МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

ВЕБЗАСТОСУНОК ДЛЯ ІНТЕРАКТИВНОЇ ВІЗУАЛІЗАЦІЇ РОБОТИ АЛГОРИТМІВ

Виконав: студент групи 2П-22
Спеціальності
121 Інженерія програмного забезпечення
Володимир БЛИЗНЮК
Керівник:
Станіслав МАРЧЕНКО

Черкаси 2026

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ

Циклова комісія комп'ютерної інженерії та інформаційних технологій

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувачка циклової комісії КІ та ІТ

_____ Наталя ФАЛЬЧЕНКО
(підпис)
«_____» _____2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Близнюку Володимиру Петровичу

1. Тема кваліфікаційної роботи Вебзастосунок для інтерактивної візуалізації роботи алгоритмів

Керівник роботи Марченко Станіслав Віталійович, викладач першої категорії
затверджені наказом закладу фахової передвищої освіти
від «06» жовтня 2025 року № 84/1-у.
2. Строк подання студентом кваліфікаційної роботи 02.06.2026
3. Вихідні дані до кваліфікаційної роботи: мова програмування JavaScript (стандарт ES6+), бібліотека побудови інтерфейсів React.js, середовище розробки Visual Studio Code, бібліотеки для створення інтерактивних анімацій (Framer Motion, CSS3 Transitions), мова опису діаграм UML (діаграми класів та варіантів використання).
4. Зміст кваліфікаційної роботи: аналіз предметної області та огляд наявних рішень (сфери застосування візуалізації обчислень, метрики якості та інформативності візуалізації алгоритмів і обчислень, огляд наявних програмних рішень, постановка задачі на розроблення програмного продукту); проєктування та програмна реалізація вебзастосунку (формування вимог до програмного забезпечення; попередня архітектура програмної системи; проєктування інтерфейсу користувача; програмна реалізація ключових модулів); впровадження та тестування вебзастосунку (стратегії та середовища тестування; оцінювання зручності використання; розгортання застосунку та налаштування CI/CD).
5. Дата видачі завдання 15.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	14.10.2025	
2	Розділ 1. Аналіз предметної області та огляд наявних рішень	09.12.2025	
3	Розділ 2. Проєктування та програмна реалізація вебзастосунку	10.03.2026	
4	Розділ 3. Впровадження та тестування застосунку	28.04.2026	
5	Висновки	12.05.2026	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	26.05.2026	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	02.06.2026	
8	Подання кваліфікаційної роботи на затвердження завідувачці циклової комісії (за 7 днів до захисту)	10.06.2026	

Студент _____ Володимир БЛИЗНЮК
(підпис)

Керівник роботи _____ Станіслав МАРЧЕНКО
(підпис)

АНОТАЦІЯ

Кваліфікаційну роботу присвячено актуальній проблемі підвищення ефективності вивчення фундаментальних основ програмування через інструменти інтерактивної візуалізації. Головним результатом роботи є проєктування та розробка вебзастосунку для наочного представлення роботи складних алгоритмів та структур даних, що дозволяє детально відстежувати зміни станів системи в реальному часі.

У роботі детально проаналізовано та програмно реалізовано компонентно-орієнтований підхід до побудови інтерфейсу на базі бібліотеки React.js. Суть запропонованого рішення полягає у створенні модульної архітектури, де логіка обчислювального алгоритму відокремлена від рівня візуального відображення. Спроєктовано систему синхронізації внутрішнього стану програми з анімаційними переходами (Framer Motion / CSS3 Transitions), що забезпечує високу плавність та точність відтворення кожного кроку роботи алгоритму.

Функціональні можливості розробленого застосунку включають інтерактивне керування процесом візуалізації: покрокове виконання (step-by-step), регулювання швидкості анімації, динамічне генерування вхідних наборів даних та візуальну індикацію ключових операцій (порівняння, перестановка, обхід вузлів).

Програмний продукт реалізовано мовою JavaScript з використанням сучасних стандартів екосистеми фронтенд-розробки. Проведено експериментальне тестування за методологією Black Box, яке підтвердило стабільність роботи застосунку та його високу продуктивність при візуалізації великих масивів даних.

Ключові слова: ВІЗУАЛІЗАЦІЯ АЛГОРИТМІВ, REACT.JS, JAVASCRIPT, ВЕБЗАСТОСУНОК, СТРУКТУРИ ДАНИХ, UI/UX, UML.

ABSTRACT

The qualification work is dedicated to the current problem of increasing the efficiency of studying the fundamentals of programming through interactive visualization tools. The main result of the work is the design and development of a web application for a visual representation of the work of complex algorithms and data structures, which allows you to track changes in system states in real time in detail.

The work analyzes in detail and programmatically implements a component-oriented approach to building an interface based on the React.js library. The essence of the proposed solution is to create a modular architecture, where the logic of the computational algorithm is separated from the visual display level. A system for synchronizing the internal state of the program with animation transitions (Framer Motion / CSS3 Transitions) has been designed, which ensures high smoothness and accuracy of reproduction of each step of the algorithm.

The functionality of the developed application includes interactive control of the visualization process: step-by-step execution, animation speed adjustment, dynamic generation of input data sets and visual indication of key operations (comparison, permutation, node traversal).

The software product is implemented in JavaScript using modern standards of the front-end development ecosystem. Experimental testing using the Black Box methodology was conducted, which confirmed the stability of the application and its high performance when visualizing large data sets.

Keywords: ALGORITHMS VISUALIZATION, REACT.JS, JAVASCRIPT, WEB APPLICATION, DATA STRUCTURES, UI/UX, UML.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

AV Візуалізація алгоритмів (Algorithm Visualization).

BFS Пошук у ширину (Breadth-First Search).

BOM Об’єктна модель браузера (Browser Object Model).

CAGR Середньорічний темп приросту (Compound Annual Growth Rate).

CI/CD Безперервна інтеграція та безперервне розгортання (Continuous Integration / Continuous Deployment).

DFS Пошук у глибину (Depth-First Search).

DOM Об’єктна модель документа (Document Object Model).

D3.js JavaScript-бібліотека для обробки та візуалізації даних на основі вебстандартів (Data-Driven Documents).

EdTech Освітні технології (Educational Technology).

ES6+ Сучасні стандарти мови програмування JavaScript (ECMAScript 6 і новіші).

FPS Кількість кадрів за секунду, частота оновлення екрана (Frames Per Second).

LLM Велика мовна модель (Large Language Model).

SaaS Програмне забезпечення як послуга (Software as a Service).

SPA Односторінковий вебзастосунок (Single Page Application).

SVG Формат масштабованої векторної графіки (Scalable Vector Graphics).

UI Інтерфейс користувача (User Interface).

UML Уніфікована мова моделювання (Unified Modeling Language).

UX Досвід взаємодії користувача з інтерфейсом (User Experience).

WebGL Програмна бібліотека для рендерингу тривимірної графіки у браузері без сторонніх плагінів (Web Graphics Library).

ВСТУП

Сучасна індустрія розробки програмного забезпечення та IT-освіти переживає етап цифрової трансформації, де особливе місце посідає автоматизація процесів навчання. Фундаментальною базою для підготовки інженерів-програмістів є вивчення алгоритмів та структур даних, проте їхня абстрактна природа створює значний когнітивний бар'єр для початківців. Згідно з останніми аналітичними звітами у сфері EdTech, глобальний ринок цифрових освітніх платформ оцінюється у понад 450–520 млрд доларів США із середньорічним темпом приросту (CAGR) на рівні 15–18%. При цьому попит на інтерактивні інструменти візуалізації складних процесів зріс на 60% у порівнянні з попередніми роками, оскільки традиційні статичні методи навчання (лекції, підручники) демонструють низьку ефективність при поясненні динамічних систем.

Поява сучасних фронтенд-технологій, зокрема бібліотеки React.js, відкрила нові можливості для створення високоефективних односторінкових застосунків (SPA) з миттєвим відгуком інтерфейсу. Проте більшість наявних рішень для візуалізації алгоритмів часто мають перевантажений інтерфейс, відсутність гнучкого налаштування вхідних даних або низьку продуктивність при роботі з великими масивами інформації. Це створює нагальну потребу в розробці оптимізованих вебзастосунків, які забезпечують точну, покрокову візуалізацію логіки обчислень у реальному часі, поєднуючи високу швидкість рендерингу з інтуїтивно зрозумілим UX-дизайном.

Об'єктом дослідження виступають процеси візуального представлення та інтерактивного моделювання роботи алгоритмів і структур даних у середовищі сучасних вебзастосунків.

Предмет дослідження – методи, архітектурні рішення та програмні засоби для розроблення динамічних інтерфейсів візуалізації обчислювальних процесів.

Метою роботи є підвищення ефективності та наочності процесу вивчення фундаментальних основ програмування шляхом розроблення вебзастосунку на базі бібліотеки React, що реалізує метод інтерактивної покрокової візуалізації

обчислювальних алгоритмів та структур даних. Для досягнення мети необхідно вирішити такі завдання:

- здійснити системний аналіз існуючих методів візуалізації алгоритмів сортування та пошуку, визначити їхні переваги та недоліки в контексті освітнього процесу;
- обґрунтувати вибір технологічного стека та компонентної архітектури програмного продукту, що забезпечуватиме високу швидкість відгуку інтерфейсу;
- розробити алгоритм екстракції станів обчислювального процесу на кожній ітерації для їх подальшої передачі на рівень візуального відображення;
- спроєктувати та імплементувати програмну логіку керування часовими затримками та анімаційними переходами для синхронізації кроків алгоритму з UI-елементами;
- реалізувати функціональність динамічного генерування вхідних наборів даних та інтерактивні інструменти контролю безпосередньо у вікні перегляду;
- провести тестування розробленого рішення за критеріями коректності візуального відтворення, продуктивності при роботі з великими масивами даних та зручності інтерфейсу.

Розроблений програмний продукт дозволятиме інтегрувати наочну демонстрацію алгоритмів в освітній процес технічних спеціальностей. Застосунок надаватиме можливість значно скоротити час на опанування складних концепцій роботи зі структурами даних, знижуючи поріг входження для студентів-початківців та забезпечуючи зручне середовище для експериментів з вхідними даними.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД НАЯВНИХ РІШЕНЬ

1.1 Сфери застосування візуалізації обчислень

У сучасній інформатиці та інженерії програмного забезпечення візуалізація алгоритмів (AV) виступає ключовим інструментом для дослідження, верифікації та розуміння динаміки обчислювальних процесів. На відміну від статичного аналізу вихідного коду, інтерактивна візуалізація дозволяє спостерігати за еволюцією структур даних (масивів, дерев, графів) у реальному часі. Головною сферою застосування AV є побудова точної ментальної моделі обчислення: відстеження індексів, змін станів пам’яті, перевірка алгоритмічних інваріантів та аналіз граничних випадків [1-4].

У межах кваліфікаційної роботи візуалізацію алгоритмів доцільно розглядати як окремий програмний механізм перетворення трас виконання у зрозумілі для користувача стани інтерфейсу. Тому предметом аналізу є не тільки педагогічна наочність, а й коректність моделі станів, відокремлення алгоритмічної логіки від шару відображення, керованість анімаційного часу, тестованість переходів між станами та придатність рішення до розширення новими алгоритмами (рис. 1.1). Такий підхід узгоджується з базовими положеннями аналізу алгоритмів, дослідженнями освітньої ефективності візуалізації алгоритмів та оглядами програмної візуалізації для початкового навчання програмуванню [1-4].

Критично важливо, що наявність рухомих об’єктів на екрані сама по собі не гарантує навчального результату. Метааналіз досліджень ефективності AV показує, що пасивне спостереження за анімацією часто дає слабший результат, ніж очікується, якщо студент не залучений до прогнозування наступного кроку, зміни вхідних даних, пояснення проміжного стану або пошуку помилки [2; 3]. Отже, для розроблюваного вебзастосунку базовою вимогою має бути забезпечення активної взаємодії: покроковий режим, паузу, повернення до

попереднього стану, зміну параметрів і зіставлення візуального кроку з логікою алгоритму.

Рисунок 1.1 – Сфери застосування інтерактивної візуалізації алгоритмів

Для комплексного розуміння прикладного значення технологій AV, доцільно класифікувати та деталізувати основні сфери їх інтеграції в сучасну комп’ютерну індустрію та академічне середовище. Вища освіта в галузі комп’ютерних наук є найбільш масовою та традиційною для застосування AV. Вивчення складних абстракцій та динамічних процесів виключно за допомогою текстового вихідного коду або статичних діаграм часто створює високий когнітивний бар’єр для здобувачів освіти. Інтерактивна візуалізація трансформує абстрактні математичні операції у наочні графічні примітиви. Вона дозволяє безпосередньо спостерігати за процесами рекурсивного спуску та підйому, динамічного балансування деревовидних структур (наприклад, AVL, червоно-чорні або B-дерева), покрокового розгортання графів за алгоритмами пошуку вглиб (DFS) чи вшир (BFS), а також за процесами релокації елементів і перестановки покажчиків у реальному часі. Можливість керувати темпом виконання, змінювати вхідні набори даних (включаючи заздалегідь згенеровані

критичні тести) та аналізувати покрокову трасування коду сприяє глибокому засвоєнню фундаментальних концепцій [2-6].

У практиці комерційної розробки програмного забезпечення інструменти AV виходять за межі чисто дидактичних завдань і стають частиною процесів дебаггінгу та оптимізації. Коли мова йде про проєктування розподілених систем, багатопотокових обчислень або складних конвеєрів обробки даних, стандартні текстові лог-файли та класичні точки зупину (breakpoints) стають малоефективними через надмірні обсяги інформації. Візуалізація станів дозволяє інженерам «побачити» аномалії в архітектурі: стани взаємоблокування (deadlocks), гонитви даних (race conditions), нерівномірний розподіл навантаження у кластерах або виявити критичні «вузькі місця» (bottlenecks) у чергах повідомлень [4; 7].

Для науковців, які займаються розробкою нових або оптимізацією існуючих мікроаргоритмів, інтерактивна візуалізація є потужним евристичним інструментом. Візуальне представлення поведінки алгоритму на великих масивах даних або специфічних матрицях дозволяє емпірично оцінити його реальну часову та просторову складність (O-нотацію). Дослідник має можливість візуально ідентифікувати приховані патерни, оцінити ефективність евристичних функцій, проаналізувати щільність розподілу даних (наприклад, оцінити якість хеш-функцій та коефіцієнт виникнення колізій) або виявити математичні закономірності, які важко формалізувати аналітичним шляхом.

Сучасним трендом в IT-індустрії також є інтеграція спрощених рішень візуалізації в платформи для автоматизованого тестування та живого кодингу під час технічних співбесід. Це дозволяє екзаменаторам оцінювати не лише фінальну працездатність написаного кандидатом коду та його синтаксичну коректність, а й хід його алгоритмічного мислення. Візуалізація показує, як саме розробник керує пам’яттю, чи коректно він реалізує рух покажчиків у зв’язаних списках та наскільки оптимізовано виконує обхід структур даних.

Відтак, вебзастосунки алгоритмічної візуалізації стають не просто ілюстративними засобами, а повноцінними середовищами інтерактивного

тестування логіки, де звичайне читання коду замінюється керованим просторово-часовим аналізом. Завдяки кросплатформеності та високій продуктивності сучасних вебтехнологій (зокрема, за рахунок використання елементів HTML5 Canvas, SVG та технологій апаратного прискорення графіки WebGL), такі системи забезпечують безбар'єрний, миттєвий доступ до потужного аналітичного інструментарію через браузер, повністю нівелюючи потребу у локальному розгортанні складних компіляторів та специфічних середовищ розробки (IDE) [8-10].

Методика візуального аналізу роботи алгоритмів ґрунтується на просторовому відображенні математичних понять. Оскільки будь-який обчислювальний процес розгортається у часі та просторі комп'ютерної пам'яті, основним підходом до його дослідження є покрокове відтворення (анімація). Цей метод забезпечує користувачеві повний контроль над плином часу обчислень, дозволяючи виконувати послідовний перегляд рядків коду у прямому та зворотному напрямках. Проте сучасний аналіз не обмежується лише фіксацією кроків, а вимагає системного розкриття внутрішньої логіки алгоритму за допомогою різних видів відображення. Зокрема, безперервне відстеження станів спрямоване на наочну зміну побудови структур даних, тоді як відображення потоку керування зосереджується на підсвічуванні активних гілок розгалуження, частоти викликів функцій та розгортанні дерев рекурсії.

Відповідно до наукових концепцій Д. Кейма та правил Б. Шнейдермана, розробка інтерфейсів інтерактивних систем візуалізації має підпорядковуватися чіткій послідовності взаємодії, яка пристосована до особливостей алгоритмів і включає такі три етапи [11; 12]:

1. Забезпечення загального огляду структури даних, що дозволяє оцінити початковий набір інформації, наприклад, первинне розташування елементів у масиві чи загальну будову складного графа.

2. Впровадження інструментів масштабування та вибіркового очищення (фільтрації), за допомогою яких користувач може відокремити

потрібні частини графа, відсікти неактивні гілки або зосередитися на конкретній ділянці обчислень.

3. Деталізація поточного стану за запитом, яка полягає у миттєвому виведенні точних значень змінних, індексів указівників чи вагових коефіцієнтів ребер при взаємодії з конкретним графічним елементом системи.

Таблиця 1.2 – Систематизація сфер застосування AV з позицій інженерії програмного забезпечення

Сфера застосування	Що саме візуалізується	Програмно-інженерний результат	Критичне обмеження
Навчання алгоритмів і структур даних	Стан масиву, графа, дерева, покажчиків, черги або стеку на кожній ітерації	Формування ментальної моделі виконання та зв'язку між кодом, даними й результатом	Без активного залучення користувача анімація може перетворитися на пасивну демонстрацію [2; 3]
Налагодження та верифікація логіки	Переходи між станами, інваріанти, граничні випадки, помилкові переходи	Раннє виявлення дефектів у логіці алгоритму та сценаріях взаємодії	Візуалізація повинна точно відповідати трасі виконання, інакше вона приховує дефекти замість їх виявлення [4; 7]
Порівняння алгоритмічних стратегій	Кількість операцій, глибина рекурсії, довжина шляху, обсяг переглянутого простору	Обґрунтований вибір алгоритму під конкретний клас задачі	Порівняння має супроводжуватися єдиними вхідними даними та однаковими метриками [1; 16]
Інженерне проєктування UI/UX	Фокус уваги, кольорові стани, темп анімації, щільність керуючих елементів	Зменшення когнітивного навантаження та підвищення керованості інтерфейсу	Надлишкова деталізація може збільшити навантаження і знизити зрозумілість [5; 6; 13]

Головна перевага таких методів полягає у залученні зорового сприйняття інженера для швидкого виявлення прихованих зразків та закономірностей, таких як особливості обходу графів у глибину чи переміщення вказівників у лінійних масивах. Ефективність цього процесу досягається через просторове та колірне позначення абстрактних об'єктів. Механізм перетворення переводить математичні величини у графічні форми, де кожному стану комп'ютера відповідає певна зовнішня ознака. Наприклад, числове значення елемента відображається у вигляді висоти стовпчика або радіуса вузла, а динамічна зміна

стану – як-от порівняння двох елементів чи перестановка в пам’яті – позначається миттєвою зміною кольору або геометричним переміщенням об’єкта. Це дозволяє суттєво знизити розумове навантаження на дослідника, перетворюючи складний аналіз текстових звітів та знімків пам’яті на зрозуміле спостереження за наочними змінами.

З позиції когнітивного проєктування така схема має обмеження. Якщо на одному кадрі одночасно змінюється забагато елементів, користувач бачить рух, але не розуміє причинно-наслідковий зв’язок між командою алгоритму та зміною структури даних. Тому візуальне кодування має підпорядковуватися принципам мультимедійного навчання: виділяти релевантний об’єкт, прибирати другорядний шум, узгоджувати підпис, колір і рух та не дублювати інформацію без потреби [5; 6; 13]. Для розроблюваного застосунку це означає потребу в мінімальній кількості одночасних акцентів: активні індекси, порівнювані елементи, результат операції та пояснення поточного кроку.

Процес, зображений на рисунку 1.2, ілюструє практичне застосування покрокового аналізу на прикладі пошуку найкоротшого шляху на дискретній сітці з перешкодами. Візуальна модель чітко розмежовує просторові стани алгоритму за допомогою контрастного колірного кодування:

- топологічні орієнтири: початковий (синій квадрат ліворуч) та цільовий (червоний квадрат праворуч) вузли;
- фронт хвилі та метрика: числові маркери зі значеннями «5» та «10» відображають динамічну зміну вартості переміщення (ваги ребер або значень цільової функції) на різних ділянках графа;
- простір пошуку: досліджена область виділена синьою сіткою, обмежувальні стіни – зеленим кольором, а невідвідані вузли залишаються в темній зоні;
- результат обчислення: фінальний оптимальний маршрут візуалізовано суцільною жовтою лінією.

