МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Комп'ютерних наук і технологій
(повне найменування факультету)
Системний аналіз та обчислювальна математика
(повне найменування кафедри)

Пояснювальна записка

до дипломної роботи
Бакалавр
(ступінь вищої освіти)

на тему Розробка та дослідження інтелектуальної
(назва теми)
багатокористувацької системи для гри в шахи на основі веб-технологій

Виконав(ла): студент(ка) 4 курсу, групи КНТ 813-сп

Спеціальності 124 Системний аналіз
(код і найменування спеціальності)

Інтелектуальні технології та прийняття рішень в складних системах
Освітня програма (спеціалізація)

ШУТЬ К.В.
(ПРІЗВИЩЕ та ініціали)

Керівник ШИРОКОРАД Д.В.
(ПРІЗВИЩЕ та ініціали)

Рецензент ДУМІН О.М.
(ПРІЗВИЩЕ та ініціали)

2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
Кафедра системного аналізу та обчислювальної математики
Ступінь вищої освіти бакалавр
Спеціальність 124 Системний аналіз
(код і найменування)
Освітня програма (спеціалізація) Інтелектуальні технології та прийняття рішень в складних системах
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри САОМ
Еліна ТЕРЕЩЕНКО
« 8 » червня 2026 року

З А В Д А Н Н Я
Розділ 1 НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

Шуть Кирило Вячеславович
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка та дослідження інтелектуальної багатокористувацької системи для гри в шахи на основі веб-технологій

керівник проєкту (роботи) к.ф.-м.н., доцент ШИРОКОРАД Д.В.
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від « 23 » квітня 2026 № 196
2. Строк подання студентом проєкту (роботи) « 8 » червня 2026 року
3. Вихідні дані до проєкту (роботи) Тема дипломної роботи. Наукові та навчально-методичні джерела з розробки інтелектуальних багатокористувацьких систем. Методи та алгоритми реалізації шахової логіки.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Аналіз процесів прийняття рішень та взаємодії користувачів у багатокористувацьких інтелектуальних системах. Розробка шахового рушія. Реалізація AI-суперника на основі алгоритмів Minimax та Alpha-Beta Pruning. Розробка багатокористувацького режиму гри. Забезпечення серверної валідації синхронізації станів гри. Розробка адаптивного користувацького інтерфейсу. Дослідження ефективності алгоритмів Minimax та Alpha-Beta Pruning. Аналіз результатів роботи системи.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів) Презентація 15 слайдів
6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1	Широкорад Д.В.	01.04.2026	20.04.2026
2	Широкорад Д.В.	01.04.2026	20.04.2026
3	Широкорад Д.В.	01.04.2026	06.05.2026
4	Широкорад Д.В.	01.04.2026	20.05.2026
5	Широкорад Д.В.	01.04.2026	01.06.2026
Нормоконтроль	Широкорад Д.В.	01.04.2026	01.06.2026

7. Дата видачі завдання « 1 » листопад 2025 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області та існуючих багатокористувацьких шахових систем	листопад 2025	Виконано
2	Дослідження методів реалізації шахової логіки та алгоритмів прийняття рішень	листопад 2025	Виконано
3	Проєктування архітектури інтелектуальної багатокористувацької системи	листопад 2025	Виконано
4	Розробка шахового рушія з підтримкою правил гри	грудень 2025	Виконано
5	Реалізація AI-суперника на основі алгоритму Minimax	січень 2026	Виконано
6	Дослідження ефективності алгоритмів Minimax та Alpha-Beta Pruning	лютий 20256	Виконано
7	Розробка багатокористувацького режиму гри	березень 2026	Виконано
8	Реалізація серверної валідації та синхронізації станів гри	квітень 2026	Виконано
9	Розробка адаптивного користувацького інтерфейсу	травень 2026	Виконано
10	Тестування системи, аналіз результатів та оформлення пояснювальної записки	травень 2026	Виконано

Студент(ка) _____ Кирило ШУТЬ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Дмитро ШИРОКОРАД
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Дипломна робота: 76 стор., 11 табл., 8 джерел.

Пояснювальна записка присвячена розробленню та дослідженню інтелектуальної багатокористувацької вебсистеми для гри в шахи. У роботі розглянуто повний цикл створення програмного рішення: від аналізу предметної області й формалізації шахових правил до реалізації клієнт-серверної взаємодії, AI-суперника та оцінювання ефективності пошукових алгоритмів.

Об’єктом дослідження є процеси прийняття рішень і взаємодії користувачів у багатокористувацьких інтелектуальних системах. Предметом дослідження визначено методи реалізації шахової логіки, алгоритмів штучного інтелекту та синхронізації стану у вебзастосунках реального часу.

У межах дипломного проєкту створено клієнтську частину на основі React[1] і PixiJS[2], серверну частину на Node.js[3] із використанням Socket.IO[4], а також спільний модуль ChessEngine, який застосовується як на клієнті, так і на сервері. Реалізовано генерацію легальних ходів, перевірку шаху, мату й пату, рокірування, взяття на проході, перетворення пішака, режими гри з ботом і через мережу.

Дослідницька частина роботи зосереджена на порівнянні стратегій RandomBot, GreedyBot, MinimaxBot і MinimaxBot з Alpha-Beta Pruning[5]. Проаналізовано залежність кількості переглянутих позицій від глибини пошуку, вплив упорядкування ходів на відсікання та практичну придатність алгоритмів для браузерного шахового застосунку.

ЗМІСТ

ВСТУП

Шахи є зручною предметною областю для дослідження інтелектуальних програмних систем, оскільки поєднують формально визначені правила, велику кількість можливих позицій і потребу у стратегічному прогнозуванні. На відміну від випадкових ігор, результат шахової партії значною мірою залежить від якості аналізу стану, вибору ходу та здатності передбачати відповідь суперника.

Актуальність роботи зумовлена поширенням вебтехнологій, які дають змогу створювати інтерактивні багатокористувацькі застосунки без встановлення окремого програмного забезпечення. Для шахової системи це означає необхідність об’єднати графічний інтерфейс, коректну ігрову логіку, мережеву синхронізацію та алгоритми штучного інтелекту в одному узгодженому рішенні.

Метою дипломного проєкту є розробка та дослідження інтелектуальної багатокористувацької системи для гри в шахи на основі вебтехнологій із застосуванням алгоритмів штучного інтелекту. Досягнення цієї мети передбачає створення шахового рушія, реалізацію AI-суперника, побудову мережевого режиму гри, забезпечення серверної перевірки ходів і розроблення адаптивного інтерфейсу.

Об’єктом дослідження виступають процеси прийняття рішень та взаємодії користувачів у багатокористувацьких інтелектуальних системах. Предмет дослідження становлять методи й алгоритми реалізації шахової логіки, штучного інтелекту та клієнт-серверної взаємодії у вебзастосунках.

У роботі використано практичний підхід: теоретичні положення розглядаються через їх застосування у створеній системі. Особливу увагу приділено тому, як одна й та сама модель стану шахової партії використовується для відображення дошки, перевірки ходів, роботи ботів і серверної валідації.

Практичне значення результатів полягає у створенні працездатного вебзастосунку, що підтримує гру проти комп’ютерного суперника та гру між

двома користувачами в режимі реального часу. Результати дослідження можуть бути використані як основа для подальшого розвитку шахових навчальних систем, онлайн-платформ і демонстраційних проєктів із алгоритмів штучного інтелекту.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Особливості шахових вебсистем

Шахова веб-система — це не просто гра з переміщення фігур на дошці. Загалом, кожна дія користувача повинна інтерпретуватися за правилами гри, станом матчу та можливістю переходу до нового стану. Отже, гра є системою введення, і остаточне рішення про те, чи є хід правильним, приймається шаховою логікою.

Для багатокористувацького режиму ключовою вимогою є наявність одного авторитетного стану матчу. Якщо клієнти рухаються незалежно, а сервер не встигає координувати, навіть короткочасна втрата мережевого повідомлення може потенційно призвести до різних позицій серед гравців. У нашій системі це перевіряється на сервері, і підтверджений стан розповсюджується по системі.

Більшість сучасних шахових платформ складаються з багатьох допоміжних сервісів: рейтинги, турніри, аналіз гри, тренувальні вправи та соціальні функції.

У дипломному проекті ми зосередимося на основних інженерних компонентах, які потрібно виконати, щоб він працював належним чином: рушії правил, AI-суперники, мережеві кімнати та синхронізація.

З аналізу предметної області чітко видно, що краще відокремити логіку шахової гри від її графічного представлення. Клієнти React[1] та PixiJS[2] відповідають за взаємодію з користувачем, а ChessEngine описує зміни позицій гри незалежно від того, як вона відображається. Шахова гра є дискретною. Стан дошки змінюється не безперервно, а після окремих ходів, і кожен хід можна перевірити, застосувати та записати. Ця властивість дозволила організувати мережевий режим через події Socket.IO[4], у яких передаються ходи та знімки поточного стану.

Унікальною особливістю веб-системи шахів є стійкість до переривань з'єднання. У розробленому додатку використовується механізм повторного

підключення до кімнати, щоб гравець міг повернутися до поточної гри і зберегти свою сторону та стан гри.

Важливо зазначити, що шаховий веб-додаток загалом має ігровий, освітній та алгоритмічний характер. Для користувача він має бути інтуїтивно зрозумілим, але насправді він повинен дотримуватися правил, і якщо це не так, то навіть гарний інтерфейс не гарантує правильність гри.

На відміну від багатьох ігор у реальному часі, шахи не потребують постійної передачі координат або фізичних параметрів. Основною одиницею мережевого обміну є хід. Це також початкова клітинка, кінцева клітинка та додаткова інформація для особливих випадків. Це спрощує протокол, але також ускладнює перевірку кожної події. Важливою властивістю шахової системи є здатність відтворювати гру. Якщо відома послідовність ходів і початкова позиція, то стан гри можна відновити. У проекті також робится знімок стану, що є швидшим і зручнішим для синхронізації мережі. Також важливо щоб система не залежала від зовнішнього шахового API. Вся логіка правил реалізована у власному ChessEngine, тому результати досліджень алгоритмів штучного інтелекту базуються на роботі програмного рішення.

Шахова вебсистема також потребує чіткого розділення між дозволеним ходом і наміром користувача. Натискання на клітину може означати вибір фігури, скасування вибору або спробу ходу. Тому інтерфейс повинен передавати в рушій тільки сформовані дії, а не самостійно змінювати правила партії.

Під час проєктування шахової вебсистеми необхідно враховувати різницю між ігровим інтерфейсом і формальною моделлю партії. Інтерфейс може бути побудований різними способами, проте правила гри залишаються незмінними. Тому архітектура повинна передбачати, що будь-який візуальний вибір користувача перетворюється на строго описаний хід лише після перевірки рушієм.

Ще однією особливістю є дуже обмежений, але дуже багатий набір станів. У шахах немає випадкових подій, але є багато комбінацій фігур, атак і спеціальних правил. Тому шахи є хорошою моделлю для алгоритмів прийняття

рішень, і всі рішення можуть бути прийняті та перевірені за допомогою системи. Для користувача важливо, щоб система швидко адаптувалася, але не менш важливо, щоб відповідь була правильною. Якщо бот здатний відповісти за дуже короткий час, але порушує правила, то така реалізація не має наукової та практичної цінності. Таким чином, ми розглядаємо механізм правил як основу всієї системи.

