Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра математики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: **«ОСТОВНІ ДЕРЕВА ТА АЛГОРИТМИ ЇХ ПОБУДОВИ»**

Виконала: студентка 4-го року навчання
освітньої програми «Прикладна математика»,
спеціальності 113 Прикладна математика

Бабій Ангеліна Олександрівна

Керівник: Тимошкевич Л. М.
кандидат фіз.-мат. наук, ст. викладач

Рецензент:

Кваліфікаційна робота захищена
з оцінкою _____

Секретар ЕК_____
(підпис)

«_____» _____ 20___р.

Київ - 2024

Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра математики

ЗАТВЕРДЖУЮ
Зав.кафедри математики,
_____ *Чорней Р. К.*
(підпис)
“____”_____ 2023

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
для кваліфікаційної роботи
студентці 4-го курсу, факультету інформатики
Бабій Ангеліні Олександрівні

Тема: «Остовні дерева та алгоритми їх побудови»

Зміст кваліфікаційної роботи:

Анотація

1. Вступ

2. Огляд основних означень та тверджень, що пов’язані з теорією графів та остовних дерев.

3. Теорема Келі. Код Прюфера.

4. Розв’язання задач на застосування теореми Келі та коду Прюфера. Код Прюфера на мові програмування Python.

5. Огляд та розбір алгоритмів побудови остовних дерев.

6. Реалізація алгоритмів побудови остовних дерев на Python.

Висновки

Список літератури

Дата видачі “____”_____ 2023 Керівник _____
(підпис)

Завдання отримала _____
(підпис)

Графік підготовки кваліфікаційної роботи до захисту

Графік узгоджено «_____» _____ 2023р.

№ з/п	Перелік етапів	Термін виконання	Підпис
1.	Отримання теми роботи.	16.10.2023	
2.	Ознайомлення з темою кваліфікаційної роботи.	31.10.2023	
3.	Планування структури роботи.	20.11.2023	
4.	Робота з наданою науковою літературою, розбір теорії.	15.12.2024	
5.	Дослідження результатів отриманих в літературі.	15.02.2024	
6.	Робота над практичною частиною, початкове оформлення роботи.	30.04.2024	
7.	Попередній огляд кваліфікаційної роботи.	10.05.2024	
8.	Попередній захист кваліфікаційної роботи.	15.05.2024	
9.	Внесення остаточних правок.	26.05.2024	
10.	Захист кваліфікаційної роботи.	05.06.2024	

Науковий керівник _____
(ПІБ)

Виконавець кваліфікаційної роботи _____
(ПІБ)

Зміст

Анотація

Метою роботи було дослідження кількості остовних дерев повного графа K_n, огляд теореми Келі, коду Прюфера та їх застосування.
У роботі було розглянуто низку задач на застосування теореми Келі та коду Прюфера у поєднанні з комбінаторним підходом до поставлених задач.
Також при розгляді алгоритму коду Прюфера були створені програми на мові програмування Python для детальної візуалізації процесу кодування та відновлення дерев за допомогою коду Прюфера.

1 Вступ

1.1 Актуальність

Напевно, у сучасному світі не лишилося наук, що існують самостійно, не беручи участі у розвитку інших. Величезне надбання математичних напрацювань використовується ледве не в усіх напрямках в тому чи іншому вигляді: розділ статистики активно використовується соціологами, математичний аналіз – фізиками та економістами, геометрія – архітекторами та інженерами.

Отак і сучасна наука зацікавлена в дослідженнях пов'язаними з остовними деревами та алгоритмами їх побудови насамперед через різноманіття галузей застосування. Остовні дерева мають численні практичні застосування, очевидно, у сфері комп'ютерних наук, а саме, розробка ефективних маршрутизацій в комп'ютерних мережах та телекомунікаційних системах. Але окрім цього, дослідження остовних дерев має значний міждисциплінарний потенціал, з можливістю застосування в різних галузях, таких як фізика, хімія, біологія, економіка та соціальні науки.

Так остовні дерева можуть допомогти з вирішенням таких проблем, як планування оптимальних маршрутів для транспортування товарів та людей, оптимізацією логістичних мереж та ланцюгів постачання, моделювання структури біологічних мереж та аналіз соціальних зв'язків та поширення інформації в онлайн-спільнотах.

У цій роботі розглянемо безпосередньо визначення та необхідну для розуміння остовних дерев теорію, питання їх кількості для довільного графа, а також алгоритми побудови. Практичне дослідження містить задачі та реалізації у форматі коду Python для кращого розуміння.

1.2 Мета дослідження

1. Розглянути основні означення та теоретичні відомості з теорії графів та остовних дерев.

2. Опрацювати теорему Келі та код Прюфера — таким чином, дати відповідь на питання щодо кількості остовних дерев.

3. Створити реалізації алгоритмів для кодування та відновлення дерев за допомогою коду Прюфера на Python.

4. Оглянути та проаналізувати основні алгоритми побудови остовних дерев на зважених та незважених графах.

5. Створити реалізації алгоритмів Прима та Крускала для побудови остовних дерев на Python.

2 Основні означення та теореми

2.1 Основні означення з теорії графів

Почнемо із введення основних означень з теорії графів, що знадобляться при безпосередньому огляді теми.

Означення 2.1.0.
Графом $G = (V, E)$ називається така пара впорядкованих множин V і E, де V — множина вершин, що є непорожньою і скінченною, а E — множина ребер, що складається з невпорядкованих пар елементів множини вершин.

Означення 2.1.1.
Простим неорієнтованим графом G є пара множин (V, E), де E — довільна підмножина множини $V^{(2)}(E \subseteq V^{(2)})$; позначають $G = (V, E)$. Елементи множини V називають *вершинами графа* G, а множини E — його *ребрами*. Відповідно V — це множина вершин, а E —- множина ребер графа G.

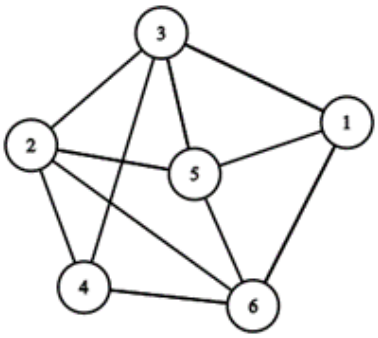


Рис. 2.1.1. Простий неорієнтоаний граф з 6 вершинами

Означення 2.1.2.
Вершину v і ребро e називають *інцидентними*, якщо v – кінець ребра e.

Означення 2.1.3.
Дві вершини графа називаються *сусідніми* або *суміжними*, якщо існує ребро, яке їх з'єднує. Два ребра називають *суміжними*, якщо вони мають спільну вершину.

Означення 2.1.4.

Циклом називається послідовність ребер $e_1, e_2, \ldots, e_k$, в якій всі ребра попарно різні, а для кожного $i = 2, \ldots, k-1$ один із кінців ребра e_i є кінцем ребра e_{i-1}, а інший – кінцем ребра e_{i+1}, і вільні кінці ребер e_k та e_1 також збігаються.

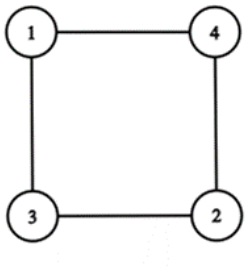


Рис. 2.1.4. Приклад циклу

Означення 2.1.5.

Якщо зв'язний граф має лише один цикл, то він називається *уніциклічним.*

Означення 2.1.6.

Цикл називається *простим*, якщо всі вершини у ньому попарно різні.

Означення 2.1.7.

Граф $G_1 = (V_1, E_1)$ називають *підграфом* графа $G = (V, E)$, якщо $V_1 \subseteq V$ та $E_1 \subseteq E$.

Означення 2.1.8.

Операція вилучення вершини v із графа $G = (V, E)$ полягає у вилученні з множини V елемента v, а з множини E — усіх ребер, інцидентних v.

Означення 2.1.9.

Операція вилучення ребра ee з графа $G = (V, E)$ — це вилучення елемента ee з множини E. Усі вершини зберігаються.

Означення 2.1.10.

Степенем $\delta(v)$ *вершини* v називають кількість інцидентних їй ребер. Вершину 0 степеня називають *ізольованою*, а вершину 1 степеня — *висячою.*

2.2 Дерева

Означення 2.2.1.

Дерево — це зв'язний ациклічний граф.

Означення 2.2.2.

Деяке дерево T графа G будемо називати *остовним*, за умови, якщо T є підграфом G та містить всі вершини графа G.

Означення 2.2.3.

Позначене дерево — це дерево, в якому кожній вершині присвоєне унікальне значення. Вершини позначеного дерева на n вершинах зазвичай позначаються першими n натуральними числами: $1, 2, \ldots n$.