Такий підхід робить інтуїтивно зрозумілим просторове тлумачення математичної умови зупинки алгоритму, фіксуючи точний момент досягнення

кінцевої мети. Отже, впровадження подібних методів покрокового візуального аналізу у сучасні програмні системи створює зручні умови для верифікації, налагодження та дослідження складності обчислювальних процесів у реальному часі.

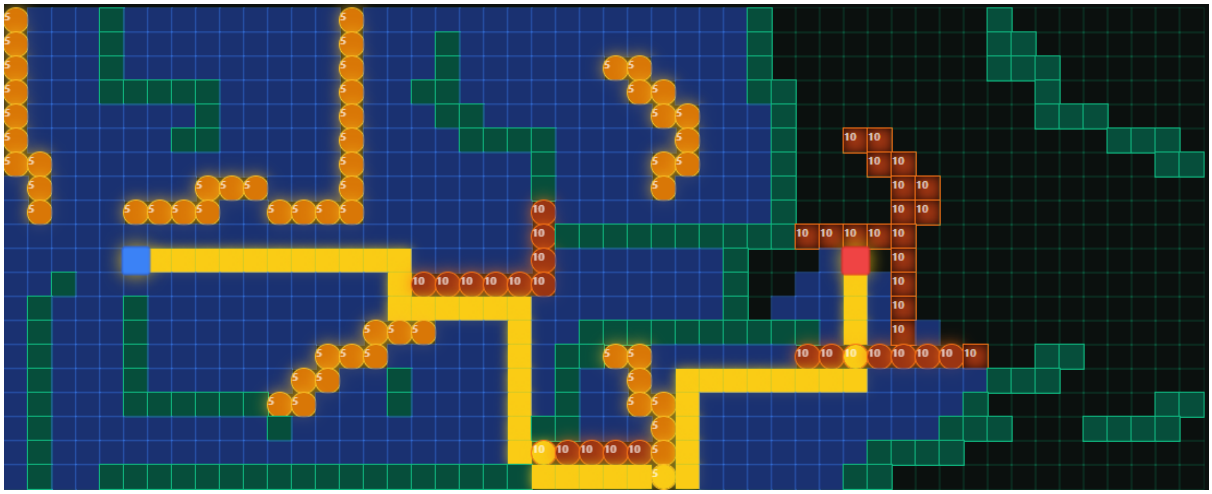

Рисунок 1.2 – Візуалізація динаміки алгоритму пошуку через просторове кодування станів

Водночас, приклад на рисунку 1.2 демонструє й типову небезпеку алгоритмічних візуалізацій: колірна насиченість і велика кількість маркерів можуть створювати враження інформативності, хоча користувачеві може бути незрозуміло, які саме вузли вже остаточно оброблені, які лише поставлені в чергу, а які відкинуті за умовою алгоритму. Тому для графових алгоритмів необхідно явно розділяти щонайменше чотири стани: невідвіданий вузол, вузол у фронті пошуку, оброблений вузол і вузол фінального маршруту. Така декомпозиція робить візуалізацію перевірюваною, а не лише декоративною [1; 14; 15].

Розробка ефективного вебдодатку для візуалізації алгоритмів вимагає чіткого розуміння математичної моделі кожного обчислювального процесу. Основою для оцінки продуктивності алгоритмів є асимптотична нотація, яка описує залежність часу виконання або використаної пам’яті від обсягу вхідних

даних. Усі ці метрики візуалізуються на платформі для формування у користувачів розуміння ефективності коду [1; 16].

Перша велика група алгоритмів, реалізованих у застосунку, стосується сортування масивів. Базові алгоритми, такі як сортування бульбашкою (Bubble Sort), сортування вибором (Selection Sort) та сортування включенням (Insertion Sort), базуються на попарному порівнянні елементів. Їхня математична складність у найгіршому випадку становить $O(n^2)$, що робить їх неефективними для великих наборів даних, але ідеальними для базового навчання. Натомість, швидке сортування та сортування злиттям (Merge Sort) використовують парадигму «розділяй і володарюй», знижуючи часову складність до $O(n \log n)$.

Друга група охоплює графові алгоритми та пошук шляху, які базуються на обробці вершин (V) та ребер (E). Алгоритми пошуку в ширину (BFS) та пошуку в глибину (DFS) мають лінійну складність відносно розміру графа $O(V + E)$. Для знаходження найкоротшого шляху у зважених графах реалізовано алгоритм Дейкстри. Його класична математична оцінка становить $O(V^2)$, однак при використанні структури даних «черга з пріоритетом» складність оптимізується до $O(E + V \log V)$ [1; 17].

Візуалізація також охоплює базові структури даних: масиви, зв'язні списки, стеки, черги та дерева. Процеси додавання, видалення та доступу до елементів у цих структурах мають різну алгоритмічну вартість. Наприклад, доступ за індексом у масиві виконується за $O(1)$, тоді як пошук елемента у незбалансованому бінарному дереві пошуку може деградувати до лінійного часу $O(n)$. Глибоке розуміння цих математичних характеристик є критично важливим для правильної інтерпретації анімацій та метрик, що виводяться у розробленому застосунку.

Отже, у межах розроблення доцільно зберегти обмежений, але інженерно завершений набір алгоритмів: сортування як приклад лінійної структури даних, BFS/DFS як приклад обходу графа та алгоритм Дейкстри як приклад пошуку оптимального шляху у зваженому графі. Надмірне розширення переліку

алгоритмів без якісного трасування, керування станами та тестів знижуватиме якість програмного продукту.

1.2 Метрики якості та інформативності візуалізації алгоритмів і обчислень

Якість візуалізації програмних алгоритмів не може трактуватися як єдина, виключно інтуїтивна властивість на кшталт «наочності» чи «зрозумілості». У сучасній теорії візуалізації обчислювальних процесів вона розглядається як складна, багатокомпонентна характеристика. Ця характеристика формується на кількох взаємопов’язаних рівнях абстракції: від розуміння глобальної навчальної мети до технічних аспектів рендерингу графічних примітивів у браузері користувача.

Для інженерії програмного забезпечення метрики якості мають бути пов’язані не тільки з естетикою зображення, а й із якістю самого програмного продукту. Тому оцінювання AV-застосунку доцільно будувати на перетині трьох площин: коректність алгоритмічної траси, зручність взаємодії користувача та технічна якість реалізації. Така логіка відповідає ISO 9241-11, де зручність використання розглядається через результативність, ефективність і задоволеність у визначеному контексті використання, а також ISO/IEC 25010, де якість ПЗ охоплює функціональну придатність, продуктивність, зручність використання, надійність, супроводжуваність і переносимість [18; 19].

Згідно з концепцією вкладеної моделі Тамари Мюнцнер, процес проєктування та оцінювання візуалізації складається з чотирьох каскадних рівнів. Помилка на вищому рівні неминуче поширюється на нижчі, зводячи нанівець усі технічні зусилля. Іншими словами, ідеально оптимізований алгоритм рендерингу не компенсує помилково обраний спосіб візуального кодування, а красива анімація не допоможе користувачеві, якщо обрана абстракція даних не відповідає суті самого алгоритму [7].

Перший рівень – рівень предметної задачі (Domain situation). У контексті вебдодатку предметною задачею може бути формування алгоритмічного

мислення у студентів та розробників. На цьому етапі якість вимірюється тим, наскільки точно система відповідає потребам цільової аудиторії: чи дозволяє вона детально розібрати кожен крок сортування, чи надає можливість порівняти швидкодію різних підходів. Якщо користувач не може досягти цієї навчальної мети, інструмент вважається неефективним незалежно від його технічної реалізації.

Другий рівень – рівень абстракції даних та дій. Алгоритми оперують абстрактними математичними концептами, такими як масиви, графи, дерева або хеш-таблиці. Якість на цьому рівні залежить від того, наскільки вдало ці математичні структури трансформовано у візуальні об’єкти, з якими може взаємодіяти користувач. Наприклад, для масивів найефективнішою абстракцією є набір стовпців різної висоти, де висота пропорційна значенню елемента.

Третій рівень – рівень візуального кодування та взаємодії. Саме тут визначається, які кольори використовуються для позначення активних елементів (наприклад, червоний для порівняння, зелений для відсортованих), як працюють анімаційні переходи (CSS Transitions, Framer Motion) та яким чином реалізовано покроковий контроль. Якісна візуалізація повинна уникати когнітивного перевантаження, концентруючи увагу глядача лише на тих елементах, які змінюють свій стан у поточній ітерації алгоритму.

Четвертий, найнижчий рівень – алгоритмічна реалізація. На цьому етапі оцінюється технічна продуктивність самого візуалізатора: частота кадрів (FPS), ефективність використання пам’яті та швидкість реакції інтерфейсу. У випадку вебдодатків на базі React.js, це означає мінімізацію зайвих рендерів компонентів (наприклад, через використання React.memo, useCallback) та забезпечення плавності анімацій навіть при масивах з великою кількістю елементів. Тільки комплексне дотримання вимог на всіх чотирьох рівнях дозволяє створити по-справжньому якісний освітній продукт.

Саме тому, при проєктуванні подібних систем, важливо чітко розмежовувати візуалізацію алгоритмів та візуалізацію програмного забезпечення (Software Visualization, SV). Порівняльна схема абстракцій AV та

SV наведена на рисунку 1.3. У цьому контексті межа між AV та SV має практичне значення для постановки задачі. AV у роботі зосереджується на поясненні логіки алгоритму та змін структури даних, тоді як SV ширше охоплює архітектуру, залежності модулів, еволюцію коду, профілювання та налагодження програмних систем. Розроблюваний продукт належить саме до AV, але використовує окремі прийоми SV: трасування коду, часову шкалу станів, візуальне пояснення виконання та метрики продуктивності [4; 7].

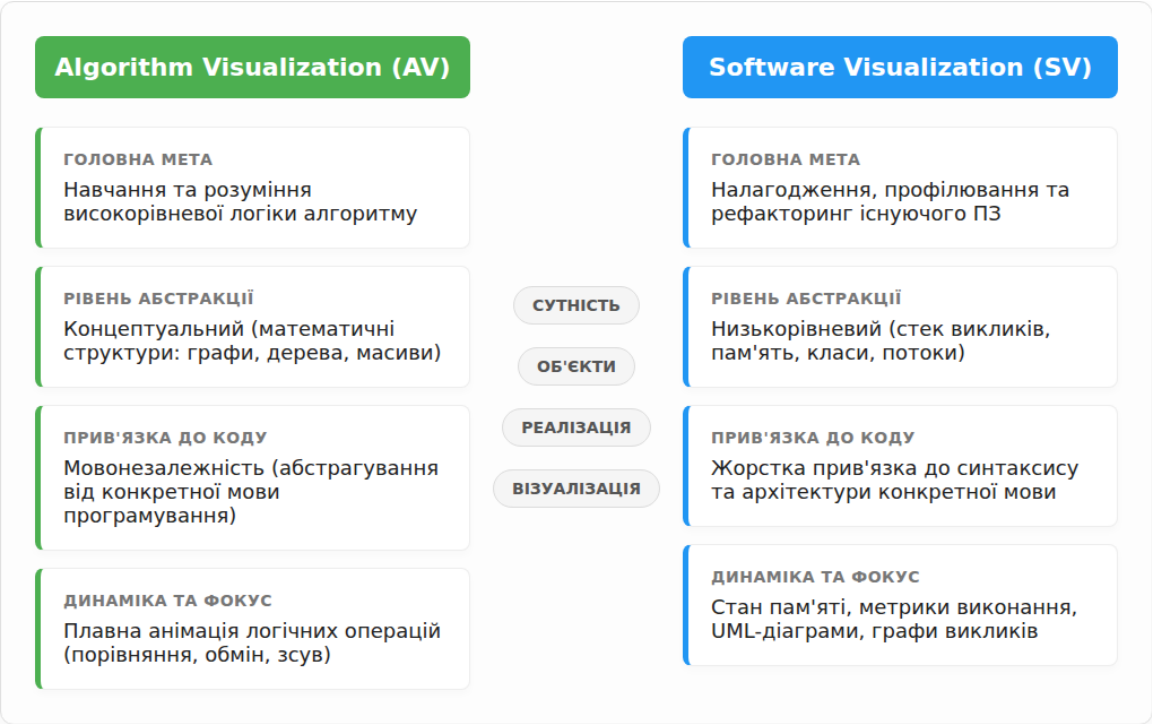

Рисунок 1.3 – Порівняльна схема абстракцій AV та SV

Перший великий клас становлять структурно-геометричні метрики. Вони є найзручнішими для автоматизації, оскільки спираються на внутрішню модель сцени. Найпростішою метрикою цього класу є нормалізована оцінка перетинів ребер. Якщо c – кількість фактичних перетинів неінцидентних ребер, а c_{max} – верхня межа перетинів для конкретного графа, то ця метрика обчислюється за формулою: $Q_{cross} = 1 - (c/c_{maxi})$, де $0 \leq Q_{cross} \leq 1$ [14; 15].

Окрім перетинів, до структурних метрик також відноситься метрика збереження ортогональності та метрика рівномірності розподілу вузлів у просторі екрану. Якщо алгоритм (наприклад, алгоритм Дейкстри або пошук у

ширину) виконується на нерівномірній сітці, користувач може хибно сприймати відстані між вузлами. Тому геометричні показники мають підтримувати симетрію та мінімізувати візуальний хаос під час перемальовування станів.

Критично слід враховувати, що структурно-геометричні метрики не є універсальним критерієм якості. Мінімізація перетинів ребер корисна для графів, однак для сортування масивів важливішими є стабільність позицій, збереження просторової прив’язки елемента та чіткість операції «порівняння/обмін». Емпіричні дослідження графових естетик показують вагомість перетинів і неперервності шляхів, але ці результати потрібно адаптувати до конкретного типу даних і навчального завдання [14; 15].

Другий, не менш важливий клас – когнітивні метрики (Cognitive Load Metrics). Вони оцінюють ступінь інтелектуального навантаження на користувача. Згідно з законом Міллера, короткочасна пам’ять людини здатна одночасно утримувати 7±2 об’єкти. Якщо під час візуалізації сортування масиву одночасно змінюють колір або рухаються більше семи елементів, ефективність сприйняття різко падає. Тому якісний візуалізатор повинен застосовувати строге фокусування: підсвічувати виключно ті змінні чи індекси, які безпосередньо беруть участь у поточній мікрооперації алгоритму [5; 6; 13].

Третій клас охоплює динамічні та анімаційні метрики. Вони базуються на часових характеристиках переходів (Transition Time). Занадто швидка анімація перешкоджає відстеженню логіки переміщення елементів (так званий ефект стробоскопа), тоді як занадто повільна – викликає втрату концентрації та роздратування. Оптимальною вважається тривалість одиничної анімації в межах 250-400 мілісекунд, що забезпечує достатній час для фіксації змін оком людини без відчуття затримок. Частота кадрів при цьому не повинна падати нижче 60 FPS для збереження ілюзії неперервності процесу [13; 18; 19].

Четвертий клас – метрики інтерактивності та контролю. Вони вимірюють гнучкість управління процесом (Time-to-Interactive, кількість дій до результату). Сучасні вимоги диктують необхідність наявності повного спектру контролю: пауза, відтворення, крок вперед та крок назад. Кількість кліків для досягнення

бажаного стану (наприклад, повернення на 5 кроків назад) повинна бути зведена до мінімуму завдяки використанню зручних повзунків прогресу [18; 19].

Зазначені метрики систематизовано в таблиці 1.3. Для подальшого проєктування це допомагає сформувати вимоги до інтерфейсу, компонентів і стану застосунку мають бути сформульовані не декларативно, а через конкретні перевірки дізнаватись: чи правильно сформовано трасу, чи можна керувати виконанням, чи не блокується UI, чи можна додати новий алгоритм без руйнування існуючої логіки.

Таблиця 1.3 – Систематизована модель метрик якості AV-застосунку

Група метрик	Що вимірюється	Приклад показника	Значення для розробки
Алгоритмічна коректність	Відповідність кожного кадру реальному кроку алгоритму	Збіг фінального масиву/шляху з результатом еталонної реалізації	Тестування має перевіряти не лише кінцевий результат, а й послідовність snapshot-станів [1; 4]
Когнітивна керованість	Обсяг інформації, яку користувач має одночасно інтерпретувати	Кількість активних візуальних акцентів на одному кроці	Потрібно уникати одночасного підсвічування несуттєвих елементів [5; 6; 13]
Графічна читабельність	Якість просторового розташування вузлів, ребер, елементів масиву	Кількість перетинів ребер, стабільність позицій, контрастність станів	Для графів критичні перетини й неперервність шляхів, для масивів — сталість позицій [14; 15]
Інтерактивність	Наскільки користувач контролює перебіг виконання	Play/Pause/Step, повернення назад, зміна швидкості	Керування має бути доступним без перезапуску алгоритму [18]
Продуктивність	Стабільність рендерингу й реакція UI	FPS, затримка реакції, час первинного завантаження	Потрібна мінімізація зайвих ререндерів і блокування головного потоку [19; 20]
Супроводжуваність	Придатність архітектури до розширення новими алгоритмами	Ізоляція логіки алгоритму, візуалізатора та стану	Компонентність React має підтримувати додавання модулів без переписування ядра [19; 20]

1.3 Огляд наявних програмних рішень

На сьогоднішній день існує низка платформ та застосунків, які надають інструменти для візуалізації алгоритмів та структур даних. Основною метою таких рішень є полегшення процесу навчання та розуміння складних обчислювальних концепцій. Для визначення вимог до власного програмного продукту доцільно проаналізувати найпопулярніші з існуючих рішень: VisuAlgo, Algorithm Visualizer та Data Structure Visualizations. Їх детальний аналіз дозволить виявити ключові недоліки, яких необхідно уникнути при проєктуванні нової системи. Основними критеріями порівняння є охоплення алгоритмів, покроковість, зв'язок із кодом, адаптивність, продуктивність, простота входження та придатність до розширення [18-20].

VisuAlgo – одна з найстаріших та найпопулярніших платформ, розроблена в Національному університеті Сінгапуру в 2011 році. Платформа охоплює величезну кількість алгоритмів, починаючи від простого сортування і закінчуючи складними алгоритмами на графах (мережеві потоки, найкоротші шляхи) та геометричними алгоритмами. Перевагою VisuAlgo є надзвичайна академічна глибина та деталізація кожного кроку, наявність вбудованих тестів та підтримка багатьох мов [24]. Вигляд вебресурсу відображено на рис. 1.4.

Проте VisuAlgo має суттєві архітектурні та UI/UX недоліки. Її інтерфейс є перевантаженим, з великою кількістю дрібних елементів управління, що ускладнює роботу з платформою на мобільних пристроях. Крім того, технологічний стек, на якому базується система, є застарілим, що періодично призводить до підгальмовування анімацій під час рендерингу складних графів на слабких пристроях. Високий поріг входження робить цей інструмент складним для абсолютних новачків.

Критика VisuAlgo не заперечує його академічної цінності: платформа є сильною як довідково-навчальне середовище з великим охопленням тем. Однак для власного застосунку такий підхід не можна копіювати механічно, оскільки велика кількість режимів, дрібних налаштувань і пояснювальних блоків підвищує інформаційну щільність інтерфейсу. У роботі доцільніше реалізувати

менший набір алгоритмів, але забезпечити чистішу архітектуру компонентів, передбачуване керування станом і зручний сценарій «обрати алгоритм – задати дані – виконати покроково – проаналізувати результат» [18; 19; 24].