Онлайн-режим додає до шахової задачі ще один вимір — узгодження стану між віддаленими клієнтами. У класичному локальному застосунку достатньо оновити один екземпляр дошки, тоді як у мережевому режимі потрібно гарантувати, що обидві сторони отримали однаковий результат одного й того самого ходу.

1.2 Вимоги до шахового рушія

Шаховий рушій є центральним модулем системи, оскільки саме він визначає, які переходи між позиціями є допустимими. У межах проєкту рушій повинен працювати однаково для інтерфейсу, ботів і серверної частини, тому його реалізовано як спільний модуль, доступний у каталогах клієнта й сервера.

Основна вимога до рушія полягає у генерації лише легальних ходів. Недостатньо врахувати геометрію руху фігури, оскільки хід може відкривати власного короля під шах. Тому після побудови кандидатів рушій виконує симуляцію позиції та вилучає варіанти, які порушують базову умову безпеки короля.

Для коректної реалізації правил необхідно зберігати не лише розташування фігур, а й допоміжний контекст партії. До такого контексту належать сторона, що має право ходу, ознака попереднього переміщення короля або тури, а також інформація про останній хід, потрібна для взяття на проході.

Рушій має підтримувати спеціальні правила шахів: рокірування, en passant і перетворення пішака. Ці правила відрізняються від звичайних переміщень тим, що залежать від історії гри або змінюють більше ніж одну фігуру. Тому вони обробляються в централізованій процедурі застосування ходу.

Двигун повинен мати певні шахові правила: рокіровка, взяття на проході та перетворення пішака. Вони відрізняються від звичайних ходів тим, що залежать від історії гри і можуть змінювати більше ніж одну фігуру. Тому вони обробляються в централізованій системі застосування ходів. Однією з найважливіших вимог є швидкий знімок стану гри та відновлення гри. У розробленому ChessEngine це використовується для синхронізації клієнтів у багатокористувацькому режимі, повторного підключення та узгодження стану після того, як сервер підтверджує хід.

Рушій повинен бути придатним до багаторазового клонування. Алгоритми Minimax[6] і Alpha-Beta Pruning[5] перевіряють велику кількість майбутніх позицій, тому під час пошуку вони не можуть змінювати реальну партію. Клонування дає змогу виконувати симуляції незалежно від поточного стану інтерфейсу.

З погляду програмної архітектури шаховий рушій має бути ізольованим від бібліотек відображення й мережевих інструментів. Це спрощує тестування, зменшує кількість залежностей і дає змогу використовувати одну логіку в різних середовищах виконання JavaScript.

Таблиця 1.1 — Варіанти представлення шахової позиції

Підхід	**Переваги**	**Недоліки**	**Доцільність для проєкту**
Масив 8x8	Проста відповідність шаховій дошці, легке звернення за координатами.	Менш зручна серіалізація та перебір усіх клітин.	Можливий, але не обраний як основний.
Плоский масив 64 клітин	Зручний перебір, проста серіалізація, сумісність із клієнтом і сервером.	Потребує методів пошуку клітини за координатами.	Обраний у ChessEngine.
Bitboard	Висока швидкість операцій, компактність, придатність для сильних рушіїв.	Складніша реалізація, нижча прозорість для навчального проєкту.	Надмірний для поточного рівня системи.
FEN-рядок	Компактний стандартний запис позиції.	Незручний для постійної роботи рушія без парсингу.	Може бути напрямом подальшого розвитку.

Окремою вимогою є серіалізація стану. Методи toSnapshot і loadSnapshot дозволяють передати позицію між сервером і клієнтом та відновити її після повторного підключення. Це робить рушій не лише локальним модулем правил, а й частиною мережевої архітектури.

Окремою вимогою для двигуна є передбачуваність зовнішнього інтерфейсу. Методи, які повертають ходи або застосовують їх, повинні мати чіткі

вхідні параметри і не залежати від графічного стану клієнта. Таким чином, ми можемо використовувати двигун однаково в тестах, ботах і на серверній стороні. Двигун повинен мати можливість працювати з інформацією, яку користувач не знає. Наприклад, при просуванні пішака клієнт може не мати можливості надати бажаний тип фігури. Тому надається варіант за замовчуванням для просування до ферзя, що не порушує правил і дозволяє завершити хід.

Для багатокористувацької гри важливо, щоб стан рушія міг бути серіалізований без втрати службових ознак. Якщо під час snapshot втратити інформацію про останній хід або про те, чи рухалася тура, спеціальні правила надалі працюватимуть неправильно.

Вимога до обробки шаху є однією з найскладніших, оскільки шах може виникати не лише внаслідок прямого взяття, а й через відкриття лінії після переміщення іншої фігури. Саме тому перевірка легальності ходу виконується через моделювання позиції після ходу.

Рушій повинен підтримувати як повну перевірку партії, так і локальні запити. Інтерфейсу потрібні доступні ходи для конкретної фігури, серверу — перевірка конкретного ходу, а AI[7] — багаторазове застосування ходів у глибину дерева пошуку.

Для дипломного проєкту важливо, що рушій реалізовано без залежності від готової шахової бібліотеки. Це дозволяє пояснити всі алгоритмічні рішення у записці та пов’язати їх із дослідженням Minimax[6] і Alpha-Beta Pruning[5].

1.3 Вимоги до багатокористувацького режиму

Багатокористувацький режим повинен забезпечувати узгоджену партію для двох гравців, які працюють із різних клієнтів. Для цього сервер створює кімнату, призначає сторонам кольори, приймає ходи та повертає обом учасникам підтверджений стан гри.

Першою вимогою є контроль черги ходів. Клієнт не повинен мати змоги виконати хід за суперника або зробити два ходи поспіль. У серверній частині це перевіряється за стороною гравця, статусом кімнати та активною стороною у ChessEngine.

Другою вимогою є валідація ходу на сервері. Навіть якщо клієнтський інтерфейс приховує недопустимі ходи, сервер не може покладатися на довіру до клієнта. Усі отримані ходи повторно перевіряються спільним рушієм перед тим, як партія переходить у новий стан.

Третьою вимогою є синхронізація після кожної успішної дії. У проєкті сервер надсилає подію з актуальним gameState, завдяки чому клієнти не будують власні незалежні версії дошки, а отримують єдиний підтверджений результат.

Оскільки мережеве середовище може бути нестабільним, система повинна обробляти відключення без негайного руйнування партії. Реалізований механізм reconnect дає змогу користувачеві повернутися до кімнати, якщо сервер ще зберігає її стан.

Багатокористувацький режим повинен бути зрозумілим не лише з технічного боку, а й з боку користувацького сценарію. Користувач має швидко зрозуміти, чи він створює кімнату, чи приєднується до вже створеної, яку сторону отримав і чому партія ще не почалася.

У системі потрібно передбачити повідомлення про помилкові ситуації: неіснуючу кімнату, заповнену кімнату, хід не тієї сторони або втрату з’єднання. Такі повідомлення не повинні розкривати зайвої технічної інформації, але мають допомагати користувачу продовжити роботу.

Узгодження сторін є важливою частиною логіки. Якщо два клієнти одночасно вважають себе білими або чорними, партія стане некоректною ще до першого ходу. Тому призначення сторін виконується сервером, а не визначається клієнтом самостійно.

Для реального часу достатньо передавати події ходів, однак сервер повинен контролювати життєвий цикл кімнати. Партія має різні стани:

очікування, активна гра, тимчасове відключення, завершення. Кожен стан задає різні допустимі дії.

Підтримка повторного підключення є практично важливою навіть для невеликого проєкту. Короткий розрив Wi-Fi або оновлення сторінки не повинні автоматично знищувати партію, якщо сервер ще має її стан і може повернути користувача до гри.

Таблиця 1.2 — Основні ризики багатокористувацької шахової партії

Ризик	**Наслідок**	**Механізм зменшення ризику у системі**
Некоректний хід клієнта	Порушення правил або нечесна перевага.	Серверна перевірка make_move через ChessEngine.
Розходження станів	Гравці бачать різні позиції.	Розсилка підтвердженого gameState після успішного ходу.
Втрата з’єднання	Переривання партії.	Позначення disconnected і підтримка rejoin_room.
Хід не тієї сторони	Порушення черговості гри.	Перевірка activeSide на сервері.
Приєднання зайвого гравця	Невизначені ролі у кімнаті.	RoomManager обмежує кімнату двома сторонами.

1.4 Вибір технологій

Технологічний стек проєкту сформовано з урахуванням потреб інтерактивного вебзастосунку. JavaScript використано як спільну мову для клієнта, сервера та шахового рушія, що зменшує ризик розходження правил між різними частинами системи.

React[1] обрано для побудови станового інтерфейсу, оскільки шахова гра має кілька режимів, екранів і залежних станів: головне меню, гру проти бота, мережеву партію, очікування суперника та повідомлення про результат. Компонентна модель React[1] дає змогу структурувати ці частини без змішування їхньої відповідальності.

PixiJS[2] застосовано для відображення шахової дошки й фігур, оскільки бібліотека забезпечує продуктивне рендерення 2D-графіки через canvas/WebGL. Для шахової дошки це важливо через часті оновлення виділення клітин, можливих ходів, анімаційних станів і масштабування під різні екрани.

Node.js[3] використано для серверної частини, оскільки він добре узгоджується з подієвою природою багатокористувацької гри. Сервер обробляє створення кімнат, приєднання гравців, ходи, відключення та повторне підключення без потреби у складній багатопотоковій моделі.

Socket.IO[4] обрано як практичну надбудову над WebSocket-комунікацією. Для дипломного проєкту важливими є не лише двосторонні повідомлення, а й кімнати, події підключення, обробка розривів і повторне з’єднання. Ці можливості зменшують обсяг допоміжного мережевого коду.

Розгортання клієнта на GitHub Pages[8] і сервера на Railway[9] відповідає розділеній архітектурі застосунку. Клієнтська частина може обслуговуватися як статичний ресурс, тоді як сервер зберігає активні кімнати та виконує авторитетну перевірку ходів.

Вибір єдиної мови програмування для основних частин системи також спрощує підтримку типів даних. Об’єкти ходів, snapshot стану та опис фігур

можуть передаватися між клієнтом і сервером без складного перетворення між мовами.

React[1] зручний для керування станами інтерфейсу, однак сам по собі не є оптимальним інструментом для малювання шахової дошки. Тому його поєднано з PixiJS[2]: React[1] відповідає за структуру екранів і логіку станів, а PixiJS[2] — за графічний шар дошки.

Socket.IO[4] було обрано з урахуванням того, що мережевий режим потребує не тільки надсилання повідомлень, а й підтримки кімнат. Для шахової партії кімната є природною одиницею ізоляції, оскільки кожна гра має власних учасників і власний стан.

Node.js[3] добре підходить для такої серверної логіки, оскільки події ходів не потребують важких обчислень на сервері. Основне навантаження полягає у перевірці ходу, оновленні стану кімнати й розсиланні результату.

