ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І.І.МЕЧНИКОВА
(повне найменування вищого навчального закладу)

Факультет математики, фізики та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра математичного забезпечення комп'ютерних систем
(повна назва кафедри (предметної, циклової комісії))

Кваліфікаційна робота

на здобуття рівня вищої освіти «бакалавр»
(рівень вищої освіти)

на тему Алгоритми пошуку на графових структурах, що зберігаються в базах даних
Search algorithms on graph structures stored in databases

Виконала: студентка денної форми навчання
спеціальності 123 – Комп'ютерна інженерія .
(шифр і назва напряму підготовки, спеціальності)
Освітня програма «Комп'ютерна інженерія»
(назва освітньої програми)
Чернова Олена Юріївна
(прізвище, ім'я, по-батькові)

Керівник к.ф.-м.н. Антоненко О.С.
(науковий ступінь, вчене звання, прізвище та ініціали, **підпис**)

Рецензент к.ф.-м.н, доц. Петрушина Т.І.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
№___ від «___»_____ 2024 р.

Завідувач кафедри
_____ _Євгеній МАЛАХОВ_
(підпис) (ім'я, прізвище)

Захищено на засіданні ЕК № ___
протокол № __ від «___» ____ 2024 р.
Оцінка _____/_____/_____
(за національною шкалою, шкалою ECTS, бали)

Голова ЕК
_____ _Алла КОБОЗЄВА_
(підпис) (ім'я, прізвище)

Одеса - 2024

АНОТАЦІЯ

Дипломну роботу присвячено реалізації алгоритмів пошуку на графових структурах, що зберігаються в базах даних. Для демонстрації та візуалізації роботи розроблених алгоритмів створено інформаційну систему. Використовувались такі алгоритми, як: пошук в глибину, пошук у ширину та алгоритм пошуку мінімального шляху (алгоритм Дейкстри). Для зберігання та аналізу даних обрано реляційну базу даних PostgreSQL.

Систему спроектовано у вигляді веб-застосунку з веб-сервером Internet Information Services за допомогою мови програмування високого рівня C#, шаблону проектування MVC (який вбудовано у технології компанії Microsoft – ASP.NET Core 6 MVC) у середовищі розробки Microsoft Visual Studio Community 2022. Для візуального представлення графічних структур застосовано карти OpenStreetMap з використанням JavaScript бібліотеки Leaflet.

Систему протестовано на різній кількості вершин та ребер, які утворюють графову структуру. Візуалізовано та анімовано алгоритми пошуку на картах. Розроблена система працює коректно та може застосовуватись для автоматизованого аналізу даних, що скорочує час обробки інформації та дозволяє оцінити роботу реляційної бази даних щодо графових структур.

ABSTRACT

The diploma work is devoted to the implementation of search algorithms on graph structures stored in databases. An information system has been created to demonstrate and visualize the work of the developed algorithms. The following algorithms were used: depth-first search, breadth-first search and the minimum path search algorithm (Dijkstra's algorithm). For data storing and analyze was chosen relational database PostgreSQL.

The system was designed as a web application with web server Internet Information Services with C# as high-language programming language, with design pattern MVC (which is build in technology of company Microsoft – ASP.NET Core 6 MVC), in the development environment Microsoft Visual Studio Community 2022. For visual representation of graph structures applied maps OpenStreeMap using the Javascript library Leaflet.

The system was tested with different amounts of vertex and edges, which form graph structure. Search algorithms are visualized and animated on the map. The system which was created works correctly and can be used for automated data analysis, which reduces the time of information processing and allows to evaluate the work of relational databases for graph structures.

ЗМІСТ

ВСТУП

Наразі існує багато задач, які вирішують за допомогою графових структур з використанням сучасних технологій – прокладання шляхів різних типів (для велосипедів, для автомобільного транспорту, для пішоходів тощо) між певними пунктами призначення, складання сімейного дерева, зв'язок між веб-сторінками, соціальна мережа і т.д. Такі дані необхідно зберігати та обробляти, для цього існують бази даних (БД) – як місце зберігання та система управління базами даних – відповідно для обробки. Графові структури можна зберігати та обробляти, відповідно, як у графовій, так і в реляційній базі даних. Чи є правильним рішенням використовувати в даному випадку реляційні БД та як їх використовувати є комплексним актуальним питанням, на яке дана робота спробує показати своє рішення.

Мета дипломної роботи – реалізація алгоритмів пошуку на графових структурах, що зберігаються в базах даних. Для демонстрації та візуалізації роботи реалізованих алгоритмів необхідно створити інформаційну систему (ІС).

Для досягнення зазначеної мети потрібно виконати наступні дії:

1) виконати аналіз предметної області з застосуванням графових алгоритмів до баз даних;

2) спроектувати інформаційну систему для демонстрації алгоритмів;

3) спроектувати модель застосунку;

4) обрати технології для реалізації мети роботи;

5) створити базу даних, що зберігає графовову структуру даних;

6) реалізувати у базі даних наступні алгоритми: пошук в глибину (depth -first search), пошук в ширину (breadth-first search) та алгоритм Дейкстри (Dijkstra's algorithm);

7) створити веб-застосунок, де візуалізувати, отримані в результаті роботи алгоритмів, дані на карті.

1 ДОВІДКОВА ІНФОРМАЦІЯ

1.1 Загальні відомості про графові структури

Майже будь-яку інформацію можна представити у вигляді графової структури, визначити точки (які назвемо вершини або вузли) і зв'язки між точками (котрі можна назвати ребрами або дугами). І от вже маємо графову структуру в загальному розумінні, яку показано на рис. 1.1.

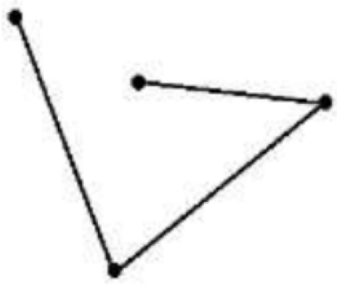


Рисунок 1.1 – Загальний вигляд графа [1]

Графом (G) є абстрактний комбінаторний об'єкт, який складається з двох множин V (vertex) – довільна множина вершин, або об'єктів і E (edge) – множина ребер, або пар об'єктів з V [1].

Є різні види графів. За типом ребер, їх поділяють на орієнтовані (є направлений напрямок з однієї початкової вершини до іншої кінцевої; на рисунках показано стрілками) та неорієнтовані (між двома вершинами напрямок є як з однієї до іншої, так і з іншої до даної однієї, тобто двонаправлений). У даному прикладі, на рис. 1.1 напрямок неорієнтований. Також є графи, котрі є зваженими – є вага ребра, наприклад відстань між двома вершинами, або будь-яке інше значення, яким можна охарактеризувати цей зв'язок; у випадку представлення соціальної мережі у вигляді графа, де вузлами є профілі користувачів, а ребрами наявність друзів, ребро може мати значення "є другом".

Граф може мати цикли (називають циклічний) або ні (такий тип називають ациклічним). Може мати багато ребер з однієї вершини до іншої, які ще називають кратними, а граф є мультиграфом [1]. Важливою ознакою при визначенні графа, а в подальшому при знаходженні шляхів, є петлі. Петля – зв'язок вузла до самого себе. Даний тип графа, який може містити петлі називається псевдографом [1].

Існує багато різних способів представлення графа: матриця суміжності, список суміжності, список ребер і т.д.

1.2 Графові алгоритми

Граф потрібно мати змогу обходити, знаходити оптимальні шляхи між вершинами, знаходити сусідів вузла, чи існують маршрути тощо. Для таких цілей існують спеціальні графові алгоритми, які також варіюються за складністю. Звісно, залежно від умов задачі та від конкретного типу графа є різні алгоритми, які підходять саме для вирішення поставленої задачі.

1.2.1 Пошук в глибину (Depth-first search)

Пошук в глибину (Depth-first search, DFS) – рекурсивний алгоритм, що як походить з назви робить обхід графа в глибину, шукає всі вершини графа. Починаючи з обраної початкової вершини, пошук в глибину досліджує суміжну вершину, до котрої можна піти, тобто сусіда, далі сусіда цього сусіда і т.д. Алгоритм обходить в глибину настільки, наскільки це можливо; шукає ребра з останньої знайденої вершини, котра має ребра, в яких алгоритм не потрапляв. Коли всі ребра вершини досліджено, DFS повертається назад для того, щоб продовжити шукати ті вершини, в яких алгоритм ще не потрапляв. Пошук триває до тих пір, поки всі вузли з початкового вузла, до котрих можна потрапити, не відвідані [2].

На рис. 1.2 показано детально як працює пошук в глибину. Тобто

алгоритм починає свою роботу з вершини з номером один, далі йде у вершину з номером два, яка має дві суміжні вершини з номером чотири та п'ять, заходимо спочатку до четвертої, а потім до п'ятої; пошук повертається до вершини з номером один, так як ще лишається вузол, який не пройшли (третій), тому обхід йде до вершини з номером три. Отже, можемо побачити, що пошук здійснюється в глибину від кожної вершини.

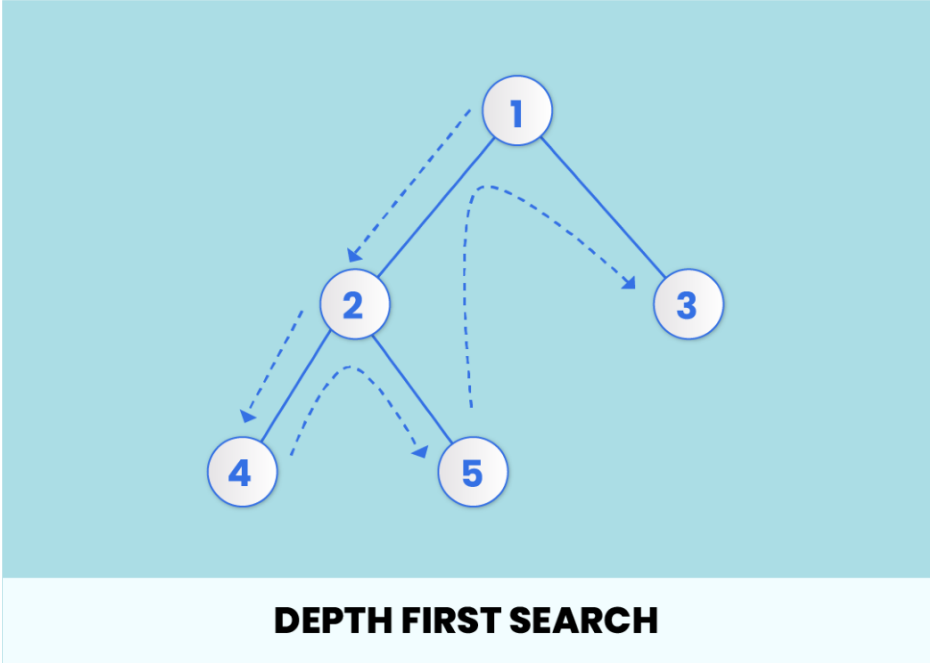


Рисунок 1.2 – Пошук в глибину [3]

1.2.2 Пошук в ширину (Breadth-first search)

Пошук в ширину (Breadth-first search, BFS) – рекурсивний алгоритм, як походить з назви робить обхід графа в ширину та шукає всі вершини графа. Починаючи від початкової вершини, алгоритм досліджує суміжні вершини, тобто сусідів та поки не обійде всіх сусідів на нижчий рівень (пошук суміжних вершин сусідів, тобто сусідів цих сусідів) не спускається. Також пошук у ширину створює дерево в ширину з коренем (початкова вершина), яке містить усі доступні вершини.

