ПРИРОДНИЧІ ТА ТЕХНІЧНІ НАУКИ

УДК 004:005:51

ОПТИМІЗАЦІЯ ДАНИХ ТА ПОБУДОВА ІНТЕРФЕЙСІВ У ВЕБДИЗАЙНІ ІЗ ВИКОРИСТАННЯМ АЛГОРИТМІЧНИХ СТРУКТУР

С. І. Заворотна, Н. Р. Веселовська

Анотація. У статті розглянуто роль алгоритмічних структур і структур даних у процесі вебдизайну та веброзробки. Проаналізовано застосування алгоритмів сортування, пошуку, оптимізації та побудови графів у побудові користувацьких інтерфейсів, навігації сайту та обробці даних. Окрему увагу приділено структурі Document Object Model (DOM) та алгоритмам, які забезпечують ефективну роботу з елементами сторінки. Показано, як алгоритмічний підхід впливає на продуктивність вебресурсів і досвід користувача.

Ключові слова: алгоритми, структури даних, вебдизайн, інтерфейс, оптимізація, DOM, графи.

Вступ. У сучасному веб-дизайні алгоритмічні структури відіграють ключову роль, адже за допомогою алгоритмів здійснюється обробка, сортування, пошук та візуалізація інформації. Вебсайт сьогодні – це не просто набір сторінок, а складна система, у якій дані постійно оновлюються та оптимізуються. Саме алгоритми дозволяють забезпечити швидкодію, інтуїтивність і динамічність вебінтерфейсів.

Метою статті є аналіз ролі алгоритмічних структур у вебдизайні та аналіз способів оптимізації даних і побудови ефективних користувацьких інтерфейсів на основі алгоритмів і структур даних.

Основна частина. Алгоритмічні структури лягли в основу сучасного підходу до вебдизайну й веброзробки: інтерфейс – це не лише графічне відображення даних, а й сукупність алгоритмів, що забезпечують їх обробку, доставку й відображення. Щоби зрозуміти, як ці алгоритми впливають на роботу інтерфейсу, необхідно розглянути як класичні алгоритми сортування й пошуку, так і структури даних – дерева, графи, хеш-таблиці, а також специфічні техніки оптимізації й управління подіями.

Сортування – одна з найпоширеніших операцій у вебдодатках. QuickSort є часто обраним алгоритмом через свою простоту й середню часову складність O(n log n), але він має гірший випадок $O(n^2)$ у разі невдалого вибору опорного елемента; тому в практичних реалізаціях часто використовують випадковий вибір піводного елемента або «median-of-three» для зниження ймовірності деградації. MergeSort забезпечує гарантовану часову складність O(n log n) у всіх випадках і є стабільним (зберігає порядок рівних елементів), проте потребує додаткової пам’яті O(n), що робить його менш привабливим для середовищ з обмеженою пам’яттю. HeapSort також гарантує O(n log n) і виконується in-place (O(1) додаткової пам’яті), але не є стабільним. BubbleSort має часову складність $O(n^2)$ і зазвичай не застосовується на реальних обсягах даних – його місце навчальне. У вебконтексті вибір алгоритму сортування залежить від очікуваних обсягів та вимог до стабільності: для невеликих локальних списків (до сотень елементів) різниця в практичній швидкості незначна, але для каталогів із тисячами товарів сортування краще виконувати на сервері або у WebWorker, а на клієнті застосовувати ефективні алгоритми (або покладатися на оптимізовані вбудовані реалізації рушія JavaScript). Варто також пам’ятати, що конкретна реалізація сортування в середовищі виконання (рушію браузера) може бути оптимізованою або стабільною / нестабільною – тому варто перевіряти поведінку на цільових платформах.

Пошук даних у вебінтерфейсах має низку реалізацій залежно від структури даних. Лінійний пошук (linear search) має O(n) і підходить для невеликих масивів або коли дані не впорядковані. Бінарний пошук має O(log n) і вимагає відсортованого набору; він добре підходить для автодоповнення в полях пошуку, якщо підтримується динамічне впорядкування. Для префіксного пошуку й автокомпліту корисні Trie-структури: вони дозволяють знаходити всі слова з певним префіксом за час, пропорційний довжині запиту O(m), де m – довжина ключа, але потребують більше пам’яті, що є компромісом між швидкістю та обсягом збереження. Хеш-таблиці (вбудовані Map / Object у JavaScript) забезпечують амортизований доступ O(1) і

широко використовуються для кешування й індексування, але в гіршому випадку (колізії) час доступу може деградувати.

