Лабораторна робота №2

Графи

Хімяк Вікторія та Стратюк Арсеній

У цій роботі реалізовано декілька функцій для роботи з графами, включно з їх зчитуванням у різних форматах (матриця інцидентності, матриця суміжності, словник суміжності) та обходами графів за допомогою алгоритмів пошуку глибини (DFS) та ширини (BFS). Обходи реалізовані як ітеративно, так і рекурсивно для різних форматів подання графа.

Опис функцій

read_incidence_matrix(filename: str) -> list[list[int]]

Ця функція відповідає за зчитування матриці інцидентності графа з файлу та повернення її у вигляді списку списків. З самого початку відкривається файл та відбувається ітерація по кожному рядку з файлу, під час якої розпізнаються вершини та ребра графа. Після визначення кількості вершин і ребер, створюється матриця розміром edge_count на vertices_count (n×m), повністю заповнена нулями. Згодом ітеруючись по кожному ребру, визначаємо:

- якщо це петля, ставимо у відповідність значення 2
- якщо вершина x (початок ребра) - ставиться у відповідність 1
- якщо вершина y (кінець ребра) - ставиться у відповідність -1

read_adjacency_matrix(filename: str) -> list[list[int]]

Ця функція відповідає за зчитування матриці суміжності графа з файлу та повернення її у вигляді списку списків. Спершу відкривається файл та відбувається ітерація по кожному рядку з файлу, під час якої розпізнаються

ребра, які додаються до списку edges. Згодом всі вершини додаються до множини nodes для визначення загальної кількості вершин, визначаються розміри та створюється матриця заповнена 0. І тоді для кожного наявного ребра, значення в матриці встановлюється як 1.

read_adjacency_dict(filename: str) -> dict[int, list[int]]

Ця функція відповідає за зчитування матриці та повернення словника (номер вершини: список суміжних вершин). З самого початку відкривається файл та відбувається ітерація по кожному рядку з файлу, під час якої розпізнаються вершини (x та y):

- якщо вершини x ще не має у нашому списку, вона додається як ключ зі значенням – порожній список.

- якщо вершина, яка має ребро з x, то вона додається до списку x.

- якщо вершина y з'являється як кінцева точка, вона також додається як ключ зі значенням – порожній список.

iterative_adjacency_dict_dfs(graph: dict[int, list[int]], start: int) -> list[int]

З початку ініціалізуються змінні visited (відвідані вершини), path (послідовність обходу), stack (стек із початковою вершиною). Початкова вершина витягується зі стеку, позначається як відвідана та додається до результату. Її «сусіди» додаються до стеку у зворотному порядку і ця дія повторюється до того моменту, поки стек не буде пустим.

iterative_adjacency_matrix_dfs(graph: list[list], start: int) -> list[int]

Спершу ініціалізуємо змінні num_vertices (кількість вершин), visited (щоб позначати, чи була вершина відвідана чи ні), path (список, у який додаються вершини в порядку їх відвідування), explore (щоб зберігати індекс ‘сусіда’) та search (вершини, які потрібно перевірити). Заходячи в основний цикл (відбувається, допоки наявні вершини, які потрібно перевірити), витягується останній елемент зі стеку та якщо він не був ще перевірений, ця вершина відзначається як відвідана та додається до шляху. Ініціалізується змінна next_vertex, для того, щоб використати її для пошуку наступної сусідньої

вершини. Поступово перебираються всі ‘сусіди’ поточної вершини, і якщо вона є ‘сусідом’, але ще не була відвідана, то зберігаємо її як наступну вершину для відвідування та зберігаємо індекс наступного сусіда для поточної вершини. Якщо вже всі сусіди були відвідані, то видаляємо поточну вершину зі стеку, якщо ж знайшли сусідню вершину для переходу – то додаємо її в стек, щоб потім перевірити.

recursive_adjacency_dict_dfs(graph: dict[int, list[int]], start: int) -> list[int]

Ініціалізується змінна visited, яка використовується для зберігання вже відвіданих вершин, а список result буде зберігати порядок обходу графа. Згодом викликається функція dfs. Поточна вершина позначається як відвідана та додається до result. Відбувається ітерація по сусідніх вершинах, і якщо вона ще не була відвідана, ми будемо рекурсивно про ній проходитись.

recursive_adjacency_matrix_dfs(graph: list[list[int]], start: int) -> list[int]

Ініціалізуються змінні visited (список із вершин, які вже були відвідані) та path (щоб записувати порядок відвідування вершин). Тоді викликається рекурсивна функція explore, яка виконує основну логіку DFS для поточної вершини. Спершу позначаємо поточну вершину вже як відвідану, додаємо її до списку path та перевіряємо всіх ‘сусідів’ вершини. Якщо між вершиною current та next_vertex є ребро і ця вершина ще не була відвідана, то викликаємо знову функцію explore.

iterative_adjacency_dict_bfs(graph: dict[int, list[int]], start: int) -> list[int]