На рис. 1.3 детально зображено як працює пошук в ширину. Як бачимо,

вершина з номером один є початковою, починаємо з неї обхід, вона має дві суміжні вершини, а саме вузол з номером два та вузол з номером три. Тому алгоритм спочатку йде до вершини з номером два (яка також має дві суміжні вершини – вузол з номером чотири та вузол з номером п'ять), а потім до вершини з номером три. Далі пошук здійснюються почергово до суміжних вершин вузла з номером два (тобто до вершин з номерами чотири, п'ять).

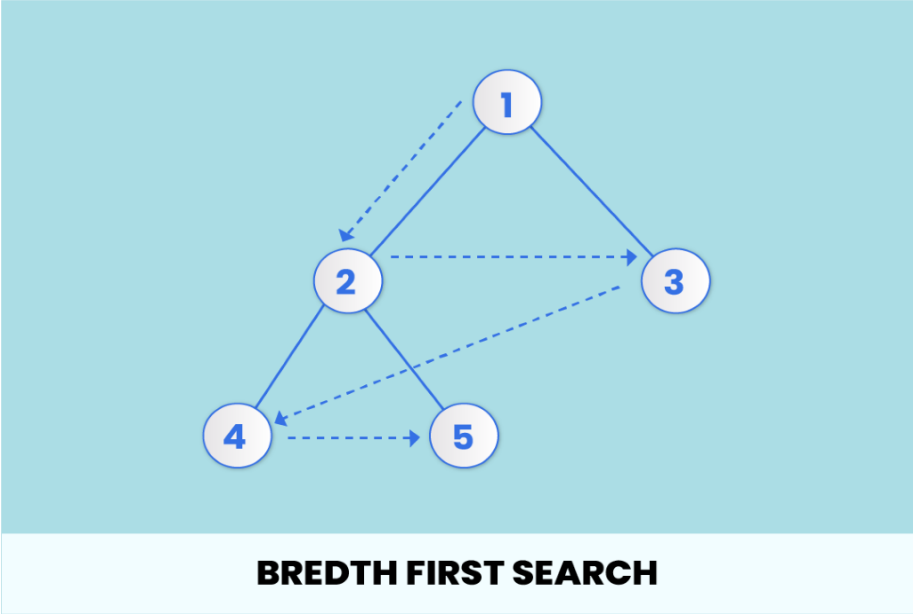


Рисунок 1.3 – Пошук в ширину [3]

1.2.3 Алгоритм Дейкстри (Dijkstra's algorithm)

Алгоритм Дейкстри шукає мінімальний маршрут з однієї початкової обраної вершини до обраної другої на орієнтованому зваженому графі, для його роботи потрібні невід'ємні ваги ребер. Названо на честь його винахідника вченого Едсгера Дейкстри. Алгоритм зберігає набір вершин, для яких загальні ваги найкоротшого шляху від джерела вже визначені [2]. Алгоритм починає працювати з обраної початкової вершини, знаходячи найкоротший шлях до всіх інших вершин графа. Це відбувається шляхом підтримки набору вершин, де мінімальний шлях від обраної вершини вже

знайдено, та набору вершин, де мінімальний шлях потрібно визначити. Алгоритм вибирає, на кожному кроці, вершину з найкоротшою відомою відстанню від початкової вершини та додає її до набору вершин із відомими найкоротшими шляхами. Далі алгоритм перевіряє всі сусідні вершини, яких ще немає в наборі вершин із відомими найкоротшими шляхами, оновлює їхні відстані, якщо знайдено коротший шлях. Цей процес повторюється доти, доки всі вершини не будуть додані до набору вершин із відомими найкоротшими шляхами або доки не буде досягнуто вершину призначення.

Найпростіший приклад роботи алгоритму Дейкстри наведено на рис. 1.4. Як бачимо, граф має чотири вершини (A, B, C, D) є орієнтованим та зваженим. З вершини A до вершини D можна потрапити або через шлях від A до D (вартість шляху дорівнюватиме три), або через вершину C (де вартість шляху вже дорівнюватиме значенню два). Тому мінімальний шлях з вершини A до D є через вузол C.

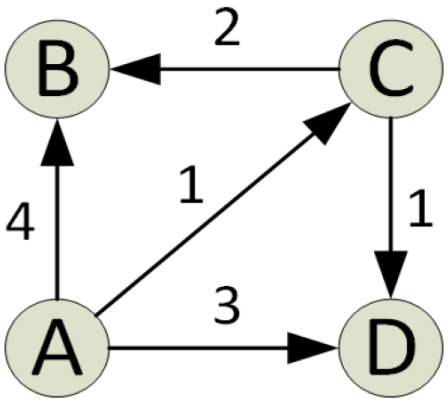


Рисунок 1.4 – Алгоритм Дейкстри [4]

Алгоритм Дейкстри використовується в різних сферах життя, наприклад:

– в протоколах маршрутизації (OSPF – Open Shortest Path First, IS-IS – Intermediate System to Intermediate System) – для того, щоб обчислити найкоротший шлях між двома маршрутизаторами на основі топології мережі та вартості з'єднання (link costs), де вартість з'єднання, зазвичай, базується на

таких факторах, як пропускна здатність, затримка та надійність зв'язку;

– GPS-навігації – вершинами є перехрестя доріг або цікаві точки, а ребрами є дороги або шляхи, що їх з'єднують; ваги зазвичай є або відстанню між вузлами, або передбачуваним часом, за яким можна потрапити від обраної початкової вершини до кінцевої;

– в розкладі авіакомпанії – для того, щоб знайти найкоротший маршрут між двома аеропортами, вершинами є самі аеропорти, а ребрами є рейси між ними; в даному випадку, ваги розраховано на основі відстані або часу подорожі;

– у робототехніці – для визначення шляху робота, тобто пошук найкоротшого шляху, за яким робот може пройти з однієї початкової точки розташування до іншої кінцевої, уникаючи будь-яких перешкод; вершинами є місця, які робот може займати, а ребрами є можливі переміщення між ними, ваги базуються на вартості або часу переміщення;

– у розробці ігор – для пошуку мінімального шляху між поточним розташуванням персонажа та цільовим місцем розташуванням, уникаючи будь-які перешкоди або ворогів; вершинами є місця, куди персонаж може рухатися, а ребрами є можливі переміщення між ними [5].

1.3 Графові бази даних

Для розміщення та з подальшою взаємодією з графовими структурами в базах даних, існують спеціальний тип БД, який призначений саме для зберігання, обробки та пошуку на графах – графові бази даних. Графова база даних – це систематичний набір даних, що підкреслює зв'язки між різними сутностями даних. Вони використовують математичну теорію графів (використання вершин, ребер і напрямків, з'єднань, алгоритмів тощо) для відображення зв'язків. Важливою перевагою графових баз даних є відсутність схем, так як графові БД зберігають дані як мережу сутностей і зв'язків. Вузли – це вершини, що зберігають об'єкти даних. Кожен вузол

може мати необмежену кількість та типи зв'язків. Кожен вузол (у деяких випадках ребро) має властивості або атрибути, які його описують. Графи з властивостями мають назву графи властивостей [6].

Графові БД часто відносять до числа NoSQL, тому на відміну від реляційних баз даних, графові не використовують структуровану мову запитів SQL. Майже кожна графова база даних має свою мову запитів, наприклад Neo4j – використовує мову Cypher, ArangoDB – використовує мову AQL, TigerGraph – GSQL (TigerGraph Query Language), Aerospike – Gremlin, Microsoft Azure Cosmos DB – Gremlin [7].

Важлива перевага БД даного типу полягає у відсутності схем. БД здатні керувати структурованими, неструктурованими або напівструктурованими даними – це робить їх більш гнучкими у порівнянні з реляційними БД, які оперують над структурованими даними у вигляді таблиць або схем [8].

1.4 Реляційні бази даних

У реляційній БД дані оформлені у вигляді множини двовимірних таблиць, де кожна описує той чи інший тип сутності і її властивості. При цьому, стовпець таблиці відповідає одній властивості чи атрибуту сутності, а рядок — екземпляру сутності з конкретними значеннями атрибутів [9]. Дану модель створив Е.Ф. Кодд, він представив реляційну модель даних із відношеннями для представлення даних, з реляційною алгеброю та реляційним численням для роботи з даними та обмеженням реляційної цілісності – для контролю узгодженості та повноти даних [8]. Реляційні БД використовують традиційну структуровану мову запитів SQL, яка є потужним інструментом.

Графову структуру можна різно зберігати в реляційній базі даних. Наприклад, у вигляді двох таблиць – для вершин та для ребер. Таблиця вершин матимете первинний ключ та інші атрибути, які характеризуватимуть

вершину. Таблиця ребер матимете початковий та кінцевий вузол у вигляді посилання на первинний ключ до таблиці вершин, а також вагу (у випадку зваженого графа). Також графову структуру можливо представити у вигляді однієї таблиці, яка має такі атрибути, як: унікальний ідентифікатор (первинний ключ), початковий вузол, кінцевий вузол та вага (якщо граф є зважений). Є випадки, коли немає унікального ідентифікатора та для перевірки унікальності кожного кортежу додається складений ключ з {початкова вершина, кінцева вершина}, тобто об'єднання початкової та кінцевої вершини. Але у даному випадку граф не зможе мати кратні ребра.

1.5 Аналіз предметної області

Для виконання зазначеної мети дипломної роботи обрано реляційну базу даних через уніфіковану мову запитів SQL та використання реляційної алгебри. Отже, для цього необхідно провести аналіз предметної області у пошуках схожих підходів, напрацювань і т.д. для формування знань щодо стану та розвитку даної теми у галузі.

Згідно з підручником [10], де цілий розділ присвячено графам у реляційних базах даних, створення графових структур та написання алгоритмів на мові SQL можна зробити декількома способами. Наприклад, найбільш поширеним методом моделювання графів є список суміжності. Є дві таблиці, перша – виключно для вершин, а друга – для створення зв'язків між ними, тобто список суміжності. Також можливо представити граф у вигляді матриці суміжності. Матриця суміжності – це квадратний масив, рядки якого є вихідними вузлами, а стовпці – вхідними вузлами. Якщо в комірці є одиниця, то це означає те, що між двома вузлами є ребро. Також автор навів багато різних прикладів для пошуку шляхів, серед яких є: пошук всіх шляхів між вузлами, пошук найкоротших шляхів від одного вузла до інших вузлів з та без рекурсивного запиту, визначення відстані різних маршрутів тощо.

Дослідження [11] присвячено порівнянню ефективності найкоротшого шляху Дейкстри в графовій (Neo4j) та реляційній (PostgreSQL) базі даних, де у якості набору даних (dataset) виступає транспортна мережа OpenStreetMap. Метою дослідження є визначення різниці щодо швидкості обчислення алгоритму Дейкстри у наведених вище базах даних. Найпопулярнішою реалізацією найкоротшого шляху в реляційній базі даних є pgRouting, котра реалізована поверх бази даних PostgreSQL. Графові БД вже мають реалізацію найкоротшого шляху в своєму ядрі. Автори дослідження приходять до наступних висновків: графові системи керування даними не є механізмами маршрутизації та не підходять для повного обходу графа, який використовується в обчисленнях найкоротшого шляху. Головне їх призначення – це обхід локального графа властивостей у таких випадках використання, як соціальні мережі, механізми рекомендацій, керування комп'ютерною мережею тощо. В більшості випадків Neo4j БД перевершує pgRouting, але потрібно враховувати використання пам'яті. Адже пам'ять виявилася проблемою при роботі з великими транспортними мережами.

