списку кандидатів у рандомізованому жадібному алгоритмі?

22. Яким чином можна формувати список кандидатів для рандомізованого жадібного алгоритму?

23. Які переваги та недоліки довгого та короткого списків кандидатів?

24. Який алгоритм швидше – класичний жадібний алгоритм чи рандомізований жадібний алгоритм?

25. Які можливі варіанти адаптації рандомізованого жадібного пошуку? Як адаптація пошуку впливає на ефективність алгоритму?

26. Що таке алгоритм 2-opt і для чого він використовується?

27. Чим відрізняєтья 2-opt від перестановки двох вершин маршруту?

28. Як вибирають ребра в алгоритмі 2-opt?

29. У яких випадках після модифікації ребер необхідно виконувати реверс підмаршруту?

30. Які головні критерії зупинки роботи алгоритму локального покращення?

31. Які алгоритми локального покращення доцільно застосовувати за асиметричної задачі комівояжера?

32. Як збільшення кількості перестановок за алгоритмом 2-opt впливає на довжину найкращого маршруту?

33. Чи завжди більша кількість ітерацій 2-opt гарантує кращий розв’язок?

34. Як можна виявити момент, коли подальші перестановки ребер не покращать розв’язок?

35. Якщо час на пошук сильно обмежений, то яка стратегія краща: (а) запустити спочатку усі мультистарти жадібного алгоритму, вибрати з них кращий і запустити 2-opt чи (б) для кожного жадібного розв’язку відразу запускати 2-opt?

36. Яка часова складність алгоритму 2-opt у найгіршому випадку?

37. Як вибір початкового маршруту впливає на ефективність 2-opt?

38. Як можна експериментально оцінити вплив кількості перестановок в 2-opt на якість розв’язку?

39. Чи є сенс після усіх перестановок вершин прогнати найкращий маршрут через алгоритм 2-opt?

40. Чи є сенс після усіх модифікацій ребер за алгоритмом 2-opt переставляти вершини в найкращому маршруті?

41. Які складнощі застосування алгоритму 2-opt за асиметричної задачі комівояжера?

42. Чи можна розпаралелити пошук найкращого маршруту за алгоритмом 2-opt?

43. Чи зміняться результати жадібного пошуку, якщо для розрахунку матриці відстаней замість евклідової метрики помилково застосувати міську метрику (city-block metric)?

44. Чи можна зменшити маршрут комівояжера, якщо дозволити йому відвідувати одне і те саме місто кілька разів?

45. Яким є внесок вітчизняних науковців у вирішення задачі про прокладення магістралі?

РОЗДІЛ 5 ІНФОРМАЦІЙНИЙ ПОШУК ТА АНАЛІЗ СОЦІАЛЬНИХ МЕРЕЖ НА ОСНОВІ ЦЕНТРАЛЬНОСТІ ВЕРШИН ГРАФУ

5.1 Поняття центральності вершини

Центральність – це характеристика важливості вершини графу в контексті предметної області. Центральність можна подати у вигляді функції дійсної змінної, визначеної на вершинах графу.

Вирізняють декілька типів центральності: за впливовістю, за посередництвом, за близькістю, за степенем вершини тощо. Спершу розглянемо *центральність за впливовістю*, яку також називають *власною центральністю* (eigencentrality). За цією характеристикою оцінюють рівень впливовості вершини графу. Вважається, що вплив однієї вершини на інші передається по інцидентних дугах. Чим вище значення впливості вершини, тим сильніше вплив на мережу. Але, якщо з вершини графу виходить багато дуг, тоді її вплив розпорошується. Для оцінювання центральності за впливовістю розроблено кілька підходів, один із яких – Google PageRank і є предметом подальшого розгляду.

5.2 Алгоритм Google PageRank

Google PageRank – це алгоритм ранжування вебсторінок у пошуковій системі Google. PageRank розроблено у Стенфордському університеті Ларрі Пейджем (Larry Page) та Сергієм Бріном (Sergey Brin) в 1996 р. Перша стаття з його описом вийшла в 1998 р. [27]. Наведемо деякі теоретичні відомості з тієї статті, а також з інших робіт на цю тему [28, 29].

За принципом алгоритму PageRank вебсторінка вважається важливою, якщо на неї посилаються багато інших важливих сторінки. Щоб оцінити вплив вебсторінок будується орієнтований граф, вершини якого відповідають вебсторінкам, а дуги – посиланням. Якщо на сторінці X є посилання на сторінку Y, тоді на графі буде дуга $X \rightarrow Y$. Кожна вершина графу зважується числом, що характеризує її важливість. Це число розробники алгоритму назвали PageRank. Важливість сторінки P_i розраховується як сума відносних важливостей усіх сторінок, що мають посилання на P_i:

$$r(P_i) = \sum_{P_j \in B_{P_i}} \frac{r(P_j)}{\left|P_j\right|}, \qquad (5.1)$$

де B_{P_i} – множина вебсторінок, що мають посилання на P_i;

$\left|P_j\right|$ – кількість вихідних посилань з вебсторінки P_j.

Вважається, що будь-яка вебсторінка рівномірно розподіляє свій вплив на усі вебсторінки, на які вона має посилання. Тому у формулі (5.1) важливість $r(P_j)$ поділена на кількість вихідних посилань з вебсторінки P_j.

У (5.1) важливості сторінок P_j наперед невідомі. Для їх ідентифікації використовується ітеративний підхід. На початку роботи алгоритму усім вебсторінкам приписують однакову важливість: $r(P_i)=\frac{1}{n}$, $i=\overline{1,n}$, де n – кількість вебсторінок (кількість вершин графу). Після цього, формула (5.1) застосовується послідовно для кожної вершини графу. Протягом однієї ітерації важливість сторінок не змінюється. Важливості оновлюються після завершення ітерації, після того, як (5.1) застосовано для усіх вершин графа. Після кількох ітерацій важливості вершин стабілізуються і майже не змінюються від ітерації до ітерації. Ітеративну зміну важливості запишемо так:

$$r^{<k+1>}(P_i)=\sum_{P_j\in B_{P_i}}\frac{r^{<k>}(P_j)}{\left|P_j\right|}, \qquad (5.2)$$

де в кутових дужках $<\ >$ зазначено номер ітерації. Для графа з рис. 5.1 покроковий приклад розрахунку важливостей вебсторінок для перших 7 ітерацій подано у табл. 5.1. В останньому стовпці таблиці наведено ранг вебсторінок – порядковий номер сторінки за відсортованим списком важливості. Для аналізованого графу процес збігається за 49 ітерацій до таких значень: $r(P_1)=0.1$, $r(P_2)=0.15$, $r(P_3)=0.32$, $r(P_4)=0.21$, $r(P_5)=0.21$. Проте, як видно з табл. 5.1, вже після п’яти ітерацій важливості вебсторінок стають дуже близькими до кінцевих.

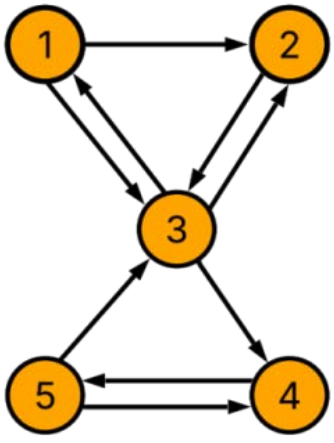


Рисунок 5.1 – Граф для ранжування п’яти вебсторінок

Таблиця 5.1 – Покрокове ранжування вебсторінок для графу з рис. 5.1

Важливість	Ітерація								Ранг
	0	1	2	3	4	5	6	7	
$r(P_1)$	0.2	0.06	0.13	0.1	0.11	0.11	0.1	0.11	5
$r(P_2)$	0.2	0.17	0.17	0.17	0.16	0.17	0.16	0.16	4
$r(P_3)$	0.2	0.4	0.3	0.32	0.33	0.3	0.33	0.31	1
$r(P_4)$	0.2	0.17	0.23	0.18	0.22	0.2	0.21	0.21	2
$r(P_5)$	0.2	0.2	0.17	0.23	0.18	0.22	0.2	0.21	3

За (5.1)–(5.2) можна обрахувати важливість кожної вебсторінки окремо. Проте, на практиці зручніше здійснити розрахунки у матричній формі для усіх вебсторінок одночасно. Важливості вебсторінок запишемо у вектор стану S розміром $1\times n$. Інформацію про переходи між вебсторінками запишемо у матрицю перехідних ймовірностей **H** розміром $n\times n$. В матриці **H** *i*-*j*-й елемент дорівнює $h_{ij}=\frac{1}{|P_i|}$, якщо є посилання з вебсторінки *i* на вебсторінку *j*, та $h_{ij}=0$, якщо таких посилань немає. Для графа з рис. 5.1 матриця **H** є такою:

$$\mathbf{H}=\begin{Vmatrix} 0 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0.33 & 0.33 & 0 & 0.33 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0.5 & 0.5 & 0 \end{Vmatrix}.$$

Сума елементів будь-якого рядка матриці **H** дорівнює 1. Це означає, що переходи з будь-якої вебсторінки на інші вважають випадковими подіями, які утворюють повну групу.

Ітераційний процес за формулою (5.2) у матричному вигляді записується так:

$$S^{<k+1>}=S^{<k>}\cdot\mathbf{H}. \qquad (5.3)$$

Ітераційний процес за (5.3) є реалізацією простого лінійного стаціонарного процесу. Тому, розрахунок значень вектора *S* за матриці **H** можна здійснити за методом степеневих ітерацій:

$$S^{<k>}=S^{<0>}\cdot\mathbf{H}^{\mathrm{k}}.$$

За формулою (5.3) здійснюється і моделювання ланцюгів Маркова, тобто марковських випадкових процесів з дискретним часом. Для таких

процесів відомо, що стаціонарний стан системи, стан, коли $k \to \infty$, можна розрахувати і без степеневих ітерацій, а за власним вектором E матриці **H:** $S^{<\infty>} = \frac{E}{\sum E}$. Власний вектор знаходиться з такого рівняння:

$$E \cdot \mathbf{H} = \lambda_{\max} \cdot \mathbf{H},$$

де $\lambda_{\max}$ – максимальне власне значення матриці **H**.

Для матриць перехідних ймовірностей $\lambda_{\max}=1$, тому власний вектор знаходиться з рівняння $E \cdot \mathbf{H} = \mathbf{H}$. Зауважимо, що E – це лівий власний вектор, тому під час його розрахунку в математичних пакетах матрицю **H** потрібно транспонувати.