Виконує обхід графу методом пошуку в ширину (BFS) за допомогою ітеративного підходу. Граф передається у вигляді словника (adjacency dict), де ключі відповідають вершинам, а значення – списку сусідів для кожної вершини. Алгоритм починається з вказаної початкової вершини (start). Використовується черга для обробки, а також множина для позначення відвіданих вершин. Цей метод дозволяє ефективно обробляти кожну вершину графа лише один раз.

iterative_adjacency_matrix_bfs(graph: list[list[int]], start: int) -> list[int]

Аналогічно виконує BFS, але граф представлений у вигляді матриці суміжності (adjacency matrix). Тут кожен рядок і стовпець представляють вузли, а елементи матриці вказують на наявність зв'язку між вузлами. Обхід також використовує чергу для підтримки порядку відвідання та множину для контролю відвіданих вершин.

recursive_adjacency_dict_bfs(graph: dict[int, list[int]], start: int) -> list[int]

Реалізує BFS за допомогою рекурсії. Вона використовує вкладену допоміжну функцію bfs_helper, яка виконує основну роботу. Рекурсивний підхід дозволяє більш компактно реалізувати алгоритм, але може бути менш ефективним у використанні пам'яті через виклики стека.

recursive_adjacency_matrix_bfs(graph: list[list[int]], start: int) ->list[int]

Виконує те саме, що й попередня, але працює з матрицею суміжності. Рекурсивний підхід зручний для короткого коду, але може бути обмежений максимальним рівнем рекурсії для дуже великих графів.

adjacency_matrix_radius(graph: list[list]) -> int

Обчислює радіус графа (мінімальний ексцентриситет усіх вершин графа) за допомогою BFS. Спочатку вона знаходить відстані від кожної вершини до всіх інших за допомогою вкладеної функції get_distances. Потім обчислюються ексцентриситети для кожної вершини, і вибирається мінімальне значення.

adjacency_dict_radius(graph: dict[int, list[int]]) -> int

Працює аналогічно попередній, але граф представлений у вигляді словника. Вона використовує BFS для знаходження відстаней і обчислює ексцентриситети для визначення радіуса графа.

Основна частина скрипта

У наступному коді реалізується порівняльний аналіз продуктивності різних алгоритмів обходу графів, включаючи пошук у ширину (BFS) та пошук у глибину (DFS), з використанням як ітеративного, так і рекурсивного підходів. Графи представлені у вигляді матриць суміжності та словників суміжності, що дозволяє оцінити ефективність алгоритмів на різних типах структур даних.

Код починається з перевірки функціональності за допомогою модуля doctest, який автоматично запускає вбудовані тести. Потім визначається декоратор time_decorator, який вимірює час виконання функцій. Цей декоратор використовується для всіх алгоритмів обходу графів, що дозволяє автоматизувати збір даних про їхню продуктивність.

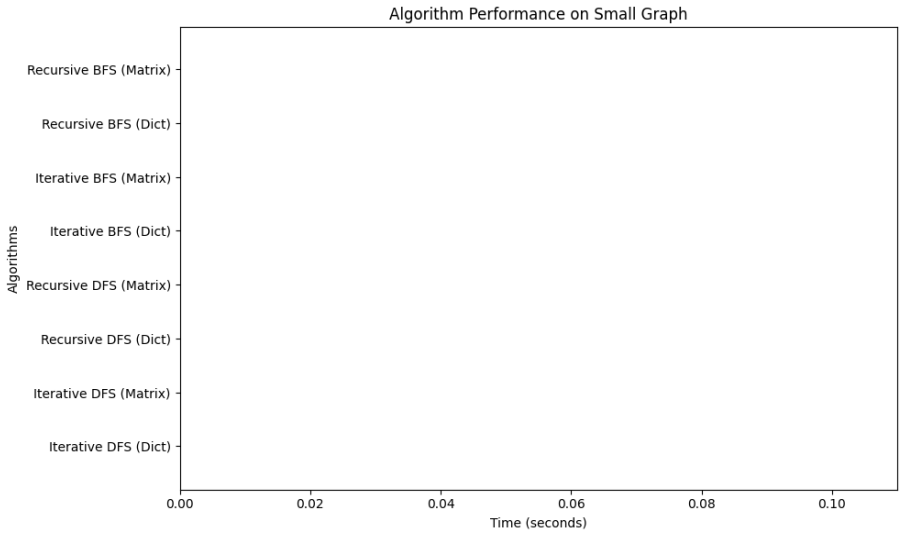
Алгоритми включають ітеративні та рекурсивні реалізації DFS та BFS для графів, представлених у вигляді матриць та словників суміжності. Усі ці алгоритми обгорнуті у time_decorator, щоб забезпечити точне вимірювання часу виконання кожного з них.

