ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ І. І. МЕЧНИКОВА

(повне найменування закладу вищої освіти)

Факультет математики, фізики та інформаційних технологій

(повне найменування факультету)

Кафедра інформаційних технологій

(повна назва кафедри)

Кваліфікаційна робота

на здобуття ступеня вищої освіти «Бакалавр»

«Розробка веб-застосунку для візуалізації пошукових алгоритмів»

(тема кваліфікаційної роботи українською мовою)

«Development of a web application for visualizing search algorithms»

(тема кваліфікаційної роботи англійською мовою)

Виконав: здобувач денної форми навчання
спеціальності 122 Комп'ютерні науки
(код, назва спеціальності)
Освітня програма Комп'ютерні науки
(назва)
ГУДЕВИЧ Володимир Станіславович
(прізвище, ім'я, по-батькові здобувача)

Керівник старш. викл. Вохменцева Т.Б. _____
(науковий ступінь, вчене звання, прізвище, ініціали) (підпис)

Рецензент к.т.н., доцент кафедри інформаційних технологій НУ ОЮА, Гура В.І.
(науковий ступінь, вчене звання, прізвище, ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
Інформаційних технологій
№ ___ від ___ _____ 2025 р.

Завідувачка кафедри

_____ КАЗАКОВА Надія
(підпис) (прізвище,ім'я)

Захищено на засіданні ЕК № _____
протокол № ___ від _____ 2025 р.

Оцінка _____ / _____ / _____
(за національною шкалою/шкалою ECTS/ бали)

Голова ЕК

_____ КОПИЧЕНКО Іван
(підпис) (прізвище,ім'я)

Одеса 2025

АНОТАЦІЯ

Кваліфікаційну роботу бакалавра присвячено розробці та візуалізації алгоритмів пошуку в графових і сіткових структурах. Основною метою є створення інтерактивного навчального інструменту для демонстрації роботи алгоритмів пошуку шляхів. У роботі подано теоретичний огляд основ теорії графів, моделей подання графів і поглиблений аналіз алгоритмів пошуку, таких як BFS, DFS, A*, модифікований A*, алгоритм Дейкстри та його варіант для сіткових структур.

Для практичної реалізації створено вебзастосунок, що працює локально в браузері без потреби у серверній частині. Систему реалізовано з використанням HTML, CSS і JavaScript, а також бібліотек Leaflet.js (для інтерактивної мапи) та Turf.js (для геопросторових обчислень). У графовому режимі використано словники суміжності та координати реальних міст Франції, Польщі й України. Сітковий режим реалізує клітинне поле, обмежене географічними межами країн.

Користувацький інтерфейс є інтерактивним, повністю зосередженим в одному HTML-файлі та стилізованим за допомогою окремого CSS-файлу. Усі елементи керування працюють динамічно, без перезавантаження сторінки. Функціональна логіка побудована на основі модульної архітектури: кожен алгоритм пошуку реалізовано як окремий компонент, що взаємодіє з інтерфейсом залежно від обраного режиму. Систему протестовано на штучно створених графах і реальних географічних даних, що підтвердило її стабільність під час зміни режимів, параметрів і повторного запуску алгоритмів. Розроблений інструмент є ефективним засобом для вивчення основ алгоритмічного мислення, проведення самостійних досліджень і наочного ознайомлення з принципами роботи пошукових алгоритмів.

ABSTRACT

The bachelor's qualification work is devoted to the development and visualization of search algorithms in graph and grid structures. The main objective is to create an interactive educational tool for demonstrating the operation of pathfinding algorithms. The thesis presents a theoretical overview of graph theory fundamentals, graph representation models, and an in-depth analysis of search algorithms such as BFS, DFS, A*, a modified A*, Dijkstra’s algorithm, and its adaptation for grid-based systems.

For practical implementation, a web application was developed that runs locally in the browser without requiring a server-side component. The system is built using HTML, CSS, and JavaScript, along with the Leaflet.js library (for interactive mapping) and Turf.js (for geospatial computations). In graph mode, adjacency dictionaries and coordinates of real cities in France, Poland, and Ukraine are used. The grid mode implements a cell-based field bounded by the geographical borders of these countries.

The user interface is interactive, fully contained within a single HTML file, and styled with a separate CSS file. All control elements operate dynamically without reloading the page. The functional logic follows a modular architecture: each search algorithm is implemented as a separate component that interacts with the interface based on the selected mode. The system has been tested on both artificially generated graphs and real-world geographic data, confirming its stability during mode switches, parameter adjustments, and repeated algorithm executions. The developed tool is an effective means for studying algorithmic thinking, conducting independent experiments, and visually understanding the principles behind search algorithms.

ЗМІСТ

Стор.

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

Скорочення

API – Application Programming Interface

BFS – Breadth-First Search (пошук у ширину)

CSS – Cascading Style Sheets

DFS – Depth-First Search (пошук у глибину)

DOM – Document Object Mode

HTML – HyperText Markup Language

IDEF0 – методологія моделювання функціональних систем

JSON – JavaScript Object Notation

UI – User Interface.

UX – User Experience

Умовні позначення

A* – A-star

FR – скорочення назви країни (France)

PL – скорочення назви країни (Poland)

UA – скорочення назви країни (Ukraine)

Терміни

A* – евристичний алгоритм пошуку найкоротшого шляху.

GeoJSON – формат представлення географічних об’єктів у вигляді JSON.

IDEF0 (Function Modeling) – методологія функціонального моделювання і графічного опису процесів, призначена для формалізації і опису бізнес-процесів.

JS / JavaScript – мова програмування, що використовується для динаміки вебзастосунків.

Leaflet.js – JavaScript-бібліотека для відображення інтерактивних мап.

Turf.js – JavaScript-бібліотека для просторового аналізу в браузері.

ВСТУП

Згідно з сучасними дослідженнями у сфері комп'ютерних наук та освіти, візуалізація алгоритмів є одним із найефективніших засобів для формування алгоритмічного мислення та поглибленого розуміння принципів роботи структур даних. Зростання попиту на доступні, інтуїтивно зрозумілі освітні інструменти, а також розвиток клієнтських вебтехнологій сприяли появі нових форм подання навчального матеріалу – зокрема, через інтерактивні візуалізації у браузері. На практиці такі інструменти дозволяють не лише спостерігати за виконанням алгоритмів у реальному часі, а й активно взаємодіяти з даними, змінюючи параметри, експериментуючи з різними структурами та умовами.

Однією з ключових тем в алгоритмічному навчанні є пошук шляху в графах і сітках. Алгоритми пошуку, такі як A*, Dijkstra, BFS і DFS, застосовуються в численних галузях – від логістики й геонавігації до штучного інтелекту та обробки зображень. Проте традиційне викладання цих алгоритмів зазвичай обмежується теоретичним описом і текстовим псевдокодом, що часто ускладнює засвоєння для студентів-початківців.

У межах цього проєкту планується створити вебзастосунок, який дозволить в інтерактивному режимі візуалізувати роботу пошукових алгоритмів на графових і сіткових структурах. Система буде реалізована з використанням HTML, CSS і JavaScript та працюватиме повністю в браузері без серверної частини. Особливістю реалізації стане можливість перемикання між режимами – графом і сіткою – та вибір однієї з кількох попередньо реалізованих країн із реальними географічними координатами. Алгоритми будуть реалізовані як окремі модулі, а обробка логіки запуску – централізована в основному керуючому компоненті. Для побудови маршруту використовуватимуться дані про міста, координати, географічні межі країн та клітинна структура з можливістю додавання перешкод.

Очікується, що після запуску алгоритму користувач зможе спостерігати за анімованим процесом пошуку з кольоровим виділенням пройдених вузлів,

знайденого шляху та помітною візуалізацією перебігу виконання. Такий підхід сприятиме кращому розумінню логіки роботи алгоритмів через пряму інтеракцію.

Метою роботи є розробка інтерактивного вебзастосунку для візуалізації алгоритмів пошуку шляху з орієнтацією на освітню функцію.

Об'єктом дослідження є процес візуального представлення алгоритмів пошуку у браузерному середовищі.

Предметом дослідження є методи реалізації графових та сіткових структур, алгоритми пошуку шляху, а також інтерфейсні підходи до візуального подання навчального матеріалу.

Дана кваліфікаційна робота бакалавра складається з 63 сторінок, 24 рисунків, 2 таблиць, 20 джерел посилань та 4 додатків.

1 ТЕОРЕТИЧНІ ОСНОВИ СТРУКТУРИ ГРАФІВ І АЛГОРИТМІВ ПОШУКУ

1.1 Дослідження поняття зв’зок в контексті графових структур

Зв’язок – це невід’ємна частина буття. Абстрактні властивості, такі як «апельсин має колір помаранчевий», є прикладом зв'язку між об'єктом і його ознакою; слово «помаранчевий» при цьому пов’язане з поняттям «колір», яке, у свою чергу, належить до категорії «візуальні властивості», що входять у ширший простір «сенсорного сприйняття». Жодне поняття не може існувати в повній ізоляції від інших – ні фізично, ні концептуально. Воно завжди перебуває у зв’язку з навколишнім середовищем, взаємодіє з іншими об’єктами, пов’язане із власними ознаками, такими як назва, колір, форма, функція. Це стосується не лише абстрактних понять, а й цілком матеріальних речей. Наприклад, людина фізично пов’язана з одягом, який виготовлено з матеріалів, що були оброблені на фабриках, куди їх доставили через логістичні ланцюги з різних країн світу. Але й це ще не кінець: кожен предмет одягу має культурне, економічне та соціальне походження, що з’єднує людину з історією, традиціями, працею інших людей і глобальною системою виробництва. Таким чином, навіть простий фізичний зв’язок людини з предметом розгортається в широку мережу взаємозалежностей.

1.2 Огляд основ теорії графів

До певного часу зв’язки існували лише на інтуїтивному рівні – вони були присутні у світі, але не мали чіткої форми чи мови опису. Однак постійна потреба розвитку математики та зусилля видатного вченого Леонарда Ейлера дозволили формалізувати ці зв’язки: надати їм структуру, яку можна вивчати, аналізувати й застосовувати в різних сферах.

Як описується в науково-популярній статті журналу Scientific American[1], формалізація зв’язків уперше відбулася у 18 столітті, коли мешканці прусського міста Кенігсберг билися над головоломкою: як знайти пішохідну доріжку через місто, яка б перетинала кожен із семи старовинних мостів, що зображені на рис. 1.1, рівно один раз.

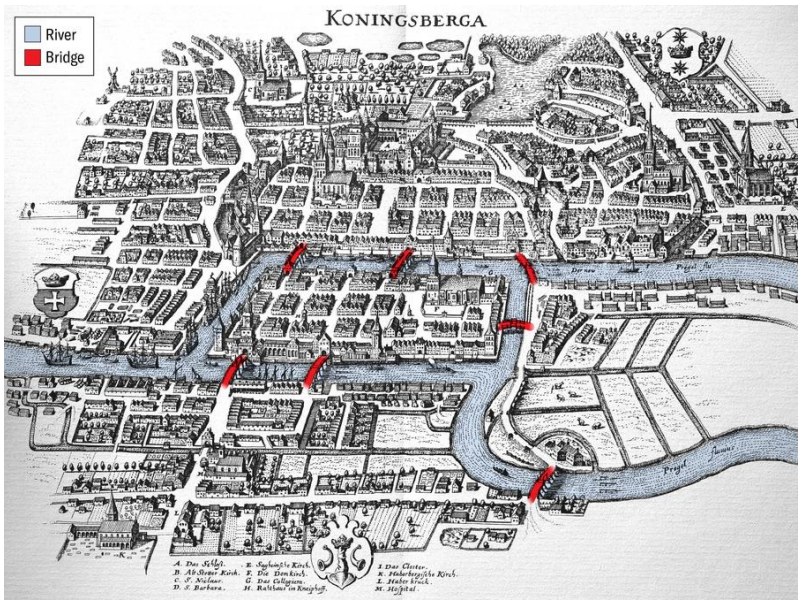

Рисунок 1.1 – Задача кенігсбергських мостів

Мости перекинуті через річку з двома великими островами. Як би вони не планували свої маршрути, вони не могли уникнути повторення мосту.

Ця проблема не давала спокою місцевим мислителям, які врешті-решт написали листа до відомого математика Леонарда Ейлера, благаючи його вгамувати їхню цікавість. Ейлер відповів зневажливо, стверджуючи, що проблема має «мало відношення до математики». У певному сенсі він мав рацію, адже відповідної математики ще не було винайдено. Незважаючи на свою початкову відмову, Ейлер врешті-решт розв'язав головоломку про сім мостів Кенігсберга, не підозрюючи, що в процесі цього він породив нову галузь математики – теорію графів.

В статті [1] підкреслено, що підхід Ейлера, як і багато інших методів створення математичних моделей реальних задач зав’язано на процесі відсіювання зайвої інформації до тих пір, поки не залишаться лише основні елементи - процес, який називається абстрагуванням. Наприклад, якщо подивитися на рис. 1.1, можна зробити висновок, що багато особливостей карти не впливають на питання, яке розглядається. Довжину мостів, розміри земельних масивів, навіть географічну орієнтацію землі та мостів можна відкинути. Важливо лише те, які ділянки землі з'єднуються з іншими і скільки разів. З цього можна створити набагато простішу діаграму, що складається лише з кіл та ліній, які позначають сушу та мости відповідно (див. рис. 1.2).

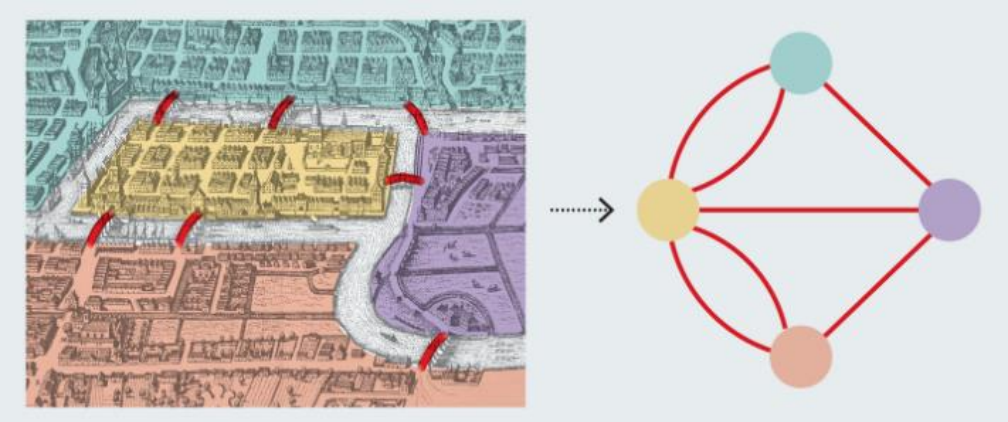

Рисунок 1.2 – Абстрагування задачі кенігсбергських мостів

Сучасна математична мова називає це «графом». Кола називаються «вершинами», а лінії «ребрами». Сьогодні теорія графів є основною галуззю математики та інформатики з широким спектром застосувань. Графи не обов'язково мають зображати землю та мости. Вони можуть представляти соціальні мережі, абстрактні ознаки апельсину, факт використання людиною одягу, білкові взаємодії, державні кордони, нейронні мережі, всесвітню павутину або будь-які інші дані, що зв’язані між собою.

1.3 Дослідження характеристик графа

Очевидно, що поява нової галузі математики – теорії графів – з часом призвела до виникнення великої кількості термінів, класифікацій та алгоритмів. У сучасній графовій теорії існують десятки різновидів графів: орієнтовані й неорієнтовані, зважені, мультиграфи, гіперграфи, динамічні графи тощо. Усе це забезпечує гнучкість моделювання реальних систем, однак для цілей кваліфікаційної роботи бакалавра достатньо зосередитись лише на базових властивостях графа, що мають значення для візуалізації алгоритмів пошуку, які будуть детальніше описані у наступних розділах. Для зручності їх можна поділити на дві групи: ті, що стосуються ребер, і ті, що належать до вершин.

1.3.1 Аналіз властивостей ребер графа

Природа задачі кенігсберзьких мостів дозволяє побудувати граф із високим рівнем абстракції. Просте з’єднання вершин ребрами відображає саму суть головоломки, яку необхідно було вирішити. Проаналізувавши причину допустимості такого рівня спрощення, можна дійти важливого висновку: усі зв’язки в цій моделі мають однакову природу.

Контрастом до цього є структура понять, пов’язаних із людиною та її одягом. Наприклад: «людина носить одяг», «одяг виготовлений із тканини», «тканина вироблена на фабриці», «виробництво регулюється трудовим правом», або ж «одяг має культурне походження» (рис. 1.3). У цьому випадку мова вже йде про семантичну мережу, у якій типи зв’язків відрізняються: «має», «носить», «виготовлений із», «регулюється» тощо [2]. Така мережа не може бути представлена графом із однотипними ребрами, оскільки кожен зв’язок має свою унікальну роль і значення в контексті.

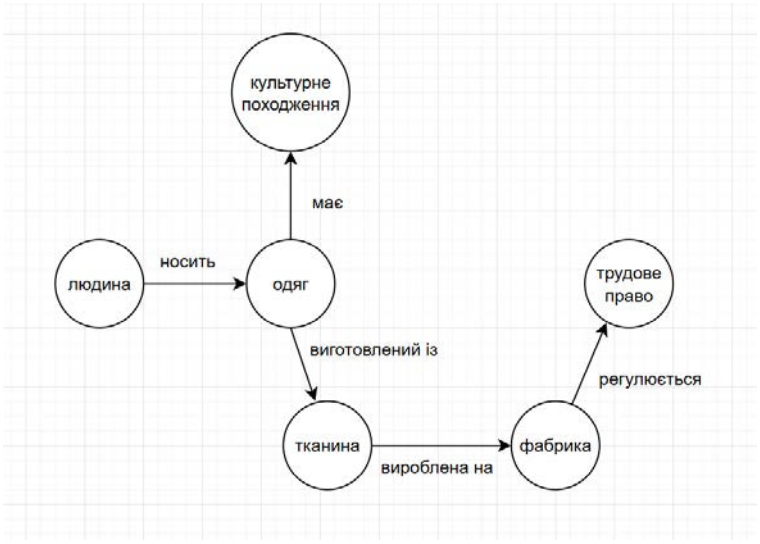

Рисунок 1.3 – Приклад семантичної мережі

Слід зазначити, що семантична мережа не є єдиним прикладом графа з різнотипними ребрами. Подібні структури застосовуються у багатьох галузях: у транспортних мережах (де ребра можуть означати різні типи транспорту), у соціальних мережах (зі зв’язками типу «друг», «підписник», «коментатор»), у біологічних або генетичних взаємодіях, а також у системах семантичного вебу (RDF-графи, онтології). У таких графах кожне ребро несе власне семантичне навантаження й часто потребує розширеної структури для зберігання типу зв’язку.

Оскільки метою кваліфікаційної роботи бакалавра є візуалізація пошукових алгоритмів, що працюють у рамках класичних моделей, у подальшому будуть використовуватися лише графи з однотипними ребрами – тобто такі, де всі зв’язки інтерпретуються однаково з точки зору логіки алгоритму. У задачі Кенігсберга кожен зв’язок можна описати однаковим словосполученням – «з’єднані мостом». Тобто кожна вершина – це окрема ділянка суші, а кожне ребро – просто факт наявності моста між ними, без додаткової семантики або типізації (див. рис. 1.4).

Рисунок 1.4 – Приклад графу з однотипними ребрами

З рис. 1.3 можна побачити, що для відображення відношень семантичної мережі використовується додавання стрілок на кінцях ребер. Такий умовний знак використовується для зображення напрямленості зв’язків між вершинами і означає, що зв’язок має чітко визначений напрям – від джерела до цілі.

Іншими словами, стрілка визначає напрям логічного або функціонального зв’язку – вона вказує, від якого елемента походить відношення і на який елемент воно спрямоване. Це дає змогу коректно інтерпретувати структуру семантичної залежності, де кожна вершина відіграє конкретну роль: одна – як джерело дії або властивості, інша – як її носій або об’єкт. У випадках, подібних до «людина носить одяг» або «виробництво регулюється правом», така направленість є критично важливою, оскільки зміна напрямку зв’язку спотворює первинний зміст або призводить до семантичної помилки – як у випадку, зображеному на рис. 1.5: конструкція «одяг носить людину» суперечить реальній логіці взаємодії між об’єктами.

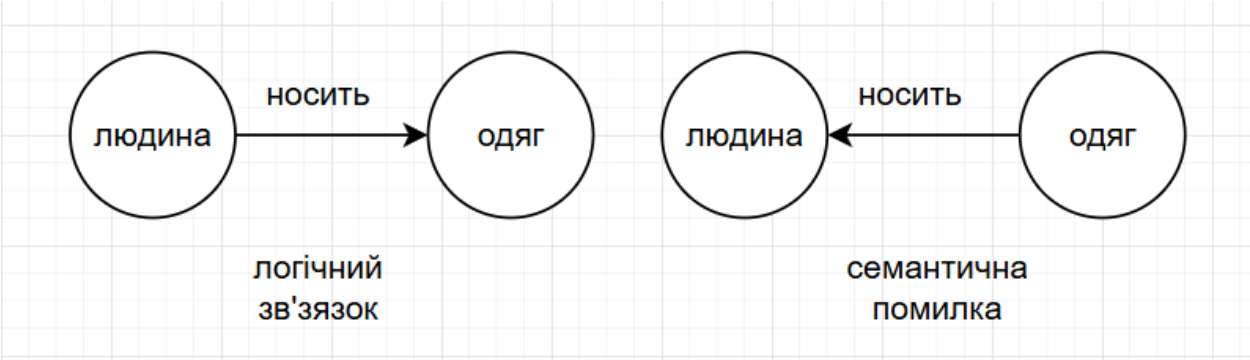

Рисунок 1.5 – Приклад семантичної помилки в направленому графі

У статті електронного ресурсу Вікіпедії [3] для графів, у яких усі ребра мають чітко визначений напрям, використовується термін «направлений граф» (directed graph). Відповідно, графи, у яких жодне з ребер не має напрямку, називаються «ненаправленими» (undirected graphs).

