Міністерство освіти і науки України
Національний університет “Києво-Могилянська академія”
Факультет інформатики
Кафедра математики

Кваліфікаційна робота

освітній ступінь — бакалавр

на тему: **“ЗАСТОСУВАННЯ ПАТЕРНІВ ПРОЕКТУВАННЯ В СУЧАСНОМУ C++ ДЛЯ РЕАЛІЗАЦІЇ ОПЕРАТОРІВ НА ГРАФАХ”**

Виконав: студент 4-го року навчання
освітньої програми “Прикладна математика”,
спеціальності 113 Прикладна математика

Нагорний Юрій Андрійович

Керівник: Бублик В. В.
кандидат фіз.-мат. наук, доцент

Рецензент: _____

Кваліфікаційна робота захищена
з оцінкою _____

Секретар ЕК_____
(підпис)

“_____” _____ 20___р.

Київ - 2026

Зміст

АНОТАЦІЯ

Кваліфікаційну роботу присвячено проблемі підвищення обчислювальної ефективності структурних графових операторів шляхом їх інтеграції у подіє-орієнтовану архітектуру спільного обходу засобами сучасного стандарту C++20. Встановлено архітектурні обмеження класичних узагальнених бібліотек (на прикладі BGL) та сформульовано математичні критерії сумісності операторів із механізмом спільного обходу.

Розроблено оригінальний алгоритм побудови реберного графа та адаптовано ітеративний алгоритм розпізнавання реберних графів ILIGRA до виконання за один прохід пошуку в ширину (BFS). Спроєктовано прототип графової бібліотеки із застосуванням статичних патернів Відвідувач та Компонувальник на базі концептів (Concepts) із нульовими накладними витратами.

Обчислювальні експерименти довели, що застосування спільного обходу забезпечує до 16% приросту продуктивності завдяки мінімізації промахів кешу процесора, а декомпозиція алгоритмів на події Відвідувача скорочує час виконання масивних структур до 27% за рахунок агресивного вбудовування функцій (inlining) компілятором.

Ключові слова: теорія графів, реберний граф, алгоритм ILIGRA, патерни проєктування, C++20, спільний обхід, патерн Відвідувач.

ABSTRACT

This qualification paper is devoted to the problem of improving the computational efficiency of structural graph operators by integrating them into an event-driven joint traversal architecture using modern C++20 standard tools. The architectural limitations of classic generic libraries (using BGL as an example) are established, and the mathematical criteria for the compatibility of operators with the joint traversal mechanism are formulated.

An original algorithm for line graph construction is developed, and the iterative line graph recognition algorithm ILIGRA is adapted for execution in a single pass of the breadth-first search (BFS). A prototype of a graph library is designed utilizing static Visitor and Composite design patterns based on Concepts with zero-overhead abstractions.

Computational experiments have proven that the application of joint traversal provides up to a 16% performance increase by minimizing CPU cache misses. Furthermore, the decomposition of algorithms into Visitor events reduces the execution time for massive structures by up to 27% due to aggressive function inlining by the compiler.

Keywords: graph theory, line graph, ILIGRA algorithm, design patterns, C++20, joint traversal, Visitor pattern.

ВСТУП

Актуальність теми. Графи є фундаментальним математичним інструментом для моделювання складних систем різної природи. Особливе місце в теорії графів посідають реберні графи (line graphs) [12], які дозволяють перенести фокус аналізу з вершин на їх зв'язки. Моделювання за допомогою реберних графів знаходить широке застосування в задачах маршрутизації мережевого трафіку, пошуку вразливостей топології, кластеризації даних та розподілі ресурсів. Водночас задача відновлення прообразу реберного графа є обчислювально складною і вимагає застосування сучасних ітеративних алгоритмів, таких як ILIGRA (Iterative Line Graph Recognition Algorithm), запропонований у 2015 році.

Для розв'язання прикладних графових задач традиційно застосовуються спеціалізовані програмні бібліотеки. The Boost Graph Library (BGL) [20] є стандартом реалізації графових алгоритмів за принципами узагальненого програмування. Архітектурні рішення BGL, зокрема застосування патернів проєктування Відвідувач (Visitor) та Компонувальник (Composite) для оптимізації алгоритмів, залишаються концептуально актуальними і сьогодні. Проте бібліотека була розроблена на базі застарілого стандарту C++98. Використання громіздких конструкцій тогочасного метапрограмування призводить до високої синтаксичної складності. Із появою стандарту C++20 (зокрема концептів, діапазонів та семантики ідеальної передачі [15, 14]) виникла можливість переосмислення та осучаснення цих класичних патернів для досягнення абстракцій з нульовими накладними витратами (zero-overhead abstractions).

Об'єкт дослідження. Обчислювальні процеси та алгоритми структурних перетворень на графах.

Предмет дослідження. Графові оператори в контексті подіє-орієнтованого обходу та сучасні статичні патерни проєктування C++ як засіб їхньої ефективної реалізації.

Мета дослідження. Визначення алгоритмічних вимог до графових операторів для їхнього ефективного виконання під час спільного обходу, розробка нових та адаптація існуючих топологічних алгоритмів (зокрема побу-

дови реберного графа та алгоритму ILIGRA) до подіє-орієнтованої архітектури, а також програмна реалізація цих операторів із застосуванням сучасних засобів метапрограмування та патернів проєктування C++20.

Завдання дослідження. Для досягнення поставленої мети було сформульовано такі завдання:

1. Провести аналіз класичних підходів до реалізації узагальнених графових бібліотек (на прикладі BGL) та виявити їхні архітектурні обмеження.

2. Сформулювати математичні та алгоритмічні критерії, яким мають відповідати графові оператори для їхньої ефективної інтеграції у механізм спільного обходу (BFS/DFS).

3. Розробити оригінальний алгоритм побудови реберного графа, здатний працювати строго в межах одного проходу пошуку в ширину.

4. Адаптувати сучасний ітеративний алгоритм ILIGRA для відновлення прообразу реберного графа до виконання в межах одного проходу пошуку в ширину.

5. Спроєктувати та реалізувати архітектуру прототипу сучасної графової бібліотеки мовою C++20 з використанням концептів (Concepts) та статичного поліморфізму.

6. Провести обчислювальні експерименти для емпіричної оцінки продуктивності та локальності даних під час спільного обходу графів великої розмірності.

Методи дослідження. У роботі застосовано апарат дискретної математики та теорії графів (для формалізації операторів та доведення їхніх властивостей), теорію алгоритмів та аналіз обчислювальної складності (для оцінки асимптотичної поведінки), методи узагальненого програмування та метапрограмування (для проєктування архітектури).

Наукова новизна. У роботі вперше запропоновано та формалізовано адаптацію алгоритму розпізнавання реберних графів ILIGRA, а також алгоритму побудови реберного графа, до виконання у парадигмі подіє-орієнтованого обходу в ширину (BFS).

Практичне значення дослідження. Створено робочий прототип сучасної високопродуктивної графової бібліотеки мовою C++, який дозволяє користувачам ефективно комбінувати складні топологічні оператори зі збереженням оптимального використання кешу процесора. Розроблені логічні алгоритми та їхні подіє-орієнтовані адаптації є універсальними і можуть бути імплементовані в інших графових фреймворках, що підтримують ідею спільного обходу через патерн Відвідувач.

Апробація результатів дослідження. Основні теоретичні та практичні результати кваліфікаційної роботи доповідалися та обговорювалися на XIV Всеукраїнській науковій конференції молодих математиків, Український державний університет імені Михайла Драгоманова, Київ, Україна, 7—8 травня 2026 р. Матеріали конференції розміщено у відкритому доступі [2].

Структура та обсяг роботи. Кваліфікаційна робота складається із анотації, вступу, чотирьох розділів, висновків та списку використаних джерел. У першому розділі проведено аналіз класичних підходів та проблематики графових бібліотек. Другий розділ присвячено проєктуванню сучасної архітектури прототипу на базі C++20. У третьому розділі досліджуються та адаптуються конкретні графові оператори. Четвертий розділ містить результати обчислювальних експериментів. Варто зазначити, що україномовна термінологія об'єктно-орієнтованого програмування, використана у тексті роботи (наприклад, використання терміна "відсилка" для англійського *reference*), базується на академічному підручнику В. В. Бублика [1].

1 АНАЛІЗ ПІДХОДІВ ДО РЕАЛІЗАЦІЇ УЗАГАЛЬНЕНИХ ГРАФОВИХ БІБЛІОТЕК МОВОЮ C++

1.1 Основні поняття теорії графів

Розглянемо деякі означення з теорії графів необхідні для розуміння роботи, подані згідно з посібників [3], [12] та [8].

Означення 1.1. Графом (неорієнтованим графом) G називають пару (V, E), де V — непорожня скінченна множина елементів, що називаються вершинами (або точками), а E — задана множина невпорядкованих пар різних вершин з V, які називаються ребрами (або лініями).

Означення 1.2. Якщо $e = (v, w)$ — ребро графа, то вершини v і w називають кінцями ребра e. Також кажуть, що ребро e з'єднує вершини v і w.

В контексті обходу графа кінці ребер називають "джерело" та "ціль" відповідно до порядку обходу вершин в ребрі.

Означення 1.3. Якщо $(v, w) \in E$, то кажуть, що вершини v і w суміжні, інакше вершини несуміжні. Два ребра називають суміжними, якщо вони мають спільну вершину. У цій роботі такі вершини та ребра також називають сусідніми.

Означення 1.4. Вершину v і ребро e називають інцидентними, якщо вершина v один із кінців ребра e.

Означення 1.5. Степенем $\deg(v)$ вершини v називають кількість інцидентних їй ребер.

Означення 1.6. Орієнтованим графом G називають пару (V, E), де E — це множина впорядкованих пар вершин (u, v). На відміну від неорієнтованого графа, тут кожне ребро (яке називають дугою) має чіткий напрямок: воно веде від початкової вершини u до кінцевої вершини v. Пара (u, v) не є тотожною парі (v, u), що графічно зображується як стрілка.

Означення 1.7. Список суміжності (adjacency list) — це спосіб подання графа, за якого кожній вершині $v \in V$ ставиться у відповідність перелік (список) усіх вершин, безпосередньо з'єднаних із нею ребрами (її сусідів). Візуалізовано на рисунку 1

Структурно це набір списків, де базовим елементом є вершина, а вмістом її списку — усі суміжні з нею вершини. Для орієнтованих графів у список вершини v записують лише ті вершини, до яких ідуть дуги від v. Цей спосіб є найбільш ефективним для збереження та обробки розріджених графів (у яких кількість ребер значно менша за максимально можливу).

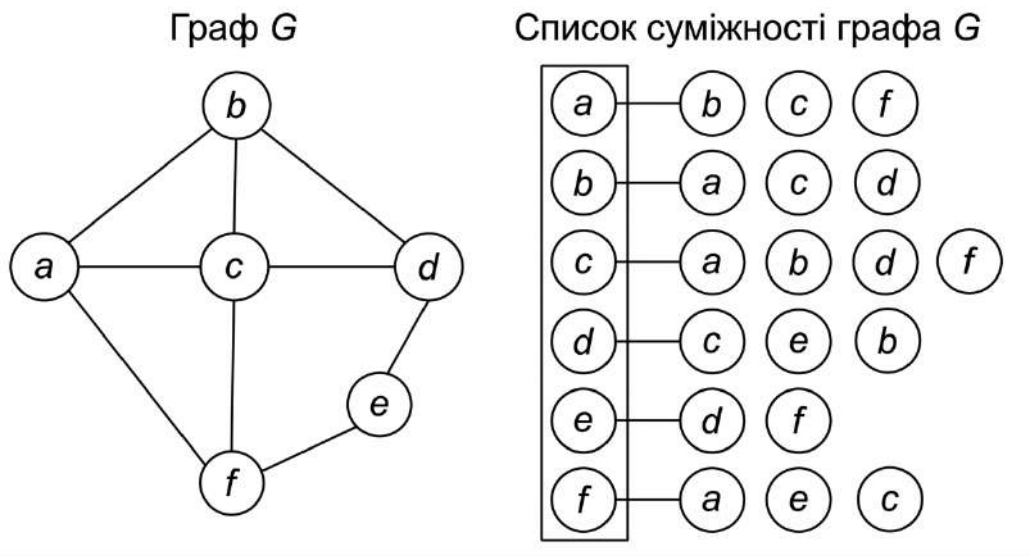

Рис. 1: Приклад та візуалізація списку суміжності.

Означення 1.8. Циклом називається послідовність вершин у графі, у якому перша та остання вершини збігаються ($v_1 = v_n$), а всі ребра у послідовності є різними. Простий цикл — це цикл, у якому всі вершини (крім першої та останньої) також не повторюються.

Означення 1.9. Граф G' є підграфом графа G, якщо множина його вершин та множина ребер є підмножинами відповідних множин графа G.

У контексті цієї роботи під терміном "підграф" розуміється індукований підграф — це підграф, який зберігає абсолютно всі ребра вихідного графа G, обидва кінці яких належать до нової підмножини вершин.

Означення 1.10. Підграф називається повним, якщо кожна його пара вершин з'єднана ребром (тобто в ньому присутні всі можливі ребра для даної кількості вершин). Такий підграф також називають клікою.

Означення 1.11. Граф називається зв'язним, якщо для будь-якої пари його різних вершин існує шлях, що їх з'єднує.

Означення 1.12. Компонентою зв'язності графа G називають його зв'язний підграф такий, що не є власним підграфом жодного іншого зв'язного підграфа графа G (тобто це максимальний за включенням зв'язний підграф).

Означення 1.13. Вершина зв'язного графа, видалення якої (разом з інцидентними їй ребрами) збільшує кількість компонент зв'язності графа, називається точкою зчленування (або шарніром).