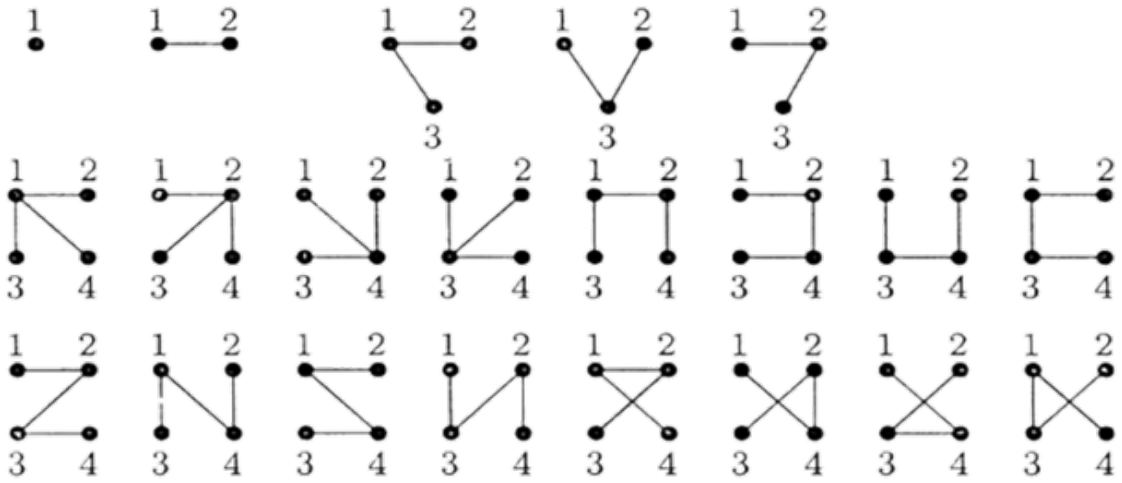


Рис. 2.2.3. Позначені дерева для випадків $n \leq 4$

Враховуючи означення 2.2.1.-2.2.3., наведемо граф:

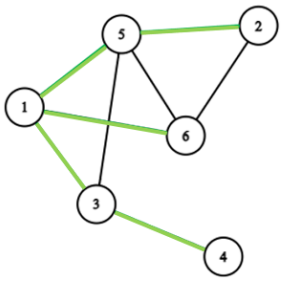


Рис. 2.2.1.-2.2.3. Приклад

На довільному графі G маємо підграф T, що водночас є деревом (зв'язний ациклічний граф), позначеним (кожна вершина містить унікальне значення) та остовним (містить всі вершини графа G) деревом.

Означення 2.2.4.
Кореневе дерево – дерево з відміченою вершиною-коренем.

Означення 2.2.5.
Листом називають висячу вершину дерева.

Означення 2.2.6.
Висота кореневого дерева — це відстань від кореня до найвіддаленішого листа.

Означення 2.2.7.
Гілка дерева T у вершині x — дерево з коренем x, породжене множиною всіх нащадків вершини x.

Означення 2.2.8.
Ліс — це неорієнтований граф без циклів, тобто множина дерев. Іншими словами, ліс — це граф, в якому будь-які два вузли з'єднані не більше ніж одним шляхом, і який не містить замкнених шляхів — циклів. Кожне з'єднане піддерево в лісі називається *деревом*.

Теорема 2.2.1.
Якщо G – дерево, то кількість ребер у ньому на 1 менше від числа вершин.

Теорема 2.2.2.
Будь-який зв'язний граф має остовне дерево.

Теорема 2.2.3.
У дереві будь-яка пара вершин з'єднана єдиним ланцюгом.

Теорема 2.2.4.

При додаванні до дерева будь-якого нового ребра утворюється цикл.

Теорема 2.2.5.

При видаленні з дерева будь-якого ребра він перетворюється на незв'язний граф.

2.3 Додаткові означення

Додамо кілька означень, що не стосуються напряму теорії графів, проте будуть необхідні для роботи з остовними деревами.

Означення 2.3.1.
Відображення $f : X \to Y$ називається *бієкцією(взаємно однозначною відповідністю)*, якщо для кожного елемента y з множини Y існує єдиний елемент x з множини X такий, що $f(x) = y$.

Означення 2.3.2.
Відображення $f : X \to Y$ є *сюр'єктивним*, якщо для кожного елемента y з множини Y, існує щонайменше один елемент x з множини X такий, що $f(x) = y$.

Означення 2.3.3.
Відображення $f : X \to Y$ є *ін'єктивним*, якщо для кожного елемента y з множини Y існує не більш як один елемент x з множини X такий, що $f(x) = y$. Інакше кажучи, f є ін'єктивним, якщо з рівності $f(x) = f(x')$ для x та x' з X випливає рівність $x = x'$.

3 Код Прюфера. Теорема Келі.

3.1 Код Прюфера.

Означення 3.1.1.

Код Прюфера для дерева – це послідовність $x(T) = (x_1, x_2, \ldots, x_n)$, елементами якої є вершини дерева.

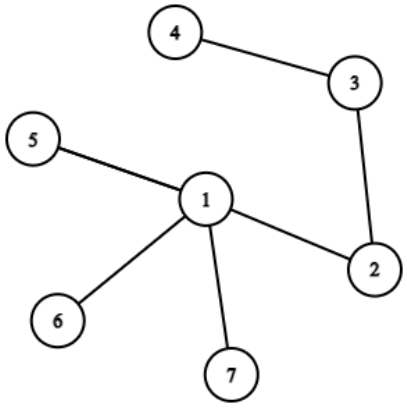
Алгоритм побудови коду Прюфера:

Нехай T є дерево з вершинами, пронумерованими числами $\{1, 2, \ldots, n\}$. Побудова послідовності Прюфера для дерева T ведеться шляхом послідовного видалення вершин з дерева, поки не залишаться тільки дві вершини.

1. Вибирається висяча вершина k з найменшим номером.
2. В послідовність записується номер вершини m, з якою з'єднана вершина k.
3. Вершина k видаляється.
4. Якщо лишається лише 2 вершини, отримано код Прюфера; якщо більше – повторюються кроки 1-3 на вершинах, що лишилися.

В результаті отримується послідовність $(p_1, \ldots, p_{n-2})$, складена з чисел $\{1, 2, \ldots, n\}$.

Розглянемо приклад:



Довільний граф, який ми спробуємо закодувати

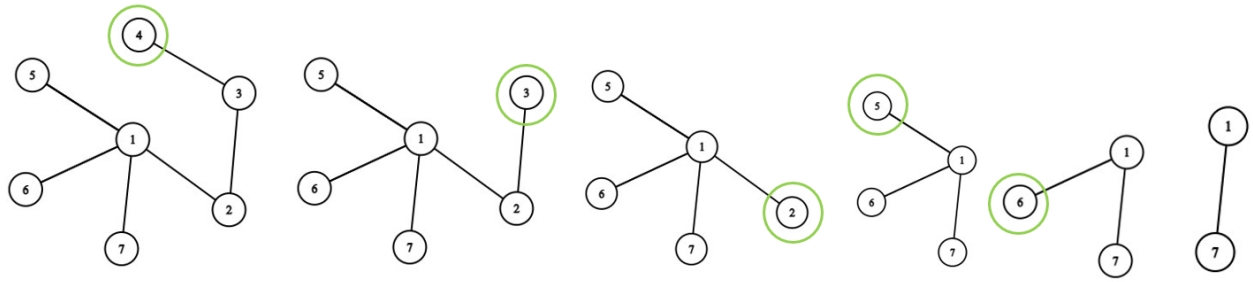


Рис. 3.1.1. Алгоритм коду Прюфера на графі

Оберемо висячу вершину з найменшим номером — 4. Видаляємо її і записуємо номер суміжної — 3. Для наступного кроку висячою вершиною з найменшим номером буде 3, тоді видалимо її і запишемо 2. Повторивши ці кроки до тих пір, поки не залишиться 2 вершини, отримаємо код Прюфера.
Запишемо його: 3, 2, 1, 1, 1

З алгоритму побудови коду Прюфера випливає, що вершина дерева степеня d зустрічається в коді Прюфера рівно $d-1$ раз (у зазначеному прикладі: 4 – степінь вершини 1, а в коді Прюфера 1 зустрічається 3 рази). З цього випливає твердження про число позначених дерев з n вершинами – теорема Келлі.

3.2 Декод коду Прюфера.

Насправді зворотня операція – побудова дерева з послідовності Прюфера $p = (p_1, \ldots, p_{n-2})$ – також можлива. Розглянемо відповідний алгоритм та приклад.

1. Будується список вершин графа $s = (1, \ldots, n)$.
2. Вибирається перший номер i_1, який не зустрічається в послідовності p.
3. Додається ребро (i_1, p_1).
4. Видаляється i_1 з s і p_1 з p.
5. Процес повторюється до того моменту, коли послідовність p стає порожньою.

У цей момент список s містить рівно два числа i_{n-1} і n. Залишається додати ребро (i_{n-1}, n), і дерево побудовано.

Відновимо дерево за отриманим кодом у 3.1.
Отриманий код Прюфера: 3, 2, 1, 1, 1. Залишкове ребро: (1, 7).
Отриманий список має довжину 5, отже список вершин для відновлення дерева: 1, 2, 3, 4, 5, 6, 7.