Варто зазначити, що існують також змішані графи (mixed graphs), у яких можуть одночасно використовуватися як орієнтовані, так і неорієнтовані ребра [4]. Проте, наявність різних типів зв’язків у межах однієї структури означає, що ребра такого графа є неоднотипними. Саме тому змішані графи точно не відповідають вимогам до моделей, що використовуються в даній роботі.

З’ясувавши, що змішані графи не відповідають вимогам до моделі, необхідної для реалізації алгоритмів у межах даної кваліфікаційної роботи бакалавра, доцільно звернутися до аналізу двох базових типів графів – орієнтованих і неорієнтованих, аби обрати найбільш доцільний для поставленої задачі.

Хоча в реальному світі наявність шляху між двома об’єктами зазвичай означає можливість руху в обох напрямках (що наближає модель до неорієнтованого графа), у межах цього проєкту граф буде свідомо реалізовано як орієнтований. Це забезпечує додаткову гнучкість під час роботи з алгоритмами, даючи змогу, за потреби, моделювати односторонні зв’язки або свідомо обмежувати напрям пошуку – зокрема, шляхом використання вершинних потенціалів (докладніше про це йтиметься у розділі про алгоритм

A*). Таким чином, орієнтованість ребер у даному випадку виконує не лише структурну функцію, а й слугує механізмом контролю над поведінкою алгоритму.

Останньою, але не менш важливою властивістю ребер графа, яку доцільно розглянути, є можливість призначення ваги. Згідно зі статтею електронного ресурсу ScienceDirect [5], граф, у якому хоча б одному ребру присвоєно числове значення, вважається зваженим графом (weighted graph). Ребра, для яких вага не задана явно, за замовчуванням вважаються такими, що мають вагу 1 (рис 1.6) На відміну від ситуації з поєднанням орієнтованих і неорієнтованих ребер, де утворюється змішаний граф, у випадку ваг сам факт наявності хоча б одного числового значення визначає всю структуру як зважену.

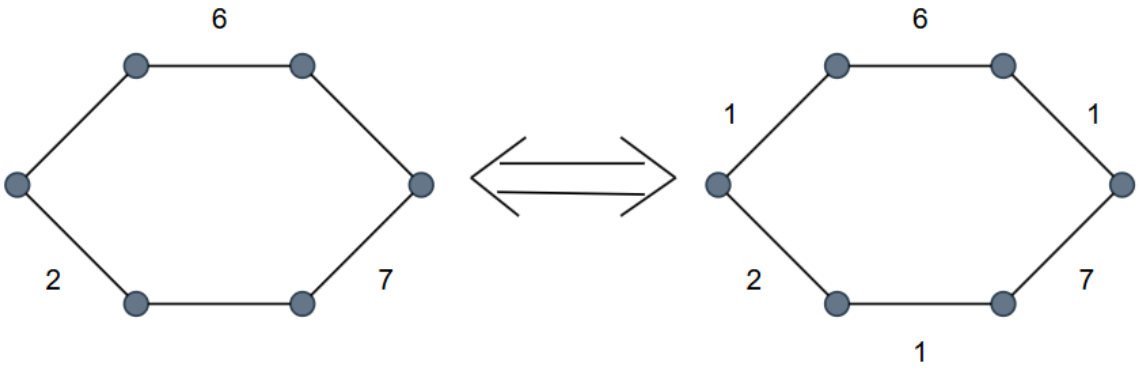

Рисунок 1.6 – Еквівалентність ребер графа без явно заданих вагів одиниці

Не всі типи зв’язків між вершинами можуть бути охарактеризовані числовою вагою. Вагу доцільно застосовувати у тих випадках, коли зв’язок має кількісну або оцінювану характеристику: це може бути відстань, час проходження, вартість, частота, інтенсивність взаємодії, ймовірність, пропускна здатність, обсяг переданих даних або рівень впливу. Такий підхід є природним для транспортних, фінансових, соціальних або технічних мереж. Натомість існують зв’язки, які не мають кількісного змісту і не можуть бути адекватно виражені через вагу. Це, зокрема, категоріальні та логічні

відношення: «є частиною», «належить до категорії», «є прикладом», «має тип», «є причиною». У таких випадках вага була б штучною або неінформативною.

Отже, властивості ребер, які будуть необхідними для реалізації алгоритмів у межах цієї роботи, можна сформулювати так:

– однаковий тип ребер (забезпечує уніфіковану інтерпретацію зв'язків, що критично важливо для універсальності реалізації);

– орієнтованість (дозволяє контролювати напрям пошуку та моделювати односторонні зв'язки);

– вага (задає «вартість» переходу між вершинами й необхідна для роботи алгоритмів оптимізації).

1.3.1 Аналіз властивостей вершин графа

У класичній теорії графів вершини можуть репрезентувати будь-які сутності: міста, села, озера, станції, об'єкти інфраструктури тощо. Це не суперечить застосуванню пошукових алгоритмів, за умови що зв'язки між вершинами мають однакову інтерпретацію – наприклад, «можна дістатися пішки» або «існує маршрут». Таким чином, граф, у якому вершинами є Київ, Біла Церква, село Зелений Гай та озеро Світязь, але всі вони з'єднані зв'язком «можна дістатися», цілком може вважатися класичним і бути основою для роботи алгоритмів пошуку. Потенційна проблема виникає лише у випадках, коли логіка самого алгоритму або умови задачі вимагають обмежень щодо типів вершин – наприклад, якщо дозволено пересування лише через міста, але не через села або водойми. У такій ситуації потрібна додаткова семантична інформація про вершини, а модель графа перестає бути повністю однорідною з точки зору алгоритму.

У межах даної роботи всі вершини графа вважатимуться однотипними, тобто такими, що виконують рівнозначну роль у структурі пошуку. Це спрощення дозволяє зосередитися на реалізації та аналізі самих алгоритмів без необхідності врахування додаткових семантичних ознак вершин. Однак у

перспективі можлива модифікація моделі з додаванням різних типів вершин (наприклад, міста, села, перешкоди, цільові об’єкти), що відкриває шлях до більш складних сценаріїв. Такий підхід може бути особливо корисним для моделювання реальних систем з додатковими правилами чи логікою взаємодії.

Хоча питання однотипності вершин заслуговує на окрему увагу, наступна властивість має ще більшу значущість у контексті побудови пошукових алгоритмів. Йдеться про вершинний потенціал – величину, яка може бути використана для штучного обмеження напряму пошуку в орієнтованому графі. Такий підхід дозволяє впливати на логіку переміщення між вершинами, зокрема, у випадках, коли потрібно задавати пріоритети напрямів або забороняти певні зворотні переходи. Теорія та приклад практичного використання вершинного потенціалу буде детально розглянуто в розділі про евристичну функцію в алгоритмі A*.

1.4 Опис видів представлення графових структур

Найбільш поширена форма представлення графів – це саме та, яку вперше використав Леонард Ейлер у рішенні задачі кенігсберзьких мостів: граф, зображений у вигляді точок (вершин), з’єднаних лініями (ребрами). Проте сам факт зв’язку між об’єктами не обов’язково має бути візуалізований саме так. Те, що передає графічне зображення – а саме наявність зв’язків між об’єктами – може бути також представлено за допомогою альтернативних структур даних, таких як список суміжності, графова база даних, двовимірна сітка тощо. Граф, який створено Ейлером, з однотипними, неозначеними за вагою ребрами, можна представити за допомогою звичайної матриці суміжності (табл 1.1).

Таблиця 1.1 – Матриця суміжності для графа на рис. 1.2

	З	Ж	Ф	П
З	0	2	1	0
Ж	2	0	1	2
Ф	1	1	0	1
П	0	2	1	0

Позначення вершин: З – зелена, Ж – жовта, Ф – фіолетова, П – помаранчева. Значення «2» у комірках матриці означає наявність двох паралельних ребер між відповідними вершинами, «1» – одного ребра, а «0» – відсутність зв’язку.

Для виконання кваліфікаційної роботи бакалавра з переліку можливих видів представлень графа використовуватиметься список суміжності та сіткову його репрезентацію.

Згідно статті на освітньому порталі Khan Academy [6], список суміжності – це структура даних, де для кожної вершини зберігається список усіх вершин, з якими вона безпосередньо з’єднана ребрами. Зазвичай це реалізується у вигляді масиву, словника або списку, де кожен елемент відповідає вершині графа, а його значення – це масив суміжних вершин. Приклад наведено на рис 1.7.

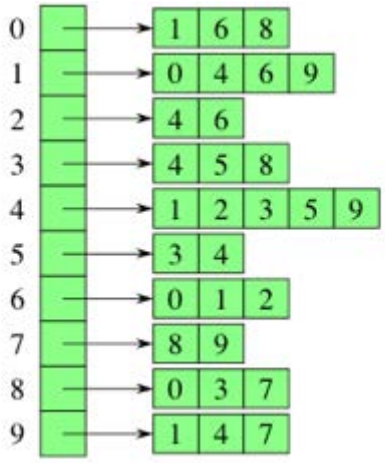

Рисунок 1.7 – Список суміжності

Щодо сіткової репрезентації графа, це добре пояснено на освітньому порталі Red Blob Games [7]: сітка розглядається як граф, у якому вершини відповідають клітинкам, а зв’язки визначаються за принципом сусідства у заданих напрямках (зазвичай – ліворуч, праворуч, вгору та вниз). Візуальне представлення такої структури представлено на рис. 1.8

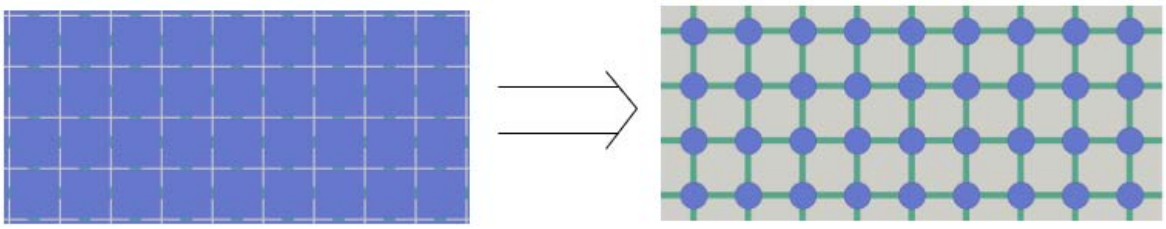

Рисунок 1.8 – Перетворення сітки в графову структуру

1.5 Дослідження роботи алгоритмів пошуку шляху в графах

Перше графове мислення з’явилося ще у XVIII столітті завдяки роботі Леонарда Ейлера над задачею кенігсберзьких мостів, графи тривалий час залишалися суто математичним поняттям – абстракцією для моделювання зв’язків. У такому вигляді вони не потребували формального опису процесу обходу чи пошуку – це виконувалося інтуїтивно або вручну. Однак усе змінилося з появою обчислювальних машин.

Комп’ютери, на відміну від людини, не здатні до інтуїтивного міркування – вони потребують точно визначених інструкцій. Будь-яку дію необхідно описати через алгоритм, тобто послідовність формалізованих кроків. І коли виникло завдання переміщення в структурі типу графа – наприклад, прокладання маршруту в лабіринті, мережі або на платі – з’явилась об’єктивна потреба створити універсальні алгоритми пошуку шляху.

Саме в цьому контексті класичне поняття графа – з вершинами і ребрами – набуло комп’ютерної форми. Графи почали реалізовуватися у вигляді

матриць, списків суміжності, сіткових структур, як описано у попередньому розділі . Тобто це все ще були ті самі графи, що й за часів Ейлера, але вже перекладені на мову машинних структур даних, готові до обробки алгоритмами.

Так, у середині XX століття були створені перші фундаментальні пошукові алгоритми – зокрема, обхід у ширину (BFS), обхід у глибину (DFS), алгоритм Дейкстри. Їх основна мета – автоматизувати процес пошуку шляху в графі, що для комп'ютера став звичайною формою подання реальних зв'язків: від маршрутів і схем до логічних або фізичних структур. Окрім згаданих класичних алгоритмів, у межах цієї кваліфікаційної роботи також розглянуто алгоритм A* – евристичний метод, розроблений дещо пізніше, у 1968 році, дослідниками Пітером Хартом, Нілом Нілссоном та Бертрамом Рафаелем. Він був створений для вирішення задач пошуку шляху в просторі, де необхідно не лише знайти маршрут, а й зробити це максимально ефективно з точки зору обчислювальних ресурсів.

1.5.1 Аналіз роботи алгоритму пошуку в ширину (BFS)

BFS і DFS – це алгоритми з відносно простою логікою та мінімальними вимогами до структури графа, що дозволяє легко їх реалізувати й застосовувати в різних контекстах. Як зазначено у конспекті лекцій з курсу «Методи і системи штучного інтелекту» [8], стратегія пошуку в ширину (Breadth-first search) вибирає для розширення той вузол, який має найменшу глибину. Під глибиною вузла розуміється кількість ребер, які потрібно пройти від стартової вершини до поточної. Іншими словами, перш ніж здійснити перевірку вузлів на глибині d + 1, алгоритм спочатку опрацьовує всі вузли, що знаходяться на глибині d. Варто уточнити, що BFS гарантує повне опрацювання всіх вузлів на рівні d перед переходом до рівня d + 1, однак не визначає точний порядок обходу всередині рівня – він залежить від методу запису черги при реалізації алгоритму, але це майже не впливає на загальне

виконанная алгоритму. На рис. 1.9 зображено модифіковану версію сіткового графа з рис. 1.8 із нанесеним значенням глибини d для кожної вершини та послідовність обходу алгоритму пошуку в ширину .

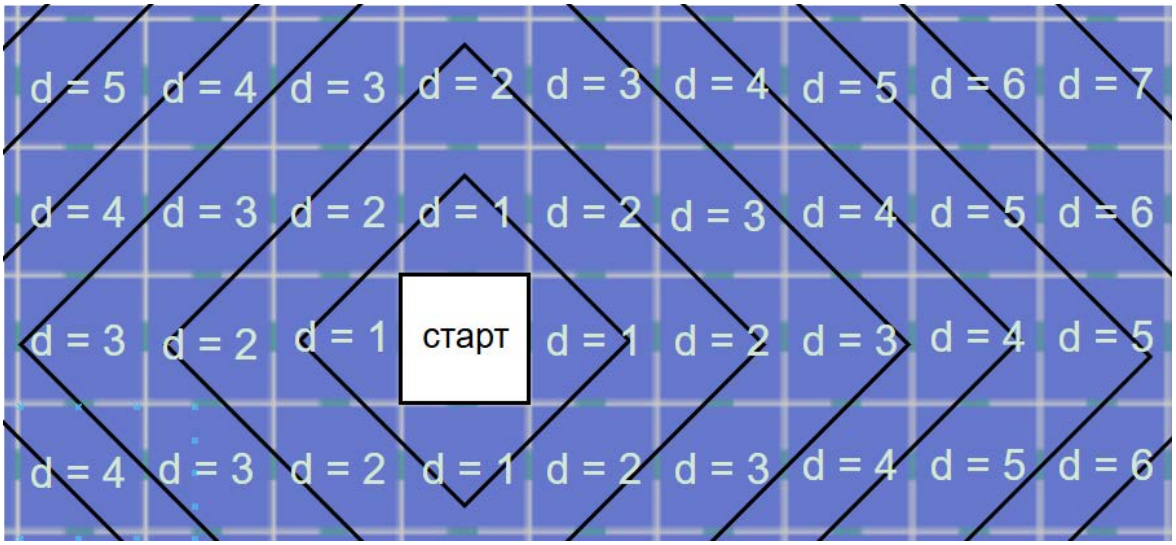

Рисунок 1.9 – Сітковий граф з додани вагами для кожної вершини та послідовністю обходу алгоритму пошуку в ширину

1.5.2 Аналіз роботи алгоритму пошуку в глибину (DFS)

У пошуку в глибину (DFS) існує декілька підходів до реалізації, які залежать від типу структури даних, що використовується (рекурсія або явний стек), а також від порядку розгляду сусідів. Незважаючи на ці варіації, загальний принцип алгоритму залишається незмінним: він намагається пройти якомога глибше по одному з напрямків, перш ніж повернутися назад і спробувати інші шляхи. Така стратегія веде до нерівномірного, «звивистого» обходу простору, який часто нагадує спонтанне блукання. Порівняно з іншими алгоритмами, пошук у глибину швидше досягає віддалених ділянок, проте не гарантує знаходження найкоротшого шляху. Саме завдяки цим властивостям DFS добре ілюструє ідею локального заглиблення та важливість порядку обходу в алгоритмах пошуку. Прикладом такої реалізації може слугувати

алгоритм пошуку в глибину, використаний на освітньому сайті Pathfinding Visualizer [9] (рис. 1.10), де для кожної клітинки сітки перевіряються сусіди у фіксованому порядку, а сам обхід здійснюється за допомогою стека. Це забезпечує наочну демонстрацію принципу заглиблення та дозволяє спостерігати, як порядок обходу впливає на форму маршруту.

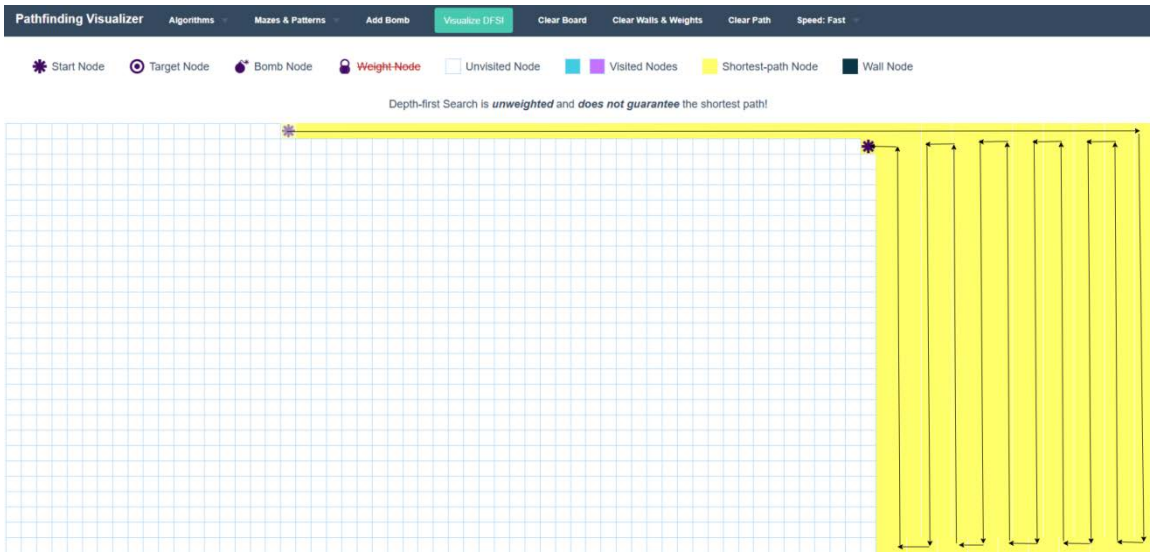

Рисунок 1.10 – Приклад реалізації алгоритму пошуку в глибину

1.5.3 Аналіз роботи алгоритму Дейкстри

На відміну від структурних алгоритмів обходу, таких як BFS і DFS, які базуються на простих правилах переходу між вершинами, алгоритм Дейкстри є більш універсальним та обчислювально орієнтованим. Він враховує вагу ребер та забезпечує знаходження найкоротших шляхів у зважених графах.

Як зазначено на освітньому вебресурсі W3Schools [10], алгоритм Дейкстри може бути узагальнено у вигляді шести основних кроків:

– встановити початкові відстані для всіх вершин: 0 для стартової вершини та безкінечність для всіх інших;

– вибрати неопрацьовану вершину з найменшою відстанню від початку та зробити її поточною (на першому кроці це буде стартова вершина);

– для кожної неопрацьованої сусідньої вершини поточної обчислити відстань від джерела і оновити її, якщо нова відстань менша за поточну;

– позначити поточну вершину як відвідану (відвідані вершини більше не розглядаються);

– повернутися до кроку 2 та повторювати, доки всі вершини не будуть відвідані;

– у результаті отримується найкоротший шлях від стартової вершини до кожної іншої вершини графа.

1.5.4 Аналіз роботи евристичного алгоритму A*

Алгоритм A* (A-star) є пошуковим методом, що поєднує елементи як жадібного пошуку за евристикою, так і методу Дейкстри, який враховує реальну вартість шляху. Його основна ідея полягає у використанні комбінованої функції оцінки f(n) = g(n) + h(n), де g(n) – це вартість шляху від початкової точки до поточної, а h(n) – евристична оцінка відстані до цілі. Завдяки цьому алгоритм демонструє більш цілеспрямовану поведінку порівняно з методом Дейкстри, уникаючи зайвих обходів.