У дослідженні [12] також порівнюється продуктивність графової бази даних Neo4j з продуктивністю реляційної бази даних PostgreSQL. Моделлю даних, яка використовується для перевірки можливостей цих баз даних є TPC -H, що є тестом підтримки прийняття рішень. Він складається з набору бізнес -орієнтованих спеціальних запитів, які заповнюють БД. Продуктивність порівнювалася на основі таких категорій, як: поглинання даних (data ingestion; завантаження даних в БД), розмір на диску, середовище з обмеженим обсягом пам'яті, запити, різні розміри даних. У результаті роботі сформовані висновки на основі категорій, серед яких важливо підкреслити категорію запити та категорію про середовище з обмеженою пам'яттю: PostgreSQL перевершив Neo4j майже в кожному запиті за різних налаштувань (Neo4j показала перевагу лише з об'єднаннями з більш ніж п'ятьома таблицями та для великих наборів даних), з 1 ГБ оперативної пам'яті та з 1 ГБ набору даних Neo4j не вистачило пам'яті для 4 із 7 запитів на відміну від PostgreSQL,

який впорався з всіма запитами. Всі результати гарно проілюстровано в таблицях та рисунках у дослідженні.

Також у мережі Інтернет існує чимало електронних джерел, які показують підходи щодо створення графових структур у мультимодельних базах даних. Такі типи баз даних, зазвичай мають розширення, яке розроблено з ціллю розгортання графової бази даних у даному середовищі. Наприклад, реляційна БД PostgreSQL має дане розширення – Apache AGE, яке надає функціональність графової БД у існуючій реляційній БД [13]. На противагу спеціальним розширенням, деякі інтернет джерела, як наприклад [14] пропонують використовувати PostgreSQL як графову базу даних без додаткових розширень. Для цього автор пропонує приклади, структура яких схожа з прикладами з підручника [10] – створення двох таблиць (окремо для вершин і окремо для ребер) та використання рекурсивних запитів.

1.6 Постановка задачі

Виконати проектування алгоритмів пошуку на графових структурах, що зберігаються в реляційній базі даних.

Для цього потрібно реалізувати наступні задачі, а саме:

1) написати три алгоритми пошуку – пошук в глибину, пошук в ширину та алгоритм Дейкстри за допомогою структурованої мови запитів SQL;

2) створити графову структуру, на якій продемонструвати результати роботи алгоритмів;

3) створити візуальну частину, за допомогою якої продемонструвати та покроково візуалізувати роботу трьох алгоритмів на даних, що зберігаються в реляційній БД;

4) забезпечити візуальну частину інтерактивними засобами – вибір початкової вершини, вибір кінцевої вершини (для алгоритму Дейкстри), вибір одного з трьох алгоритмів та включення анімації.

2 ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

Для реалізації проекту обрано трирівневу архітектуру клієнт-сервер та шаблон MVC (Model-View-Controller).

В інформаційних системах, які використовують бази даних, логічно виділяються 3 основних шари [15]:

а) шар представлення (інтерфейс, який призначений для користувача) – містить програмні засоби, завдяки яких користувач може взаємодіяти з даними;

б) шар бізнес-логіки – включає програмні засоби, в котрих відображаються принципи, правила та залежність поведінки об'єктів предметної області системи;

в) шар даних – містить програмні засоби, які надають дані застосункам, що їх обробляють.

На рис. 2.1 показано трирівневу архітектуру.

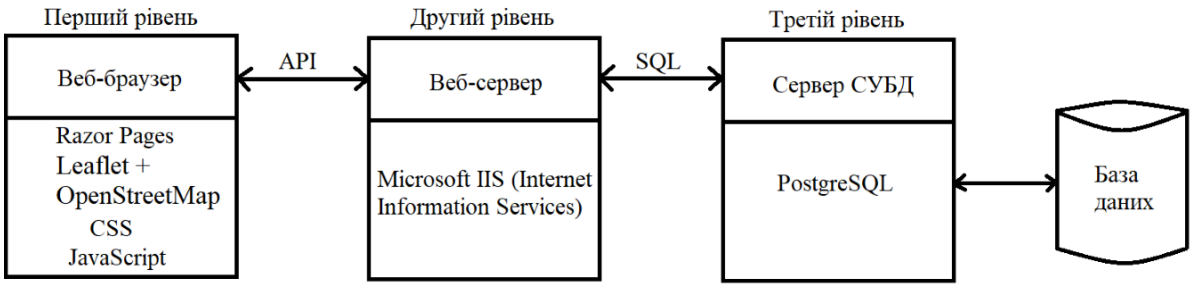


Рисунок 2.1 – Трирівнева архітектура

При використанні шаблону MVC (рис. 2.2) інформаційна система ділиться на три окремі блоки, а саме:

а) Model (Модель) – здійснює маніпулювання даними застосунка, надання даних Представленню і реагування на команди Контролера шляхом зміни свого стану;

б) View (Представлення) – призначений для користувача інтерфейс,

відповідає за отримання необхідних даних з Моделі та відображення їх користувачеві, а також за отримання від користувача команд для роботи з даними і відправку цих команд Контролеру;

в) Controller (Контролер) – відповідає за перетворення команд користувача, отриманих від Представлення, в набір дій над Моделлю з метою її зміни.

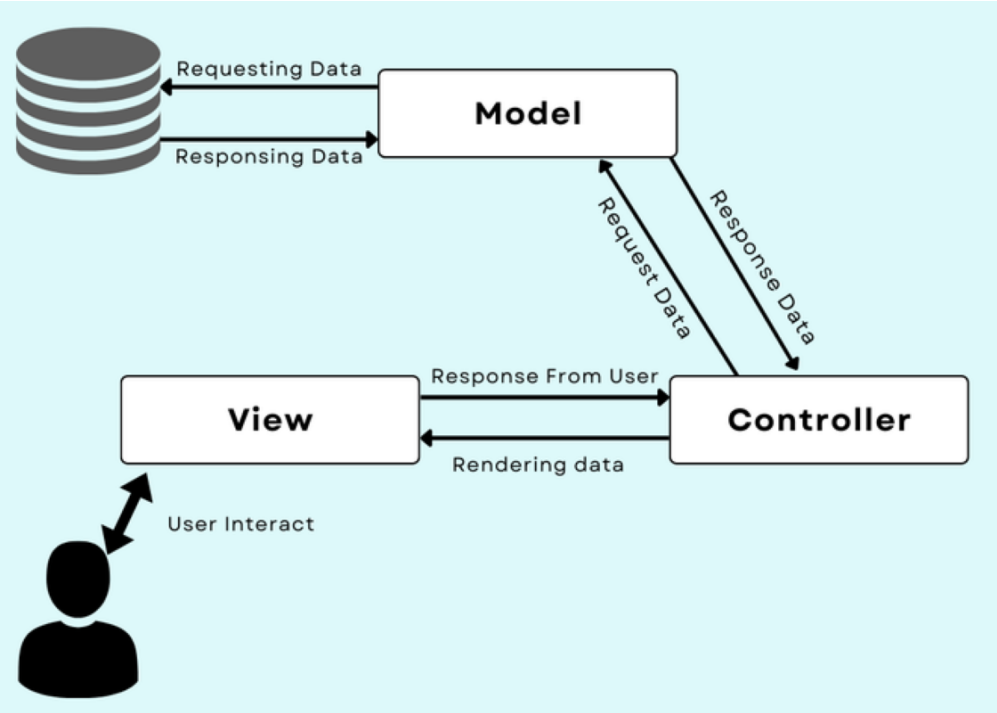


Рисунок 2.2 – Шаблон MVC [16]

2.1 Інформаційне моделювання предметної області

Сутністю є об'єкт (реальний або уявлений) предметної області, інформація про який повинна зберігатися в базі даних.

Кожна сутність характеризується певним наборів атрибутів (кожен з яких має свій тип, який є абстракцією конкретного типу даних, що підтримується СУБД), що відображає її властивості.

Сутності поєднуються між собою за допомогою зв'язків, кожен з яких може мати один з наступних типів [15]:

– один-до-одного (1:1) – коли один екземпляр першої сутності зв'язаний з одним екземпляром другої сутності;

– один-до-багатьох (1:N) – коли один екземпляр першої сутності зв'язаний з декількома екземплярами другої сутності;

– багато-до-багатьох (M:N) – коли один екземпляр першої сутності зв'язаний з декількома екземплярами другої сутності, і один екземпляр другої сутності зв'язаний з декількома екземплярами першої сутності.

У табл. 2.1 наведено опис сутностей для даної ІС.

Таблиця 2.1 – Опис сутностей

Ім'я атрибута	Призначення атрибута	Обмеження
vertex (вершина)		
id_vertex	ідентифікатор вершини	первинний ключ
lat	широта	не порожнє
lon	довгота	не порожнє
name	назва	не порожнє
edge_list (список ребер)		
id_edge	ідентифікатор ребра	первинний ключ
from_node	початкова вершина	зовнішній ключ для зв'язку з сутністю vertex(id_vertex), не порожнє
to_node	кінцева вершина	зовнішній ключ для зв'язку з сутністю vertex(id_vertex), не порожнє
weight	вага	не порожнє

На рис. 2.3 показано діаграму сутність-зв'язок для графової структури.

Розглянемо зв'язок між сутностями vertex та edge_list. Кожна вершина може мати один або багато значень у edge_list. Між даними сутностями існує зв'язок один до багатьох.

Також зв'язок формалізований за допомогою додавання у таблицю edge_list поля from_node та поля to_node, які є зовнішніми ключами для посилання на поле id_vertex таблиці vertex.

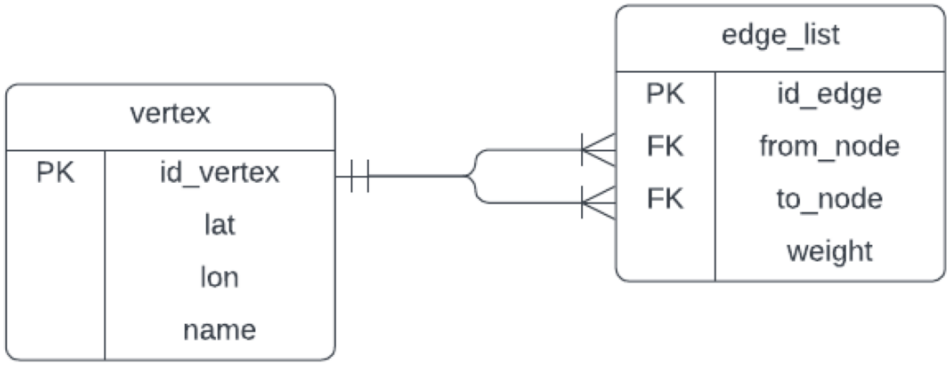


Рисунок 2.3 – Діаграма сутність-зв'язок для графової структури

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Програмна модель застосунку

Програмна модель є веб-застосунком. Можливо зайти через браузер на веб-сторінку, тестувати систему (вводити дані) та ці дані відправлятимуться через веб-сервер Microsoft Internet Information Services до сервера СУБД PostgreSQL, далі сервер шукає потрібні значення у БД.

Відповідна ієрархія веб-сторінок виглядає так: є головна сторінка, через яку можна потрапити до однієї з двох розгалужених гілок (одна для маленької графової структури, а інша для великої), а з них можливо тестувати заданий граф відповідно з алгоритмами пошуку.

3.2 Вибір програмного забезпечення

Для виконання постановки задачі обрано такі інструменти, як:

– СУБД PostgreSQL;

– мова програмування C#;

– технології компанії Microsoft – ASP.NET Core 6 MVC, веб-сервер Internet Information Services;

– карти OpenStreetMap [17];

– Leaflet [18] – JavaScript бібліотека для роботи з картами;

– бібліотека класів Npgsql для роботи з СУБД PostgreSQL;

– CSS;

– JavaScript;

– середовище розробки від компанії Microsoft Visual Studio Community 2022.