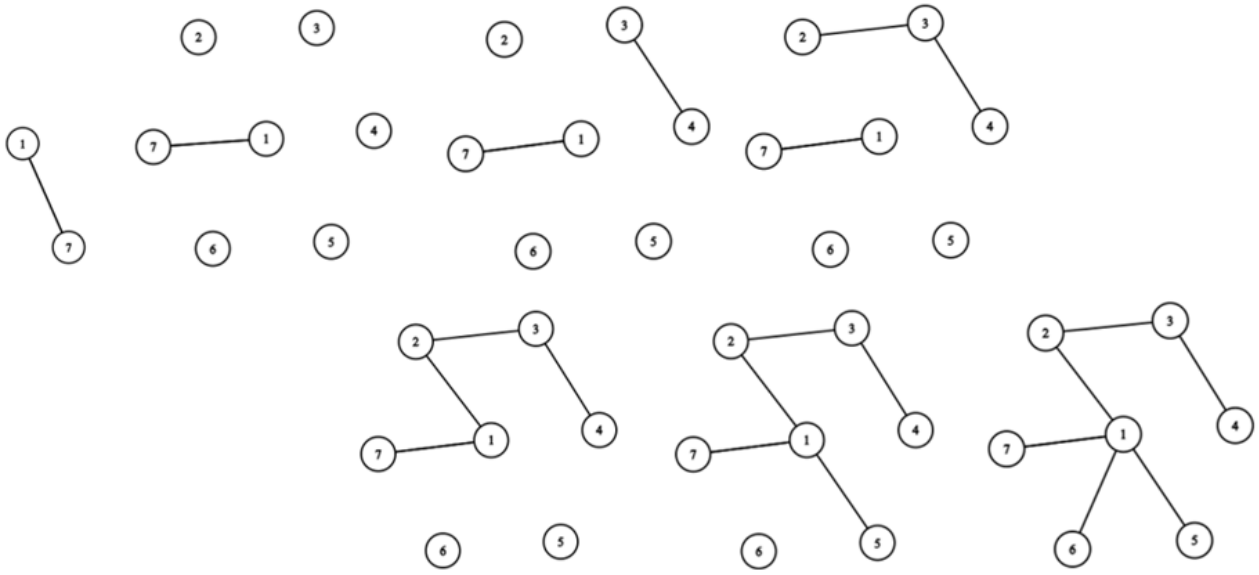


Рис. 3.2.1. Відновлення дерева за кодом Прюфера

Позначимо залишкове ребро та всі вершини для відновлення графа. Оберемо вершину, якої нема в коді Прюфера, з найменшим номером — 4, та поєднає-

мо її з вершиною, яка є першою в коді Прюфера — 3. Маємо утворене ребро (3,4), при цьому, видаляємо відповідно 3 з послідовності коду Прюфера та 4 з наявних вершин.

Наступна вершина з найменшим номером, якої нема в коді Прюфера — 3 (оскільки ми видалили її з послідовності). З'єднаємо її з першою вершиною в оновленому коді — 2. Отримали ребро (2,3), а вершини 2 і 3 видаляємо з відповідних послідовностей. Повторюємо операцію до спустошення обох послідовностей — повного відновлення графу.

3.3 Теорема Келі.

Теорема 3.2.1.
Теорема Келі: Число позначених дерев з n вершинами дорівнює n^{n-2}.

У контексті остовних дерев, можемо говорити про те, що кількість остовних дерев повного графа K_n визначатиметься числом n^{n-2}.

Доведень теореми Келі існує кілька: серед них доведення через рекурсію або ж методом подвійного рахунку. Але, напевно, найбільш вдалим та елегантним буде доведення через проведення взаємнооднозначної відповідності: кожне остовне дерево може бути закодоване за допомогою коду Прюфера, який є послідовністю довжиною $n-2$, складеною з n чисел.

$\triangle$ **Доведення:** Запишемо вершини графа (K_n) як $\{1, 2, \ldots, n\}$. Побудуємо взаємнооднозначне відображення між остовними деревами K_n і послідовностями довжини $n-2$, де кожний елемент є натуральним числом від 1 до n. Для доведення теореми цього достатньо, адже при підрахунку за комібнаторним правилом добутку, таких послідовностей матимемо n^{n-2}.
Припустимо, що T – дерево на вершинах $\{1, 2, \ldots, n\}$. Поставимо цьому дереву послідовність $p_1, p_2, \ldots, p_{n-2}$ коду Прюфера цього дерева відповідно.

Для доведення в обидві сторони, побудуємо зворотну відповідність. Існує послідовність $p_1, p_2, \ldots, p_{n-2}$ з елементами з $\{1, 2, \ldots, n\}$. Враховуючи, що кожна вершина p степеня d зустрічається у послідовності дерева T рівно $d\breve{}1$ раз (випливає з побудови кода Прюфера), можемо сказати, що вершини, яких нема в послідовності, є висячими.
Почнемо з вибору вершини L_1 з найменшим номером і сполучимо її ребром з p_1, після чого видалимо L_1 з послідовності номерів: $V_1 = V \setminus \{L_1\}$. Тепер виберемо вершину $L_2 \in V_1$ з найменшим номером, яка не зустрічається у послідовності $p_2, \ldots, p_{n-2}$, з'єднаємо L_2 з p_2 і покладемо $V_2 = V_1 \setminus \{L_2\}$.
Повторимо таку операцію $n-2$ рази. Ми використаємо всю послідовність і проведемо $n-2$ ребра. Таким чином залишиться безліч V_{n-2} з двох вершин і одне непроведене ребро дерева T.

Дві вершини з V_{n-2} потрібно сполучити ребром: їхні степені в даному графі дорівнюють кількості входжень цих вершин у послідовність $p_1, p_2, \ldots, p_{n-2}$, тобто, на 1 менше, ніж їхній степінь в дереві T. $\triangle$

Підсумовуючи, можемо говорити про те, що:

- Кожний код Прюфера є унікальним і визначає одне дерево. Оскільки код Прюфера для дерева з n вершинами має довжину $n-2$ і кожний елемент може бути будь-якою з n вершин, то кількість можливих кодів Прюфера дорівнює n^{n-2}.

- Оскільки кожний код Прюфера однозначно визначає дерево, то кількість різних дерев з n вершинами також дорівнює n^{n-2}.

4 Задачі та практична частина Python.

4.1 Задача 1

Чи може зв'язний граф мати рівно два остовних дерева?

У випадку, коли деякий граф є деревом, для нього існуватиме лише 1 остовне дерево. Інакше, кількість остовних дерев визначатиметься довжиною простого циклу, оскільки при видаленні якогось 1 ребра з циклу, граф залишатиметься зв'язним, в той час, як видалення ребра, що не є частиною простого циклу, призведе до утворення лісу. Мінімальна довжина циклу — 3. Отже, кількість остовних дерев може дорівнювати 1, 3, 4, 5,. . . , але не 2.

4.2 Задача 2

Чому дорівнює кількість позначених дерев з $n > m$ вершинами, де степінь вершини 1 дорівнює m?

C_{n-2}^{m-1} способів позицій для номера вершини 1, оскільки довжина послідовності коду Прюфера на 2 менше за кількість вершин, а степінь вершини на 1 більша за кількість входжень цієї вершини в код Прюфера.
Залишається $n-m-1$ позицій в коді Прюфера для номеру будь-якої вершини, окрім 1.
Тоді враховуючи всі вершини, крім 1, маємо $(n-1)^{n-m-1}$ заповнень цих позицій.
Звідси маємо $C_{n-2}^{m-1} \cdot (n-1)^{n-m-1}$ позначених дерев.

4.3 Задача 3

Нехай n та k — натуральні числа $2 \leq k \leq n$. Скільки існує остовних дерев на $[n]$, у яких перша і друга вершина з'єднані шляхом довжини $k-1$.

Забираючи 1 та 2 вершину, матимемо C_{n-2}^{k-2} варіантів обрати $k-2$ вершини серед $n-2$.

Після того, як ми вибрали $k-2$ вершини, потрібно врахувати те, що вони можуть бути в різних послідовностях, формуючи шлях. Кількість перестановок — $(k-2)!$.
Таким чином маємо $C_{n-2}^{k-2} \cdot (k-2)! = \frac{(n-2)!}{(n-k)!}$ шляхів довжини $k-1$
Оскільки кожний код Прюфера визначає позначене дерево, знайдемо кількість кодів Прюфера для графів, що міститимуть потрібний нам шлях, враховуючи вже, що ми маємо $n-k$ вершин для цього (не враховуємо k вершин, що є в шляху): kn^{n-k-1} За правилом множення отримаємо:

$$kn^{n-k-1} \cdot \frac{(n-2)!}{(n-k)!}$$

4.4 Задача 4

Скільки існує кореневих позначених дерев з n вершинами, де вершина n є листом і не є коренем?

Відкинемо вершину, що є листом і не є коренем. Тепер маємо позначене дерево з $n-1$ вершинами, за формулою Келі таких дерев матимемо $(n-1)^{(n-1)-2}$.
Після цього, повернемо вершину, яку ми заюрали на початку: до кожної з цих вершин $(n-1)$ ми можемо додати раніше прибраний лист n та кожна з них $(n-1)$ може бути коренем. Враховуючи це, таких дерев матимемо:

$$(n-1)^{n-3} \cdot (n-1) \cdot (n-1) = (n-1)^{n-1}$$

4.5 Задача 5

Скільки існує кореневих позначених дерев з n вершинами, де вершини $n, n-1, \ldots n-k+1$ є листами та не є коренями?

Задача 5 є узагальненням задачі 4.
Маємо k листів, що не є коренями. Як і за логікою задачі 4, приберемо їх.
Тоді число позначених дерев з $n-k$ вершинами є $(n-k)^{(n-k)-2}$ за теоремою Келі.
Врахуємо, що будь-яка вершина може бути коренем — $(n-k)$.