Обраний стек також відповідає вимогам розгортання. Статичний клієнт може бути опублікований на GitHub Pages[8], тоді як серверна частина, що потребує постійного процесу, розгортається окремо на Railway[9].

У сукупності ці технології дозволили створити систему, де клієнтська, серверна й алгоритмічна частини взаємодіють без зайвих проміжних шарів. Це важливо для бакалаврського проєкту, у якому потрібно не лише отримати працюючий застосунок, а й пояснити його архітектуру.

1.5 Теоретичні основи шахових рушіїв

Шаховий двигун розглядає правила гри як операції над станом дошки. Для програмного забезпечення позиція не є зображенням, а структурою даних, що містить фігури, їх координати, сторону, яка ходить, та інші особливості, необхідні для деяких спеціальних правил.

Типовий процес шахового двигуна складається з трьох кроків: побудова псевдолегальних ходів, перевірка безпеки короля та застосування обраного ходу. Якщо спочатку використовувати правила локального руху фігур, то ви оціните позицію ходу на дошці в цілому.

У розробленій системі позиція представлена у вигляді набору клітинок, кожна з яких має координати і може містити фігуру. Ця модель досить проста для серіалізації у знімок і може бути використана для пошуку фігур, перевірки атакованих клітинок та для моделювання позиції майбутніх ходів. Важливо розрізняти легальний хід та атаковану клітинку.

Наприклад, пішак атакує по діагоналі, хоча рухається вперед, і король не може бути використаний у рекурсивній перевірці, оскільки він завжди виробляється. Таким чином, двигун має власний механізм для пошуку цілей атаки. Результат гри полягає не лише в шаху, але й у кількості прийнятних відповідей. Мат відбувається, якщо сторона знаходиться під шахом і не має легального ходу; пат відбувається, коли немає ходів, але король не атакований.

Отже, двигун повинен мати можливість перебирати всі фігури сторони і перевіряти їх можливості. Теоретична модель двигуна безпосередньо впливає на штучний інтелект. Алгоритми пошуку не генерують ходи самостійно, а покладаються на ChessEngine.

Таким чином, бот аналізує лише варіанти, надані правилами, і не дублює складну шахову логіку у власному коді. Таким чином, теоретична модель двигуна в роботі безпосередньо пов'язана з його реалізацією. Ось чому ChessEngine не є частиною інтерфейсу користувача та мережі, але використовується в обох частинах системи.

У теорії шахових двигунів важлива повнота генерації. Двигун повинен не лише забороняти заборонені ходи, але й не пропускати жодної можливої опції. Втрата одного легального ходу може змінити результат пошуку штучного інтелекту або неправильне рішення про мат чи пат.

Інша важлива концепція - це інваріант стану. Після кожного ходу дошка повинна залишатися в узгодженому стані: фігура не може бути на двох клітинках

одночасно; активна сторона повинна змінюватися рівно один раз; спеціальні службові особливості гравця повинні бути правильного характеру для гри в цілому, а спеціальні службові особливості повинні бути актуальними для історії гри.

Для перевірки шаху двигун фактично вирішує проблему контролю клітинки короля. Це відрізняється від типового питання, які ходи доступні, оскільки деякі з них атакують клітинки інакше, ніж рухаються, коли немає цілі для атаки. Найочевидніший приклад - пішак. Обчислення мату та пату вимагає не лише локальної, але й глобальної перевірки позиції. Недостатньо знати, що король атакований чи не атакований; важливо знати, чи має вся сторона хоча б один спосіб змінити позицію, не порушуючи правил. З точки зору реалізації двигун є детермінованим. Той самий стан і той самий хід завжди повинні генерувати той самий результат. Це необхідно для валідації сервера, відновлення стану та порівняння алгоритмів штучного інтелекту.

Прозора модель двигуна була обрана в дипломному проекті для аналізу та тестування. Вона не оптимізована на рівні професійних шахових програм, але чітко показує зв'язки правил гри, генерації ходів та алгоритмів пошуку.

1.6 Висновки до розділу 1

У першому розділі визначено предметну специфіку шахових вебсистем, сформульовано вимоги до рушія, мережевого режиму та технологічного стеку. Обґрунтовано доцільність використання спільного ChessEngine, клієнтської частини на React[1] і PixiJS[2], серверної частини на Node.js[3] та подієвої взаємодії через Socket.IO[4].

РОЗДІЛ 2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ

2.1 Загальна структура проєкту

Архітектуру проєкту побудовано за принципом розділення відповідальності між клієнтом, сервером і спільною ігровою логікою. Такий поділ дозволяє уникнути дублювання правил шахів і водночас підтримувати різні режими гри: локальну партію проти бота та мережеву партію між користувачами.

Каталог client містить інтерфейс користувача, екрани гри, сервіс мережевої взаємодії та модулі AI[7], які виконуються у браузері. Саме ця частина відповідає за відображення дошки, вибір фігур, підсвічування ходів, запуск бота та показ поточного стану партії.

Каталог server реалізує авторитетну логіку багатокористувацького режиму. Сервер створює кімнати, зберігає прив’язку гравців до сторін, обробляє події Socket.IO[4], перевіряє ходи через ChessEngine і надсилає клієнтам узгоджений стан.

Каталог shared містить ChessEngine і спільні структури, що використовуються обома частинами системи. Це рішення є важливим для дипломного проєкту, оскільки одна модель правил застосовується для візуального режиму, AI-пошуку та серверної валідації.

Загальна структура системи забезпечує достатню модульність для розвитку. Новий бот, додатковий екран або інший спосіб розгортання можуть бути додані без зміни базових правил гри, якщо вони взаємодіють із ChessEngine через наявний інтерфейс.

Структура client/server/shared також полегшує аналіз проєкту в пояснювальній записці. Кожен каталог відповідає окремому аспекту системи: користувацькій взаємодії, мережевій координації або шаховим правилам.

Спільний модуль не містить знань про те, чи гра відбувається локально, проти бота або через мережу. Це важливий показник правильної декомпозиції, оскільки правила шахів залишаються незмінними для всіх режимів.

Клієнтська частина має більше станів користувацького досвіду, ніж власне шахових станів. Наприклад, очікування суперника, вибір бота або повідомлення про reconnect не є частиною правил шахів, але необхідні для зручної роботи застосунку.

Серверна частина навпаки мінімізує інтерфейсні рішення й концентрується на достовірності дій. Вона не повинна знати, як саме клієнт відобразить хід, але повинна гарантувати, що цей хід дозволений і належить правильному гравцю.

Архітектурна схема також спрощує подальше тестування. ChessEngine можна перевіряти ізольовано, Socket.IO-події — через мережеві сценарії, а інтерфейс — через взаємодію користувача з дошкою.

2.2 Клієнтська архітектура

Клієнтська частина поєднує React-компоненти, PixiJS-відображення та модулі штучного інтелекту. React[1] організовує переходи між екранами й керує станами, які мають логічне значення для користувача: вибір режиму, очікування суперника, хід бота, завершення партії.

Відображення дошки винесено в графічний шар на основі PixiJS[2]. Це дає змогу масштабувати дошку відповідно до розміру вікна, швидко оновлювати клітини та фігури, а також відокремити графічну побудову від логіки перевірки ходів.

Під час вибору фігури клієнт звертається до ChessEngine і отримує список легальних ходів. Отже, інтерфейс не обчислює правила самостійно, а лише

показує користувачеві дозволені варіанти, що зменшує ймовірність помилок у взаємодії.

У режимі гри проти бота клієнт використовує локальний екземпляр рушія й передає його AI-модулю для пошуку відповіді. Після вибору ходу бот застосовує його через той самий механізм, що й користувач, тому обидва джерела ходів працюють у спільній моделі.

У мережевому режимі клієнт не вважає хід завершеним до отримання підтвердження сервера. Це відповідає авторитетній архітектурі: локальна дія користувача перетворюється на запит, а реальна зміна дошки відбувається після події move_applied.

Адаптивність інтерфейсу досягається через перерахунок розмірів ігрової області та компактне розміщення елементів керування. Завдяки цьому система залишається придатною для гри як на широкому екрані, так і на мобільному пристрої.

Компонент GameScreen виконує роль координатора ігрового процесу на клієнті. Він отримує стан рушія, передає його графічній дошці, реагує на вибір клітин і запускає відповідну дію залежно від режиму гри.

У режимі гри проти бота клієнт має контролювати момент, коли користувач уже зробив хід, але AI[7] ще обчислює відповідь. Такий проміжний стан потрібний для запобігання подвійним діям і для коректного відображення очікування.

Мережевий режим додає ще один рівень асинхронності. Користувач може ініціювати хід, але клієнт повинен дочекатися серверного підтвердження. Це впливає на інтерфейс: він має обробляти як успішний хід, так і повідомлення про помилку.

Графічний шар PixiJS[2] працює з координатами дошки, виділеними клітинами й фігурами. При цьому логічні координати залишаються пов’язаними з ChessEngine, що дозволяє уникнути ситуації, коли картинка й стан партії існують окремо.

Адаптивне масштабування не повинно змінювати правила вибору клітин. Незалежно від розміру екрана натискання має однозначно відповідати конкретній клітині, а тому перерахунок координат є важливою частиною клієнтської реалізації.

2.3 Серверна архітектура

Серверна частина виконує роль координатора мережевої партії. Вона не займається відображенням дошки, але відповідає за створення кімнат, збереження стану, перевірку прав гравців і розсилання результатів обом клієнтам.

Для кожної кімнати сервер зберігає екземпляр ChessEngine, статус партії та дані про гравців білої й чорної сторони. Така структура дозволяє ізолювати партії одна від одної та обробляти кілька незалежних ігор у межах одного серверного процесу.

Подія create_room створює нову кімнату й призначає першого користувача одній зі сторін. Подія join_room приєднує другого гравця та переводить кімнату в ігровий стан, якщо обидві сторони зайняті.

Подія make_move є найбільш відповідальною операцією серверної частини. Перед застосуванням ходу сервер перевіряє існування кімнати, її статус, належність користувача до сторони, чергу ходу та коректність переданих координат.

Після проходження попередніх перевірок сервер викликає ChessEngine.makeMove. Якщо рушій відхиляє хід, клієнт отримує повідомлення invalid_move; якщо хід успішний, усім учасникам кімнати надсилається move_applied з актуальним станом гри.

Серверна авторитетність важлива не лише для захисту від некоректних дій, а й для синхронізації. Навіть якщо клієнт має власний ChessEngine, фінальним джерелом істини в мережевій партії є стан, сформований сервером.

Обробка disconnect не завершує партію миттєво. Сервер позначає гравця як тимчасово відключеного і залишає можливість повернення протягом визначеного часу. Це підвищує практичну стабільність гри в реальних мережевих умовах.