Розробники алгоритму PageRank майже відразу виявили кілька проблем. Одна з них – проблема стічних рангів. Вона полягає в тому, що деякі вебсторінки не містять посилання на інші вебсторінки. В матриці **H** ці вершини описуються дугою-петлею, яка зважена одиничною ймовірністю. Такі вершини називаються *термінальними* (dangling) або *стічними* (sink). Термінальні вершини від ітерації до ітерації накопичують все вищу і вищу важливість, монополізуючи переданий вплив з інших вебсторінок. Як приклад, на рис. 5.2 наведено граф з термінальною вершиною 3. В стаціонарному режимі, коли $k \to \infty$, важливість вершини 3 дорівнює 1, а вершин 1 та 2 – 0.

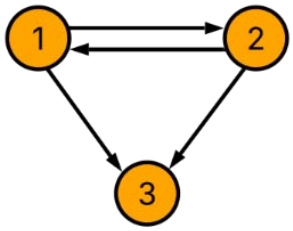


Рисунок 5.2 – Приклад вебграфу з стічним рангом вершини

Термінальні вершини можуть призвести і до кластеризації мережі, коли важливість однієї підмножини вебсторінок обнуляється, і перерозподіляється на інші. На вебграфі з рис. 5.3, вершини 4, 5 та 6 утворюють кластер накопичування рангу (hoard PageRank) – після 13 ітерацій моделювання за формулою (5.3) важливість вершин становить:

$$S^{<13>} = \begin{pmatrix} 0 & 0 & 0 & 0.27 & 0.13 & 0.2 \end{pmatrix}.$$

Також є проблема циклів. Граф з рис. 5.4 описує ситуацію, коли перша вебсторінка посилається лише на другу, а друга – лише на першу, що утворює нескінчений цикл. Якщо за формулою (5.3) розпочати ітеративний процес

моделювання з початкового стану $S^{<0>} = (1 \quad 0)$, то він не збіжиться. Матимуть місце гонки – стан системи циклічно перемикатиметься між $(1 \quad 0)$ та $(0 \quad 1)$.

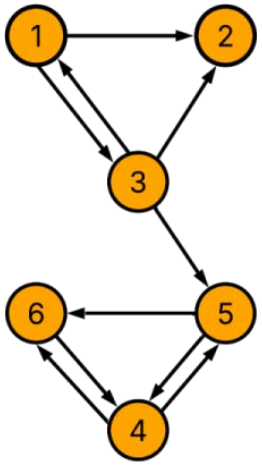


Рисунок 5.3 – Граф з кластеризацією вебсторінок

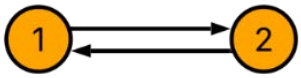


Рисунок 5.4 – Граф, який створює гонки під час ранжування

Для вирішення зазначених проблем запроваджено кілька модифікацій. Вводиться концепція випадкового вебмандрівника (random surfer). Це користувач, який випадково переходить за посиланнями з однієї вебсторінки на іншу на деякому вебграфі. Коли він відкриває вебсторінку з декількома вихідними посиланнями, то випадково вибирає одну, переходить по ній, і продовжує свій обхід нескінченно. Якщо випадковий вебмандрівник потрапляє у стічну вебсторінку (картинки, pdf-файли тощо), тоді вважається, що він залишає її відразу на наступному кроці і з однаковими шансами переходить на будь-яку іншу веб-сторінку. Для опису такого алгоритму діяльності вебмандрівника у рядках термінальних вершин матриці перехідних ймовірностей усім елементам присвоюємо значення $\frac{1}{n}$. Для вебграфу з рис. 5.3 матриця **H** модифікується у таку матрицю **G**:

$$\mathbf{G} = \begin{Vmatrix} 0 & 0.5 & 0.5 & 0 & 0 & 0 \\ 0.17 & 0.17 & 0.17 & 0.17 & 0.17 & 0.17 \\ 0.33 & 0.33 & 0 & 0 & 0.33 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{Vmatrix}.$$

Випадковий вебмандрівник може без наводок перейти на інші сторінки не лише з термінальної вершини, але і з будь-якої іншої. Такий спонтанний перехід називається телепортацією. Можливість телепортації вебмандрівника описують шляхом рандомізації матриці перехідних ймовірностей. Для цього кожен елемент матриці **G** перераховують у такий спосіб:

$$\mathrm{G}_{ij} = \alpha \mathrm{G}_{ij} + \frac{(1-\alpha)}{n}, \quad i = 1,\ 2, ...,\ n, \quad j = 1,\ 2, ...,\ n,$$

де $\alpha \in (0,\ 1)$ – коефіцієнт так званої дисциплінованості вебмандрівника. Чим менший цей коефіцієнт, тим більші шанси телепортації.

Зазначена рандомізація матриці перехідних ймовірностей означає, що з ймовірністю α перехід з вершини відбувається за посиланнями з відповідної вебсторінки, а з ймовірністю $(1-\alpha)$ перехід відбувається випадково. Типове значення α, яке застосовується на практиці, становить 0.85. Це означає, що у 85% випадків вебмандрівник на поточній сторінці вибирає одне із запропонованих посилань, а у 15% телепортується на випадкову сторінку.

Після описаних модифікацій отримуємо Google-матрицю перехідних ймовірностей. Вона є не лише стохастичною, але і *примітивною*. Примітивність матриці означає, що вона є *нескорочуваною* (irreducible) та *аперіодичною* (aperiodic). Нескорочувана матриця перехідних ймовірностей задає повнозв'язним граф, на якому є шлях між будь-якою парою вершин. Аперіодичність означає, що можна повернутись в будь-яку вершину за різну кількість кроків, а не лише за фіксовану, наприклад, за 2 або 3 кроки. Це дозволяє уникнути гонок, які мають місце для графу з рис. 5.4.

Отже, ітераційний процес (5.3) модифіковано у такий спосіб:

$$S^{<k+1>} = S^{<k>} \cdot \mathbf{G}. \qquad (5.5)$$

5.3 Застосування графів для аналізу соціальних мереж

Соціальна мережа – це онлайнова платформа, яку люди використовують, щоб вибудувати соціальні зв'язки та взаємодіяти з іншими людьми, з якими мають спільні кар'єрні чи особисті інтереси, погляди, досвід, вподобання, заняття або зв'язки в реальному житті. Під час аналізу

соціальних мереж їх учасників називають акторами. Кожен актор є соціальною одиницею. Якщо всі актори одного типу, наприклад, люди у групі, тоді вони утворюють так звану однодомну мережу. Дводомною називається соціальна мережа з двома типами акторів, наприклад, учні та вчителі у школі. Окремим типом дводомних мереж є мережі належності, яку утворюють множина акторів та множина подій (клуби, організації, громадські організації), до яких може відноситись актор або підгрупа акторів.

Однією із цілей застосування теорії графів є кластеризація соціальної мережі, тобто формалізоване виявлення в ній груп. Група – це фрагмент графу соціальної мережі, щільність внутрішніх зв'язків в якому домінує над щільністю зовнішніх зв'язків. Малозв'язні групи, що мають розріджені зв'язки всередині, але в той самий час об'єднують декілька щільних підгруп – є дуже вразливими. Зв'язки в таких підгрупах, зазвичай, є подібними та надлишковими, тому для групи вони мають меншу цінність, ніж ті, що проходять через слабкі зв'язки між підгрупами.

Якою має бути ідеальна підгрупа за ефективністю взаємодії, надійністю та живучістю – питання все ще залишається відкритим. В [34] висунуто гіпотезу, що ідеальною підгрупою є повнозв'язна компонента графу – сильний альянс. Відповідно до цієї гіпотези розроблено цілий ряд моделей підгруп, зокрема:

- *k*-clique – максимальний підграф, в якому відстань між двома будь-якими парами вершин не більше *k*;
- *k*-plex – максимальний підграф, який містить g вершин та має не менше як (g–k) суміжних вершин у підграфі;
- *k*-core – максимальний підграф, кожна вершина якого має не менше *k* суміжних вершин всередині підграфу;
- *LS*-множина – підграф, в якому довільна множина вершин має більше внутрішніх ребер, ніж зовнішніх;
- λ-множина – підграф, в якому міра зв'язності будь-якої пари вершин всередині підгрупи більша, ніж поза нею. Зв'язність вершин $\lambda(i, j)$ визначається через мінімальну кількість зв'язків, які потрібно видалити, щоб вершини *i* та *j* стали недосяжними.

Пошук підгруп досить трудомістка задача, наприклад, складність пошуку *LS*-множин та λ-множин дорівнює $O\left(n^5\right)$, де n – кількість акторів соціальної мережі і, відповідно, кількість вершин аналізованого графу. Тому точні алгоритми виявлення підгруп можна застосувати лише для малих соціальних мереж. Розглянемо простіші методи дослідження соціальних мереж на основі аналізу центральності вершин.

В попередньому підрозділі вже розглядалась центральність вершини за впливовістю у орієнтованих вебграфах. Існують й інші показники

центральності, які застосовуються для аналізу мереж. В соціальних мережах центральність описує наскільки впливовим є місце актора в ній. Для аналізу важливості в соціальних мережах застосовують такі показники центральності як-от: центральність за степенем вершини (degree centrality), центральність за посередництвом (betweeness centrality), центральність за близькістю (closeness centrality) та центральність за впливовістю (eigenvector centrality) [30]. Проаналізуємо перші 3 показники центральності.

Центральність за степенем вершини ставить у відповідність кожній вершині нормалізовану кількість інцидентних ребер. Вершини з високим показником центральності за степенем вершини вважаються більш впливовими у соціальних мережах, оскільки вони мають багато зв'язків з іншими учасниками соціальних мереж і можуть передавати їм інформацію, ресурси або вплив. Центральність за степенем вершини враховує лише прямі зв'язки та не враховує силу чи якість цих зв'язків і загальну топологію мережі. Математично центральність за степенем вершини записується так:

$$C(v) = \frac{\deg(v)}{n-1},$$

де $\deg(v)$ – кількість ребер, інцидентних вершині v, тобто степінь вершини;

$n-1$ – максимально можливий степінь вершини.

Центральність за посередництвом є показником центральності на основі найкоротших маршрутів. Цей показник розглянуто в розділі 3, проте не наголошувалося, шо він характеризує центральність вершини графу. Вершини з високою центральністю за посередництвом вважаються топологічно ключовими, оскільки вони відіграють важливу роль у з'єднанні різних частин соціальної мережі. Через такі вершини може відбуватись передача основного обсягу інформації, ідей та інших ресурсів між різними спільнотами.