У фільтрації даних і комбінованих запитах застосовують деревоподібні структури та пошукові дерева. Наприклад, B-дерева або індекси на сервері (у СУБД) дозволяють швидко виконувати діапазонні запити та фільтри за кількома полями. Вебінтерфейси часто комбінують локальні фільтри (для миттєвих UI-відповідей) і серверні запити (коли обсяг даних великий), що мінімізує навантаження на мережу й покращує відгук інтерфейсу. Водночас потрібно враховувати: прості жадібні стратегії фільтрації можуть бути не оптимальними для глобальної оптимізації відбору (існують ситуації, коли greedy-підхід дає субоптимальні результати – класичний приклад це завдання про рюкзак з невідповідними монетами), тому для складних умов варто використовувати повні алгоритми пошуку або індекси.

Document Object Model (DOM) є фундаментальною структурою для вебсторінок: вона відображає HTML як ієрархічне дерево вузлів, і робота з DOM реалізується через обходи та маніпуляції вузлами. Алгоритми обходу (DFS, BFS) мають лінійну часову складність O(n) від кількості вузлів і застосовуються як для повного проходу, так і для селективних оновлень. Проте прямі маніпуляції з DOM можуть бути дорогими з точки зору перерахунку стилів та повторного рендерингу, тому сучасні бібліотеки використовують проміжні рівні оптимізації. React, наприклад, застосовує концепцію Virtual DOM і диф-алгоритми (diffing), які порівнюють попередні і нові віртуальні дерева та обчислюють мінімальний набір змін. Теоретично наївне порівняння дерев може мати квадратичну складність, але практичні реалізації використовують евристики й ключі (keys) для зведення складності близько до O(n). Однак і тут існують обмеження: за умови значних оновлень великої кількості вузлів сам процес дифінгу може створити навантаження; у таких випадках варто застосовувати оптимізації, наприклад пагінацію, віртуалізацію списків (windowing) або делегування рендерингу на сервер.

Управління подіями й асинхронними операціями в інтерфейсі ґрунтується на структурах черг і стеків: Event Loop JavaScript обробляє callback-функції через чергу (task queue), а стек викликів (call stack) гарантує послідовність виконання синхронних операцій. Для зменшення навантаження на процесор у відповідь на часті події (введення, прокрутка, зміна розміру) застосовують техніки debounce і throttle. Debounce відкладає виконання функції доти, поки серія подій не припиниться (корисно для автозаповнення), а throttle обмежує частоту викликів до фіксованого інтервалу (корисно під час обробки scroll). Обидві техніки підвищують продуктивність інтерфейсу, але мають свої компроміси: неправильно налаштований debounce може зробити систему нечутливою, а занадто агресивний throttle – уповільнити реакцію.

Оптимізація передачі й збереження даних складається з декількох шарів. Lazy loading («ліниве завантаження») знижує початкове навантаження на мережу, відкладаючи завантаження ресурсів до моменту потреби; практика показує, що впровадження lazy loading для зображень і фреймів значуще зменшує TTFB (time to first byte) та Time to Interactive. Проте lazy loading може впливати на індексацію контенту пошуковими ботами або на доступність, якщо не реалізований з урахуванням progressive enhancement; IntersectionObserver API у браузері є рекомендованим інструментом для коректної реалізації lazy loading. Кешування – ще одна базова техніка: загальноприйняті алгоритми кеш-заміщення, такі як LRU (Least Recently Used), дозволяють зберігати релевантні дані в обмеженій пам’яті, забезпечуючи операції вставки і видалення за амортизований O(1) за допомогою поєднання хеш-структури й двозв’язного списку. Однак як і будь-який кеш, LRU має обмеження: він не підлаштовується під специфіку робочого навантаження (наприклад, циклічний доступ) і потребує тонкого налаштування розмірів кешу.

Розуміння структури сайту як графа відкриває можливості для використання алгоритмів обходу та маршрутизації. У невагомих графах (без ваг) BFS гарантує знаходження найкоротшого шляху в кількості ребер, тоді як у зважених графах потрібні алгоритми на кшталт Dijkstra (O(E + V log V) з використанням купи) або A* для конкретних евристик. У контексті SPA маршрутизація та оптимізація навігації покладаються на подібні концепції: графова модель дозволяє виявляти ізольовані сторінки, оптимізувати внутрішні посилання та будувати

sitemap для пошукових систем. Проте графові підходи вимагають актуалізації за умови динамічного генерування контенту й за великої кількості вузлів – у такому випадку аналітика й індексація потребують розподілених обчислень.