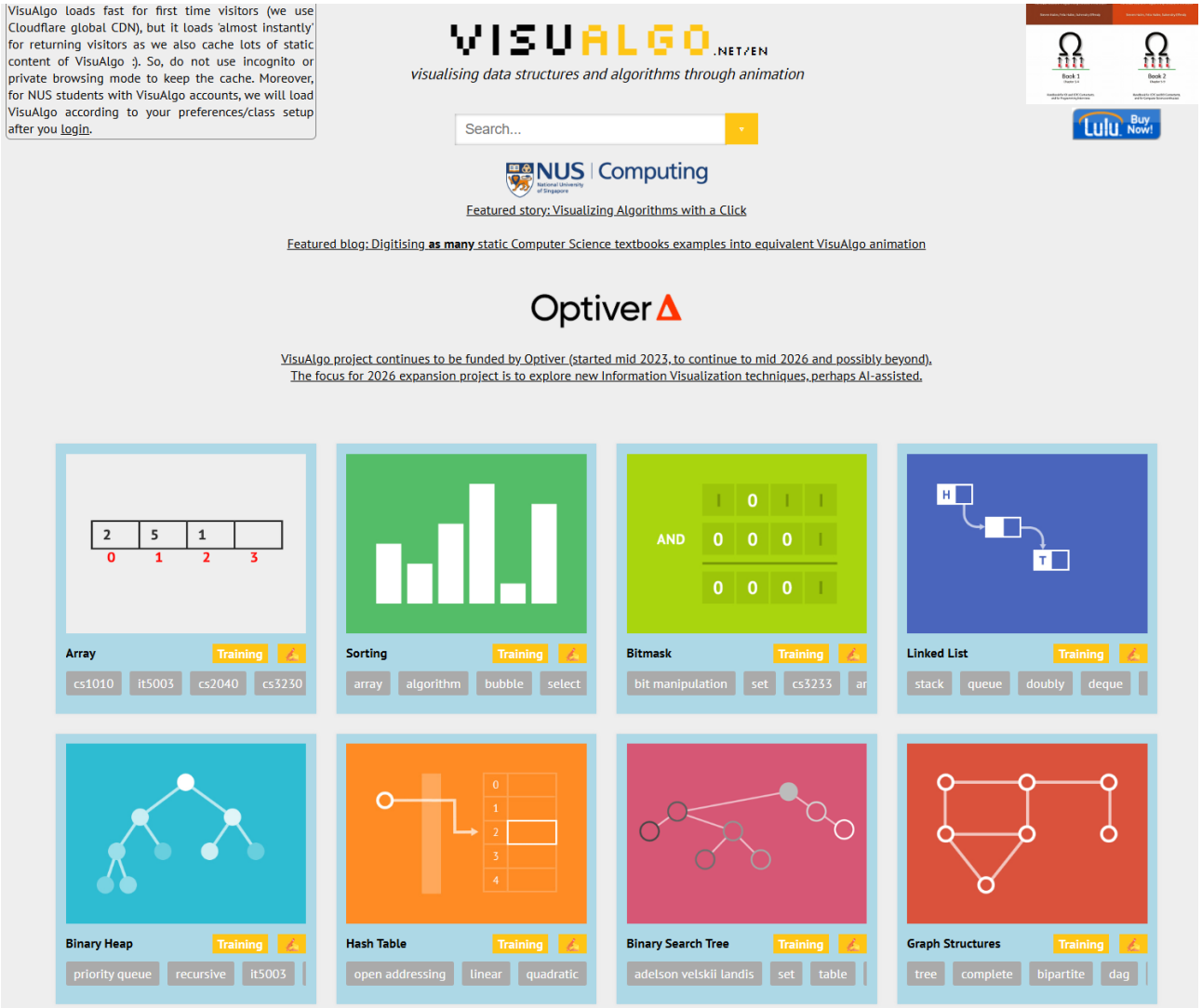

Рисунок 1.4 – Вебресурс visualgo.net

Algorithm Visualizer – це ще один потужний інструмент з відкритим вихідним кодом (Open Source), який акцентує увагу на зв’язку між графічним представленням та самим програмним кодом. Екран розділений на три частини: область візуалізації, консоль виводу та редактор коду з підсвічуванням синтаксису. Головною перевагою системи є можливість бачити, який саме рядок коду виконується в даний момент [25].

Однак, підхід Algorithm Visualizer має свої вразливості. Необхідність одночасно стежити за візуальними елементами, текстовими логами та підсвіченими рядками коду створює надмірне когнітивне навантаження. Користувацький інтерфейс нагадує повноцінне інтегроване середовище розробки (рис. 1.5), що може відлякувати студентів, які лише починають знайомитися з програмуванням. Платформа також страждає від повільного первинного завантаження через значну кількість підключених бібліотек.

Для Algorithm Visualizer сильною стороною є саме прив’язка візуального стану до виконуваного коду, що варто врахувати у власному продукті. Проте повноцінний редактор коду, консоль і складна інфраструктура виконання збільшують поріг входження та обсяг реалізації. Тому для кваліфікаційної роботи раціональнішим є компроміс: не створювати IDE у браузері, а додати панель псевдокоду або пояснення поточного кроку, синхронізоване з уже підготовленою трасою алгоритму [4; 25].

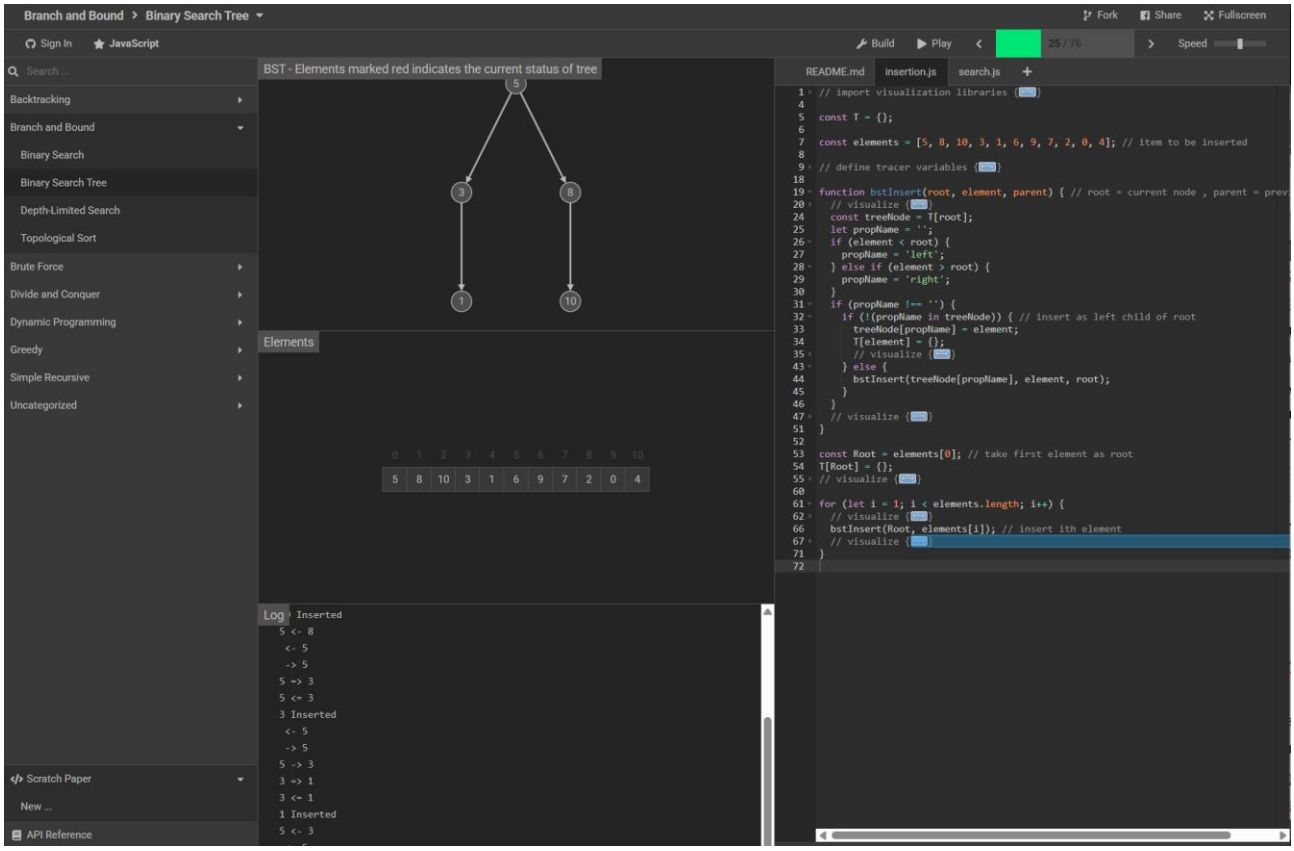

Рисунок 1.5 – Вебресурс algorithm-visualizer.org

Data Structure Visualizations – академічний вебресурс для інтерактивної демонстрації структур даних та алгоритмів, розроблений Девідом Ґаллесом у Department of Computer Science, University of San Francisco [26]. На відміну від вузькоспеціалізованих демонстраторів сортування, цей інструмент охоплює значно ширший набір тем: стеки, черги, зв'язні списки, бінарні дерева пошуку, AVL-дерева, червоно-чорні дерева, B-дерева, B+ дерева, хеш-таблиці, купи, алгоритми сортування, пошук у ширину, пошук у глибину, алгоритм Дейкстри, алгоритми побудови мінімального кістякового дерева, топологічне сортування та окремі задачі динамічного програмування.

Сильною стороною Data Structure Visualizations є його інженерна простота: користувач не лише спостерігає за готовою анімацією, а може виконувати операції над структурами даних – додавати, видаляти, шукати елементи, змінювати параметри та керувати відтворенням (рис. 1.6). У системі передбачено загальні елементи керування анімацією: запуск і пауза, покроковий рух уперед і назад, пропуск до завершення операції, зміна швидкості анімації та налаштування розміру області Canvas. Це робить ресурс корисним не лише для демонстрації алгоритмів на лекції, а й для самостійного експериментування студентів з різними вхідними даними.

Водночас цей інструмент має суттєві обмеження з позиції сучасної інженерії програмного забезпечення. Його інтерфейс є функціональним, але візуально застарілим; структура навігації орієнтована радше на перелік окремих демонстрацій, ніж на цілісний освітній сценарій. У системі немає повноцінної адаптивної UI/UX-моделі, інтегрованого пояснювального шару українською мовою, сценаріїв поступового навчання, модульної ролі викладача, а також сучасного компонентного підходу, властивого React-застосункам. Крім того, сам автор ресурсу зазначає, що попередні версії проєкту проходили еволюцію від Java/Swing і Flash до JavaScript, а кодова база має ознаки історичного розвитку й не є зразком сучасної фронтенд-архітектури. Отже, Data Structure Visualizations доцільно розглядати як змістовно сильний, але технологічно та UX-орієнтовано застарілий аналог, який підтверджує потребу у створенні сучаснішого

вебзастосунку з компонентною архітектурою, адаптивним інтерфейсом, керованим станом і чітким фокусом на навчальних сценаріях.

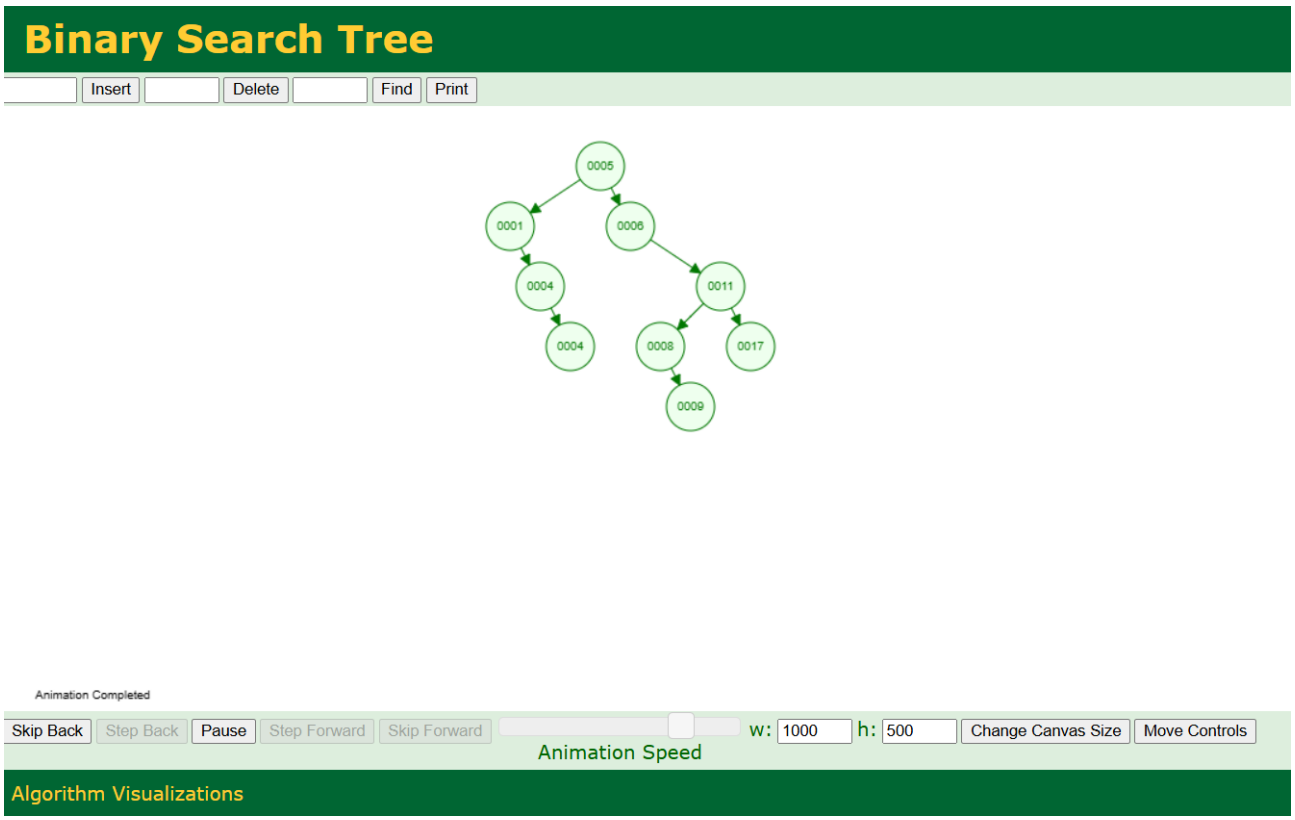

Рисунок 1.6 – Вебресурс Data Structure Visualizations [26]

Аналіз наведених рішень показує, що хоча на ринку існують потужні інструменти, більшість з них або мають застарілий та перевантажений інтерфейс (VisuAlgo, Data Structure Visualizations), або є занадто складними для початківців (Algorithm Visualizer). Наведена таблиця чітко обґрунтовує потребу в розробці нового сучасного вебзастосунку на базі екосистеми React.js, який поєднуватиме інтуїтивно зрозумілий інтерфейс, високу продуктивність та гнучкість розширення.

Звідси, огляд аналогів безпосередньо формує технічну логіку власного рішення: створюється вебзастосунок із чітко відокремленими відповідальностями. Алгоритм генерує послідовність станів, шар керування зберігає позицію на часовій шкалі, а UI лише відображає поточний знімок стану

і події користувача. Саме така схема є доцільною для React-реалізації, оскільки зменшує зв’язність компонентів і спрощує тестування.

Таблиця 1.4 – Інженерна інтерпретація недоліків аналогів для постановки вимог

Виявлена проблема аналогів	Наслідок для користувача	Вимога до власного застосунку
Перевантаження інтерфейсу навчальними блоками, кнопками та режимами	Новачок витрачає увагу на інтерфейс замість логіки алгоритму	Мінімальна панель керування з очевидними діями Play, Pause, Step, Reset
Великий розрив між кодом і візуальним станом або, навпаки, надлишок IDE-елементів	Користувач або не розуміє причину зміни, або перевантажується одночасним читанням коду, консолі й графіки	Панель псевдокоду/ пояснення кроку без повноцінного редактора коду
Обмеження лише одним класом алгоритмів	Не формується переносиме розуміння різних структур даних	Підтримка мінімум двох груп: сортування та графові алгоритми
Непрозора продуктивність при великих наборах даних	Анімація стає рваною, а користувач плутає затримку UI з логікою алгоритму	Контроль розміру вхідних даних, оптимізація ререндерів, вимірювання FPS/затримки
Складність подальшого розширення	Додавання нового алгоритму потребує змін у багатьох частинах системи	Відокремлення алгоритмічної траси, сховища станів і візуального компонента [19; 20]

1.4 Постановка задачі на розробку програмного продукту

З огляду на програмно-інженерний характер роботи, постановка задачі має охоплювати не тільки функціональність кінцевого інтерфейсу, а й архітектурні обмеження. Ключовим проєктним рішенням є подання виконання алгоритму у вигляді масиву дискретних станів (snapshots). Кожен стан повинен містити дані для відображення, тип операції, активні елементи, пояснення кроку та, за потреби, значення метрик. Такий підхід дає змогу відокремити обчислення від анімації, забезпечити покрокове виконання, повернення назад і тестування коректності без прив’язки до конкретної швидкості рендерингу [1; 4; 20-23].

Відповідно до проведеного аналізу предметної області, розроблюваний програмний продукт повинен задовольняти наступним функціональним вимогам:

1) надання користувачеві набору класичних алгоритмів (наприклад, алгоритми сортування: Bubble Sort, Quick Sort, Merge Sort тощо) для дослідження;

2) забезпечення можливості динамічної генерації вхідних даних (масивів чисел) різного розміру, а також ручного введення значень;

3) реалізація панелі керування процесом візуалізації, яка включає функції запуску, паузи, покрокового виконання та регулювання швидкості анімації;

4) наочна кольорова та просторова індикація поточного стану алгоритму (порівняння елементів, перестановка, обхід).

До функціональних вимог також доцільно додати синхронізоване пояснення кроку алгоритму: назву поточної операції, активні індекси або вершини, причину переходу до наступного стану та короткий текстовий коментар. Це перетворює анімацію з декоративного ефекту на інструмент пояснення причинно-наслідкових зв’язків. Для графових алгоритмів така вимога особливо важлива, оскільки один і той самий рух фронту пошуку може відповідати різним логічним станам: постановці вузла в чергу, вилученню з черги, релаксації ребра або фіксації найкоротшої відстані.

До нефункціональних вимог належать використання компонентного підходу на базі бібліотеки React.js для забезпечення модульності, масштабованості та зручності підтримки коду; висока продуктивність та плавність анімацій завдяки оптимізованому управлінню станом та використанню ефективних інструментів; адаптивний та інтуїтивно зрозумілий інтерфейс, що коректно відображається у сучасних веббраузерах. Нефункціональні вимоги потрібно деталізувати за моделлю якості ПЗ. Для функціональної придатності система має правильно відтворювати результат алгоритму та проміжні стани; для продуктивності – не допускати помітного блокування інтерфейсу; для зручності використання – забезпечувати зрозумілий сценарій роботи без надлишкових режимів; для супроводжуваності – дозволяти додавати нові алгоритми через

єдиний формат знімка; для переносимості – коректно працювати у сучасних браузерах без встановлення додаткового ПЗ.

Результатом виконання кваліфікаційної роботи має стати повністю функціональний вебзастосунок, готовий до використання в освітньому процесі як допоміжний інтерактивний інструмент для студентів, що вивчають базові алгоритми та структури даних. На рисунку 1.7 узагальнено архітектурну ідею майбутнього вебзастосунку. Алгоритмічна логіка не повинна напряму керувати DOM або анімаціями: вона формує послідовність станів, а рівень інтерфейсу лише відображає обраний стан і реагує на команди користувача. Завдяки цьому можна незалежно тестувати генератор станів, оптимізувати React-компоненти, змінювати швидкість відтворення та додавати нові способи візуального представлення без переписування самого алгоритму [20; 23].

Рисунок 1.7 – Програмно-інженерний конвеєр перетворення алгоритму у візуалізовану послідовність станів

Загалом, у першому розділі кваліфікаційної роботи виконано комплексний системний аналіз предметної області та обґрунтовано доцільність розробки вебзастосунку для інтерактивної візуалізації алгоритмів. Дослідження сфер застосування технологій AV показало, що графічне представлення динаміки обчислень є критично важливим для сучасної IT-освіти, оскільки воно знижує когнітивне навантаження при вивченні складних структур даних. Проведений аналіз математичних основ, асимптотичної складності та метрик якості дозволив визначити оптимальні параметри інтерфейсу, зокрема необхідність підтримки стабільних 60 FPS та тривалості анімацій у межах 250-400 мс для правильного сприйняття користувачем.

Уточнений аналіз показує, що доцільність розробки визначається не відсутністю аналогів як таких, а наявністю незакритої ніші між академічно потужними, але перевантаженими платформами та мінімалістичними демонстраторами окремих алгоритмів. Власний програмний продукт має зайняти середню позицію: обмежене, але достатнє охоплення алгоритмів; чітке покрокове керування; пояснення станів; адаптивний інтерфейс; контроль продуктивності; компонентна архітектура, придатна до розширення.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКУ

2.1 Формування вимог до програмного забезпечення

Формування вимог до такого програмного продукту має спиратися не лише на бажаний перелік функцій, а й на результати аналізу предметної області, виконаного у першому розділі. Зокрема, було встановлено, що ефективність вебзастосунку для візуалізації алгоритмів визначається поєднанням трьох груп властивостей: коректністю відтворення алгоритмічної траси, зрозумілістю взаємодії користувача з візуальною моделлю та технічною стабільністю інтерфейсу під час виконання анімацій.