У задачах, де простір пошуку можна представити як сітку або географічний граф, A* дозволяє ефективно зменшити кількість перевірених вершин, особливо коли використовується добре підібрана евристика – наприклад, мангеттенська для сіток із рухом по чотирьох напрямках, або евклідова для простору з реальними координатами. Це робить алгоритм A* придатним для задач навігації, побудови маршрутів на картах або аналізу структурованих просторів [11]. Візуально такі властивості можна спостерігати через характерну траєкторію пошуку: вона тяжіє до прямої між початком і ціллю, хоча й враховує обхід перешкод або складні зв’язки у графі (рис. 1.11).

Рисунок 1.11 – Порівняння роботи алгоритма Дейкстри та алгоритма A*

2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ І ТЕХНОЛОГІЙ ВІЗУАЛІЗАЦІЇ

2.1 Візуальне подання алгоритмів як дидактичного інструменту

Згідно з освітніми дослідженнями у сфері комп'ютерних наук, візуалізація алгоритмів є ефективним інструментом навчання, що сприяє глибшому розумінню логіки обчислень і структур даних.

Зокрема, у роботі «Designing Educationally Effective Algorithm Visualizations» [12], яка базується на експериментах із використанням HalVis – інтерактивного візуалізаційного середовища, спеціально створеного для навчання алгоритмів – зазначено, що можливість студентів безпосередньо взаємодіяти з навчальним матеріалом суттєво покращує засвоєння складних концепцій.

Додатково проаналізовано чинники, які забезпечують вищу ефективність навчання за умови застосування візуалізаційних засобів порівняно з традиційними методами викладання. До основних із них належать:

- активна взаємодія з алгоритмом;
- візуалізація абстрактних понять через знайомі образи або метафори;
- мультимодальний підхід (поєднання тексту, графіки, анімації);
- покрокове виконання з можливістю контролю темпу;
- варіативність вхідних даних для експериментів.

Деталізований опис кожного з факторів подано в додатку А, табл. А.1.

Перераховані вище пункти можуть слугувати основою для порівняльного аналізу програмних рішень, подібних за функціональністю до розробленого в межах цієї роботи застосунку.

2.2 Огляд існуючих рішень

Одним із найвідоміших освітніх ресурсів для вивчення алгоритмів є Red Blob Games – інтерактивна стаття-демонстрація, створена Амітом Пателем. Цей формат поєднує текст, анімацію та елементи взаємодії, створюючи інтуїтивне середовище для самостійного дослідження роботи алгоритмів. Головне меню вебзастосунку зображено на рис. 2.1, де перелічено основні доступні теми статей.

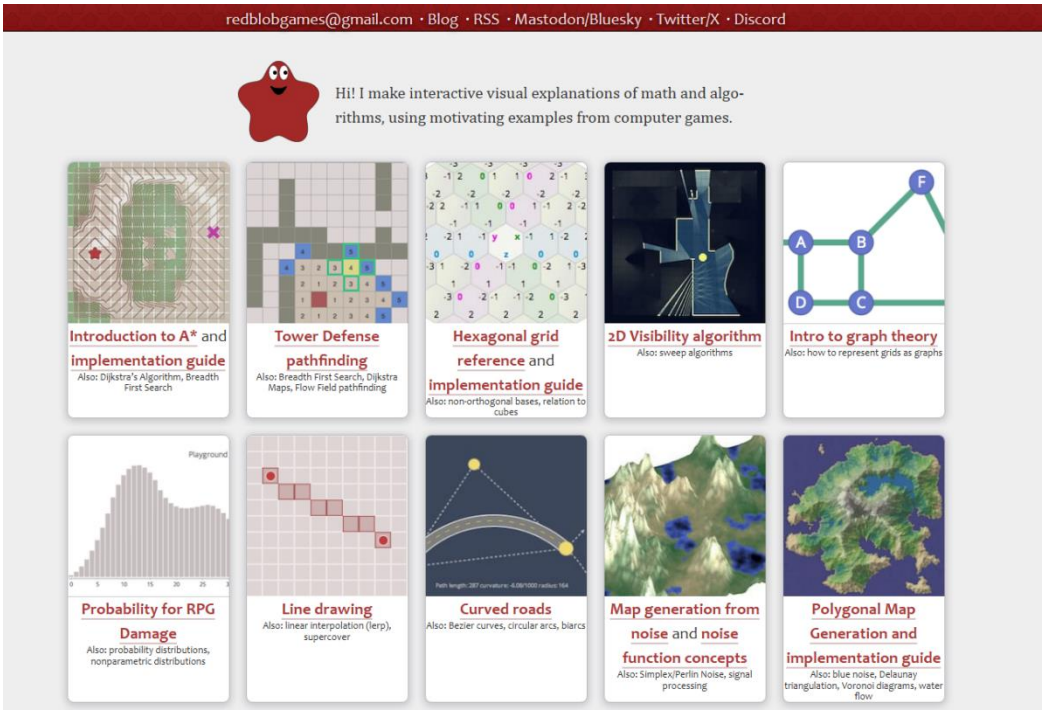

Рисунок 2.1 – Головне меню вебзастосунку Red Blob Games

Виходячи з інформації в додатку А, табл А.2, де наведено аналіз цього ресурсу відповідно до п’яти освітніх факторів ефективної візуалізації алгоритмів, Red Blob Games є якісно реалізованим навчальним ресурсом, що поєднує інтуїтивну візуалізацію з текстовими поясненнями у форматі інтерактивної освітньої статті. Попри деякі обмеження у взаємодії та варіативності графової структури, платформа ефективно виконує свою освітню функцію. Завдяки вдало підібраному формату подання та фокусу на

ключових елементах роботи алгоритму, ресурс дозволяє студенту активно включитися в процес навчання, формуючи інтуїтивне розуміння основ пошуку на графі.

Наступним прикладом навчального ресурсу, що візуалізує алгоритми, є платформа VisuAlgo [13], яка зосереджується переважно на структурах даних та класичних алгоритмах комп'ютерних наук. Головне меню вебзастосунку зображено на рис. 2.2, де, подібно до Red Blob Games, представлено перелік основних доступних тем у вигляді інтерактивних статей або демонстрацій.

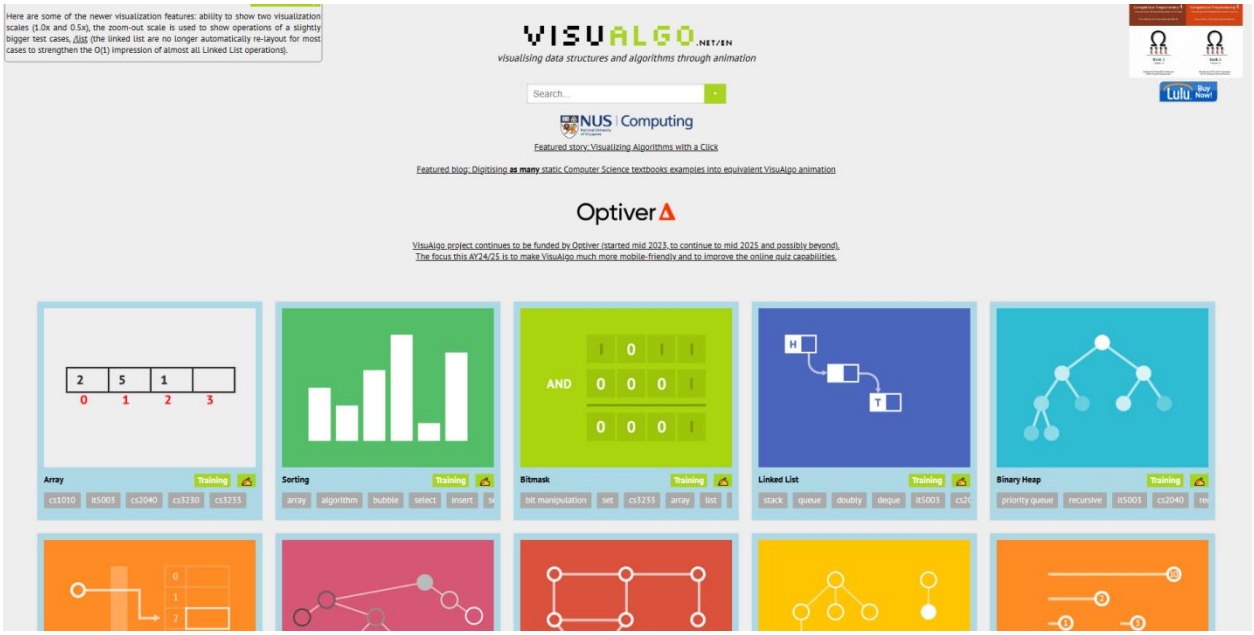

Рисунок 2.2 – Головне меню вебзастосунку VisuAlgo

Виходячи з інформації в додатку А, табл. А.3, де наведено аналіз ресурсу відповідно до п'яти освітніх факторів ефективної візуалізації алгоритмів, VisuAlgo є потужним навчальним інструментом, що поєднує інтерактивність, мультимодальність і покрокову візуалізацію. Завдяки синхронізації псевдокоду з графічною візуалізацією, а також наявності текстових пояснень, платформа створює повноцінне освітнє середовище для глибокого опанування класичних алгоритмів і структур даних. Основним обмеженням VisuAlgo є недостатня варіативність у створенні власних графів та налаштуванні задач, проте це

компенсується добре структурованим викладом матеріалу та широким охопленням алгоритмів, що робить ресурс цінним у формальному навчальному контексті.

Ще одним прикладом візуалізаційного інструменту, який активно використовується у навчальному середовищі, є Pathfinding Visualizer – проєкт, створений Клементом Міхаїлеску як демонстрація роботи алгоритмів пошуку на сітці. Цей застосунок відомий завдяки візуально динамічному інтерфейсу, що дозволяє дослідити поведінку пошукових алгоритмів у режимі реального часу. Його основна мета – спростити інтуїтивне розуміння того, як алгоритми знаходять шлях у сітковому середовищі, що робить його ефективним як для самостійного вивчення, так і для демонстрацій у класі. Інтерфейс застосунку зображено на рис. 2.3, де представлено типовий сітковий граф, точки старту й фінішу, а також приклад створення перешкод, що імітують реальні обмеження простору.

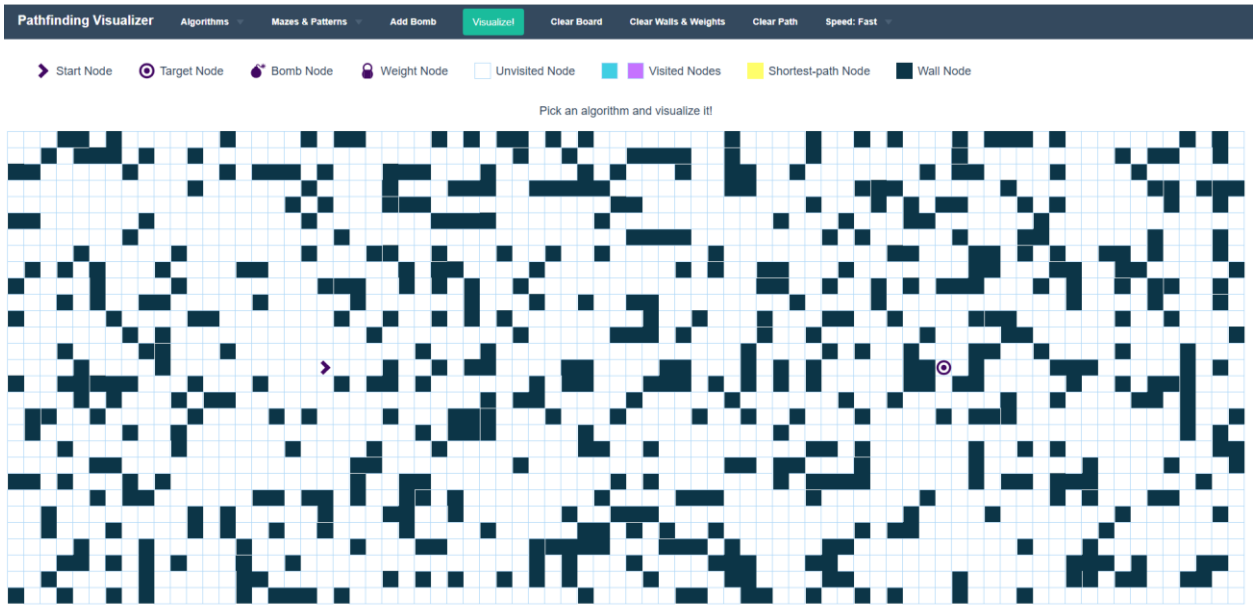

Рисунок 2.3 – Головне меню вебзастосунку Pathfinding Visualizer

Виходячи з інформації в додатку А, табл. А.4, де проаналізовано функціональні можливості цієї платформи відповідно до п’яти освітніх

факторів ефективної візуалізації алгоритмів, Pathfinding Visualizer є прикладом наочної демонстрації пошукових алгоритмів у сітковому середовищі, зосередженої на простоті використання та візуальній ясності. Платформа забезпечує достатній рівень взаємодії та варіативності, дозволяючи користувачу вручну змінювати конфігурацію простору, експериментувати з перешкодами та обирати алгоритми. Незважаючи на відсутність повного контролю над кроками виконання і брак текстового супроводу чи псевдокоду, інтерфейс залишається інтуїтивно зрозумілим, що робить ресурс корисним для початкового знайомства з алгоритмами пошуку та порівняння їхньої поведінки.

2.3 Постановка задачі

З огляду на результати аналізу існуючих платформ (Red Blob Games, VisuAlgo, Pathfinding Visualizer), можна сформулювати концепцію візуалізаційного застосунку, що ґрунтується на сильних сторонах попередніх рішень і прагне враховувати деякі з їхніх обмежень. Основу побудови становлять п’ять освітніх критеріїв ефективної візуалізації алгоритмів, визначених у підрозділі 2.1: активна взаємодія, наочність структури, мультимодальність, кроковість та варіативність.

З урахуванням цього сформульовано перелік основних завдань реалізації:

– реалізувати інтерфейс вибору графа (через карту міст або клітинки сітки);

– створити інтерфейс вибору між різними пошуковими алгоритмами (A*, BFS, DFS, Dijkstra);

– забезпечити підтримку двох типів графів: сіткового та довільного (містового);

– реалізувати можливість вибору стартової та цільової вершини користувачем;

– візуалізувати процес пошуку із затримкою між кроками, щоб підкреслити послідовність дій;

– забезпечити базову анімацію обробки вузлів та побудови знайденого шляху.

На відміну від розглянутих вище проєктів, дана система передбачає поєднання гнучкості графових сценаріїв із прямим візуальним контролем, зберігаючи баланс між простотою й розширюваністю. Проєкт має навчальну спрямованість і призначений як для використання студентами, так і для демонстрацій викладачами у процесі викладання алгоритмів.

3 ОБҐРУНТУВАННЯ АРХІТЕКТУРНО-ТЕХНОЛОГІЧНИХ РІШЕНЬ

3.1 Вибір інструментарію реалізації (мови, фреймворки, бібліотеки)

Для обґрунтованого вибору технологій реалізації вебзастосунку проведено порівняльний аналіз доступних інструментів, мов програмування та бібліотек, які можуть бути використані для побудови інтерфейсу та інтерактивної візуалізації. Таблиця 3.1 узагальнює переваги кожного інструменту, розглядає можливі альтернативи та подає аргументацію конкретного вибору відповідно до потреб проєкту.

Таблиця 3.1 – Порівняльний аналіз обраних технологій та аргументація вибору

Інструмент	Переваги	Альтернативи	Причина вибору
HTML5	Стандартизований і підтримується всіма сучасними браузерами. Забезпечує семантичну структуру сторінки.	XHTML, старіші версії HTML, але вони менш зручні і не підтримують сучасні можливості.	Забезпечує кросбраузерну сумісність та відповідає стандартам W3C.
CSS3	Дозволяє створювати адаптивні, привабливі та структуровані інтерфейси. Підтримує сучасні	Bootstrap, Tailwind, SCSS – потребують додаткової конфігурації або менш гнучкі в кастомізації.	Дозволяє створити візуально привабливий та зручний інтерфейс для

Продовження таблиці 3.1

Інструмент	Переваги	Альтернативи	Причина вибору
CSS3	ефекти та анімації.		взаємодії користувача.
JavaScript	Є універсальною мовою для реалізації логіки поведінки на стороні клієнта. Підтримується всіма браузерами.	TypeScript, jQuery – перший складніший для новачків, другий вважається застарілим.	Надає можливість реалізувати всю логіку взаємодії без необхідності використання бекенду.
Leaflet.js	Спеціалізована бібліотека для інтерактивної роботи з картою. Підтримує маркери, шари, маршрути.	OpenLayers, Google Maps API – перший складніший у використанні, другий має обмеження та ліцензії.	Є легкою, добре задокументованою та популярною бібліотекою з великою спільнотою.
Turf.js	Дозволяє виконувати просторові обчислення (розрахунок відстані, перетини, буфери) прямо в браузері.	OpenLayers, геосервери – більш складні у використанні, потребують серверної частини.	Зручно інтегрується з Leaflet.js і дозволяє виконувати необхідні геообчислення без сервера.

Для повнішого розуміння ролі кожної технології у реалізації вебзастосунку, далі наведено опис, доповнюючий інформацію в таблиці 3.1.

Мову розмітки HTML5 обрано завдяки її стабільності, універсальній підтримці всіма сучасними браузерами, а також здатності ефективно інтегруватися з CSS та JavaScript. На відміну від старіших версій або альтернативних технологій (як-от Flash чи Silverlight), HTML5 не потребує додаткових плагінів, є безпечним, адаптивним і дозволяє реалізувати доступний, семантично структурований інтерфейс користувача [15].

Мову стилів CSS3 обрано як стандарт для оформлення, оскільки вона дозволяє реалізувати адаптивний дизайн, анімації та стилізацію без використання сторонніх бібліотек. Порівняно з inline-стилями або старішими підходами (як-от таблиці стилів у HTML 4.01), CSS3 забезпечує чистіший код, кращу модульність і більшу гнучкість у дизайні інтерфейсу, що критично важливо для підтримки зручного навчального середовища [16].

Мову програмування JavaScript обрано як основну мову програмування на боці клієнта завдяки її універсальності, сумісності з усіма сучасними браузерами та підтримці інтерактивної поведінки на сторінці. Альтернативи на кшталт TypeScript або WebAssembly потребують компіляції або складніших середовищ розробки, тоді як JavaScript дозволяє швидко розробити та протестувати функціонал. Крім того, вона ідеально підходить для реалізації інтерактивної логіки запуску алгоритмів, обробки подій та візуалізації процесу пошуку шляху в режимі реального часу [17].

Leaflet.js – це легка, продуктивна та зручна у використанні бібліотека для інтерактивної роботи з картами. Її вибрано для реалізації графового середовища, оскільки вона надає можливість розміщення елементів на географічному полотні, масштабування, обробки подій та додавання шарів з просторовими об’єктами. Альтернативи, як-от OpenLayers або Mapbox GL, мають більший функціонал, проте потребують більше ресурсів або складнішої конфігурації. Leaflet.js ідеально підходить для задач візуалізації на основі сітки або абстрактного графа, не перевантажуючи систему [18].

Turf.js – це модульна бібліотека для просторового аналізу в браузері, яка дозволяє виконувати геометричні операції, як-от обчислення площ, визначення

належності координат до геооб’єктів, побудову буферів тощо. У межах проєкту її застосовано для перевірки, чи координати міста належать межам вибраної країни. Це дозволяє динамічно формувати граф лише в межах обраної території, що особливо важливо для точності й відповідності візуалізації. На відміну від більш важких рішень, Turf.js повністю сумісна з Leaflet.js і не вимагає сторонніх серверів або API, що забезпечує автономність, швидкість роботи та контроль над геоданими [19].

3.2 Проєктування архітектури системи: логічна та фізична структура

З метою формалізованого представлення архітектури та функціональних залежностей вебзастосунку обрано методологію IDEF0 – одну з найбільш поширених технік моделювання функціональних систем [14]. Її використання дозволяє наочно і структуровано описати основні процеси, що відбуваються в межах системи візуалізації пошукових алгоритмів, а також показати, як ці процеси взаємодіють між собою та з користувачем. поєднати технічну структуру коду з педагогічними завданнями.

Основною функцією розробленого вебзастосунку є ознайомлення користувачів з пошуковими алгоритмами шляхом візуального подання логіки їх роботи в інтерактивному середовищі.. Зазначена функція представлена у вигляді верхньорівневої IDEF0-моделі (блок A0) на рис. 3.1.

Рисунок 3.1 – Верхньорівнева модель основної функції вебзастосунку в нотації IDEF0 (блок A0)

Для повноцінного розуміння структури верхнього рівня системи згідно IDEF0-моделлю, у додатку А, табл. А.1 наведено деталізований опис усіх типів елементів взаємодії з основною функцією: входів, контролю, механізмів і виходів. Кожен із пунктів описує, яку роль відіграє відповідна складова у процесі надання послуг візуалізації пошукових алгоритмів, як це визначено в блоці A0 на рис. 3.1.