Центральність за посередництвом можна використовувати для виявлення мостів між різними спільнотами в соціальній мережі. *Мостом* є вершина, видалення якої з графу утворює більше ніж одну компоненту зв'язності. Центральність за посередництвом обчислюється так:

$$g(v) = \frac{1}{N} \sum_{\forall s, v, t:\, s \neq v,\, s \neq t,\, v \neq t,} \frac{\sigma_{st}(v)}{\sigma_{st}},$$

де $\sigma_{st}(v)$ – кількість найкоротших маршрутів від вершини *s* до вершини *t*, які проходять через вершину v;

σ_{st} – кількість найкоротших маршрутів від вершини *s* до вершини *t*;

N – загальна кількість найкоротших маршрутів на графі, в яких вершина *v* не є крайньою.

Центральність за посередництвом нормалізована – вона набуває значень з інтервалу [0, 1]. Це досягається за рахунок дільника *N*. Його значення для орієнтованих графів дорівнює $(n-1)(n-2)$, а для неорієнтованих воно буде удвічі меншим: $0.5(n-1)(n-2)$. Оскільки мова йде про графи соціальних мереж, то, залежно від специфіки зв'язків, граф може розглядатися і як неорієнтований, і як орієнтований.

Центральність за близькістю визначає, наскільки близько вершина знаходиться до всіх інших вершин графу. Центральність за близькістю розраховується як обернена величина середньої найкоротшої відстані від аналізованої вершини до всіх інших вершин. Чим більша для вершини центральність за близькістю, тим ближче вона розташована до всіх інших вершин графу. За центральністю за близькістю можна оцінити як швидко інформація або вплив поширюються від аналізованої вершини до всіх інших вершин графу. На практиці вершини з високим показником центральності за близькістю доцільно розглядати як кандидатів для швидкого поширення інформації. З іншого боку, такі вершини привабливі для атак, якщо ціллю є руйнування відповідної соціальної спільноти. Центральність за близькістю розраховується так:

$$C(v) = \frac{n-1}{\sum\limits_{\forall v,u:\, v \neq u} d(v,u)},$$

де $d(v,u)$ – довжина найкротшого маршруту між вершинами v та u.

Проілюструємо застосування показників центральності на прикладі соціального графу «Клуб карате Зехарі» (Zachary's karate club). Цей граф сформовано в рамках дослідження зв'язків між учасниками клубу карате [31, 32]. Граф має 34 вершини, які відповідають членам клубу карате. Ребра графа позначають позаклубні зв'язки членів цього клубу. Одного дня між адміністратором та інструктором клубу виник конфлікт, який призвів до розколу клубу на дві групи. Одна група утворила новий клуб зі старим інструктором, а члени іншої групи знайшли нового інструктора або кинули карате. На основі аналізу графу Вайн Зехарі (Wayne Zachary) правильно

відніс усіх членів клубу до своїх груп за виключенням одного учасника. Розберемося, як це відбулося.

Показники центральності за степенем вершини, за посередництвом та за близькістю для графу «Клуб карате Зехарі» подано на рис. 5.5–5.7. На цих рисунках вершина 0 позначає інструктора, а вершина 33 – адміністратора. Кожна вершина зафарбована кольором відповідно до показника центральності. Показники центральності за степенем вершини є найвищими для інструктора та адміністратора (табл. 5.2). Показники центральності за посередництвом також є найвищими для інструктора та адміністратора, проте порядок інший. Показники центральності за близькістю є найвищими для інструктора та члена клубу з номером 2, а адміністратор займає третє місце. Направду, відрив між ними символічний. На основі трьох показників центральності, вершини 0 та 33 найбільш важливі для мережі, навколо яких власне і відбувся розкол клубу.

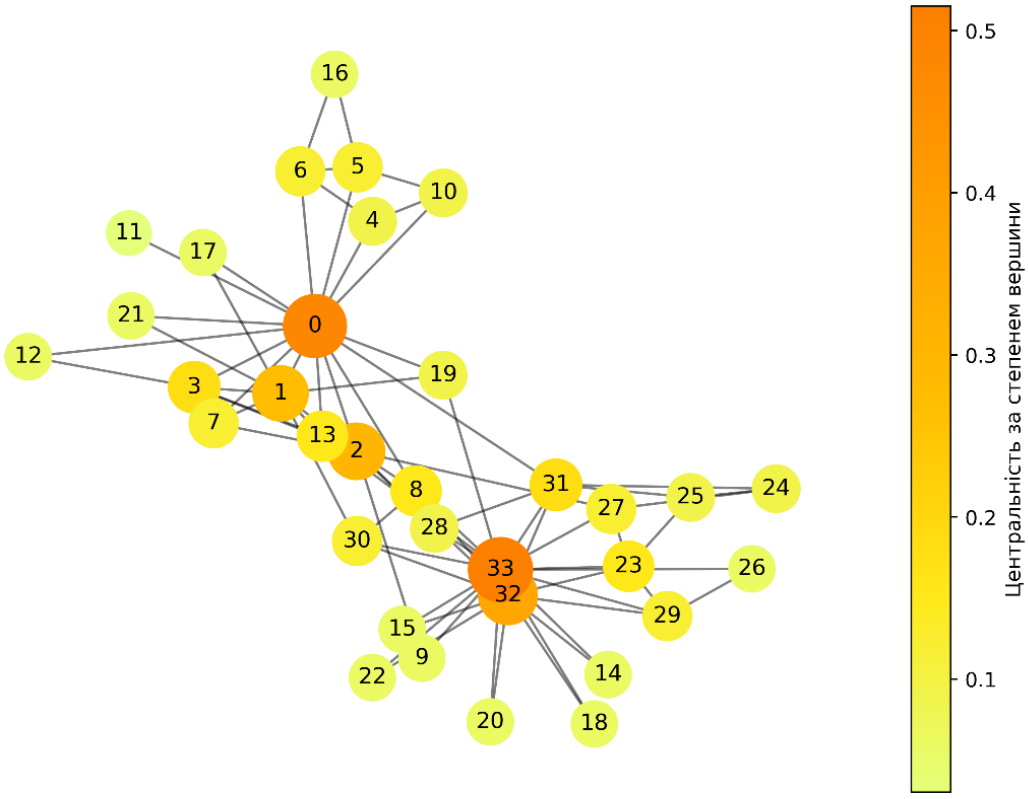


Рисунок 5.5 – Ілюстрація центральності за степенем вершини у графі «Клуб карате Зехарі»

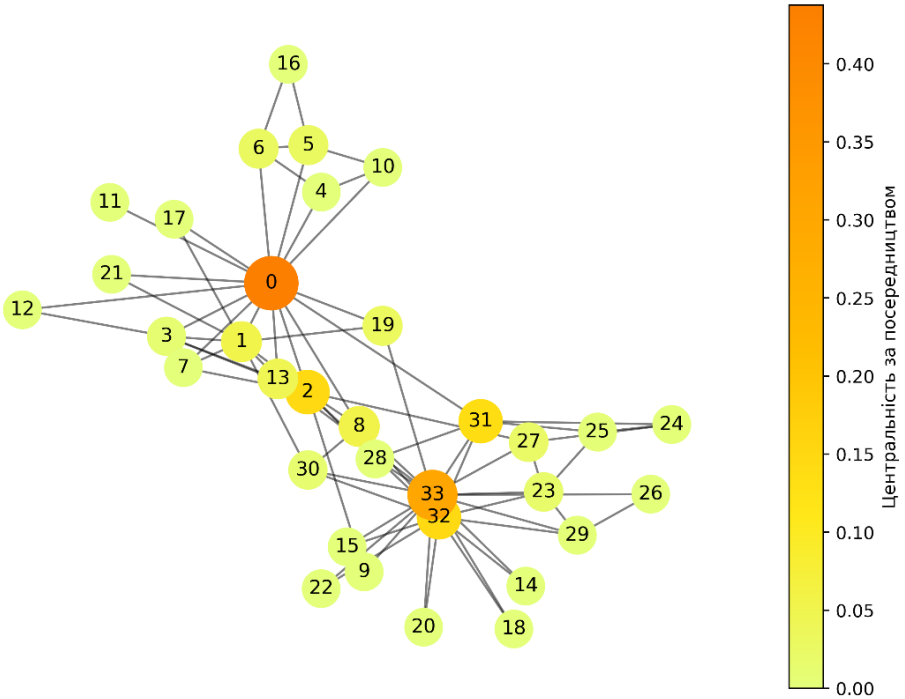


Рисунок 5.6 – Ілюстрація центральності за посередництвом у графі «Клуб карате Зехарі»

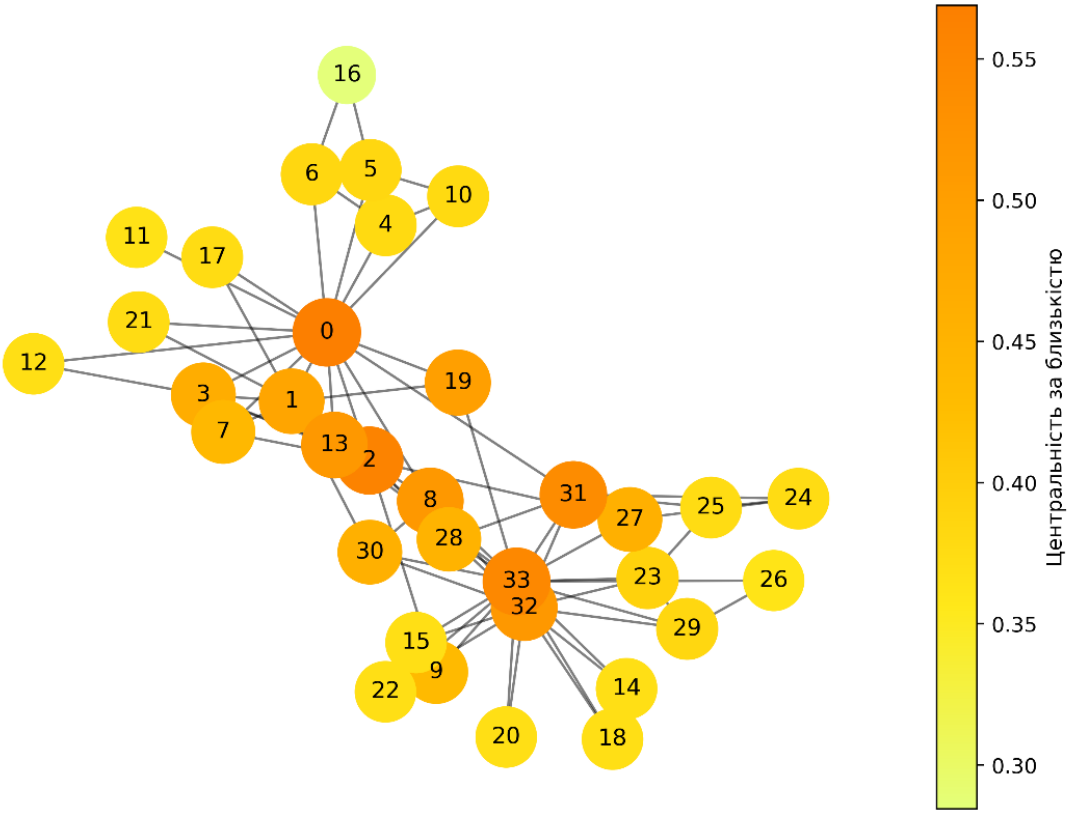


Рисунок 5.7 – Ілюстрація центральності за близькістю у графі «Клуб карате Зехарі»