Важливо також врахувати, що розроблюваний застосунок не повинен механічно повторювати функціональність великих академічних платформ на кшталт VisuAlgo або Algorithm Visualizer. У межах кваліфікаційної роботи доцільніше реалізувати обмежений, але завершений набір можливостей: вибір алгоритму, підготовку вхідних даних, запуск і покрокове керування виконанням, відображення проміжних станів, пояснення поточного кроку та візуальну індикацію ключових операцій. Такий підхід зменшує ризик перевантаження інтерфейсу і водночас дозволяє показати основні програмно-інженерні рішення: декомпозицію системи на модулі, ізоляцію алгоритмічної логіки від UI, керування станом і повторне використання компонентів.

Для створення конкурентоспроможного сучасного вебзастосунку з інтерактивної візуалізації алгоритмів базових вимог (таких як проста анімація масивів) недостатньо. Вимоги до вебзастосунку поділено на функціональні, нефункціональні та сценарні. Функціональні вимоги визначають, які дії має виконувати система; нефункціональні – з якими якісними характеристиками ці дії мають виконуватися; сценарні вимоги описують очікувану поведінку системи з позиції користувача. Базові функціональні вимоги зібрано в таблиці 2.1.

Таблиця 2.1 – Трасування функціональних вимог до сценаріїв і критеріїв приймання

ID	Вимога	Сценарій	Критерій приймання
FR-1	Вибір алгоритму	Користувач обирає алгоритм сортування, BFS/DFS або пошуку шляху; система очищує несумісні параметри попереднього сценарію.	Обраний модуль завантажується без перезавантаження сторінки; UI показує опис, складність і доступні параметри.
FR-2	Підготовка вхідних даних	Генерація масиву або графової сітки; ручне задання старту, фінішу, перешкод і ваг.	Некоректні дані не запускають алгоритм; користувач отримує зрозуміле повідомлення про помилку.
FR-3	Керування відтворенням	Запуск, пауза, крок уперед, скидання, регулювання швидкості.	Після паузи стан не втрачається; наступний крок продовжується з тієї самої позиції траси.
FR-4	Візуальне кодування станів	Активний елемент, порівняння, оброблений вузол, стіна, шлях, помилка.	Для кожного стану існує однозначний клас відображення; два взаємовиключні стани не можуть бути активними одночасно.
FR-5	Трасування псевдокоду	Поточний snapshot містить номер рядка псевдокоду та коротке пояснення.	Підсвічений рядок відповідає фактичній операції алгоритму, а не лише часовій позиції анімації.
FR-6	Збір метрик	Підрахунок порівнянь, обмінів, відвіданих вузлів, довжини маршруту, часу відтворення.	Метрики оновлюються після кожного кроку та скидаються після зміни алгоритму або набору даних.

Наведені функціональні вимоги утворюють логічний ланцюг роботи користувача із застосунком. Спочатку користувач обирає клас алгоритму або конкретний алгоритм, далі формує вхідні дані, після чого запускає процес візуалізації та керує його перебігом. Кожен крок алгоритму повинен перетворюватися на окремий стан інтерфейсу, який містить не лише графічне представлення, а й службову інформацію: активні індекси, тип виконуваної операції, поточні значення змінних, кількість порівнянь або перестановок, а для графових алгоритмів – стан вузла, черги, фронту пошуку й фінального маршруту.

З погляду інженерії програмного забезпечення особливо важливо, щоб алгоритм не змінював елементи інтерфейсу напряму. Його завданням є

формування послідовності знімків стану, які далі інтерпретуються шаром візуалізації. Такий підхід підвищує тестованість системи, оскільки правильність алгоритму можна перевіряти окремо від рендерингу, а коректність відображення – окремо від математичної логіки. Крім того, це полегшує розширення застосунку: додавання нового алгоритму не потребує повного переписування інтерфейсу, якщо новий модуль повертає дані у спільному форматі.

Функціональні вимоги також мають бути пов’язані з педагогічною метою застосунку. Наприклад, функція «Step Forward» потрібна не лише як елемент керування, а як механізм уповільнення алгоритмічного процесу до рівня, придатного для осмислення. Функція «Step Back» важлива для аналізу помилок і повторного перегляду складного фрагмента. Регулювання швидкості анімації дозволяє адаптувати демонстрацію до різного рівня підготовки користувача. Динамічна генерація вхідних даних забезпечує можливість порівнювати поведінку алгоритму на випадкових, майже впорядкованих, обернено впорядкованих або спеціально підібраних наборах даних.

Наведене трасування вимог важливе для подальшого тестування: кожна функція має не лише бути реалізована у вигляді елемента інтерфейсу, а й мати перевірюваний результат. Для AV-застосунку це особливо критично, оскільки візуальна правильність проміжного стану має таке саме значення, як і правильність фінального результату алгоритму.

Якщо функціональні вимоги відповідають на питання «що саме має робити система», то нефункціональні вимоги визначають, наскільки якісно вона має це робити. Для вебзастосунку інтерактивної візуалізації алгоритмів це особливо важливо, оскільки навіть коректно реалізований алгоритм може бути практично непридатним для навчання, якщо інтерфейс блокується, анімація відтворюється ривками, стани оновлюються із затримкою або користувач не розуміє, яка дія відбулася на поточному кроці.

Нефункціональні вимоги доцільно розглядати через модель якості програмного забезпечення. Функціональна придатність означає, що застосунок

правильно відтворює як кінцевий результат алгоритму, так і проміжні стани. Продуктивність передбачає, що оновлення стану та анімаційні переходи не створюють відчутного блокування інтерфейсу. Зручність використання означає наявність зрозумілої навігації, передбачуваних кнопок керування, помітного колірного кодування та відсутність зайвих дій для запуску базового сценарію. Супроводжуваність виявляється у можливості додавати нові алгоритми без руйнування наявної структури коду. Переносимість забезпечується роботою у сучасних браузерах без встановлення додаткового програмного забезпечення. Сценарії нефункціональних вимог до розроблюваного вебзастосунку зібрано в таблиці 2.2.

Таблиця 2.2 – Сценарії нефункціональних вимог до вебзастосунку

Атрибут якості	Стимул	Очікувана реакція системи	Перевірюваний показник
Продуктивність	Користувач запускає алгоритм на великому масиві або щільній сітці.	Інтерфейс залишається керованим; анімація не блокує основний потік.	Стабільна частота кадрів, відсутність “зависання” кнопок керування.
Надійність стану	Користувач намагається змінити граф під час виконання алгоритму.	Система блокує небезпечну дію або переводить її в чергу.	Не виникає суперечливого стану: вузол не може бути одночасно “стіною” і частиною шляху.
Зручність використання	Студент виконує перший запуск без інструктора.	Основні дії доступні за 1-2 кліки, а помилки пояснені в інтерфейсі.	Користувач розуміє, який параметр треба змінити для повторного запуску.
Доступність	Користувач працює з клавіатури або має обмежене сприйняття кольорів.	Фокус, підписи й контраст не залежать лише від кольору.	Кнопки мають текстові мітки, видимий фокус і достатню контрастність.
Супроводжуваність	Потрібно додати новий алгоритм без зміни ядра UI.	Новий модуль реалізує спільний контракт генерації snapshot-станів.	Існуючі алгоритми та компоненти керування не змінюються.

Систематизація нефункціональних вимог показує, що між окремими характеристиками якості існують потенційні конфлікти. Наприклад, підвищення деталізації пояснення може покращити зрозумілість, але водночас збільшити інформаційну щільність інтерфейсу. Збільшення кількості анімаційних ефектів

може зробити взаємодію візуально привабливішою, але погіршити продуктивність на слабших пристроях. Додавання повноцінного редактора коду могло б розширити функціональність, однак суттєво підвищило б складність системи й поріг входження для студентів-початківців.

Тому в межах цієї роботи пріоритет надається не максимальному набору функцій, а стабільності базового навчального сценарію. Ключовими рішеннями є мінімізація кількості одночасних активних елементів, поділ алгоритмічного ядра й візуального шару, використання керованого глобального стану, блокування зміни структури даних під час виконання алгоритму та підтримка єдиного формату знімків стану. Саме ці обмеження роблять систему передбачуваною, тестованою та придатною до подальшого розвитку.

Для уточнення вимог з позиції кінцевого користувача доцільно використати сценарний підхід. У межах розроблюваної системи основними користувачами є студент, який вивчає алгоритми й структури даних, та викладач, який використовує застосунок як демонстраційний інструмент під час пояснення навчального матеріалу. Студенту важливо мати можливість самостійно змінювати вхідні дані, запускати алгоритм у зручному темпі, повертатися до попереднього кроку та бачити пояснення поточної операції. Викладачу важливо швидко підготувати демонстрацію, показати різницю між алгоритмами, зупинити виконання у потрібний момент і пояснити проміжний стан без перевантаження аудиторії деталями реалізації.

Сценарії користувача дають змогу перевести загальні вимоги у конкретні ситуації використання. Наприклад, вимога «підтримувати покрокове виконання» у сценарному вигляді означає, що студент може натиснути кнопку переходу до наступного кроку й одразу побачити, які елементи порівнюються, яка умова перевіряється та як змінюється структура даних. Вимога «динамічно генерувати вхідні дані» означає, що користувач може швидко створити новий набір значень і повторити виконання алгоритму без перезавантаження сторінки. Сценарна деталізація вимог до вебзастосунка наведена в таблиці 2.3.

Таблиця 2.3 – Сценарна деталізація вимог до вебзастосунку

Епік	Основний користувач	Потреба користувача	Критерії приймання
Інтерактивне графове середовище	Викладач, студент	Користувач має змогу створювати або змінювати структуру графового поля, щоб досліджувати поведінку алгоритмів пошуку на різних конфігураціях.	Користувач може змінювати конфігурацію сітки до запуску алгоритму; заблоковані вузли не беруть участі в пошуку; система візуально розрізняє старт, фініш, перешкоди, відвідані вузли та фінальний маршрут.
Синхронізація виконання алгоритму та пояснення коду	Студент-початківець	Користувач має бачити, яка логічна операція виконується на поточному кроці, щоб пов’язати зміну візуального стану з алгоритмічною дією.	Під час кожного кроку виконання підсвічується відповідний рядок псевдокоду; пояснення змінюється синхронно зі станом візуалізації; у покроковому режимі користувач може простежити причинно-наслідковий зв’язок між командою алгоритму та зміною структури даних.
Керування виконанням і часовою шкалою станів	Студент, викладач	Користувач має контролювати темп виконання алгоритму, щоб зупинятися на складних моментах, повторювати окремі кроки та порівнювати проміжні стани.	Користувач може запускати, призупиняти та відновлювати виконання без перезавантаження сторінки; перехід уперед і назад змінює стан візуалізації коректно; зміна швидкості не порушує послідовність кроків алгоритму.
Динамічна підготовка вхідних даних	Студент, викладач	Користувач має швидко формувати різні набори вхідних даних, щоб досліджувати поведінку алгоритмів у типових і граничних випадках.	Користувач може створити новий набір даних без перезавантаження застосунку; система не дозволяє змінювати дані під час активного виконання алгоритму; після очищення всі службові стани алгоритму скидаються.
Мовна гнучкість інтерфейсу	Україномовний та англомовний користувач	Користувач має змогу працювати із застосунком зрозумілою мовою, щоб коректно сприймати терміни, підказки та пояснення алгоритмів.	У хедері або панелі налаштувань доступний перемикач мови; зміна мови оновлює текстові елементи без перезавантаження сторінки; технічні терміни подаються узгоджено в усіх частинах інтерфейсу.

Таким чином, оновлений набір вимог описує систему не як простий аніматор, а як повноцінне інтерактивне середовище (IDE-подібний застосунок) для глибинного аналізу алгоритмічних процесів, що повністю відповідає академічним стандартам сучасних освітніх платформ.

2.2 Попередня архітектура програмної системи

Для реалізації інтерактивного вебзастосунку було обрано сучасний стек фронтенд-технологій, що забезпечує високу продуктивність, модульність коду та швидкість розробки. Основною мовою програмування виступає JavaScript (ES6+), вибір якої зумовлений повною підтримкою браузерами та наявністю механізму функцій-генераторів, що необхідно для реалізації покрокового виконання алгоритмів.

Ядром архітектури є бібліотека React.js. Її використання обґрунтовується концепцією віртуального DOM, що дозволяє мінімізувати ресурсомісткі операції перемальовування інтерфейсу при кожній зміні стану масиву. Це гарантує плавність анімацій навіть при масивах на 100 і більше елементів. Крім того, компонентний підхід React дозволяє інкапсулювати логіку кожного алгоритму в окремий модуль, що значно полегшує масштабування системи.

Як інструмент збірки проєкту застосовано Vite, який забезпечує миттєвий «гарячий» перезапуск (Hot Module Replacement) під час розробки та створює оптимізовані бандли для продакшену. Для стилізації інтерфейсу використано фреймворк Tailwind CSS, що дозволяє описувати дизайн безпосередньо в компонентах через утилітарні класи, забезпечуючи гнучкість і консистентність візуального стилю. Анімаційні переходи елементів реалізовані за допомогою Framer Motion.

Офіційна документація Vite визначає його як інструмент, що поєднує швидкий dev-server з HMR та оптимізоване production-збирання [27]. Для керування глобальним станом у застосунку доцільно використовувати легковагове сховище Zustand, оскільки воно надає hook-орієнтований API без надлишкової шаблонності [28]. Tailwind CSS у цій роботі застосовується не лише як засіб швидкої стилізації, а як спосіб підтримати єдину систему візуальних станів через повторювані допоміжні класи [29]. Для мікроанімацій буде застосовуватися Motion for React (Framer Motion), який орієнтований на UI-анімації production-рівня та дає змогу описувати переходи декларативно на рівні компонентів [30].

Таблиця 2.3 – Архітектурні рішення та їх інженерне обґрунтування

Рішення	Обрана технологія	Перевага для системи	Ризик / обмеження
ADR-1	React як UI-ядро	Компонентна модель, декларативний рендеринг і локалізація станів.	Потрібно контролювати зайві ререндери та не змішувати алгоритмічну логіку з JSX.
ADR-2	Vite як інструмент збірки	Швидкий цикл розробки, HMR, просте production-збирання.	Необхідно контролювати розмір бандлу та не додавати важкі бібліотеки без потреби.
ADR-3	Zustand / локальні хуки для стану	Мінімальна шаблонність, зручне розділення стану візуалізації та налаштувань.	Глобальне сховище не повинно зберігати великі масиви історії без обмеження.
ADR-4	Tailwind CSS	Єдина палітра станів, швидке прототипування, адаптивні класи.	Утилітарні класи треба систематизувати, інакше зростає хаотичність розмітки.
ADR-5	Motion / CSS transitions	Плавні переходи між snapshot-станами.	Анімації мають бути відключувані або спрощені для reduced motion і слабких пристроїв.

Глобальна компонентна архітектура системи логічно розділена на три основні блоки: бічна панель для вибору алгоритмів (Sidebar), панель управління (ControlPanel) та головне полотно для відтворення графіки (VisualizerCanvas). Усі ці компоненти об’єднані кореневим контейнером (App Container) та глобальним сховищем Zustand, як зображено на рис. 2.1.

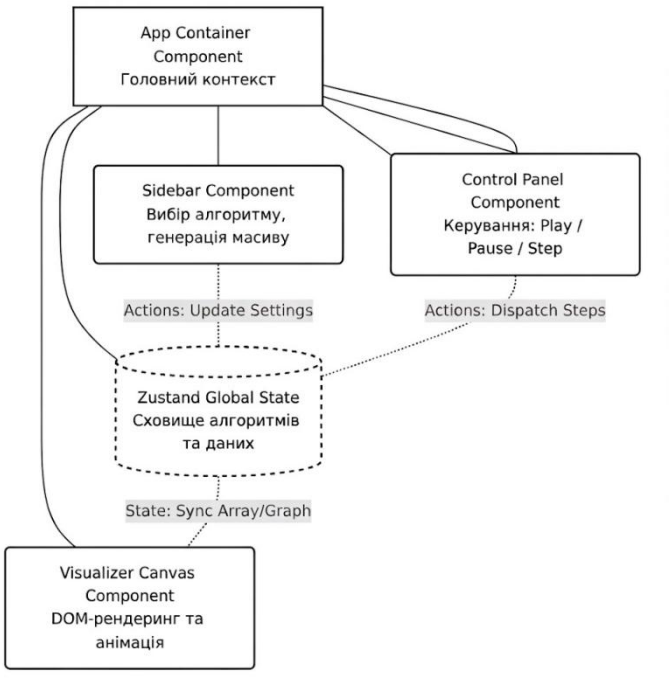

Рисунок 2.1 – Організаційна схема та потік даних вебзастосунку

Для проєктування та опису високорівневої логічної структури програмної системи візуалізації алгоритмів було використано спрощену C4-нотацію. Користувач є кінцевим споживачем, який взаємодіє з графічним інтерфейсом веб-додатка через протокол HTTPS; він здійснює налаштування параметрів графів, обирає алгоритми для візуалізації та керує процесом їх виконання (запуск, пауза, покроковий перегляд).

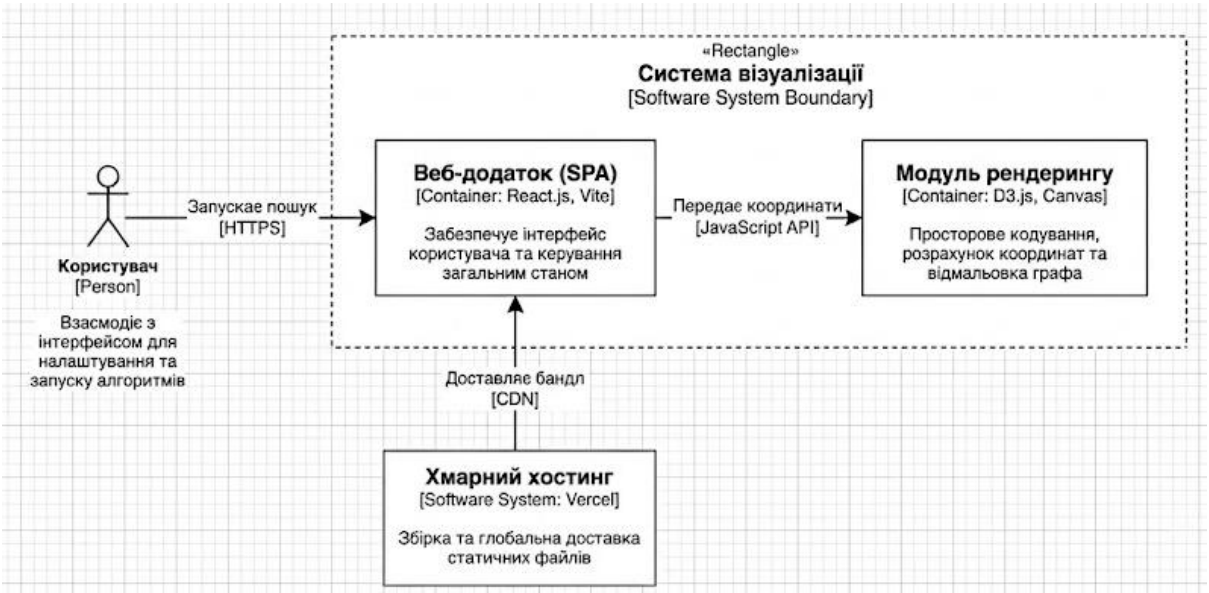

Рисунок 2.3 – Діаграма контейнерів (C4 model) програмної системи візуалізації алгоритмів

Вебдодаток (SPA) – головний клієнтський застосунок, побудований з використанням бібліотеки React.js та інструменту збірки Vite. Цей контейнер є ядром взаємодії з користувачем: він забезпечує маршрутизацію, відображення UI-елементів та керування загальним станом додатку під час виконання алгоритмів.

Модуль рендерингу – ізольований графічний модуль, реалізований за допомогою D3.js та технології HTML5 Canvas. Він відповідає за обчислювально складну частину візуалізації: просторове кодування, розрахунок координат вершин та ребер графа, а також їх плавну відмальовку. Модуль рендерингу отримує необхідні дані від SPA-додатка через внутрішній JavaScript API.

Хмарним хостингом для розгортання вебдодатка виступає платформа Vercel. Вона автоматизує процес збірки проєкту та забезпечує швидку глобальну доставку статичних файлів (бандлу) до браузера користувача за допомогою мережі доставки контенту (CDN).

Для моделювання внутрішньої динаміки системи під час безпосередньої візуалізації алгоритмів було побудовано UML-діаграму послідовностей. Вона демонструє синхронний та асинхронний обмін повідомленнями між ключовими логічними вузлами: Контролером інтерфейсу (UI Controller), Менеджером стану (State Manager) та Генератором алгоритмів (Algorithm Engine) у процесі роботи застосунку (рис. 2.4).