СУБД PostgreSQL обрано через декілька причин:

1) є безкоштовною;

2) підтримка рекурсивних запитів (WITH RECURSION);

3) є велика і детальна документація.

Мова C# – об'єктно-орієнтована мова програмування, що вбудована у програмну технологію .NET Framework. ASP.NET Core 6 (де вбудовано Razor Pages) – є частиною технології .NET Framework, багатофункціональна платформа для створення веб-додатків, має наступні переваги:

1) безкоштовна платформа;

2) вбудовано шаблон проектування Model-View-Controller;

3) підтримка стандартної розмітки HTML, яка об'єднана разом з бекендом-класу, тобто мови C# (за допомогою фреймворка Razor Pages);

4) використовує вбудований веб-сервер Internet Information Services (IIS), що є досить керованим та автономним веб-сервером.

Карти OpenStreetMap обрано завдяки наступним перевагам:

1) є безкоштовними;

2) гарно працюють з JavaScript бібліотекою Leaflet;

3) зовнішній вигляд є наочним і зрозумілим.

Leaflet обрано через те, що:

1) є платформою з відкритим кодом;

2) є велика та докладна документація;

3) наявність різноманітних плагінів;

4) користується значною популярністю для створення інтерактивних карт.

Npgsql є постачальником даних ADO.NET (постачається з Microsoft .NET Framework) з відкритим кодом для PostgreSQL, дозволяє програмам отримувати доступ до сервера бази даних PostgreSQL.

CSS – для дизайну сторінок, обрано через сумісність з різними пристроями, поєднується з Razor Pages для розробки зрозумілого інтерфейсу.

Javascript використовується для роботи з бібліотекою Leaflet, для написання скриптів функцій.

Середовище Microsoft Visual Studio Community обрано через те, що воно є безкоштовним та дозволяє розробити веб додаток, у якому об'єднати

всі вибрані технології.

3.3 Створення баз даних

Створимо базу даних, у якій зберігатимемо невелику графову структуру. Для створення бази даних потрібно написати команду CREATE TABLE, у лапках вказавши ім'я бази даних.

```
CREATE DATABASE "Graph";
```

Для того, щоб створити граф створимо дві таблиці – перша для вершин, а друга для ребер. За допомогою SQL запиту (лістинг 3.1) створено таблицю для вершин, яка складається з: унікального ідентифікатора (id_vertex) – є первинним ключем, широти (lat), довготи (lon) та назви (name).

```
CREATE TABLE vertex(
id_vertex serial not null primary key,
lat varchar(10) not null,
lon varchar(10) not null,
name varchar(25) not null
)
```

Лістинг 3.1 – SQL-запит для створення таблиці vertex

Таблиця ребер графа (лістинг 3.2), а саме: унікальний ідентифікатор (id_edge), який є первинним ключем, початковий вузол (from_node), кінцевий вузол(to_node), вага (weight), далі обов'язково перевіряємо умову нерівності початкового вузла щодо кінцевого – для запобігання петель.

```
CREATE TABLE edge_list(
id_edge serial not null primary key,
from_node integer not null,
to_node integer not null,
weight integer not null,
check(from_node != to_node),
foreign key (from_node) references vertex (id_vertex) on delete
restrict on
```

Лістинг 3.2 – SQL-запит для створення таблиці edge_list

```
update cascade,
foreign key (to_node) references vertex (id_vertex) on delete
restrict on
update cascade
)
```

Лістинг 3.2, аркуш 2

Таблиця vertex є батьківською, а таблиця edge_list є дочірньою таблицею, у якій два поля from_node та to_node є зовнішніми ключами для посилання на поле id_vertex таблиці vertex. Створимо іншу базу даних – для зберігання великої графової структури. Створимо базу даних з іншим іменем, використовуючи попередній запит для створення БД:

```
CREATE DATABASE "BigGraph";
```

Таблиця для вершин, яка дещо відрізняється від іншої в попередній БД (лістинг 3.3).

```
CREATE TABLE vertex(
id_vertex bigint not null primary key,
lon varchar(30) not null,
lat varchar(30) not null
)
```

Лістинг 3.3 – SQL-запит для створення таблиці vertex БД "BigGraph"

Таблиця tmp (лістинг 3.4) – для зберігання проміжних даних.

```
CREATE TABLE tmp(
id varchar(30),
osm_id bigint,
source bigint,
target bigint,
length real,
foot varchar(30),
car_forward varchar(30),
car_backward varchar(30),
bike_forward varchar(30),
bike_backward varchar(30),
train varchar(30),
wkt text )
```

Лістинг 3.4 – SQL-запит для створення таблиці tmp

Запит на створення таблиці для ребер співпадає з лістингом, але замінимо тип даних integer у полях from_node, to_node на bigint.

3.4 Заповнення баз даних

Спочатку виконаємо заповнення бази даних "Graph". Всього вершин 13, наведемо приклади заповнення деяких з них:

```
INSERT INTO vertex(lat,lon,name) VALUES('50.722895', '-3.525101',
'Exeter');
INSERT INTO vertex(lat,lon,name) VALUES('51.507289', '-0.127270',
'London');
INSERT INTO vertex(lat,lon,name) VALUES('51.212734', '0.982178',
'Kent Downs');
```

Створимо структуру графа, додавши відношення між вузлами (початковий вузол, тобто звідки та кінцевий вузол, тобто куди) та вагу, заповнивши таблицю edge_list:

```
INSERT INTO edge_list (from_node, to_node, weight) VALUES (1, 2,
252680);
INSERT INTO edge_list (from_node, to_node, weight) VALUES (2, 3,
83705);
INSERT INTO edge_list (from_node, to_node, weight) VALUES (3, 1,
320237);
```

Для того, щоб заповнити базу даних "BigGraph" потрібно виконати низку дій. Спершу, завантажемо датасет (набір даних) звідси [19] – це сервер Geofabrik для безкоштовного завантаження відкритих даних з OpenStreetMap. Результатом завантаження є файл типу pbf, що необхідно перетворити на графову структуру. Для цього завантажемо [20] мову rust, яка потрібна для завантаження [21] osm4routing2:

```
cargo install osm4routing
```

Відкриваємо командний рядок (cmd) в папці, куди помістили pbf файл, потім прописуємо команду:

```
osm4routing monaco-latest.osm.pbf
```

Внаслідок отримуємо дві таблиці у форматі csv – nodes (вершин) та edges (ребер). Щоб імпортувати отримані дані, використовується команда [22]:

```
copy vertex from 'C:/шлях до таблиці/nodes.csv' delimiter ',' CSV
HEADER;
```

За цим принципом, імпортуємо файл edges до таблиці tmp, а потім з неї додаємо лише 3 поля (source, target, length) до таблиці edge_list, з перевіркою умови наявності петель – якщо вони є, то даний кортеж пропускається:

```
INSERT INTO edge_list(from_node, to_node, weight) SELECT source,
target, length FROM tmp WHERE source != target ON CONFLICT DO
NOTHING
```

Наступний запит видаляє таблицю tmp:

```
DROP TABLE tmp
```

Для перевірки та візуалізації роботи алгоритмів достатньо залишити 4000 кортежів з початкового 5079 у таблиці ребер, тому робимо запит:

```
DELETE FROM edge_list WHERE id_edge > 4000
```

3.5 Реалізація алгоритмів

Всі алгоритми пошуку реалізовано за допомогою тимчасових таблиць, рекурсивних запитів WITH RECURSIVE [23] та функцій на мові plpgsql (розширення мови SQL).

За допомогою оператора RECURSIVE відбувається звернення запиту до власних результуючих даних, забезпечуючи їх рекурсивну обробку. Запит можна назвати скоріше ітераційним, але комітетом зі стандартизації SQL вирішено залишити поняття RECURSIVE [9].

Даний алгоритм складається з двох частин: перша частина (ще називають якорем) є нерекурсивною та виконується один раз (повертає один

рядок з вихідною точкою ієрархії або частини ієрархії); друга частина є рекурсивною, де є посилання на власний результат запиту (на тимчасову таблицю). Дві частини поєднуються оператором UNION ALL (повертає всі значення, які можуть повторюватися, тобто дублікати) або UNION (повертає унікальні значення, відкидаючи дублікати).

Для створення алгоритмів знадобляться тимчасові таблиці [24] – для збереження проміжного результату з рекурсивних запитів. Тимчасова таблиця – це таблиця, визначення та дані якої видимі лише в межах поточного сеансу чи транзакції. Дані таблиці створюються та керуються подібно до звичайних таблиць, але головною відмінністю є те, що вони автоматично видаляються в кінці сеансу або транзакції. Також тимчасові таблиці можна видалити у явному виді, використовуючи наступну команду:

```
DROP TABLE ім'я_таблиці;
```

Для об'єднання всіх зазначених дій (створення рекурсивних запитів та тимчасових таблиць з можливістю вибору певної початкової вершини, а для алгоритму Дейкстри ще й кінцевої) потрібно все додати до створених функцій. Створення функцій відбувається за допомогою визначення CREATE OR REPLACE FUNCTION, де в залежності від алгоритму, функція може приймати одне або декілька вхідних значень. У якості результату функція повертатимете таблицю значень.

3.5.1 Пошук в глибину (Depth first search, DFS)

Створимо функцію DFS (див. додаток А, лістинг А.1), яка приймає значення start_node цілого типу у якості параметра (обхід відбуватиметься з номера введеної вершини), тобто з якого початкового вузла розпочати обхід графової структури. Функція повертає результат в таблицю з наступними полями: унікальний ідентифікатор кортежу (id), кінцевий вузол (to_node) та назва вершини (name).

Спочатку створюється тимчасова таблиця tmp_dfs з двома полями (унікальний ідентифікатор та кінцевий вузол), у яку в подальшому занесено результати рекурсивної функції. Далі йде рекурсивна фунція search_dfs [25] з параметрами from_node, to_node, path та have_cycle. Відповідно початкова та кінцева вершина, шлях (є числовим масивом, до якого додаються кінцеві вершини) та параметр для визначення циклу. Якщо відкинути have_cycle, то обхід не зможе завершитися та утвориться нескінченний цикл, з чого слідує наступне – обхід працюватимете тільки для ациклічного графа. Тому для розробки алгоритму, що має змогу визначати цикл, додається параметр типу boolean – може приймати значення або true (істина), або false (хиба). Тобто обчислюємо масив вже відвіданих значень [26]. Результатом цього рекурсивного запиту є таблиця з чотирма атрибутами (параметрами search_dfs), яку відсортовано за шляхом (path), де показано детально як відбувався обхід за алгоритмом пошук в глибину. Ці дані, за допомогою запиту INSERT, додаємо до тимчасової таблиці. Потім, за допомогою оператора SELECT, обираємо ідентифікатор (id_tmp) з мінімальним значенням і кінцевий вузол з тимчасової таблиці (за допомогою CONCAT об'єднуємо значення широти та довготи з псевдонімом "coords" для повернення у вигляді to_node типу TEXT) та назву з таблиці вершин, виконуємо внутрішнє з'єднання (INNER JOIN). Групуємо однакові рядки to_node і name з використанням GROUP BY та сортуємо за ідентифікатором, за замовчуванням у порядку зростання (ORDER BY); якщо потрібно у порядку спадання, то після ідентифікатора ставиться DESC. Ці результати повертає функція DFS. Далі видаляємо тимчасову таблицю (DROP TABLE).