Та, повертаючись до графа з усіма вершинами, додамо листи, які раніше прибрали для обчислення позначених дерев — $(n-k)^k$.
Отримаємо: $(n-k)^{(n-k)-2}(n-k)(n-k)^k = (n-k)^{n-1}$

4.6 Задача 6

Доведіть, що при $2 \leq k \leq n-1$ виконується рекурентне співвідношення

$$\mathcal{F}_n^{k-1} = n\mathcal{F}_n^k$$

Кореневе дерево будемо обходити за напрямком від кореня назовні.
Щоб довести співвідношення, побудуємо відображення між множинами $\mathcal{F}_n^{k-1}$ та $\mathcal{F}_n^k$.
Можемо взяти ліс, що містить $k-1$ дерево, знайдемо в ньому вершину k, заберемо її. Всі ребра та вершини, які сполучалися з графом через цю вершину, винесемо як окреме кореневе дерево.
У всіх випадках, крім одного, вийде дерево з $\mathcal{F}_n^k$.
Винятком є випадок, коли гілка забрана від першого дерева і при цьому вона містить вершину n. Тоді поміняємо мітки 1 і k місцями.
Дослідимо кількість прообразів, що має довільний ліс з $\mathcal{F}_n^k$.
Щоб побудувати прообраз лісу, візьмемо в ньому k-е дерево і доєднаємо до якоїсь з вершин першого, другого, ..., $(k-1)$-го дерева(у всіх тих випадках, де не було корекції).
У прообразах, де відбулась корекція, доєднаємо перше дерево до будь-якої вершини k-го дерева і при цьому поміняємо мітки 1 і k.
Таким чином, з будь-якої з n вершин можна побудувати прообраз.
Тобто $\mathcal{F}_n^{k-1} = n\mathcal{F}_n^k$.

4.7 Задача 7

Доведіть, що кількість лісів, з вершинами в множині $[n]$, які складаються з k кореневих дерев, дорівнює $C_{n-1}^{k-1}n^{n-k}$.

Маємо лише один ліс з $n-1$ деревами.

Рівність з задачі 6 $\mathcal{F}_n^{k-1} = n\mathcal{F}_n^k$ застосуємо кілька разів. Отримаємо $\mathcal{F}_n^k = n^{n-k-1}$.

Кількість лісів, де вершина n є в будь-якому дереві, окрім першого, дорівнює $\mathcal{F}_n^k$.

Отже, кількість лісів без обмежень на вершину n: kn^{n-k-1}.

Кількість виборів коренів для k дерев з n вершин: C_n^k.

Тоді матимемо кількість лісів: $C_n^k kn^{n-k-1} = C_{n-1}^{k-1} n^{n-k}$.

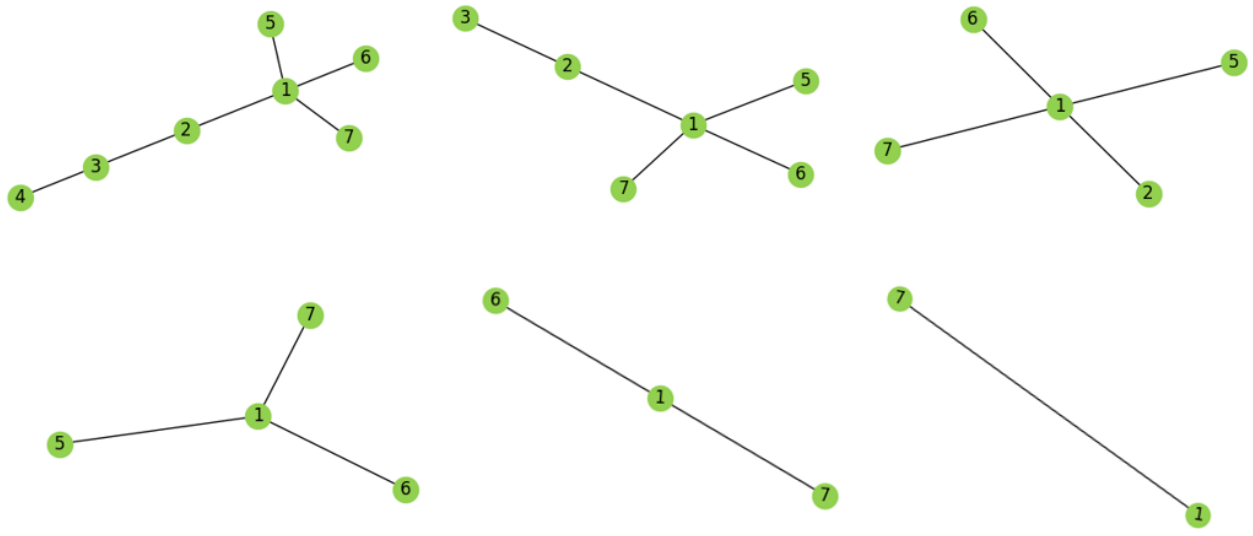
4.8 Кодування та відновлення дерев кодом Прюфера на Python

Алгоритм коду Прюфера — як прямий, так і обернений декод — є нетиповим та цікавим, тому одним з завдань цієї роботи стало написання практичної зрозумілої візуалізації для обох операцій кодування та відновлення.

Кодування дерев кодом Прюфера на Python

```
def prufer_code(graph):
    code = []
    G = graph.copy()
    plot_graph(G)
    while len(G.nodes) > 2:
        leaves = sorted(node for node in G.nodes if G.degree(node) == 1)
        leaf = leaves[0]
        neighbor = next(G.neighbors(leaf))
        code.append(neighbor)
        G.remove_node(leaf)
        plot_graph(G)
    return code
```

Вивід: візуалізація покрокового видалення кожної вершини.



Та відповідно кінцевий код Прюфера для графа, заданого на початку.

```
Код Прюфера для даного графа: [3, 2, 1, 1, 1]
```

Відновлення дерев за наявним кодом Прюфера на Python

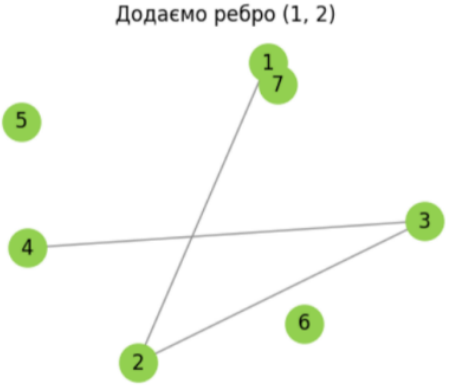
```
def prufer_to_tree(prufer_code, initial_edge):
    vertices = len(prufer_code) + 2
    vertex_set = set(range(1, vertices + 1))

    G = nx.Graph()
    G.add_nodes_from(vertex_set)
    G.add_edge(*initial_edge)

    degree = {i: 1 for i in vertex_set}
    for node in prufer_code:
        degree[node] += 1

    for node in prufer_code:
        for v in vertex_set:
            if degree[v] == 1:
                G.add_edge(node, v)
                degree[node] -= 1
                degree[v] -= 1
                vertex_set.remove(v)
                break

    u, v = vertex_set
    G.add_edge(u, v)

    return G
```

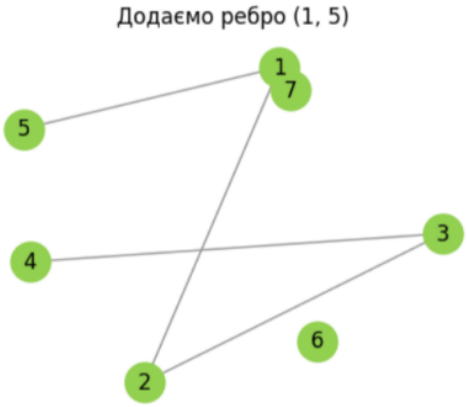
Вивід: візуалізація поступового відновлення дерево з кожним кроком декодування коду Прюфера.

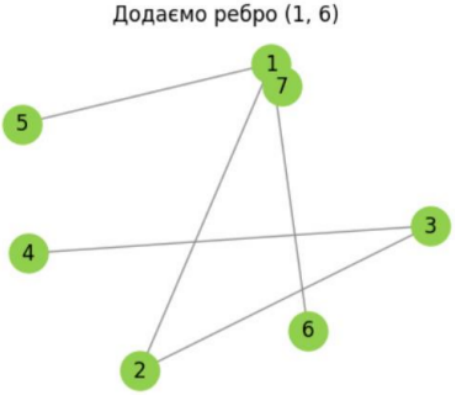

Початкове ребро
1
7
5
3
4
6
2

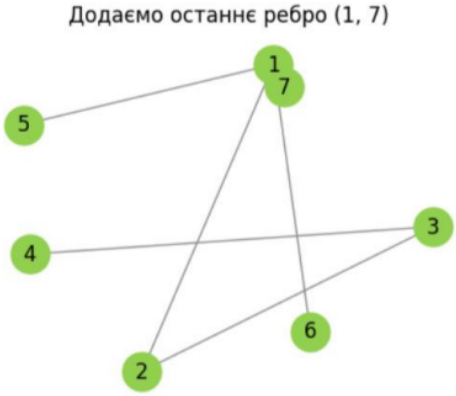

Додаємо ребро (3, 4)
1
7
5
3
4
6
2


Додаємо ребро (2, 3)
1
7
5
3
4
6
2


Додаємо ребро (1, 2)
1
7
5
3
4
6
2


Додаємо ребро (1, 5)
1
7
5
3
4
6
2


Додаємо ребро (1, 6)
1
7
5
3
4
6
2


Додаємо останнє ребро (1, 7)
1
7
5
3
4
6
2

5 Алгоритми побудови остовних дерев

Говорячи про остовні дерева, можна розглядати не тільки питання їхньої кількості для деякого графа, а й безпосередньо процес побудови. Оглянемо декілька основних алгоритмів, а також їхні властивості, переваги та недоліки.