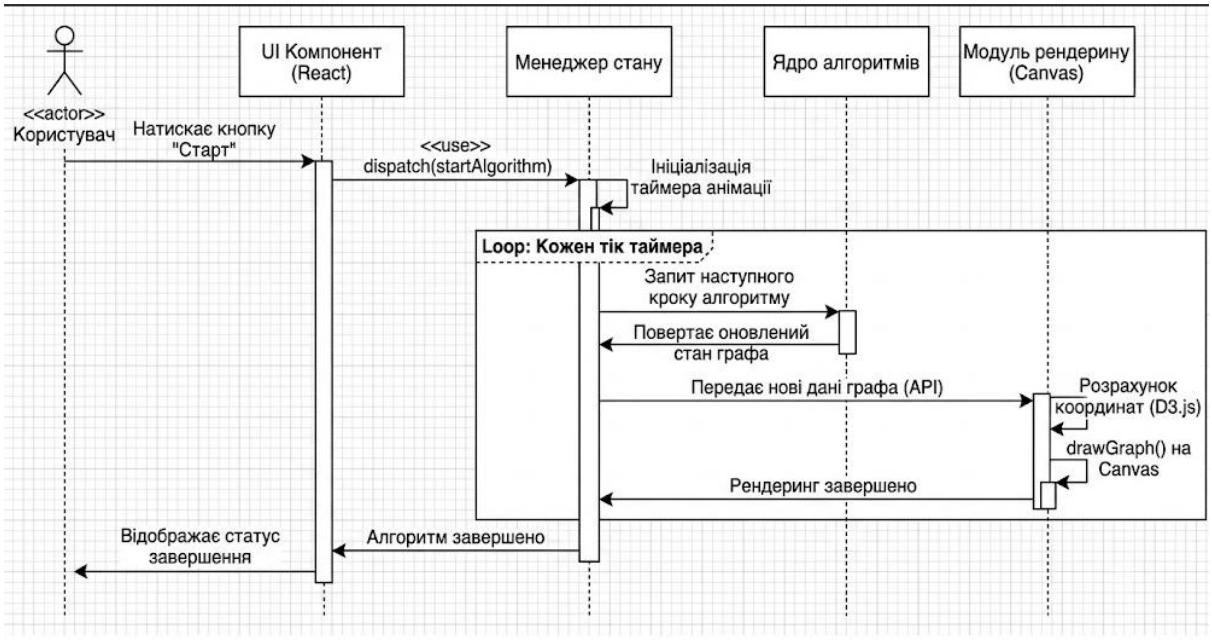

Рисунок 2.4 – UML-діаграма послідовності процесу візуалізації

Процес ініціюється натисканням кнопки запуску, після чого Контролер викликає Генератор алгоритмів. На кожній ітерації генератор за допомогою механізму призупинення виконання повертає новий проміжний стан структури даних. Цей стан передається до Менеджера стану, що своєю чергою ініціює оновлення віртуального DOM у React та передачу нових координат до модуля візуалізації. Така архітектурна модель забезпечує чітке та надійне розділення обчислювальної логіки від графічного представлення.

Розподіл відповідальностей між основними компонентами систематизовано в таблиці 2.4. Він знижує зв’язаність системи. Якщо у подальшому буде додано, наприклад, алгоритм A* або сортування пірамідою, зміни мають торкнутися переважно модуля Algorithm Engine та опису псевдокоду, а не панелі керування, базового полотна або загального сховища стану.

Таблиця 2.4 – Розподіл відповідальності між основними компонентами

Компонент	Відповідальність	Обмеження
App Container	Композиція сторінки, маршрутизація, ініціалізація глобального контексту.	Не містить реалізації конкретних алгоритмів.
Sidebar / AlgorithmSelector	Вибір алгоритму, параметрів, типу даних, розміру масиву або сітки.	Передає лише валідовані параметри до сховища.
ControlPanel	Керування життєвим циклом відтворення: play, pause, step, reset, speed.	Не змінює дані напряму, а викликає дії контролера відтворення.
Algorithm Engine	Формування послідовності snapshot-станів для обраного алгоритму.	Не залежить від React-компонентів та стилів.
VisualizerCanvas	Відображення поточного snapshot у вигляді стовпців, вузлів, ребер або шляху.	Отримує готовий стан і не виконує алгоритмічних обчислень.
CodeTracePanel	Показ псевдокоду, активного рядка й короткого пояснення операції.	Синхронізується через stepId / lineId snapshot-стану.
MetricsPanel	Відображення кількості операцій, відвіданих вузлів, довжини шляху, часу.	Обчислює лише похідні показники з поточного стану.

2.3 Проєктування інтерфейсу користувача

Першою точкою входу для користувача є навігаційний хаб (Головна сторінка). Його спроєктовано за принципом вітрини (Showcase), де кожен навчальний модуль (наприклад, алгоритми сортування або графи) представлений окремою інтерактивною карткою. Це дозволяє користувачеві швидко зорієнтуватися в доступній функціональності. Хаб має сучасний адаптивний дизайн, що коректно відображається на різних пристроях, а також містить перемикач мов для розширення аудиторії користувачів (рис. 2.5).

Рисунок 2.5 – Навігаційний хаб системи візуалізації алгоритмів

Візуальна концепція застосунку побудована на принципах мінімалізму та когнітивного розвантаження. Головний робочий простір поділено на дві основні зони: панель конфігурації (Sidebar) та область візуалізації (Canvas). Панель конфігурації дозволяє користувачеві обирати конкретний алгоритм із випадаючого списку, генерувати нову структуру графа, додавати вузли та ребра вручну. Область візуалізації займає понад 70% екрана, забезпечуючи максимальну наочність для відстеження анімацій. Кольорова палітра побудована на контрастах: нейтральний фон та яскраві акцентні кольори для індикації поточних операцій (наприклад, синій – невідвідані вузли, зелений – відвідані, червоний – поточні активні шляхи).

Окремим аспектом проєктування UI є забезпечення адаптивності. Використання фреймворку Tailwind CSS дозволило реалізувати гнучку

сітку (CSS Grid) та Flexbox-контейнери, які автоматично перелаштовують макет залежно від роздільної здатності екрана користувача. Темна тема обрана як основна базова колірна схема не лише з естетичних міркувань, а й для зниження когнітивного та зорового навантаження під час тривалої роботи з графовими структурами, що повністю відповідає сучасним стандартам розробки інтегрованих середовищ (IDE) та аналітичних дашбордів. На рис. 2.6 наведено інтерфейс розробленого вебзастосунку, де чітко простежується мінімалістичний підхід до організації робочого простору.

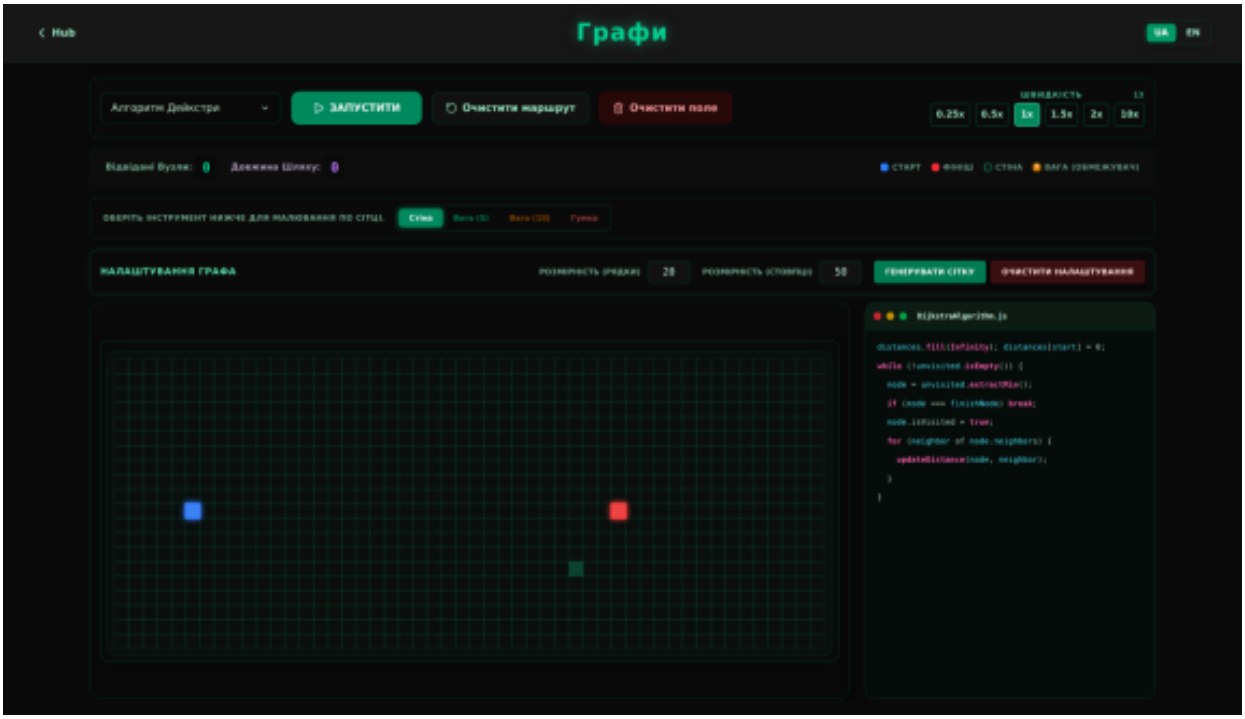

Рисунок 2.6 – Головний робочий простір системи візуалізації

Під час проєктування інтерфейсу необхідно враховувати не лише responsive-макет, а й доступність. WCAG 2.2 встановлює вимоги до контрастності, видимості фокуса, керування з клавіатури та уникнення прихованих елементів фокуса [31]. Для розроблюваного застосунку це означає, що колір не повинен бути єдиним способом передавання стану: активні вузли мають супроводжуватися підписом, іконкою або зміною форми, а кнопки керування повинні мати видимі текстові мітки та коректні мітки регіонів.

Таблиця 2.5 – Систематизація візуальних станів інтерфейсу

Стан	Візуальне кодування	Призначення
Невідвіданий вузол	Нейтральний колір, тонка межа, без анімації.	Не привертає увагу, формує фон для роботи алгоритму.
Вузол у черзі / стеку	Світлий акцент, пунктирна межа або м’яке підсвічування.	Користувач бачить фронт пошуку, але не плутає його з фінальним шляхом.
Поточний вузол	Контрастний колір, коротка pulse-анімація, підпис “current”.	Фіксує причинно-наслідковий центр поточного кроку.
Відвіданий вузол	Стабільний холодний відтінок без повторної анімації.	Показує пам’ять алгоритму та зменшує повторне зчитування.
Фінальний шлях	Товща лінія / яскравий контур / окремий маркер.	Відокремлює результат від процесу пошуку.
Недоступна дія	Disabled, знижена прозорість, пояснювальний tooltip.	Запобігає помилкам зміни графа під час виконання.

Особливу увагу приділено панелі керування часом (Timeline Control). Вона спроєктована за аналогією з медіаплеєром, що інтуїтивно зрозуміло більшості користувачів. Панель містить кнопки: «Play/Pause», «Step Forward», «Step Backward» (за наявності кешування станів) та повзунок керування швидкістю (Playback Speed). Технічно ця панель пов’язана з глобальним станом програми, і її елементи динамічно блокуються (Disable) залежно від поточного статусу алгоритму (наприклад, кнопка «Step» активна лише під час паузи).

Для підвищення загального рівня юзабіліті інтерфейс збагачено системою мікроінтеракцій та візуального зворотного зв’язку. Наприклад, у стані виконання графового алгоритму (Execution state) всі інпути для зміни топології графа чи додавання нових вершин автоматично блокуються (переходять у стан Disabled із відповідним зменшенням прозорості та зміною курсору на «not-allowed»). Це гарантує захист від непередбачуваних мутацій стану (State Mutations) під час роботи генератора. Додатково в інтерфейсі передбачено інформаційну панель для відображення динамічних метрик (наприклад, кількості відвіданих вершин або поточної ваги найкоротшого шляху), що безпосередньо підвищує навчальну ефективність розробленого програмного продукту. Детальний вигляд розробленої панелі керування наведено на рисунку 2.7, яка забезпечує гнучке управління життєвим циклом візуалізації.

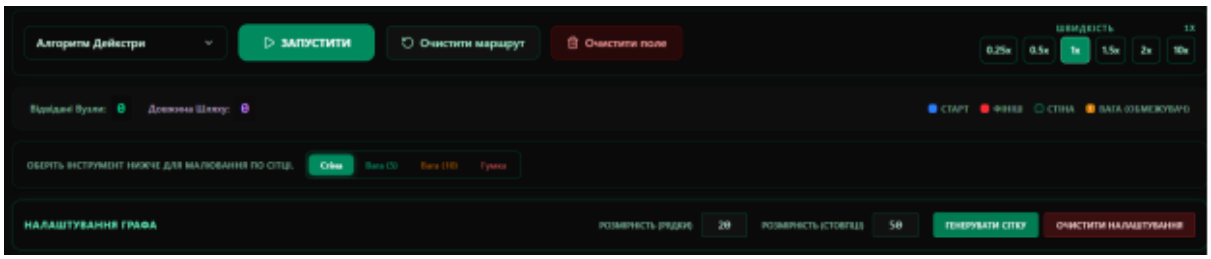

Рисунок 2.7 – Панель керування візуалізацією алгоритмів

Розробка інтерфейсу системи візуалізації алгоритмів вимагає особливого підходу до семантики кольорів, оскільки користувач має миттєво зчитувати стан кожного елемента графа чи масиву без додаткових текстових підказок. Для реалізації цього завдання застосовано метод скінченних автоматів для UI, де кожен вузол графа може перебувати лише в одному строго визначеному візуальному стані: «невідвіданий», «у черзі/стеку», «поточний активний», «відвіданий» або «частина знайденого шляху».

З огляду на психофізіологію зорового сприйняття, для індикації цих станів було інтегровано спеціалізовану палітру кольорів на базі фреймворку Tailwind CSS. Невідвідані вузли мають нейтральний приглушений відтінок (slate-800 або gray-700), що зливається з темним фоном Dark Mode, не відволікаючи увагу. Поточний активний вузол, який алгоритм (наприклад, BFS чи Дейкстри) обробляє в дану мілісекунду, маркується яскравим акцентним кольором (жовтим або червоним), що створює максимальний контраст і фокусує погляд користувача. Відвідані вузли переходять у холодні тони (світло-синій або зелений), формуючи візуальний «слід» алгоритму (алгоритмічну пам’ять екрана). Знайдений найкоротший шлях виділяється неоновим золотим або фіолетовим градієнтом, що психологічно сприймається як «винагорода» та успішне завершення обчислювального процесу.

Крім того, застосунок враховує патерни F-подібного (F-pattern) сканування сторінки, що притаманні людському оку при роботі з екранами. Панель керування та інструменти малювання (Toolbox) розміщені у лівій частині або зверху, тобто там, куди користувач дивиться в першу чергу. Водночас панель псевдокоду розташована у правій частині або знизу, супроводжуючи

візуалізацію як допоміжний контекст, який читається після фокусування на центральній області Canvas.

Проєктування якісного UX/UI передбачає не лише статичну красу (User Interface), але й динамічну поведінку компонентів (User Experience). Однією з найголовніших проблем візуалізаторів є так званий ефект «зорового стрибка», коли елементи раптово з’являються або зникають, через що мозок не встигає зафіксувати логіку зміни алгоритму. Щоб вирішити цю проблему, в інтерфейс було інтегровано бібліотеку Framer Motion (або CSS Transitions), яка забезпечує плавну інтерполяцію між станами.

При додаванні нового вузла на Grid Canvas відбувається мікроінтеракція масштабування – вузол плавно «виростає» від 0 до 100% свого розміру за 200 мс з ефектом відскоку (spring/bounce animation). Це дає мозку користувача час усвідомити зміну топології. Аналогічний підхід застосовано до анімації поширення алгоритму: зміна кольору вузла відбувається не миттєво (0ms), а з плавним переходом (fade-in), що створює візуальний ефект «хвилі» під час обходу в ширину (BFS).

Важливим інженерним рішенням у проєктуванні ергономіки є принцип Poka-yoke – «захист від помилок». Оскільки маніпуляція графом під час виконання алгоритму може призвести до фатальних помилок у React-стані (рекурсивні зациклення, переповнення стеку викликів), весь інтерфейс Sidebar та Header покривається невидимим блокуючим шаром (Overlay) або всі інтерактивні елементи отримують атрибут disabled під час прослуховування стану isAnimating. Користувач отримує миттєвий зворотний зв’язок: курсор змінюється на перекреслене коло, а прозорість кнопок (opacity) падає до 50%, що невербально повідомляє про неможливість зміни параметрів до завершення або зупинки (Pause) поточного алгоритму. Такий проактивний дизайн перешкоджає виникненню фрустрації та підвищує загальний рівень задоволеності продуктом.

Окремо слід передбачити режим зменшення руху для користувачів, чутливих до інтенсивних анімацій. У такому режимі застосунок може зберігати логіку snapshot-станів, але замінювати пружинні переходи на коротке

підсвічування або миттєву зміну класу. Це не погіршує алгоритмічну коректність, але підвищує доступність і переносимість інтерфейсу на слабші пристрої [30; 31].

2.4 Програмна реалізація ключових модулів

Основою життєвого циклу застосунку є гнучка система управління станом, яка базується на використанні React-хуків. Враховуючи вимоги до продуктивності, було вирішено відмовитися від складних глобальних сховищ (наприклад, Redux) на користь локального компонентного стану (useState) та мутабельних референсів (useRef). Це архітектурне рішення обумовлене необхідністю мінімізувати кількість перемалювань (re-renders) компонента під час високочастотних алгоритмічних анімацій (табл. 2.6).

На рівні реалізації доцільно розділити два різні типи стану: стан домену алгоритму та стан інтерфейсу. До стану домену належать масив, граф, список snapshot-кроків, поточний індекс кроку, метрики й інваріанти. До UI-стану належать відкриті панелі, обрана тема, локалізація, активний tooltip і стан кнопок. Змішування цих шарів створює типову помилку React-застосунків: будь-яка косметична зміна запускає повторний рендер важкої візуальної сцени.

Таблиця 2.6 – Мінімальна модель стану модуля візуалізації

Поле стану	Зміст	Інженерне обмеження
algorithmId	Ідентифікатор обраного алгоритму.	Змінюється лише при виборі іншого модуля.
inputModel	Масив або графова сітка з параметрами.	Валідований до запуску; не змінюється під час active execution.
steps[]	Послідовність snapshot-станів.	Може кешуватися, але має обмеження за розміром.
currentStep	Індекс поточного кроку.	Є єдиним джерелом істини для Canvas, псевдокоду й метрик.
playbackStatus	idle / running / paused / completed / error.	Визначає доступність кнопок і безпечні переходи.
speed	Множник швидкості або затримка між кроками.	Не повинен змінювати послідовність логічних станів.
metrics	Кількість операцій, довжина шляху, відвідані вузли.	Оновлюється разом із snapshot, а не окремими таймерами.

Практична перевага такого поділу полягає в тому, що один і той самий snapshot може бути використаний для рендерингу, для тесту, для пояснення псевдокоду та для відновлення стану після паузи. Це робить механізм відтворення передбачуваним і зменшує ризик розсинхронізації між графікою, кодом і метриками. Приклад реалізації механізму паузи за допомогою об’єкта Promise наведено на лістингу 2.1.

Лістинг 2.1 – Механізм призупинення виконання алгоритму

```
const pauseRef = useRef(null);
const checkPause = async () => {
  if (pauseRef.current) {
    await pauseRef.current; // Очікування вирішення промісу
  }
};
const resume = () => {
  if (resolvePauseRef.current) {
    resolvePauseRef.current(); // Продовження виконання
    pauseRef.current = null;
  }
};
```

Ключовим механізмом керування потоком виконання (пауза/відновлення) є використання об’єктів Promise. Змінна pauseResolver діє як бар’єр: якщо користувач натискає кнопку «Пауза», поточний алгоритм призупиняється в очікуванні вирішення промісу, не блокуючи при цьому основний потік браузера (Main Thread). Логіка цього процесу детально зображена за допомогою UML-діаграми діяльності. Як видно з рисунку 2.7, алгоритм генерує повний масив кроків (visitedSteps), після чого інтерфейс поступово відтворює їх. Такий підхід гарантує, що важкі математичні обчислення виконуються миттєво і синхронно, а візуалізація розтягується у часі незалежно від розрахункової частини, забезпечуючи плавність у 60 FPS. У браузері планування кадрових змін прив’язано до requestAnimationFrame(), оскільки цей механізм викликає callback перед наступним repaint і краще узгоджується з циклом оновлення екрана, ніж фіксований setTimeout. Водночас важкі обчислення не слід запускати у кожному кадрі: алгоритм має або заздалегідь сформувати трасу, або видавати кроки пакетами, щоб інтерфейс залишався інтерактивним.