Таблиця 2.1 — Події Socket.IO[4] у серверній архітектурі

Подія	**Напрям**	**ризначення**
create_room	клієнт -> сервер	Створення кімнати і призначення першому гравцю білої сторони.
join_room	клієнт -> сервер	Приєднання другого гравця до кімнати за ідентифікатором.
make_move	клієнт -> сервер	Передавання наміру виконати хід з параметрами fromId, toId, promotionTo.
move_applied	сервер -> клієнти	Розсилка підтвердженого ходу та нового стану партії.
invalid_move	сервер -> клієнт	Повідомлення про відхилення некоректного або несвоєчасного ходу.
rejoin_room	клієнт -> сервер	Відновлення участі у партії після втрати з’єднання.

Для повторного підключення використовується подія rejoin_room. Користувач передає дані попередньої сесії, після чого сервер перевіряє можливість відновлення участі та повертає клієнту поточний стан дошки.

Архітектура сервера не потребує бази даних для поточного варіанта системи, оскільки кімнати існують у пам’яті процесу. Це спрощує реалізацію дипломного прототипу, але водночас визначає обмеження щодо довготривалого зберігання партій.

Наявність HTTP-маршруту health використовується для перевірки доступності серверного застосунку після розгортання. Це допоміжний, але корисний елемент для середовища Railway[9], де важливо контролювати працездатність сервісу.

Клас RoomManager виконує роль окремого шару керування партіями. Він приховує деталі створення кімнати, пошуку гравця, позначення відключення та очищення завершених станів. Завдяки цьому основний серверний файл зосереджений на обробці подій Socket.IO[4].

Ідентифікатор кімнати формується як короткий унікальний код, який зручно передати іншому користувачу. Такий підхід достатній для дипломного застосунку, де кімната створюється на час конкретної партії, а не є постійним обліковим записом або турнірним ресурсом.

Серверна перевірка параметрів ходу має кілька рівнів. Спочатку перевіряється формат запиту, потім право гравця діяти у відповідній кімнаті, далі черга ходу, і лише після цього викликається шаховий рушій. Така послідовність зменшує кількість некоректних звернень до ChessEngine.

Подія invalid_move є не просто повідомленням про помилку, а механізмом збереження стабільності партії. Якщо клієнт надсилає недопустиму дію, сервер не змінює стан кімнати, а повертає відповідь, що дозволяє інтерфейсу залишитися синхронізованим із реальною позицією.

Сервер не реалізує шахові правила вручну. Це принципове архітектурне рішення: перевірка статусу кімнати належить серверу, а перевірка шахового

змісту ходу належить ChessEngine. Завдяки цьому відповідальність модулів не змішується.

Використання Railway[9] для розгортання сервера відповідає потребам прототипу, оскільки платформа дозволяє швидко запустити Node.js-сервіс із зовнішнім доступом. При цьому логіка серверної частини залишається незалежною від конкретного провайдера.

Авторитетна серверна модель має ще одну перевагу: вона дозволяє зберігати логіку довіри в одному місці. У багатокористувацькій грі клієнт завжди потенційно може бути модифікований, тому правильність партії не повинна залежати від того, чи чесно клієнт сформував запит. Сервер приймає запит лише як намір користувача, а не як готовий результат.

У контексті Socket.IO[4] сервер фактично працює як диспетчер подій. Проте простого пересилання повідомлень між клієнтами недостатньо, оскільки тоді кожен клієнт мав би самостійно вирішувати, чи є хід допустимим. У розробленій системі сервер не пересилає неперевірений хід, а спершу застосовує його до власного ChessEngine.

Життєвий цикл кімнати дозволяє розмежувати допустимі операції. Наприклад, у стані waiting можлива подія join_room, але не повноцінний обмін ходами; у стані playing допускається make_move; після finished нові ходи не повинні змінювати партію. Така модель зменшує кількість граничних помилок.

Серверна частина також виконує роль джерела синхронізації для reconnect. Якщо клієнт повертається після розриву, він не повинен самостійно реконструювати партію за локальними даними. Сервер повертає поточний snapshot, що усуває залежність від того, які події клієнт устиг отримати до відключення.

У дипломному проєкті не використано складну систему автентифікації, оскільки предметом роботи є шахова логіка, AI[7] та мережевий режим. Проте навіть без реєстрації сервер контролює сторону гравця в кімнаті, і цього достатньо для демонстрації принципу авторитетної валідації.

2.4 Спільний шаховий рушій

Спільний шаховий рушій є основою узгодженості системи. Його використання в client і server усуває ситуацію, коли клієнт і сервер по-різному трактують один і той самий хід або спеціальне правило.

Внутрішній стан ChessEngine містить масив клітин, активну сторону та інформацію про останній хід. Кожна фігура має тип, сторону та службові ознаки, зокрема hasMoved, що необхідна для перевірки рокірування й початкових ходів.

Метод getAvailableMoves формує доступні ходи для обраної фігури з урахуванням поточної позиції. Для користувача цей метод визначає підсвічування дошки, а для AI[7] — множину гілок дерева пошуку.

Метод isMoveLegal використовується як точкова перевірка кандидата. Він корисний у серверній частині, де потрібно швидко визначити, чи може отриманий від клієнта хід бути застосований до поточної позиції.

Методи toSnapshot і loadSnapshot забезпечують передачу стану між процесами та відновлення позиції на клієнті. У мережевому режимі саме snapshot є формою узгодженого стану після серверного підтвердження.

Спільний рушій також зменшує кількість потенційних дефектів під час розвитку системи. Якщо правило змінюється або уточнюється в одному місці, воно автоматично стає доступним для всіх режимів, що використовують цей модуль.

Для AI[7] особливо важливо, що ChessEngine повертає не просто геометричні ходи, а вже відфільтровані легальні варіанти. Інакше Minimax[6] витрачав би ресурси на позиції, які не можуть виникнути в реальній партії, а результати дослідження були б викривлені.

Для сервера спільний рушій виконує роль незалежного арбітра. Клієнт може надіслати будь-який об’єкт ходу, але сервер приймає його лише тоді, коли ChessEngine підтверджує відповідність правилам і поточному стану.

Для клієнта рушій є джерелом інтерактивних підказок. Підсвічування доступних ходів не вимагає окремої логіки інтерфейсу, оскільки список варіантів формується тим самим модулем, який надалі застосує хід.

Таке потрійне використання одного модуля є важливим результатом проєктування. Воно демонструє, що шахова логіка не прив’язана до конкретного режиму гри, а є самостійною програмною сутністю.

2.5 Синхронізація стану у реальному часі

Синхронізація стану в реальному часі організована як обмін подіями між клієнтами й сервером. Клієнт надсилає намір виконати хід, а сервер після перевірки надсилає результат усім учасникам відповідної кімнати.

Такий підхід відрізняється від повністю локальної гри тим, що клієнт не є остаточним власником стану. Він може показати користувачеві доступні дії, але позиція у мережевій партії оновлюється тільки після отримання серверного повідомлення.

Socket.IO[4] дає змогу логічно групувати користувачів у кімнати. Завдяки цьому події однієї партії не потрапляють до інших клієнтів, а сервер може адресно розсилати зміни лише тим користувачам, які беруть участь у конкретній грі.

Для уникнення розходжень після кожного ходу передається не тільки інформація про сам хід, а й актуальний gameState. Це важливо, оскільки спеціальні правила можуть змінювати кілька елементів позиції, наприклад рокірування переміщує короля і туру.

У випадку відновлення з’єднання клієнт отримує повний стан партії, а не намагається відтворити пропущені події. Такий механізм є простішим і надійнішим для дипломного прототипу, де головним є гарантувати однакову позицію після reconnect.

Синхронізація також пов’язана з відображенням статусів. Клієнт повинен знати, чи очікується суперник, чи триває партія, яка сторона має хід і чи завершилася гра матом, патом або іншим результатом. Ця інформація входить до стану, що надсилається сервером.

Синхронізація тому розглядається як баланс між простотою протоколу і достатньою надійністю для браузерної гри.

У системах реального часу важливо розрізняти оптимістичне та авторитетне оновлення. Оптимістичний підхід швидше показує дію на клієнті, але може створити відкат після відмови сервера. У проєкті обрано авторитетний підхід, що краще відповідає потребі коректності шахової партії.

Передача повного стану після ходу збільшує обсяг повідомлення порівняно з передачею лише координат, але робить систему стійкішою до пропущених подій. Для шахів такий обсяг є прийнятним, оскільки позиція містить обмежену кількість клітин і фігур.

Синхронізація також враховує результат партії. Якщо після ходу виникає мат або пат, сервер надсилає цей стан обом клієнтам, і вони однаково завершують гру. Це виключає ситуацію, коли один клієнт вважає партію завершеною, а інший продовжує приймати ходи.

При reconnect особливо важливо не покладатися на локальну історію клієнта. Користувач може повернутися після того, як суперник зробив хід. Тому відновлення виконується через актуальний snapshot, сформований сервером.

Така модель синхронізації є достатньою для двох гравців і добре відповідає поставленим задачам. Вона не потребує складних механізмів вирішення конфліктів, оскільки в шахах одночасні ходи за правилами неможливі.

Синхронізацію можна розглядати як задачу підтримання спільного стану між кількома копіями інтерфейсу. У шаховій грі ця задача спрощується тим, що одночасність ходів за правилами неможлива: у кожен момент часу активною є лише одна сторона. Сервер використовує цю властивість для перевірки черги.

Передача snapshot після кожного ходу також спрощує обробку спеціальних правил. Клієнту не потрібно окремо знати, яку фігуру було знято під час en passant або куди перемістилася тура під час рокірування. Він отримує готовий стан, сформований рушієм на сервері.

У разі мережевої затримки клієнт може певний час перебувати у стані очікування підтвердження. Це нормальна властивість авторитетної моделі. Вона робить інтерфейс трохи менш миттєвим, але забезпечує коректність, що для шахової системи важливіше за оптимістичне локальне оновлення.

Для двох гравців не потрібні складні алгоритми узгодження конфліктів, які застосовуються в редакторах спільного редагування. Шахова партія має сувору чергу ходів, тому конфліктні дії найчастіше зводяться до недопустимого ходу або ходу неактивної сторони.

Синхронізація стану є також основою довіри між гравцями. Кожен учасник отримує не приватну версію подій, а той самий підтверджений стан кімнати. Це робить гру передбачуваною і зменшує ризик суперечливих ситуацій.

2.6 Висновки до розділу 2

У другому розділі описано архітектуру розробленої системи. Показано, що розділення на client, server і shared дає змогу поєднати адаптивний інтерфейс, авторитетний мережевий режим і єдину шахову логіку, яка використовується для гри користувача, роботи ботів і серверної перевірки.

Таблиця 2.2 — Модулі та їх призначення в додатку

Модуль	**Призначення**	**Основні складові**
client	Побудова інтерфейсу, запуск локальної гри, рендеринг дошки та взаємодія з користувачем.	React, PixiJS, GameScreen, MainMenu, ControllerView, AI-модулі.
server	Керування кімнатами, перевірка мережевих ходів, синхронізація стану між гравцями.	Express, Socket.IO, RoomManager, обробники подій.
shared	Зберігання універсальної шахової логіки, незалежної від інтерфейсу та транспорту.	ChessEngine, конфігурації клітин і ходів, перелік типів фігур.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ФУНКЦІОНАЛЬНИХ МОДУЛІВ

3.1 Реалізація шахової логіки

Реалізація логіки шахів починається з налаштування початкової позиції. ChessEngine генерує 64 клітини, розміщує всі фігури на стандартних рядах і встановлює білий бік як активний відповідно до правил початку гри.