5.1 Пошук в ширину та вглиб

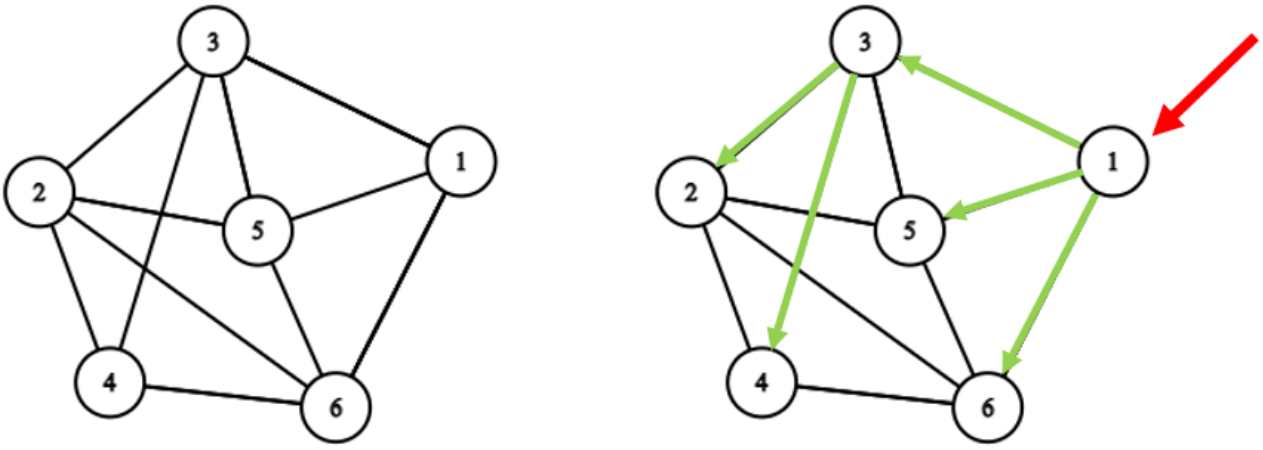
Алгоритми пошуку в ширину та вглиб є доволі популярними алгоритмами пошуку на графах та, в тому числі, часто використовуються для побудови остовних дерев.

Пошук в ширину

Кроки BFS-алгоритму (Breadth-First-Search – пошук в ширину)

1. Починаємо з довільної вершини v. Включаємо вершину v у чергу.
2. Розглянути вершину, яка перебуває на початку черги; деяка вершина k. Якщо для всіх вершин, суміжних із вершиною k, уже визначено BFS-номери, то перейти до кроку 4, інакше - до кроку 3.
3. Нехай $\{k, n\}$ - ребро, у якому номер BFS(n) не визначено. Позначити це ребро потовщеною суцільною лінією, визначити BFS(n) як черговий BFS-номер, включити вершину у чергу й перейти до кроку 2.
4. Виключити вершину k із черги. Якщо черга пуста, то зупиняємось.

Розглянемо приклад BFS-алгоритму на графі 2.1.1.



Обхід починаємо з вершини 1 та обходимо всі суміжні – 3, 5, 6. Коли вони пройдені, спускаємось на вершину 3 – проходимо всі суміжні з нею – 2, 4. Таким чином, вже на даний момент маємо всі вершини початкового графу задіяними в дереві, а кількість ребер $5 = 6 - 1 = n - 1$. Тож мінімальне остовне дерево знайдене.

Пошук в глибину

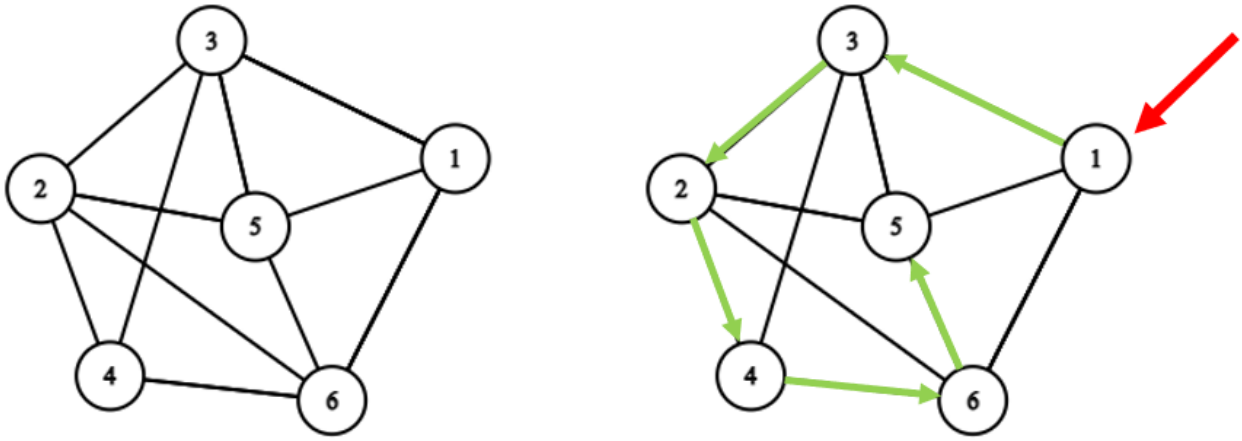
1. Починаємо з довільної вершини v. Включаємо цю вершину в чергу.

2. Розглянемо вершину у верхівці стеку: нехай це вершина x. Якщо всі ребра, інцидентні вершині x, позначено, то перейти до кроку 4, інакше — до кроку 3.

3. Нехай $\{x, y\}$ — непозначене ребро. Якщо DFS(y) уже визначено, то позначити ребро $\{x, y\}$ штриховою лінією та перейти до кроку 2. Якщо DFS(y) не визначено, то позначити ребро $\{x, y\}$ потовщеною суцільною лінією, визначити DFS(y) як черговий DFS-номер, включити цю вершину в стек і перейти до кроку 2.

4. Вилучаємо вершину x зі стеку. Якщо стек порожній, то зупиняємось, інакше — переходимо до кроку 2 для повторення.

Розглянемо приклад DFS-алгоритму також на графі 2.1.1.



Обхід починаємо з вершини 1 та «спускаємося» вглиб до 3. Таким чином потрібно спускатися допоки не дійдемо до висячої вершини, або ж в нашому випадку, допоки не обійдемо всі вершини. Якщо відбувається спуск у висячу вершину, а також є незадіяні вершини, ми повертаємось на вершину, з якої починали, та спускаємось вниз аналогічно. Таким чином, маємо всі вершини початкового графу задіяними в дереві, а кількість ребер $5 = 6 - 1 = n - 1$. Тож мінімальне остовне дерево знайдене.

Аналіз BFS та DFS алгоритмів для знаходження остовних дерев

При використанні алгоритмів BFS та DFS для пошуку мінімальних остовних дерев, перед нами постають як і переваги цих методів, так і недоліки. Перевагою насамперед є простота та зрозумілість алгоритмів пошуку в ширину та в глибину. Проте при застосуванні цих теоретичних знань на практиці (наприклад, питання логістики та перевезень), стає очевидним дуже суттєвий недолік: велика відстань між вершинами, особливо у випадку пошуком вглиб, оскільки тоді отримані дерева будуть "довгими"та "глибокими".
Отже, потрібний алгоритм, що дозволить знаходити мінімальне остовне дерево з меншими відстанями між вершинами, ніж це роблять алгоритми DFS та BFS.

5.2 Постановка проблеми. Жадібні алгоритми.

У попередньому пункті було згадано про непрактичність алгоритмів пошуку вширину та вглиб на реальних проблемах та задачах. Часто, говорячи про практичні питання логістики чи прокладення шляхів, оптимізацій маршруту, можна говорити й про "вартість"переправлення між пунктами. Перекладаючи цю проблему на математичну мову, наша задача зводиться до знаходження найменшого остовного дерева з врахуванням ваг на ребрах графа. Для цього введемо означення зваженого графа.

Означення 5.2.1.

Зважений граф — це граф, у якому кожне ребро асоційоване з певною числовою вагою.

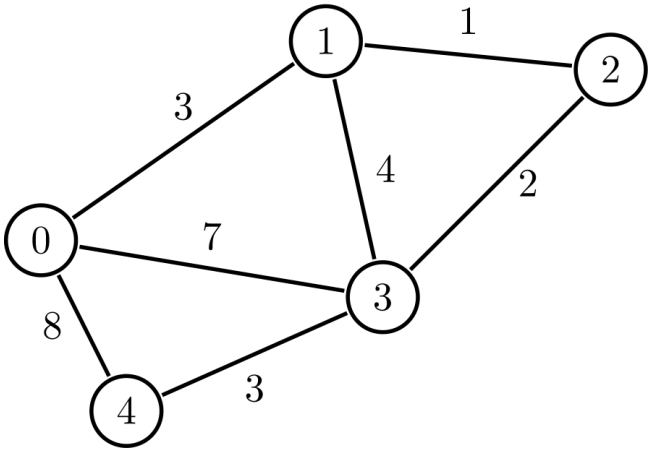


Рис. 5.2.1. Зважений граф

Покладемо, що кожний пункт, що потрібно відвідати під час перевезень — вершини графа, ребра — шляхи, що сполучають пункти, а відповідно ваги на ребрах — "вартість"перевезення. Метою поставимо оптимізацією планування шляху таким чином, щоби відвідати всі пункти, проте вартість подорожі була мінімальною. Таким чином, зводимо задачу до знаходження мінімального остовного дерева на деякому графі.

Означення 5.2.1.

Мінімальне остовне дерево — це підграф зв'язного, зваженого графа, що містить всі його вершини та має мінімальну сумарну вагу ребр.

Зважаючи на опитмізаційну умову про мінімальну суму вагів ребр, у вирішенні поставленої проблеми варто звертати увагу на жадібні алгоритми — алгоритми, що при виборі наступного ребра для включення в мінімальне остовне дерево завжди обирають ребро з найменшою вагою, яке не утворює циклу з вже включеними ребрами.