Таблиця 5.2 – Топ-15 вершин за різними показниками центральності

Вершина	За степенем	Вершина	За посеред-ництвом	Вершина	За близь-кістю
33	0.51	0	0.44	0	0.57
0	0.48	33	0.3	2	0.56
32	0.36	32	0.15	33	0.55
2	0.30	2	0.14	31	0.54
1	0.27	31	0.13	8	0.51
3	0.18	8	0.06	13	0.51
31	0.18	1	0.05	32	0.51
8	0.15	13	0.05	19	0.5
13	0.15	19	0.03	1	0.48
23	0.15	5	0.03	3	0.46
5	0.12	6	0.03	27	0.45
6	0.12	27	0.02	30	0.45
7	0.12	23	0.02	28	0.45
27	0.12	30	0.01	7	0.44
29	0.12	3	0.01	9	0.43

За даними з табл. 5.2 видно, що за центральністю за посередництвом можна виділити два мости – вершини 0 та 33. Їх центральність значно відрізняється від наступних за рейтингом вершин. Видаливши одну або обидві ці вершини можна суттєво вплинути на зв'язність між усіма учасниками клубу, утворивши щонайменше 2 компоненти зв'язності. Вершини 0 та 33 також лідирують за центральністю за степенем, проте їх відрив від наступних вершин не сильно помітний. Вершин з високим рівнем цього показника центральності відносно мало, що вказує на низький рівень спілкування усіх учасників поза межами клубу. Показник центральності за близькістю засвідчує, що більшість учасників знаходяться в околі один до одного, тому інформація між ними передається доволі швидко.

5.4 Завдання для самостійного дослідження

Базовий рівень

1. Реалізувати програмно алгоритм Google PageRank.

2. Побудувати вебграф згідно з варіантом та ідентифікувати важливості кожної його вебсторінки за алгоритмом Google PageRank. Побудувати залежність рівня збіжності алгоритму Google PageRank від кількості ітерацій. Для прикладу, на рис. 5.8 подано таку залежність для вебграфу із 2500 вебсторінок. Видно, що алгоритм фактично збігається за 22 ітерації. Для оцінювання збіжності доцільно використати евклідову норму між векторами важливостей вершин на сусідніх ітераціях. Зупинити ітераційний процес варто, коли різниця між поточним і попереднім вектором важливостей

становитеме менше 10^{-6}. Експерименти провести для випадку, коли α=0.85. Вивести топ-25 вебсторінок за рівнем важливості (табл. 5.3).

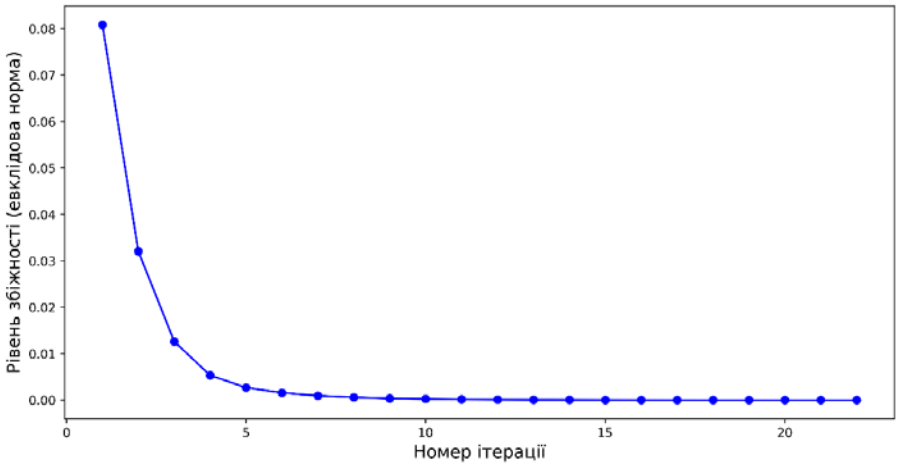


Рисунок 5.8 – Стабілізація вектора впливовості протягом ітерацій

Таблиця 5.3 – Топ-25 вебсторінок за важливістю, розрахованою за алгоритмом Google PageRank

Вебсторінка	Важливість	Ранг
https://en.wikipedia.org/wiki/Mathematics	0.002001	1
https://en.wikipedia.org/wiki/Model-driven_engineering	0.001264	2
https://en.wikipedia.org/wiki/Software_quality_assurance	0.001199	3
https://en.wikipedia.org/wiki/Feature-driven_development	0.001191	4
https://en.wikipedia.org/wiki/Team_software_process	0.001189	5
https://en.wikipedia.org/wiki/Software_testing	0.001141	6
https://en.wikipedia.org/wiki/Data_modeling	0.001130	7
https://en.wikipedia.org/wiki/Calculus	0.001117	8
https://en.wikipedia.org/wiki/Incremental_build_model	0.001108	9
https://en.wikipedia.org/wiki/Real_number	0.001100	10
https://en.wikipedia.org/wiki/Algeria	0.001076	11
https://en.wikipedia.org/wiki/Software_configuration_management	0.001061	12
https://en.wikipedia.org/wiki/Extreme_programming	0.001048	13
https://en.wikipedia.org/wiki/Cleanroom_software_engineering	0.001041	14
https://en.wikipedia.org/wiki/Extreme_programming_practices	0.001010	15
https://en.wikipedia.org/wiki/Adaptive_software_development	0.000968	16
https://en.wikipedia.org/wiki/Digital_object_identifier	0.000947	17
https://en.wikipedia.org/wiki/Software_development	0.000931	18
https://en.wikipedia.org/wiki/Topology	0.000926	19
https://en.wikipedia.org/wiki/Software_maintenance	0.000897	20
https://en.wikipedia.org/wiki/Software_design	0.000888	21
https://en.wikipedia.org/wiki/Test-driven_development	0.000879	22
https://en.wikipedia.org/wiki/Personal_software_process	0.000874	23
https://en.wikipedia.org/wiki/Software_construction	0.000868	24
https://en.wikipedia.org/wiki/Unified_process	0.000866	25

Варіанти завдань генеруються скриптом, який наведено нижче. Скрипт створює вебграф на основі зв'язків деякої підмножини вебсторінок Вікіпедії. Для цього використовується Python-бібліотека wikipedia. Генерування вебграфу здійснюється таким чином. Згідно із варіантом вибирається початкова вебсторінка, з якої розпочинається обхід графу в ширину. Для цієї вебсторінки дістаємо усі посилання на інші вебсторінки та вибираємо деяку підмножину з них, щоб дійти глибше і побудувати вебграф з більшою кількістю дуг. Додаємо дуги від початкової вебсторінки до вебсторінок з вибраної підмножини посилань. Продовжуємо цю процедуру для щойно доданих вебсторінок до тих пір, поки у вебграфі не буде 2500 вершин. Для генерування вебграфу необхідно у функцію `generate_dataset_for_variant` передати номер варіанта. Функція побудує вебграф та збереже його у файли, оскільки ця операція є ресурснозатратною. Щоб прочитати дані про вебграф використовується функція `read_graph_dataset` – вона зчитує збережені файли і повертає список вебсторінок з посиланнями на них та матрицю суміжності. Файл `NodesUrls.csv` містить номер вебсторінки та її посилання. Файл `NodeToNodeMapping.csv` містить список суміжності для згенерованого вебграфу.