Для реалізації пошука в глибину для великого графа з БД "BigGraph" запит дещо відрізняється (тому що дані зберігаються у вигляді типу bigint), відбувається зміна типів з integer на bigint – start_node приймає у якості параметра тип bigint, у тимчасовій таблиці to_node має тип bigint, а в рекурсивному запиті шлях приймає має значення bigint.

3.5.2 Пошук у ширину (Breadth first search, BFS)

Структура даного пошуку схожа з попереднім алгоритмом. Створюємо функцію BFS (лістинг А.2), яка приймає номер введеної вершини у якості параметра. Створюємо тимчасову таблицю tmp_bfs, рекурсивний запит search_bfs. Результатом цього рекурсивного запиту є таблиця з п'ятьома атрибутами (параметрами search_bfs), яку відсортовано за глибиною (depth), де містяться дані того, як відбувався обхід за алгоритмом пошуку в ширину. Далі обираємо дані з тимчасової таблиці (ідентифікатор з мінімальним значенням, кінцевий вузол – об'єднання значень широти та довготи за допомогою функції CONCAT), з таблиці вершин обираємо назву, виконуємо внутрішнє з'єднання та групуємо за кінцевою вершиною (to_node) та назвою (name), сортуємо у порядку зростання за ідентифікатором (id_tmp). Видаляємо тимчасову таблицю.

Для реалізації пошука в ширину для графової структури з БД "BigGraph" запит також зазнає змін: відбувається зміна типів з integer на bigint.

3.5.3 Алгоритм Дейкстри (Dijkstra's Algorithm)

Створимо функцію DIJKSTRAS (лістинг А.3), яка має наступні вхідні параметри – номер початкової вершини (start_node) та номер кінцевої вершини (end_node). Функція повертає таблицю з такими полями: from_node (координати початкової вершини), from_name (назва початкового вузла), to_node (координати кінцевої вершини), to_name (назва кінцевого вузла), weight (вага), path (шлях у координатах), path_name (шлях у вигляді назв).

Створена тимчасова таблиця матимете наступні поля: ідентифікатор (первинний ключ), номер початкової вершини, номер кінцевої вершини, вага, шлях (масив номерів кінцевих вершин у числовому значенні). Створимо рекурсивний запит dijkstras, де з обраного початкового вузла (start_node)

починається обхід. У другій частині рекурсивного запиту посилаємось на значення from_node на власний результат запиту, а to_node беремо з таблиці edge_list – для того, щоб запит шукав тільки з цієї початкової вершини. Також сумуємо поточну вагу з попередньою. Обираємо з результату рекурсивного запиту значення, ставимо умову WHERE (кінцевий вузол дорівнює значенню end_node), сортуємо за зростанням за вагою, ставимо ліміт 1 та це все додаємо до тимчасової таблиці tmp_dijkstras за допомогою оператора INSERT. Повертаємо запит до результуючої таблиці, вибравши всі значення з підзапиту tt, шлях у координатах та шлях у вигляді назв. Для того, щоб перевести числовий масив path (з тимчасової таблиці) у символьний масив path та символьний масив path_name, потрібно скористатися наступним [27]: використати UNNEST(tmp_dijkstras.path) WITH ORDINALITY AS "paths" (WITH ORDINALITY – до стовпців результату функції додається додатковий стовпець типу bigint, у цьому стовпці нумеруються рядки набору результатів функції, починаючи з 1 [28]), зробити внутрішнє з'єднання (INNER JOIN) таблицею vertex та вибрати шлях у координатах та шлях у вигляді назв, відповідно ARRAY_AGG(CONCAT(v.lat, ', ', v.lon) ORDER BY ORDINALITY) AS "path" та ARRAY_AGG(to_p.name ORDER BY ORDINALITY) AS "path_name". Тобто функція UNNEST бере кожен елемент масиву та додає його до окремого рядка. Далі відбувається з'єднання за первинним ключем, а потім ARRAY_AGG збирає результати до масиву. Об'єднання значень широти та довготи відбувається за допомогою функції CONCAT. Також видаляємо тимчасову таблицю.

Реалізація алгоритму Дейкстри також змінюється для графа з БД "BigGraph" – зміна типів на bigint.

3.6 Програмна реалізація інтерфейсу

Веб-застосунок побудовано з використанням шаблону MVC. Коли вводимо дані для тестування (номер вершини, натискаємо на кнопки і т.д.),

то це все відбувається у представленні (Razor Page), яке надсилає дані до контролера для обробки. Всього представлень дев'ять: чотири для маленького графа, чотири для великого та одне для головної сторінки. Всього контролерів є два (для невеликого графа – GraphController, а для великого BigGraphController) та, відповідно, моделі є дві (Graph та BigGraph), а також додаткова модель DataAccess для підключення до БД. Кожен з контролерів є спадкоємцем абстрактного класу Controller – базовий клас для контролера MVC.

Наведено код програми контролера (лістинг 3.5), де обирається відповідне перенаправлення до іншого методу, в залежності від алгоритму, який обрали у представленні. Кожен метод має своє представлення, наприклад метод Algorithm_dfs (для алгоритму пошуку в глибину) має представлення з аналогічною назвою, яке демонструє відповідні результати з методу, так само як метод Algorithm_bfs (для алгоритму пошуку у ширину) має представлення для відображення результатів з методу Algorithm_bfs.

У даному прикладі, показано метод Algorithm, в якому перевіряються умови (в залежності від обраного алгоритму, відбуватиметься перенаправлення до належного методу). Якщо обрали алгоритм Дейкстри, тоді після перевірки попередніх умов, створюємо об'єкт класу DataAccess (створює кожен метод контролера) та використовує його метод для з'єднання до БД (GetConnection) та виконуємо запит, котрий поверне результат, що відобразимо у представленні.

```
     public IActionResult Algorithm(Graph algo)
        {
            if(algo.algorithm == "Dfs" && algo.from_node != 0)
            {
                return RedirectToAction("Algorithm_dfs", "Graph",
new { from_node = algo.from_node });
            }
            if (algo.algorithm == "Bfs")
            {
                return RedirectToAction("Algorithm_bfs",
```

Лістинг 3.5 – Код методу Algorithm

```
"Graph", new { from_node = algo.from_node }); }

            List<Graph> display_gr = new List<Graph>();
            var connect = new DataAccess();
            using (var con = connect.GetConnection()) {
                con.Open();
                string sql = "select * from dijkstras('" +
algo.from_node + "', '" + algo.to_node + "')";
                using var a = new NpgsqlCommand(sql, con);
                using NpgsqlDataReader reader =
a.ExecuteReader();
                while (reader.Read())
                {
                    var list = new Graph();
                    list.from_coords =
Convert.ToString(reader["from_node"]);
                    list.from_name =
Convert.ToString(reader["from_name"]);
                    list.to_coords =
Convert.ToString(reader["to_node"]);
                    list.to_name =
Convert.ToString(reader["to_name"]);
                    list.weight =
Convert.ToInt32(reader["weight"]);
                    var arr = reader.GetValue("path") as
string[];
                    list.path = arr;
                    var arr2 = reader.GetValue("path_name") as
string[];
                    list.path_name = arr2;
                    display_gr.Add(list); }
                con.Close(); }
            return View("Dijkstra_algorithm", display_gr); }
```

Лістинг 3.5, аркуш 2

Для виведення відправки з БД до читання контролерами результатів запитів, а також відображення графової структури у представленні, написано додаткові SQL-запити (наведені у додатку Б).

Для того, щоб прочитати граф з БД, написано функцію get_graph (див. додаток Б, лістинг Б.1), яка повертає таблицю з наступними результатами – координати початкової та кінцевої вершин, назву початкової та кінцевої вершини, вагу між вузлами. Це відбувається за допомогою оператора вибірки (SELECT) з функцією конкатенації CONCAT для початкової та кінцевої вершини, далі відбувається два внутрішніх з'єднання (перше – для початкової

вершини, а друге – для кінцевої).

Для того, щоб мати змогу побачити тільки ті вузли, з яких можливо почати обхід (тобто обрати початкову вершину, з якої точно є один або більше певних шляхів), написано функцію from_nodes (лістинг Б.2). Функція повертає номер вершини разом з назвою. Це відбувається з використанням оператора вибірки з оператором DISTINCT – повертає лише унікальні кортежі, дублікати видаляє. Потім йде внутрішнє з'єднання.

Для повернення лише тих вузлів, до яких можна потрапити (тобто обрати кінцеву вершину, до якої точно є один або більше певних шляхів) написано функцію to_nodes (лістинг Б.3). Функція схожа з функцією from_nodes, але повертає кінцеві номера вершини разом з назвою.

Функція get_big_graph (лістинг Б.4) призначена для відображення великої графової структури, де у вершини немає назви, тому, на відміну від функції для відображення невеликого графа (функція get_graph), тип даних стає BIGINT, а повернена таблиця містить координати початкової та кінцевої вершини та вагу. Аналоги у вигляді функцій from_nodes та to_nodes для даного графа не потрібні, так як він не має назви для кожної вершини.

3.7 Проектування за допомогою Leaflet

Щоб мати змогу працювати з картами та відображати їх, до заголовка представлення потрібно додати необхідні посилання – посилання на leaflet файл css та js (лістинг 3.6).

```
    <link rel="stylesheet"
href="https://unpkg.com/leaflet@1.9.4/dist/leaflet.css"
integrity="sha256-p4NxAoJBhIIN+hmNHrzRCf9tD/miZyoHS5obTRR9BMY="
crossorigin="" />
    <script
src="https://unpkg.com/leaflet@1.9.4/dist/leaflet.js"
integrity="sha256-20nQCchB9co0qIjJZRGuk2/Z9VM+kNiyxNV1lvTlZBo="
crossorigin=""></script>
```

Лістинг 3.6 – Посилання для завантаження та роботи з картами

Потім у body потрібно додати:

```
<div id="map"></div>
```

Далі створимо скрипт і додамо код для ініціалізації карти на веб-сторінку (лістинг 3.7) з додаванням шару OpenStreetMap до карти.

```
var map = L.map('map').setView([51.505, -0.09], 7);

L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
            attribution: '&copy; <a
href="https://www.openstreetmap.org/copyright">OpenStreetMap</a>
contributors' }).addTo(map);
```

Лістинг 3.7 – Код для завантаження карти

Після зчитування з БД, дані зберігаються у моделі та передаються до представлення типу Razor Pages. Щоб потім працювати з цими даними з використанням Leaflet, зробимо серіалізацію Model у формат JSON:

```
var data = @Html.Raw(Json.Serialize(Model));
```

У циклі (лістинг 3.8) зчитуємо координати та назви початкової та кінцевої вершини, вагу. За допомогою circleMarker на карті з'являтимуться кругові маркери зеленого кольору з відповідною позначкою у відповідній вершині. Для того, щоб намалювати лінію між двома координатами, використовується полілінія (polyline), яка містить отримані значення координат початкової та кінцевої вершини – sp1, sp2. Для додавання стрілок напрямків використано плагін Leaflet PolylineDecorator [29].

```
for(i in data){
            var item = data[i];
            var sp1 = item.from_coords.split(",");
            var sp2 = item.to_coords.split(",");
            var fr = item.from_name;
            var to = item.to_name;
            var w = item.weight;
            let x = L.circleMarker(sp1, { color: 'green', fill:
'green' }).addTo(map).bindTooltip('Name: ' + fr + '<br/>' + 'lat,
long: ' + sp1,
```

Лістинг 3.8 – Код для завантаження графа на карту

```
                    {
                        permanent: true,
                        direction: 'top'
                    });
                let y = L.circleMarker(sp2, { color: 'green', fill:
'green' }).addTo(map).bindTooltip('Name: ' + to + '<br/>' + 'lat,
long: ' + sp2,
                    {
                        permanent: true,
                        direction: 'top'
                    });

            polyline1 = L.polyline([sp1, sp2], {
                color: 'red'
                    }).addTo(map).bindTooltip(w + (' meters'), {
                        permanent: true,
                        direction: 'top'
                    });
                    L.polylineDecorator(polyline1, {
                        patterns: [
                            {
                                offset: 10,
                                repeat: 100,
                                symbol: L.Symbol.arrowHead({
                                    pixelSize: 10,
                                    pathOptions: {
                                        color: 'black',
                                        fillOpacity: 1,
                                        weight: 0,
                                    },
                                }),
                            },
                        ],
                    }).addTo(map);
        }
```

Лістинг 3.8, аркуш 2

Анімація роботи алгоритмів (лістинг 3.9) відбувається завдяки наступному: до веб-сторінки додамо кнопку, при натисканні на яку спрацюватиме функція play_animation(), внаслідок кнопка стане неактивною (більше натиснути на неї не зможемо).