Означення 1.14. Деревом називається зв'язний граф, який не містить жодного циклу.

Означення 1.15. Граф $G = (V, E)$ називають двочастковим, якщо існує таке розбиття $\{V_1, V_2\}$ множини його вершин V на дві підмножини (частки), що для довільного ребра $(v, w) \in E$ або $v \in V_1$ і $w \in V_2$, або $v \in V_2$ і $w \in V_1$.

Означення 1.16. Реберним графом $L(G)$ графа G називається такий граф, вершини якого відповідають ребрам графа G, і дві вершини в $L(G)$ суміжні тоді й тільки тоді, коли відповідні їм ребра суміжні в G.

Означення 1.17. Прообразом (root graph) реберного графа $L(G)$ є вихідний граф G, з якого він був утворений. Прообразом вершини в $L(G)$ є відповідне ребро в G, а прообразом ребра в $L(G)$ є пара суміжних ребер у G (які мають спільну вершину).

Означення 1.18. Властивості елементів та зважений граф. У загальному випадку елементам графа (вершинам і ребрам) можуть приписуватися додаткові атрибути або параметри, такі як мітки, кольори, імена чи стани. Найважливішим окремим випадком розміченого графа є зважений граф — граф, кожному ребру якого (а іноді й вершинам) поставлено у відповідність певне дійсне число, що називається вагою. Вага ребра зазвичай відображає фізичну відстань, вартість, час або пропускну здатність зв'язку між відповідними вершинами.

Означення 1.19. У контексті цієї роботи під графовим оператором (або оператором на графах) розуміється математичне відображення $\Phi : \mathcal{G} \to \mathcal{X}$, де областю визначення $\mathcal{G}$ є певний клас графів (тобто аргументом зліва завжди виступає граф), а областю значень $\mathcal{X}$ може бути простір графів, сімейство підмножин вершин чи ребер, множина булевих значень (для перевірки структурних властивостей), числова множина або їхня комбінація. Наприклад, оператор переходу до реберного графа $L(G)$ відображає граф у граф, тоді як оператор пошуку клік відображає граф у множину підмножин його вершин.

1.2 Огляд та проблематика класичних рішень на прикладі The Boost Graph Library

The Boost Graph Library (BGL) була першою графовою бібліотекою, що реалізувала графи та алгоритми згідно з парадигмою узагальненого програмування на рівні абстракції стандартної бібліотеки C++ (STL). Її реалізації базуються на статичному поліморфізмі та метапрограмуванні стандарту C++98, що дозволяє відділити алгоритми від структур даних. На момент публікації це було настільки значне досягнення, що сам автор STL, Александр Степанов, відзначив у передмові до The Boost Graph Library: User Guide and Reference Manual [20], що робота стала вагомим кроком у розвитку графових бібліотек.

У BGL реалізовано два основні графові класи: список суміжностей та матриця суміжностей. Обидва класи мають шаблонні аргументи, залежно від значень яких вже на етапі компіляції визначається чи є створений графовий об'єкт орієнтованим графом, чи підтримує він паралельні ребра (більше одного ребра між двома вершинами) та петлі (ребра, де кінці збігаються), чи мають ребра вагу та чи є якісь інші властивості у ребер та вершин.

Проте високий рівень абстракції BGL дозволяє використання в алгоритмах не тільки їхніх власних графових об'єктів, а й імпортованих з інших бібліотек, або навіть простого масиву масивів, що імітує список суміжностей. Таку підтримку забезпечує використання вже реалізованих у бібліотеці варіантів патерна Адаптер (Adapter), а також можливість написати власний Адаптер для будь-яких інших графових об'єктів. Цей патерн слугує обгор-

ткою, що дозволяє алгоритмам знаходити необхідні методи у класах графів з відмінними від BGL назвами чи будовою.

Щодо наповнення бібліотеки BGL, то автори реалізували значну кількість різноманітних графових алгоритмів спираючись на Introduction to Algorithms [9]. Наприклад: топологічне сортування, перевірка на планарність (алгоритм Боєра-Мирволда), пошук мінімального кістякового дерева (алгоритм Пріма), перевірка на ізоморфізм графів, алгоритм максимального потоку Едмондса-Карпа. Хоча більшість алгоритмів реалізовано у відповідності до псевдокоду з книги, подекуди розробники BGL здійснили перехід від послідовних алгоритмів до гнучкої системи точок подій (event points), застосувавши патерн Відвідувач в комбінації із алгоритмами обходу типу BFS (Breadth-First Search) , DFS (Depth-First Search).

Розглянемо детальніше як працює така комбінація. В бібліотеці BGL реалізовано алгоритми обходу:

- Пошук в ширину (`breadth_first_search`).

- Пошук в глибину (`depth_first_search`).

Ці алгоритми, формально, реалізовані як функції, що приймають об'єкт Відвідувача, обходять граф, відзначають кольорами пройдені вершини, і в конкретних точках-подіях обходу викликають відповідні методи у переданого Відвідувача. Наприклад, алгоритм обходу в глибину натрапляє на білу вершину, це означає, що він вперше відвідує вершину $v \in V(G)$. Він змінює колір вершини на сірий та викликає метод Відвідувача відповідний до події "відкриття вершини", куди v та G передаються як аргументи. По закінченню обробки вершини та її сусідів, її колір змінюється на чорний. Розглянемо події обходів.

- Стартова вершина (`start_vertex`): Викликається для першої вершини обходу на початку алгоритму.

- Відкриття вершини (`discover_vertex`): Викликається при першому відвідуванні вершини, коли її колір білий. Після події вершина додається у BFS чергу або DFS стек обходу зі зміною кольору на сірий.

- Розгляд/дослідження вершини (`examine_vertex`): Викликається для сірої вершини витягнутої з BFS черги чи DFS стеку перед розглядом її ребер та суміжних вершин.

- Деревне ребро (`tree_edge`): Викликається при розгляді інцидентних ребер сірої вершини, коли інший кінець ребра біла вершина.

- Зворотне ребро (`back_edge`): Викликається при розгляді інцидентних ребер сірої вершини, коли інший кінець ребра сіра вершина. Є лише в DFS.

- Сіра ціль (`gray_target`): Аналогічна попередній подія для BFS обходу.

- Перехресне ребро (`forward_or_cross_edge`): Викликається при розгляді інцидентних ребер сірої вершини, коли інший кінець ребра чорна вершина. Є лише в DFS.

- Чорна ціль (`black_target`): Аналогічна попередній подія для BFS обходу.

- Не деревне ребро (`non_tree_edge`): Об'єднання сценаріїв Сіра ціль та Чорна ціль. Є лише в BFS.

- Завершення вершини (`finish_vertex`): Викликається після обходу всіх суміжних вершин їй вершин. Після події колір вершини змінюється на чорний.

Можна помітити, що деякі унікальні події пошуку в глибину мають аналоги в пошуку в ширину (виникають за ідентичних умов). Хоча з точки зору програмування події ідентичні, з математичної точки зору вони мають різний зміст. Такий розподіл подій, як і ідею розфарбування вершин при обході, автори BGL взяли з Introduction to Algorithms разом з алгоритмами. Подія `back_edge` означає, що обхід у глибину натрапив на вершину з якої прийшов — утворився цикл. В той час як `gray_target`, якщо уявляти обхід в ширину як хвилі, то це рух перпендикулярний хвилі, альтернативний шлях. Подія `forward_or_cross_edge` означає, що обхід вглиб натрапив на вершину із вже завершеним обходом, або розгалуженням поточного, тобто фіксує топологічне скорочення шляху до нащадка або перехресний зв'язок з іншою

гілкою лісу обходу. В той час як `black_target` це ребро в протилежному напрямку від розповсюдження обходу в ширину.

Кожен клас Відвідувача у свою чергу має містити ці методи-події. Таке відокремлення логіки обходу графа від обчислень, що виконуються над його елементами, дозволило в поєднанні з патерном Компонувальником забезпечити можливість виконання декількох алгоритмів за один обхід графа. Детальніше про застосування цих патернів у розділі 1.4.

Однак, попри всі інноваційні ідеї, основна частина бібліотеки була написана за часів C++98. Через це, для забезпечення високого рівня абстракції та заборони некоректного використання алгоритмів користувачем, авторам бібліотеки довелося спиратися на `boost::graph_traits` та систему SFINAE (Substitution Failure Is Not An Error). Це призвело, по-перше, до надзвичайно великих за обсягом текстів помилок компіляції при спробі некоректного використання бібліотеки (так звані template instantiation errors). Оскільки такі помилки часто вказували не на першопричину на рівні інтерфейсу, а на збій інстанціювання в глибинах коду бібліотеки, авторам BGL довелося застосувати спеціальний механізм — Boost Concept Check Library (BCCL). Він перехоплював помилки користувача на ранніх етапах компіляції через систему макросів перевірки типів. Але натомість збільшувався час компіляції та перевантажувався код алгоритмів допоміжними конструкціями.

Крім того, складність використання бібліотеки зумовлювалася й іншими чинниками. Наприклад, відсутність доданого через десять років у C++11 інструменту `auto`, без якого при створенні кожного об'єкту доводилося вручну декларувати його тип, який часто складався з глибоко вкладених типів у поєднанні з шаблонними конструкціями. Іншим прикладом складності є використання вкладених `std::pair` та ітераторів для роботи з послідовностями елементів, що призводило до значної синтаксичної перевантаженості.

Підсумовуючи, бібліотека й досі залишається значним досягненням у галузі узагальненого програмування. Проте через її надмірну складність вона так і не стала популярним прикладним рішенням, і користувачі часто надавали перевагу менш гнучким бібліотекам. Із появою стандарту C++20 стало можливим переосмислити архітектуру BGL і використати її найкращі надбання для проєктування сучасної бібліотеки графових алгоритмів.

1.3 Переваги засобів метапрограмування C++20 в контексті графових бібліотек

У 2020 році було прийнято стандарт C++20 — масштабне оновлення мови програмування, що принесло із собою низку ключових нововведень (детальніше у [14]). Одним із найважливіших стали концепти (Concepts) — нативний механізм мови для встановлення обмежень на параметри шаблонів, що заміняє собою SFINAE та Boost Concept Check Library (BCCL) уникаючи їх недоліків. Концепти забезпечують прозорий статичний поліморфізм, що дозволяє користувачу явно бачити, які є вимоги до шаблонного аргументу функції. В контексті графових бібліотек це надає можливість вимагати від користувача не будь-що із потрібними методами, а тип, який задовольняє певні вимоги. Крім того при спробі передати об'єкт, що порушує вимоги концепта, компілятор видає помилку з чітко описаними порушеними вимогами використання. Наприклад: "Тип `T` не задовольняє концепт `IncidenceGraph`, бо в ньому немає метода `out_edges`".

Також концепти, а точніше вирази `if constexpr (requires ...)`, дозволяють прозоре та зрозуміле розгалуження поведінки алгоритму під час компіляції. Так, наприклад, один і той самий алгоритм компілюється у два майже ідентичні варіанти, де під час обходу ребер, варіант для неорієнтованих графів робить перевірку чи не оброблялося це ребро з іншого кінця. А у варіанті для орієнтованих графів компілятор вирізає цю перевірку з об'єктного коду, адже рух по ребру відбувається лише в один бік і перевірка не має сенсу.

Іншим важливим нововведенням стали діапазони (Ranges). При ітеруванні множинами об'єктів у BGL часто використовувалася пара ітераторів (`begin` та `end`). Діапазони ж дозволяють спростити синтаксис ітерування до, наприклад, `for (auto v : g.vertices())`, що є природним відповідником математичному $\forall v \in V(G)$ без зайвих обгорток. Цю ідею доповнюють `std::views`, що дозволяють елегантно моделювати побудову підмножин із заданими властивостями $\{v \in V \mid P(v)\}$ за допомогою лінивих обчислень (lazy evaluation) як у лістингу 1.

```
for (auto v : g.vertices()
    | std::views::filter([&](auto v){ return p(v); })) {...}
```

Лістинг 1: Ітерування відфільтрованою підмножиною

Підсумовуючи, стандарт C++20 дозволяє вирішити основну проблему бібліотеки BGL, а саме — складність використання як плату за високий рівень абстракції. Розглянуті у цьому підрозділі засоби дозволяють зберегти узагальненість без накладних витрат, але значно спростити досвід використання бібліотеки та покращити підтримуваність коду.

1.4 Патерни проектування в розробці графових алгоритмів

Попри всі вищезгадані недоліки пов'язані із використанням у BGL застарілих засобів метапрограмування, в архітектурі бібліотеки є й рішення актуальні зараз. Мова йде про застосування патернів проєктування для реалізації графових алгоритмів. Патерни проєктування — це типові архітектурні рішення поширених проблем при розробці програмного забезпечення. Вони забезпечують надійність та гнучкість архітектури, а також покращують підтримуваність коду. Основні патерни були описані ще у 1994 групою авторів, відомою як "Gang of Four" [11] (далі GoF), тож ці рішення можна назвати перевіреними часом. Тим не менш, список не вважається фінальним, а навіть ті патерни, що були представлені у GoF зазнавали з часом змін.

Як і було зазначено в розділі присвяченому огляду архітектури BGL, частина графових алгоритмів у бібліотеці реалізована на базі патерна Відвідувача (Visitor). Проте реалізація цього патерна зазнала сильних змін порівняно із класичним варіантом у GoF. На відміну від класичної реалізації де використовується подвійна диспетчеризація через пізнє зв'язування [4], у BGL застосовується статичний поліморфізм на основі шаблонів функцій. А замість методів для різних типів викликачів, Відвідувач BGL має методи для різних подій обходу графа, що слугують викликачами. І останньою відмінністю BGL Відвідувача є зберігання стану. Кожен метод-подія не обробляє незалежно вершину чи ребро, а виконує свою частину алгоритму в процесі послідовного обходу графу. Варіативність подій дозволяє розбити алгоритм

на різні частини залежно від контексту обходу, а контекст виконання та результат алгоритму отримується з полів, що зберігає Відвідувач.