Скрипт для генерування вебграфу:

```
import ast
import random
import re
from time import sleep

import numpy as np
import pandas as pd
import wikipedia


def extract_links_from_html(html):
    pattern = r'<a\s+(?:[^>]*?\s+)?href=["\'](.*?)["\']'
    res = re.findall(pattern, html)
    excluded_links = {"File:", "Category:", "Template:",
"Template_talk:", "Help:", "Portal:", "Talk:", "Wikipedia:",
                      "Special:",
                      "wikipedia.org",
"https://www.wikidata.org", "/ISBN", "Digital_object_identifier",
"/ISSN",
```

```
"/International_Organization_for_Standardization",
"/Semantic_Scholar", "/Bibcode", "GS1",
                      "/wiki/Wiki"}
    return [l for l in res if "/wiki/" in l and all(e not in l
for e in excluded_links)]


def traverse_links(start_query="Discrete mathematics"):
    wiki_page = wikipedia.page(start_query)

    links_threshold = 50
    first_random_links_count = 10

    links = [l for l in
dict.fromkeys(extract_links_from_html(wiki_page.html())).keys()]
    links = links[:min(links_threshold, len(links))]
    links = random.choices(links, k=min(first_random_links_count,
len(links)))

    processed_urls = {wiki_page.url}

    curr_node_index = 0
    visited_nodes = {start_query}
    nodes_info = {curr_node_index: wiki_page.content}
    node_query_mapping = {curr_node_index: start_query}
    query_node_mapping = {start_query: curr_node_index}
    node_to_node_mapping = {}
    nodes_urls = {curr_node_index: wiki_page.url}

    all_indexes = [curr_node_index]
    all_links = [links]

    failed_links = set()

    nodes_threshold = 2500
    while len(all_links) > 0 and len(visited_nodes) <
nodes_threshold:
        curr_links = all_links.pop(0)
        curr_parent_index = all_indexes.pop(0)
        for link in curr_links:
            query = link.split("/")[-1]

            if query not in visited_nodes:
```

```
                try:
                    page = wikipedia.page(query)
                except Exception as e:
                    print(f"Failed to retrieve: {query}, {e}")
                    failed_links.add(link)
                    continue

                if page.url in processed_urls:
                    continue

                new_links = [l for l in
dict.fromkeys(extract_links_from_html(page.html())).keys()]
                new_links = new_links[:min(links_threshold,
len(new_links))]
                new_links = random.choices(new_links,
k=min(first_random_links_count, len(new_links)))

                curr_node_index = curr_node_index + 1
                all_indexes.append(curr_node_index)
                all_links.append(new_links)

                nodes_info[curr_node_index] = page.content
                node_query_mapping[curr_node_index] = query
                query_node_mapping[query] = curr_node_index
                nodes_urls[curr_node_index] = page.url

                visited_nodes.add(query)
                processed_urls.add(page.url)
                print(f"Processed: {page.url},
{curr_node_index}")
                sleep(0.1)

            node_to_node_mapping[curr_parent_index] =
node_to_node_mapping.get(curr_parent_index, []) + \

[query_node_mapping[query]]

    return visited_nodes, node_to_node_mapping, nodes_info,
node_query_mapping, query_node_mapping, nodes_urls, failed_links


def generate_dataset_for_variant(variant):
    if variant <= 0 or variant > 50:
        raise Exception("Variant must be between 0 and 50")
```

```
    wiki_page = wikipedia.page("Discrete mathematics")
    links = [l for l in
dict.fromkeys(extract_links_from_html(wiki_page.html())).keys()]
    successful_links = []
    for link in links:
        try:
            p = wikipedia.page(link)
            successful_links.append(link)
            print(f"Processed: {link}, {p.url}")
        except Exception as e:
            print(f"Failed to retrieve: {link}, {e}")
        if len(successful_links) >= 50:
            break

    query = successful_links[variant % len(successful_links)]
    visited_nodes, node_to_node_mapping, nodes_info,
node_query_mapping, query_node_mapping, nodes_urls, failed_links
= traverse_links(
        query)

    pd.DataFrame({"node_index": node_to_node_mapping.keys(),
"links": node_to_node_mapping.values()}).to_csv(
        "NodeToNodeMapping.csv", index=False, sep=";")
    pd.DataFrame({"node_index": node_query_mapping.keys(),
"query": node_query_mapping.values()}).to_csv(
        "NodesQuery.csv", index=False, sep=";")
    pd.DataFrame({"node_index": nodes_urls.keys(), "url":
nodes_urls.values()}).to_csv(
        "NodesUrls.csv", index=False, sep=";")
    pd.DataFrame(failed_links,
columns=["failed_link"]).to_csv("FailedLinks.csv", index=False,
sep=";")


def read_graph_dataset():
    nodo_to_node_df = pd.read_csv("NodeToNodeMapping.csv",
sep=";")
    nodo_to_node_df["links"] =
nodo_to_node_df["links"].apply(ast.literal_eval)

    nodes_url = {}
    nodes_url_df = pd.read_csv("NodesUrls.csv", sep=";")
    nodes_count = nodes_url_df.shape[0]
```

```
    for index, row in nodes_url_df.iterrows():
        nodes_url[row["node_index"]] = row["url"]

    adjacency_matrix = np.zeros((nodes_count, nodes_count))
    for index, row in nodo_to_node_df.iterrows():
        for target in row["links"]:
            adjacency_matrix[row["node_index"], target] = 1

    return nodes_url, adjacency_matrix
```

3. Провести серію експериментів, запускаючи алгоритм Google PageRank за таких коефіцієнтів дисциплінованості вебмандрівника: $\alpha \in \{0.65,\ 0.75,\ 0.85,\ 0.95,\ 0.999\}$. Для кожного експерименту зберегти топ-25 вебсторінок за впливовістю, а також кількість ітерацій, які було виконано для досягнення збіжності. Побудувати графік залежності кількості ітерацій, необхідних для виконання умови збіжності, від значення коефіцієнта дисциплінованості вебмандрівника.

4. Розглядається вебсторінка, яка має другий ранг за $\alpha = 0.85$. Встановити, яку мінімальну кількість додаткових посилань необхідно зробити з цієї веб-сторінки на інші, щоб вивести її на першу позицію. Реалізувати пошук за повним перебором та за жадібним алгоритмом. За жадібним алгоритмом на кожній ітерації додається посилання, яке найсильніше скорочує відставання від лідера.

5. Встановити, яку мінімальну кількість посилань необхідно видалити з веб-сторінки, що має другий ранг, щоб мінімізувати її відставання від лідера за рівнем впливовості.

6. Встановити, яку мінімальну кількість посилань на сторінку з першим рангом необхідно видалити, щоб сторінка з другого рангу перейшла на перший.

Поглиблений рівень

1. Дослідити показники центральності за степенем вершини, за посередництвом та за близькістю. Дослідження проводяться на графах соціальної мережі Facebook. Початкові дані для дослідження доступні з відкритого джерела графів різних типів – NetworkRepository, що знаходиться за посиланням – https://networkrepository.com/socfb.php [33]. Там є окрема категорія графів побудованих на основі соціальної мережі Facebook.

Назви графів для кожного варіанта подано в табл. 5.4.

2. Програмно реалізувати обрахунок показників центральності за степенем вершини, за посередництвом та за близькістю.

3. Обрахувати показники центральності для графу, заданого у варіанті. Граф візуалізувати у форматі рис. 5.9. Якщо у графу є більше однієї

компоненти зв’язності, як це показано на рис. 5.9, то обрахунки необхідно проводити для найбільшої компоненти зв’язності. Порівняти розподіли показників центральності, використовуючи графіки, аналогічні до рис. 5.10–5.12. Розподіл центральності за степенем вершини (рис. 5.10) плавно спадає від максимального значення до мінімального. Це вказує на те, що більшість вершин мають багато зв’язків, як це видно і з рис. 5.9. Розподіл центральності за посередництвом різко обривається (рис. 5.11) – за цим показником важливих вершин значно менше. Розподіл центральності за близькістю (рис. 5.12) дуже повільно спадає – вершини знаходяться дуже близько одна від одної, тому інформація може швидко поширюватися між учасникам соціальної мережі.

Таблиця 5.4 – Варіанти завдань

Варіант	Граф	Варіант	Граф
1	socfb-American75	26	socfb-Harvard1
2	socfb-Amherst41	27	socfb-Haverford76
3	socfb-Auburn71	28	socfb-Howard90
4	socfb-BC17	29	socfb-Indiana
5	socfb-BU10	30	socfb-JMU79
6	socfb-Baylor93	31	socfb-JohnsHopkins55
7	socfb-Berkeley13	32	socfb-Lehigh96
8	socfb-Bingham82	33	socfb-MIT
9	socfb-Bowdoin47	34	socfb-MIT8
10	socfb-Brandeis99	35	socfb-MSU24
11	socfb-Brown11	36	socfb-MU78
12	socfb-Bucknell39	37	socfb-Maine59
13	socfb-CMU	38	socfb-Maryland58
14	socfb-Cal65	39	socfb-Mich67
15	socfb-Carnegie49	40	socfb-Michigan23
16	socfb-Colgate88	41	socfb-Middlebury45
17	socfb-Columbia2	42	socfb-Mississippi66
18	socfb-Cornell5	43	socfb-NYU9
19	socfb-Dartmouth6	44	socfb-Northeastern19
20	socfb-Duke14	45	socfb-Northwestern25
21	socfb-Emory27	46	socfb-NotreDame57
22	socfb-FSU53	47	socfb-OR
33	socfb-GWU54	48	socfb-Oberlin44
24	socfb-Georgetown15	49	socfb-Oklahoma97
25	socfb-Hamilton46	50	socfb-Penn94