Додамо отримані результати роботи певного алгоритму до масиву, потім в циклі зчитуємо координати та додаємо круговий маркер жовтого кольору до карти, функція призупиняє своє виконання на одну секунду (за допомогою функції sleep [30]), а потім знову продовжуватиме обробляти

масив. Таким чином, відбувається анімація на карті.

```
function sleep(ms) {
            return new Promise(resolve => setTimeout(resolve,
ms))
        }
        async function play_animation() {
            var bu = document.getElementById('btn_play');
            bu.disabled = true;
            var arr = [];
            for (var j in data) {
                if (data[j].coordinates != null) {
                    arr.push(data[j].coordinates.split(","));
                }
            }
            var n = 1;
            for (var x = 0; x < arr.length; x++) {
                var markerA = L.circleMarker(arr[x], { color:
'yellow', fill: 'yellow' }).addTo(map).bindTooltip('Number: ' +
n,
                    {
                        permanent: true,
                        direction: 'top'
                    });
                    n++;
                await sleep(1000);
            }
        }
```

Лістинг 3.9 – Код функції анімації та призупинення роботи

3.8 Інструкція користувача

При вході на головну сторінку інформаційної системи (рис. 3.8) можемо побачимо карту OpenStreetMap – на яку можна натискати, створюватиметься новий маркер та у полі нижче з'являтимуться його координати; всі значення – координати у полі, маркер на карті оновлюватимуться тоді, коли нове клацання по мапі відбуватиметься. Також на веб-сторінці наведено короткий опис алгоритмів, які використовуються для пошуку на графових структурах у реляційній БД [31], та дві кнопки з посиланнями для переходу до прикладів – граф із бази даних "Graph" ("SMALL GRAPH") та граф із бази даних "BigGraph" ("BIG GRAPH").

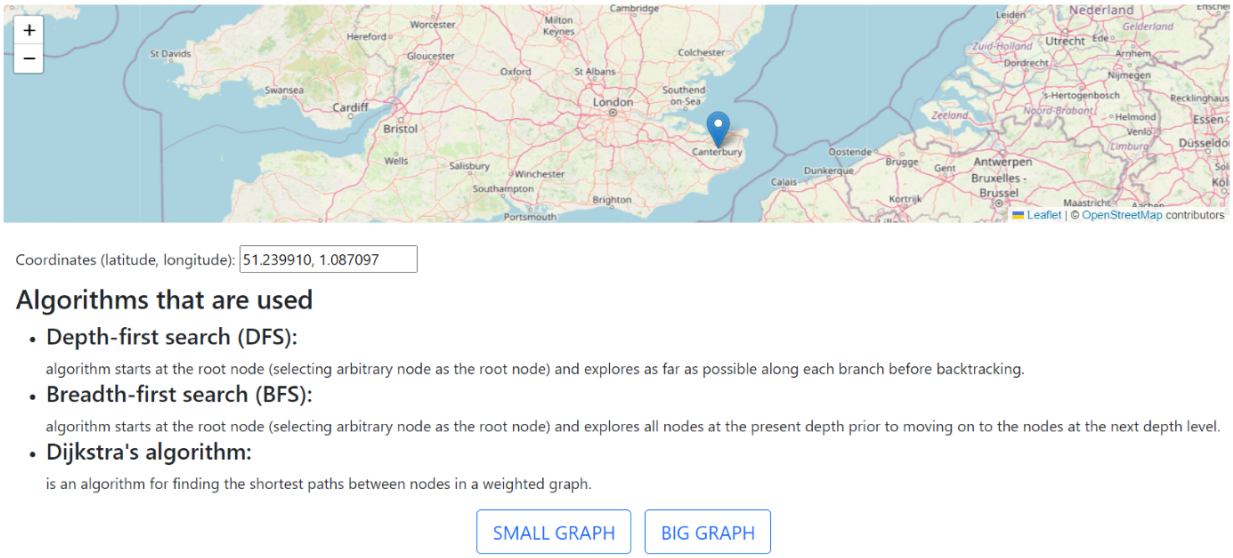


Рисунок 3.8 – Головна сторінка

Натиснемо на кнопку "SMALL GRAPH", яка завантажить сторінку з першим прикладом. На рис. 3.9 показано карту з графом, де в правій частині вікна можна обрати алгоритм, початкову вершину та кінцеву.

Рисунок 3.9 – Граф із БД "Graph"

На рис. 3.10 показано всі доступні початкові вершини (завдяки функції from_nodes), а на рис. 3.11 всі доступні кінцеві вершини (за допомогою функції to_nodes).

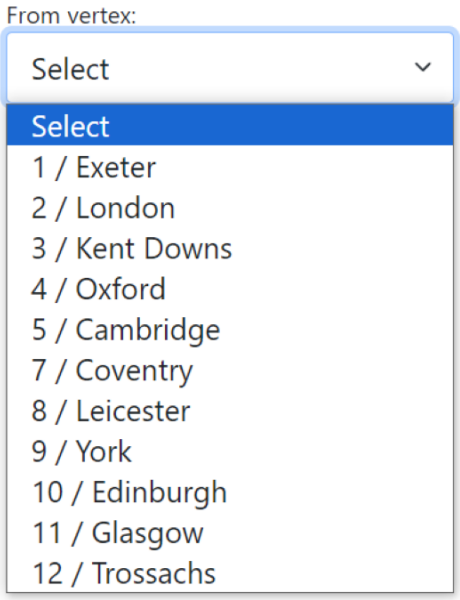


Рисунок 3.10 – Список початкових вершин

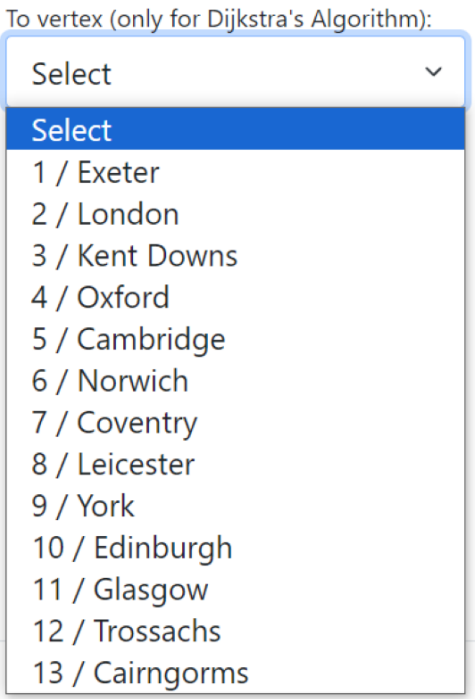


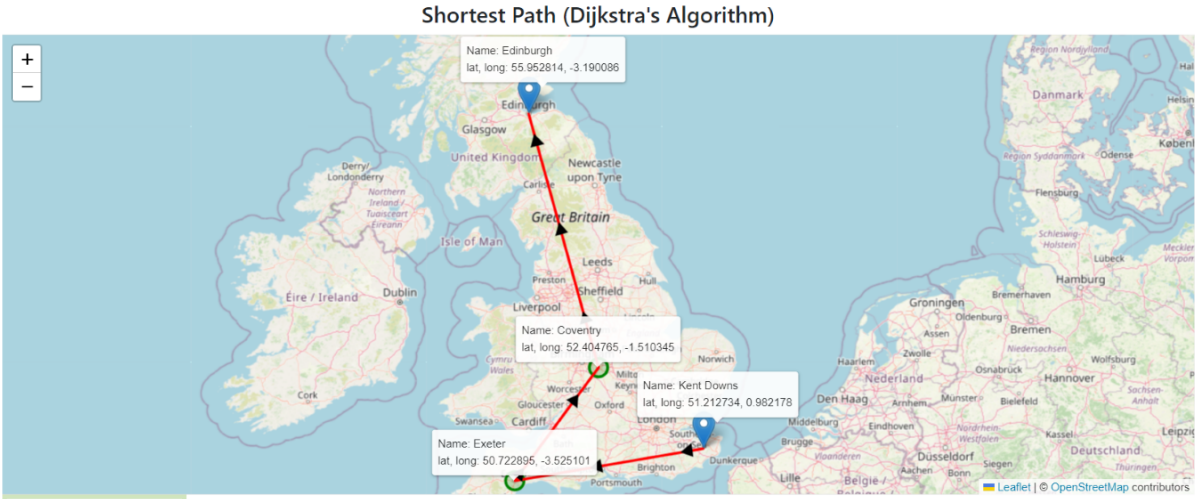
Рисунок 3.11 – Список кінцевих вершин

Оберемо пошук в глибину з вершини з номером один (Exeter). Вводимо алгоритм пошуку DFS та обираємо вершину. Результати пошуку показано на рис. 3.12. Також перевіримо роботу алгоритму пошуку у ширину (рис. 3.13) з цієї ж початкової вершини, але тепер вводимо алгоритм пошуку BFS.

Рисунок 3.12 – Пошук в глибину (з БД "Graph")

Рисунок 3.13 – Пошук в ширину (з БД "Graph")

Перевіримо роботу алгоритму Дейкстри (рис. 3.14).



From point	From name	To point	To name	meters	Path	Path Name
51.212734, 0.982178	Kent Downs	55.952814, -3.190086	Edinburgh	962748	50.722895, -3.525101; 52.404765, -1.510345; 55.952814, -3.190086	Exeter; Coventry; Edinburgh

Рисунок 3.14 – Алгоритм Дейкстри (з БД "Graph")

Візуалізуємо графову структури з бази даних "BigGraph" (рис. 3.15). На головній сторінці натиснемо на кнопку "BIG GRAPH".

Як бачимо, на кругові маркери на мапі можна натиснути та отримати id вершини, а також досить навести на ребро та з'явиться значення відстані у метрах.