Для кожної фігури двигун створює окрему схему для створення псевдолегальних ходів. Лінійні фігури перевіряють напрямки до першої перешкоди, кінь рухається дискретно, а король намагається перевірити сусідні клітини, а також умови для рокіровки. Логіка пішака є найбільш залежною від контексту.

Двигун враховує напрямок руху для сторони, можливість першого двоклітинного ходу, діагональні захоплення, захоплення на проході та досягнення останнього ряду для просування. Згенеровані кандидати ще не вважаються легальними ходами. Кожен з них перевіряється шляхом симуляції на клонованому стані: якщо після такого ходу король гравця під атакою, варіант видаляється з результатів.

Виявлення шаху здійснюється шляхом знаходження короля відповідної сторони та аналізу атакованих клітин суперника. Для цього ми використовуємо окрему логіку цілей атаки, яка не змішується зі звичайними легальними ходами і не створює рекурсивних залежностей.

Рокіровка перевіряє кілька умов одночасно: король і відповідна тура не повинні були раніше рухатися, проміжні клітини повинні бути вільними, і король не може проходити через атаковану клітину. Після того, як двигун підтвердив хід, він переміщує обидві фігури.

Захоплення на проході базується на інформації про попередній хід. Якщо пішак суперника щойно перемістився на дві клітини і опинився поруч з іншим пішаком, то інший пішак може зробити спеціальне діагональне захоплення на порожній клітині і видалити фігуру позаду неї.

Таблиця 3.1 — Етапи генерації легального ходу

Етап	**Зміст операції**	**Результат**
Вибір фігури	Перевіряється наявність фігури на клітині та відповідність активній стороні.	Недопустимий вибір одразу повертає порожній список.
Псевдолегальні ходи	Формуються клітини, що відповідають геометрії руху фігури.	Отримується набір кандидатів без урахування шаху власному королю.
Симуляція	Кожен кандидат застосовується до клонованого рушія.	Оригінальна партія не змінюється.
Перевірка шаху	Визначається, чи атакований власний король після ходу.	Небезпечні ходи відкидаються.
Повернення результату	Залишаються лише ходи, допустимі за повними правилами.	Контролер або AI отримує легальні ходи.

Перетворення пішака обробляється під час застосування ходу. Якщо користувач або бот передає параметр promotionTo, рушій створює відповідну фігуру; якщо параметр відсутній, використовується ферзь як типовий і найпрактичніший варіант.

Після успішного ходу ChessEngine змінює activeSide, а потім перевіряє стан партії для наступної сторони. Це дає змогу відразу визначити, чи призвів хід до шаху, мату, пату або звичайного продовження гри.

Мат визначається як ситуація, у якій король сторони перебуває під шахом і сторона не має жодного легального ходу. Пат фіксується за тієї самої

відсутності ходів, але без шаху. Отже, обидва результати залежать від повного перебору можливостей сторони.

Метод _sideHasAnyLegalMove перевіряє всі фігури відповідного кольору й завершує роботу одразу після знаходження хоча б одного легального ходу. Це рішення є універсальним і не потребує окремого опису конкретних матових чи патових конфігурацій.

Централізована процедура застосування ходу зменшує ризик розходження між звичайними й спеціальними правилами. Саме в ній обробляються переміщення фігури, взяття, рокірування, en passant, promotion і службове оновлення стану.

Симуляційні ходи використовують той самий механізм, що й реальна партія, але виконуються на копії рушія. Завдяки цьому Minimax[6] може аналізувати майбутні варіанти без пошкодження поточного стану дошки.

Практична перевага такої реалізації полягає в тому, що користувач, сервер і бот працюють із однаковим набором правил. Якщо хід недопустимий для ChessEngine, він не буде прийнятий ні в локальній грі, ні в мережевій партії.

Основним компромісом є додаткове навантаження, пов’язане з клонуванням позицій. Однак для обраних глибин AI-пошуку та навчально-дослідницького масштабу проєкту цей підхід забезпечує достатній баланс між простотою реалізації й коректністю.

Для кожної клітини зберігаються координатні характеристики, які дозволяють звертатися до позиції як через індекс масиву, так і через шахову логіку рядків і колонок. Це спрощує обчислення зміщень і перевірку меж дошки.

Під час руху лінійних фігур рушій послідовно переходить у вибраному напрямі, доки не досягне краю дошки або зайнятої клітини. Якщо зайнята клітина містить фігуру суперника, вона включається як можливе взяття, після чого промінь зупиняється.

Для коня використовується інший підхід, оскільки його рух не залежить від проміжних клітин. Рушій перевіряє вісім потенційних зміщень і відкидає ті, що виходять за межі дошки або ведуть на клітину з власною фігурою.

Король має найкоротший звичайний рух, але найскладнішу перевірку безпеки. Навіть якщо сусідня клітина вільна або зайнята суперником, хід не допускається, якщо ця клітина перебуває під атакою.

Реалізація спеціальних правил у спільному методі застосування ходу спрощує подальшу підтримку. Наприклад, сервер і бот не повинні знати, що під час рокірування переміщується тура, оскільки після виклику ChessEngine вони отримують уже коректний стан.

Snapshot стану містить достатньо інформації для відновлення позиції на іншому клієнті. Це особливо важливо для мережевого режиму, де клієнт може пропустити окрему подію через тимчасову втрату з’єднання.

Під час перевірки ходу рушій працює з координатами клітин, а не з візуальними пікселями. Це дозволяє повністю відокремити правила від способу відображення дошки. Якщо інтерфейс буде змінено, шахова логіка залишиться незмінною.

Для запобігання помилкам у спеціальних правилах важливо правильно оновлювати службові ознаки після кожного ходу. Якщо hasMoved не змінити після переміщення короля або тури, система може помилково дозволити рокірування в майбутньому.

Останній хід зберігається не для історії партії в повному сенсі, а для конкретної шахової потреби — визначення можливості en passant. Це приклад того, як навіть одне спеціальне правило впливає на модель стану рушія.

Перевірка атакованих клітин повинна бути обережною для короля. Якщо визначати атаку через звичайні легальні ходи суперника, можна отримати хибну рекурсію, оскільки легальність ходів суперника також залежить від безпеки його короля.

Рушій не використовує окремі шаблони для конкретних матових позицій. Замість цього він застосовує загальне правило: якщо сторона під шахом і не має легального ходу, це мат. Такий підхід універсальний і працює для різних конфігурацій.

Пат визначається аналогічно через відсутність ходів, але без шаху. Це важливо, оскільки зовні пат може бути схожим на програшну ситуацію, проте за правилами шахів він завершує партію нічиєю.

Під час promotion рушій повинен замінити пішака на нову фігуру тієї самої сторони. Ця операція виконується після переміщення на останню горизонталь, тому вона є частиною застосування ходу, а не попередньої генерації.

Клонування позиції використовується не тільки для AI[7], а й для перевірки легальності ходів. Симуляція дозволяє поставити питання: що буде після цього ходу, і чи залишиться власний король у безпеці.

Такий підхід є обчислювально дорожчим за деякі оптимізовані методи, але він значно прозоріший. Для дипломної роботи прозорість важлива, оскільки дає змогу обґрунтувати кожен крок перевірки правил.

Під час реалізації рушія важливо уникати прихованих залежностей між методами. Наприклад, метод отримання доступних ходів не повинен випадково змінювати активну сторону або останній хід, оскільки він використовується не лише для реальної гри, а й для підсвічування та AI-пошуку.

Обробка взяття має бути узгоджена для звичайних і спеціальних ходів. У звичайному випадку фігура суперника міститься на кінцевій клітині, тоді як en passant вилучає пішака з іншої клітини. Тому спеціальні наслідки неможливо коректно реалізувати тільки простим переміщенням фігури.

Рушій також має враховувати, що фігура після ходу може змінити свій тип. Promotion є прикладом переходу, у якому ідентичність об’єкта на клітині змінюється: пішак завершує рух, але в новому стані дошки на цій клітині вже перебуває інша фігура.

Під час клонування потрібно передавати не лише розташування фігур, а й службові властивості. Якщо копія позиції не містить hasMoved або lastMove, симуляції AI[7] можуть дозволяти ходи, яких не було б у реальній партії.

Усі ці рішення показують, що шахова логіка не є набором незалежних правил для фігур. Це цілісна система стану, де історія ходів, безпека короля, активна сторона і спеціальні переходи взаємно впливають одне на одного.

3.2 Реалізація штучного інтелекту

Штучний інтелект у системі реалізовано не як окремий шаховий рушій, а як набір стратегій вибору ходу поверх ChessEngine. Це означає, що всі боти отримують тільки легальні ходи й не дублюють правила руху фігур.

RandomBot є базовим рівнем і вибирає випадковий хід із доступних. Його призначення полягає не в сильній грі, а в перевірці взаємодії інтерфейсу, рушія та механізму виконання ходу без складної логіки оцінювання.

GreedyBot використовує локальну оцінку без глибокого прогнозування. Він віддає перевагу ходам із безпосередньою матеріальною вигодою, зокрема взяттям цінних фігур. Така стратегія демонструє обмеження короткозорого вибору: вигідний на поточному ході варіант може виявитися позиційно слабким.

MinimaxBot аналізує дерево ходів на задану кількість півходів. На рівнях, де ходить бот, вибирається максимальна оцінка; на рівнях суперника передбачається мінімізація цієї оцінки. Таким чином моделюється протидія раціонального опонента.

Оцінювання позиції виконується через функцію evaluateBoard[11]. Вона враховує матеріальну вартість фігур, позиційні таблиці, мобільність і фактори безпеки короля. Поєднання цих складових дає боту більш змістовну оцінку, ніж простий підрахунок матеріалу.

Матеріальна оцінка залишається базовою, оскільки в шахах втрата фігури істотно впливає на майбутні можливості. Водночас сама по собі вона не відображає активність фігур, контроль центру чи небезпеку для короля.

Позиційні таблиці використовуються для заохочення розміщення фігур на корисних клітинах. Наприклад, централізація коня або активніше розташування фігур може отримати додаткову вагу навіть без негайного матеріального здобутку.

Мобільність оцінюється через кількість доступних ходів. Більша мобільність зазвичай означає ширший вибір і меншу скутість позиції. У проєкті цей компонент має помірну вагу, щоб він доповнював матеріал, але не переважав його.

Таблиця 3.2 — Компоненти оцінювальної функції

Компонент	**Призначення**	**Вплив на поведінку бота**
Матеріал	Порівняння сумарної вартості фігур сторін.	Бот уникає невигідних втрат і прагне вигравати фігури.
Позиційні таблиці	Невеликі бонуси за корисні клітини для кожного типу фігур.	Фігури розміщуються активніше, а не лише зберігають матеріал.
Мобільність	Порівняння кількості легальних ходів.	Перевага надається позиціям із більшою свободою дій.
Безпека короля	Штраф за шах і бонус за локальний захист.	Бот уникає позицій, де король негайно вразливий.