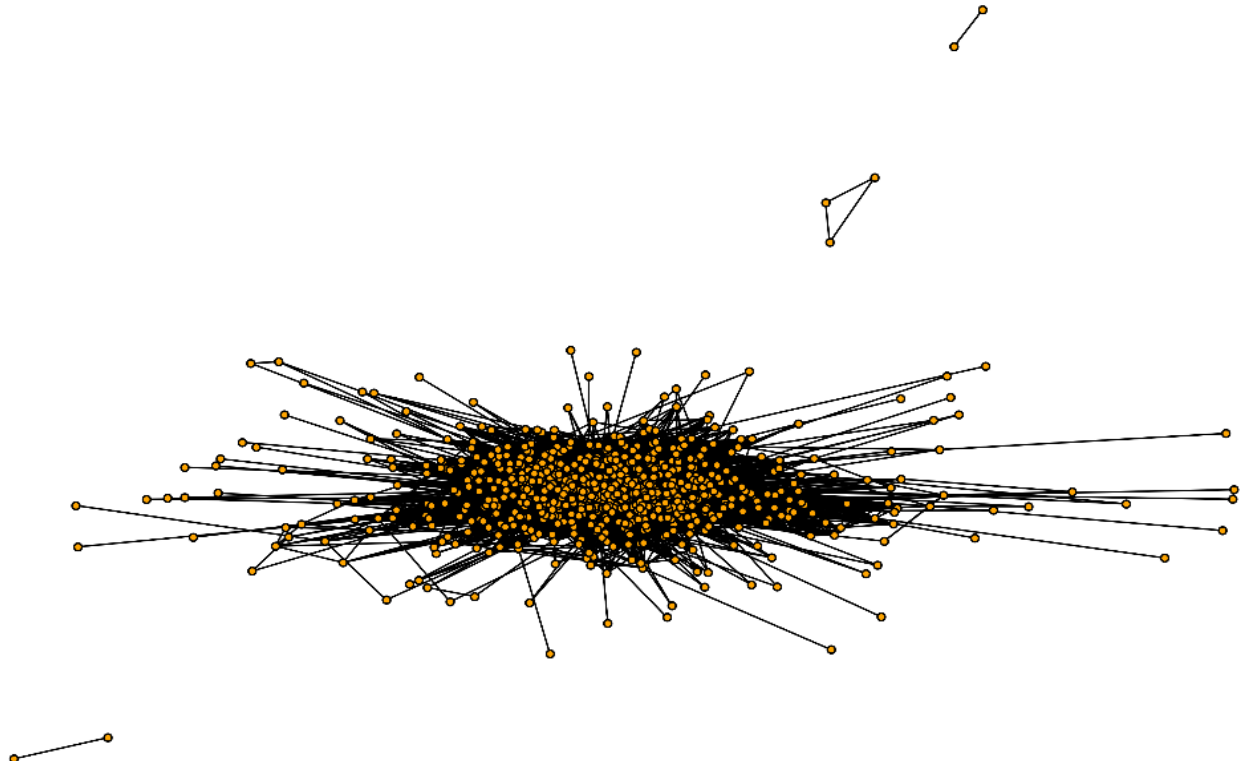


Рисунок 5.9 – Граф на основі соціальної мережі Facebook для задачі Caltech36

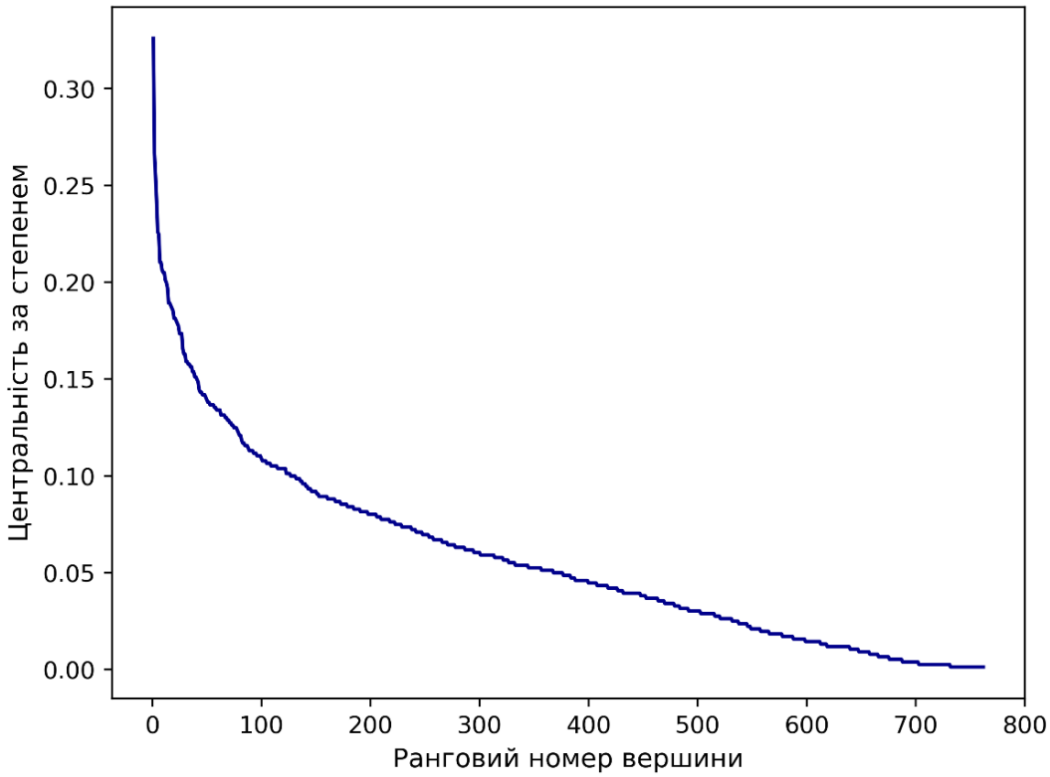


Рисунок 5.10 – Розподіл центральності за степенем вершини

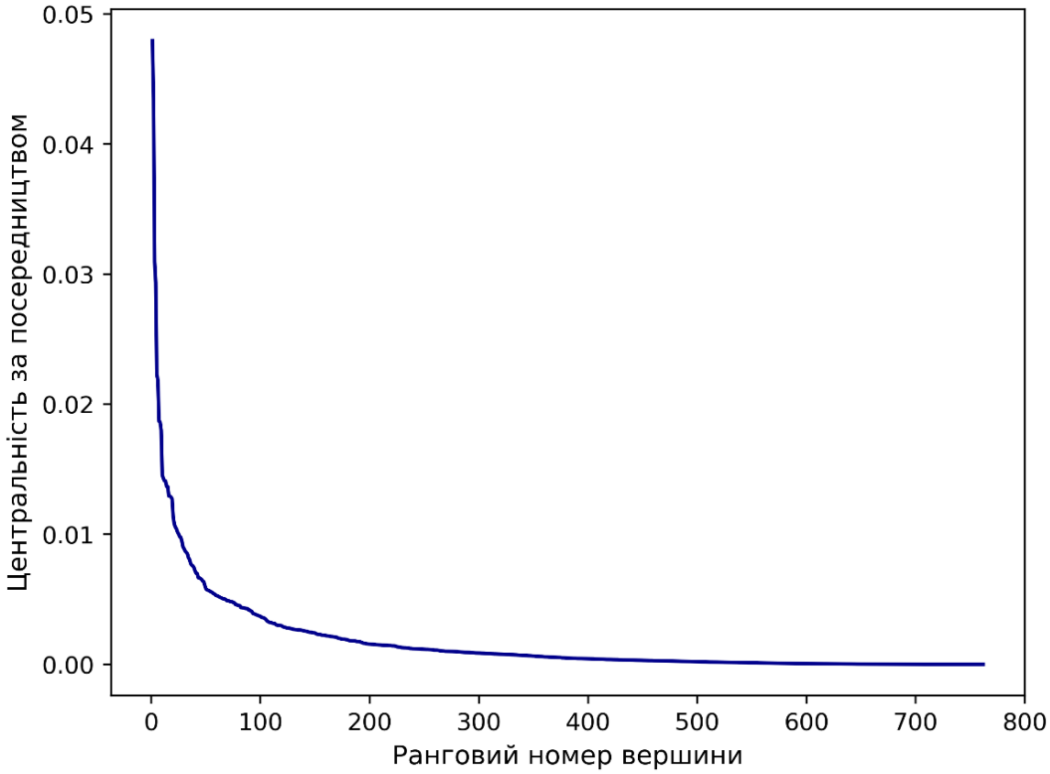


Рисунок 5.11 – Розподіл центральності за посередництвом

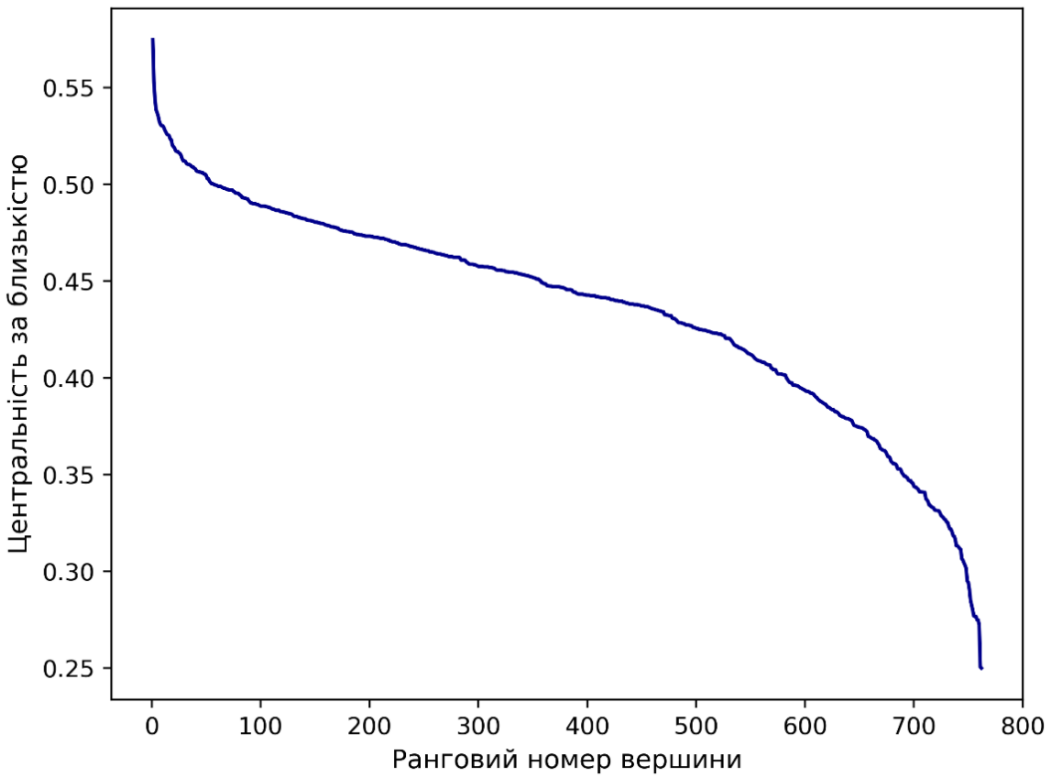


Рисунок 5.12 – Розподіл центральності за близькістю

4. Відобразити топ-10 вершин за різними показниками центральності аналогічно до табл. 5.2 та порівняти їх.

5. Провести аналіз найкоротших маршрутів між усіма парами вершин та перевірити на практиці гіпотезу про 6 рукостискань. Ця гіпотеза полягає у тому, що найкоротший соціальний ланцюжок між будь-якими двома особами містить лише 5 посередників, і пройти за цим ланцюжком можна за 6 рукостискань. Для розрахунків центральності за посередництвом використовують довжини найкоротших маршрутів. Їх можна використати і для цього завдання. Необхідно порахувати частоту довжин найкоротших маршрутів і результати подати аналогічно до рис. 5.13. Середня довжина найкоротшого маршруту для цього графу становить 2.3, а максимальна – 6.

6. Провести аналіз кореляції між показниками центральності. Для цього необхідно проранжувати вершини за показниками центральності. Для групи вершин з однаковим значенням показника центральності використовувати середнє значення відповідних рангів. Результати подати у форматі рис. 5.14–5.16. Кожен маркер на цих рисунках відповідає одній вершині. По осі *x* подано ранг вершин за одним типом центральності, а по осі *y* – ранг цієї самої вершини за іншим типом центральності.

7. Почергово видалити одну, дві та три вершини з найвищими показниками центральності за посередництвом та перевірити, як це вплине на зв’язність графу та на довжини найкоротших маршрутів.