Щоб не писати пусті методи для зайвих конкретному Відвідувачу подій, клас спадкується від базового Відвідувача. Наприклад, `default_bfs_visitor`, в якому вже є порожні реалізації всіх методів-подій BFS. Ці методи не віртуальні, кожен конкретний Відвідувач реалізовує потрібні йому методи, для яких відбувається перекриття імен (name hiding) базового. Оскільки алгоритми обходу приймають Відвідувачів довільного шаблонного типу без динамічного поліморфізму, якщо компілятор не знаходить реалізації певного метода в похідному класі, він підставляє виклик із базового класу. Втім, на етапі оптимізації, компілятор повністю вирізає такі порожні виклики. Це дозволяє розробнику описувати лише корисну логіку, в той час як згенерований машинний код виконується так само швидко, якби алгоритм був написаний монолітним блоком.

Таке відокремлення логіки обходу графа від обчислень, що виконуються над його елементами, дозволило в поєднанні з патерном Компонувальником (Composite) забезпечити можливість виконання декількох алгоритмів за один обхід графа. Реалізовано це через утиліту `multi_visitor`, яка за допомогою засобів метапрограмування об'єднує в собі декілька об'єктів-Відвідувачів. Алгоритм обходу отримує такий Композитний Відвідувач, і коли викликає певну подію, Компонувальник перехоплює її та послідовно делегує цей виклик відповідному методу кожного з вкладених Відвідувачів. Оскільки, на відміну від опису в GoF, у BGL реалізовано статичний варіант патерна, це розгортання відбувається на етапі компіляції і позбавлене будь-яких накладних витрат на диспетчеризацію.

Іншим прикладом оптимізації через спільне використання є застосування патерна Впровадження залежностей (Dependency Injection) [19]. Замість того, щоб Відвідувач сам створював свої структури стану, наприклад, масив відстаней від стартової вершини, користувач створює такі структури ззовні і передає їх у Відвідувач через аргументи конструктора. Це дозволяє перевикористовувати виділену пам'ять між різними запусками алгоритмів (уникаючи затратних викликів malloc/new), а також дозволяє користувачу вирішувати, де ці дані розміщуються (у кеші процесора чи на диску). Крім того, оскільки

тип контейнера задається шаблоном, в деяких ситуаціях це дозволяє користувачу змінювати стратегію виконання алгоритму, детальніше описано у розділі 2.3 на прикладі реалізованому у цій роботі.

Проте якщо можливість надавати свій контейнер для результату алгоритму це зручно, то необхідність створювати контейнери та структури для проміжних обчислень часто буває обтяжливою, це виправляє патерн проєктування Фасад. Він надає можливість викликати алгоритм через простий інтерфейс. Він самостійно створює всі не задані користувачем тимчасові структури, передає їх у об'єкт Відвідувача та запускає обхід графа.

Крім названих у цьому розділі в архітектурі BGL використовуються й інші патерни проєктування, але оскільки у практичній частині цієї роботи вони не застосовуються, їх детальний розгляд виходить за межі даного дослідження.

2 СТВОРЕНА АРХІТЕКТУРА ПРОТОТИПУ СУЧАСНОЇ ГРАФОВОЇ БІБЛІОТЕКИ

У попередньому розділі було проведено аналіз класичних підходів до побудови графових бібліотек на прикладі The Boost Graph Library та виявлено низку їхніх архітектурних обмежень, зумовлених використанням застарілих стандартів узагальненого програмування. На основі цих висновків, у даному розділі представлено архітектуру власного прототипу сучасної графової бібліотеки, розробленого в межах цієї кваліфікаційної роботи. Головною метою запропонованої архітектури є збереження узагальненості та високої продуктивності алгоритмів при одночасному спрощенні їхнього використання та підтримуваності завдяки реалізації класичних рішень BGL засобами стандарту C++20. Проєктування бібліотеки розпочинається зі створення фундаментального шару абстракцій — графових концептів та базових структур даних.

2.1 Графові концепти та клас AdjacencyList

Для підтримки незалежності алгоритмів та операторів від конкретних реалізацій граф у роботі визначається описаними нижче C++20 концептами із прикладами коду в лістингу 2.

- Базовий граф `BaseGraph` має вказувати чи є граф орієнтованим та містити типи дескрипторів вершини та ребра. Де об'єкт останнього має надавати дескриптори обох кінців ребра.

- Граф списку вершин `VertexListGraph` включає вимоги `BaseGraph` та має містити методи доступу до діапазону вершин та їх кількості.

- Граф інцидентності `IncidenceGraph` включає вимоги `BaseGraph` та має містити методи доступу до діапазону суміжних вершин вершини v та степеня вершини.

- Граф списку ребер `EdgeListGraph` включає вимоги `BaseGraph` та має містити методи доступу до діапазону ребер та їх кількості.

```
template <typename G>
concept BaseGraph = requires {
    typename G::vertex_descriptor;
    typename G::edge_descriptor;
    { G::directedness } -> std::convertible_to<DirectednessEnum>;
    requires requires(typename G::edge_descriptor e) {
        { e.source }  -> std::convertible_to<
            typename G::vertex_descriptor>;
        { e.target }  -> std::convertible_to<
            typename G::vertex_descriptor>;
    };
};

template <typename G>
concept VertexListGraph = BaseGraph<G> && requires(const G & g) {
    { g.num_vertices() } -> std::convertible_to<size_t>;
    { g.vertices() } -> std::ranges::range;
};

template <typename G>
concept IncidenceGraph = BaseGraph<G>
&& requires(const G & g, typename G::vertex_descriptor v) {
    { g.vertex_degree(v) } -> std::convertible_to<size_t>;
    { g.out_edges(v) } -> std::ranges::range;
};

template <typename G>
concept EdgeListGraph = BaseGraph<G> && requires(const G & g) {
    { g.num_edges() } -> std::convertible_to<size_t>;
    { g.edges() } -> std::ranges::range;
};
```

Лістинг 2: Графові концепти

Згідно цим концептам у роботі реалізовано клас `AdjacencyList` із шаблонним параметром `Directedness`. Як очевидно з назви, цей клас — програмна реалізація списку суміжності, а аргумент шаблону визначає чи є створений об'єкт орієнтованим чи неорієнтованим графом.

Цей клас зберігає вершини $v_i \in V;\ i \in [0; |V| - 1]$ у вигляді послі-

довних індексів контейнера `std::vector<StoredVertex>` розміру $|V|$. За кожним індексом i у контейнері зберігається об'єкт `StoredVertex`, який по суті є контейнером `std::vector<StoredEdge>` суміжних до v_i вершин. Кожен об'єкт `<StoredEdge>` зберігає індекс $j \in [0; n-1]$ суміжної до v_i вершини v_j та ідентифікатор ребра `edge_id` $\in [0; |E|-1]$. Ідентифікатор потрібен для доступу до довільних властивостей ребра у зовнішньому контейнері `std::vector<EdgePropertyType>` за сталий $O(1)$ час.

Варто зазначити, що цей клас не реалізовує жодних методів крім тих, що вимагають описані на початку концепти. Крім того, після ініціалізації об'єкт графу є незмінним. Також конструктор `AdjacencyList` не підтримує паралельні ребра (більше одного ребра між двома вершинами) та петлі (ребра, де обидва кінці збігаються).

2.2 Реалізація механізмів обходу (BFS та DFS)

Для реалізації операторів на графах у цій роботі запозичено ідею BGL із використанням патерна Відвідувач в комбінації з алгоритмами обходу. Конкретно реалізовано C++20 модифікації BFS та DFS обходів. Детальніше архітектуру пошуку в ширину розглянуто нижче.

Основою пошуку в ширину є шаблонна функція `breadth_first_visit`. Вона приймає об'єкт, тип якого має задовольняти концепти `VertexListGraph` та `IncidenceGraph`. Наступні два аргументи це стартова вершина обходу та відсилка (reference) на `std::vector` кольорів обходу. Останнім аргументом є об'єкт Відвідувача переданий універсальною відсилкою. Її використання дозволило уникнути обмежень BGL на передачу відвідувача копіюванням. Це надало можливість користувачу передавати затратні для копіювання чи некопіювальні Відвідувачі відсилками або переносом (`std::move`), залишаючи при цьому здатність алгоритму приймати малорозмірні Відвідувачі копіями.

Логіка функції майже не відрізняється від BFS у BGL. Створюється черга вершин `std::queue<vertex_descriptor>`, розглядаються її ребра та суміжні вершини, після чого вони додаються в чергу, а розглянута вершина звідти прибирається. В процесі подій розгляду викликаються відповідні методи Відвідувача. Проте запропонована реалізація у порівнянні з BGL була

доповнена ще однією подією: `forward_edge`. Ця подія фактично є об'єднанням подій `tree_edge` та `gray_target`, її зміст полягає у створенні єдиної події, що буде ініціюватися для кожного ребра неорієнтованого графа лише раз. Це гарантується тим, що `forward_edge` викликається лише коли ціль ребра має не чорний колір — відповідно з іншого кінця ребра, інцидентні до поточного ребра, ще не оброблялися. Ця подія буде використана при створенні оператора побудови реберного графа.

Однією з найважливіших змін реалізації BFS порівняно з варіантом з BGL стало застосування `if constexpr (requires ...)` виразів перед кодом виклику методів-подій. Зроблено це для перевірки наявності відповідного метода у переданого в алгоритм обходу Відвідувача. Виглядає така перевірка як показано в лістингу 3. Це дозволяє при створенні користувачем власного Відвідувача уникнути вимоги щодо реалізації методів для всіх BFS подій, або спадкування від базового Відвідувача, як це було в BGL, оскільки компілятор генерує виклики лише для фактично реалізованих подій. Єдиним недоліком такого підходу є незначне збільшення коду функції `breadth_first_visit`, адже таку перевірку треба прописати перед кодом виклику кожного метода-події.

```
if constexpr (requires { visitor.examine_vertex(u, g); }) {
    visitor.examine_vertex(u, g);
}
```

Лістинг 3: Перевірка наявності метода

Оскільки черга обходу наповнюється суміжними до обійдених вершинами обхід охопить лише одну компоненту зв'язності. Для того, щоб обхід пройшов всі компоненти зв'язності, виклик `breadth_first_visit` відбувається через функцію `breads_first_search`. Вона приймає всі ті самі аргументи, що і `breadth_first_visit`, крім стартової вершини. Фактично весь код функції це перебір вершин з умовою, що якщо вершина v відмічена білим кольором (ще не відвідувалась), вона передається як стартова вершина у функцію `breadth_first_visit`. Після виконання BFS обходу продовжується цикл перевірок, наступна знайдена біла вершина є початком нової компоненти зв'язності, для якої запускається новий обхід.

Будова алгоритму DFS обходу вглибину багато в чому схожа. Аналогічні функції `depth_first_visit` та `depth_first_search` з таким же набором аргументів. В середині обходу відбуваються ті самі перевірки наявності метода у Відвідувача на етапі компіляції через `if constexpr (requires ...)` вирази. Сам алгоритм реалізовано на базі стеку (LIFO), а не рекурсії — цю ідею було запозичено із BGL проте з деякими відмінностями. Стек викликів рекурсії симулюється через `std::vector<VertexFrame>`, де об'єкт `VertexFrame` зберігає стан обходу у полях: `vertex_descriptor u` — відповідає поточну вершину обходу, а поле `size_t current_edge_idx` — за порядковий номер суміжної до u вершини, в яку пішло заглиблення. Такий підхід разом із використанням діапазонів дозволив позбутися зберігання ітераторів у `VertexFrame` як це було в BGL. Проте це призвело до обмеження на метод `g.out_edges(u)`, зобов'язуючи його повертати тип, що задовольняє концепт Random Access Range (підтримує оператор "`[]`"). Проте, для більшості структур на базі списку суміжності, де ребра зберігаються в цільних блоках пам'яті, такий концепт підтримується нативно.

Іншою зміною в порівнянні з BGL є використання єдиного алгоритми обходу в глибину для орієнтованих і неорієнтованих графів. Оскільки у цій роботі не розглядаються мультиграфи, щоб обхід у глибину не повернувся назад, виконується перевірка батьківської вершини через попередній запис у стеку. Це дозволило уникнути реалізації окремого DFS алгоритму із розфарбуванням вершин для неорієнтованого графа, як це було зроблено в BGL, щоб не заборонити обходу в глибину повернення у вершину назад по паралельному ребру.

2.3 Модернізація архітектури графових операторів засобами сучасного C++

Використання в архітектурі BGL патернів Відвідувач, Впровадження залежностей, Фасад мало явні переваги, проте тогочасна реалізація страждала від обмежень старого стандарту C++98. У цьому підрозділі розглянуто модифікації згаданих трьох патернів реалізованих у цій роботі згідно стандарту C++20.

Бібліотека BGL вимагала від користувача при створенні Відвідувачів спадкуватися від спеціальних базових Відвідувачі. Це робилося для того, щоб алгоритм міг викликати всі події обходу і компілятор не видавав помилок. У запропонованій архітектурі від такого примусового успадкування повністю відмовилися. Замість цього використовується перевірка наявності методів під час компіляції на стороні алгоритмів обходу, як було описано в попередньому розділі. Якщо методу немає, компілятор просто ігнорує цей виклик. Це робить класи Відвідувачів легкими, незалежними і не обтяженими зайвими ієрархіями класів.