Важливим є також розуміння взаємозв'язку алгоритмічних рішень із UX та етичними аспектами. Персоналізація, що використовує алгоритми рекомендацій і класифікації, покращує релевантність контенту, але водночас може створювати «filter bubble» – замкнутий масив контенту, який обмежує експозицію користувача. Алгоритмічні модулі можуть упереджено ставитись до окремих категорій користувачів, якщо навчальні дані не репрезентативні; отже, під час впровадження персоналізації треба враховувати питання прозорості, контролю користувача й захисту приватних даних (GDPR-сумісність запитів та механізмів зберігання).

Із практичної точки зору, у процесі проєктування інтерфейсів важливо дотримуватися правил розподілу обчислень між клієнтом і сервером. Тяжкі обчислювальні задачі (великі сортування, масивні фільтри або квазірекурсивні алгоритми обходу) краще виконувати на сервері або у фонових потоках (WebWorkers), щоби не блокувати UI. Також застосовують індексацію й попередню агрегацію (precompute), щоб пришвидшити відповіді у разі повторних запитів.

Зрештою, сучасні інструменти й фреймворки надають засоби, що спрощують впровадження алгоритмічних патернів: від віртуальних DOM і систем state-management до вбудованих механізмів кешування та SSR / SSG (server-side rendering / static site generation) у системах типу Next.js. Проте кожен інструмент має свої обмеження: SSR підвищує початкову вартість генерації сторінок, але покращує SEO і TTFB; SSG підходить для статичного контенту, але втрачає гнучкість для динамічних персоналізованих сторінок.

Висновки. Алгоритмічні структури є основою ефективного вебдизайну, оскільки вони забезпечують не лише функціональність, а й оптимізацію даних і швидкодію інтерфейсів. Інтеграція алгоритмів у дизайн дозволяє створювати динамічні, інтелектуальні системи, що адаптуються під потреби користувача. Подальші дослідження у цій сфері сприятимуть розвитку алгоритмічно-орієнтованих підходів до побудови вебсайтів і цифрових продуктів.

Abstract. This article examines the role of algorithmic structures and data organization in modern web design and development. It discusses how algorithms for sorting, searching, and optimization are applied in building user interfaces and improving performance. The paper also explores the use of tree and graph data structures in DOM representation and site architecture. Algorithmic thinking is shown to be key to creating efficient, adaptive, and user-centered web interfaces.

Keywords: algorithms, data structures, web design, optimization, interface, DOM, graphs.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Document Object Model (DOM) Level 3 Core Specification. *W3C.* URL: https://www.w3.org/TR/DOM-Level-3-Core/

2. Preobrazhenskaya T, Bakaev M., Avdeenko T. The Role of Data Structures Design in Modern Web Applications Development. Conference: Strategic Technology (IFOST), 8th International Forum on. 2013. Vol. 2. URL: https://www.researchgate.net/publication/261380698

3. Jasper van Riet, Malavolta I., Taher A. Chaleb. Optimize along the way: An industrial case study on web performance. *Journal of Systems and Software.* 2023. Vol. 190. URL: https://www.sciencedirect.com/science/article/pii/S0164121222002692

4. Freeman E., Robson E. Head First HTML and CSS. Canada: O'Reilly Media, 2021. URL: https://books.google.com.ua/books?id=BZIYQtV6yKsC&printsec=copyright&hl=uk#v=onepage&q&f=false

5. Introduction to Algorithms / T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. MIT Press, 2022. URL: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/

6. Nielsen J. Usability Engineering. Academic Press, 2020. URL: https://surl.li/czsfgu

7. Duckett J. JavaScript and JQuery: Interactive Front-End Web Development. Wiley, 2019. URL: https://surl.li/fsnpdq

8. Welling L., Thomson L. PHP and MySQL Web Development. Addison-Wesley, 2020. URL: https://repository.unikom.ac.id/32751/1/php%20and%20mysql%20web%20dev.pdf

9. Knuth D. E. The Art of Computer Programming. Addison-Wesley. URL: https://surl.li/yfmoxq

10. Двірничук К. В., Вацек Д. О. Веб-програмування та веб-дизайн. Київ: КНТ, 2020.

11. Коваленко О. О. Алгоритми та структури даних. Вінниця: ВНТУ, 2025. 114 с.

12. Пасічник В. В., Пасічник О. В. Веб-дизайн. Львів: «Магнолія-2006», 2018. 518 с.