Безпека короля враховується через штраф за шах і бонуси, пов’язані із захисним оточенням. Це дозволяє боту уникати позицій, у яких формально збережено матеріал, але король перебуває під значним тиском.

Для Alpha-Beta Pruning[5] важливим є порядок перегляду ходів. У проєкті використовується впорядкування, яке надає пріоритет матовим ходам, взяттям,

перетворенням і ходам із шахом. Такий порядок збільшує ймовірність раннього відсікання неперспективних гілок.

AI-модуль також збирає метрики пошуку: кількість відвіданих вузлів, кількість оцінювань і число відсічених гілок. Ці дані використовуються в дослідницькому розділі для порівняння Minimax[6] та Alpha-Beta Pruning[5].

Реалізація кількох ботів дає змогу порівнювати рівні складності в межах одного застосунку. Випадкова, жадібна й пошукова стратегії демонструють різні підходи до прийняття рішень у шаховій грі.

Під час пошуку бот не звертається до DOM або графічних компонентів. Його вхідними даними є стан ChessEngine, а вихідним результатом — обраний хід. Така ізоляція дозволяє досліджувати алгоритм незалежно від візуального шару.

Кожен кандидатний хід застосовується до клонованого рушія. Після цього алгоритм або переходить на наступну глибину, або викликає оцінювальну функцію. Ця схема повторюється рекурсивно до досягнення заданої межі пошуку або термінального стану.

Оцінка мату має пріоритет над звичайними числовими компонентами. Це необхідно, щоб бот не порівнював завершення партії з матеріальними перевагами меншого масштабу і не пропускав очевидну перемогу.

Упорядкування ходів не змінює шахової сили функції оцінювання напряму, але впливає на продуктивність Alpha-Beta Pruning[5]. Якщо перспективні ходи розглядаються першими, алгоритм швидше знаходить межі, які дозволяють відсікти інші гілки.

Зібрані під час пошуку метрики можна використовувати не лише для розділу дослідження, а й для налагодження. Раптове збільшення кількості вузлів або відсутність відсічень може вказувати на проблеми з порядком ходів або глибиною.

Обрана реалізація AI[7] відповідає масштабу вебзастосунку. Вона не використовує складні професійні методи на кшталт таблиць транспозицій, але

демонструє основні принципи і дозволяє порівнювати стратегії в реальному інтерфейсі.

RandomBot фактично слугує контрольним варіантом. Він показує, як поводиться система без позиційного аналізу, і дає базу для порівняння з більш осмисленими стратегіями.

GreedyBot демонструє перехід від випадкового вибору до локального критерію. Він уже реагує на матеріальні можливості, але не враховує, чи зможе суперник одразу використати слабкість після такого ходу.

MinimaxBot є найбільш важливим для дослідження, оскільки формує дерево майбутніх ходів. Його поведінка залежить від глибини, оцінювальної функції та оптимізації порядку ходів.

Під час аналізу кожного ходу бот повинен пам’ятати, яка сторона максимізує оцінку. Якщо бот грає білими, позитивна оцінка відповідає перевазі білих; якщо чорними, інтерпретація повинна бути узгодженою з обраною стороною.

Оцінювальна функція не повинна бути надто складною для браузерного застосунку. Якщо кожне оцінювання позиції буде дуже дорогим, навіть Alpha-Beta Pruning[5] не забезпечить комфортної швидкодії.

Тому в проєкті використано помірно деталізовану оцінку: матеріал, позиційні таблиці, мобільність і безпека короля. Такий набір достатній, щоб бот поводився осмислено, але не перевантажував систему.

Метрики пошуку дозволяють спостерігати за внутрішньою роботою AI[7]. Кількість вузлів показує розмір фактично переглянутого дерева, кількість оцінювань — звернення до evaluateBoard[11], а prunedBranches — ефект відсікання.

У практичній грі користувач бачить лише хід бота, але для дослідження важливо мати числове пояснення цього вибору. Саме тому метрики є частиною реалізації, а не лише допоміжним інструментом налагодження.

Під час вибору ходу MinimaxBot повинен порівнювати варіанти, що можуть мати близькі оцінки. У таких випадках порядок ходів або дрібні

компоненти оцінювальної функції можуть вплинути на остаточний вибір. Це не є помилкою, а типовою властивістю евристичного оцінювання.

Матеріальні значення фігур використовуються як базова шкала, на якій будуються інші компоненти. Якщо позиційні бонуси мають бути корисними, вони повинні бути співмірними з матеріальною оцінкою, але не настільки великими, щоб бот легко віддавав фігури за незначну активність.

Упорядкування ходів через перевірку captures, checks і promotions частково наближає пошук до шахової інтуїції. Людина також зазвичай спершу розглядає форсовані ходи: матові загрози, шахи та вигідні взяття. У програмі це використано для підвищення ефективності відсікання.

Оскільки AI[7] виконується на клієнті, важливо обмежувати обчислювальну складність. На відміну від серверного рушія, який перевіряє один хід за раз, бот аналізує багато позицій, тому навіть невелике подорожчання evaluateBoard[11] множиться на кількість листків дерева.

Реалізована система ботів створює основу для експериментів. Можна змінювати глибину, порядок ходів або ваги оцінки та спостерігати, як це впливає на метрики. Така властивість робить проєкт придатним не лише для гри, а й для навчального дослідження.

3.3 Реалізація багатокористувацького режиму

Багатокористувацький режим реалізовано через сервер Socket.IO[4], який обробляє кімнати та події гри. Кожна партія має власний roomId, стан ChessEngine і набір підключених гравців.

Під час створення кімнати сервер формує ідентифікатор і повертає його клієнту. Другий користувач може приєднатися за цим ідентифікатором, після чого партія переходить зі стану очікування до активної гри.

Кожен гравець пов’язується з конкретною стороною. Це дозволяє серверу перевіряти, чи має користувач право виконувати хід саме зараз. Якщо сторона не збігається з activeSide, сервер відхиляє дію.

Подія make_move містить координати початкової та кінцевої клітини, а також за потреби параметр promotionTo. Сервер не застосовує ці дані безпосередньо, а передає їх у ChessEngine для повної перевірки правил.

Після успішного ходу сервер надсилає обом клієнтам подію з оновленим станом. Завдяки цьому дошки гравців синхронізуються не за припущеннями клієнта, а за єдиним результатом серверної обробки.

Механізм reconnect підтримує повернення до активної партії після тимчасового розриву. Клієнт зберігає дані сесії, а сервер при повторному підключенні відновлює участь користувача й передає актуальну позицію.

У клієнтському сервісі socketService мережеві події винесено в окремий шар. Це дозволяє компонентам інтерфейсу викликати зрозумілі функції створення кімнати, приєднання або надсилання ходу, не працюючи безпосередньо з низькорівневими деталями Socket.IO[4].

Після створення кімнати клієнт зберігає службові дані сесії, зокрема ідентифікатор кімнати та сторону. Це потрібно не для довготривалого облікового запису, а для практичного сценарію повторного підключення під час тієї самої партії.

Стан очікування суперника є окремим етапом життєвого циклу кімнати. Поки друга сторона не приєдналася, сервер не повинен приймати звичайні ігрові ходи, оскільки партія ще не має двох учасників.

Коли обидва гравці приєднані, сервер змінює статус кімнати на активний. Надалі кожна подія make_move перевіряється з урахуванням цього статусу, що запобігає ходам у завершених або ще не готових кімнатах.

Після завершення партії стан кімнати може бути позначений як finished або ended залежно від сценарію. Це дає змогу серверу відрізняти нормальне завершення шахової гри від припинення через зовнішні обставини.

Таким чином багатокористувацький режим реалізує не лише обмін ходами, а повний життєвий цикл кімнати: створення, очікування, активну гру, тимчасові відключення, повторне підключення та завершення.

3.4 Інтерфейс користувача та розгортання

Інтерфейс користувача побудовано так, щоб основні режими були доступні з головного меню. Користувач може обрати гру проти бота або багатокористувацький режим, не змінюючи налаштування середовища.

Ігровий екран містить шахову дошку, інформацію про активну сторону, стан партії та елементи керування. У режимі проти бота додатково враховується стан обчислення ходу, щоб користувач розумів, коли AI[7] аналізує позицію.

Графічна дошка реалізована за допомогою PixiJS[2]. Вона масштабується відповідно до доступної області, зберігаючи пропорції клітин і зручність вибору фігур. Такий підхід важливий для адаптивного інтерфейсу.

У багатокористувацькому режимі інтерфейс показує статус кімнати, очікування суперника, результат приєднання та можливість повторного входу. Це робить мережеву взаємодію зрозумілою для користувача без технічних повідомлень.

Клієнт розгорнуто на GitHub Pages[8] як статичний вебзастосунок. Серверну частину розгорнуто на Railway[9], де вона виконує обробку Socket.IO-подій і підтримує активні кімнати.

Розділення розгортання відповідає архітектурі системи: клієнт обслуговує інтерфейс і локальні сценарії, а сервер залишається доступним для мережевих партій та авторитетної перевірки ходів.

3.5 Методика тестування системи

Тестування системи виконувалося за модулями, оскільки різні частини проєкту мають різні критерії коректності. Для шахового рушія головним є відповідність правилам, для AI[7] — стабільність вибору ходу, для мережевого режиму — синхронізація й серверна перевірка.

Перевірка рушія охоплювала стандартні та спеціальні ситуації: звичайні переміщення фігур, шах, мат, пат, рокірування, взяття на проході та перетворення пішака. Окремо аналізувалися ходи, які геометрично можливі, але відкривають власного короля.

Для тестування генерації ходів використовувалися позиції з різною кількістю фігур і перешкод. Це дало змогу перевірити, чи зупиняються лінійні фігури на першій зайнятій клітині та чи правильно обробляються взяття.

AI[7] перевірявся через запуск ботів у типових позиціях і аналіз зібраних метрик. Для MinimaxBot оцінювалася кількість відвіданих вузлів, число викликів оцінювальної функції та ефект від Alpha-Beta Pruning[5].

Таблиця 3.3 — Напрями тестування розробленої системи

Напрям	**Що перевіряється**	**Очікуваний результат**
Шаховий рушій	Легальні ходи, шах, мат, пат, спеціальні правила.	Рушій повертає лише допустимі ходи та правильний статус партії.
AI	Вибір ходу ботами, оцінювання, метрики пошуку.	Бот повертає легальний хід і не змінює реальну позицію під час аналізу.
Мережевий режим	Кімнати, черговість, синхронізація, reconnect.	Сервер підтверджує лише коректні ходи й розсилає єдиний стан.
Інтерфейс	Меню, масштабування дошки, блокування вводу.	Користувач бачить актуальний стан і не може виконати заборонену дію.
Розгортання	Доступність клієнта і сервера, CORS, health endpoint.	Клієнт підключається до сервера у production-середовищі.

Мережевий режим тестувався через сценарії створення кімнати, приєднання другого гравця, виконання ходів за чергою та спроби некоректних дій. Особливу увагу приділено тому, щоб сервер відхиляв ходи неактивної сторони.