Одним із синтаксичних досягнень, що суттєво спростило роботу з Відвідувачами, стало використання механізму виведення аргументів шаблону класу (Class Template Argument Deduction, CTAD), доданого у стандарті C++17. У класичному C++ розробнику доводилося або вручну прописувати довгі списки типів у кутових дужках, або створювати допоміжні функції-генератори.

Завдяки спеціальному "посібнику з виведення типів" (deduction guide), реалізованому для кожного Відвідувача, створення об'єкта Відвідувача відбувається автоматично на основі переданих у конструктор контейнерів. Це не лише робить код лаконічнішим, але й усуває ризик помилки при ручному вказанні типів.

Ще однією архітектурною зміною є відмова від ітераторів при передачі контейнерів результатів та проміжних станів. У розробленій бібліотеці оператори працюють з цілісними діапазонами (Ranges), які перевіряються на етапі компіляції за допомогою концептів C++20 (наприклад, `WritableMask` чи `IntegerVertexMap`). Це робить інтерфейси безпечнішими та лаконічнішими: Відвідувач приймає один об'єкт-контейнер (зазвичай через універсальне посилання), а компілятор гарантує, що цей об'єкт підтримує необхідні операції (наприклад, довільний доступ до елементів чи можливість запису).

Патерн Впровадження залежностей (Dependency Injection) у розробленій архітектурі реалізовано з використанням семантики переміщення (move semantics) та механізму ідеальної передачі аргументів (perfect forwarding) (детальніше про них у [15] та [17]). На відміну від бібліотеки BGL, яка використовує додаткові абстракції, наприклад `PropertyMap`, запропонована архіте-

ктура передає необхідні залежності безпосередньо через конструктори Відвідувачів за допомогою універсальних посилань (`T&&`). Використання функції `std::forward` дозволяє алгоритму ефективно приймати як уже існуючі буфери пам'яті (lvalue), уникаючи їхнього копіювання, так і тимчасові об'єкти (rvalue), автоматично застосовуючи до них операцію переміщення. Такий підхід усуває накладні витрати на копіювання об'ємних тимчасових структур для збереження проміжних результатів (наприклад, масиви батьківських вершин чи відстаней), що є критично важливим для збереження швидкодії та раціонального використання пам'яті під час обробки графів великої розмірності.

Цей патерн також надає користувачу ще одну важливу перевагу — можливість самостійно обирати баланс між використанням оперативної пам'яті та швидкістю роботи алгоритму. Наочним прикладом цього є реалізовані оператори двочастковості та пошуку точок зчленування.

Для запису результату (маски вершин) оператор приймає будь-який контейнер, здатний зберігати бінарні значення. Якщо користувач передає аргументом `std::vector<bool>`, стандартна бібліотека автоматично застосовує оптимізацію пам'яті, зберігаючи вісім булевих значень в одиному байті. Це значно заощаджує оперативну пам'ять, але вимагає додаткового процесорного часу на бітові операції при кожному зверненні до елемента. Якщо ж швидкодія є критичнішою за пам'ять, розробник може передати контейнер типу `std::vector<uint_8>`. Тоді обсяг використаної пам'яті зросте у вісім разів, проте доступ до значень відбуватиметься миттєво без додаткових обчислень. Так архітектура не нав'язує жорстких структур даних, а дозволяє гнучко підлаштовувати алгоритм під пріоритети користувача.

Хоча впровадження залежностей (DI) надає гнучкість, воно водночас змушує користувача власноруч створювати тимчасові контейнери, які не мають прямого відношення до кінцевого результату. Для вирішення цієї проблеми у бібліотеці реалізовано патерн Фасад у вигляді високорівневих функцій-операторів. Ці функції займаються виділенням пам'яті під проміжні структури, створенням об'єкта Відвідувача та запуском алгоритму обходу.

Водночас варто зазначити, що у цій роботі було свідомо відмовлено від реалізації механізму іменованих параметрів (Named Parameters), який вико-

ристовується у BGL. Хоча іменовані параметри дозволяють передавати аргументи у довільному порядку, але реалізація такого механізму виходить за рамки мети цієї роботи.

Підсумовуючи, хоча ідея використання цих патернів для графових бібліотек були закладені ще розробниками BGL, застосування сучасних інструментів C++ (концептів, діапазонів, семантики переміщення та виведення типів) дозволило створити архітектуру для графових операторів зрозумілу для користувача та зручну у підтримці для розробника. Конкретні реалізації алгоритмів на базі цієї архітектури детально розглядаються в наступному розділі.

2.4 Сучасний патерн Компонувальник для спільного обходу графів

Виконання декількох незалежних графових операторів зазвичай вимагає проведення окремих обходів графа для кожного з них. Однак для великих структур даних такий підхід є затратним через дублювання роботи з доступу до пам'яті та часті промахи кешу процесора (cache misses) [13, 10]. Одним із рішень цієї проблеми є ідея спільного обходу, яка пропонує виконувати всі необхідні операції під час єдиного обходу графа.

Як зазначалося в підрозділі 1.4, базова ідея вирішення цієї задачі була виконана розробниками бібліотеки за допомогою утиліти `multi_visitor`. Вона дозволила об'єднувати декілька об'єктів-Відвідувачів через статичний варіант патерна Компонувальника і послідовно делегувати їм події обходу. Оскільки розгортання викликів у BGL відбувається на етапі компіляції, такий підхід позбавлений накладних витрат на динамічну диспетчеризацію.

У цій роботі таку архітектурну ідею було запозичено та переосмислено з використанням засобів сучасного стандарту C++20. Якщо класична реалізація `multi_visitor` у BGL спиралася на громіздкі механізми метапрограмування минулих стандартів (наприклад, рекурсивні шаблони та списки типів `boost::mpl`), то розроблений у цій роботі клас `BfsCompositeVisitor` досягає тієї ж абстракції з нульовими накладними витратами (zero-overhead abstraction), але з використанням зрозумілих сучасних мовних конструкцій.

Завдяки використанню варіативних шаблонів (variadic templates), клас `BfsCompositeVisitor` приймає довільну кількість типів і зберігає їхні екземпляри у стандартній структурі `std::tuple`, що дозволяє компілятору ідентифікувати типи всіх Відвідувачів заздалегідь. А для реалізації безпечної статичної диспетчеризації подій було застосовано механізм концептів C++20. У спеціальному просторі імен Композита визначено набір концептів, що перевіряють наявність відповідних методів у переданого Відвідувача. Приклад такого концепту наведено у лістингу 4.

```
namespace composite {
template <typename V, typename Edge, typename G>
concept HasExamineEdge = requires(V vis, Edge e, const G & g) {
vis.examine_edge(e, g);
};
}
```

Лістинг 4: Концепт для перевірки наявності методу

Завдяки цим концептам, алгоритму не потрібно вимагати від користувацьких Відвідувачів успадкування від базових класів із порожніми віртуальними методами (як це вимагається у класичному об'єктно-орієнтованому патерні).

Сама ж механіка перехоплення та делегування подій, яка у BGL вимагала складного шаблонного коду, у розробленій архітектурі реалізується значно простіше завдяки використанню `std::apply`, виразів згортання (fold expressions) та `if constexpr`. Як показано у лістингу 5, коли відбувається подія обходу, Компонувальник застосовує лямбда-вираз до кожного елемента кортежу `_visitors`.

Під час компіляції функція `std::apply` розгортає параметри кортежу, а вираз згортання `(call_if_exists(vis_args), ...)` перетворюється на лінійну послідовність викликів. А завдяки `if constexpr`, компілятор фізично не генерує код для тих Відвідувачів, які не мають реалізації відповідного методу. У результаті створюється монолітний оптимізований машинний код, що повністю відтворює логіку класичного `multi_visitor`, але є значно простішим у підтримці та розширенні.

```
template <BaseGraph G>
void examine_vertex(typename G::vertex_descriptor u, const G& g){
    auto call_if_exists = [&](auto& vis) {
        if constexpr (composite::HasExamineVertex<decltype(vis),
           decltype(u), G>) {
            vis.examine_vertex(u, g);
        }
    };
    std::apply([&](auto&... vis_args) {
    (call_if_exists(vis_args), ...); }, _visitors);
}
```

Лістинг 5: Диспетчеризація події всередині Компонувальника

Для зручності інстанціювання композитного Відвідувача без явного вказування довгих шаблонних типів, за аналогією з BGL, було реалізовано допоміжну функцію `make_bfs_composite`. Вона використовує ідеальну передачу аргументів (perfect forwarding) та автоматичне виведення типів.

Так, застосування засобів C++20 дозволило зберегти всі переваги класичного підходу спільного обходу, суттєво зменшивши при цьому синтаксичну складність коду та обсяг внутрішніх абстракцій бібліотеки.

3 ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ГРАФОВИХ ОПЕРАТОРІВ У КОНТЕКСТІ СПІЛЬНОГО ОБХОДУ

3.1 Оператори, що підтримують спільний обхід

У даному розділі представлено реалізацію графових операторів, логіка яких дозволяє використовувати для них механізми спільного обходу. Вибір наведених операторів зумовлений необхідністю валідації розробленого програмного каркаса: оператори двочастковості та пошуку точок зчленування слугують для перевірки коректності адаптації класичних алгоритмів відповідно до стандарту C++20, а оператор побудови реберного графа є оригінальною розробкою, що демонструє розширюваність системи.

Оператор перевірки двочастковості графа можна формалізувати як відображення $b : \mathcal{G} \to (\{1\} \times \Phi) \cup \{(0, \varnothing)\}$, де $\mathcal{G}$ — множина всіх можливих графів, а Φ — множина функцій розфарбування $\varphi : V \to \{c_1, c_2\}$ (c_1, c_2 — ідентифікатори часток). Для довільного графа $G = (V, E)$ оператор визначає чи є граф двочастковим та будує відповідне відображення вершин у множину часток, якщо граф двочастковий:

$$b(G) = \begin{cases} (1, \varphi), & \text{якщо } \forall \{u, v\} \in E : \varphi(u) \neq \varphi(v) \\ (0, \varnothing), & \text{в іншому випадку} \end{cases}$$

На відміну від класичної реалізації у бібліотеці BGL, де цей алгоритм базується на обході в глибину (DFS) та композиції декількох дуже простих Відвідувачів, у цій роботі оператор реалізовано на основі пошуку в ширину (BFS) за допомогою єдиного Відвідувача `BipartiteGraphVisitor`. Така зміна будови зумовлена не стільки пошуком асимптотичних переваг, скільки уніфікацією стилю та побудовою наочного прикладу BFS Відвідувача.

Конструктор `BipartiteGraphVisitor` приймає `partition_map` — контейнер, тип якого має задовольняти концепт `WritableMask`. Цей концепт вимагає наявності інтерфейсу для запису бінарних значень за індексом. Як і зазначалося у попередніх розділах, таке застосування патерна Впроваджен-

ня залежностей дозволяє користувачу самому вирішувати чи буде контейнер з результатом заощаджувати пам'ять завдяки оптимізації векторів стандартної бібліотеки чи збереже швидкий доступ до елементів.

Логіка подій майже не має відмінностей з BGL.

- Стартова вершина: Задається частка стартової вершини поточної компоненти зв'язності.
- Деревне ребро: Недослідженому кінцю ребра призначається частка протилежна від частки поточної вершини.
- Недеревне ребро: Перевіряється належність обох кінців ребра до різних часток. Якщо виявляється, що частки збігаються, це свідчить про наявність у графі циклу непарної довжини. У такому разі алгоритм миттєво припиняє роботу, генеруючи виняток (`std::exception`).

Реалізовано Фасад-функцію `bipartite_graph_operator`. Вона автоматично створює всі необхідні тимчасові структури даних, інстанціює об'єкт Відвідувача та запускає процес обходу графа, надаючи користувачеві простий інтерфейс отримання результату.

Оператор пошуку точок зчленування графа можна формалізувати як відображення $a : \mathcal{G} \to \mathcal{P}(V)$, де $\mathcal{P}(V)$ — множина всіх підмножин V, а $\omega(G)$ — кількість компонент зв'язності графа. Для довільного графа знаходить множину точок зчленування графа $G = (V, E)$:

$$a(G) = \{v \in V \mid \omega(G \setminus \{v\}) > \omega(G)\}$$

Оператор реалізовано із запозиченням ключових ідей BGL у адаптації алгоритму Тар'яна [22] до виконання під час обходу в глибину (DFS). Проте у цій роботі оператор спрямований виключно на пошук точок зчленування для валідації системи та побудови наочного прикладу DFS Відвідувача.

Конструктор `ArticulationPointsVisitor` приймає `mask` — контейнер, тип якого має задовольняти описаний вище концепт `WritableMask`. Варто зазначити, що Відвідувач оператора повертає не множину точок зчленування, а відображення вершин у бінарний статус, що відповідає чи є вершина точкою зчленування. Зроблено це для уникнення повторів, адже деякі вершини

зчленування знаходяться алгоритмом декілька разів. Сама ж множина буде формуватися Фасад-функцією, щоб надати користувачеві можливість уникнути зайвих обчислювальних витрат, якщо йому достатньо відображення. Наступні три аргументи конструктора відповідають за структури даних алгоритму Тар'яна. Їхні типи мають задовольняти концепт `IntegerVertexMap`, що вимагає наявності інтерфейсу для доступу до цілочисельних значень за індексом вершини:

- `dtime` — відображення вершин у множину часу (кроків) обходу вершин. У відповідність вершині записується номер кроку обходу в глибину на якому вперше було відвідано вершину.

- `low` — відображення вершин у множину часу (кроків) обходу вершин. У відповідність вершині записується найменше значення кроку обходу в глибину на якому було відвідано вершину.