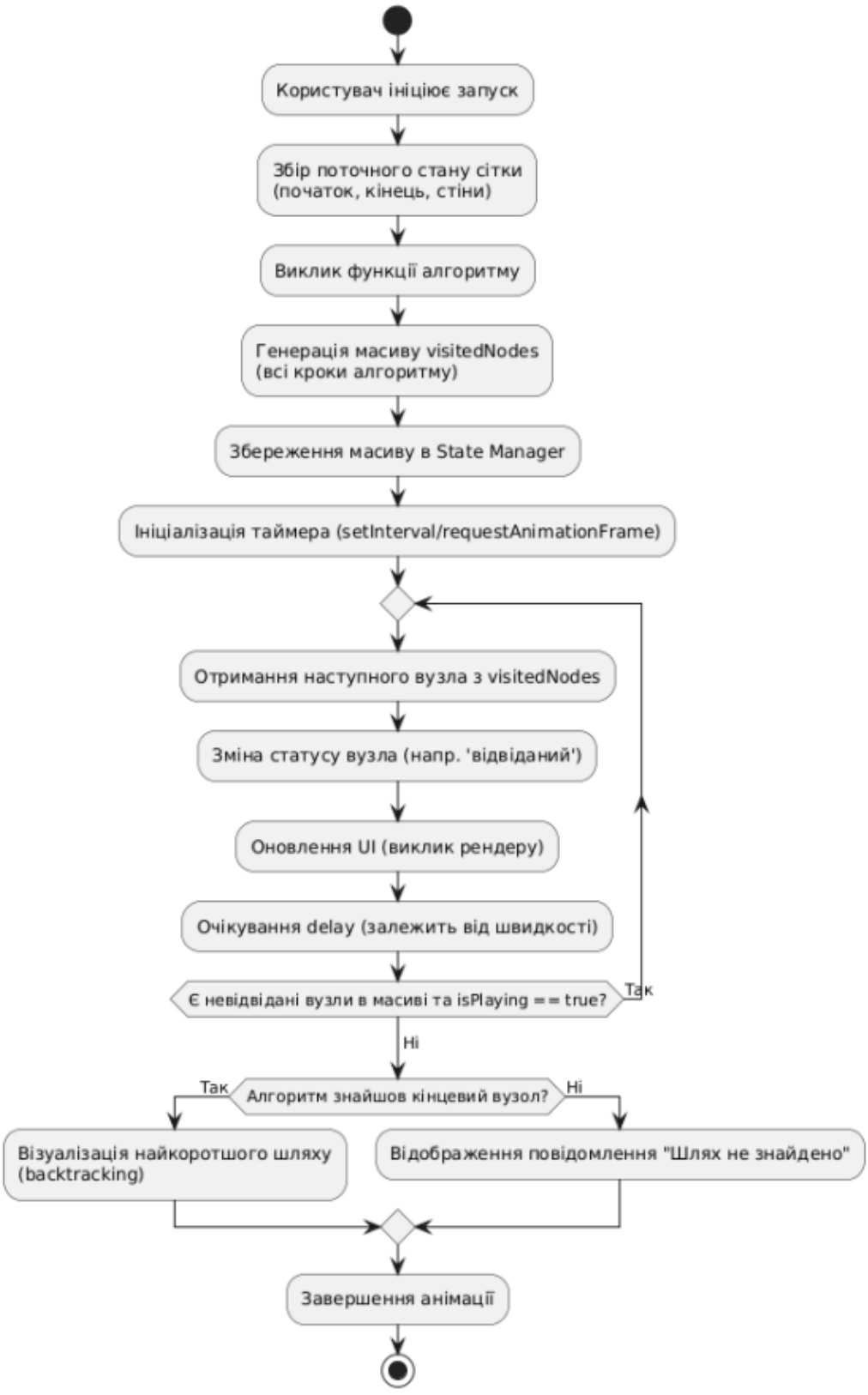

Рисунок 2.7 – UML-діаграма діяльності механізму алгоритмічної анімації

Для візуалізації графових алгоритмів у двовимірному просторі використовується система координат (Grid System). Кожен елемент сітки є

об’єктом, що містить набір метаданих (координати, стан відвідування, вагу, евристичні оцінки для алгоритму A*). Процес взаємодії користувача із сіткою та поведінка алгоритмів формалізовані у вигляді скінченного автомата. Життєвий цикл одного вузла графа проілюстровано на UML-діаграмі станів (рис. 2.8).

Модель вузла графової сітки повинна бути мінімальною, але достатньою для відтворення BFS, DFS, Dijkstra та A*. Надлишкове збереження в кожному вузлі тимчасових UI-прапорців ускладнює скидання стану та повторний запуск. Раціональніше мати стабільну топологію графа окремо від тимчасового стану алгоритму: стіни, ваги та координати належать до inputModel, а ознаки visited, frontier, previous, distance – до snapshot конкретного запуску.

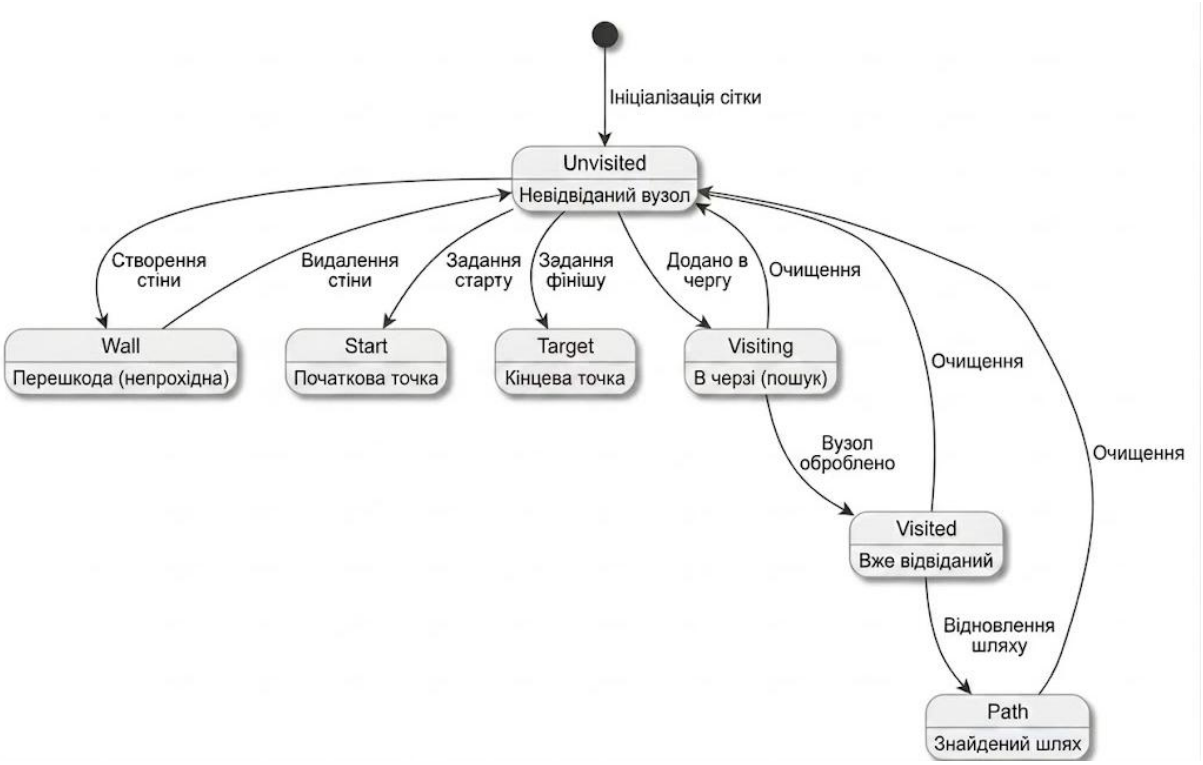

Рисунок 2.8 – UML-діаграма станів вузла графової сітки

Рисунок 2.8 демонструє, що початковим станом будь-якого вузла є «Невідвіданий». Користувач за допомогою миші може малювати перешкоди, переводячи вузол у стан «Стіна». Під час роботи алгоритму пошуку стан вузлів послідовно змінюється на «Відвіданий», а після досягнення цільової точки алгоритм рекурсивно повертається назад (Backtracking), формуючи стан

«Найкоротший Шлях». Завдяки суворому дотриманню цієї моделі станів виключаються логічні помилки (наприклад, спроба алгоритму пройти крізь стіну). Візуалізація графового алгоритму продемонстрована на рис. 2.9.

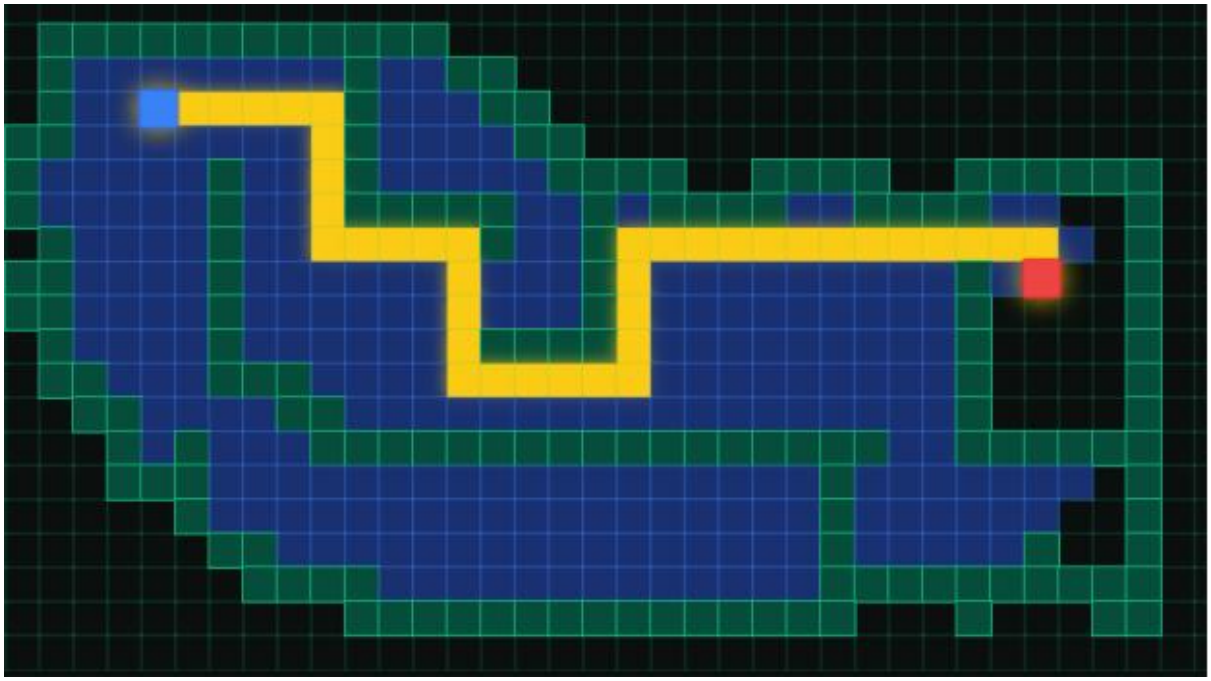

Рисунок 2.9 – Візуалізація алгоритму на графовій сітці

Для підвищення освітньої цінності розробленого програмного забезпечення було впроваджено модуль підсвічування та трасування псевдокоду. Суть цієї функції полягає в синхронному відображенні виконуваного рядка абстрактного алгоритму відповідно до графічної анімації на сітці. З метою збереження мінімального розміру фінального бандлу, замість інтеграції великих бібліотек типу Monaco Editor чи Prism.js, було розроблено власний оптимізований лексичний аналізатор. Його роботу зображено на рис. 2.10. Власний аналізатор є виправданим лише тому, що система працює з обмеженим псевдокодом, а не з повною мовою програмування. Регулярні вирази в JavaScript призначені для пошуку комбінацій символів у рядках [32], однак складні шаблони можуть погіршувати продуктивність і створювати ризик неконтрольованого часу зіставлення. Тому токенізатор має використовувати короткі, передбачувані правила для ключових слів, чисел, операторів і коментарів, а не намагатися аналізувати довільний JavaScript-код.

Як показано на рисунку 2.10, вхідний рядок псевдокоду проходить через токенізатор, заснований на регулярних виразах (Regex). Система класифікує токени (ключові слова, змінні, оператори) і динамічно призначає їм відповідні кольорові класи фреймворку Tailwind CSS. Далі згенеровані елементи рендеряться у віртуальний DOM, а система автоматично прокручує панель коду до активного рядка за допомогою методу scrollIntoView(). Це створює ефект повноцінного середовища розробки (IDE).

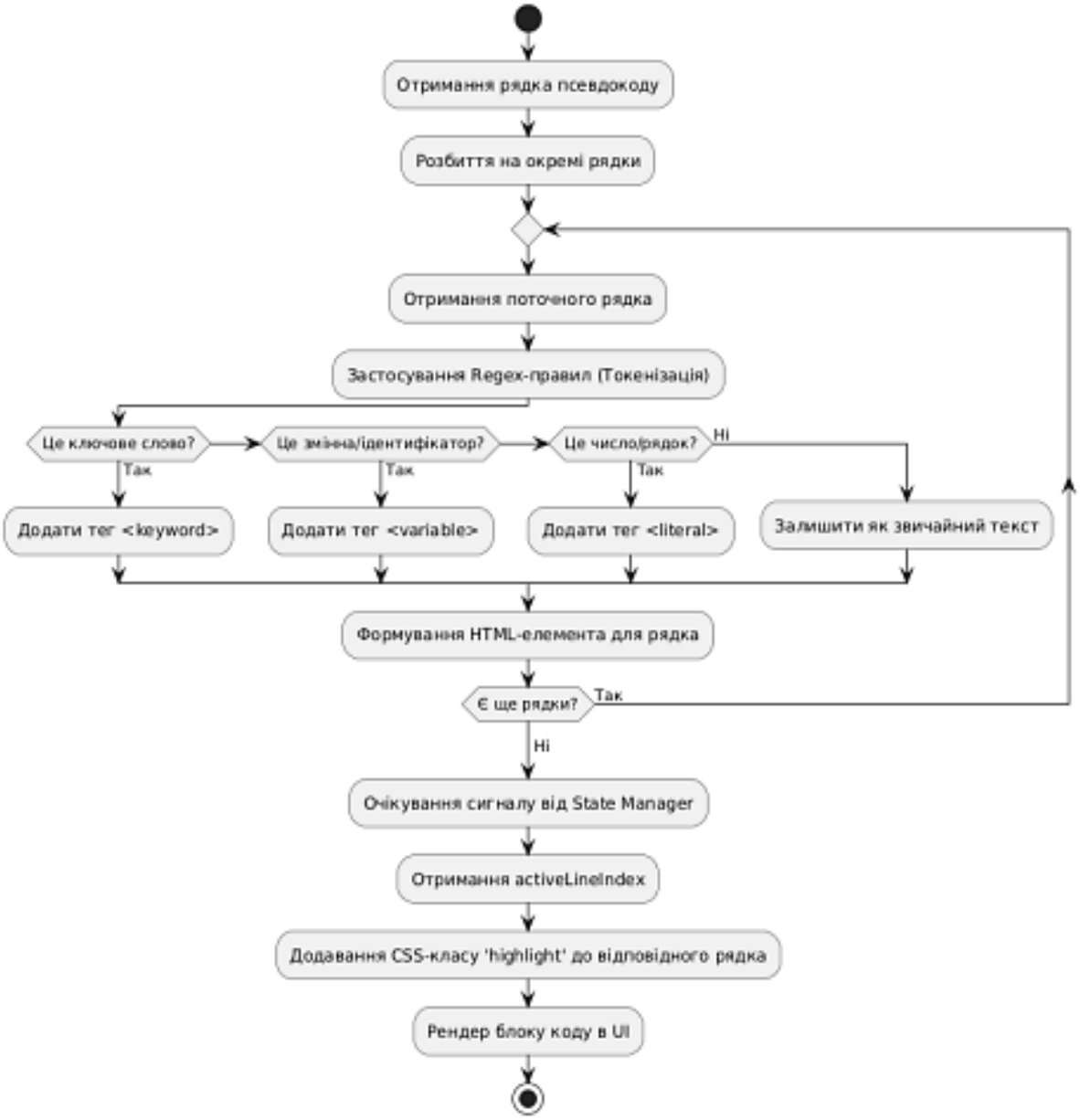

Рисунок 2.11 – UML-діаграма активності модуля лексичного аналізу коду

Автоматичне прокручування активного рядка псевдокоду за допомогою scrollIntoView() є доречним як допоміжна мікроінтеракція: цей метод прокручує контейнер так, щоб вибраний елемент став видимим для користувача [33]. Водночас прокручування не повинно спрацьовувати надто часто, інакше воно відволікатиме від центральної області візуалізації. Практично доцільно викликати його лише при зміні lineId, а не при кожній зміні кольору вузлів.

Отже, в другому розділі кваліфікаційної роботи було детально розглянуто процес проєктування та програмної реалізації вебзастосунку для інтерактивної візуалізації алгоритмів. На початковому етапі сформовано чіткі функціональні та нефункціональні вимоги, а також розроблено специфікацію за допомогою користувацьких історій, що дозволило визначити вектор розвитку інтерфейсу та базовий набір можливостей системи.

Обґрунтовано вибір технологічного стека: використання React.js для побудови компонентної архітектури забезпечило ізоляцію стану відображення від обчислювальної логіки, а застосування Tailwind CSS у комбінації з CSS-анімаціями та механізмами Promise-затримок дозволило реалізувати плавне (60 FPS) відтворення кроків алгоритмів. За допомогою діаграм UML (активності, станів) формалізовано ключові процеси: життєвий цикл вузлів на координатній сітці, механізм управління паузою/відновленням та загальну логічну архітектуру.

Особливу увагу приділено проєктуванню інтерфейсу користувача з урахуванням когнітивних навантажень, що втілилося у строгій колірній диференціації станів і впровадженні модуля лексичного аналізу та трасування коду. Власний оптимізований лексичний аналізатор дозволяє синхронізувати графічні зміни об’єктів із відповідним рядком абстрактного псевдокоду в режимі реального часу. У результаті створено надійну, масштабовану програмну основу, повністю підготовлену до подальшого тестування та інтеграції.

РОЗДІЛ 3 ВПРОВАДЖЕННЯ ТА ТЕСТУВАННЯ ВЕБЗАСТОСУНКУ

3.1 Стратегії та середовища тестування

Для перевірки працездатності розробленого вебзастосунку було обрано стратегію ручного функціонального тестування та тестування інтерфейсу користувача. Метою тестування є перевірка коректності роботи алгоритмів, стабільності анімацій при різних навантаженнях та відповідності інтерфейсу заявленим вимогам. Тестування проводилося в локальному середовищі на базі браузера Google Chrome (версія 120+) із використанням інструментів розробника (Chrome DevTools) для моніторингу продуктивності (FPS, пам’ять).

Тестові випадки в контексті функціонального та системного тестування вебзастосунку зібрано в таблиці 3.1. Під час виконання сценарію ТС-01 здійснювалася ініціалізація нового набору даних за допомогою взаємодії з елементом генерації випадкового масиву. Програма повинна коректно розрахувати пропорції елементів та відобразити стовпці на графічному полотні. У результаті перевірки зафіксовано штатну поведінку системи, масив побудовано без затримок у потоці рендерингу, графічні артефакти відсутні. Тест визнано успішним.

Сценарій ТС-02 передбачав ініціалізацію процесу візуалізації алгоритму сортування. Очікувалась активація покрокового виконання операцій порівняння та перестановки з відповідним колірним виділенням активних елементів. При тестуванні зафіксовано плавну зміну кадрів анімації, логіка підсвічування повністю збігається з кроками роботи алгоритму. Тест визнано успішним.

Метою тесту ТС-03 є перевірка можливості призупинення виконання анімаційного процесу. Користувач здійснює натискання на керуючий елемент паузи в момент активного переміщення об'єктів. Під час виконання зафіксовано миттєву зупинку таймерів та фіксацію поточних позицій елементів у DOM-дереві. Тест визнано успішним.