Після загального представлення функціональної моделі вебзастосунку на рівні A0, доцільно розглянути її декомпозицію на окремі підпроцеси. На рис. 3.2 наведено деталізовану IDEF0-діаграму, яка описує логіку виконання основної функції – ознайомлення користувачів з пошуковими алгоритмами шляхом візуального подання – через чотири підпроцеси (A1–A4), що відображають ключові етапи взаємодії користувача з системою: від запуску інтерфейсу до демонстрації роботи алгоритмів.

Рисунок 3.2 – Декомпозиція основної функції вебзастосунку в нотації IDEF0 (блок A1–A4)

Усі структурні елементи побудованої діаграми наведено в додатку А, табл. А.2. Далі здійснено декомпозицію підпроцесів А2 і А3 з метою детальнішого представлення їх внутрішньої структури.

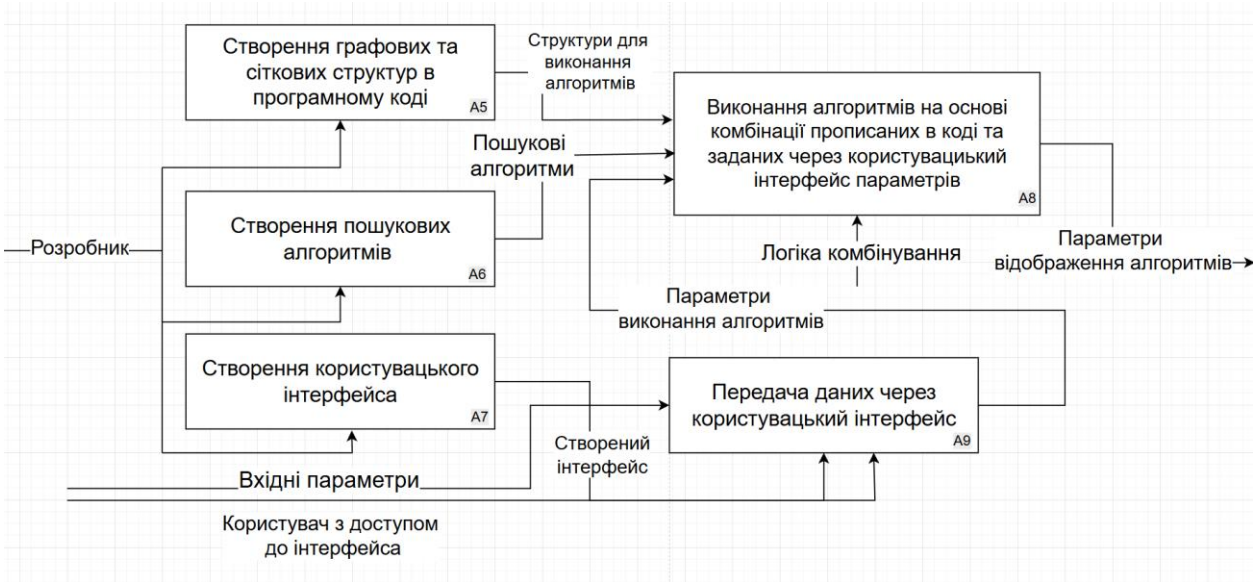

Рисунок 3.3 – Декомпозиція підпроцесів А2 і А3 у нотації IDEF0 (блок А5-А9)

Діаграма ілюструє деталізацію процесів введення параметрів користувачем (А2) та виконання алгоритмів (А3). Вона відображає зв’язок між

створенням користувацького інтерфейсу, логікою реалізації пошукових алгоритмів і взаємодією з параметрами, які задає користувач. Усі структурні елементи побудованої діаграми наведено в додатку А, табл. А.3. Надалі представлено окрему декомпозицію процесу демонстрації роботи алгоритмів (А4) з метою пояснення механізму формування маршруту.

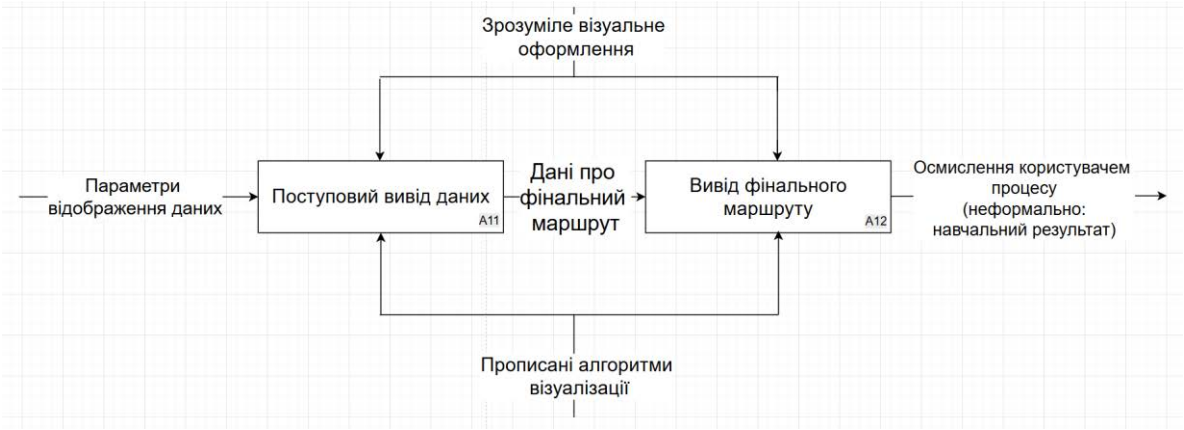

Рисунок 3.4 – Декомпозиція підпроцесу А4 у нотації IDEF0 (блок А11-А12)

Усі структурні елементи побудованої діаграми наведено в додатку А, табл. А.4.

Завершивши побудову логічної структури системи засобами IDEF0-моделювання, можна перейти до розгляду схеми фізичної реалізації проєкту. Структурна діаграма на рис. 3.5 відображає загальні принципи взаємодії між компонентами вебзастосунку.

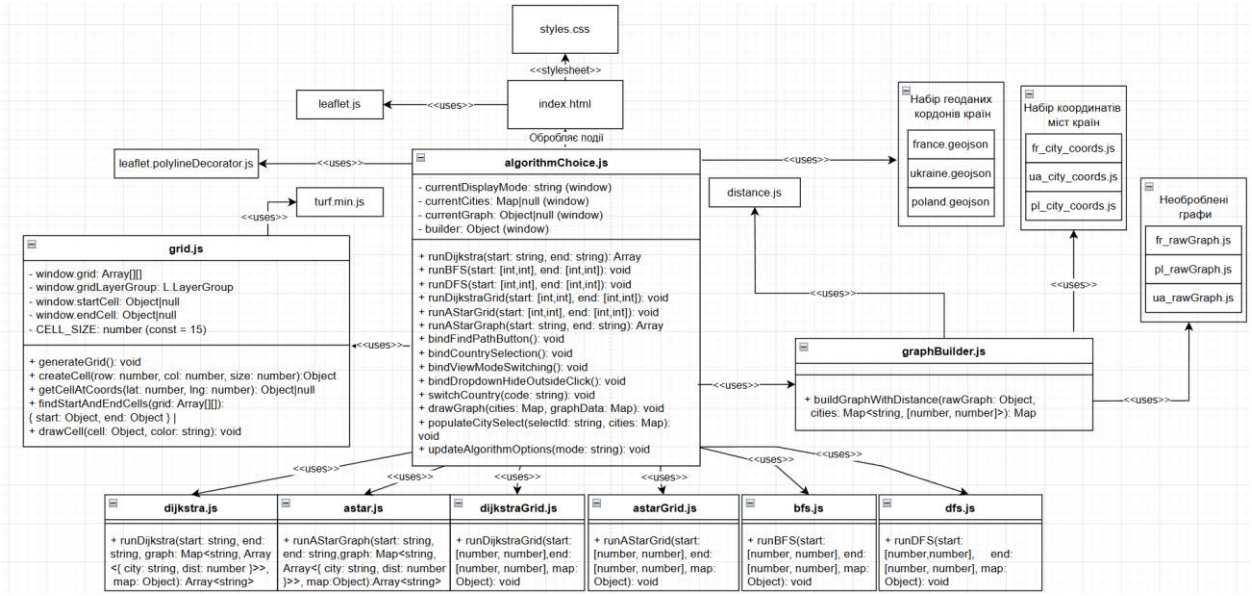

Рисунок 3.5 – Структурна діаграма компонентів вебзастосунку

З рис 3.5 видно, що вебзастосунок реалізовано у вигляді клієнтської односторінкової програми, структура якої умовно поділяється на три рівні: інтерфейс користувача (UI), модулі логіки (JS) та дані для візуалізації. Основою інтерфейсу слугує файл index.html, який містить базову розмітку, панелі керування й елементи взаємодії з користувачем. Візуальне оформлення визначається у файлі styles.css.

Функціональну частину застосунку реалізовано за допомогою JavaScript-модулів, розділених за призначенням:

– алгоритми (модулі bfs.js, dfs.js, dijkstra.js, де кожен файл містить окрему реалізацію пошукового алгоритму, яку можна викликати з головного модуля);

– графова логіка (модулі graphBuilder.js, ua_rawGraph.js, ua_rawGraph.js та ін., які відповідають за зчитування геоданих та побудову структури графа);

– сіткова логіка (файл grid.js, який містить функції побудови сітки на основі координат і взаємодії з користувачем у режимі grid);

– керування алгоритмами (файл algorithmChoice.js, який об'єднує обробку подій запуску, виклик відповідного алгоритму та взаємодію з візуалізацією);

– допоміжні бібліотеки (turf.min.js для геообчислень, а також leaflet.polylineDecorator.js для стилізації маршрутів на карті).

Візуалізація здійснюється за допомогою бібліотеки Leaflet.js, яка дозволяє відображати як сітку (у вигляді клітинок), так і графи – на основі GeoJSON-файлів, зокрема ukraine.geojson, poland.geojson, france.geojson. Географічні координати зберігаються у відповідних файлах, таких як ua_city_coords.js, pl_city_coords.js тощо.

Вебзастосунок на поточному етапі функціонує локально в браузері, не потребуючи серверної частини, що забезпечує максимальну швидкість взаємодії, простоту розгортання та незалежність від зовнішніх ресурсів. У перспективі можлива інтеграція серверної логіки, бази даних та розгортання застосунку у відкритому доступі через інтернет.

3.3 Обґрунтування UI/UX рішень на основі сценаріїв користувача

Під час створення інтерфейсу системи візуалізації пошукових алгоритмів (див. рис. 3.6) головним завданням було забезпечити освітню ефективність і зручність використання. Для цього проєктування здійснювалося на основі принципів UX-дизайну, орієнтованих на зменшення когнітивного навантаження та підвищення інтерактивності[20].

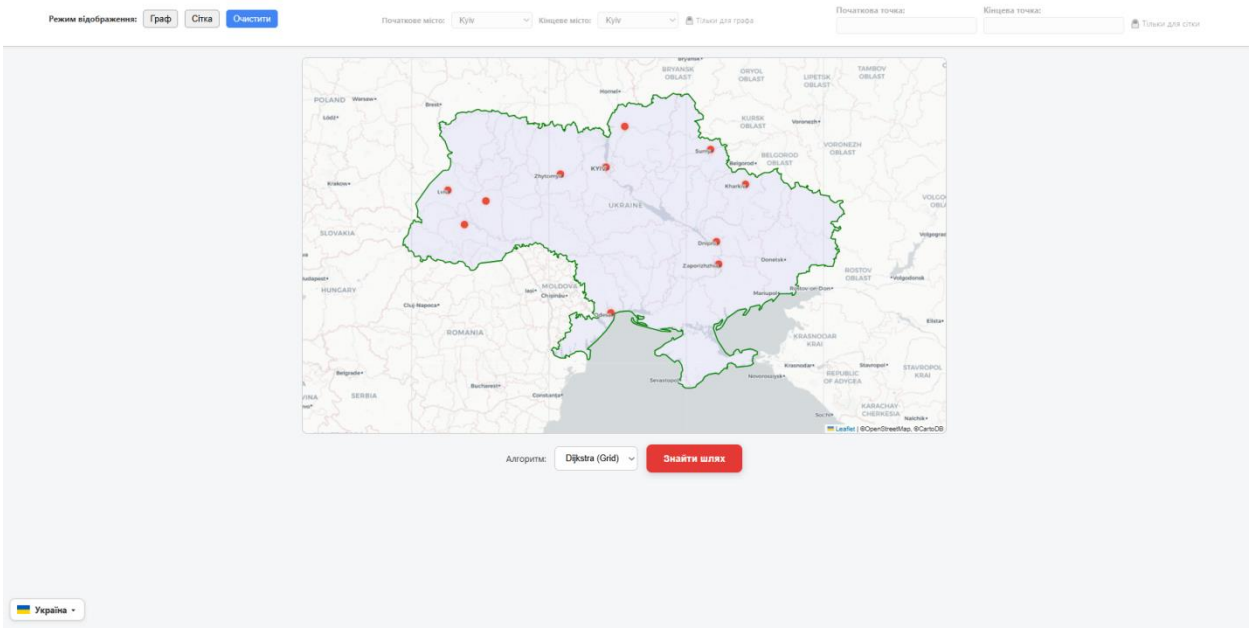

Рисунок 3.6 – Інтерфейс системи візуалізації пошукових алгоритмів

Зокрема, було обрано шість ключових принципів інтерфейсного дизайну, які забезпечують баланс між функціональністю та простотою [20]. Водночас, реалізовані елементи інтерфейсу безпосередньо відповідають типовим сценаріям взаємодії, у яких студент змінює параметри, спостерігає за алгоритмом або досліджує його логіку. Нижче розглянуто обидві ці площини – спочатку як набір принципів, потім як набір сценаріїв, – що у сукупності дає повну картину обґрунтованості прийнятих UI/UX рішень у даному вебзастосунку. Почнемо з переліку набору принципів інтерфейсного дизайну, він складається з:

– принцип мінімалізму та фокусування на функціональності;
– використання кольорових метафор для покрокового слідкування;
– принцип "контроль без перевантаження";
– підтримка сценаріїв поступового ознайомлення;
– відображення статусу алгоритму в реальному часі;
– відсутність блокуючих дій та гнучкість взаємодії.

Реалізацію кожного принципу детальніше розглянуто в додатку Б, табл Б.1.

Оскільки під час розробки інтерфейсу системи візуалізації пошукових алгоритмів ключовим орієнтиром стали потреби кінцевого користувача – студента, який навчається основ алгоритмічного мислення, дизайн було побудовано з урахуванням типових сценаріїв взаємодії та базових принципів UX, які описано у додатку Б, табл. Б.2.

Таким чином, розроблений інтерфейс поєднує в собі принципи інтуїтивного дизайну, педагогічно обґрунтовані UX-рішення та практичні сценарії взаємодії. Він не лише підтримує ефективне засвоєння алгоритмічного матеріалу, але й забезпечує гнучке середовище для самостійного навчання, експериментування та візуального дослідження логіки пошуку в графах.

4 РЕАЛІЗАЦІЯ ТА ВЕРИФІКАЦІЯ ВЕБ-ЗАСТОСУНКУ

4.1. Опис реалізації основних програмних модулів

Також варто розглянути логіку роботи кожного з основних файлів, включаючи функціональні блоки, методи, структури даних та взаємодію з інтерфейсом користувача. Для ілюстрації використано фрагменти коду (лістинги) та діаграму залежностей, яка представлена на рис. 3.5, що демонструє фізичну структуру і взаємозв’язки між компонентами системи.

По-перше слід розглянути модуль algorithmChoice.js, який виконує роль центрального компонента в логіці вебзастосунку. Він зображений у центрі діаграми на рис. 3.5 як ключова ланка між користувацьким інтерфейсом, реалізованими алгоритмами пошуку, структурою графа та сітковою моделлю. Саме цей модуль відповідає за обробку подій взаємодії користувача із застосунком, запуск відповідного алгоритму, а також оновлення стану карти залежно від вибраного режиму (граф або сітка). Крім того, через algorithmChoice.js здійснюється імпорт необхідних функціональних модулів та динамічне налаштування параметрів, що використовуються іншими частинами системи. У верхній частині модуля визначено глобальні змінні, які зберігають поточний стан застосунку та забезпечують доступ до ключових даних з інших частин програми (див. лістинг 4.1)

```
window.currentDisplayMode = 'none';
window.currentCities = null;
window.currentGraph = null;
window.builder = null;
```

Лістинг 4.1 – Глобальні змінні у модулі algorithmChoice.js

Зміст змінних наступний:

– currentDisplayMode – режим візуалізації, що обраний користувачем: "graph", "grid" або "none". Ця змінна визначає, які елементи інтерфейсу будуть активні та який тип алгоритму доступний.

– currentCities – структура Map, яка містить координати міст обраної країни. Вона завантажується динамічно при перемиканні між країнами і використовується для побудови графа.

– currentGraph – необроблений граф країни, тобто структура типу Map<string, Array<string>>, що описує зв’язки між містами. Застосовується як вхід до побудови зваженого графа.

– builder – результат виконання функції buildGraphWithDistance(...) з модуля graphBuilder.js, яка створює структуру з вагами ребер між містами. Цей об'єкт необхідний для роботи алгоритмів Dijkstra та A* у графовому режимі.

Окрема частина модуля відповідає за ініціалізацію взаємодії користувача з елементами інтерфейсу. Зокрема, встановлюються обробники подій на кнопки перемикання режимів візуалізації (graph, grid, none), кнопку запуску пошуку шляху (findPathBtn), а також на випадаюче меню вибору країни. Ці елементи забезпечують повний цикл керування застосунком з боку користувача. Обробник кнопки запуску шляху наведено у лістингу 4.2

```
    document.getElementById('findPathBtn').addEventListener('cli
ck', () => { ... });
```

Лістинг 4.2 – Обробник подій кнопки findPathBtn

Завдяки йому, в залежності від вибраного алгоритму (dijkstra, bfs, dfs, astar, dijkstraGrid, astarGrid), викликається відповідна функція з імпортованих модулів. Перед запуском виконуються перевірки: наприклад, чи обрані початкова та кінцева точки, чи доступна карта тощо. Таким чином, дана логіка забезпечує адаптивність залежно від режиму (graph або grid) та гарантує, що запуск відбудеться лише за наявності необхідних умов.

Ще один важливий блок слухачів пов’язаний із перемиканням режиму візуалізації наведено у лістингу 4.3

```
document.querySelectorAll('.toggle-btn').forEach(btn => {
  btn.addEventListener('click', () => { ... });
});
```

Лістинг 4.3 – Обробник події перемикання режиму відображення (граф або сітка)

Цей обробник відповідає за оновлення інтерфейсу та зміни глобального режиму currentDisplayMode. Також динамічно оновлюється список доступних алгоритмів (algorithmSelect) відповідно до обраного режиму. Крім того, автоматично активується відповідна панель параметрів (graph-controls, grid-point-panel) і оновлюється відображення карти. Нарешті, додано слухачі до елементів випадаючого меню вибору країни (див. лістинг 4.4)

```
document.querySelectorAll('.country-item').forEach(item => {
  item.addEventListener('click', () => { ... });
});
```

Лістинг 4.4 – Обробник події зміни вибраної країни (Франція, Україна або Польща)

Завдяки ним користувач може обрати країну із списку, після чого викликається функція switchCountry(), яка завантажує відповідні координати міст, необроблений граф та географічні межі з GeoJSON-файлів, а також оновлює карту й інтерфейс.

Функція switchCountry() (див. додаток Г, лістинг Г.1) є однією з ключових функцій у структурі модуля, вона відповідає за зміну активної країни на мапі та оновлення відповідних візуалізаційних і логічних компонентів. Її виклик відбувається під час вибору користувачем країни з меню, і вона є критично важливою для реалізації мультирегіонального підходу.

Після отримання коду країни (FR, UA або PL), функція:

- асинхронно імпортує координати міст, сирий граф і геодані кордонів;
- оновлює глобальні змінні currentCities, currentGraph і builder, використовуючи функцію buildGraphWithDistance(...);
- очищає попередні графічні шари карти (міста, граф, кордони, сітка);
- повторно відображає маркери міст через Leaflet API;
- у разі активного режиму graph – викликає drawGraph(...) для побудови візуалізації зв'язків;
- завантажує та додає на карту полігон з кордонами країни у форматі GeoJSON.

Це забезпечує перемикання між країнами без потреби в перезавантаженні сторінки.

Центральною логікою взаємодії з користувачем у модулі algorithmChoice.js є обробник натискання кнопки «Знайти шлях» (findPathBtn). Цей обробник прив'язаний до кнопки з ідентифікатором #findPathBtn і відповідає за запуск пошукового алгоритму на основі вибору користувача в інтерфейсі (див. додаток Г, лістинг Г.2). При натисканні, він зчитує обраний алгоритм, стартову та кінцеву точки, після чого викликає відповідну функцію запуску, яка реалізує конкретну пошукову логіку для графа або сітки. У разі некоректного вводу (наприклад, не вибрані точки) виконується перевірка з повідомленням користувачу.

Функція «updateAlgorithmOptions», яку наведено у додатку Г, лістингу Г.3, відповідає за динамічне оновлення переліку алгоритмів у випадаючому меню, в залежності від активного режиму: граф або сітка.