5.3 Алгоритм Прима

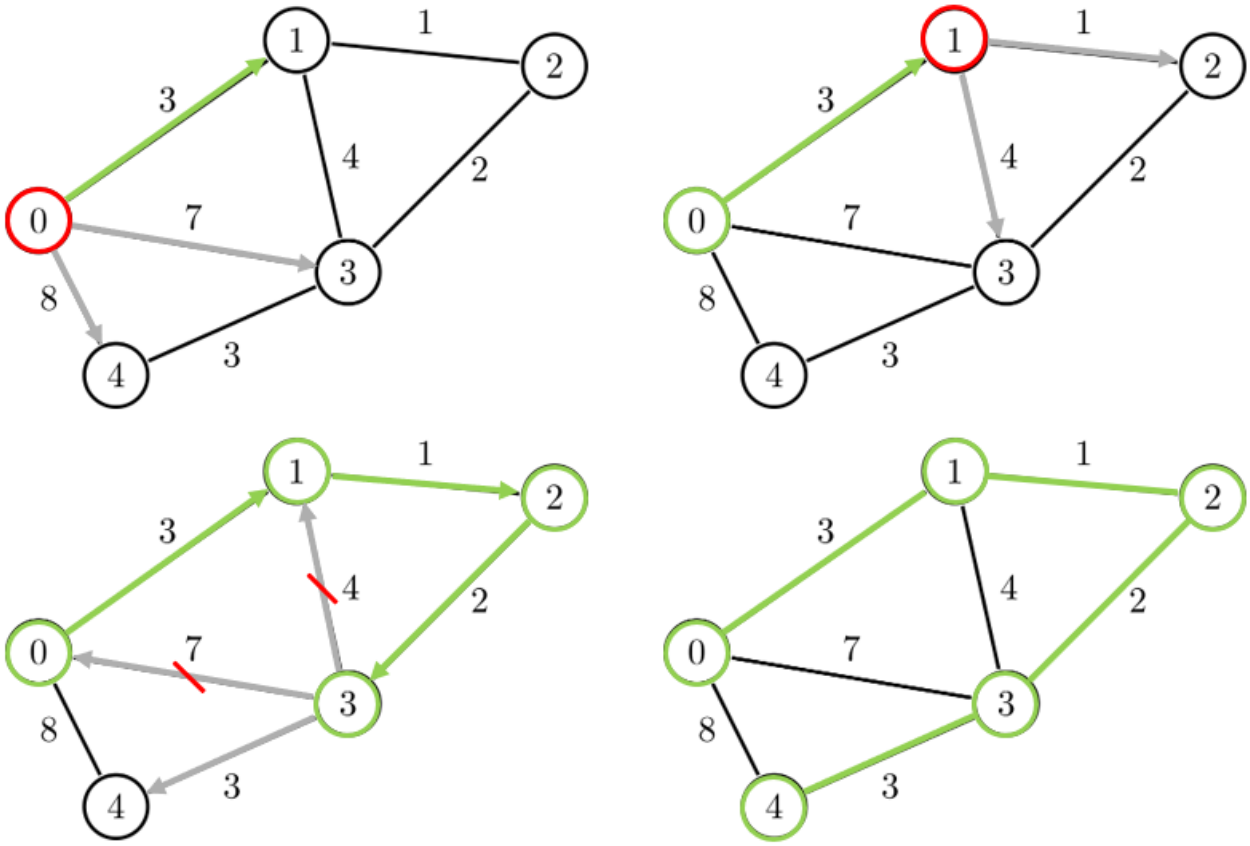
Алгоритм Прима будує мінімальне остовне дерево (MST), починаючи з однієї вершини і поступово додаючи до дерева найменше ребро, яке з'єднує вже включену вершину з будь-якою іншою вершиною, що ще не включена в дерево.

1. Виберемо будь-яку вершину як початкову.
2. Знайдемо ребро з найменшою вагою, яке з'єднує включену вершину з невключеною.
3. Додаймо це ребро і відповідну вершину до MST.
4. Повторимо кроки 2-3, доки всі вершини не будуть включені в MST.

Алгоритм Прима для пошуку мінімального шляху

Маємо початкову точку відправлення вантажівки — 0, та 6 складів, де має побувати водій та забрати вантаж. Значення між складами — відповідна відстань в умовних одиницях. Оптимізаційна проблема в такому випадку — побудова найкоротшого шляху, без циклів (нема сенсу приїжджати в один і той же склад двічі), з найменшою витратою часу та бензину — тобто через найкоротші можливі шляхи. Побудуємо маршрут, використовуючи алгоритм Прима.

Точка відправлення — 0, отже обираємо її за початкову вершину. Можливі варіанти проходу (в дужках — ваги): до вершин 1(3), 3(7), 4(8). Мінімальним є шлях до складу 1, тож відправляємось туди. Зі складу 1 можемо вирушити або до складу-вершини 2(1), або 3(4). Обираємо мінімальний шлях до вершини 2. З вершини 2 маємо лише шлях до вершини 3, тож обираємо його. З складу-вершини 3 маємо 3 шляхи, проте 2 з них утворюватимуть цикл і не є найбільш оптимальними, тож, фактично, шлях один — до вершини 4(3). Таким чином, нам вдалося прокласти шлях з відправочного пункту до усіх складів з мінімальною витратою часу та бензину.



Оцінка алгоритму Прима

Часова оцінка алгоритму Прима залежить від його реалізації.
Говорячи про найпростішу реалізацію звичайною чергою, можемо говорити про складність $O(V^2)$:

- Ініціалізація черги займатиме $O(V)$, V — кількість вершин.
- Знаходження ребра з мінімальною вагою — $O(V^2)$, оскільки ми повторюємо операцію для кожної з V вершин, де будь-яка з цих вершин в найгіршому випадку може мати $V - 1$ ребер.
- Таким чином, складаючи загальну часову складність, отримаємо: $O(V^2 + VE) = O(V^2)$, $E \leq V - 1$.

Для покращення оцінки і зменшення часової складності алгоритму Прима, можемо реалізувати його на основі бінарної купи. Тоді:

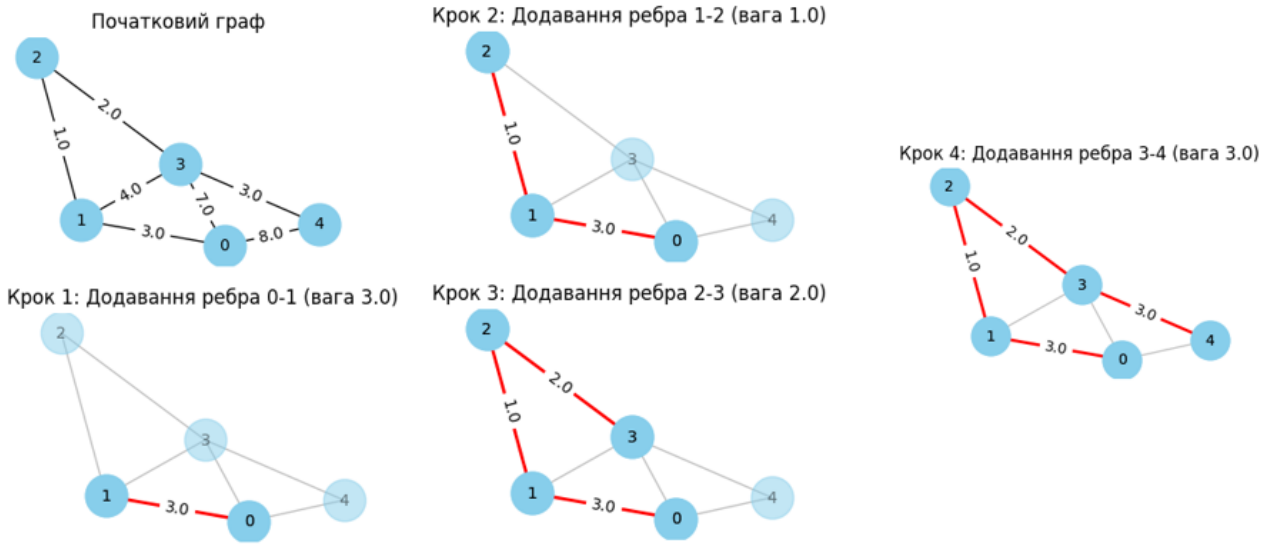
- Ініціалізація черги займатиме $O(V)$, V — кількість вершин.

- Знаходження мінімуму у бінарній купі — $O(logV)$, а оскільки ми повторюємо операцію V разів — $O(VlogV)$.
- Виконуючи оновлення на кожному кроці для кожного ребра отримаємо складність $O(ElogV)$, E — кількість ребер.
- $O(V)$ опустимо як константу, тоді: $O(VlogV + ElogV) = O(ElogV)$

Реалізація алгоритму Прима на Python

```
def prim_mst(graph, pos):
    start_node = list(graph.nodes())[0]
    visited = set([start_node])
    edges = [(data['weight'], start_node, to) for to, data in graph[
       start_node].items()]
    heapq.heapify(edges)
    mst = []
    step = 1

    while edges:
        weight, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, weight))
            visualize_mst_step(graph, mst, pos, step)
            step += 1
            for to_next, data in graph[to].items():
                if to_next not in visited:
                    heapq.heappush(edges, (data['weight'], to, to_next))

    return mst
```

Вивід: візуалізація алгоритму.



Повний код з частиною візулізації та вводу даних у додатку.