Тест ТС-04 спрямований на верифікацію функцій зміни масштабу часу візуалізації. При перемиканні коефіцієнтів швидкості автоматизований скрипт

перевіряє зміну інтервалів між кадрами. Фактично зафіксовано пропорційну зміну затримки виконання кроків, анімація адаптується коректно. Тест визнано успішним.

У сценарії ТС-05 перевіряється коректність очищення попереднього стану при виборі іншого алгоритму з випадного списку. Очікується скидання лічильників порівнянь та перебудова робочої області. Тестування підтвердило, що старі процеси успішно знищуються, застосунок готовий до запуску нового алгоритму. Тест визнано успішним.

Перевірка механізму інтернаціоналізації застосунку. Дія полягає у зміні прапорця локалі в шапці сайту. Очікується миттєва підстановка рядкових констант із файлів локалізації для кнопок, заголовків та блоку відображення псевдокоду. Компоненти перемалювалися успішно, поточний стан застосунку при цьому не порушився. Тест визнано успішним.

Таблиця 3.1 – Тест-кейси функціонального та системного тестування

ID	Ддія користувача	Очікуваний результат	Фактичний результат	Статус
ТС-01	Генерація масиву зі 100 випадкових елементів	На екрані (Canvas) миттєво відмальовується 100 стовпців різної висоти	Масив згенеровано, затримок рендерингу немає	Успішно
ТС-02	Запуск алгоритму "Сортування бульбашкою" (Play)	Елементи масиву починають порівнюватися (підсвічування кольором) та мінятися місцями	Анімація відтворюється плавно, кольори змінюються коректно	Успішно
ТС-03	Натискання кнопки "Пауза" під час сортування	Анімація миттєво зупиняється на поточному кроці алгоритму	Стан зафіксовано, таймери призупинено	Успішно
ТС-04	Використання повзунка "Швидкість" (Speed)	Швидкість анімації переміщення елементів збільшується/зменшується	Затримка між кроками змінюється відповідно до повзунка	Успішно
ТС-05	Побудова стін на графовій сітці кліком миші	Вибрані комірки сітки зафарбовуються у колір "перешкоди"	Комірки стають непрохідними для алгоритму (A*, BFS)	Успішно
ТС-06	Зміна мови інтерфейсу (UA -> EN)	Всі підписи кнопок та текст псевдокоду змінюються на англійську	Переклад застосовується без перезавантаження сторінки	Успішно

Автоматизація UI-тестів була проведена засобами тестового фреймворку Playwright. Результати успішного виконання продемонстровано на рис. 3.1.

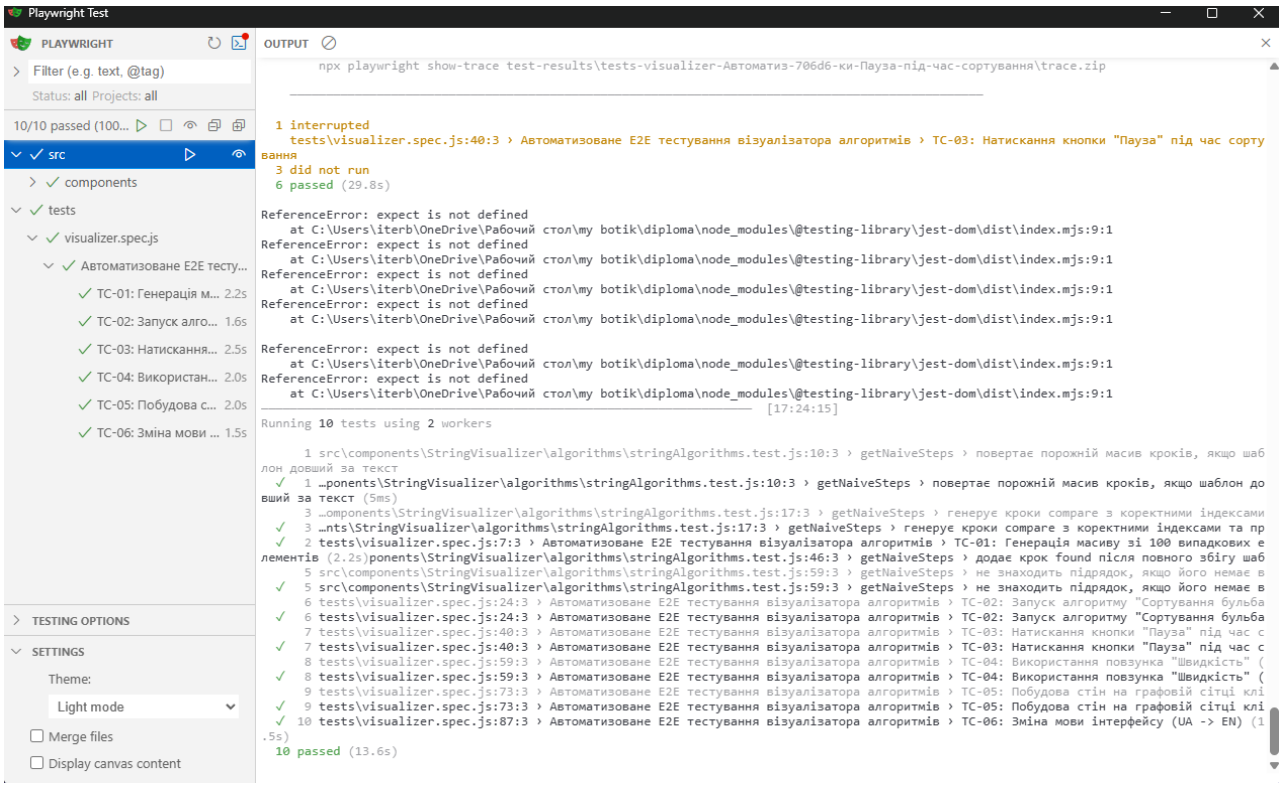

Рисунок 3.1 – Результати виконання автоматизованих E2E-тестів у середовищі Playwright

Подальші тестові випадки зібрано в таблиці 3.2. Вони стосувались переважно роботи графових алгоритмів. У сценарії TC-07 перевірялася здатність евристичного алгоритму A-Star (A*) знаходити оптимальний шлях у надзвичайно складних умовах. На інтерактивній сітці формується багаторівневий лабіринт у вигляді спіралі з кількома хибними відгалуженнями. Програма повинна коректно розрахувати вагу кожної вершини та побудувати маршрут, який точно огинає всі стіни без спроб перетнути непрохідні ділянки по діагоналі. Фактичний результат показав абсолютну математичну точність алгоритму, лінія шляху була відмальована виключно по дозволених комірках сітки. Тест визнано успішним.

Сценарій TC-08 тестує реактивність системи та обробку подій (Event Loop) на зміну вхідних даних після повного завершення попередніх обчислень.

Користувач захоплює курсором маркер фінішної точки та динамічно перетягує його в іншу частину екрана, коли лінію попереднього шляху вже відмальовано. Очікувалось, що система миттєво зніматиме старе виділення та виконає синхронний перерахунок найкоротшого шляху до нових координат у режимі реального часу. Під час тестування перерахунок відбувався без затримок, старі візуальні артефакти успішно стиралися з полотна. Тест визнано успішним.

Таблиця 3.2 – Тестування графового модуля

ID	Опис (Дія користувача)	Очікуваний результат	Фактичний результат
ТС-07	Огинання багаторівневого лабіринту алгоритмом A*	Шлях будується оптимально, огинаючи всі перешкоди без перетину стін	Успішно
ТС-08	Динамічна зміна фінішу після знаходження шляху	Маршрут миттєво перераховується від старту до нової точки	Успішно
ТС-09	Симуляція повної ізоляції (тупик) під час активного пошуку	Алгоритм безпечно зупиняється, виводить помилку без зависання браузера	Успішно
ТС-10	Очищення сітки (Clear) під час виконання асинхронної анімації	Анімація безпечно переривається, об'єкти видаляються без помилок витоку пам'яті	Успішно
ТС-11	Стрес-тест продуктивності при 90% заповненні сітки стінами	Рендеринг обходу відбувається без втрати кадрів (FPS) та деградації UI	Успішно
ТС-12	Спроба переміщення стартового вузла за межі поля (Edge Case)	Вузол блокується на межі сітки, координати не набувають від'ємних значень	Успішно

Перевірка стійкості алгоритму до неможливих умов та захисту від нескінченних циклів здійснювалась у тесті ТС-09. Навколо цільового вузла будується щільне замкнене кільце з перешкод. Після запуску пошуку програма має дослідити весь доступний простір графа, вичерпати чергу пріоритетів та безпечно припинити роботу. У результаті виконання скрипта застосунок успішно зупинив виконання циклу while, уникнув переповнення стека викликів та вивів відповідне інформаційне повідомлення про неможливість досягнення цілі. Тест визнано успішним.

Тестування механізмів безпечного переривання асинхронних операцій та запобігання витокам пам’яті проведено в тестовому випадку ТС-10. Безпосередньо під час активної фази візуалізації пошуку (поширення хвилі)

користувач натискає кнопку повного очищення поля. Система зобов’язана не лише очистити візуальне представлення сітки, але й примусово зупинити всі активні таймери (setTimeout/requestAnimationFrame) та процеси у фоні. Перевірка консолі розробника підтвердила відсутність помилок спроби оновлення демонтованих компонентів, стан застосунку успішно скинуто. Тест визнано успішним.

Стрес-тестування графічного рушія та перевірка ефективності роботи з великими масивами даних (DOM-елементами або Canvas) виконувалось у ТС-11. Сітка розміром понад 1000 комірок максимально заповнювалася стінами, залишаючи лише один вузький звивистий шлях. Мета тесту – переконатися, що обробка такої кількості непрохідних вузлів та рендеринг складної геометрії маршруту не викликає зависання головного потоку браузера. Спостереження підтвердили стабільну частоту кадрів, відгук інтерфейсу залишився на високому рівні. Тест визнано успішним.

Верифікація обробки граничних випадків при взаємодії з елементами управління інтерфейсом була проведено в ході тесту ТС-12. Робот здійснює спробу примусового перетягування маркерів старту та фінішу за межі видимої області інтерактивної сітки. Система повинна мати вбудовані обмеження, які не дозволять координатам вузлів набути від'ємних значень або перевищити максимальні розміри матриці. Зафіксовано коректне блокування об’єктів на крайніх межах поля, внутрішня структура даних графа не була пошкоджена некоректними індексами. Тест визнано успішним. Результати виконання автоматизованих Playwright-тестів показано на рисунку 3.2.

3.2 Оцінювання зручності використання

Для оцінки якості інтерфейсу та придатності використання розробленого програмного продукту застосовано методологію юзабіліті-тестування відповідно до стандарту ISO 9241-11:2018. Оцінювання здійснювалося за трьома ключовими метриками: результативність (effectiveness), ефективність (efficiency) та задоволеність (satisfaction). У тестуванні брала

участь фокус-група з $R = 5$ респондентів із різним рівнем технічної підготовки. Кожному користувачеві було запропоновано виконати $N = 3$ типових завдання, які охоплюють основний функціонал застосунку:

– налаштування вхідних даних та вибір параметрів: користувач повинен самостійно задати початкові умови (наприклад, ввести власний масив елементів або згенерувати набір даних) та обрати зі списку конкретний алгоритм для подальшої роботи;

– управління процесом візуалізації: користувач має запустити виконання алгоритму, перевірити роботу інтерактивних елементів керування (натиснути паузу, змінити швидкість анімації, перейти на крок вперед/назад) та довести процес до логічного завершення;

– аналіз отриманих результатів: після завершення візуалізації користувач повинен знайти блок із підсумковою інформацією (статистика виконання, кількість ітерацій, оцінка складності тощо) та очистити робочу область для нового запуску.

–

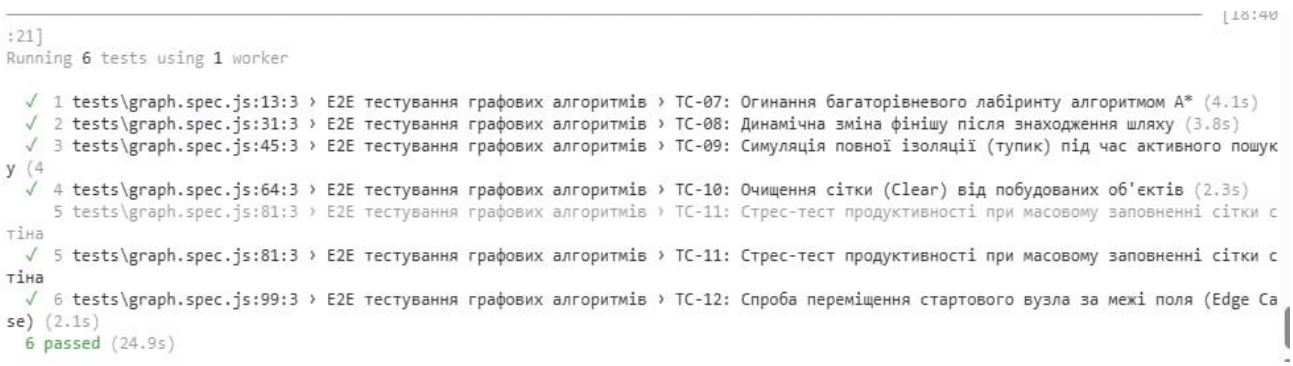

Рисунок 3.2 – Результати виконання автоматизованих E2E-тестів модуля графових алгоритмів у середовищі Playwright

Результативність – це ступінь реалізації запланованої діяльності та досягнення запланованих результатів. Загальну результативність (E) обчислено за формулою:

$$E = \frac{\sum_{i=1}^{N} \sum_{j=1}^{R} n_{ij}}{R \cdot N} \times 100\%$$

де n_{ij} — результат виконання і-го сценарію j-м користувачем (дорівнює 1, якщо сценарій успішно завершено; інакше – 0). Результати виконання завдань користувачами систематизовано в табл. 3.3.

Таблиця 3.3 – Результати виконання завдань (результативність)

Респондент	Завдання 1	Завдання 2	Завдання 3	Успішних
Користувач 1	1	1	1	3
Користувач 2	1	1	1	3
Користувач 3	1	0	1	2
Користувач 4	1	1	1	3
Користувач 5	1	1	1	3
Разом	5	4	5	14

Загальна результативність продукту:

$$E = \frac{14}{5 \cdot 3} \times 100\% = \frac{14}{15} \times 100\% \approx 93{,}3\%.$$

Відповідно до отриманих результатів, вона оцінюється як «відмінна», оскільки переважна більшість респондентів успішно впоралася з поставленими завданнями самостійно.

Ефективність визначається ресурсами, які користувач витрачає з метою забезпечення точної та повної реалізації цілей. У цьому випадку ключовим ресурсом є час виконання завдання. Часову відносну ефективність (P) обчислено за формулою:

$$P = \frac{\sum_{i=1}^{N} \sum_{j=1}^{R} (n_{ij} \cdot t_{ij})}{\sum_{i=1}^{N} \sum_{j=1}^{R} t_{ij}} \times 100\%,$$

де t_{ij} – час (у секундах), витрачений j-м респондентом на i-те завдання; n_{ij} – коефіцієнт успішності виконання завдання (1 або 0). Оскільки у чисельнику формули час множиться на коефіцієнт успішності n_{ij}, час невдалої спроби користувача 3 під час виконання Завдання 2 (60 секунд) не враховується як «корисний», проте додається до загальних витрат часу в знаменнику. Часові показники респондентів наведено в табл. 3.4.

Таблиця 3.4 – Час виконання завдань (ефективність)

Респондент	Завдання 1, с	Завдання 2, с	Завдання 3, с	Загальний час, с
Користувач 1	15	30	12	57
Користувач 2	12	25	10	47
Користувач 3	18	60 (невдача)	14	92
Користувач 4	14	28	11	53
Користувач 5	16	32	15	63
Разом	75	175	62	312

Розрахунок часової відносної ефективності:

$$P = \frac{312 - 60}{312} \times 100\% = \frac{252}{312} \times 100\% \approx 80{,}7\%$$

Отриманий показник свідчить про високу швидкість взаємодії користувачів із компонентами інтерфейсу та логічно оптимізовану побудову екранів застосунку.

Задоволеність відображає суб'єктивне сприйняття продукту користувачем та відсутність дискомфорту під час роботи. Оцінювання здійснювалося методом формалізованого анкетування після завершення практичних тестів (табл. 3.5). Респондентам було запропоновано оцінити низку тверджень (наприклад, «Інтерфейс зрозумілий та інтуїтивний») за 5-бальною шкалою (від 1 – абсолютно не згоден, до 5 – повністю згоден).

Таблиця 3.5 – Оцінки респондентів за 5-бальною шкалою (задоволеність)

Респондент	Зрозумілість	Зручність	Привабливість	Враження	Середній
Користувач 1	5	4	5	5	4.75
Користувач 2	5	5	5	4	4.75
Користувач 3	4	4	5	4	4.25
Користувач 4	5	5	5	5	5.00
Користувач 5	5	5	4	5	4.75
Разом	4.8	4.6	4.8	4.6	4.70

Середній бал задоволеності обчислюється як середнє арифметичне оцінок за всіма критеріями:

$$\textit{Середній бал} = \frac{4{,}8 + 4{,}6 + 4{,}8 + 4{,}6}{4} = 4{,}70$$

Підсумковий показник задоволеності склав 4,70 з 5 можливих. Це свідчить про високий рівень ергономічності системи, візуальну привабливість дизайну та загальну комфортність взаємодії користувача із розробленим застосунком.

3.3 Розгортання застосунку та налаштування CI/CD

Фінальним етапом життєвого циклу розробки стало розгортання вебзастосунку у хмарному середовищі. Оскільки проєкт є суто клієнтським, для хостингу було обрано платформу Vercel. Її перевагами є глобальна мережа доставки контенту (CDN Edge Network), що забезпечує мінімальну затримку при завантаженні сторінки з будь-якої точки світу, та нативна підтримка проєктів, зібраних за допомогою Vite.

Для забезпечення автоматизації процесу оновлення застосунку було налаштовано пайплайн CI/CD (Continuous Integration / Continuous Deployment). Загальну послідовність цього процесу, яка наочно демонструє етапи проходження коду від локального середовища до релізу, зображено на рис. 3.3.

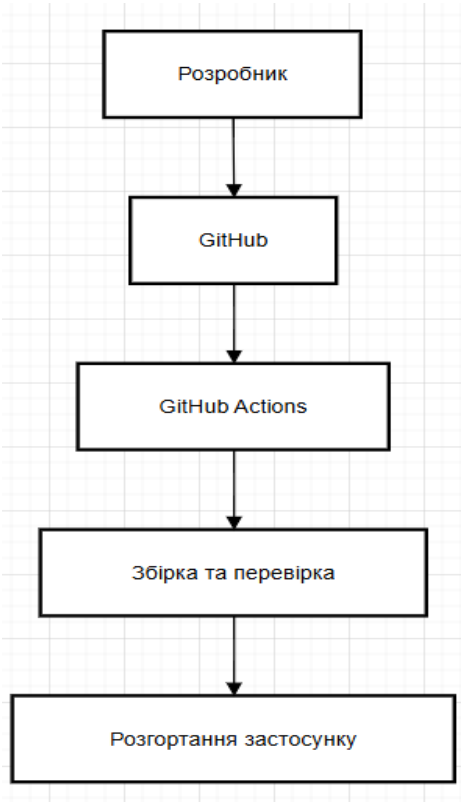

Рисунок 3.3 – Схема процесу безперервної інтеграції та розгортання (CI/CD)

Як видно з наведеної схеми, життєвий цикл оновлення починається з розробника, який вносить зміни та відправляє їх у централізований репозиторій на базі GitHub. Платформа Vercel інтегрована безпосередньо з гілкою «main» репозиторію.

Це означає, що при кожному коміті (git push) до основної гілки автоматично спрацьовують тригери (зокрема за допомогою GitHub Actions або внутрішніх засобів CI/CD платформи Vercel), які ініціюють етап збірки та перевірки. На цьому кроці система виконує складання (build) проєкту, запускає лінтери (ESLint) та перевіряє код на наявність помилок.

У разі успішного проходження всіх автоматизованих перевірок, пайплайн переходить до фінального етапу – розгортання застосунку на робочому домені. Такий підхід повністю нівелює ризики людського фактора під час деплою та дозволяє миттєво доставляти оновлення користувачам.

Практичний результат роботи налаштованого пайплайну можна спостерігати в панелі управління обраного хмарного провайдера. Після первинного завантаження коду на GitHub, платформа автоматично ініціювала збірку та деплой проєкту. Результат успішного розгортання вебзастосунку для інтерактивної візуалізації алгоритмів представлено на рисунку 3.4.