- `parents` — відображення вершин у множину вершин. У відповідність вершині записується її деревний предок.

Логіка подій є спрощеним варіантом з BGL.

- Стартова вершина: Перевіряється достатність розмірів усіх переданих контейнерів (діапазонів). Стартова вершина задається сама собі як предок.

- Відкриття вершини: Задається час відкриття вершини та збільшується значення годинника.

- Деревне ребро: Не дослідженому кінцю ребра (ціль) поточна вершина (джерело) задається як предок. Підраховується кількість гілок дерева обходу від стартової вершини компоненти зв'язності.

- Зворотне ребро: Перевіряється чи час відкриття цілі менший ніж мінімальний у джерела. Зменшуємо мінімальний час джерела, якщо умова справджується.

- Завершення вершини: Якщо це стартова вершина обходу в компоненті зв'язності, вона позначається як точка зчленування, якщо від неї було

нараховано більше однієї гілки дерева обходу. Для інших вершин оновлюється мінімальний час предка, якщо мінімальний час нащадка (вершини на завершенні) менший. Вершина-предок позначається точкою зчленування, якщо її час відкриття менший за мінімальний у нащадка. Тобто нащадок не зміг досягти предків предка іншим шляхом.

Реалізовано дві Фасад-функції: `articulation_points_mask_operator` та `articulation_points_operator`. Перша автоматично створює всі необхідні тимчасові структури даних, інстанціює об'єкт Відвідувача та запускає процес обходу графа, вимагаючи від користувача лише граф та контейнер-відображення для запису результатів. Друга функція повертає конкретно підмножину вершин зчленування, тому в якості аргументів вимагає граф та діапазон для запису дескрипторів вершин. В її коді викликається функція `articulation_points_mask_operator`, потім, ітеруючись по заповненому нею контейнеру, вихідний діапазон наповнюється вершинами відміченими як точки зчленування.

Оператор побудови реберного графа можна формалізувати як відображення $L : \mathcal{G} \to \mathcal{G}$, де $\mathcal{G}$ — множина всіх можливих графів. Для довільного графа $G = (V, E)$ ставить у відповідність граф $L(G) = (E, E')$, де множина ребер E' визначається як:

$$E' = \{\{e_1, e_2\} \mid e_1, e_2 \in E, e_1 \cap e_2 \neq \varnothing\}$$

У роботі представлено повністю авторську реалізацію оператора під час BFS обходу, яка базується на властивостях реберного графа. Оператор працює виключно для неорієнтованих графів, при спробі виконання для орієнтованого графа видасть помилку. В конструкторі приймає першим аргументом контейнер `incident_edges`, що має задовольняти `VertexCollectionMap`. Цей концепт вимагає діапазон з доступом за індексом, який зберігає діапазон цілочисельних значень. Фактично це структура даних типу "масив масивів" дескрипторів вершин. Наступні два аргументи мають задовольняти концепти `AddVertexCallback` та `AddEdgeCallback` і, як зрозуміло з типів, бути функціями додавання ребер та вершин.

Логіка подій така:

- Стартова вершина: Перевіряється відповідність розміру `incident_edges` до кількості вершин графа. Також перевіряється чи є граф неорієнтованим.

- Переднє ребро: Створюється вершина реберного графа, вона додається в `incident_edges` як інцидентне ребро для вершини-джерела та вершини-цілі.

- Завершення вершини: Створюється ребра реберного графа між всіма парами вершин реберного графа, що позначені як ребра, інцидентні певній вершині у графі-прообразі.

Важливо зазначити, що алгоритм можна спростити, якщо вимагати у вхідного графа наявність `edge_id` у дескриптора ребер для відображення ребер-прообразів у множину вершин реберного графа. Однак метою даної розробки було створення універсального оператора, що належно працюватиме із графами без підтримки властивостей ребер, уникаючи при цьому використання ресурсозатратних контейнерів, наприклад, хеш-таблиці з високими накладними витратами часу на кшталт `std::unordered_map`.

Реалізовано дві однойменні Фасад-функції `line_graph_operator`. Одна приймає граф та функції створення вершини та ребра, про які йшлося вище. Створює, резервує місце та передає у Відвідувач множину для інцидентних ребер. Створює Відвідувач та запускає обхід. Інша функція, що є довизначенням (Overloading) попередньої, створена для зручного використання із реалізованим у роботі класом `AdjacencyList`. Вона приймає лише граф, а повертає пару кількість вершин та множину ребер, яку приймає конструктор `AdjacencyList`.

3.2 Обмеження реалізації через патерн Відвідувач

Ефективність адаптації графових операторів у архітектуру на базі обходу (BFS, DFS) із застосуванням патерна Відвідувач визначається низкою фундаментальних теоретичних обмежень. Не кожен оператор $\mathcal{F} : \mathcal{G} \to \mathcal{X}$ може бути ефективно реалізований у межах такого архітектурного підходу. На основі аналізу реалізованих у роботі та у BGL опрераторів виділено три

ключові критерії (характеристики), які визначають сумісність оператора з виконанням під час подіє-орієнтованого обходу:

Асимптотична сумісність обчислень та локальна обчислюваність. Класичні алгоритми обходу графа мають лінійну часову складність $O(|V| + |E|)$. Для того щоб спільний обхід декількох операторів зберігав цю складність, локальна обчислювальна складність всередині кожної точки події для поточної вершини u або ребра e має бути обмеженою. Обчислення у точці події можуть виконуватися за амортизований час $O(1)$ або за час, пропорційний ступеню поточної вершини $O(\deg(u))$. Якщо логіка оператора вимагає виконання локальних обходів (підграфів довільного радіуса) на кожному кроці, це призводить до руйнування лінійної складності базового обходу та створює надлишкові накладні витрати пам'яті й часу, які неможливо спільно використовувати іншими операторами в рамках запропонованої архітектури.

Послідовна інформаційна локальність. Патерн Відвідувач інкапсулює логіку, яка виконується над графом дискретно та послідовно в часі. Оператор задовольняє критерію послідовної локальності, якщо для прийняття обчислювального рішення на кроці t йому достатньо інформації про поточний елемент (вершину u або ребро e), його безпосередній окіл $N(u)$ та поточний внутрішній стан самого Відвідувача S_t. Якщо математична логіка оператора вимагає одночасного (глобального) аналізу структурних властивостей графа у його просторово віддалених частинах, такий оператор не може бути реалізований через класичний послідовний обхід.

Однопрохідна повнота обчислень. Цей критерій визначає здатність оператора збирати всю необхідну інформацію та виконувати всі необхідні дії за один повний прохід по структурі графа. Оператор має працювати як детермінований автомат, який на кожному кроці оновлює свій стан на основі поточної події та локального околу $N(u)$. Оператори, що порушують цей критерій і вимагають багаторазових ітерацій по топології графа, позбавлені можливості використання єдиного спільного BFS/DFS обходу з іншими

операторами.

3.3 Адаптація алгоритму ILIGRA для побудови прообразу реберного графа через патерн Відвідувач

У попередньому підрозділі було сформульовано набір критеріїв сумісності, яким має відповідати графовий оператор для інтеграції у механізм спільного обходу через патерн Відвідувач. З метою практичної валідації цих умов на задачі високого рівня складності було обрано оператор побудови прообразу реберного графа $L^{-1}(G)$.

Оператор побудови прообразу реберного графа можна формалізувати як відображення $L^{-1} : \mathcal{G} \to (\{1\} \times \mathcal{G}) \cup \{(0, \varnothing)\}$, де $\mathcal{G}$ — множина всіх можливих неорієнтованих графів. Для довільного вхідного графа $H \in \mathcal{G}$ оператор визначає, чи належить він до класу реберних графів, і в разі позитивної відповіді — відновлює його граф-прообраз:

$$L^{-1}(H) = \begin{cases} (1, G), & \text{якщо } \exists G \in \mathcal{G} : L(G) \cong H \\ (0, \varnothing), & \text{в іншому випадку} \end{cases}$$

Математичним фундаментом, що гарантує коректність роботи такого оператора, є теорема Вітні про ізоморфізм графів [23]. Згідно з нею, якщо H є зв'язним реберним графом, то його прообраз G існує та відновлюється однозначно з точністю до ізоморфізму. Єдиним винятком із цього правила є граф K_3 (трикутник), який є реберним графом для двох неізоморфних графів-прообразів: повного графа K_3 та графа-зірки $K_{1,3}$. Для обробки цього випадку алгоритм має бути детермінованим і повертати заздалегідь визначений базовий варіант (зазвичай $K_{1,3}$).

Класичне рішення цієї задачі — алгоритм Русопулоса [18]. Але для цієї роботи також важлива сумісність із подіє-орієнтованим підходом, а цей алгоритм хоча і має лінійну асимптотичну складність, проте спирається на теорему Крауса [12] про розклад графа на кліки. Це призводить до порушення критерію однопрохідної повноти обчислень, оскільки алгоритм концептуально вимагає двох етапів: спочатку розподілення вершин і ребер на кліки

(розкладу Крауса), і лише потім — побудови прообразу на їх основі. Виконувати перший етап під час спільного обходу поверхнево видається можливим, але у такому разі алгоритм доведеться розбити на дві неповноцінні частини, і лише одну передавати у Компонувальник. Ця ідея не перевірялася на практиці, оскільки було знайдено швидший алгоритм, що має менше проблем для спільного обходу та не базується на Теоремі Крауса.

Сучасний алгоритм ILIGRA (Iterative Line Graph Recognition Algorithm) [16], розроблений у 2015 році, уникає використання розкладу на кліки. Він ітеративно відновлює прообраз спираючись на базові властивості реберного графа. Крім першого кроку, алгоритм оперує виключно локальними даними обходу. Це робить алгоритм структурно сумісним із виведеними критеріями та дозволяє інкапсулювати його логіку в події патерна Відвідувач та ефективно застосувати Компонувальник для спільного обходу.

Концептуально алгоритм ILIGRA базується на принципі локального ітеративного відновлення. Перший крок є найскладнішим і найбільш затратним у алгоритмі. На цьому етапі алгоритм бере довільну стартову вершину реберного графа і виконує глибокий аналіз її оточення, іноді виходячи за локальний окіл. Мета цього кроку — правильно класифікувати суміжні вершини реберного графа та визначити множини сусідів для обох кінців ребра-прообразу у графі-прообразі. Як тільки сусідів першого ребра-прообразу розподілено по кінцях, алгоритм переходить у лінійну ітеративну фазу. Рухаючись графом далі, він аналізує перетини околів нових вершин із вже обробленою частиною структури, що дозволяє однозначно почергово приєднувати нові ребра-прообрази до вже визначених.

Класична реалізація. Під час практичної імплементації класичного варіанту алгоритму було проведено детальний аналіз першоджерела. Хоча автори дослідження ILIGRA надали відкритий C++ код, було виявлено суттєві архітектурні розбіжності між строгим математичним описом у статті та їхньою практичною реалізацією. З метою збереження алгоритмічної чистоти та гарантії математичної коректності, у цій роботі було прийнято свідоме рішення імплементувати узагальнений алгоритм суворо за його формальним текстовим описом. З оригінального коду було запозичено лише окремі ефе-

ктивні низькорівневі трюки, які не суперечили загальній архітектурі.

Функція `iligra` приймає на вхід реберний граф `H`, та дві функції, що задовольняють концепти `AddVertexCallback` та `AddEdgeCallback` для повернення графа-прообразу. Оскільки формальний опис ILIGRA активно використовує операції над множинами (перетини, об'єднання, різниці), для реалізації складного кроку ініціалізації було застосовано стандартні асоціативні контейнери `std::set`. Це забезпечило ідеальну семантичну відповідність математичній моделі статті. В той же час основний ітеративний цикл алгоритму, на який припадає головне обчислювальне навантаження, було оптимізовано з використанням контейнерів `std::vector`, що забезпечило високу просторову локальність та швидкість доступу під час обходу вершин. Повна заміна `std::set` на оптимізовані контейнери є простором для подальшого покращення реалізації поза межами даного дослідження.

Реалізація через Відвідувач. Адаптація алгоритму ILIGRA через обхід в ширину із застосуванням Відвідувача базується на ітеративній природі відновлення графа-прообразу. Основна архітектурна складність такої адаптації виникає на етапі ініціалізації. Окрім генерації першого ребра прообразу, цей етап частково визначає належність його сусідів. У класичній реалізації алгоритму для таких напіввизначених вершин (ребер-прообразів) використовується спеціалізована черга. Натомість інваріантність черги стандартного BFS-обходу унеможливлює динамічне вилучення напіввизначених на ініціалізації вершин.

Для розв'язання цієї проблеми в архітектуру Відвідувача впроваджено використання маркерного стану (константа `UNASSIGNED`) у контейнері відображення `v_ln` вершин реберного графа у один із їхніх кінців-прообразів. Введення маркерних перевірок на початку методів-подій забезпечило Відвідувачу здатність коректно ідентифікувати частково опрацьовані вершини та уникати їх хибної повторної ініціалізації, зберігаючи при цьому цілісність BFS.

Незначну нелокальність стартового кроку було прийнято як компроміс, оскільки подальша ітеративна частина алгоритму повністю задовольняє критерії з попереднього підрозділу, завдяки встановленню, що стандартні стани

(кольори) вершин при обході в ширину прямо корелюють із рівнями визначеності ребер-прообразів. Саме ця властивість дозволила зробити ефективну декомпозицію логіки ILIGRA на такі події класу `InverseLineGraphVisitor`:

- Стартова вершина: Відповідає за етап ініціалізації. Під час цієї події алгоритм бере першу вершину реберного графа `n_1`, створює для неї два кінці `v_1` та `v_2` відповідного ребра `l_n1` в прообразі, і розподіляє суміжні до `n_1` вершини реберного графа між цими кінцями. Напіввизначені вершини записуються у `v_ln`, щоб подальші події обходу обробили їх коректно завдяки маркерним перевіркам.