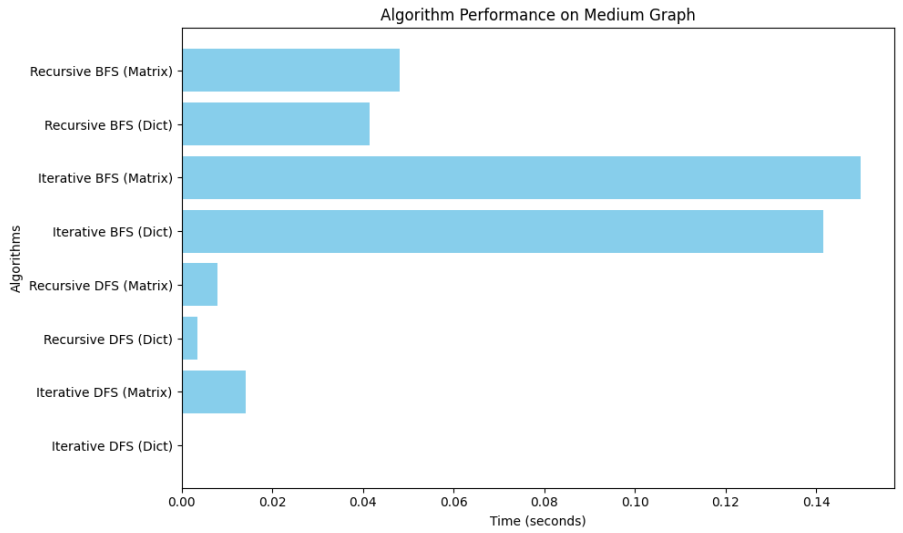
Для тестування продуктивності створюються три зразки графів: малий, середній та великий. Малий граф заданий вручну, а середній і великий графи генеруються випадково за допомогою бібліотеки NumPy. Графи зберігаються у вигляді словників та матриць, що дозволяє перевірити алгоритми на різних представленнях графів.

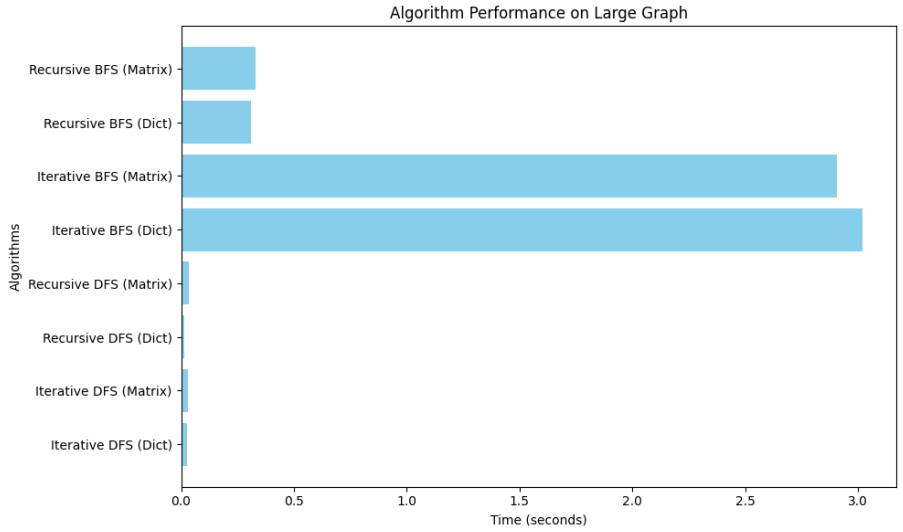
Далі проводиться ітерація по кожному графу та кожному алгоритму. Час виконання обчислюється за допомогою декоратора, а результати зберігаються в словнику results. Для кожного графа результати виконання алгоритмів виводяться на екран, щоб користувач міг спостерігати за процесом аналізу.

Для візуалізації результатів використовується функція plot_results, яка створює горизонтальні гістограми. Ці графіки показують час виконання кожного алгоритму на кожному графі, що дозволяє легко порівняти їхню продуктивність. Візуалізація реалізована за допомогою бібліотеки Matplotlib.

Код надає зручний інструмент для аналізу продуктивності різних алгоритмів роботи з графами. Він демонструє, як структура даних графа та спосіб реалізації алгоритму можуть впливати на ефективність, особливо для великих графів.







Результати показують, що для маленьких графів час виконання алгоритмів майже нульовий, що є очікуваним, оскільки обсяг даних дуже малий. Однак із збільшенням розміру графа час виконання починає зростати, зокрема для BFS алгоритмів, які займають значно більше часу, ніж DFS. Наприклад, для середнього графа алгоритми BFS займають від 0.15 до 0.04 секунд, а DFS – значно менше.

Для великих графів час виконання для алгоритмів BFS значно збільшується. Ітеративні методи BFS займають понад 2.9 секунд, що відображає високу складність обробки великих графів. У той же час, рекурсивні методи BFS, хоча і швидші за ітеративні, також значно повільніші за DFS. Це можна пояснити тим, що BFS вимагає обробки більшої кількості зв'язків одночасно, що призводить до більших затрат часу, особливо в разі великої кількості вершин і ребер. Причому складність алгоритмів однакова O(V + E).

До того ж незвично, що рекурсивні алгоритми, перевершили ітеративні в ефективності навіть без використання декоратора @lru_cache. Це можна пояснити особливостями імплементації, адже рекурсивні виявилися значно простішими.