Користувач може працювати або в режимі графа (де алгоритми працюють з містами та з'єднаннями), або в режимі сітки (де працюють клітинкові алгоритми). Щоб уникнути плутанини та забезпечити інтуїтивний UX, ця функція очищує <select> і додає тільки відповідні варіанти.

Функція drawGraph(cities: Map, graphData: Map) відповідає за візуалізацію графа на мапі Leaflet на основі даних про міста й графові зв'язки

між ними (див. додаток Г, лістинг Г.4). Призначення: Відобразити всі ребра графа, використовуючи координати з мапи міст та структуру суміжності з графа. Основним алгоритмом дій є проходження по кожному вузлу графа, створення polyline для кожного сусіда та додавання декоративної стрілки з бібліотеки polylineDecorator.

Функція populateCitySelect(), яку наведено у додатку Г, лістингу Г.5, відповідає за заповнення випадаючого списку міст на основі Map-структури з назвами та координатами та дає користувачу можливість вибрати початкове та кінцеве місто для запуску алгоритму. Основним алгоритмом дій є отримання посилання на тег <select>, очищення його вмісту, додання кожного міста як <option>.

Для забезпечення роботи алгоритмів пошуку шляху в проєкті реалізовано два допоміжні модулі, які відповідають за формування базових структур даних: графа у випадку алгоритмів на основі міст (Dijkstra та A*) та сітки у випадку алгоритмів для двовимірного простору (BFS, DFS, Dijkstra Grid, A* Grid). Ці модулі – graphBuilder.js і grid.js – виконують ключову функцію підготовки вхідних даних у відповідному форматі. Саме на їх основі відбувається подальша робота основних алгоритмів, що реалізовані в окремих модулях. Далі наведено детальний опис логіки цих двох модулів.

Для режиму роботи з графом реалізовно функцію buildGraphWithDistance(), яку наведено у додатку Г, лістингу Г.6. Вона формує зважений орієнтований граф на основі початкової карти з'єднань (rawGraph) та координат міст (cityCoords). Функція дозволяє представити міста як вузли графа, а шляхи між ними – як орієнтовані ребра з довжинами, які відповідають географічним відстаням. На виході маємо Map, де кожне місто має список об’єктів формату { name: neighborName, distance: value }, які можна безпосередньо подавати алгоритмам Dijkstra чи A*, що працюють із графами.

У режимі роботи з сіткою кожна клітинка представляє вершину графа, а її сусіди – це безпосередньо прилеглі (вгору, вниз, вліво, вправо) клітинки. Модуль grid.js відповідає за створення такої сітки, відображення її на мапі та

взаємодію з користувачем через вибір стартової та цільової клітинки. Цей модуль реалізує як логічну, так і візуальну структуру, на якій потім працюють пошукові алгоритми.

Функція generateGrid() є основним методом побудови прямокутної сітки на карті (див. додаток Г, лістинг Г.7). Вона ініціалізує глобальні змінні grid, startCell, endCell та створює шар gridLayerGroup, який відповідає за відображення клітинок на карті Leaflet. Границями сітки виступають координати області перегляду карти, які діляться на рівномірні прямокутники з кроком latStep по широті та lngStep по довготі. Перед додаванням прямокутника функція перевіряє, чи центр кожної клітинки знаходиться всередині кордонів обраної країни (через метод turf.booleanPointInPolygon). Якщо умова виконується, створюється прямокутник Leaflet, який додається на карту і стає клітинкою (cell). Кожна така клітинка може виконувати роль стіни, стартової або фінішної точки, яка визначається клацанням миші. Стилізація та оновлення пов’язаних полів вводу відбувається динамічно залежно від стану. Всі допустимі клітинки зберігаються у двовимірному масиві grid.

Розглянемо функцію runDijkstra(), яка є алгоритмом пошуку Дейкстри для графового режиму та реалізована в окремому модулі dijkstra.js (див. додаток Г, лістинг Г.8). Ця функція відповідає за пошук найкоротшого шляху між двома містами на графі з урахуванням ваг (відстаней) між вершинами. У контексті проєкту граф формується як об'єкт типу Map, де ключами є назви міст, а значеннями – масиви об'єктів-сусідів з вказаними відстанями.

Функція runDijkstra(start, end, graph, map) отримує чотири параметри: start (назва початкового міста), end (назва кінцевого міста), graph (побудований зважений граф), map (об'єкт Leaflet-карти для подальшої візуалізації).

На першому етапі функція ініціалізує допоміжні структури: distances (карта відстаней від стартового вузла до інших), previous (карта попередників для відновлення шляху), visited (множина вже відвіданих вузлів).

Далі відбувається ініціалізація всіх відстаней як нескінченність, крім стартового вузла, для якого встановлюється 0. Основний цикл ітеративно

вибирає найближчу неперевірену вершину, оновлює значення її сусідів (релаксація) та додає її до списку відвіданих. Якщо на якомусь етапі обрана вершина збігається з кінцевою, алгоритм завершується достроково. Завершення циклу означає, що або знайдено найкоротший шлях, або між вузлами немає з'єднання. На завершальному етапі шлях відновлюється за допомогою структури previous, і функція повертає масив міст, які утворюють найкоротший маршрут. Цей масив повертається як результат роботи функції runDijkstra(). У проєкті ця функція використовується безпосередньо в algorithmChoice.js.

Функція runAStarGraph(start, end, graph, map) реалізує алгоритм A* (A-star) для знаходження найкоротшого шляху в орієнтованому графі, де вершини представляють міста, а ребра – відстані між ними (див. додаток Г, лістинг Г.9). Основною відмінністю від алгоритму Дейкстри є використання евристичної функції – у цьому випадку для використовується географічна відстань між містами, розрахована за формулою гаверсинуса (Haversine distance). Такий підхід дозволяє ефективно оцінювати відстань "напряму" між вершинами та забезпечує евристичну допустимість у географічних графах.*. Як і в runDijkstra, для зберігання стану використовуються distances, previous, visited, а також додатково – estimatedTotal, що враховує евристичну оцінку. У процесі виконання відбувається поступова візуалізація розглянутих ребер жовтими лініями, а коли шлях знайдено, фінальний маршрут виводиться червоною лінією. Вбудована візуалізація дозволяє користувачу інтуїтивно зрозуміти, як працює стратегія A*, чим вона відрізняється від Дейкстри, і як евристика впливає на поведінку пошуку.

Функція runBFS(start, end, map), яку наведено у додатку Г, лістингу Г.10, реалізує алгоритм пошуку в ширину для знаходження найкоротшого шляху між клітинками сітки. Цей підхід не враховує ваги або відстані, а оперує лише структурою сітки, просуваючись рівнями від стартової точки до цільової. Для зберігання стану використовується черга, в яку додаються сусідні клітинки (що не є стінами і ще не були відвідані). Також ведеться облік попередників для

кожної відвіданої клітинки з метою відновлення шляху після досягнення фінішу.

Під час виконання алгоритму кожен крок візуалізується: оброблені клітинки забарвлюються в сірий або інший колір, що дозволяє користувачеві бачити хід пошуку. Після знаходження шляху клітинки маршруту підсвічуються червоним. Завдяки цій динаміці студент може зрозуміти, як працює стратегія пошуку в ширину, як формуються фронти пошуку, і чому цей алгоритм завжди знаходить найкоротший шлях у кількості кроків, але не враховує геометричну відстань.

Функція runDijkstraGrid(start, end, map), яку наведено у додатку Б, лістингу Б.11, застосовує класичний алгоритм Дейкстри для знаходження найкоротшого шляху на сітці, де кожна клітинка є вершиною графа, а ребра між сусідніми клітинками мають однакову вагу. Алгоритм поступово оновлює найменші відстані до всіх досяжних клітинок, починаючи з початкової, поки не буде знайдена фінішна точка або всі можливості не вичерпані.

Особливістю реалізації є використання пріоритетної черги (або її симуляція), що дозволяє ефективно обирати клітинку з найменшою відстанню на кожному кроці. Візуально користувач бачить поступове заповнення сітки хвилею, що демонструє рівномірне «розтікання» пошуку. Після досягнення мети відображається знайдений шлях.

4.2. Верифікація функціональної коректності застосунку

Для підтвердження того, що розроблений вебзастосунок працює відповідно до визначених вимог, було проведено ручне тестування основних функціональних сценаріїв. Верифікація охоплювала як поведінку користувацького інтерфейсу, так і правильність виконання пошукових алгоритмів на графі та сітці. Основною метою тестування було забезпечення того, що кожна з реалізованих функцій спрацьовує відповідно до логіки, описаної в технічному проєкті.

Одним з ключових етапів стала перевірка коректної ініціалізації карти та побудови графа після вибору країни (рис 4.1). У результаті дій користувача на екрані мають з'явитися маркери міст та ребра, що з’єднують їх згідно з графовою структурою. Цей сценарій підтверджує, що механізм імпорту geoJSON-даних, координат та побудови графа працює правильно.

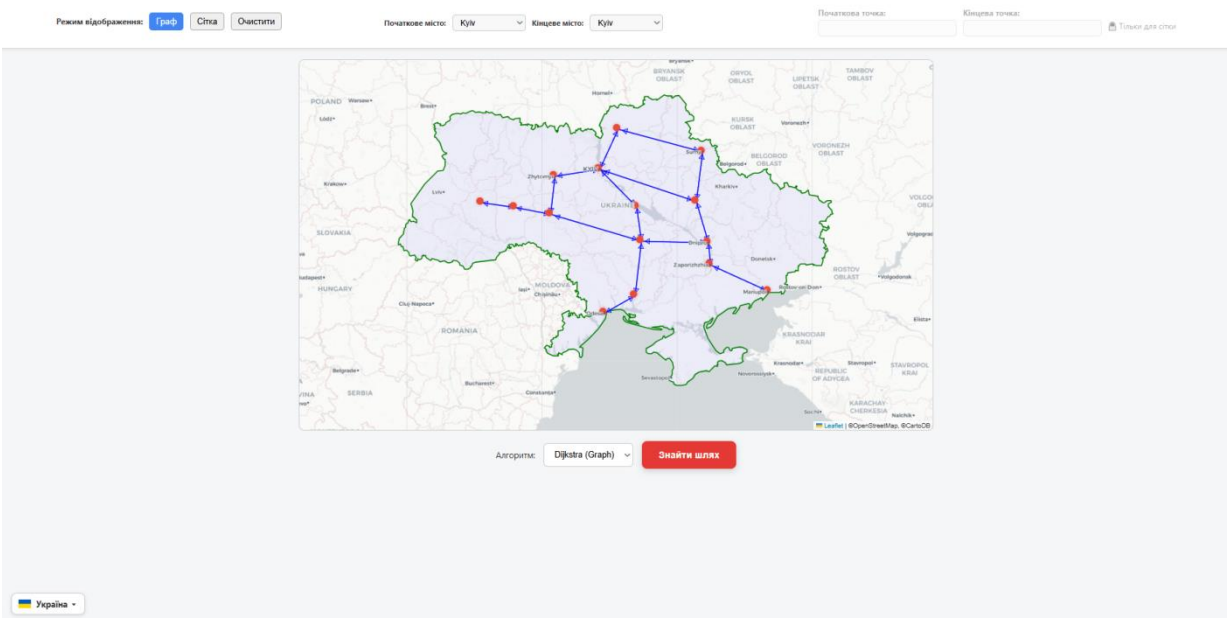

Рисунок 4.1 – Перевірка ініціалізації карти та побудови графа

Після вибору початкового та кінцевого міста та запуску алгоритму Дейкстри, система повинна спочатку відобразити процес обчислення найкоротших шляхів у вигляді пунктирних ліній (що відповідають перевіреним ребрам), а потім – остаточний найкоротший маршрут червоною лінією. Така візуалізація дозволяє верифікувати, що обчислений шлях відповідає очікуваному результату згідно з вагами графа (див. рис 4.2).

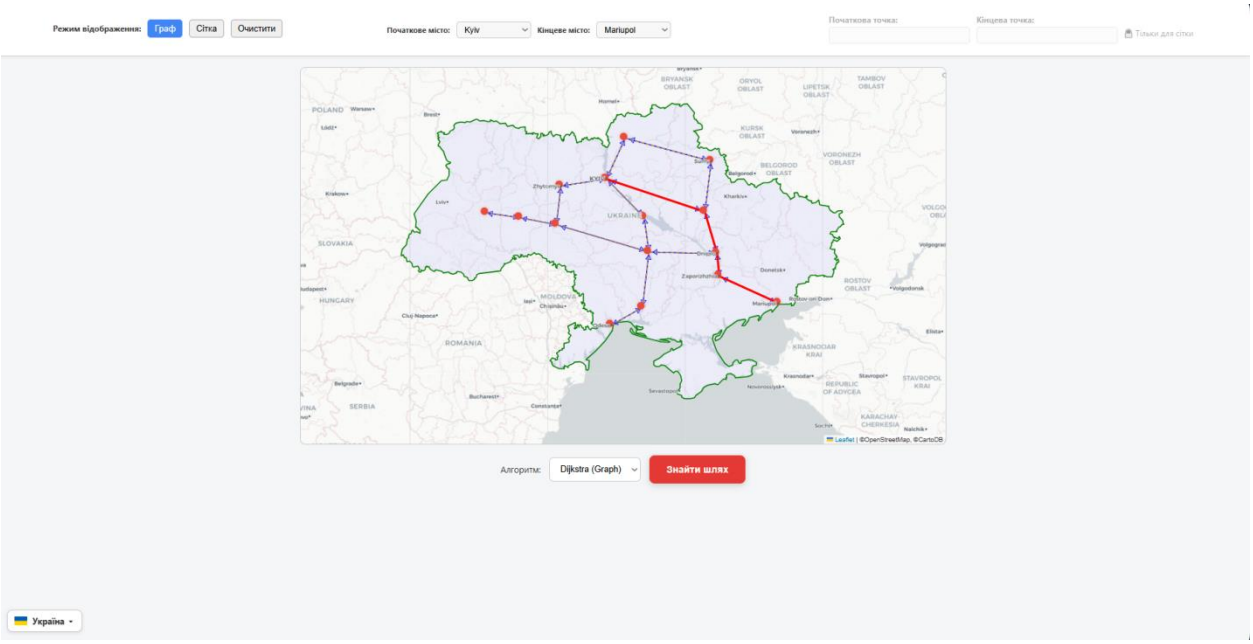

Рисунок 4.2 – Перевірка побудови шляху в графовому режимі

Для верифікації роботи алгоритмів на сітці було обрано два типових сценарії: просте прямолінійне з’єднання точок (BFS) та пошук ефективного маршруту за допомогою евристики (A*). У першому випадку, користувач вручну встановлював початкову та кінцеву точку на відкритій сітці, після чого алгоритм BFS послідовно перевіряв клітинки в ширину, що наочно демонструвалося завдяки поступовому підсвічуванню вузлів. Згідно з теоретичними очікуваннями, BFS завжди знаходив маршрут із мінімальною кількістю переходів, результат зображено на рис. 4.3.

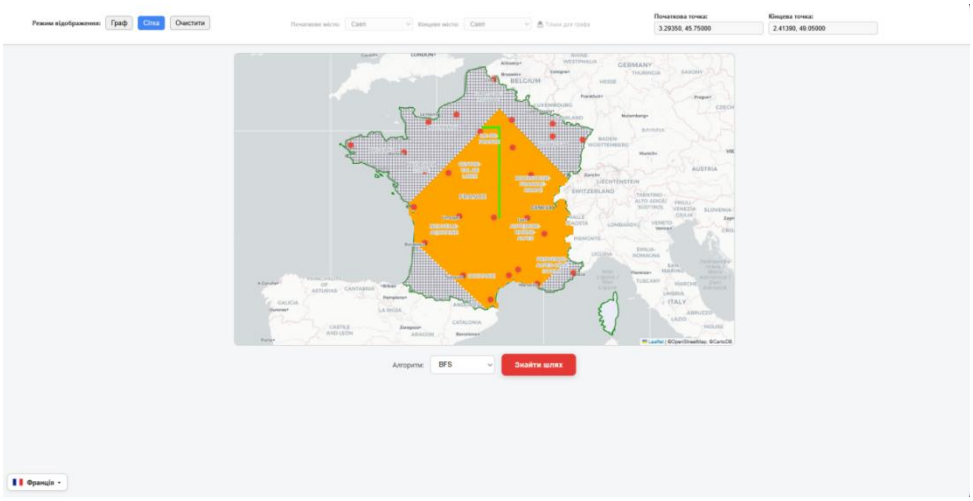

Рисунок 4.3 – Результат роботи алгоритма пошуку в ширину в режимі сітки

У другому випадку алгоритм A* також працював на сітці без ваги, проте при виборі наступних клітинок для обробки враховував евристичну відстань до цільової точки. Це дозволяло зменшити кількість перевірених вузлів і будувати більш прямолінійні траєкторії (див. рис. 4.4). Важливо, що обидва алгоритми забезпечували коректне знаходження маршруту, але їхня поведінка суттєво відрізнялася – саме це демонструвалося завдяки поступовій анімації, що підсвічує як перевірені клітинки, так і фінальний маршрут.

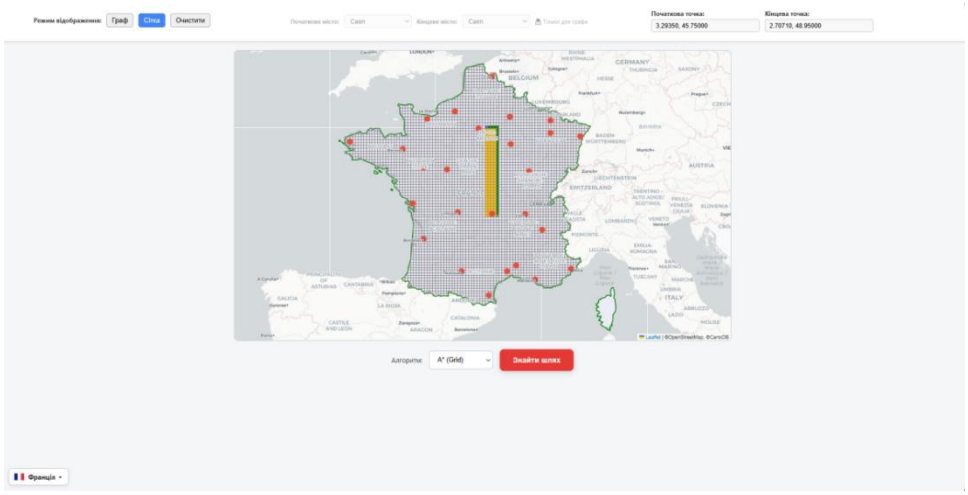

Рисунок 4.4 – Результат роботи алгоритма A* пошуку в режимі сітки

Таким чином, верифікація вебзастосунку підтвердила відповідність фактичної поведінки очікуваній для кожного з реалізованих алгоритмів. Зокрема, користувач мав змогу вибирати режим (граф чи сітка), початкову та кінцеву точки, а також тип алгоритму. Усі ці комбінації правильно оброблялися системою: запуск алгоритмів супроводжувався візуалізацією перевірених вершин або клітинок, а фінальний маршрут позначався окремим кольором. Жодних аномалій, логічних помилок чи зависань під час тестування не було зафіксовано, що дозволяє зробити висновок про функціональну коректність розробленого застосунку.

5 ОЦІНКА РЕЗУЛЬТАТІВ ТА НАПРЯМИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

5.1. Оцінювання ефективності реалізованого програмного рішення

Розроблений візуалізаційний застосунок для алгоритмів пошуку шляху було створено відповідно до попередньо визначених функціональних вимог і освітніх критеріїв. Для оцінки його ефективності проведено систематичний аналіз реалізованих функцій, відповідності цільовому призначенню, а також технічної якості інтерфейсу. Оцінювання здійснюється на основі відповідності п’яти ключовим параметрам, сформульованим у підрозділі 2.1: активна взаємодія, наочність структури, мультимодальність, кроковість та варіативність.

Перший параметр, активна взаємодія, реалізований через можливість користувача безпосередньо впливати на структуру графа: вибирати стартову та кінцеву вершини, перемикатися між сітковим та графовим режимами, редагувати стіни на клітинках сітки та керувати запуском алгоритмів. Така інтерактивність стимулює дослідницький інтерес і дозволяє користувачу перевіряти власні гіпотези щодо поведінки алгоритму.

Другий параметр, наочність структури, реалізований через семантичну відповідність візуального представлення структурі даних. У сітковому режимі використано прямокутну сітку, кожна клітинка якої відображає окремий вузол графа; у графовому – географічну карту міст із маркерами та дугами, що відображають сусідство між вершинами. В обох випадках візуальна структура напряму пов'язана з логікою алгоритму.

Третій параметр, мультимодальність, забезпечено поєднанням графічних, текстових і числових елементів: карта або сітка, підсвічування пройдених вузлів, поступова анімація побудови шляху, відображення координат, імен міст та виведення повідомлень про помилки. Це дозволяє адаптуватися до різних стилів сприйняття інформації користувачем.

Четвертий параметр, кроковість, реалізовано за допомогою затримок між ітераціями алгоритму. Поступова обробка клітинок або міст дає змогу користувачу простежити, як алгоритм формує вибір, змінює відстані та будує остаточний маршрут. Така покрокова анімація робить невидимі внутрішні процеси алгоритму зрозумілими.