Сценарії повторного підключення перевіряли здатність системи відновити участь гравця після розриву з’єднання. Успішним результатом вважалося повернення до тієї самої кімнати з актуальним станом дошки.

Адаптивний інтерфейс перевірявся на різних розмірах вікна. Основними критеріями були збереження пропорцій дошки, доступність елементів керування та відсутність перекриття інформаційних блоків.

Таблиця 3.4 — Приклади тестових сценаріїв для шахового рушія

Сценарій	**Вхідні умови**	**Очікуваний результат**
Подвійний хід пішака	Пішак на початковій горизонталі, клітини попереду порожні.	Хід на дві клітини дозволений, lastMove позначає двокроковий хід.
Рокірування	Король і тура не рухалися, шлях вільний, клітини не атаковані.	Король і тура переміщуються одночасно.
Заборона самошаху	Фігура прикриває лінію атаки на власного короля.	Хід цієї фігури не входить до списку легальних.
Мат	Сторона під шахом і не має легальних ходів.	makeMove повертає статус checkmate.
Пат	Сторона не під шахом і не має легальних ходів.	makeMove повертає статус stalemate.

Тестування не обмежувалося одиничними перевірками, оскільки помилки в шаховій системі часто проявляються після кількох послідовних ходів. Тому важливим елементом було проходження повних коротких партій у різних режимах.

Негативні сценарії також тестувалися окремо, тобто спроби зробити хід, який не повинен бути прийнятий. Це переміщення фігури проти правил, переміщення на свою фігуру, переміщення під шах і спроба зробити хід, коли це не твоя черга.

Спеціальні правила можуть бути протестовані не тільки для позитивного випадку, але й для випадку, коли правило не повинно застосовуватися. Наприклад, рокіровка не може бути виконана після того, як король або тура вже рухалися, а взяття на проході повинно бути дозволено тільки після того, як пішак зробив хід на два поля.

Ми довели в мережевому тестуванні, що клієнти отримують однаковий знімок після кожного підтвердженого ходу. Це важливіше, ніж локальне відображення окремого клієнта, оскільки багатокористувацька гра втрачає сенс, якщо позиції виходять з рівноваги.

Тестування ШІ включало порівняння поведінки різних стратегій в одній і тій самій позиції. Очікувалося, що RandomBot покаже дуже обмежену логіку, GreedyBot реагуватиме на матеріальні рішення, а MinimaxBot враховуватиме відповіді суперника.

Після розгортання була перевірена взаємодія між клієнтом GitHub Pages[8] і сервером Railway[9]. Ми звернули увагу на налаштування CORS і стабільність з'єднання Socket.IO[4], оскільки система, яка працює локально, може піддаватися різним мережевим умовам після публікації.

Для двигуна позиції з невеликою кількістю фігур корисні для ізоляції одного правила. Наприклад, перевірка руху коня може бути виконана без додавання інших фігур і, таким чином, ускладнення аналізу. Перевірка на шах і мат, з іншого боку, вимагає більш складних конфігурацій, де всі фігури взаємодіють. Такий сценарій дозволить нам побачити, чи врахував двигун блокування, захоплення атакуючої фігури та втечу короля.

Тестування на стороні сервера повинно включати правильні ходи, а також спроби порушити протокол. Наприклад, клієнт може надіслати хід з неправильним квадратом або повторити старий хід, коли стан гри змінився. Для сценаріїв Socket.IO[4] нам потрібно перевірити порядок подій. Створення кімнати, приєднання, початок гри, хід і завершення повинні відбуватися в передбачуваному порядку; інакше клієнт може мати інший стан у грі, який не відображається на його екрані.

Тестування інтерфейсу повинно також враховувати не тільки дошку, але й поведінку клієнта під час помилок. Якщо сервер відхиляє хід, клієнт повинен бути в правильній позиції, а не в неправильній.

У дослідженні ШІ важливо запускати алгоритми на одних і тих самих позиціях. Тільки тоді можна добре порівняти кількість вузлів, ефект обрізання та

зміну обраного ходу в залежності від глибини. Результати тестів повинні інтерпретуватися відповідно до масштабу проекту. Ми не прагнемо довести турнірну силу бота; однак, ми хочемо показати правильність реалізації та дослідити властивості алгоритму.

3.6 Висновки до розділу 3

У третьому розділі описано реалізацію основних модулів: шахової логіки, AI-суперника, багатокористувацького режиму, інтерфейсу та тестування. Показано, що функціональні частини системи пов’язані через спільний ChessEngine і працюють у єдиній моделі стану.

РОЗДІЛ 4 ДОСЛІДЖЕННЯ АЛГОРИТМІВ ШТУЧНОГО ІНТЕЛЕКТУ

4.1 Алгоритм Minimax

У розробленій системі алгоритм Minimax[6] використовується як основний механізм вибору ходу інтелектуальним суперником. Його застосування пов’язане з тим, що шахи є грою з повною інформацією: обидві сторони бачать однакову позицію, а результат кожного ходу визначається формальними правилами. Тому вибір ходу можна подати як задачу пошуку в дереві можливих станів.

У межах реалізованого проєкту алгоритм працює не з абстрактними позиціями, а з клонованими екземплярами ChessEngine. Кожна вершина дерева пошуку відповідає певному стану шахової дошки, а кожне ребро — одному легальному ходу, отриманому з рушія. Завдяки цьому дерево пошуку не містить неможливих або заборонених правилами ходів. Це важливо, оскільки AI[7] аналізує саме той простір рішень, який доступний реальному гравцеві.

Формально стан шахової партії можна позначити як s, множину легальних ходів у цьому стані — як M(s), а результат застосування ходу m до стану s — як Result(s,m). Глибина пошуку позначається як d. Тоді значення позиції для алгоритму Minimax[6] можна подати у такому вигляді:

$$V(s,d) = \text{Eval}(s), \text{ якщо } d = 0 \text{ або } s \text{ є термінальним станом;}$$

$$V(s,d) = \max V(\text{Result}(s,m), d - 1), \text{ якщо хід виконує AI[7];}$$

$$V(s,d) = \min V(\text{Result}(s,m), d - 1), \text{ якщо хід виконує суперник,}$$

де Eval(s) — функція оцінювання позиції. Вона повертає числове значення, яке характеризує вигідність поточного стану для бота.

У практичній реалізації Minimax[6] працює за таким принципом. Якщо хід виконує бот, алгоритм вибирає варіант із максимальною оцінкою. Якщо хід виконує суперник, передбачається, що він обере найгірший для бота варіант,

тобто позицію з мінімальною оцінкою. Таким чином моделюється раціональна протидія двох сторін.

Приклад реалізації алгоритму Minimax[6] мовою JavaScript наведено нижче:

```
function minimax(engine, depth, maximizingPlayer) {
    if (depth === 0 || engine.isGameOver()) {
        return evaluateBoard(engine);
    }

    const moves = engine.getAllLegalMoves();

    if (maximizingPlayer) {
        let bestValue = -Infinity;

        for (const move of moves) {
            const newEngine = engine.clone();
            newEngine.makeMove(move);

            const value = minimax(newEngine, depth - 1, false);
            bestValue = Math.max(bestValue, value);
        }

        return bestValue;
    }

    let bestValue = Infinity;

    for (const move of moves) {
        const newEngine = engine.clone();
        newEngine.makeMove(move);

        const value = minimax(newEngine, depth - 1, true);
        bestValue = Math.min(bestValue, value);
    }

    return bestValue;
}
```

У цьому фрагменті engine відповідає поточному екземпляру шахового рушія, depth — кількості півходів, які залишилося проаналізувати, а maximizingPlayer визначає, чи поточний рівень дерева відповідає ходу бота. Метод clone() створює копію поточного стану, щоб симуляція не змінювала реальну партію. Метод makeMove() застосовує хід до клонованої позиції, а evaluateBoard()[11] повертає числову оцінку стану дошки.

Окремо можна подати функцію вибору найкращого ходу, яка перебирає всі легальні ходи бота і вибирає той, що має найкращу оцінку після рекурсивного аналізу:

```
function findBestMove(engine, depth) {
    const moves = engine.getAllLegalMoves();
    let bestMove = null;
    let bestValue = -Infinity;

    for (const move of moves) {
        const newEngine = engine.clone();
        newEngine.makeMove(move);

        const value = minimax(newEngine, depth - 1, false);

        if (value > bestValue) {
            bestValue = value;
            bestMove = move;
        }
    }

    return bestMove;
}
```

У такій реалізації хід не оцінюється сам по собі. Алгоритм оцінює позицію, яка виникає після цього ходу. Це важливо, оскільки один і той самий хід може бути сильним або слабким залежно від загального стану дошки, розташування фігур і можливих відповідей суперника.

Термінальними станами в розробленій системі є мат, пат або інше завершення партії. Мат оцінюється значно сильніше, ніж звичайна матеріальна чи позиційна перевага. Це потрібно для того, щоб бот не ігнорував можливість завершити партію або уникнути негайної поразки. Пат розглядається як нічийний результат, тому його оцінка не прирівнюється до виграшу чи програшу.

Глибина пошуку є одним із головних параметрів Minimax[6]. Вона визначає, на скільки півходів уперед бот аналізує позицію. Наприклад, глибина 1 означає, що бот оцінює лише власні можливі ходи. Глибина 2 враховує також відповідь суперника. Глибина 3 дозволяє оцінити власний наступний хід після відповіді суперника.

Залежність якості аналізу від глибини можна подати так (табл. 4.1):

Таблиця 4.1 – Залежність якості аналізу від глибини пошуку

Глибина пошуку	**Що аналізує бот**	**Характеристика рішення**
1	Лише власний хід	Швидке рішення, але без урахування відповіді суперника
2	Власний хід і відповідь суперника	Краще врахування ризиків і можливих втрат
3	Власний хід, відповідь суперника і наступний власний хід	Більш змістовний прогноз, але значно більші обчислювальні витрати

Основним недоліком Minimax[6] є швидке зростання кількості позицій, які потрібно проаналізувати. Якщо середня кількість легальних ходів у позиції дорівнює b, а глибина пошуку становить d, то орієнтовна кількість вузлів дерева становить:

$$N(d) \approx b^d$$

Для шахів значення b може бути досить великим. Навіть у початковій позиції існує 20 легальних ходів, а в розвинених позиціях їхня кількість може бути більшою. Тому збільшення глибини лише на один рівень суттєво підвищує кількість симуляцій.

У межах дослідження було проаналізовано залежність кількості відвіданих вузлів від глибини пошуку. Результати наведено в таблиці 4.2.

Таблиця 4.2 – Приклади пошуку для різних стартових позицій

Позиція	**Глибина пошуку**	**Кількість легальних ходів**	**Відвідано вузлів**	**Оцінено позицій**
Початкова позиція	1	20	20	20
Початкова позиція	2	20	420	400
Початкова позиція	3	20	3272	2852
Позиція після 1.e4 e5	1	29	29	29
Позиція після 1.e4 e5	2	29	864	835
Позиція після 1.e4 e5	3	29	4600	3735
Розвинена позиція	1	33	33	33
Розвинена позиція	2	33	963	930
Розвинена позиція	3	33	4138	3173

За наведеними даними можна побудувати графік залежності кількості відвіданих вузлів від глибини пошуку. На осі X відкладається глибина пошуку, а на осі Y — кількість відвіданих вузлів. Такий графік показує, що збільшення глибини має нелінійний характер. Кількість позицій зростає не поступово, а стрибкоподібно, оскільки кожен новий рівень дерева породжує додаткові варіанти ходів.