- Розгляд вершини: Сигналізує про перехід до нової вершини-прообразу. Спочатку перевірка, чи вершина не була напіввизначена через суміжність до `n_1`. Тоді створюється нова вершина графа-прообраза `current_v`. Завершується ребро-прообраз на основі раніше визначеного іншого кінця цього ребра в `v_ln` та `current_v`.

- Деревне ребро: Відповідає за первинне опрацювання білих вершин обходу. Під час цієї події ребру-прообразу вершини цілі призначається перший кінець — `current_v` створена в попередній події. Крім того, вершина ціль додається до тимчасового контейнера `C` для подальшої валідації.

- Сіра ціль: Опрацьовує альтернативні переходи до вже напіввизначених ребер-прообразів сірих вершин. Якщо виявляється, що обидва кінці ребра-прообразу вже мають різні значення у `v_ln`, ця подія завершує (створює) ребро-прообраз. Вершина ціль аналогічно додається до контейнера `C`.

- Завершення вершини: Виконує роль валідатора реберності. Коли всі сусіди поточної вершини оброблені, множина вершин у контейнері `C` зобов'язана утворювати кліку. Якщо перевірка `is_clique` дає негативний результат, алгоритм миттєво перериває обхід генеруванням винятку, сигналізуючи про нереберність вхідного графа.

Для зручного виклику Відвідувача та перехоплення винятків нереберності реалізовано Фасад-функцію `inverse_line_graph_operator`. Вона сама створює та виділяє пам'ять на допоміжні структури, наприклад, `v_ln` та трансформує винятки нереберності графа у булевий результат.

Другий, високорівневий Фасад для використання із розробленим у роботі класом `AdjacencyList` реалізовано як довизначення обох класичної та Відвідувач-реалізації. Ці функції приймають лише вхідний реберний граф і повертають необхідні конструктору `AdjacencyList` дані для створення об'єкта у місці виклику. Ці функції будуть активно застосовуватися в проведенні експериментів з наступного розділу.

Підсумовуючи, успішна адаптація алгоритму ILIGRA підтверджує, що через обхід із застосуванням Відвідувача можливо реалізувати не лише базові алгоритми збору графових метрик, але й складні ітеративні топологічні перетворення. У ширшому контексті всього розділу було продемонстровано, що за умови дотримання виведених критеріїв алгоритмічної сумісності, різнопланові оператори на графах можуть бути уніфіковані до єдиної архітектури із підтримкою спільного обходу. Така архітектурна стандартизація за допомогою сучасних засобів C++20 не лише забезпечує високу модульність та інкапсуляцію коду, але й закладає фундаментальні передумови для глобальної оптимізації обчислювальної продуктивності, експериментальне дослідження якої наведено у наступному розділі роботи.

4 ОБЧИСЛЮВАЛЬНІ ЕКСПЕРИМЕНТИ ТА ОЦІНКА ЕФЕКТИВНОСТІ

4.1 Методика проведення обчислювальних експериментів

Метою проведення обчислювальних експериментів є емпірична перевірка ефективності запропонованих архітектурних рішень (зокрема спільного обходу через патерн Компонувальник) та оцінка швидкодії адаптованого алгоритму ILIGRA порівняно з класичними підходами. Для забезпечення об'єктивності та відтворюваності результатів тестування проводилося в контрольованому середовищі із фіксованою конфігурацією апаратного та програмного забезпечення.

Апаратне та програмне забезпечення. Усі обчислювальні експерименти виконувалися на базі портативної робочої станції (модель MSI GV72 8RD) з такими апаратними характеристиками:

Процесор: Intel Core i7-8750H (базова частота 2.20 ГГц, 6 фізичних ядер, 12 логічних потоків). Важливим для специфіки графових обчислень є наявність 9 МБ кеш-пам'яті третього рівня (L3 Smart Cache), що суттєво впливає на локальність даних під час обходу великих структур.

Оперативна пам'ять: 32 ГБ стандарту DDR4 із тактовою частотою 2400 МГц.

Програмне середовище базувалося на 64-розрядній операційній системі Windows 10. Компіляція коду здійснювалася компілятором MSVC (Microsoft Visual C++) з підтримкою стандарту C++20. Збірка виконувалася у конфігурації "Release" із застосуванням прапорця максимальної оптимізації швидкодії "`/O2`" (Maximize Speed), що дозволило компілятору повною мірою застосувати механізми розгортання циклів (loop unrolling), вбудовування функцій (inlining) та вирізання мертвого коду (dead code elimination) під час розкриття шаблонів Відвідувача та Компонувальника.

Методика вимірювання часу. Оскільки графові алгоритми на сучасних процесорах виконуються за долі секунди, для точного вимірювання часу виконання застосовувався `std::chrono::high_resolution_clock` — системний таймер високої роздільної здатності зі стандартної бібліотеки C++.

Для мінімізації впливу фонових процесів операційної системи на чистоту експерименту, тести запускалися серіями у різні проміжки часу. У межах однієї серії алгоритми виконувалися у циклі N разів (де $N \geq 30$). Перед початком фіксації часу виконувався так званий "прогрівочний" (warm-up) запуск алгоритму для завантаження структур графа у кеш процесора (L1-L3), результати якого не враховувалися.

Для порівняння альтернативних підходів (наприклад, окремого виконання операторів та зі спільним обходом) використовувався один і той самий екземпляр графа, завантажений в оперативну пам'ять, що гарантувало абсолютно ідентичні початкові умови. Фінальним результатом вважалося середнє арифметичне (або медіанне) значення часу серед усіх ітерацій циклу.

Об'єкти тестування. Варто наголосити, що до програми обчислювальних експериментів не було включено оператори перевірки двочастковості графа та пошуку точок зчленування. Логіка алгоритмів в їх основі запозичена з BGL, а зроблена у цій роботі сучасними засобами C++ реалізація не передбачає отримання додаткового обчислювального прискорення (runtime gain). Зазначені оператори використовувалися виключно як інструменти архітектурної валідації розробленої бібліотеки, зокрема для перевірки коректності компіляції концептів, інтерфейсів взаємодії Відвідувачів та механізмів мапування станів. Натомість фокус обчислювальних експериментів зміщено на оператор побудови реберного графа $L(G)$ на основі запропонованого алгоритму та оператор адаптованого ILIGRA ($L^{-1}(G)$). Вони оперують масивними структурами даних і дозволяють емпірично оцінити ефект локальності даних під час спільного обходу.

Набори тестових даних. Тестування проводилося на згенерованих наборах графів, що дозволило гнучко контролювати їхні метрики: кількість вершин $|V|$, кількість ребер $|E|$ та щільність розподілу зв'язків. Оскільки ал-

горитм ILIGRA орієнтований на розріджені графи (sparse graphs), а ефективність спільного обходу сильно залежить від промахів кешу (cache misses), для обчислювальних експериментів було сформовано два класи структур. Обидва класи графів було реалізовано на базі моделі Барабаші–Альберта [5] за допомогою оптимізованого алгоритму Batagelj–Brandes [6]:

Розріджені графи генерувалися за мінімальних значень параметра приєднання ($m \leq 3$). Така топологія утворює класичні безмасштабні мережі (scale-free networks) з низькою щільністю, що дозволяє оцінити граничну ефективність алгоритму ILIGRA у цільовому для нього середовищі.

Щільні графи генерувалися шляхом суттєвого масштабування параметра m відносно кроку додавання вершин. Це дозволило отримати структури з високою щільністю зв'язків, що підвищило навантаження на пам'ять під час обходів та дозволило наочно продемонструвати переваги використання спільного обходу.

Варто зазначити, що оскільки алгоритм ILIGRA вимагає на вхід реберний граф (який не містить заборонених підграфів за теоремою Бейнеке [7]), для підготовки тестових даних спочатку генерувався випадковий граф-прообраз, після чого до нього застосовувався оператор побудови реберного графа $L(G)$. Час роботи цього оператора не враховувався у фінальних замірах швидкодії оператора побудови прообразу реберного графа $L^{-1}(G)$, оскільки цей процес включається до етапу ініціалізації тестового середовища.

Методика верифікації коректності. Для автоматичної валідації результатів виконання операторів $L(G)$ та $L^{-1}(G)$ у межах роботи було імплементовано `heuristic_isomorphism_check` — функцію швидкої евристичної перевірки на ізоморфізм.

Оскільки точне розв'язання задачі ізоморфізму графів є обчислювально складною проблемою, для підтвердження того, що граф-прообраз G є ізоморфним графу G', отриманому в результаті циклу операцій $G \to L(G) \to L^{-1}(L(G))$, алгоритм аналізує лише ключові топологічні інваріанти. А саме: збіг кількості вершин, кількості ребер та збіг відсортованих послідовностей степенів вершин. Порівняння цих метрик є необхідною, хоча й не достатньою умовою повного ізоморфізму, що робить функцію простим та швидким ін-

струментом автоматизованого контролю якості роботи операторів на графах великого розміру, для яких ймовірність неспрацювання такої перевірки дуже мала.

4.2 Оцінка продуктивності сумісного обходу при побудові реберного графа

Оцінка накладних витрат застосування патернів Фасад та Компонувальник. Цей етап обчислювальних експериментів був спрямований на емпіричну перевірку архітектурної гіпотези щодо того, яку ціну швидкодії доводиться платити за використання додаткових шарів абстракції. Для цього було проведено серію тестів (по 30–50 ітерацій для кожної розмірності графів Барабаші-Альберта), у яких порівнювався час виконання оператора побудови реберного графа $L(G)$ за трьох різних стратегій запуску: через спрощений інтерфейс Фасад, через ізольованого об'єкта-відвідувача та за умови його інкапсуляції у патерн Компонувальник (клас `BfsCompositeVisitor`). Фіксовано розрідженість $m = 3$.

Результати експериментів наведено у таблиці 1 та на рисунку 2.

Табл. 1: Час виконання оператора $L(G) = H$ для різних патернів

$\|V(G)\|$	$\|V(H)\|$	$\|E(H)\|$	**Фасад, мс**	**Відвідувач, мс**	**Компонув., мс**
10 000	30 000	$6.0 \cdot 10^5$	4.8	4.3	4.3
25 000	75 000	$1.4 \cdot 10^6$	12.7	11.6	11.7
50 000	150 000	$3.0 \cdot 10^6$	27.6	26.5	26.3
100 000	300 000	$6.8 \cdot 10^6$	66.2	65.4	65.0
150 000	450 000	$1.0 \cdot 10^7$	111.6	109.4	107.0

Аналіз отриманих результатів свідчить про високу ефективність розробленої архітектури та дозволяє детально розмежувати вплив різних підходів на загальну швидкодію. Насамперед, використання інтерфейсу Фасад демонструє відхилення від чистого Відвідувача на 1–2%.

Така незначна різниця має чітке системне пояснення. Відповідно до ба-

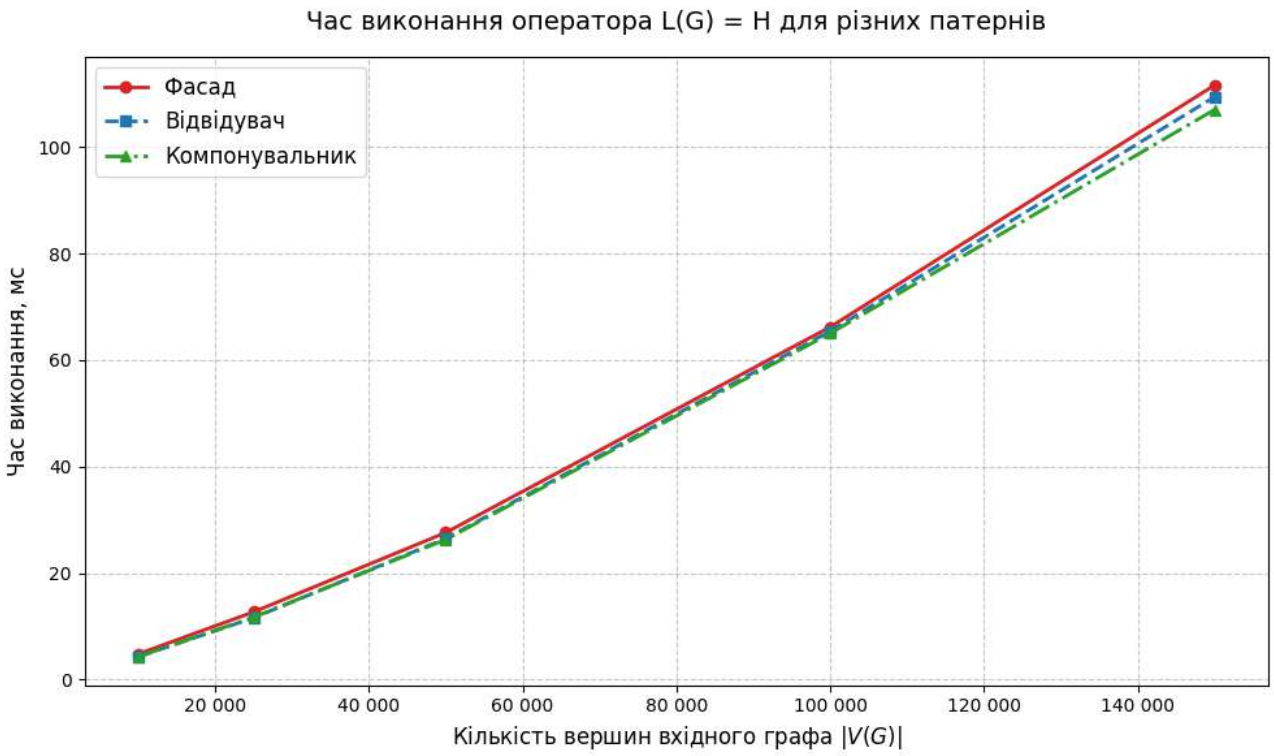

Рис. 2: Порівняння накладних витрат патернів проєктування

зової для C++ ідіоми RAII (Resource Acquisition Is Initialization), концепцію якої запровадив Б'ярн Страуструп [21], керування життєвим циклом об'єктів тісно пов'язане з їхньою областю видимості. Синхронізація таймерів в експерименті (виділення строгого блоку видимості для об'єктивного врахування часу деалокації пам'яті в усіх підходах) вирівняла основні показники. Проте архітектура Фасада все ще вимагає самостійної ініціалізації проміжних структур всередині свого виклику та виконання дублюючого циклу для підрахунку розмірностей графа. Натомість архітектура на базі Відвідувача з використанням Впровадження залежностей залишається дещо ефективнішою моделлю, оскільки дає змогу об'єднати підготовчі цикли, що два шари Фасадів запускають окремо.