П’ятий параметр, варіативність, підтримується через можливість запуску різних алгоритмів (BFS, DFS, A*, Dijkstra), використання двох типів графів, зміну структури вхідних даних (через редагування графа або сітки) і обрання стартових та цільових точок. Завдяки цьому забезпечується широкий спектр навчальних сценаріїв – від простих до складних.

5.2. Аналіз отриманих результатів у контексті навчального застосування

У контексті освітнього процесу розроблений вебзастосунок продемонстрував значний потенціал як інструмент інтерактивного вивчення пошукових алгоритмів. Його структура та функціональні можливості узгоджуються з ключовими чинниками ефективності навчання з використанням візуалізаційних інструментів.

Завдяки гнучкій підтримці двох типів графів – географічного (містового) та сіткового – студенти можуть порівнювати роботу одного й того ж алгоритму в різних умовах: реалістичній топології, що моделює транспортні шляхи між містами, та абстрактній дискретній площині. Це розширює їхнє розуміння абстрактних структур графів і адаптивність алгоритмів до реальних задач.

Особливо ефективним є застосування анімаційного підходу. Поступова візуалізація проходження вузлів і побудови шляху дозволяє студентам буквально «бачити» логіку алгоритму, що важко досягти за допомогою лише текстових або псевдокодових пояснень. Така форма подачі інформації значно знижує когнітивне навантаження та сприяє кращому засвоєнню матеріалу.

Окрема увага заслуговує можливість проведення експериментів: студенти можуть змінювати розміщення стін у сітці, модифікувати стартові та цільові точки, а також спостерігати, як це впливає на поведінку алгоритму. Це заохочує до формування гіпотез, їх перевірки та розвитку інженерного мислення.

Таким чином, результати реалізації демонструють, що застосунок не лише відповідає освітнім стандартам, а й перевищує можливості статичних або лінійних інструментів викладання. Його використання може стати цінним компонентом у сучасному навчальному середовищі, зокрема у змішаному або дистанційному форматі.

5.3. Пропозиції щодо розвитку функціоналу та масштабування проєкту

Поточна реалізація веб-застосунку демонструє високий рівень модульності, що відкриває широкі перспективи для подальшого розвитку системи. Архітектура побудована таким чином, що кожен функціональний блок – генерація сітки, побудова графа, алгоритми пошуку, UI-компоненти – є незалежним модулем з чітко визначеним інтерфейсом. Завдяки цьому можливо не лише легко підтримувати і розширювати існуючий код, а й інтегрувати нові алгоритми або типи візуалізації без порушення основної логіки застосунку.

Одним із ключових напрямів масштабування є розширення географічного охоплення та інтеграція з реальними геоданими. Вже зараз у системі реалізовано підтримку декількох країн через geoJSON-файли (наприклад, France.geojson, Ukraine.geojson), що дозволяє обмежити генерацію сітки відповідними адміністративними межами. Це створює потенціал для використання в освітніх курсах із геоінформатики, логістики, просторового аналізу, а також для створення тематичних симуляцій (наприклад, евакуаційні маршрути, логістичні задачі, аналіз доступності).

Окрему перспективу становить підтримка динамічних або користувацьких geoJSON-файлів, що відкриває шлях до інтеграції з платформами на кшталт OpenStreetMap. Завдяки використанню бібліотеки Leaflet, яка забезпечує відображення геопросторових даних на інтерактивній мапі, графи у застосунку мають безпосередній зв’язок із координатами реального світу. Вузли графа відповідають географічним точкам – містам, а ребра відображають зв’язки між ними на основі просторових відстаней. Такий підхід дозволяє не лише візуалізувати алгоритми пошуку у знайомому користувачам середовищі, але й надалі відштовхуватися від цієї інтеграції для моделювання задач у логістиці, міському плануванні чи навігації. Це розширює застосунок за межі суто навчального інструмента і робить його потенційно корисним у практичних геоінформаційних системах.

Також варто розглянути можливість додавання нових алгоритмів, зокрема тих, що враховують вагу, обмеження або час – наприклад, A* з динамічними евристиками, IDA*, алгоритми пошуку із запам’ятовуванням (Memory-efficient search). Завдяки гнучкій модульній структурі такі інтеграції є технічно досяжними без істотної перебудови застосунку.

Підсумовуючи, проект має достатній потенціал не лише для використання в поточному освітньому форматі, але й для еволюції у повноцінну платформу з можливістю персоналізації, підтримки користувацьких сценаріїв і багатопрофільного застосування.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено вебзастосунок для візуалізації пошукових алгоритмів у графовому та сітковому середовищі. Система реалізована засобами HTML, CSS та JavaScript і функціонує повністю локально в браузері без потреби у серверній частині.

Функціональна логіка побудована на основі модульної архітектури. Усі алгоритми пошуку реалізовано в окремих файлах директорії Algorithms, де зокрема представлені: astar.js, dijkstra.js, bfs.js, dfs.js, а також окремі реалізації для сіткової структури (astarGrid.js, dijkstraGrid.js). Це забезпечує гнучке розширення та супровід коду. Усього реалізовано шість алгоритмів, зокрема A*, Dijkstra, сітковий A*, сітковий Dijkstra, BFS та DFS.

Основний контролер логіки – файл algorithmChoice.js, який відповідає за обробку подій, запуск алгоритмів та оновлення візуалізації відповідно до вибраного режиму. Залежно від типу структури (граф або сітка), дані передаються в один з модулів алгоритмів, після чого виконується покрокова анімація маршруту.

Графова структура зберігається у вигляді словників суміжності в файлах fr_rawGraph.js, pl_rawGraph.js, ua_rawGraph.js, а координати міст – у відповідних файлах fr_city_coords.js, pl_city_coords.js, ua_city_coords.js. Для обчислення відстаней між вершинами використовується distance.js, який реалізує гаверсинову метрику. Побудова графа відбувається за допомогою модуля graphBuilder.js, що генерує зважену структуру на основі географічних даних.

Сітковий режим реалізовано у модулі grid.js, де формується клітинне поле з можливістю задання стартової та цільової точок. Для перевірки належності клітин до території країни використовується бібліотека Turf.js (turf.min.js) у поєднанні з GeoJSON-файлами (france.geojson, poland.geojson, ukraine.geojson), які визначають кордони відповідних країн.

Візуалізація реалізована за допомогою бібліотеки Leaflet.js, зокрема її додаткового компонента leaflet.polylineDecorator.js, що забезпечує оформлення маршрутів. Карта ініціалізується безпосередньо в HTML-документі (index.html) через тег <script>, без використання окремого модуля.

Користувацький інтерфейс побудовано в index.html та стилізовано за допомогою файлу styles.css. Усі елементи керування – вибір алгоритму, країни, типу структури та запуск пошуку – є інтерактивними і не потребують перезавантаження сторінки.

Система протестована на моделях з вручну створеним графом (13 вершин, 16 ребер), а також на структурі з реальними містами Франції (понад 30 маршрутів). Окрім того, проведено тестування роботи алгоритмів на клітинній сітці, обмеженій географічними межами країни. Всі алгоритми продемонстрували стабільну роботу при зміні параметрів, переключенні режимів та повторних запусків без оновлення сторінки.

Інтерфейс розроблено з урахуванням принципів ефективного навчання: активна взаємодія, покрокова анімація, кольорові підказки, контрольований темп, варіативність вхідних даних.

Система має потенціал до подальшого розвитку: додавання серверної частини, інтеграція з базами даних, багатокористувацький режим або публічне розгортання в мережі Інтернет.

Розроблений вебзастосунок може бути корисним усім, хто самостійно вивчає алгоритми, цікавиться візуальним навчанням або хоче експериментально дослідити їхню поведінку в реальному часі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Web-сторінки:

1. Murtagh J., How a Classic Bridge-Crossing Puzzle Inspired New Math [Електронний ресурс] – Режим доступу: https://www.scientificamerican.com/article/how-the-seven-bridges-of koenigsberg-spawned-new-math

2. Семантична мережа [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Семантична_мережа

3. Directed Graph [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Directed_graph

4. Mixed Graph [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Mixed_graph

5. Weighted graph in Computer Science [Електронний ресурс] / T. Friedrich, A. Fanelli, M. Bentert – Режим доступу: https://www.sciencedirect.com/topics/computer-science/weighted-graph

6. Representing graphs [Електронний ресурс] – Режим доступу: https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs

7. Grids and Graphs [Електронний ресурс] – Режим доступу: https://www.redblobgames.com/pathfinding/grids/graphs.html

8. Чмир І. О., Методи та системи штучного інтелекту. Методичний посібник. [Електронне видання] / Одеса, 2023. – 31 с. – Режим доступу: http://eprints.library.odeku.edu.ua/id/eprint/11693

9. Pathfinding Visualizer [Електронний ресурс] – Режим доступу: https://clementmihailescu.github.io/Pathfinding-Visualizer/

10. DSA Dijkstra’s Algorithm [Електронний ресурс] – Режим доступу: https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php

11. Rozhon V., The hidden beauty of the A* algorithm [Електронний ресурс] – Режим доступу: https://vasekrozhon.wordpress.com/2024/09/04/the-hidden-beauty-of-the-a-algorithm

12. Designing Educationally Effective Algorithm Visualizations [Електронний ресурс] / Hansen S., Narayanan N.H., Hegarty M. // Journal of Visual Languages and Computing. – Academic Press, 2002. – Режим доступу: https://www.eng.auburn.edu/~naraynh/jvlc.pdf

13. Visualising data structures and algorithms through animation [Електронний ресурс] – Режим доступу: https://visualgo.net/en

14. IDEF0 [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/IDEF0

15. HTML [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/HTML

16. CSS [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/CSS

17. JavaScript [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/JavaScript

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

19. Turf: Getting started [Електронний ресурс] – Режим доступу: https://turfjs.org/docs/getting-started

20. Stevens E, 7 fundamental user experience (UX) design principles all designers should know [Електронний ресурс] – Режим доступу: https://www.uxdesigninstitute.com/blog/ux-design-principles/

ДОДАТОК А

Опис факторів ефективності навчання із застосуванням візуалізаційних інструментів

Таблиця А.1 – Фактори ефективності навчання із застосуванням візуалізаційних інструментів

Фактор	Характеристика
Активна взаємодія з алгоритмом	Одним із найважливіших факторів, що позитивно впливають на якість засвоєння матеріалу, є можливість студента впливати на хід виконання алгоритму: змінювати параметри, обирати вхідні дані, запускати або зупиняти виконання, переходити між кроками. Такий рівень взаємодії перетворює пасивного спостерігача на активного учасника навчального процесу, що сприяє кращому розумінню причинно-наслідкових зв’язків у логіці роботи алгоритму. На відміну від демонстрацій у форматі відео або статичних схем, інтерактивні застосунки дозволяють студенту експериментувати, досліджувати альтернативні сценарії та краще запам’ятовувати поведінку алгоритму в різних умовах.
Візуалізація абстрактних понять через знайомі образи або метафори	У системах, подібних до HalVis, важливо не лише показувати технічні

Продовження таблиці А.1

Фактор	Характеристика
Візуалізація абстрактних понять через знайомі образи або метафори	кроки алгоритму, а й допомагати студенту осмислити їх через візуальні асоціації або зрозумілі структури. Навіть якщо не використовується безпосереднє моделювання об'єктів з реального світу (наприклад, дорожніх карт чи фізичних систем), сам спосіб представлення інформації – у вигляді графа, сітки, шляху чи блоків – вже може виконувати роль ментальної моделі, яка наближує абстракцію до візуально зрозумілої форми.
Мультимодальний підхід (поєднання тексту, графіки, анімації)	Ефективна візуалізація об'єднує графічне представлення, пояснювальний текст, псевдокод і, за можливості, словесні коментарі. Такий дає змогу студенту сприймати складну інформацію різними способами, що підвищує глибину засвоєння. Наприклад, якщо користувач бачить граф, паралельно читає коротке пояснення та спостерігає за активним рядком псевдокоду, він краще розуміє зв'язок між структурою та логікою алгоритму.

Продовження таблиці А.1

Фактор	Характеристика
Покрокове виконання з можливістю контролю темпу	Повільна візуалізація або затримка між кроками (наприклад, таймери) допомагають студенту зосередитись на послідовності дій. Це дає змогу краще сприйняти, що саме виконує алгоритм, і дозволяє побачити, як змінюється стан графа з часом. Така пауза між подіями виступає аналогом навчального «крок за кроком».
Варіативність вхідних даних для експериментів	Можливість експериментувати з різними вхідними даними можливість обрати тип графа, стартову та цільову вершину, або видалити деякі ребра створює достатню гнучкість для базових експериментів. Це дозволяє користувачу досліджувати поведінку алгоритму в різних умовах і порівнювати, як змінюється маршрут чи швидкість пошуку. Така варіативність підсилює розуміння механіки алгоритмів.

Таблиця А.2 – Аналіз вебзастосунку Red Blob Games відповідно до п’яти освітніх факторів ефективної візуалізації алгоритмів

Фактор	Характеристика
Активна взаємодія з алгоритмом	У Red Blob Games користувач має змогу частково впливати на хід виконання алгоритму. Зокрема, можна змінювати положення стартової та цільової вершини, а також у реальному часі спостерігати, як алгоритм реагує на ці зміни. Проте більш глибокого рівня взаємодії – такого як ручне крокування, зупинка або редагування внутрішніх параметрів – не передбачено. Незважаючи на це, інтерактивність усе ж відіграє важливу роль у процесі навчання: користувач може експериментувати з різними сценаріями, змінюючи вхідні точки, і таким чином досліджувати, як алгоритм змінює траєкторію пошуку. Таким чином, Red Blob Games перетворює навчальний процес на активний досвід, хоча і з дещо обмеженим набором керованих елементів.
Візуалізація абстрактних понять через знайомі образи або метафори	У Red Blob Games використовується проста та добре зрозуміла двовимірна сітка, яка виконує роль універсальної метафори простору для пошуку. Хоча візуалізація не оперує об'єктами з реального світу (наприклад, містами, дорогами чи фізичними об'єктами), сама структура сітки інтуїтивно сприймається як модель маршруту або навігаційного поля. Такий підхід створює ментальну модель, де клітинки сітки уособлюють вершини, а сусідство – зв’язки між ними. Колірне кодування (наприклад, для відкритих,

Продовження таблиці А.2

Фактор	Характеристика
Візуалізація абстрактних понять через знайомі образи або метафори	закритих та оброблених вузлів), стрілки напрямку та візуалізація пріоритетів забезпечують зрозуміле представлення дій алгоритму. Таким чином, хоча система не використовує буквальних аналогій із реальним світом, її візуальний стиль сприяє перекладу абстрактних алгоритмічних понять у наочні форми, що відповідає освітній меті критерію.
Мультимодальний підхід (поєднання тексту, графіки, анімації)	Red Blob Games поєднує кілька каналів подання інформації, що створює мультимодальний освітній досвід. Пояснення роботи алгоритмів супроводжуються графічною візуалізацією, формулами, коментарями в тексті та в деяких випадках – графіками евристичних функцій (наприклад, у випадку A*). Користувач не лише бачить, як змінюється стан сітки, а й читає паралельне пояснення: що таке функція g(n), як обчислюється h(n), чому певні вузли отримали вищий пріоритет. Такий супровід сприяє глибшому розумінню алгоритму, оскільки поєднує візуальне сприйняття з логічною інтерпретацією. Хоча псевдокод у прямому вигляді не виводиться, інші елементи – як текст, анімація та формули – забезпечують достатній рівень мультимодальності, щоб користувач міг опанувати базові принципи роботи алгоритму.

Продовження таблиці А.2

Фактор	Характеристика
Покрокове виконання з можливістю контролю темпу	У Red Blob Games не реалізовано повноцінного крокового режиму, в якому користувач міг би вручну переходити від одного кроку до наступного. Проте система використовує плавну візуалізацію з помітною затримкою між змінами станів сітки. Такий тип подачі забезпечує часову структуру подій, яка дозволяє студенту спостерігати за тим, як алгоритм поступово розширює область пошуку, оновлює значення та змінює колір клітинок. Ця анімаційна пауза виконує функцію навчального уповільнення – користувач встигає обробити інформацію і пов’язати її з логікою алгоритму. Таким чином, хоч система і не надає прямого керування кроками, візуально-часове уповільнення забезпечує базову крокovість, достатню для ефективного сприйняття.
Варіативність вхідних даних для експериментів	У Red Blob Games користувач має обмежені, але достатні можливості для модифікації вхідних даних. Доступна зміна стартової та цільової вершини, а також вибір серед кількох алгоритмів пошуку, які відображаються в межах фіксованої сіткової структури. Проте користувач не має можливості створювати власний граф, змінювати розмір сітки або конфігурацію ребер. Також відсутня функція блокування клітинок чи задання ваг вручну, що обмежує кількість сценаріїв для експериментів. Попри це, навіть базова варіативність – можливість змінити вихідні умови завдання – дозволяє

Продовження таблиці А.2

Фактор	Характеристика
Варіативність вхідних даних для експериментів	досліджувати вплив початкових параметрів на хід виконання алгоритму. У навчальному контексті цього може бути достатньо для засвоєння ключових принципів роботи алгоритмів пошуку.

Таблиця А.3 – Аналіз платформи VisuAlgo відповідно до п’яти освітніх факторів ефективної візуалізації алгоритмів

Фактор	Характеристика
Активна взаємодія з алгоритмом	VisuAlgo забезпечує високий рівень інтерактивності, що дає студенту змогу активно впливати на хід виконання алгоритму. Користувач може не лише обрати тип алгоритму або структури даних, а й крок за кроком переглядати його виконання, ставити на паузу, повертатися назад, або швидко прокручувати до потрібного етапу. Інтерфейс передбачає підсвічування активних елементів, відображення змін у структурі в реальному часі, а також можливість перемикатися між автоматичним і ручним режимами відтворення. Такий підхід повністю відповідає критерію активної взаємодії: користувач не просто спостерігає, а сам керує навчальним процесом, що сприяє глибшому розумінню логіки алгоритму та дає змогу досліджувати його поведінку в деталях.

Продовження таблиці А.3

Фактор	Характеристика
Візуалізація абстрактних понять через знайомі образи або метафори	VisuAlgo використовує графічне зображення структур даних (дерев, графів, черг, стеків тощо), які виконують роль наочних моделей для вивчення алгоритмів. Хоча платформа не намагається відтворити аналогії з реальним світом, вона спирається на візуальну інтуїцію, притаманну схемам і блоковим структурам, що є звичними у навчанні комп'ютерних наук. Наприклад, вузли графа або елементи масиву подаються у вигляді кольорових прямокутників, зв'язки – у вигляді стрілок або дуг, що дозволяє користувачу швидко співвіднести абстрактну логіку з візуальною формою. Таким чином, хоча VisuAlgo не використовує буквальні метафори з побуту чи фізичного світу, її візуальна подача підтримує побудову ментальних моделей, які полегшують сприйняття структури та динаміки алгоритму.
Мультимодальний підхід (поєднання тексту, графіки, анімації)	VisuAlgo вирізняється високим рівнем мультимодальності, поєднуючи графічну візуалізацію, текстові пояснення, псевдокод, а також анотації змінних і поточних операцій. Кожен алгоритм супроводжується інтерактивним зображенням, яке відображає зміну стану структур у реальному часі, а також підсвіченим псевдокодом, що синхронізований з

Продовження таблиці А.3

Фактор	Характеристика
Мультимодальний підхід (поєднання тексту, графіки, анімації)	візуалізацією. Крім того, у деяких модулях користувач може переглядати вміст пам’яті, черг, стеків або масивів у вигляді таблиць чи блоків. Такий підхід дозволяє студенту сприймати одну й ту саму дію алгоритму з кількох точок зору, що суттєво покращує засвоєння матеріалу. Наявність текстових пояснень і структурованої подачі теорії робить VisuAlgo ефективним мультимодальним середовищем для вивчення алгоритмів.
Покрокове виконання з можливістю контролю темпу	VisuAlgo надає користувачеві повний контроль над темпом виконання алгоритму, що повністю відповідає освітньому критерію крокованості. Система дозволяє виконувати алгоритм покроково – вручну переходити від однієї операції до наступної, повертатися назад, ставити на паузу або змінювати швидкість анімації. Водночас інтерфейс зберігає візуальну плавність між кроками, що полегшує сприйняття послідовності дій. Такий підхід не лише допомагає студенту зосередитись на поточному етапі, а й формує зв’язки між візуалізацією та логікою алгоритму. Крокове виконання особливо цінне для складних алгоритмів із рекурсією, множинними умовами чи повтореннями, де автоматичне виконання може ускладнювати розуміння.

Продовження таблиці А.3

Фактор	Характеристика
Варіативність вхідних даних для експериментів	У VisuAlgo можливості змінювати вхідні дані залежать від конкретного алгоритму чи структури, що розглядається. Для деяких алгоритмів, зокрема сортування чи роботи з деревами, користувач може задати власні числові набори або конфігурації, що дозволяє вивчати поведінку алгоритму на різних прикладах. Проте у випадку з алгоритмами на графах або візуалізацією пошуку, структура графа зазвичай фіксована – користувач не має змоги змінювати кількість вершин, ребра чи їхні ваги. Це обмежує можливість експериментів і звужує спектр доступних сценаріїв для навчання. Незважаючи на це, навіть обмежена варіативність дозволяє спостерігати, як змінюється хід алгоритму залежно від вхідних умов. VisuAlgo, як освітній ресурс, робить акцент на поясненні типових алгоритмічних патернів, а не на моделюванні довільних задач.