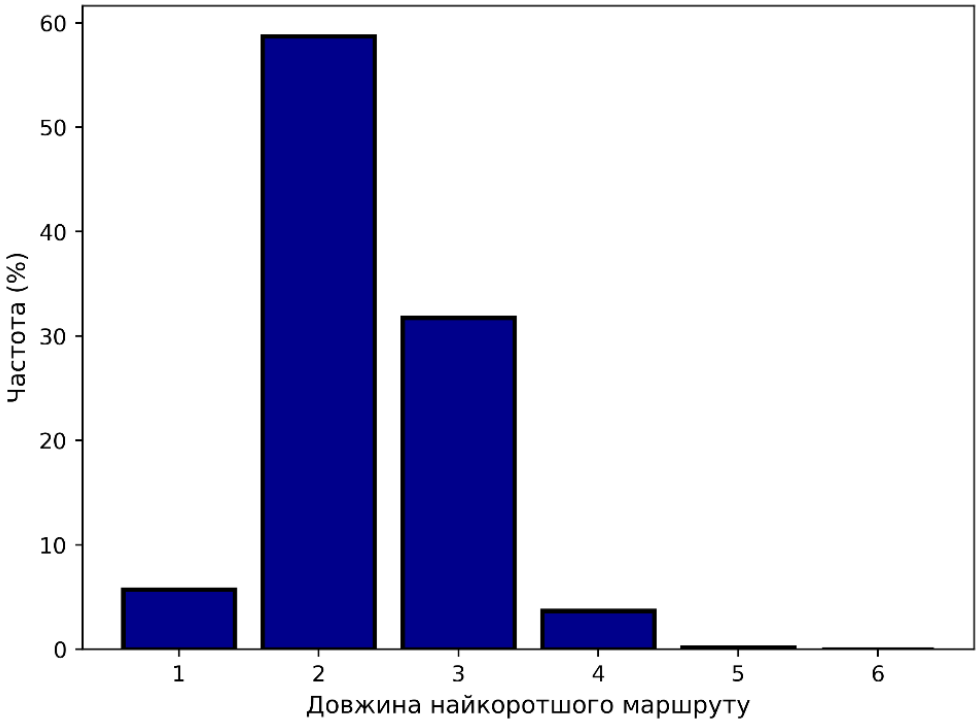


Рисунок 5.13 – Частота довжин найкоротших маршрутів

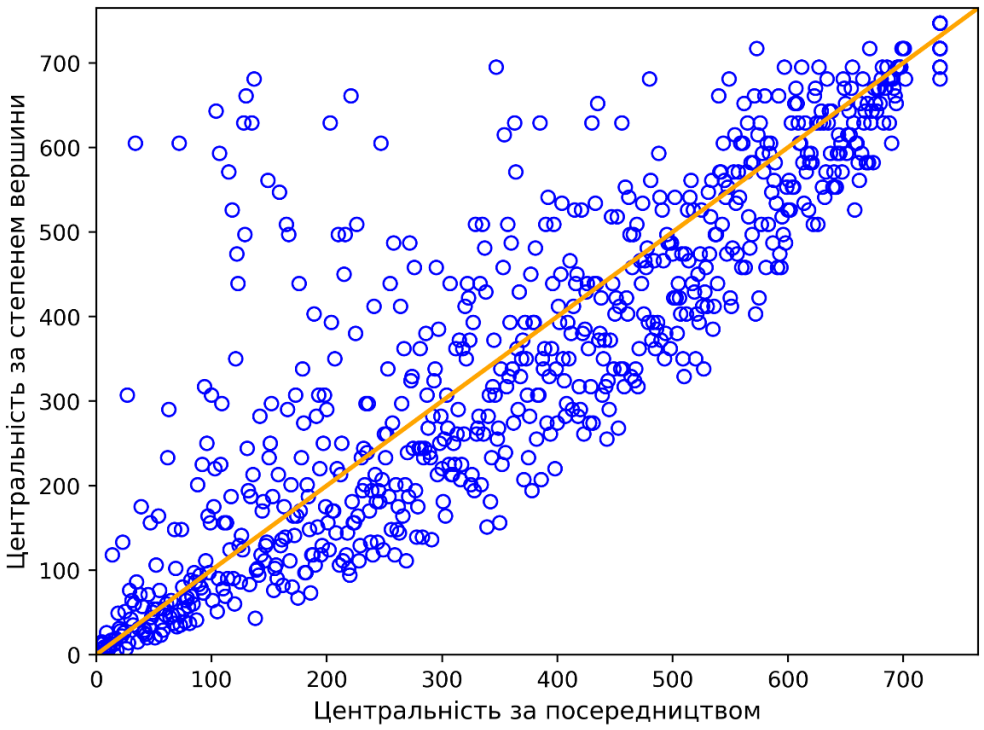


Рисунок 5.14 – Рангова кореляція між центральністю за посередництвом та центральністю за степенем вершини

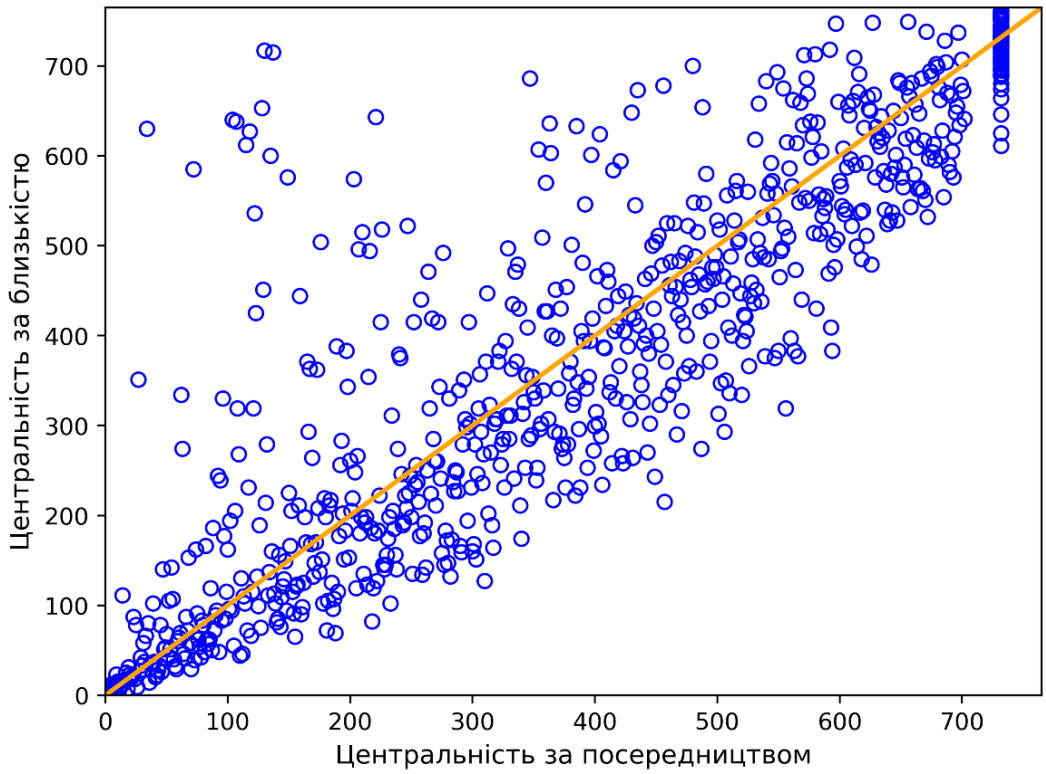


Рисунок 5.15 – Рангова кореляція між центральністю за посередництвом та центральністю за близькістю

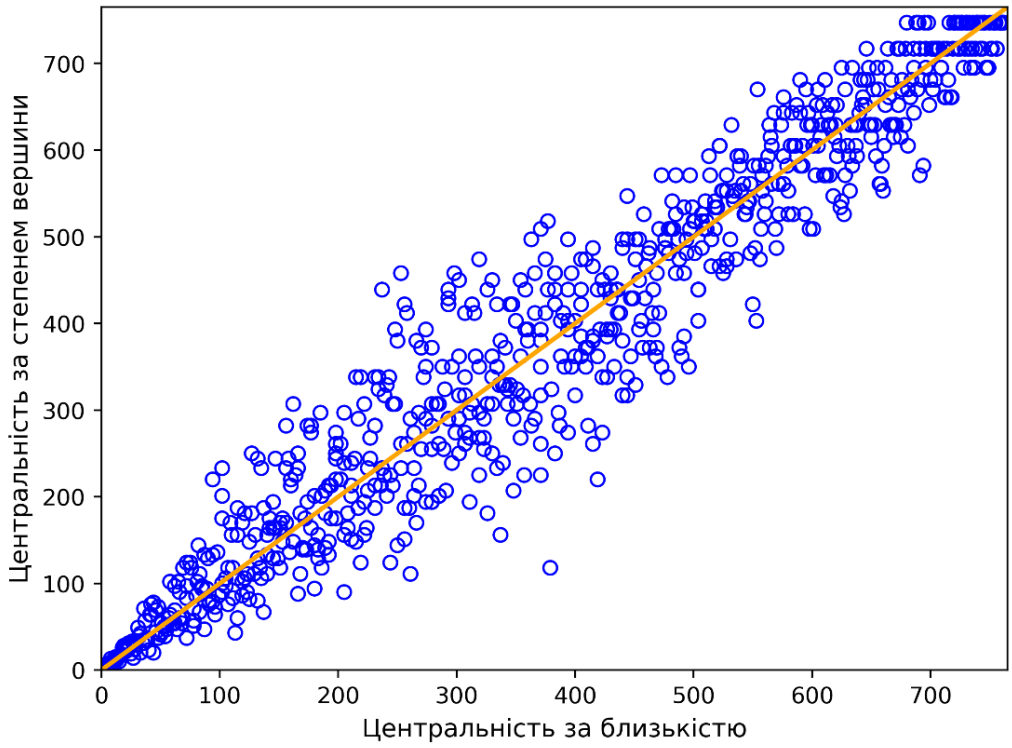


Рисунок 5.16 – Рангова кореляція між центральністю за близькістю та центральністю за степенем вершини

5.5 Питання для самоконтролю та професійного розвитку

1. У чому полягає головна ідея центральності за впливовістю?
2. Що таке Google PageRank і яка його основна мета?
3. Яку роль у Google PageRank відіграє коефіцієнт дисциплінованості вебмандрівника (параметр α)?
4. За скільки ітерацій обчислюється важливість кожної вебсторінки?
5. Чи можна розрахувати важливість кожної вебсторінки у неітераційний спосіб?
6. Навіщо рандомізують веб-граф перед застосуванням алгоритму Google PageRank?
7. Яким чином вирішується проблема термінальних вебсторінок?
8. Хто такий вебмандрівник і яка його роль у алгоритмі PageRank?
9. Що таке телепортація вебмандрівника і як її реалізувати на матриці перехідних ймовірностей?
10. Яким властивостям має відповідати примітивна матриця перехідних ймовірностей?