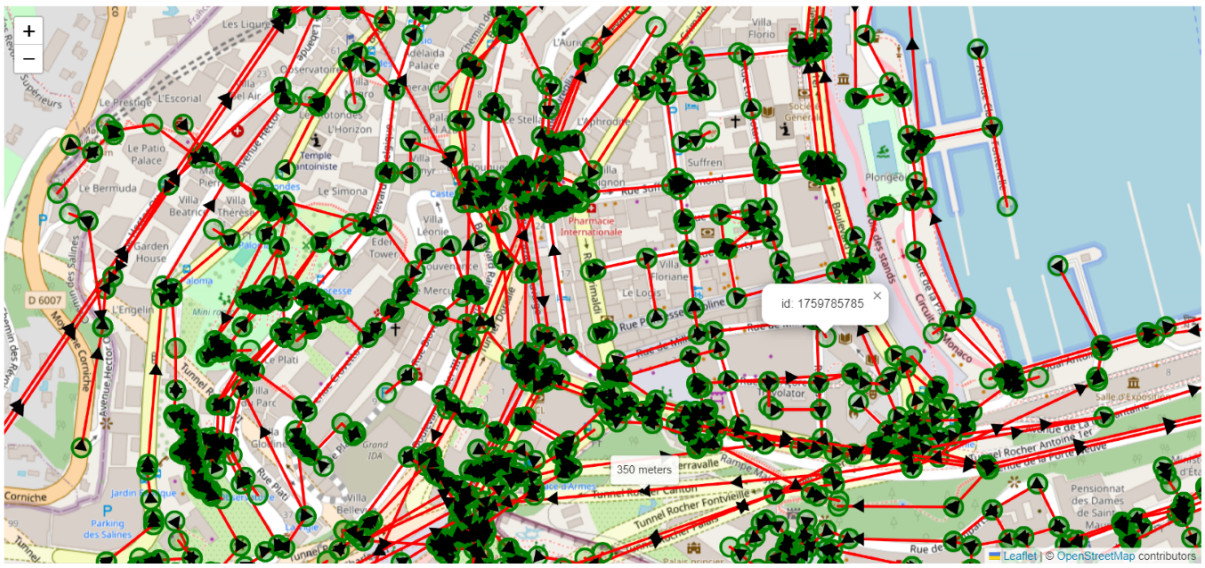


Рисунок 3.15 – Графова структура із БД "BigGraph"

Спочатку, перевіримо роботу алгоритму в глибину з вершини з номером 21912089. Результат показано на рис. 3.16.

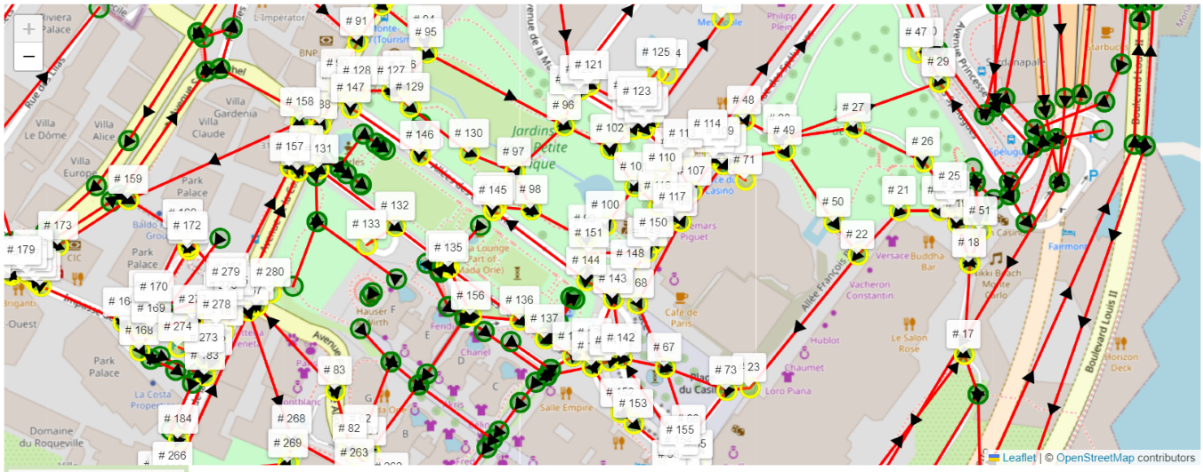


Рисунок 3.16 – Пошук в глибину (з БД "BigGraph")

Перевіримо роботу алгоритму в ширину з цієї ж вершини (21912089). Результат показано на рис. 3.17.

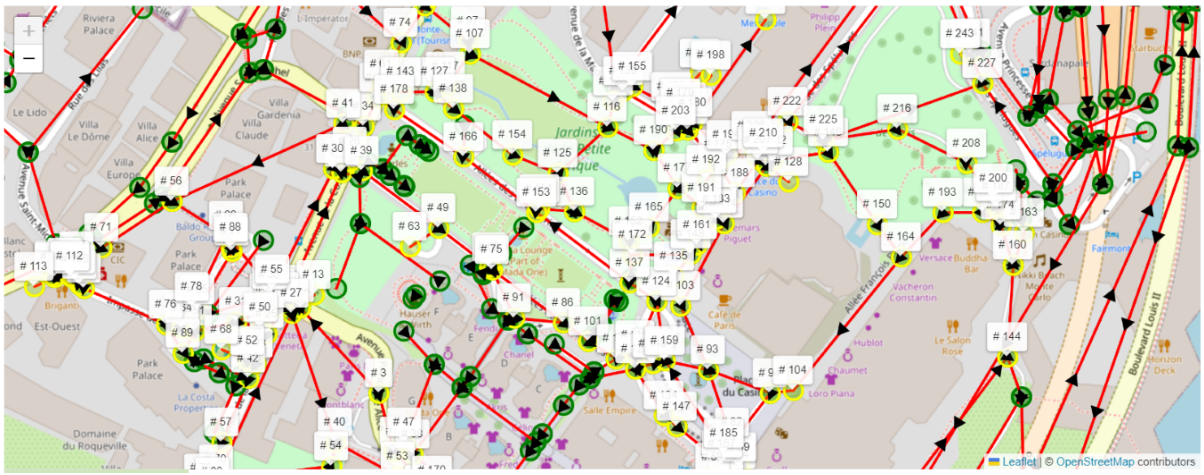


Рисунок 3.17 – Пошук в ширину (з БД "BigGraph")

Як бачимо, мітки послідовностей (шлях, як алгоритм робить обхід), при алгоритмі пошуку в ширину, суттєво відрізняються від значень міток (значення знаходяться набагато розрізнено) при алгоритмі пошуку в глубину.

Протестуємо алгоритм пошуку мінімального шляху (алгоритм

Дейкстри) на графовій структурі з бази даних "BigGraph" (рис. 3.18).

На веб-сторінці можемо побачити шлях на карті, де стрілками вказано напрям. Шлях вже проанімовано, а в таблиці нижче (на рис. 3.19) на веб-сторінці вказано деталі шляху (координати початкового вузла, координати кінцевого вузла, мінімальна вага у метрах та шлях).

Рисунок 3.18 – Алгоритм Дейкстри (з БД "BigGraph")

From point	To point	meters	Path
43.7389205, 7.425995899999999	43.7375964, 7.4291113	1024	43.7390833, 7.4257696; 43.739328199999996, 7.4256829; 43.739686, 7.4252293; 43.739721599999996, 7.4251499; 43.7396877, 7.4251233; 43.7395782, 7.425037199999999; 43.739454599999995, 7.4249689; 43.7393895, 7.4249268; 43.7391242, 7.4247735; 43.7389286, 7.424656; 43.7388584, 7.4246137; 43.7387969, 7.4245768; 43.7383814, 7.4242856999999995; 43.738226399999995, 7.424134899999999; 43.737977699999995, 7.423779499999999; 43.7377714, 7.4230075; 43.737615399999996, 7.422997; 43.7375276, 7.4234241; 43.73753, 7.423582; 43.737092399999995, 7.4238309; 43.737066, 7.423836799999999; 43.7371579, 7.4247736; 43.7371701, 7.4249187; 43.737205599999996, 7.4253078; 43.7372239, 7.4254945999999995; 43.7373052, 7.4261751; 43.7374191, 7.4268434999999995; 43.7373583, 7.426864999999999; 43.7375865, 7.4277906; 43.7376459, 7.427981099999999; 43.737696899999996, 7.4289350999999995; 43.737694399999995, 7.4289637; 43.737801399999995, 7.4291089999999995; 43.7378292, 7.4291468; 43.7375964, 7.4291113

Рисунок 3.19 – Таблиця результатів алгоритму Дейкстри (з БД "BigGraph")

ВИСНОВКИ

Реалізовано класичні графові алгоритми для даних, що зберігаються в базах даних, результат продемонстровано у вигляді побудованої інформаційної системи.

Реалізовано три графові алгоритми пошуку (пошук в глибину, пошук в ширину, алгоритм Дейкстри) за допомогою структурованої мови запитів SQL з використанням даних, що зберігаються в реляційній базі даних PostgreSQL.

Із застосуванням карт OpenStreetMap та бібліотеки Leaflet візуалізовано та анімовано пошук в глибину, пошук в ширину, алгоритм Дейкстри, початкові дані для яких можна задати в інтерактивному режимі.

Для перевірки написаних алгоритмів створено дві бази даних – для маленької графової структури (13 вершин, 16 ребер) та для великої графової структури (3859 вершин, 4000 ребер).

Результати роботи можуть застосовуватись для автоматизованого аналізу графових алгоритмів. Систему протестовано на різній кількості вершин та ребер, які утворюють графову структуру; алгоритми працюють коректно.

Розроблена система може застосовуватись у протоколах маршрутизації (OSPF, IS-IS) для візуалізації структури мережевих вузлів, які можуть зберігатися в базі даних.

Під час виконання дипломної роботи написано та опубліковано тези на всеукраїнській конференції.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Кузьменко І.М. Теорія графів: навч. посіб. для здобувачів ступеня бакалавра за освітньою програмою «Комп'ютерний моніторинг та геометричне моделювання процесів і систем» спеціальності 122 «Комп'ютерні науки» [Електронний ресурс] / Київ: КПІ ім. Ігоря Сікорського, 2020. – 71 с.

2. Introduction to algorithms / Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Fourth edition. Cambridge, Massachusett: The MIT Press. – 2022. – 1312 p.

3. Graph Data Structure - Explained With Examples [Електронний ресурс] – Режим доступу: https://www.masaischool.com/blog/graph-data-structure-explained-with-examples/

4. Dijkstra png [Електронний ресурс] – Режим доступу: https://commons.wikimedia.org/wiki/File:Apsp_dijkstra_graph.png

5. Dijkstra's Shortest path algorithm [Електронний ресурс] – Режим доступу: https://medium.com/@adsingh03011/dijkstras-shortest-path-algorithm-e9f06b31810

6. What Is a Graph Database? [Електронний ресурс] – Режим доступу: https://aws.amazon.com/nosql/graph/?nc1=h_ls

7. DB-Engines Ranking of Graph DBMS [Електронний ресурс] – Режим доступу: https://db-engines.com/en/ranking/graph+dbms

8. Which Category Is Better: Benchmarking Relational and Graph Database Management Systems / Yijian Cheng, Pengjie Ding, Tongtong Wang, Wei Lu, Xiaoyong Du. Data Science and Engineering, 2019. – Vol. 4. – P. 309-322. https://doi.org/10.1007/s41019-019-00110-3

9. Малахов Є.В. Організація баз даних: конспект лекцій / Одеса: Одес. нац. ун-т ім. І.І. Мечникова, 2020. – 206 с.

10. Joe Celko's SQL for Smarties (Third edition). Advanced SQL Programming The Morgan Kaufmann Series in Data Management Systems, 2005.

– P. 761-775. https://doi.org/10.1016/B978-012369379-2/50034-0

11. Miler M. The Shortest Path Algorithm Performance Comparison in Graph and Relational Database on a Transportation Network / M. Miler et al. // Information and Communication Technology Preliminary Communication. Promet - Traffic&Transportation, 2014. – Vol. 26, no. 1. – P. 75-82.