Таблиця А.4 – Аналіз функціональних можливостей платформи Pathfinding Visualizer відповідно до п’яти освітніх факторів ефективної візуалізації алгоритмів

Фактор	Характеристика
Активна взаємодія з алгоритмом	Pathfinding Visualizer надає користувачеві широкі можливості для безпосередньої взаємодії з алгоритмом ще до його запуску. Інтерфейс дозволяє вручну встановлювати стартову та цільову вершини,

Продовження таблиці А.4

Фактор	Характеристика
Активна взаємодія з алгоритмом	додавати перешкоди на полі (які блокують шлях), а також обирати один із кількох пошукових алгоритмів, серед яких A*, Dijkstra, BFS і DFS. Після запуску користувач може спостерігати за тим, як алгоритм у реальному часі досліджує простір. Хоча покроковий режим або зупинка виконання відсутні, система забезпечує наочну анімацію, що створює ефект поступового розгортання алгоритму. Такий рівень взаємодії відповідає принципу перетворення студента з пасивного спостерігача на активного учасника навчального процесу, даючи змогу експериментувати з конфігураціями графа та одразу бачити наслідки вибраних змін. за кроком переглядати його виконання, ставити на паузу, повертатися назад, або швидко прокручувати до потрібного етапу. Інтерфейс передбачає підсвічування активних елементів, відображення змін у структурі в реальному часі, а також можливість перемикатися між автоматичним і ручним режимами відтворення. Такий підхід повністю відповідає критерію активної взаємодії: користувач не просто спостерігає, а сам керує навчальним процесом, що сприяє глибшому розумінню логіки алгоритму та дає змогу досліджувати його поведінку в деталях.

Продовження таблиці А.4

Фактор	Характеристика
Візуалізація абстрактних понять через знайомі образи або метафори	Pathfinding Visualizer використовує двовимірну сітку як основу для подання простору пошуку, що є інтуїтивно зрозумілим способом візуалізації графа. Кожна клітинка сітки репрезентує вершину, а сусідство між клітинками – потенційні переходи між вершинами. Перешкоди подаються у вигляді затемнених клітинок, що чітко сигналізують про недоступність певних маршрутів, а вибрані стартова й цільова точки мають окреме кольорове позначення. Візуалізація оброблених, активних і неперевірених вершин відбувається за допомогою кольорової індикації, що дозволяє користувачеві візуально відстежити прогрес алгоритму та його логіку. Така форма подання не вимагає складних пояснень і виконує роль наочної метафори алгоритмічного процесу, що перетворює абстрактну логіку на зрозумілу візуальну динаміку.
Мультимодальний підхід (поєднання тексту, графіки, анімації)	Pathfinding Visualizer орієнтований насамперед на візуальну складову навчання, з акцентом на динаміку змін у графі під час виконання алгоритму. Головним джерелом інформації для користувача є анімована сітка, де в реальному часі змінюється стан клітинок: оброблені, відкриті, непрохідні та ті, що входять до знайденого шляху. Водночас у застосунку відсутній вивід псевдокоду

Продовження таблиці А.4

Фактор	Характеристика
Мультимодальний підхід (поєднання тексту, графіки, анімації)	або текстовогокоментаря, що описує логіку поточних дій алгоритму. Це обмежує мультимодальність системи: студент сприймає процес винятково через візуальні зміни. Незважаючи на це, завдяки чіткому колірному кодуванню та простому інтерфейсу, візуалізація залишається інтуїтивною та доступною, особливо для початкового ознайомлення з принципами роботи пошукових алгоритмів.
Покрокове виконання з можливістю контролю темпу	У Pathfinding Visualizer відсутній повноцінний покроковий режим, за якого користувач міг би вручну контролювати кожну дію алгоритму. Однак візуалізація виконується з помітною затримкою між етапами, яка дає змогу спостерігати, як алгоритм поступово розширює зону пошуку, обробляє вершини й формує оптимальний маршрут. Швидкість візуалізації є фіксованою, але її динаміка підібрана таким чином, щоб користувач встигав простежити за змінами й логікою дій. Така анімаційна послідовність виконує роль умовного «крокування» й дозволяє сформувати розуміння послідовності виконання алгоритму, навіть без повного контролю над кожним кроком.
Варіативність вхідних даних для експериментів	Pathfinding Visualizer надає користувачу можливість впливати на початкові умови задачі, що створює основу для базових експериментів.

Продовження таблиці А.4

Фактор	Характеристика
Варіативність вхідних даних для експериментів	Зокрема, можна вручну вибирати стартову та цільову вершини, а також додавати або видаляти перешкоди на сітці, що симулюють непрохідні області. Крім того, користувач має змогу обирати один із кількох алгоритмів для порівняння результатів роботи в різних умовах. Водночас структура графа є фіксованою сіткою, і не передбачено створення довільних графів, зміни ваг ребер чи задання складніших сценаріїв. Попри це, доступний рівень варіативності дозволяє досліджувати вплив змінних на поведінку алгоритмів, а отже – сприяє формуванню інтуїтивного розуміння логіки пошуку шляху

ДОДАТОК Б

Опис реалізації дизайну інтерфейсу та сценаріїв його поведінки

Таблиця Б.1 – Реалізація ключових принципів інтерфейсного дизайну

Назва	Реалізація
Принцип мінімалізму та фокусування на функціональності;	Принцип мінімалізму та фокусування на функціональності виражається в тому, що інтерфейс вебзастосунку спроєктовано за принципом максимальної простоти, що дозволяє користувачу зосередитися на основній меті – вивченні роботи пошукових алгоритмів. Відсутність зайвих візуальних елементів зменшує когнітивне навантаження і підвищує концентрацію на динаміці візуалізації. Такий підхід відповідає практикам проєктування освітніх інструментів, де інтерфейс виконує допоміжну, а не домінуючу роль.
Використання кольорових метафор для покрокового слідкування	Використання кольорових метафор для покрокового слідкування реалізується через візуальну трансформацію елементів під час виконання алгоритму: клітинки або вузли змінюють колір залежно від свого статусу – неактивні, переглянуті, включені до шляху тощо. Така динамічна візуалізація допомагає користувачу інтуїтивно відстежувати логіку алгоритму в реальному часі, не вдаючись до технічних деталей.

Продовження таблиці Б.1

Назва	Реалізація
Принцип "контроль без перевантаження"	Забезпечення контролю без перевантаження досягається через обмежену, але достатню кількість доступних в інтерфейсі налаштувань . Хоча користувач може вибору алгоритму, початкової та цільової вершини, інші параметри залишаються за замовчуванням. Це дозволяє зберегти баланс між гнучкістю і простотою – початківцям не потрібно глибоко занурюватися в налаштування, щоб побачити базову логіку алгоритму.
Підтримка сценаріїв поступового ознайомлення	Підтримка сценаріїв поступового ознайомлення реалізується через гнучку структуру, яка дозволяє навчатися поетапно, без необхідності змінювати середовище чи інтерфейс. Інтерфейс дозволяє переходити від базових алгоритмів (наприклад, BFS і DFS) до складніших (Dijkstra, A*) у межах однієї структури. Це реалізує освітній принцип поступового занурення та відповідає сценаріям типового користувача – від новачка до більш досвідченого студента.
Відображення статусу алгоритму в реальному часі	Відображення статусу алгоритму в реальному часі забезпечує прозорість і чіткість зворотного зв’язку під час виконання пошуку. Користувач постійно бачить, на якому етапі знаходиться виконання алгоритму: чи йде пошук, чи вже знайдено шлях, чи відбулося завершення без результату. Це реалізовано через зміну кольору вузлів та контрольних елементів, що дозволяє уникнути

Продовження таблиці Б.1

Назва	Реалізація
Відображення статусу алгоритму в реальному часі	непорозумінь та забезпечити прозорість виконання. Така динамічна зворотна інформація важлива для формування причинно-наслідкових зв'язків у навчальному процесі.
Відсутність блокуючих дій та гнучкість взаємодії.	Відсутність блокуючих дій та гнучкість взаємодії виражаються у свободі маніпуляцій без необхідності перезапуску середовища. Інтерфейс не обмежує користувача в повторному запуску, зміні параметрів або перемиканні між алгоритмами без потреби перезавантажувати сторінку. Це забезпечує швидкий зворотний цикл експериментування – важливий елемент інтуїтивного вивчення, коли студент пробує різні варіанти поведінки системи, навчаючись на відмінностях у результатах.

Таблиця Б.2 – Опис поведінки інтерфейсу в межах основних сценаріїв взаємодії

Назва сценарію	Поведінта інтерфейсу
Користувач спостерігає за виконанням алгоритму	Щоб забезпечити високу інформативність, реалізовано кольорові маркери для різних статусів вузлів (наприклад, стартова точка, ціль, відвідані вершини, шлях). Це дозволяє швидко зорієнтуватися в процесі виконання, зменшуючи когнітивне навантаження. Згідно з UX-принципом видимості стану, користувач завжди має чітке уявлення про поточний стан системи.

Продовження таблиці Б.2

Назва сценарію	Поведінта інтерфейсу
Користувач спостерігає за виконанням алгоритму	Щоб забезпечити високу інформативність, реалізовано кольорові маркери для різних статусів вузлів (наприклад, стартова точка, ціль, відвідані вершини, шлях). Це дозволяє швидко зорієнтуватися в процесі виконання, зменшуючи когнітивне навантаження. Згідно з UX-принципом видимості стану, користувач завжди має чітке уявлення про поточний стан системи.
Користувач досліджує логіку алгоритму	Покрокове оновлення сітки дозволяє студенту спостерігати, як саме алгоритм приймає рішення. Це формує причинно-наслідкові зв’язки між діями алгоритму й результатом, що узгоджується з принципом підтримки ментальної моделі користувача.
Користувач має обмежений час використання вебзастосунку	Інтерфейс створено таким чином, щоб ключові елементи управління були легко доступними, з відповідними розмірами кнопок, інтуїтивним розташуванням та зрозумілими іконками.

ДОДАТОК В

Опис структурних елементів діаграм IDEF0

Таблиця В.1 – Опис структурних елементів IDEF0-моделі для блоку А0

Назва (місце)	Характеристика
Вхід (входить ліворуч)	Користувач – це зовнішній агент, який ініціює взаємодію із системою. Саме він задає вхідні параметри (вибір алгоритму, стартової та цільової вершини) та отримує результат у вигляді візуалізації. У моделі він не є частиною механізмів, але відіграє ключову роль як джерело введення і як суб’єкт, що інтерпретує вихід.
	Вхідні параметри (початкова вершина, кінцева вершина, тип графа, алгоритм). Визначає, які саме дані обробляє система. Входи є джерелом параметрів для візуалізації.
Контроль (входить згори):	Правила анімації (затримка між кроками, порядок обходу). Визначають логіку візуалізації: з якою швидкістю показуються зміни, в якому порядку відображаються вузли.
	Логіка реалізованих алгоритмів (BFS, DFS, Dijkstra, A*). Набір правил, за якими повинні працювати пошукові алгоритми. Це не вводиться користувачем вручну, а вже вбудовано в систему.
	Обмеження UI/UX (що дозволено змінювати, як виглядає результат). Визначають, які елементи доступні для взаємодії, як подається інформація, що блокується в інтерфейсі.

Продовження таблиці В.1

Назва (місце)	Характеристика
Контроль (входить згори):	Освітні цілі проєкту (інтерактивність, покроковість, наочність). Загальна ідеологія, що формує вимоги до реалізації. Це визначає необхідність зробити систему такою, щоб вона сприяла розумінню.
Механізм (входить знизу):	Браузер користувача. Браузер виступає середовищем виконання застосунку. Саме в ньому відображається інтерфейс, обробляються події та запускаються алгоритми.
	JavaScript та JavaScript-фреймворки (Leaflet.js, Turf.js). JavaScript забезпечує динамічну взаємодію з інтерфейсом та запуск алгоритмів. Leaflet.js використовується для відображення карти, маркерів міст і ліній між вершинами. Turf.js виконує просторові розрахунки, зокрема перевірку належності клітин до меж країни та обчислення відстаней. Ці інструменти дозволяють реалізувати візуалізацію пошукових алгоритмів без використання сторонніх API.
	HTML/CSS інтерфейс. HTML і CSS формують структуру та зовнішній вигляд вебзастосунку. Вони відповідають за розташування елементів управління, кольори, шрифти та адаптивність інтерфейсу. Саме через ці компоненти користувач взаємодіє із системою, налаштовуючи параметри й запускаючи алгоритми.

Продовження таблиці В.1

Назва (місце)	Характеристика
Механізм (входить знизу):	Користувацький інтерфейс – це випадаючі списки вибору алгоритму, стартової та цільової вершини, а також кнопки запуску пошуку. Ці елементи надають користувачеві можливість задати вхідні параметри та ініціювати виконання алгоритму, забезпечуючи основний рівень взаємодії між людиною і системою.
Вихід (виходить праворуч):	Анімована візуалізація роботи алгоритму. Цей вихід представляє динамічне зображення процесу виконання обраного алгоритму пошуку. Анімація показує послідовність дій: вибір вершин, оновлення шляху, завершення пошуку. Візуальне представлення допомагає користувачу краще зрозуміти логіку алгоритму, його кроки та вплив параметрів.
	Осмислений досвід користувача (навчання). Неформальний, але ключовий результат – користувач розуміє, як працює алгоритм через взаємодію.

Таблиця В.2 – Опис структурних елементів IDEF0-моделі для блоку A1–A4

Назва (тип)	Характеристика
Користувач (вхід)	Ініціює запуск програми, відкриваючи сторінку у браузері.
Запуск вебзастосунку (процес)	Цей процес відповідає за початкове завантаження інтерфейсу у браузері користувача. Завантажуються HTML-структура, стилі CSS та JavaScript-

Продовження таблиці В.2

Назва (тип)	Характеристика
Запуск вебзастосунку (процес)	фреймворки, що забезпечують доступ до основних елементів управління та карти.
Браузер користувача (механізм)	Є середовищем виконання застосунку: інтерпретує HTML, CSS і JS-код.
Вхідні параметри (вихід та вхід)	Це конкретні дані, які користувач передає через UI (вибором точок і алгоритму).
Користувач з доступом до користувацького інтерфейсу (вихід та механізм)	Вказує, що користувач взаємодіє з доступними елементами користувацького інтерфейсу.
Передача даних через користувацький інтерфейс (процес)	Користувач взаємодіє з UI (панеллю вибору, кнопками запуску), задаючи параметри: алгоритм, стартову та цільову вершини. Ці дії формують основу для подальшої роботи алгоритмів і визначають умови виконання.
Обмеження UI/UX (контроль)	Визначають правила взаємодії: які кнопки активні, які дії дозволені.
Параметри виконання алгоритмів (вихід та вхід)	Це набір вхідних значень, що формуються внаслідок дій користувача в інтерфейсі – зокрема, вибір типу алгоритму, стартової та цільової вершини. Ці параметри передаються до програмного модуля, де використовуються для запуску відповідної логіки пошуку шляху. Вони визначають контекст, у якому працює алгоритм, і безпосередньо впливають на його хід та результат.

Продовження таблиці В.2

Назва (тип)	Характеристика
Дані, прописані в програмному коді (вхід)	Це заздалегідь реалізовані в програмному коді обмеження, умови та логіка взаємодії, які визначають порядок виконання процесів.
Спрацювання програмного коду (процес)	Цей процес активує реалізовану логіку обраного алгоритму. Алгоритмічний модуль обробляє введені параметри, запускає обчислення та формує послідовність дій для побудови шляху.
Алгоритми, які прописані в програмному коді (механізм)	Обраний користувачем алгоритм (BFS, A*, тощо) виконується згідно з наперед визначеною реалізацією.
Параметри відображення алгоритмів (вихід та вхід)	Це набір налаштувань, які визначають візуальну поведінку анімації алгоритму. До них можуть належати швидкість виконання, затримка між кроками, стиль відображення ребер, колірне кодування відвіданих вузлів тощо. Ці параметри задаються через користувацький інтерфейс і передаються до модуля візуалізації, де впливають на спосіб представлення кожного етапу алгоритму, забезпечуючи зрозуміле та послідовне графічне зображення процесу пошуку.
Демонстрація роботи алгоритмів (процес)	У цьому етапі здійснюється візуальне представлення виконання алгоритму: анімація проходження по вершинах, побудова шляху, підсвічування. Це забезпечує користувачу можливість зрозуміти логіку алгоритму через спостереження.

Продовження таблиці В.2

Назва (тип)	Характеристика
Програмний код для відображення роботи алгоритмів (механізм)	Забезпечує візуальну анімацію, зміну стану клітинок/вершин, затримку кроків тощо.
Анімована візуалізація роботи алгоритму (вихід)	Покрокове графічне відображення ходу пошуку на карті або сітці.
Осмислення користувачем процесу	Основна мета застосунку – дати користувачеві можливість зрозуміти логіку алгоритму через спостереження.

Таблиця В.3 – Опис структурних елементів IDEF0-моделі для блоку A5–A9

Назва (тип)	Характеристика
Розробник (вхід)	Розробник: Ініціатор розробки: створює інтерфейс, алгоритми, структури графа.
Створення користувацького інтерфейсу (процес)	Цей процес включає розробку HTML/CSS-структури та JavaScript-компонентів, які забезпечують взаємодію користувача з системою. Створюється візуальний інтерфейс, панель вибору алгоритму, карта або сітка для відображення результатів.
Створення пошукових алгоритмів (процес)	У цьому процесі реалізується логіка роботи алгоритмів пошуку (BFS, DFS, A*, Dijkstra) у вигляді функцій JavaScript. Ці алгоритми готуються до подальшого запуску, враховуючи можливість зовнішнього керування через UI.

Продовження таблиці В.3

Назва (тип)	Характеристика
Створення графових та сіткових структур в програмному коді (процес)	Цей етап відповідає за створення моделей графа або сітки, які виступають як базова структура для запуску алгоритмів. Включає генерацію вершин, ребер, сітки, обробку кордонів і ваг.
Вхідні параметри (вхід)	Параметри, які вводить користувач (алгоритм, точки, налаштування).
Користувач з доступом до інтерфейсу (вхід)	Користувач взаємодіє з інтерфейсом для налаштування пошуку.
Структури для виконання алгоритмів (вихід та вхід)	Сітка або граф передаються як вхідна структура для пошуку.
Пошукові алгоритми (вихід та вхід)	Готові алгоритми передаються як логіка, яку буде запущено.
Створений інтерфейс (вихід та механізм)	Результат попереднього етапу проєктування, що включає HTML-структуру, стилі CSS та елементи управління. Він забезпечує користувачеві можливість передавати параметри до системи через графічні елементи інтерфейсу.
Параметри виконання алгоритмів (вихід та вхід)	Передаються як змінні, що змінюють поведінку алгоритму.

Продовження таблиці В.3

Назва (тип)	Характеристика
Логіка комбінування (механізм)	Внутрішнє програмне правило. Поєднання графової структури, вибраного алгоритму та заданих користувачем параметрів.
Передача даних через користувацький інтерфейс (процес)	Користувач, взаємодіючи з інтерфейсом, обирає параметри: алгоритм, стартову і цільову вершини. Ці дані передаються у внутрішні функції та ініціюють виконання обчислень.
Виконання алгоритмів на основі комбінації прописаних в коді та заданих через користувацький інтерфейс параметрів (процес)	Фінальний процес, у якому запускаються алгоритми згідно з обраними параметрами. Внутрішня логіка поєднує вже закладені в код алгоритми зі змінними, які передає користувач, та виконує їх відповідно до контексту.
Параметри відображення алгоритмів (вихід)	Результати (наприклад: шлях, відвідані вершини) передаються у модуль візуалізації, який покаже їх у зрозумілій формі.

Таблиця В.4 – Опис структурних елементів IDEF0-моделі для блоку A11–A12

Назва (тип)	Характеристика
Параметри відображення даних (вхід)	Налаштування параметрів анімації, задані через інтерфейс або код.

Продовження таблиці В.4

Назва (тип)	Характеристика
Поступовий вивід даних (процес)	Цей блок відповідає за поетапну анімацію алгоритму: зміна кольору клітинок, показ процесу обходу, оновлення графа або сітки в режимі реального часу.
Дані про фінальний маршрут (вихід та вхід)	Після завершення алгоритму формується готовий маршрут – список клітинок або вершин, які ведуть від старту до цілі. Він готується до фінального виводу на карту або сітку.
Зрозуміле візуальне оформлення (контроль)	Визначає, як саме має виглядати результат: які кольори, стиль, формат відображення для сприйняття користувачем.
Прописані алгоритми візуалізації (механізм)	Це заздалегідь реалізовані у коді інструкції, як саме виводити кожен крок алгоритму на екран.
Вивід фінального маршруту (процес)	Цей процес підсвічує фінальний шлях на екрані, зазвичай іншим кольором, і закріплює результат пошуку для візуального сприйняття користувачем.
Осмислення користувачем процесу (вихід)	Завершення візуалізації дозволяє користувачу оцінити поведінку алгоритму і зрозуміти логіку його роботи – це і є ключовий навчальний результат.