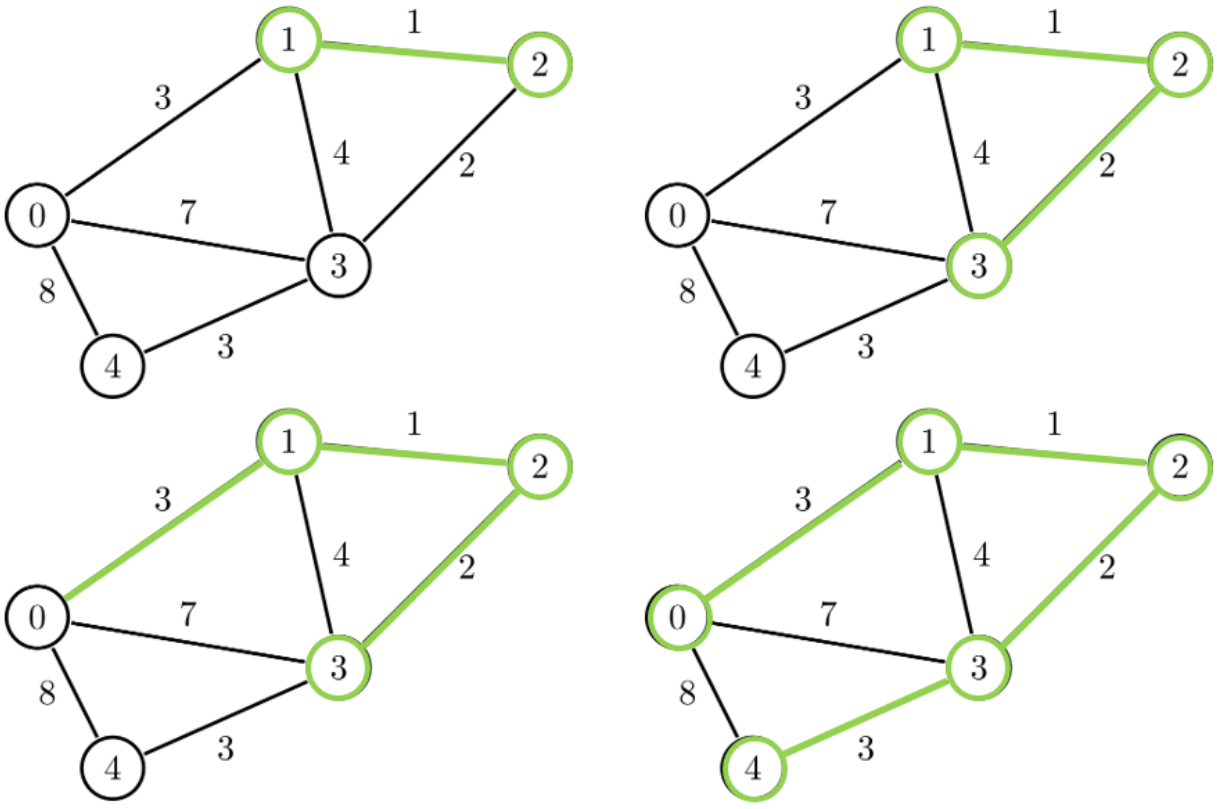
5.4 Алгоритм Крускала

Алгоритм Крускала будує мінімальне остовне дерево (MST), сортує всі ребра за зростанням ваги і покроково додає ребра до MST, не утворюючи цикли.

1. Відсортуємо всі ребра графа за зростанням ваги.
2. Почнемо з порожнього остовного дерева.
3. Виберемо найменше ребро і додаймо його до MST, якщо це не утворює циклу.
4. Повторимо крок 3, доки MST не містить $V - 1$ ребер, де V — кількість вершин у графі.

Алгоритм Крускала для пошуку мінімального шляху

Використаємо ту ж задачу, на якій було показано алгоритм Прима. Єдина відмінність полягатиме в тому, що точкою відправлення може слугувати будь-який склад — вантажівка є на кожному складі і вона має заїхати в кожний та забрати звідти вантаж.



Випишемо всі ребра у відсортованому порядку за вагами в дужках: 1-2(1), 2-3(2), 1-0(3), 3-4(3), 1-3(4), 0-3(7), 0-4(8).
У нашому випадку, можемо впорядковано додати перші $V-1$ ребер у потрібне остовне дерево, оскільки ці ребра не утворюватимуть циклу і таким чином ми отримаємо дерево, що містить всі вершини-склади з мінімальними вагами на ребрах. У цьому випадку, вантажівка може виїхати або зі складу 0, або зі складу 4, щоб оптимально зібрати вантаж з кожного.

Оцінка алгоритму Крускала

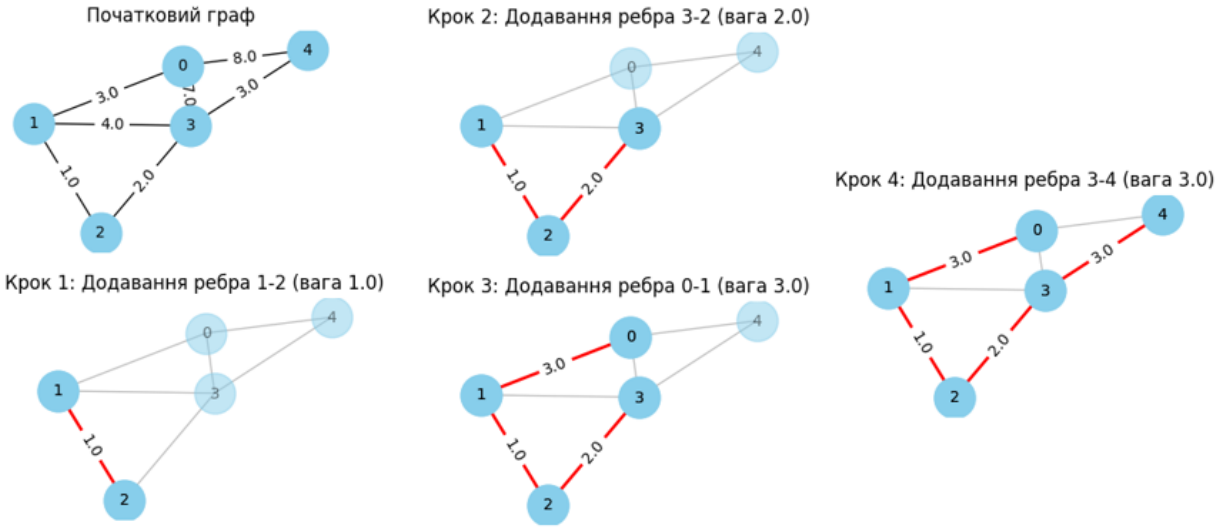
Оцінити алгоритм можемо покроково, враховуючи складність кожного етапу.

- Спочатку сортуємо E ребр за вагами. Більшість алгоритмів сортування, такі як Quick sort, Heap sort або Merge sort виконуватимуть сортування за $O(ElogE)$.
- Знаходження та об'єднання на V вершинах займатиме $O(\alpha(V))$, для E ребер — $O(E\alpha(V))$.
- Загальна часова складність: $O((E)\log E)$, оскільки у $O(E\alpha(V))$ функція $\alpha(V)$ зростає дуже повільно — настільки, що ми можемо враховувати її як константу, тож на складність алгоритму впливає лише сортування.

Реалізація алгоритму Крускала на Python

```
def kruskal_mst(graph, pos):
    edges = sorted(graph.edges(data=True), key=lambda x: x[2]['weight'])
    parent = {}
    rank = {}
    def find(node):
        if parent[node] != node:
            parent[node] = find(parent[node])
        return parent[node]
    def union(node1, node2):
        root1 = find(node1)
        root2 = find(node2)
        if root1 != root2:
            if rank[root1] > rank[root2]:
                parent[root2] = root1
            else:
                parent[root1] = root2
                if rank[root1] == rank[root2]:
                    rank[root2] += 1
    for node in graph.nodes():
        parent[node] = node
        rank[node] = 0
    mst = []
    step = 1
    for u, v, data in edges:
        if find(u) != find(v):
            union(u, v)
            mst.append((u, v, data['weight']))
            visualize_mst_step(graph, mst, pos, step)
            step += 1
    return mst
```

Вивід: візуалізація алгоритму.



Повний код з частиною візулізації та вводу даних у додатку.

5.5 Порівняння алгоритмів Прима та Крускала

Алгоритми Прима та Крускала для пошуку мінімального остовного дерева насправді мають багато чого спільного: вирішують одну проблему, обидва є жадібними алгоритмами та, фактично, мають однакову часову складність (за умови використання бінарної купи в алгоритмі Прима). Проте все-таки недоліки кожного з цих алгоритмів будуть варіюватися від поставленої задачі. Насамперед, найбільша відмінність полягає в тому, що для алгоритма Прима потрібно знати або ж самостійно призначати точку відправлення, в той час як алгоритм Крускала працює незалежно від початкової вершини. Це робить алгоритм Прима ефективним для щільних графів та простим у реалізації для графів з матрицею суміжності. Відповідно алгоритм Крускала не буде настільки ефективним в роботі з щільними графами, оскілбки вимагає сортування ребер, проте краще працюватиме з розрідженими графами.

Висновки

У кваліфікаційній роботі було досліджено остовні дерева, розглянуто питання їхньої кількість для повного графа з використанням формули Келі та коду Прюфера. Також були розглянуті різні алгоритми побудови остовних дерев та було проведено їхній аналіз. Низка задач на застосування теореми Келі та коду Прюфера разом з прикладними реалізаціями алгоритмів Прима, Крускала та коду Прюфера складають практичну частину роботи.
Отже:

1. Оглянуто основні визначення та теореми, пов'язані з теорією графів та остовних дерев.