12. Tharun Sikhinam. How does the performance of a graph database such as Neo4j compare to the performance of a relational database such as Postgres? [Електронний ресурс] – Режим доступу: https://courses.cs.washington.edu/courses/csed516/20au/projects/p06.pdf

13. Apache AGE Graph Database for PostgreSQL [Електронний ресурс] – Режим доступу: https://age.apache.org/

14. Postgres: The Graph Database You Didn't Know You Had [Електронний ресурс] – Режим доступу: https://www.dylanpaulus.com/posts/postgres-is-a-graph-database/

15. Малахов Є.В. Проектування інформаційних систем: методичні вказівки до курсового проектування з навчального курсу // Є.В. Малахов, О.І. Розновець / Одеса: Одес. нац. ун-т ім. І.І. Мечникова, 2020. – 53 с.

16. Introduction to Laravel and MVC Framework [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/introduction-to-laravel-and-mvc-framework/

17. OpenStreetMap [Електронний ресурс] – Режим доступу: https://www.openstreetmap.org/

18. Leaflet [Електронний ресурс] – Режим доступу: https://leafletjs.com/

19. Download OpenStreetMap data for this region: Monaco [Електронний ресурс] – Режим доступу: https://download.geofabrik.de/europe/monaco.html

20. Install Rust [Електронний ресурс] – Режим доступу: https://www.rust-lang.org/tools/install

21. osm4routing2 [Електронний ресурс] – Режим доступу: https://github.com/rust-transit/osm4routing2

22. Import Data from CSV into Postgres [Електронний ресурс] – Режим

доступу: https://hasura.io/docs/latest/schema/postgres/postgres-guides/import-data-from-csv/

23. WITH Queries (Common Table Expressions) [Електронний ресурс] – Режим доступу: https://www.postgresql.org/docs/current/queries-with.html

24. Postgres Temporary Tables: A Guide to Data Manipulation [Електронний ресурс] – Режим доступу: https://dev.to/0xog_pg/postgres-temporary-tables-a-guide-to-data-manipulation-2c7d

25. Search Order [Електронний ресурс] – Режим доступу: https://www.postgresql.org/docs/current/queries-with.html#QUERIES-WITH-SEARCH

26. Cycle Detection [Електронний ресурс] – Режим доступу: https://www.postgresql.org/docs/current/queries-with.html#QUERIES-WITH-CYCLE

27. PostgreSQL, unnest, and array_agg respecting order [Електронний ресурс] – Режим доступу: http://amyszczepanski.com/2019/03/25/unnest-and-array-agg-respecting-order.html

28. Table Functions [Електронний ресурс] – Режим доступу: https://postgresql.org/docs/current/queries-table-expressions.html#QUERIES-TABLEFUNCTIONS

29. Leaflet PolylineDecorator [Електронний ресурс] – Режим доступу: https://github.com/bbecquet/Leaflet.PolylineDecorator

30. How to sleep in Javascript [Електронний ресурс] – Режим доступу: https://coreui.io/blog/how-to-sleep-in-javascript/

31. Чернова О.Ю. Порівняння графових і реляційних баз даних / О.Ю. Чернова, О.С. Антоненко // Інформатика, інформаційні системи та технології: тези доповідей двадцять першої всеукраїнської конференції студентів і молодих науковців. Одеса, 26 квітня 2024 р. – Одеса, 2024. – С. 31-32.

ДОДАТОК А

Запити на створення алгоритмів

```
CREATE OR REPLACE FUNCTION DFS(start_node INTEGER)
RETURNS TABLE ("id" INTEGER, "to_node" TEXT, "name" CHARACTER
VARYING)
AS $$
BEGIN
    CREATE TEMPORARY TABLE tmp_dfs(
        id_tmp serial not null primary key,
        to_node integer not null
    );
    WITH RECURSIVE search_dfs(from_node, to_node, path,
have_cycle) AS (
    SELECT edge.from_node, edge.to_node,
    ARRAY[edge.to_node]::integer[] path, false AS have_cycle
    FROM edge_list edge
    WHERE edge.from_node = start_node
  UNION
    SELECT edge.from_node, edge.to_node,
      d.path || ARRAY[edge.to_node]::integer[],
      edge.to_node = ANY(d.path)
    FROM edge_list edge
    INNER JOIN search_dfs d ON d.to_node = edge.from_node AND
NOT have_cycle
) INSERT INTO tmp_dfs (to_node)
SELECT s.to_node
FROM search_dfs s
ORDER BY path;
RETURN QUERY
SELECT MIN(tmp.id_tmp) id_tmp, CONCAT(v.lat, ', ', v.lon)
"coords", v.name FROM tmp_dfs tmp
INNER JOIN vertex v ON tmp.to_node = v.id_vertex
GROUP BY "coords", v.name
ORDER BY id_tmp;
DROP TABLE tmp_dfs;
END;
$$
LANGUAGE plpgsql;
```

Лістинг А.1 – Пошук в глибину (DFS)

```
CREATE OR REPLACE FUNCTION BFS(start_node INTEGER)
RETURNS TABLE ("id" INTEGER, "to_node" TEXT, "name" CHARACTER
VARYING)
AS $$
BEGIN
    CREATE TEMPORARY TABLE tmp_bfs(
```

Лістинг А.2 – Пошук у ширину (BFS)

```
        id_tmp serial not null primary key,
        to_node integer not null
    );
    WITH RECURSIVE search_bfs(from_node, to_node, path, depth,
have_cycle) AS (
    SELECT edge.from_node, edge.to_node,
    ARRAY[edge.to_node]::integer[] path, 0 AS depth, false AS
have_cycle
    FROM edge_list edge
    WHERE edge.from_node = start_node
  UNION
    SELECT edge.from_node, edge.to_node,
      b.path || ARRAY[edge.to_node]::integer[], b.depth+1,
      edge.to_node = ANY(b.path)
    FROM edge_list edge
    INNER JOIN search_bfs b ON b.to_node = edge.from_node AND
NOT have_cycle
) INSERT INTO tmp_bfs (to_node)
SELECT s.to_node
FROM search_bfs s
ORDER BY depth;
RETURN QUERY
SELECT MIN(tmp.id_tmp) id_tmp, CONCAT(v.lat, ', ', v.lon)
"coords", v.name FROM tmp_bfs tmp
INNER JOIN vertex v ON tmp.to_node = v.id_vertex
GROUP BY "coords", v.name
ORDER BY id_tmp;
DROP TABLE tmp_bfs;
END;
$$
LANGUAGE plpgsql;
```

Лістинг А.2, аркуш 2

```
CREATE OR REPLACE FUNCTION DIJKSTRAS(start_node INTEGER,
                                     end_node INTEGER)
RETURNS TABLE ("from_node" TEXT, "from_name" CHARACTER VARYING,
               "to_node" TEXT, "to_name" CHARACTER VARYING,
               "weight" INTEGER, "path" TEXT[], "path_name"
CHARACTER VARYING[])
AS $$
BEGIN
    CREATE TEMPORARY TABLE tmp_dijkstras(
        id_tmp serial not null primary key,
        from_node integer not null,
        to_node integer not null,
        weight integer not null,
        path integer[] not null
    );
    WITH RECURSIVE dijkstras(from_node, to_node, depth, weight,
path, have_cycle) AS (
```

Лістинг А.3 – Алгоритм Дейкстри (Dijkstra's algorithm)

```
    SELECT edge.from_node, edge.to_node, 0 AS depth, edge.weight,
        ARRAY[edge.to_node]::integer[] path, false AS have_cycle

    FROM edge_list edge
    WHERE edge.from_node = start_node
  UNION
    SELECT d.from_node, edge.to_node, d.depth + 1,
     edge.weight + d.weight, d.path ||
     ARRAY[edge.to_node]::integer[], edge.to_node =
     ANY(d.path)FROM edge_list edge INNER JOIN dijkstras d ON
     d.to_node = edge.from_node AND NOT have_cycle
) INSERT INTO tmp_dijkstras (from_node, to_node, weight, path)
SELECT di.from_node, di.to_node, di.weight, di.path
FROM dijkstras di WHERE di.to_node = end_node
ORDER BY di.weight
LIMIT 1;
RETURN QUERY
SELECT tt.*, ARRAY_AGG(CONCAT(to_p.lat, ', ', to_p.lon) ORDER BY
ORDINALITY) AS "path",
ARRAY_AGG(to_p.name ORDER BY ORDINALITY) AS "path_name"
FROM ((SELECT CONCAT(f.lat, ', ', f.lon) "from_node", f.name
from_name,
              CONCAT(t.lat, ', ', t.lon) "to_node", t.name
to_name, tmp.weight
        FROM tmp_dijkstras tmp
        INNER JOIN vertex f ON tmp.from_node = f.id_vertex
        INNER JOIN vertex t ON tmp.to_node = t.id_vertex)) tt,
        tmp_dijkstras, UNNEST(tmp_dijkstras.path) WITH
ORDINALITY AS "paths"
INNER JOIN vertex to_p ON "paths" = to_p.id_vertex
GROUP BY tt.from_node, tt.from_name, tt.to_node, tt.to_name,
tt.weight;
DROP TABLE tmp_dijkstras;
END;
$$
LANGUAGE plpgsql;
```

Лістинг А.3, аркуш 2

ДОДАТОК Б

Запити на створення додаткових функцій

```
CREATE OR REPLACE FUNCTION get_graph()
RETURNS TABLE("from_coords" TEXT, "to_coords" TEXT,
              "from_name" CHARACTER VARYING, "to_name" CHARACTER
VARYING, weight INTEGER)
AS $$
BEGIN
RETURN QUERY
SELECT CONCAT(from_v.lat, ', ', from_v.lon) "from_coords",
CONCAT(to_v.lat, ', ', to_v.lon) "to_coords",
from_v.name from_name, to_v.name to_name, e.weight
FROM edge_list e
INNER JOIN vertex from_v ON from_v.id_vertex = e.from_node
INNER JOIN vertex to_v ON to_v.id_vertex = e.to_node;
END;
$$
LANGUAGE plpgsql;
```

Лістинг Б.1 – Створення функції get_graph

```
CREATE OR REPLACE FUNCTION from_nodes()
RETURNS TABLE("id_vertex" INTEGER, "from_name" CHARACTER VARYING)
AS $$
BEGIN
RETURN QUERY
SELECT DISTINCT from_v.id_vertex id_vertex, from_v.name
from_name
FROM edge_list
INNER JOIN vertex from_v ON from_v.id_vertex =
edge_list.from_node
ORDER BY id_vertex;
END;
$$
LANGUAGE plpgsql;
```

Лістинг Б.2 – Створення функції from_nodes

```
CREATE OR REPLACE FUNCTION to_nodes()
RETURNS TABLE("id_vertex" INTEGER, "to_name" CHARACTER VARYING)
AS $$
BEGIN
RETURN QUERY
SELECT DISTINCT to_v.id_vertex id_vertex, to_v.name to_name
FROM edge_list
INNER JOIN vertex to_v ON to_v.id_vertex = edge_list.to_node
ORDER BY id_vertex;
```

Лістинг Б.3 – Створення функції to_nodes

```
END;
$$
LANGUAGE plpgsql;
```

Лістинг Б.3, аркуш 2

```
CREATE OR REPLACE FUNCTION get_big_graph()
RETURNS TABLE("from_node" BIGINT, "to_node" BIGINT,
"from_coords" TEXT, "to_coords" TEXT,
              "weight" INTEGER)
AS $$
BEGIN
RETURN QUERY
SELECT e.from_node, e.to_node, CONCAT(fr.lat, ', ', fr.lon)
from_coords,
CONCAT(t.lat, ', ', t.lon) to_node_coords, e.weight
FROM edge_list e
INNER JOIN vertex fr ON fr.id_vertex = e.from_node
INNER JOIN vertex t ON t.id_vertex = e.to_node;
END;
$$
LANGUAGE plpgsql;
```

Лістинг Б.4 – Створення функції get_big_graph