ДОДАТОК Г

Фрагменти реалізації основних модулів

```
async function switchCountry(code) {
  let cities, rawGraph, geojsonFile;

  if (code === 'FR') {
    cities = (await import('./city_coords.js')).FR_cities;
    rawGraph                          =                          (await
import('./Graph/fr_rawGraph.js')).FR_rawGraph;
    geojsonFile = 'france.geojson';
  } else if (code === 'UA') {
    cities = (await import('./ua_city_coords.js')).default;
    rawGraph                          =                          (await
import('./Graph/ua_rawGraph.js')).UA_rawGraph;
    geojsonFile = 'ukraine.geojson';
  } else if (code === 'PL') {
    cities = (await import('./pl_city_coords.js')).default;
    rawGraph                          =                          (await
import('./Graph/pl_rawGraph.js')).PL_rawGraph;
    geojsonFile = 'poland.geojson';
  }

  window.currentCities = cities;
  window.currentGraph = rawGraph;
  window.builder = buildGraphWithDistance(rawGraph, cities);

  // Очищення попередніх шарів
  if (window.cityLayerGroup) window.cityLayerGroup.clearLayers();
  else window.cityLayerGroup = L.layerGroup().addTo(window.map);

  if                                      (window.graphLayerGroup)
window.graphLayerGroup.clearLayers();
  else window.graphLayerGroup = L.layerGroup().addTo(window.map);

  if (window.countryBoundaryLayer) {
    window.map.removeLayer(window.countryBoundaryLayer);
    window.countryBoundaryLayer = null;
  }

  if (window.gridLayerGroup) {
    window.gridLayerGroup.clearLayers();
  }

  window.grid = [];
  window.startCell = null;
  window.endCell = null;
```

Лістинг Г.1 – Реалізація функції перемикання країни switchCountry()

```
  // Додавання міст
  cities.forEach((coords, name) => {
    const icon = L.divIcon({
      className: 'city-marker',
      html: `<div class="custom-marker"></div>`,
      iconSize: [20, 20],
      iconAnchor: [10, 10]
    });
    L.marker(coords, { icon })
      .bindPopup(name)
      .addTo(window.cityLayerGroup);
  });

  if (window.currentDisplayMode === 'graph') {
    drawGraph(cities, rawGraph);
  }

  try {
    const response = await fetch(geojsonFile);
    const geojson = await response.json();
    window.countryBoundaryLayer = L.geoJSON(geojson, {
      style: {
        color: 'green',
        weight: 2,
        fillColor: 'blue',
        fillOpacity: 0.05
      }
    }).addTo(window.map);
  } catch (err) {
    console.error("Не вдалося завантажити межі:", geojsonFile,
err);
  }
}
```

Лістинг Г.1, аркуш 2

```
     document.getElementById('findPathBtn').addEventListener('cl
ick', () => {
       const start = document.getElementById('startCity').value;
       const end = document.getElementById('endCity').value;
       const algorithm =
document.getElementById('algorithmSelect').value;
       if (algorithm === 'dijkstra') {
         const path = runDijkstra(start, end, window.builder,
window.map);
       }
```

Лістинг Г.2 – Реалізація обробника натискання кнопки «Знайти шлях»

```
      else if (algorithm === 'bfs') {
        if (!startCell || !endCell) {
          alert("Виберіть старт і фініш на мапі!");
          return;
}
        runBFS([startCell.row,  startCell.col],  [endCell.row,
endCell.col], window.map);
      }
      else if (algorithm === 'dfs') {
        const { start, end } = findStartAndEndCells(grid);
        if (!start || !end) {
          alert("Виберіть старт і фініш на мапі!");
          return;
        }
        runDFS([startCell.row,  startCell.col],  [endCell.row,
endCell.col], window.map);
      }
      else if (algorithm === 'dijkstraGrid') {
        runDijkstraGrid([startCell.row,         startCell.col],
[endCell.row, endCell.col], window.map);
      }
      else if (algorithm === 'astarGrid') {
        runAStarGrid([startCell.row,            startCell.col],
[endCell.row, endCell.col], window.map);
      }
      else if (algorithm === 'astar') {
        runAStarGraph(start, end, window.builder, window.map);
      }
    });
```

Лістинг Г.2, аркуш 2

```
    function updateAlgorithmOptions(mode) {
      const select = document.getElementById('algorithmSelect');
      if (!select) return;

      select.innerHTML = ''; // Очистити старі

      const  list  =  mode  ===  'graph'  ?  graphAlgorithms  :
gridAlgorithms;

      list.forEach(algo => {
        const option = document.createElement('option');
        option.value = algo.value;
        option.textContent = algo.label;
        select.appendChild(option);
      });
    }
```

Лістинг Г.3 – Функція динамічного оновлення переліку алгоритмів

```
function drawGraph(cities, graphData) {
  if (!graphData || !cities) return;
  graphData.forEach((neighbors, city) => {
    const fromCoords = cities.get(city);
    neighbors.forEach(neighbor => {
      const toCoords = cities.get(neighbor);

      if (fromCoords && toCoords) {
        const line = L.polyline([fromCoords, toCoords], {
          color: 'blue',
          weight: 2,
          opacity: 0.7
        }).addTo(window.graphLayerGroup);

        L.polylineDecorator(line, {
          patterns: [
            {
              offset: '95%',
              repeat: 0,
              symbol: L.Symbol.arrowHead({
                pixelSize: 10,
                pathOptions: { fill: true, color: 'blue',
weight: 1 }
              })
            }
          ]
        }).addTo(window.graphLayerGroup);
      }
    });
  });
}
```

Лістинг Г.4 – Функція візуалізації графа на мапі Leaflet

```
function populateCitySelect(selectId, citiesMap) {
  const select = document.getElementById(selectId);
  if (!select) return;
  select.innerHTML = ''; // Очистка
  citiesMap.forEach((coords, name) => {
    const option = document.createElement('option');
    option.value = name;
    option.textContent = name;
    select.appendChild(option);
  });
}
```

Лістинг Г.5 – Функція заповнення випадаючого списку міст на основі Map-структури

```
export function buildGraphWithDistance(rawGraph, cityCoords)
{
  const neighborsMap = new Map();

  for (const [city, neighbors] of rawGraph) {
    if (!neighborsMap.has(city)) {
      neighborsMap.set(city, []);
    }

    for (const neighbor of neighbors) {
      const cityCoord = cityCoords.get(city);
      const neighborCoord = cityCoords.get(neighbor);

      if (!cityCoord || !neighborCoord) {
        console.warn(`Нет координат для: ${city} или
${neighbor}`);
        continue;
      }

      const distance = haversineDistance(cityCoord,
neighborCoord);
      neighborsMap.get(city).push({ name: neighbor, distance
});
    }
  }

  return neighborsMap;
}


const UA_rawGraph = new Map([
  ['Kyiv', ['Chernihiv', 'Zhytomyr', 'Dnipro']],
  ['Lviv', ['Ternopil', 'Ivano-Frankivsk']],
]);



const UA_cities = new Map([
  ['Kyiv', [50.4501, 30.5234]],
  ['Chernihiv', [51.4982, 31.2893]],
  ...
]);

export function haversineDistance(coord1, coord2) {
  const R = 6371; // Радіус Землі в км
  const toRad = deg => deg * Math.PI / 180;

  const [lat1, lon1] = coord1;
  const [lat2, lon2] = coord2;
```

Лістинг Г.6 – Функція перетворення графа на зважений

```
    const dLat = toRad(lat2 - lat1);
    const dLon = toRad(lon2 - lon1);

    const a = Math.sin(dLat/2)**2 +
              Math.cos(toRad(lat1)) * Math.cos(toRad(lat2)) *
              Math.sin(dLon/2)**2;
    const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

    return R * c;
  }
```

Лістинг Г.6, аркуш 2

```
  export function generateGrid(latStep = 0.1, lngStep = 0.1466)
{

    window.grid = [];
    window.startCell = null;
    window.endCell = null;

    const bounds = map.getBounds();
    grid = [];

    if (!window.gridLayerGroup) {
      window.gridLayerGroup = L.layerGroup();
    }

    if (!window.map.hasLayer(window.gridLayerGroup)) {
      window.map.addLayer(window.gridLayerGroup);
    } else {
       window.gridLayerGroup.clearLayers(); // очистить старую
сетку
    }


    const minLat = Math.floor(bounds.getSouth());
    const maxLat = Math.ceil(bounds.getNorth());
    const minLng = Math.floor(bounds.getWest());
    const maxLng = Math.ceil(bounds.getEast());


    const numRows = Math.floor((maxLat - minLat) / latStep);
    const numCols = Math.floor((maxLng - minLng) / lngStep);

    for (let row = 0; row < numRows; row++) {
      const lat = minLat + row * latStep;
      const gridRow = [];
```

Лістинг Г.7 – Функція побудови сіткового графа

```
          for (let col = 0; col < numCols; col++) {
            const lng = minLng + col * lngStep;
            const center = [lng + lngStep / 2, lat + latStep / 2];

            if (window.countryBoundaryGeoJSON &&
              turf.booleanPointInPolygon(
                turf.point(center),
                turf.polygon(window.countryBoundaryGeoJSON.coordin
ates))) {
              const rectangle = L.rectangle(
                [
                  [lat, lng],
                  [lat + latStep, lng + lngStep]
                ],
                {
                  color: "#888",
                  weight: 1,
                  fillOpacity: 0.05,
                  fillColor: "#ffffff",
                }
              ).addTo(window.gridLayerGroup);

              const cell = {
                row,
                col,
                latlng: center,
                isWall: false,
                rectangle,
              };

              rectangle.on('click', () => {
                if (cell === startCell) {
                  startCell = null;
                  rectangle.setStyle({
                    fillColor: '#ffffff',
                    fillOpacity: 0.05,
                  });

                   const startInput = document.getElementById('grid-
start-coord');
                  startInput.value = '';
                  return;
                }

                if (cell === endCell) {
                  endCell = null;
                  rectangle.setStyle({
                    fillColor: '#ffffff',
                    fillOpacity: 0.05,
                  });
```

Лістинг Г.7, аркуш 2

```
                const endInput = document.getElementById('grid-
end-coord');
                endInput.value = '';
                return;
              }

              if (!startCell) {
                startCell = cell;
                      rectangle.setStyle({  fillColor:  'green',
fillOpacity: 0.6 });

                const startInput = document.getElementById('grid-
start-coord');
                startInput.value = `${cell.latlng[0].toFixed(5)},
${cell.latlng[1].toFixed(5)}`;
                return;
              }

              if (!endCell) {
                endCell = cell;
                       rectangle.setStyle({  fillColor:  'blue',
fillOpacity: 0.6 });

                const endInput = document.getElementById('grid-
end-coord');
                endInput.value = `${cell.latlng[0].toFixed(5)},
${cell.latlng[1].toFixed(5)}`;
                return;
              }

            });

            gridRow.push(cell);
          } else {
            gridRow.push(null);
          }
        }

        grid.push(gridRow);
      }

      console.log("Прямокутна сітка створена:", grid);
    }
```

Лістинг Г.7, аркуш 3

```
    export async function runDijkstra(start, end, graph, map) {
      const distances = new Map();
      const previous = new Map();
```

Лістинг Г.8 – Реалізація алгоритма Дейкстри для графового режиму

```
  const visited = new Set();

  graph.forEach((neighbors, city) => {
    if (!distances.has(city)) {
      distances.set(city, Infinity);
      previous.set(city, null);
    }

    for (const neighbor of neighbors) {
      if (!distances.has(neighbor.name)) {
        distances.set(neighbor.name, Infinity);
        previous.set(neighbor.name, null);
      }
    }
  });

  distances.set(start, 0);

  while (visited.size < distances.size) {
    let currentCity = null;
    let minDistance = Infinity;

    distances.forEach((dist, city) => {
      if (!visited.has(city) && dist < minDistance) {
        minDistance = dist;
        currentCity = city;
      }
    });

    if (currentCity === null) {
      break;
    }
    if (currentCity === end) {
      break;
    }

    visited.add(currentCity);
    const neighbors = graph.get(currentCity);
    if (!neighbors) continue;

    for (const neighbor of neighbors) {
      const currentDist = distances.get(currentCity);
      const alt = currentDist + neighbor.distance;
      const knownDist = distances.get(neighbor.name);

      console.log(`  ➤ dist[${currentCity}] = ${currentDist},
distance = ${neighbor.distance}`);
      console.log(`      ➤      dist[${neighbor.name}]      =
${knownDist}, alt = ${alt}`);
```

Лістинг Г.8, аркуш 2

```
                                    const fromCoords =
window.currentCities.get(currentCity);
                                     const toCoords =
window.currentCities.get(neighbor.name);

          if (!fromCoords || !toCoords) {
            console.warn(`Нема координат для: ${currentCity} або
${neighbor.name}`);
            continue;
          }

          L.polyline([fromCoords, toCoords], {
            color: '#ffc107',
            weight: 2,
            opacity: 0.5,
            dashArray: '5, 5'
          }).addTo(map);
          await new Promise(r => setTimeout(r, 20));

          if (alt < knownDist) {
            distances.set(neighbor.name, alt);
            previous.set(neighbor.name, currentCity);
          }
        }
      }

      const path = [];
      let current = end;
      while (current) {
        path.unshift(current);
        current = previous.get(current);
      }

      if (path.length === 0 || path[0] !== start) {
        alert('Шлях не знайдено');
        return;
      }

      for (let i = 0; i < path.length - 1; i++) {
        const from = window.currentCities.get(path[i]);
        const to = window.currentCities.get(path[i + 1]);

        if (!from || !to) {
          console.warn(`Пропущено відрисовку ребра: ${path[i]} →
${path[i + 1]}`);
          continue;
        }
        L.polyline([from, to], {
```

Лістинг Г.8, аркуш 3

```
        color: 'red',
        weight: 4,
        opacity: 1
      }).addTo(map);

      await new Promise(r => setTimeout(r, 300));
    }

    return path;
  }
```

Лістинг Г.8, аркуш 4

```
  import { haversineDistance } from '../Graph/distance.js';

  export async function runAStarGraph(start, end, graph, map)
{
    if (!(graph instanceof Map)) {
      alert("Невалідний граф");
      return;
    }

    const openSet = [start];
    const cameFrom = new Map();
    const gScore = new Map();
    const fScore = new Map();
    const visited = new Set();

    graph.forEach((neighbors, city) => {
      gScore.set(city, Infinity);
      fScore.set(city, Infinity);
      for (const neighbor of neighbors) {
        if (!gScore.has(neighbor.name)) {
          gScore.set(neighbor.name, Infinity);
          fScore.set(neighbor.name, Infinity);
        }
      }
    });



    gScore.set(start, 0);
    fScore.set(start, haversineDistance(
      window.currentCities.get(start),
      window.currentCities.get(end)
    ));
      visited.add(current);
```

Лістинг Г.9 – Реалізація алгоритма A* для графового режиму

```
      while (openSet.length > 0) {
          openSet.sort((a, b) => fScore.get(a) - fScore.get(b));
           const current = openSet.shift();

      if (current === end) {
           const path = [];
           let node = current;
           while (cameFrom.has(node)) {
             path.unshift(node);
             node = cameFrom.get(node);
           }
           path.unshift(start);
           await drawPath(path, map);
           return path;
         }

         const neighbors = graph.get(current) || [];
         for (const { name: neighbor, distance } of neighbors) {
           if (visited.has(neighbor)) continue;

           const fromCoords = window.currentCities.get(current);
           const toCoords = window.currentCities.get(neighbor);
           if (!fromCoords || !toCoords) {
             console.warn(`Пропуск ребра: ${current} → ${neighbor}
(немає координат)`);
             continue;
           }
           L.polyline([fromCoords, toCoords], {
             color: '#ffc107',
             weight: 2,
             opacity: 0.5,
             dashArray: '5, 5'
           }).addTo(map);
           await new Promise(r => setTimeout(r, 20));

           const tentativeG = gScore.get(current) + distance;

           if (tentativeG < gScore.get(neighbor)) {
             cameFrom.set(neighbor, current);
             gScore.set(neighbor, tentativeG);
             fScore.set(neighbor,
               tentativeG + haversineDistance(
                 window.currentCities.get(neighbor),
                 window.currentCities.get(end)
               )
             );
             if (!openSet.includes(neighbor)) {
               openSet.push(neighbor);
             }
           }
```

Лістинг Г.9, аркуш 2

```
        }
      }

      alert("A*: шлях не знайдено");
    }

    async function drawPath(path, map) {
      for (let i = 0; i < path.length - 1; i++) {
        const from = window.currentCities.get(path[i]);
        const to = window.currentCities.get(path[i + 1]);

        if (!from || !to) {
          console.warn(`drawPath: немає координат для ${path[i]}
або ${path[i + 1]}`);
          continue;
        }

        L.polyline([from, to], {
          color: 'red',
          weight: 4,
          opacity: 1
        }).addTo(map);

        await new Promise(r => setTimeout(r, 300));
      }
    }
```

Лістинг Г.9, аркуш 3

```
    import { grid } from '../grid.js';

    export async function runBFS(start, end, map) {
      console.log("Start cell:", grid[start[0]]?.[start[1]]);
      console.log("End cell:", grid[end[0]]?.[end[1]]);
      const queue = [];
      const visited = new Set();
      const cameFrom = new Map();
      queue.push(start);
      visited.add(key(start));

      while (queue.length > 0) {
        const current = queue.shift();
        const [i, j] = current;

        const cell = grid[i]?.[j];
        if (!cell || cell.isWall) continue;

        cell.rectangle.setStyle({
```

Лістинг Г.10 – Реалізація алгоритма пошуку в ширину для сіткового режиму

```
            fillColor: 'orange',
            fillOpacity: 1,
            color: 'orange'
          });
          await delay(20);

          if (i === end[0] && j === end[1]) break;

           for (const [di, dj] of [[0, 1], [1, 0], [0, -1], [-1,
0]]) {
            const ni = i + di;
            const nj = j + dj;
            const neighbor = grid[ni]?.[nj];
            const neighborKey = key([ni, nj]);

                    if    (neighbor    &&    !neighbor.isWall    &&
!visited.has(neighborKey)) {
              queue.push([ni, nj]);
              visited.add(neighborKey);
              cameFrom.set(neighborKey, [i, j]);
            }
          }
        }

        reconstructPath(cameFrom, start, end);
      }

      function key([i, j]) {
        return `${i},${j}`;
      }

      function delay(ms) {
        return new Promise(r => setTimeout(r, ms));
      }

      function reconstructPath(cameFrom, start, end) {
        let current = end;

        if (!cameFrom.has(key(end))) {
          console.warn("Шлях до кінцевої точки не знайдено.");
          return;
        }

        while (key(current) !== key(start)) {
          const [i, j] = current;
          const cell = grid[i][j];

          if (cell) {
            cell.rectangle.setStyle({ fillColor: 'lime' });
          }
```

Лістинг Г.10, аркуш 2

```
        current = cameFrom.get(key(current));
        if (!current) break;
      }
    }
```

Лістинг Г.10, аркуш 3

```
    export async function runDijkstraGrid(startPos, endPos, map)
{
      const [startRow, startCol] = startPos;
      const [endRow, endCol] = endPos;

      const openSet = [];
      const dist = {};
      const prev = {};
      const visited = new Set();

      const key = (row, col) => `${row}_${col}`;
      const getCell = (row, col) => grid[row]?.[col];

      dist[key(startRow, startCol)] = 0;
      openSet.push({ row: startRow, col: startCol, dist: 0 });

      while (openSet.length > 0) {
        openSet.sort((a, b) => a.dist - b.dist);
        const current = openSet.shift();
        const cKey = key(current.row, current.col);

        if (visited.has(cKey)) continue;
        visited.add(cKey);

        const cell = getCell(current.row, current.col);
        if (!cell || cell.isWall) continue;

        if (cell !== startCell && cell !== endCell) {
              cell.rectangle.setStyle({  fillColor:  '#ffc107',
fillOpacity: 1 });
          await new Promise(r => setTimeout(r, 10));
        }

        if (current.row === endRow && current.col === endCol) {
          let path = [];
          let curKey = cKey;
          while (curKey in prev) {
            const [r, c] = curKey.split('_').map(Number);
            path.unshift(getCell(r, c));
            curKey = prev[curKey];
          }
```

Лістинг Г.11 – Реалізація алгоритма Дейкстри для сіткового режиму

```
            for (const c of path) {
              if (c !== startCell && c !== endCell) {
                      c.rectangle.setStyle({  fillColor:  'blue',
fillOpacity: 1 });
                await new Promise(r => setTimeout(r, 20));
              }
            }
            return;
          }

          const deltas = [[1, 0], [-1, 0], [0, 1], [0, -1]];
          for (const [di, dj] of deltas) {
            const ni = current.row + di;
            const nj = current.col + dj;
            const neighbor = getCell(ni, nj);
            const nKey = key(ni, nj);

           if (neighbor && !neighbor.isWall && !visited.has(nKey))
{
              const alt = dist[cKey] + 1;
              if (!(nKey in dist) || alt < dist[nKey]) {
                dist[nKey] = alt;
                prev[nKey] = cKey;
                openSet.push({ row: ni, col: nj, dist: alt });
              }
            }
          }
        }

        alert("Шлях не знайдено");
      }
```

Лістинг Г.11, аркуш 2