Як видно з панелі управління (дашборду) Vercel, після завершення процесу збірки застосунок отримує статус «Ready» (Готово). Платформа автоматично генерує робочі доменні імена для доступу до сайту та фіксує хеш коміту (у цьому випадку – `initial commit`), який став тригером для розгортання. Це єдине вікно надає повний контроль над станом продакшн-середовища.

Надалі буде продемонстровано процес оновлення застосунку. За такого сценарію було додано інтерактивний перемикач світлої/темної теми інтерфейсу користувача та виправлено низку мінорних програмних помилок. На першому кроці відбувається внесення локальних змін та їх фіксація. Після локального тестування ці зміни фіксуються в системі контролю версій Git з відповідним повідомленням (наприклад, feat: add theme toggler and fix bugs) та відправляються у віддалений репозиторій за допомогою команди git push (рис. 3.5).

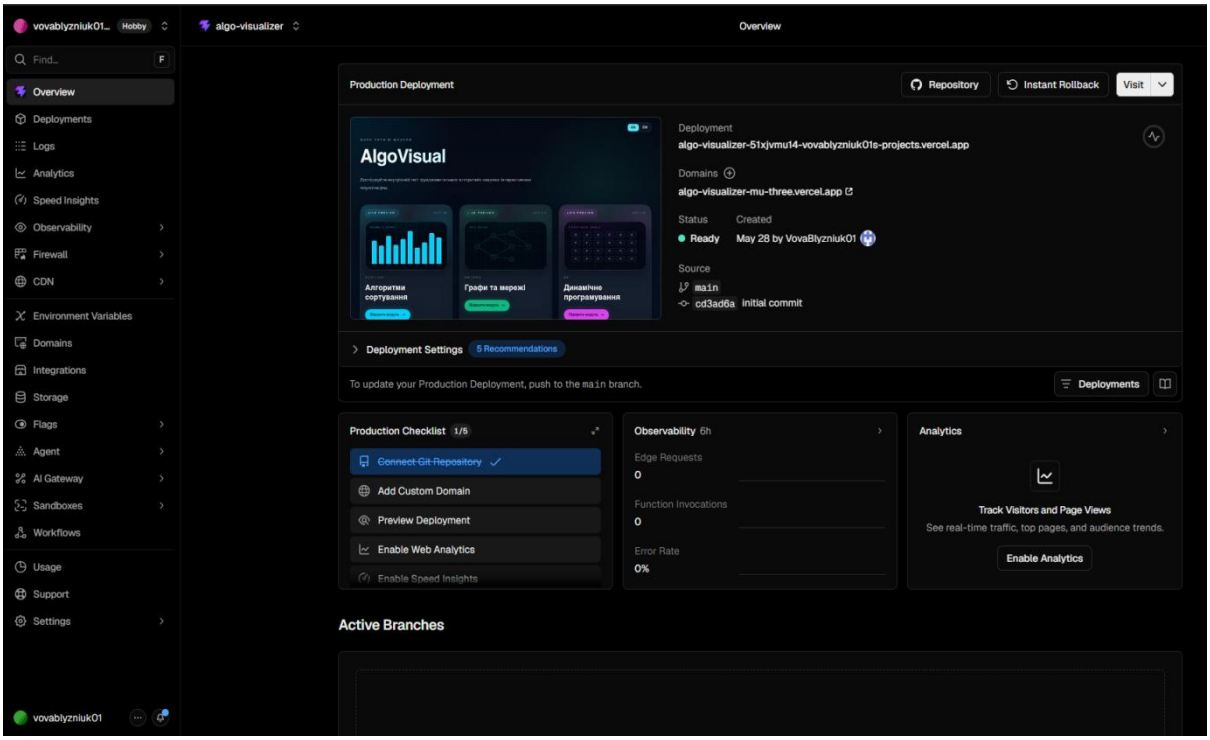

Рисунок 3.4 – Панель управління платформою Vercel із деталями успішного розгортання застосунку

```
PS C:\Users\iterb\OneDrive\Рабочий стол\my botik\diploma> git add .
warning: in the working copy of 'src/test-utils/stepReporter.js', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'src/test-utils/styleMock.cjs', LF will be replaced by CRLF the next time Git touches it
PS C:\Users\iterb\OneDrive\Рабочий стол\my botik\diploma> git commit -m "update UI"
[main 02ffc27] update UI
 31 files changed, 781 insertions(+), 18 deletions(-)
 create mode 100644 babel.config.cjs
 create mode 100644 inject_theme.py
 create mode 100644 jest.config.cjs
 create mode 100644 jest.reporter.cjs
 create mode 100644 jest.setup.cjs
 create mode 100644 src/components/GraphVisualizer/Node.optimization.jsx
 create mode 100644 src/components/Hub/Hub.cards.test.jsx
 create mode 100644 src/components/StringVisualizer/algorithms/stringAlgorithms.test.js
 create mode 100644 src/components/ThemeToggle/ThemeToggle.jsx
 create mode 100644 src/context/ThemeContext.jsx
 create mode 100644 src/test-utils/stepReporter.js
 create mode 100644 src/test-utils/styleMock.cjs
 create mode 100644 "test-results/tests-graph-E2E-\321\202\320\265\321\201\321\202\321\203\320\262\320\260\320\275\320\275\321\217-55b45-
203-\320\277\321\226\321\201\320\273\321\217-\320\267\320\275\320\260\321\205\320\276\320\264\320\266\320\265\320\275\320\275\321\217-\32
\321\205\321\203/trace.zip"
 create mode 100644 "test-results/tests-graph-E2E-\321\202\320\265\321\201\321\202\321\203\320\262\320\260\320\275\320\275\321\217-86c65-
260-\320\267\320\260-\320\274\320\265\320\266\321\226-\320\277\320\276\320\273\321\217-Edge-Case-/trace.zip"
 create mode 100644 "test-results/tests-graph-E2E-\321\202\320\265\321\201\321\202\321\203\320\262\320\260\320\275\320\275\321\217-c047d-
276-\320\273\320\260\320\261\321\226\321\200\320\270\320\275\321\202\321\203-\320\260\320\273\320\263\320\276\321\200\320\270\321\202\320
A-/trace.zip"
 create mode 100644 "test-results/tests-graph-E2E-\321\202\320\265\321\201\321\202\321\203\320\262\320\260\320\275\320\275\321\217-cd2a9-
20\264-\320\277\320\276\320\261\321\203\320\264\320\276\320\262\320\260\320\275\320\270\321\205-\320\276\320\261-\321\224\320\272\321\202
e.zip"
 create mode 100644 "test-results/tests-graph-E2E-\321\202\320\265\321\201\321\202\321\203\320\262\320\260\320\275\320\275\321\217-d596e-
\267\320\260\320\277\320\276\320\262\320\275\320\265\320\275\320\275\321\226-\321\201\321\226\321\202\320\272\320\270-\321\201\321\202\32
\320\274\320\270/trace.zip"
 create mode 100644 "test-results/tests-graph-E2E-\321\202\320\265\321\201\321\202\321\203\320\262\320\260\320\275\320\275\321\217-e109d-
\277\321\226\320\264-\321\207\320\260\321\201-\320\260\320\272\321\202\320\270\320\262\320\275\320\276\320\263\320\276-\320\277\320\276\3
2\321\203/trace.zip"
 create mode 100644 tests/graph.spec.js
 create mode 100644 tests/visualizer.spec.js
PS C:\Users\iterb\OneDrive\Рабочий стол\my botik\diploma> git push origin main
Enumerating objects: 75, done.
Counting objects: 100% (75/75), done.
Delta compression using up to 12 threads
Compressing objects: 100% (46/46), done.
Writing objects: 100% (54/54), 21.37 MiB | 5.74 MiB/s, done.
Total 54 (delta 13), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (13/13), completed with 13 local objects.
To https://github.com/VovaBlyzniuk01/algo-visualizer
   cd3ad6a..02ffc27  main -> main
```

Рисунок 3.5 – Фіксація змін та відправка коду до репозиторію GitHub

На наступному кроці здійснюється автоматичне збирання на стороні Vercel. Одразу після потрапляння нового коду в головну гілку, платформа Vercel перехоплює подію та автоматично запускає процес генерації нової версії застосунку. На панелі управління (рис. 3.6) видно, що поточний статус розгортання змінився на «Building» для коміту з назвою «update UI». Це підтверджує миттєву реакцію хмарної інфраструктури на зміни в системі контролю версій.

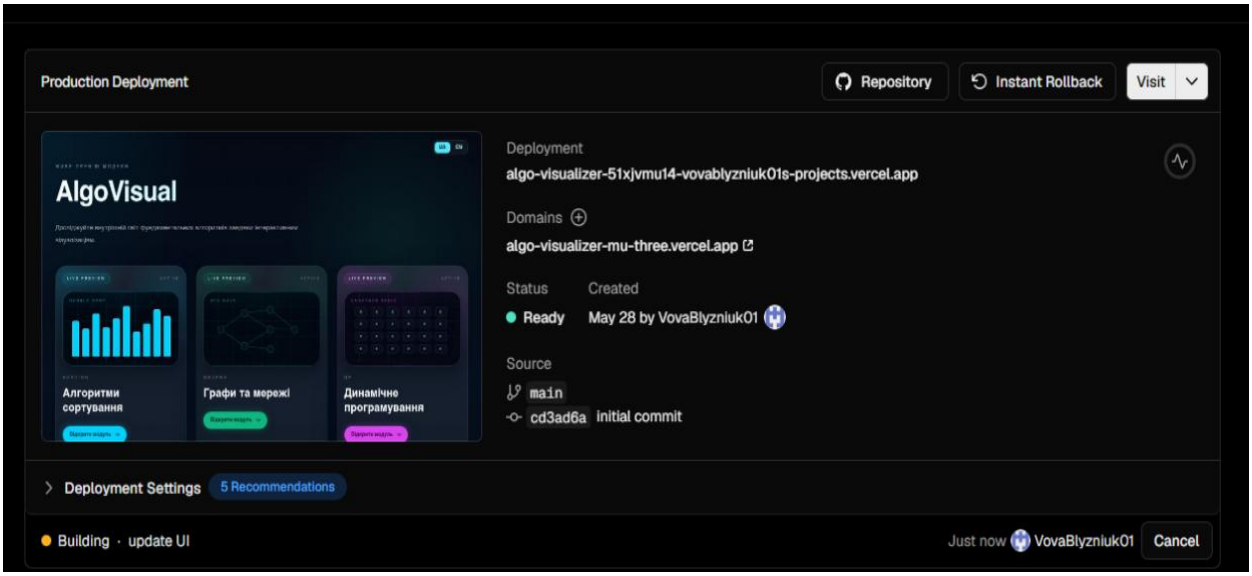

Рисунок 3.6 – Процес автоматичної збірки нової версії застосунку на платформі Vercel (статус Building)

Після завершення збірки статус змінюється на «Ready», і оновлена версія застосунку AlgoVisual стає доступною для користувачів. На рисунку 3.7 продемонстровано результати впровадження нової функціональності – інтерактивного перемикача колірних тем інтерфейсу. Завдяки налаштованому пайплайну CI/CD, користувачі отримали можливість зручно перемикатися між темним (піктограма для переходу у світлий режим) та світлим (піктограма для повернення у темний режим) оформленням. Оновлення відбулося без жодного часу простою (zero-downtime deployment), що гарантує безперервну доступність сервісу.

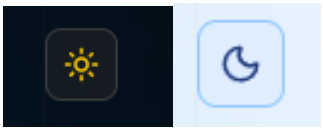

Рисунок 3.7 – Реалізований елемент керування темами інтерфейсу після автоматичного розгортання

Загалом, у межах системного та функціонального тестування було перевірено ключові сценарії роботи застосунку та окремо підтверджено, що застосунок зберігає працездатність під час роботи з великими наборами даних. Оцінка якості та юзабіліті дала змогу перевірити програмний продукт з позиції кінцевого користувача. За результатами виконання практичних завдань загальна результативність становила 93,3%, що свідчить про високий рівень досягнення користувачами поставлених цілей. Часова відносна ефективність склала 80,7%, що є прийнятним показником для інтерактивного навчального інструменту, оскільки частина часу витрачалася не лише на виконання дій, а й на усвідомлення логіки алгоритмів. Середній бал задоволеності користувачів становив 4,7 з 5, що підтверджує зручність, зрозумілість і позитивне сприйняття інтерфейсу.

Фінальним етапом стало розгортання застосунку у хмарному середовищі та налаштування процесу безперервної інтеграції й розгортання. Використання Vercel як платформи для хостингу клієнтського SPA-застосунку забезпечило швидке завантаження, автоматизовану збірку та зручне оновлення робочої версії після змін у репозиторії. Налаштований CI/CD-підхід зменшує ризик помилок під час ручного розгортання та забезпечує контрольовану доставку оновлень користувачам.

Отже, результати третього розділу підтверджують, що розроблений вебзастосунок є функціонально завершеним, працездатним і придатним до використання як допоміжний інструмент для вивчення алгоритмів і структур даних. Проведене тестування засвідчило відповідність системи основним вимогам, а результати оцінювання юзабіліті підтвердили її зручність для користувачів.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено сучасний інтерактивний вебзастосунок для візуалізації роботи алгоритмів. Проведений аналіз предметної області підтвердив необхідність створення такого інструменту для підвищення ефективності освітнього процесу під час вивчення фундаментальних алгоритмів та структур даних. Під час роботи було успішно вирішено такі завдання:

1. Здійснено проєктування архітектури застосунку на базі сучасних вебтехнологій. Обґрунтовано вибір бібліотеки React.js та функцій-генераторів мови JavaScript для реалізації покрокової анімації обчислювальних процесів.

2. Реалізовано зручний та інтуїтивно зрозумілий користувацький інтерфейс (UI) з урахуванням сучасних принципів UX-дизайну. Створено панель керування, що нагадує медіаплеєр і забезпечує повний контроль над процесом візуалізації (запуск, пауза, зміна швидкості, покрокове виконання).

3. Запрограмовано та інтегровано ключові алгоритми сортування (Bubble Sort, Quick Sort, Merge Sort) із забезпеченням наочної колірної та просторової індикації кожного кроку їхньої роботи.

4. Проведено комплексне тестування розробленого програмного продукту, яке включало як перевірку коректності алгоритмів, так і оцінку юзабіліті-метрик інтерфейсу. Застосунок продемонстрував високу продуктивність (60 FPS) та стабільність роботи.

5. Забезпечено автоматизацію процесу розгортання застосунку (CI/CD) з використанням платформи Vercel, що дозволяє швидко та безперебійно доставляти оновлення користувачам.

Отримані результати підтверджують, що розроблений вебзастосунок повністю відповідає поставленим вимогам і може ефективно використовуватися як допоміжний інструмент під час вивчення програмування. Проведене тестування також дозволило виявити напрями подальшого вдосконалення. Зокрема, перспективними є розширення набору алгоритмів, поглиблення пояснювального шару для складніших структур даних, додавання збереження

історії виконання, удосконалення аналітики навчального прогресу та розширення сценаріїв тестування на мобільних пристроях. Водночас ці напрями не є критичними для базової працездатності системи, а належать до подальшого розвитку програмного продукту.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 4th ed. Cambridge, MA : The MIT Press, 2022. 1312 p.
2. Naps T. L., Rößling G., Almstrum V., Dann W., Fleischer R., Hundhausen C., Korhonen A., Malmi L., McNally M., Rodger S., Velázquez-Iturbide J. Á. Exploring the role of visualization and engagement in computer science education. Proceedings of the Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education. 2002. P. 131–152. DOI: 10.1145/960568.782998.
3. Hundhausen C. D., Douglas S. A., Stasko J. T. A Meta-Study of Algorithm Visualization Effectiveness. Journal of Visual Languages & Computing. 2002. Vol. 13, Issue 3. P. 259–290. DOI: 10.1006/jvlc.2002.0237.
4. Sorva J., Karavirta V., Malmi L. A Review of Generic Program Visualization Systems for Introductory Programming Education. ACM Transactions on Computing Education. 2013. Vol. 13, No. 4. Article 15. P. 1–64. DOI: 10.1145/2490822.
5. Sweller J. Cognitive Load During Problem Solving: Effects on Learning. Cognitive Science. 1988. Vol. 12, Issue 2. P. 257–285. DOI: 10.1207/s15516709cog1202_4.
6. Mayer R. E. Multimedia Learning. 3rd ed. Cambridge : Cambridge University Press, 2020. 460 p.
7. Munzner T. A Nested Model for Visualization Design and Validation. IEEE Transactions on Visualization and Computer Graphics. 2009. Vol. 15, No. 6. P. 921–928. DOI: 10.1109/TVCG.2009.111.
8. Bostock M., Ogievetsky V., Heer J. D^3 Data-Driven Documents. IEEE Transactions on Visualization and Computer Graphics. 2011. Vol. 17, No. 12. P. 2301–2309. DOI: 10.1109/TVCG.2011.185.
9. Scalable Vector Graphics (SVG) 2. W3C Candidate Recommendation. URL: https://www.w3.org/TR/SVG2/ (дата звернення: 13.06.2026).

10. WebGL: 3D Graphics for the Web. Khronos Group. URL: https://www.khronos.org/webgl/ (дата звернення: 13.06.2026).
11. Shneiderman B. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. Proceedings of the IEEE Symposium on Visual Languages. 1996. P. 336–343. DOI: 10.1109/VL.1996.545307.
12. Keim D. A. Information Visualization and Visual Data Mining. IEEE Transactions on Visualization and Computer Graphics. 2002. Vol. 8, No. 1. P. 1–8. DOI: 10.1109/2945.981847.
13. Ware C. Information Visualization: Perception for Design. 4th ed. Cambridge : Morgan Kaufmann / Elsevier, 2020. 560 p.
14. Purchase H. C. Which Aesthetic has the Greatest Effect on Human Understanding? Graph Drawing: 5th International Symposium, GD 1997. Lecture Notes in Computer Science. Vol. 1353. Berlin : Springer, 1997. P. 248–261. DOI: 10.1007/3-540-63938-1_67.
15. Ware C., Purchase H., Colpoys L., McGill M. Cognitive Measurements of Graph Aesthetics. Information Visualization. 2002. Vol. 1, No. 2. P. 103–110. DOI: 10.1057/palgrave.ivs.9500013.
16. Sedgewick R., Wayne K. Algorithms. 4th ed. Boston : Addison-Wesley, 2011. 976 p.
17. Dijkstra E. W. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik. 1959. Vol. 1. P. 269–271. DOI: 10.1007/BF01386390.
18. ISO 9241-11:2018. Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts. Geneva : International Organization for Standardization, 2018. 31 p.
19. ISO/IEC 25010:2023. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. Geneva : International Organization for Standardization, 2023.
20. React Documentation. URL: https://react.dev/ (дата звернення: 03.06.2026).

21. MDN Web Docs. Iterators and generators. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_generators (дата звернення: 03.06.2026).
22. Flanagan D. JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language. 7th ed. Sebastopol : O'Reilly Media, 2020. 706 p.
23. Freeman A. Pro React 18. Berkeley : Apress, 2022. 737 p. DOI: 10.1007/978-1-4842-8082-3.
24. VisuAlgo: visualising data structures and algorithms through animation. URL: https://visualgo.net/ (дата звернення: 03.06.2026).
25. Algorithm Visualizer. GitHub repository. URL: https://github.com/algorithm-visualizer/algorithm-visualizer (дата звернення: 03.06.2026).
26. Data Structure Visualizations. URL: https://www.cs.usfca.edu/~galles/visualization/Algorithms.html (дата звернення: 03.06.2026).
27. Vite Documentation. URL: https://vite.dev/guide/ (дата звернення: 03.06.2026).
28. Zustand Documentation. URL: https://zustand.docs.pmnd.rs/ (дата звернення: 03.06.2026).
29. Tailwind CSS Documentation. URL: https://tailwindcss.com/docs (дата звернення: 03.06.2026).
30. Motion for React Documentation. URL: https://motion.dev/docs/react (дата звернення: 03.06.2026).
31. Web Content Accessibility Guidelines (WCAG) 2.2. W3C Recommendation. URL: https://www.w3.org/TR/WCAG22/ (дата звернення: 03.06.2026).
32. MDN Web Docs. Regular expressions. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions (дата звернення: 03.06.2026).
33. MDN Web Docs. Element: scrollIntoView() method. URL: https://developer.mozilla.org/en-US/docs/Web/API/Element/scrollIntoView (дата звернення: 03.06.2026).

ДОДАТКИ

Додаток А – Посилання на репозиторій та розгорнутий програмний продукт

Програмний код розробленого проєкта розташований на платформі GitHub за адресою https://github.com/VovaBlyzniuk01/algo-visualizer. Розгорнутий на платформі Vercel вебсайт доступний за посиланням https://algo-visualizer-mu-three.vercel.app/.