В той же час, порівняння ізольованого Відвідувача та Компонувальника підтверджує абсолютну відсутність накладних витрат на диспетчеризацію під час виконання (runtime). Час їхньої роботи є практично ідентичним. Відповідно, абстракція Компонувальника слугує надійним прикладом архітектури з нульовою вартістю, яка завдяки обчисленням на етапі компіляції зберігає граничну швидкодію монолітного коду.

Оцінка ефективності спільного обходу. Отримавши підтвердження відсутності накладних витрат Компонувальника на базовому рівні, наступним кроком стало дослідження його обчислювальної ефективності під час спільного виконання декількох операторів. Головна гіпотеза полягала у тому, що спільний обхід має виконуватися швидше за два послідовні запуски завдяки оптимізації локальності даних та зменшенню кількості промахів кешу процесора.

Для всебічного аналізу експеримент було розділено на дві частини: масштабування за щільністю зв'язків (табл. 2) та масштабування за розміром графа (табл. 3). В обох випадках виконувалася побудова двох незалежних реберних графів на базі одного вхідного графа.

Табл. 2: Порівняння виконання двох операторів $L(G)$ разом та послідовно при масштабуванні щільності (фіксовано $|V(G)| = 10\,000$)

m	$\lvert V(H)\rvert$	$\lvert E(H)\rvert$	**Послідовно, мс**	**Разом, мс**	**Прискорення**
10	100 000	$4.7 \cdot 10^6$	49.3	48.7	1.2%
20	200 000	$17.2 \cdot 10^6$	158.1	152.3	3.7%
30	300 000	$37.1 \cdot 10^6$	329.4	308.6	6.3%
40	400 000	$63.7 \cdot 10^6$	556.7	495.8	10.9%
50	500 000	$96.1 \cdot 10^6$	848.4	713.9	15.8%

Табл. 3: Порівняння виконання двох операторів $L(G)$ разом та послідовно при масштабуванні розміру (фіксовано розрідженість $m = 3$)

$\lvert V(G)\rvert$	$\lvert V(H)\rvert$	$\lvert E(H)\rvert$	**Послідовно, мс**	**Разом, мс**	**Прискорення**
10 000	30 000	$6.0 \cdot 10^5$	9.6	8.3	13.5%
25 000	74 000	$1.4 \cdot 10^6$	25.9	23.6	8.8%
50 000	150 000	$3.0 \cdot 10^6$	56.2	52.1	7.3%
75 000	225 000	$4.8 \cdot 10^6$	92.9	85.0	8.5%
100 000	300 000	$6.8 \cdot 10^6$	132.6	126.9	4.3%

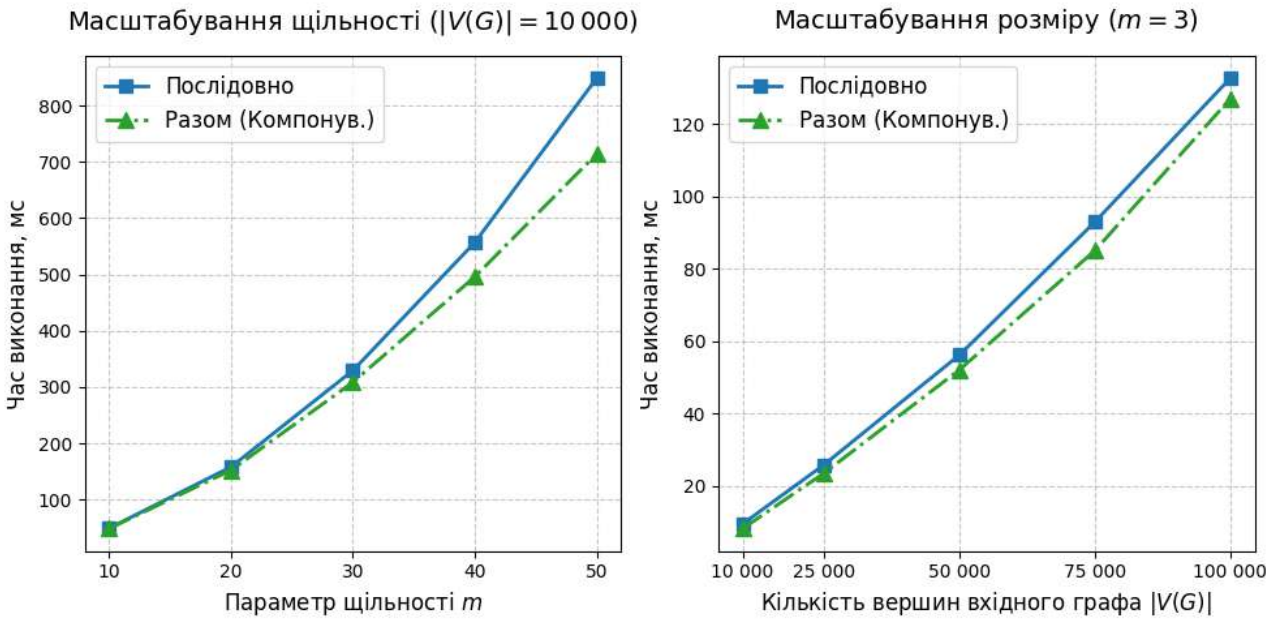

Рис. 3: Різниця часу послідовного та об'єднаного виконання.

Аналіз отриманих результатів емпірично доводить ефективність запропонованої архітектури на апаратному рівні. Найбільш показовим є перший експеримент зі щільними графами (табл. 2). Зі збільшенням параметра m (кількості ребер на вершину) експоненційно зростає розмір списків суміжності, які алгоритм має опрацювати. При послідовному виконанні кожен Відвідувач змушує процесор окремо завантажувати масивні блоки пам'яті графа з повільної оперативної пам'яті (RAM) у кеш L3, що призводить до дублювання промахів кешу. Натомість Компонувальник піднімає дані поточної вершини та її сусідів у надшвидкий кеш першого та другого рівнів (L1/L2) лише один раз, після чого процесор встигає виконати логіку обох операторів над уже завантаженими даними. Саме завдяки мінімізації звернень до шини пам'яті Компонувальник забезпечує абсолютне скорочення часу виконання на 134.5 мс для найбільш щільного графа, що еквівалентно майже 16% приросту продуктивності.

В експерименті з розрідженими графами (табл. 3) спільний обхід також стабільно перевершує послідовне виконання. Характерно, що навіть на структурах масивних обсягів ($6.8 \cdot 10^6$ ребер), які гарантовано виходять за межі 9 МБ кешу L3 тестового процесора, об'єднання Відвідувачів зберігає стійку перевагу.

Підсумовуючи, використання патерна Компонувальника не лише ство-

рює зручний інтерфейс для об'єднання операторів, але й діє як ефективний механізм системної оптимізації завдяки розгортанню на етапі компіляції та використанню спільного обходу.

4.3 Порівняльний аналіз швидкодії розробленої адаптації алгоритму ILIGRA

Порівняльний аналіз швидкодії розробленої адаптації алгоритму ILIGRA. Наступним етапом обчислювальних експериментів стало дослідження ефективності розробленої адаптації алгоритму побудови прообразу реберного графа (ILIGRA). Метою тестування було визначення того, як подіє-орієнтована архітектура на базі патерна Відвідувача впливає на загальну швидкодію порівняно з класичною (монолітною) реалізацією алгоритму, де обчислювальна логіка інкапсульована у лінійні цикли.

Для забезпечення об'єктивності експерименту тестування проводилося на ідентичних екземплярах реберних графів $L(G)$, згенерованих з розріджених графів Барабаші-Альберта (із фіксованим параметром $m = 3$). Порівнювалися три підходи: класична узагальнена реалізація ILIGRA зроблена в рамках роботи, адаптована версія для обходу в ширину (через Фасад), а також Відвідувач, інкапсульований у Компонувальник. Структурні параметри тестових графів ідентичні розмірностям, наведеним у попередніх експериментах (табл. 3). Результати вимірювань наведено у таблиці 4 та на рисунку 4.

Табл. 4: Час виконання алгоритму ILIGRA ($L^{-1}(H) = G$) за різних архітектурних підходів

$\|V(G)\|$	**Класична, мс**	**Фасад, мс**	**Компон., мс**
10 000	32.9	27.6	28.8
25 000	114.8	90.6	94.2
50 000	311.8	232.8	240.7
75 000	513.2	375.0	385.5
100 000	837.3	612.7	627.8

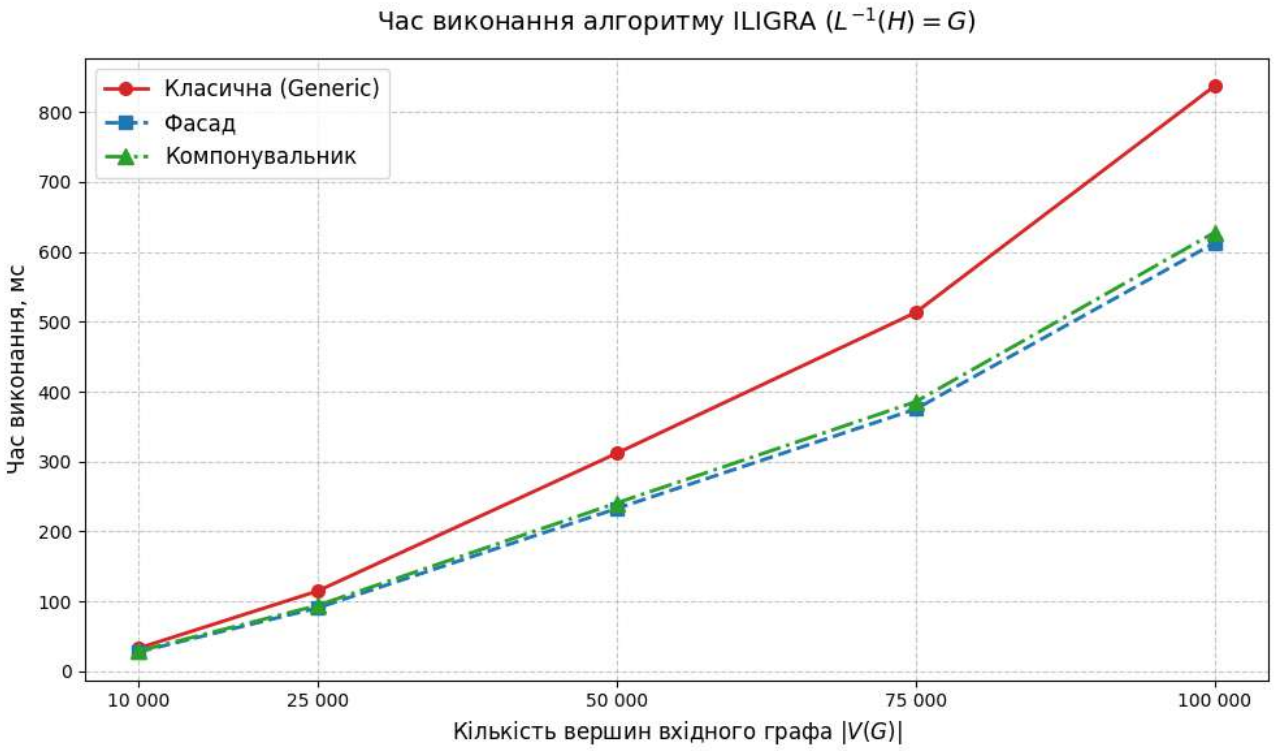

Рис. 4: Вплив архітектурної моделі на швидкодію алгоритму ILIGRA

Аналіз результатів (табл. 4) демонструє, що класична монолітна реалізація алгоритму поступається у швидкодії адаптованій версії на базі Відвідувача. На масивних структурах (понад 6.8 млн ребер) ця різниця досягає майже 27% (837.3 мс проти 612.7 мс).

Таке підвищення продуктивності пояснюється особливостями роботи оптимізатора компілятора, зокрема механізмом вбудовування функцій (inlining). У класичній реалізації складна логіка маршрутизації та виявлення клік зібрана у масивні цикли. Це створює великі блоки інструкцій, для яких компілятор не застосовує вбудовування. Натомість архітектура Відвідувача робить декомпозицію алгоритму на компактні події. Завдяки використанню статичного поліморфізму компілятор сприймає ці методи як невеликі незалежні інструкції та ефективно вбудовує їх безпосередньо у цикл базового обходу. Додаткові експерименти підтвердили, що швидкодія Відвідувача критично залежить саме від здатності компілятора згенерувати та оптимізувати унікальні гілки коду під час розкриття шаблонів. У штучно створених сценаріях, де компілятор не вбудовував події Відвідувача в BFS функцію, швидкодія адаптації була майже рівною швидкодії класичної реалізації.