11. Чим характеризується нескорочувана матриця перехідних ймовірностей?

12. Що таке аперіодична матриця перехідних ймовірностей?

13. У яких випадках за власним вектором матриці перехідних ймовірностей не вдається правильно розрахувати характеристики стаціонарного режиму марковської системи?

14. Як інтерпретувати дуже високий або дуже низький рівень важливості вебсторінки, який розраховано за алгоритмом Google PageRank?

15. Як покращити ранг власної веб-сторінки без розміщення додаткових посилань на зовнішніх ресурсах?

16. Які реальні приклади застосування алгоритмів подібних до Google PageRank окрім пошукових систем?

17. Які переваги та недоліки ранжування за PageRank порівняно з іншими показниками центральності?

18. Що таке центральність у контексті теорії графів?

19. Які основні особливості типів центральності вершин графу?

20. Що таке центральність за степенем? Як вона розраховується?

21. Яке значення у соціальній мережі має вершина з найвищим степенем?

22. Як визначається центральність за близькістю?

23. Що означає «близькість» вершини до інших вершин у графі?

24. Як обчислюється центральність за посередництвом?

25. Як центральність за посередництвом пов’язана з передачею інформації або ресурсів у мережі?

26. Який показник центральності найбільше підходить для виявлення мостів на графі?

27. Як зміниться центральність вершин, якщо видалити одну важливу вершину графу?

28. Як результати аналізу центральності графу можуть допомогти зрозуміти структуру соціальної мережі?

29. Які особливості графової моделі соціальної мережі «Клуб карате Зехарі»?

30. Чи коректно ототожнювати вершини графу з високими показниками центальності з центрами кластерів?

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Lewis, H., & Zax, R. (2019). Essential Discrete Mathematics for Computer Science. USA: Princeton University Press.
2. Rosen, K. H. (2002). Discrete Mathematics and Its Applications (7th ed.). McGraw-Hill Higher Education.
3. Carter, J. L., & Wegman, M. N. (1979). Universal classes of hash functions. Journal of Computer and System Sciences, 18(2), 143–154. https://doi.org/10.1016/0022-0000(79)90044-8
4. Dietzfelbinger, M. (1996). Universal hashing and k-wise independent random variables via integer arithmetic without primes. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 1046, pp. 567–580). Springer Verlag. https://doi.org/10.1007/3-540-60922-9_46
5. West, D. B. (2000). Introduction to Graph Theory. Prentice Hall. ISBN: 0130144002
6. Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269–271. https://doi.org/10.1007/BF01386390
7. Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM (JACM), 34(3), 596–615. https://doi.org/10.1145/28869.28874.
8. Cherkassky, B. V., Goldberg, A. V., & Radzik, T. (1996). Shortest paths algorithms: Theory and experimental evaluation. Mathematical Programming, Series B, 73(2), 129–174. https://doi.org/10.1007/BF02592101
9. Gutin, G., & Punnen, A. (2003). The traveling salesman problem and its variations (Issue 1). Springer. https://doi.org/10.16309/j.cnki.issn.1007-1776.2003.03.004
10. Garey, M. R., & Johnson, D. S. (1990). Computers and Intractability; A Guide to the Theory of NP-Completeness. USA: W. H. Freeman & Co.
11. Croes, G. A. (1958). A Method for Solving Traveling-Salesman Problems. Operations Research, 6(6), 791–812. https://doi.org/10.1287/opre.6.6.791
12. Johnson, D. S., & McGeoch, L. A. (1997). The traveling salesman problem: A case study in local optimization. Local Search in Combinatorial Optimization, 215–310. Retrieved from http://142.103.6.5/~hutter/previous-earg/EmpAlgReadingGroup/TSP-JohMcg97.pdf
13. Іглін С. П. Теорія графів. Лекції та варіанти індивідуальних домашніх завдань : навчальний посібник. Харків : НТУ «ХПІ», 2017. 146 с.
14. Іглін С. П. Теорія ймовірностей та математична статистика на базі MATLAB : навч. посібник. Харків : НТУ «ХПІ», 2019. 470 с.

15. Dijkstra algorithm visualization. URL: https://visualgo.net/en/sssp (дата звернення 01.04.2025).

16. Dijkstra algorithm. URL: https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php (дата звернення 01.04.2025).

17. NetworkX – Network Analysis in Python. Python package documentation. URL: https://networkx.org/ (дата звернення 01.04.2025).

18. Discrete and Combinatorial Optimization Datasets. TSP datasets. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html (дата звернення 01.04.2025).

19. CS Academy Online Graph Editor. URL: https://csacademy.com/app/graph_editor/ (дата звернення 01.04.2025).

20. OpenStreetMap. URL: https://www.openstreetmap.org/#map=6/49.97/33.64 (дата звернення 01.04.2025).

21. OSMnx – download, model, analyze, and visualize street networks and other geospatial features from OpenStreetMap in Python. Python package documentation. URL: https://osmnx.readthedocs.io/en/stable/ (дата звернення 01.04.2025).

22. Boeing, G. 2024. Modeling and Analyzing Urban Networks and Amenities with OSMnx. Working paper. URL: https://geoffboeing.com/publications/osmnx-paper/ (дата звернення 01.04.2025).

23. Learn Depth-First Search(DFS) Algorithm From Scratch. URL: https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm (дата звернення 01.04.2025).

24. All You Need to Know About Breadth-First Search Algorithm. URL: https://www.simplilearn.com/tutorials/data-structure-tutorial/bfs-algorithm (дата звернення 01.04.2025).

25. Мазуренко В.В., Штовба С.Д. Огляд моделей аналізу соціальних мереж // Вісник Вінницького політехнічного інституту. – 2015. – №1. – С. 62-73.

26. Biggs N. L. Discrete Mathematics. – Oxford: Oxford University Press, 2nd eds., 2002. – 425 p.

27. Page, L., Brin, S., Motwani, R., & Winograd, T. (1998). The PageRank Citation Ranking: Bringing Order to the Web. World Wide Web Internet And Web Information Systems, 54(1999–66), 1–17. https://doi.org/10.1.1.31.1768

28. Langville, A. N., & Meyer, C. D. (2011). Google's PageRank and beyond: The science of search engine rankings. Google's PageRank and Beyond: The Science of Search Engine Rankings (pp. 1–224). Princeton University Press. https://doi.org/10.1063/1.2711640

29. Austin, D. (2013). How Google Finds Your Needle in the Web's Haystack. AMS Feature Column, 1–11. Retrieved from http://www.ams.org/samplings/feature-column/fcarc-pagerank

30. Peng, S., Zhou, Y., Cao, L., Yu, S., Niu, J., & Jia, W. (2018, March 15). Influence analysis in social networks: A survey. Journal of Network and Computer Applications. Academic Press. https://doi.org/10.1016/j.jnca.2018.01.005

31. Zachary, W. W. (1977). An Information Flow Model for Conflict and Fission in Small Groups. Journal of Anthropological Research, 33(4), 452–473. https://doi.org/10.1086/jar.33.4.3629752

32. Girvan, M., & Newman, M. E. J. (2002). Community structure in social and biological networks. Proceedings of the National Academy of Sciences of the United States of America, 99(12), 7821–7826. https://doi.org/10.1073/pnas.122653799

33. Rossi, R. A., & Ahmed, N. K. (2015). The Network Data Repository with interactive graph analytics and visualization. In Proceedings of the AAAI Conference on Artificial Intelligence. Retrieved from https://networkrepository.com

34. Borgatti, S. P., Everett, M. G., & Shirey, P. R. (1990). LS sets, lambda sets and other cohesive subsets. Social Networks, 12(4), 337–357. https://doi.org/10.1016/0378-8733(90)90014-Z

Електронне навчальне видання

Микола Володимирович Петричко
Сергій Дмитрович Штовба
Олексій Миколайович Козачко

ДИСКРЕТНА МАТЕМАТИКА ДЛЯ ПРОГРАМІСТІВ

Навчальний посібник

Рукопис оформив *М. Петричко*

Редактор *Т. Старічек*

Оригінал-макет виготовила *Т. Старічек*

Підписано до видання 16.01.2026 р.
Гарнітура Times New Roman.
Зам. Р2026-006
Видавець та виготовлювач
Вінницький національний технічний університет,
Редакційно-видавничий відділ.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
press.vntu.edu.ua;
Email: irvc.vntu@gmail.com
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.

Петричко Микола Володимирович

Доктор філософії з інформаційних систем та технологій, асистент кафедри комп'ютерних систем управління Вінницького національного технічного університету. В 2019 р. закінчив Вінницький національний технічний університет за спеціальністю "Автоматизація та комп'ютерно-інтегровані технології". В 2024 р. захистив дисертацію на здобуття ступеня доктора філософії зі спеціальності 126 "Інформаційні системи та технології". Автор 26 статей з наукометрії та інформаційних систем.

mpetrychko@vntu.edu.ua

Штовба Сергій Дмитрович

Професор, доктор технічних наук, професор кафедри інформаційних технологій Донецького національного університету імені Василя Стуса, професор кафедри комп'ютерних систем управління Вінницького національного технічного університету. В 1993 р. закінчив Вінницький політехнічний інститут за спеціальністю «Конструювання та технологія електронно-обчислювальних засобів. В 1997 р. захистив кандидатську дисертацію з математичного моделювання, а в 2009 р. – докторську дисертацію з інформаційних технологій. Автор 10 книг та біля 150 статей з моделювання, штучного інтелекту та інформаційних технологій.

s.shtovba@donnu.edu.ua
shtovba@vntu.edu.ua

Козачко Олексій Миколайович

Доцент, кандидат технічних наук, доцент кафедри системного аналізу та інформаційних технологій Вінницького національного технічного університету. В 2002 р. закінчив Вінницький національний технічний університет за спеціальністю "Комп'ютеризовані системи управління і автоматика". В 2006 р. захистив кандидатську дисертацію з математичного моделювання та обчислювальних методів. Автор 5 книг та понад 50 статей з інформаційних технологій та математичного моделювання.

kozachko@vntu.edu.ua