2. Розібрано теорему Келі, що визначає кількість остовних дерев для повного графа K_n як n^{n-2}. Також розглянуто код Прюфера, що однозначно визначає остовне дерево і наводить одне з класичних доведень формули Келі.

3. Розглянуто задачі, що використовують формулу Келі, код Прюфера та комбінаторний підхід; реалізовано алгоритми кодування та декодування дерев за допомогою коду Прюфера на Python.

4. Огляд алгоритмів побудови остовних дерев на незважених графах — пошук в ширину (BFS) та вглиб (DFS).

5. Детально розібрано алгоритми Прима та Крускала для пошуку мінімальних остовних дерев на зважених графах, проаналізовано їхню складність та наведено реалізацію алгоритмів на мові програмування Python зі зручною візуалізацією.

Список літератури

1. *Cayley A.*, A theorem on trees, Quart. J. Pure Appl. Math., 23 (1889), 376–378; Collected Mathematical Papers, Vol. 13, Cambridge University Press, 1897, 26–28.

2. *Moon J. W.*, Counting labeled trees, William Clowes and Sons, 1970.

3. *Aigner M., Ziegler G. M., Hofmann K. H., Erdos P.*, Proofs from the Book, Vol. 274, Springer, Berlin, 2010.

4. *Casarotto C.*, Graph Theory and Cayley's Formula, August 10, 2006.

5. *Diestel R.*, Graph Theory, Electronic Edition, Springer, 2000.

6. *Кузьменко І.М.*, Теорія графів, навч. посіб. для здобувачів ступеня бакалавра за освітньою програмою «Комп'ютерний моніторинг та геометричне моделювання процесів і систем» спеціальності 122 «Комп'ютерні науки», С. 8-41.

7. *Трохимчук Р. М., Нікітченко М. С.*, Дискретна математика у прикладах і задачах, С. 161 – 180; 416 – 417.

8. *Карнаух Т.О., Ставровський А.Б.*, Теорія графів у задачах, навч. посіб. для студентів, які вивчають базовий нормативний курс «Дискретна математика», С. 1 – 32.

Додатки

Додаток 1 — Код Прюфера

```
import networkx as nx
import matplotlib.pyplot as plt

def input_graph():
    n = int(input("# of vertices: "))
    graph = nx.Graph()
    graph.add_nodes_from(range(1, n+1))
    for i in range(1, n+1):
        edges = list(map(int, input(f"Input adjacent vertices for {i}: ").
            split()))
        for edge in edges:
            if edge != i:
                graph.add_edge(i, edge)
    return graph

def prufer_code(graph):
    code = []
    G = graph.copy()
    plot_graph(G)
    while len(G.nodes) > 2:
        leaves = sorted(node for node in G.nodes if G.degree(node) == 1)
        leaf = leaves[0]
        neighbor = next(G.neighbors(leaf))
        code.append(neighbor)
        G.remove_node(leaf)
        plot_graph(G)
    return code

def plot_graph(G):
    plt.figure(figsize=(4, 2))
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='#92d050')
    plt.show()

graph = input_graph()

code = prufer_code(graph)
print("Prufer's code:", code)
```

Додаток 2 — Декод коду Прюфера

```
import networkx as nx
import matplotlib.pyplot as plt

def prufer_to_tree(prufer_code, initial_edge):
    vertices = len(prufer_code) + 2
    vertex_set = set(range(1, vertices + 1))
    G = nx.Graph()
    G.add_nodes_from(vertex_set)
    G.add_edge(*initial_edge)
    pos = nx.spring_layout(G)
    draw_graph(G, pos, "Initial edge")

    degree = {i: 1 for i in vertex_set}
    for node in prufer_code:
        degree[node] += 1

    for node in prufer_code:
        for v in vertex_set:
            if degree[v] == 1:
                G.add_edge(node, v)
                degree[node] -= 1
                degree[v] -= 1
                vertex_set.remove(v)
                draw_graph(G, pos, f"Adding edge ({node}, {v})")
                break

    u, v = vertex_set
    G.add_edge(u, v)
    draw_graph(G, pos, f"Adding the last edge ({u}, {v})")

    return G

def draw_graph(G, pos, title=""):
    plt.figure(figsize=(4, 3))
    nx.draw(G, pos, with_labels=True, node_color='#92d050', edge_color='
       #909090', node_size=500)
    plt.title(title)
    plt.show()

prufer_code = [int(x) for x in input("Input Prufer's code: ").split()]
initial_edge = tuple(map(int, input("Input initial edge 'v1 v2': ").split())
   )

tree = prufer_to_tree(prufer_code, initial_edge)
```

Додаток 3 — Алгоритм Прима

```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

def prim_mst(graph, pos):
    start_node = list(graph.nodes())[0]
    visited = set([start_node])
    edges = [(data['weight'], start_node, to) for to, data in graph[
       start_node].items()]
    heapq.heapify(edges)
    mst = []
    step = 1

    while edges:
        weight, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, weight))
            visualize_mst_step(graph, mst, pos, step)
            step += 1
            for to_next, data in graph[to].items():
                if to_next not in visited:
                    heapq.heappush(edges, (data['weight'], to, to_next))

    return mst

def visualize_graph(graph, pos, title):
    plt.figure(figsize=(3, 2))
    nx.draw(graph, pos, with_labels=True, node_color='skyblue', node_size
       =700, font_size=10)
    labels = nx.get_edge_attributes(graph, 'weight')
    nx.draw_networkx_edge_labels(graph, pos, edge_labels=labels)
    plt.title(title)
    plt.show()

def visualize_mst_step(graph, mst_edges, pos, step):
    mst_graph = nx.Graph()
    mst_graph.add_weighted_edges_from(mst_edges)

    plt.figure(figsize=(3, 2))
    nx.draw(graph, pos, with_labels=True, node_color='skyblue', node_size
       =700, font_size=10, edge_color='gray', alpha=0.5)
    nx.draw(mst_graph, pos, with_labels=True, node_color='skyblue',
       node_size=700, font_size=10, edge_color='red', width=2)
    labels = nx.get_edge_attributes(mst_graph, 'weight')
```

```
    nx.draw_networkx_edge_labels(mst_graph, pos, edge_labels=labels)
    plt.title(f"Step {step}: Adding edge {mst_edges[-1][0]}-{mst_edges
        [-1][1]} (weight {mst_edges[-1][2]})")
    plt.show()

num_edges = int(input("# of edges: "))
edges = []
print("Input edges '1 2 0.5' (v1 v2 weight):")
for _ in range(num_edges):
    u, v, weight = input().split()
    edges.append((int(u), int(v), float(weight)))

G = nx.Graph()
for u, v, weight in edges:
    G.add_edge(u, v, weight=weight)

pos = nx.spring_layout(G)
visualize_graph(G, pos, "Initial graph")

mst_edges = prim_mst(G, pos)
```

Додаток 4 — Алгоритм Крускала

```
import networkx as nx
import matplotlib.pyplot as plt

def kruskal_mst(graph, pos):
    edges = sorted(graph.edges(data=True), key=lambda x: x[2]['weight'])
    parent = {}
    rank = {}
    def find(node):
        if parent[node] != node:
            parent[node] = find(parent[node])
        return parent[node]
    def union(node1, node2):
        root1 = find(node1)
        root2 = find(node2)
        if root1 != root2:
            if rank[root1] > rank[root2]:
                parent[root2] = root1
            else:
                parent[root1] = root2
                if rank[root1] == rank[root2]:
                    rank[root2] += 1
    for node in graph.nodes():
        parent[node] = node
        rank[node] = 0
    mst = []
    step = 1
    for u, v, data in edges:
        if find(u) != find(v):
            union(u, v)
            mst.append((u, v, data['weight']))
            visualize_mst_step(graph, mst, pos, step)
            step += 1
    return mst

def visualize_graph(graph, pos, title):
    plt.figure(figsize=(3, 2))
    nx.draw(graph, pos, with_labels=True, node_color='skyblue', node_size
        =700, font_size=10)
    labels = nx.get_edge_attributes(graph, 'weight')
    nx.draw_networkx_edge_labels(graph, pos, edge_labels=labels)
    plt.title(title)
    plt.show()

def visualize_mst_step(graph, mst_edges, pos, step):
    mst_graph = nx.Graph()
```

```
45      mst_graph.add_weighted_edges_from(mst_edges)
46
47      plt.figure(figsize=(3, 2))
48      nx.draw(graph, pos, with_labels=True, node_color='skyblue', node_size
            =700, font_size=10, edge_color='gray', alpha=0.5)
49      nx.draw(mst_graph, pos, with_labels=True, node_color='skyblue',
            node_size=700, font_size=10, edge_color='red', width=2)
50      labels = nx.get_edge_attributes(mst_graph, 'weight')
51      nx.draw_networkx_edge_labels(mst_graph, pos, edge_labels=labels)
52      plt.title(f"Step {step}: Adding edge {mst_edges[-1][0]}-{mst_edges
            [-1][1]} (weight {mst_edges[-1][2]})")
53      plt.show()
54
55  num_edges = int(input("# of edges: "))
56  edges = []
57  print(Input edges '1 2 0.5' (v1 v2 weight):")
58  for _ in range(num_edges):
59      u, v, weight = input().split()
60      edges.append((int(u), int(v), float(weight)))
61
62  G = nx.Graph()
63  for u, v, weight in edges:
64      G.add_edge(u, v, weight=weight)
65
66  pos = nx.spring_layout(G)
67  visualize_graph(G, pos, "Initial graph")
68
69  mst_edges = kruskal_mst(G, pos)
```