Водночас порівняння Фасада та Компонувальника виявило мінімальне

зниження швидкодії на рівні 2.4% (612.7 мс проти 627.8 мс). Припускається, що оскільки алгоритм ILIGRA містить значний обсяг логіки, інтеграція у Компонувальник дещо збільшує обсяг згенерованого коду, що додає незначні накладні витрати. Проте це є цілком прийнятною архітектурною платою, враховуючи загальний виграш у швидкодії від переходу на подіє-орієнтовану модель.

Оцінка ефективності спільного обходу для оператора $L^{-1}(H)$. Фінальним етапом експериментального дослідження стала оцінка ефективності спільного обходу для обчислювально складного оператора відновлення прообразу графа (алгоритм ILIGRA). Для цього моделювалося подвійне виконання операторів, за якого порівнювалися три конфігурації: двократне послідовне виконання класичного (монолітного) варіанта алгоритму, послідовний виклик двох незалежних Відвідувачів, а також застосування Компонувальника. Структурні параметри тестових графів ідентичні розмірностям, наведеним у попередніх експериментах (табл. 3).

Результати вимірювань часу та розрахований коефіцієнт прискорення Компонувальника відносно послідовних Відвідувачів наведено в таблиці 5 та на рисунку 5.

Табл. 5: Порівняння часу виконання алгоритму відновлення графа-прообразу при подвійному навантаженні

$\|V(G)\|$	**Класичний, мс**	**Відвід., мс**	**Компонув., мс**	**Прискорення**
10 000	66.0	55.3	48.1	13.0%
25 000	232.4	184.0	164.0	10.9%
50 000	591.5	440.3	397.7	9.7%
75 000	1005.5	758.9	691.9	8.8%
100 000	1588.4	1181.3	1086.8	8.0%

Аналіз експериментальних даних повністю підтверджує тенденції, виявлені у попередньому тестуванні. Час роботи класичної реалізації алгоритму ILIGRA очікувано зріс пропорційно до кількості викликів. Натомість конфі-

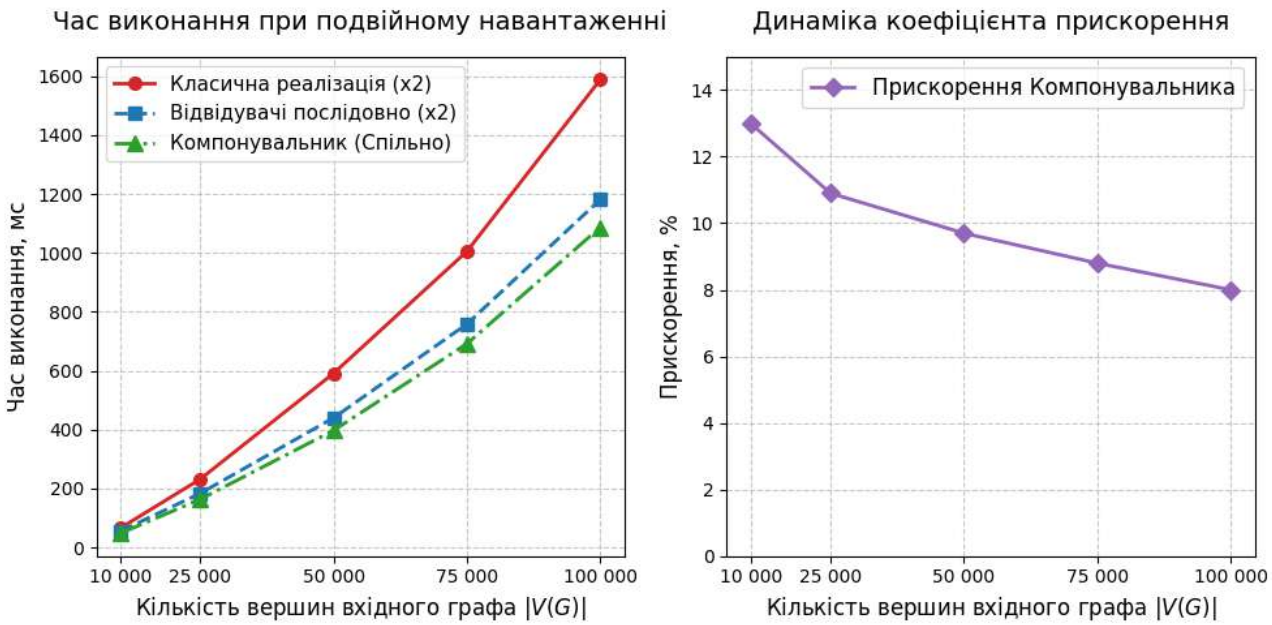

Рис. 5: Ефективність спільного виконання алгоритмів ILIGRA залежно від обсягу даних

гурація з послідовним виконанням двох Відвідувачів зберегла свою значну перевагу завдяки кращій оптимізації машинного коду та агресивному вбудовуванню подій компілятором.

Проте найважливішим результатом тесту є беззаперечне лідерство патерна Компонувальник на всіх масштабах вхідних даних. Виконання логіки двох алгоритмів у межах єдиного обходу в ширину дозволила уникнути повторного викачування масивного графа з повільної оперативної пам'яті.

Особливої уваги заслуговує динаміка коефіцієнта прискорення: він плавно знижується з 13.0% на малих графах до 8.0% на надвеликих структурах. Це пояснюється високою обчислювальною складністю самого алгоритму ILIGRA, який містить значний обсяг внутрішньої логіки на кожну ітерацію (наприклад перевірка клік). Зі збільшенням розміру графа процесор витрачає левову частку часу саме на ці інтенсивні обчислення для двох операторів одночасно. Як наслідок, час, зекономлений завдяки одноразовому завантаженню масивного графа з пам'яті, стає менш помітним у відсотковому відношенні на тлі загального часу, витраченого на математичну логіку.

Тим не менш, абсолютне скорочення часу виконання (наприклад, економія майже 95 мс для графа у 100 тис. вершин) переконливо доводить загальну ефективність адаптація алгоритмів для спільного обходу із застосу-

ванням патерна Відвідувача. Компонувальник забезпечує значний приріст продуктивності навіть для складних задач, не потребуючи при цьому змін у математичній логіці самих алгоритмів.

ВИСНОВКИ

У кваліфікаційній роботі розв'язано науково-практичну задачу підвищення ефективності графових операторів шляхом їх інтеграції у подіє-орієнтовану архітектуру на базі сучасного стандарту C++20. Відповідно до поставленої мети та завдань отримано такі результати:

1. **Проаналізовано класичні підходи** до побудови узагальнених графових бібліотек на прикладі BGL. Встановлено, що використання метапрограмування C++98 (системи SFINAE та макросів) спричиняє надмірну синтаксичну складність та збільшує час компіляції, що доцільно усунути сучасними засобами мови.

2. **Сформульовано критерії інтеграції** графових операторів у спільний обхід (BFS/DFS): асимптотична сумісність обчислень (локальна обчислювальна складність $O(\deg(u))$), послідовна інформаційна локальність та однопрохідна повнота.

3. **Розроблено та адаптовано алгоритми топологічних перетворень.** Створено оригінальний алгоритм побудови реберного графа $L(G)$ та адаптовано ітеративний алгоритм ILIGRA [16] для відновлення його прообразу. Обидва оператори успішно переведено у парадигму подіє-орієнтованого обходу, що дозволило виконувати їх за один прохід пошуку в ширину (BFS) на базі патерна Відвідувач.

4. **Спроєктовано та реалізовано архітектуру** прототипу графової бібліотеки мовою C++20. Застосування статичних патернів (Відвідувач, Компонувальник, Впровадження залежностей) на базі концептів і варіативних шаблонів дозволило об'єднувати оператори без накладних витрат на динамічну диспетчеризацію.

5. **Проведено обчислювальні експерименти**, які довели ефективність запропонованих рішень. Спільний обхід при виконанні двох операторів побудови реберного графа $L(G)$ через патерн Компонувальник забезпечив до 16% приросту продуктивності на щільних графах за рахунок зменшення промахів кешу процесора. Декомпозиція логіки ILIGRA на

ізольовані події Відвідувача дозволила компілятору виконати агресивне вбудовування функцій (inlining), що скоротило час виконання на масивних структурах до 27% порівняно з класичною монолітною реалізацією.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Бублик В. В. *Об'єктно-орієнтоване програмування: Підручник.* — Київ: ІТ-книга, 2015. — 624 с.

[2] *Матеріали XIV Всеукраїнської наукової конференції молодих математиків (7–8 травня 2026 р., м. Київ)* [Електронний ресурс]. — Київ: УДУ імені Михайла Драгоманова, 2026. — Режим доступу: `https://drive.google.com/file/d/1I5sdbEmrgRn5VO-oyP3HeXXXe9euYp76/view?usp=sharing` (дата звернення: 17.05.2026).

[3] Трохимчук Р. М., Нікітченко М. С. *Дискретна математика у прикладах і задачах: Навчальний посібник.* — Київ: ВПЦ "Київський університет 2017. — 248 с.

[4] Alexandrescu A. *Modern C++ Design: Generic Programming and Design Patterns Applied.* — Reading: Addison-Wesley, 2001. — 352 p.

[5] Barabási A.-L., Albert R. Emergence of scaling in random networks. *Science.* — 1999. — Vol. 286, no. 5439. — P. 509–512.

[6] Batagelj V., Brandes U. Efficient generation of large random networks. *Physical Review E.* — 2005. — Vol. 71, no. 3. — Art. no. 036113.

[7] Beineke L. W. Characterizations of derived graphs. *Journal of Combinatorial Theory.* — 1970. — Vol. 9, no. 2. — P. 129–135.

[8] Christofides N. *Graph Theory: An Algorithmic Approach.* — New York: Academic Press, 1975. — 400 p.

[9] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. *Introduction to Algorithms.* — 3rd ed. — Cambridge: MIT Press, 2009. — 1292 p.

[10] Fabian R. *Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules.* — Richard Fabian, 2018. — 272 p.

[11] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software.* — Reading: Addison-Wesley, 1994. — 395 p.

[12] Harary F. *Graph Theory.* — Reading: Addison-Wesley, 1969. — 274 p.

[13] Hennessy J. L., Patterson D. A. *Computer Architecture: A Quantitative Approach.* — 6th ed. — Cambridge: Morgan Kaufmann, 2017. — 936 p.

[14] Josuttis N. M. *C++20: The Complete Guide.* — Braunschweig: Nicolai Josuttis, 2022. — 716 p.

[15] Josuttis N. M. *C++ Move Semantics: The Complete Guide.* — Braunschweig: Nicolai Josuttis, 2020. — 264 p.

[16] Liu D., Trajanovski S., Van Mieghem P. ILIGRA: An Efficient Inverse Line Graph Algorithm. *Journal of Mathematical Modelling and Algorithms in Operations Research.* — 2015. — Vol. 14, no. 1. — P. 13–33.

[17] Meyers S. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14.* — Sebastopol: O'Reilly Media, 2014. — 334 p.

[18] Roussopoulos N. D. A $\max\{m, n\}$ algorithm for determining the graph H from its line graph G. *Information Processing Letters.* — 1973. — Vol. 2, no. 4. — P. 108–112.

[19] Seemann M., van Deursen S. *Dependency Injection Principles, Practices, and Patterns.* — Shelter Island: Manning Publications, 2019. — 552 p.

[20] Siek J., Lee L.-Q., Lumsdaine A. *The Boost Graph Library: User's Guide and Reference Manual.* — Boston: Addison-Wesley, 2002. — 376 p.

[21] Stroustrup B. *The C++ Programming Language.* — 4th ed. — Upper Saddle River: Addison-Wesley, 2013. — 1368 p.

[22] Tarjan R. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing.* — 1972. — Vol. 1, no. 2. — P. 146–160.

[23] Whitney H. Congruent Graphs and the Connexions of Vertices. *American Journal of Mathematics.* — 1932. — Vol. 54, no. 1. — P. 150–168.

Використання інструментів штучного інтелекту

У підготовки цієї роботи використовувалися великі мовні моделі (зокрема, **Gemini 3.1 Pro**) для виконання таких завдань:

- **Робота з текстом та форматування:** Інструменти ШІ застосовувалися для стилістичного редагування тексту, структурування розділів, консультацій щодо використання LaTeX, допомоги в оформленні списку використаних джерел (бібліографії) та перевірки кінцевого тексту на відповідність формальним вимогам положення про кваліфікаційні роботи. Усі згенеровані текстові фрагменти ретельно перевірялися, уточнювалися, доопрацьовувалися автором та відображають авторську позицію та зміст офіційних джерел.

- **Програмування та аналіз коду:** ШІ використовувався для пришвидшення орієнтування в архітектурі бібліотеки BGL та написання чорнових фрагментів допоміжного коду мовою C++. Інструменти також застосовувалися для пошуку багів та оптимізації алгоритмів. Згенерований код не використовувався автоматично: кожен фрагмент проходив обов'язкове ручне рев'ю, рефакторинг та перевірку шляхом компіляції і тестування.

- **Візуалізація та аналіз даних:** У Розділі 4 штучний інтелект допомагав у первинному оформленні результатів обчислювальних замірів та створенні графічних візуалізацій. Достовірність відображених даних та графіків була строго верифікована автором на основі оригінальних логів обчислювальних експериментів.

Інструменти ШІ не застосовувалися для автоматичного створення основного змісту роботи чи фальсифікації експериментальних даних. Відповідальність за фінальний текст, програмні реалізації та висновки повністю несе автор роботи.