Для початкової позиції кількість відвіданих вузлів зростає з 20 на глибині 1 до 420 на глибині 2 і до 3272 на глибині 3. У позиції після 1.e4 e5 кількість вузлів зростає з 29 до 864 і 4600 відповідно. У розвиненій позиції, де доступно 33 легальні ходи, на глибині 3 було відвідано 4138 вузлів.

Отримані результати підтверджують, що Minimax[6] є зрозумілим і формально обґрунтованим методом вибору ходу, однак його пряме використання має суттєві обмеження. На малих глибинах алгоритм працює швидко, але не бачить достатньо далеко. На більших глибинах якість рішення зростає, проте різко збільшується кількість обчислень.

У контексті браузерної шахової системи це особливо важливо. Якщо бот аналізує надто багато позицій, користувач відчуває затримку між власним ходом і відповіддю AI[7]. Тому Minimax[6] доцільно використовувати разом з оптимізаціями, зокрема з Alpha-Beta Pruning[5], який дозволяє скоротити кількість гілок дерева без зміни логіки вибору ходу.

Таким чином, Minimax[6] у розробленій системі виконує роль базового алгоритму прийняття рішень. Його перевагою є прозорість, формальна зрозумілість і можливість пояснити вибір ходу через аналіз дерева позицій. Основним недоліком є комбінаторне зростання кількості вузлів, що обмежує практичну глибину пошуку. Саме тому подальший аналіз ефективності AI[7] у роботі виконується з урахуванням Alpha-Beta Pruning[5] як методу оптимізації пошуку.

4.2 Alpha-Beta Pruning

Alpha-Beta Pruning[5] застосовано для скорочення дерева Minimax[6] без зміни кінцевого результату пошуку за однакової глибини й порядку оцінювання. Алгоритм зберігає межі alpha та beta, які описують уже знайдені найкращі альтернативи для сторін.

Якщо під час аналізу стає зрозуміло, що певна гілка не може вплинути на остаточний вибір, її подальше розгортання припиняється. Це особливо важливо для шахів, де кількість можливих позицій зростає експоненційно.

У проєкті Alpha-Beta Pruning[5] поєднано з упорядкуванням ходів. Першими розглядаються матові ходи, взяття, promotion і шахи. Якщо сильні ходи аналізуються раніше, межі alpha та beta оновлюються швидше, а кількість відсічень збільшується.

Відсікання не є наближенням у сенсі зміни правил пошуку. Воно вилучає лише ті варіанти, які за логікою Minimax[6] не можуть бути обрані за оптимальної гри сторін. Тому Alpha-Beta[10] зберігає якість рішення, але зменшує обчислювальні витрати.

У метриках AI[7] фіксується кількість відсічених гілок, що дає змогу кількісно оцінити ефективність оптимізації. Ці дані використовуються для порівняння позицій із різною кількістю ходів і різним ступенем тактичної насиченості.

Практичний результат Alpha-Beta Pruning[5] полягає в тому, що бот може аналізувати позиції на більшій глибині без неприйнятної затримки інтерфейсу. Для вебзастосунку це є критичною умовою зручності гри.

Параметр alpha відображає найкращу вже знайдену оцінку для сторони, що максимізує, а beta — найкращу межу для сторони, що мінімізує. Якщо ці межі перетинаються, подальший аналіз гілки втрачає сенс.

У практичній реалізації межі передаються рекурсивно разом із глибиною та інформацією про те, чи поточний рівень максимізує оцінку. Після кожного розглянутого ходу відповідна межа оновлюється.

Якщо на рівні максимізації поточна оцінка стає не меншою за beta, гілка відсікається, оскільки мінімізуюча сторона раніше вже має кращу альтернативу. Аналогічно, на рівні мінімізації відсікання відбувається, коли оцінка не перевищує alpha.

Цей механізм особливо ефективний тоді, коли перші розглянуті ходи близькі до найкращих. Тому впорядкування ходів у проєкті є не допоміжною прикрасою, а важливою частиною продуктивності AI[7].

Варто підкреслити, що Alpha-Beta[10] не робить бота “розумнішим” у сенсі іншої оцінки позиції. Він дозволяє швидше отримати той самий результат, який дав би повний Minimax[6] на відповідній глибині.

Для користувача перевага проявляється у меншій затримці відповіді. Для дослідження перевага проявляється в числових метриках, які показують, скільки гілок було відкинуто без повного розгортання.

Alpha-Beta Pruning[5] можна інтерпретувати як використання вже знайдених доказів непотрібності певних варіантів. Якщо одна сторона вже має кращий вибір, немає сенсу детально аналізувати гілку, яка гарантовано не буде прийнята за раціональної гри.

На практиці це означає, що алгоритм не витрачає час на позиції, які не можуть вплинути на рішення кореневого ходу. У шаховому дереві таких позицій може бути дуже багато, особливо коли сильний хід знаходиться рано.

Важливо, що Alpha-Beta[10] не усуває потребу в оцінювальній функції. Він лише зменшує кількість позицій, де ця функція викликається. Тому якість рішення все одно залежить від того, наскільки evaluateBoard[11] відображає корисні шахові ознаки.

Для дипломного дослідження Alpha-Beta[10] зручний тим, що його ефект вимірюється кількісно. Показник prunedBranches дозволяє продемонструвати не

абстрактну перевагу, а конкретне скорочення дерева пошуку в реалізованій системі.

У вебзастосунку оптимізація пошуку має подвійне значення: вона зменшує час відповіді бота та знижує навантаження на браузер. Це робить алгоритм не лише теоретично важливим, а й практично необхідним.

4.3 Вплив глибини пошуку та якість рішень

Глибина пошуку істотно впливає на якість рішень AI-суперника. На малій глибині бот переважно бачить безпосередню вигоду або втрату, тоді як глибший пошук дозволяє врахувати відповідь суперника.

У початковій позиції кількість доступних ходів є помірною, але вже на другому й третьому рівнях дерево швидко збільшується. Це підтверджує, що навіть стандартний початок партії створює значне навантаження для повного перебору.

Після ходів 1.e4 e5 кількість легальних варіантів для білих зростає, оскільки відкриваються нові лінії для ферзя та слонів. Для Minimax[6] це означає збільшення ширини дерева й більшу потребу в ефективному відсіканні.

Таблиця 4.3 — Метрики роботи MinimaxBot у тестових позиціях

Позиція	**Глибина**	**Легальних ходів**	**Відвідано вузлів**	**Оцінювань**	**Відсічено гілок**
Початкова позиція	1	20	20	20	0
Початкова позиція	2	20	420	400	0
Початкова позиція	3	20	3272	2852	6050
Після 1.e4 e5	1	29	29	29	0
Після 1.e4 e5	2	29	864	835	0
Після 1.e4 e5	3	29	4600	3735	21089
Розвинута позиція	1	33	33	33	0
Розвинута позиція	2	33	963	930	0
Розвинута позиція	3	33	4138	3173	27367

У розвиненій позиції після 1.e4 e5 2.Nf3 Nc6 3.Bc4 Nf6 пошук стає тактично насиченішим. Бот отримує більше активних ходів, включно з можливими атаками на слабкі пункти, тому якість оцінювальної функції має більше значення.

На глибині 1 бот фактично порівнює лише позиції після власного ходу. Це може давати швидку відповідь, але не захищає від очевидної контратаки суперника на наступному ході.

На глибині 2 алгоритм уже враховує одну відповідь опонента. Такий рівень помітно покращує захисну поведінку, оскільки бот уникає ходів, які одразу призводять до матеріальної втрати або шахової загрози.

Глибина 3 дозволяє побачити власну відповідь після реакції суперника. У практичній грі це дає змістовніші рішення, але суттєво збільшує кількість симуляцій і навантаження на браузер.

Зібрані метрики демонструють, що зростання глибини не є лінійним. Кожен додатковий півхід множить кількість позицій на число доступних ходів, тому без Alpha-Beta Pruning[5] глибший аналіз швидко стає неприйнятним.

Якість рішення залежить не тільки від глибини, а й від оцінювальної функції. Якщо функція неправильно оцінює позицію, навіть глибший пошук може послідовно обирати слабші варіанти.

Для дипломного проєкту обрана глибина є компромісом між силою бота та інтерактивністю. Мета полягає не у створенні професійного шахового рушія, а в дослідженні практичної роботи пошукових алгоритмів у вебсередовищі.

У початковій позиції на глибині 1 бот оцінює лише 20 ходів. Такий аналіз є швидким, але не відображає відповідей чорних і тому має обмежену стратегічну цінність.

На глибині 2 у початковій позиції кількість відвіданих вузлів збільшується до сотень, оскільки для кожного ходу білих розглядаються відповіді чорних. Це вже дає змогу уникати частини очевидних помилок.

На глибині 3 у тих самих умовах кількість оцінювань зростає ще більше, але Alpha-Beta Pruning[5] починає давати помітний ефект. Саме на цьому рівні стає видно практичну цінність відсікання для інтерактивного застосунку.

У позиції після 1.e4 e5 бот має більше активних можливостей, тому вибір ходу стає залежним не лише від матеріалу, а й від позиційних факторів. Це демонструє роль оцінювальної функції в ранній середині гри.

Розвинена позиція з виведеними фігурами показує, що кількість легальних ходів і тактичних мотивів зростає. У таких умовах упорядкування

ходів допомагає Alpha-Beta[10] швидше знаходити сильні гілки та відкидати слабші.

Отже, глибина пошуку повинна підбиратися з урахуванням продуктивності середовища. Для браузерного AI[7] надмірне збільшення глибини може погіршити користувацький досвід сильніше, ніж покращить якість гри.

Глибина пошуку може змінювати характер гри бота. На малій глибині він схильний до тактичних помилок, оскільки бачить лише найближчий результат. На більшій глибині бот починає уникати ходів, які одразу караються суперником.

Водночас збільшення глибини не гарантує ідеального вибору. Якщо оцінювальна функція недооцінює небезпеку або переоцінює позиційний бонус, глибший пошук лише послідовніше застосує цю неточну оцінку.

У початковій позиції різниця між деякими ходами може бути невеликою, тому бот часто обирає варіанти, що мають близьку оцінку. У таких умовах важливо аналізувати не тільки сам хід, а й метрики пошуку.

У позиціях із відкритими лініями та можливими шахами глибина має більший вплив, оскільки наслідки ходу проявляються швидше. Саме такі позиції добре показують різницю між жадібним вибором і Minimax[6].

Для користувацького досвіду важливо підібрати глибину так, щоб бот не створював довгої паузи після кожного ходу. Навіть сильніший хід може бути небажаним, якщо очікування руйнує інтерактивність гри.

Отже, якість AI[7] у вебсистемі визначається балансом трьох факторів: глибини пошуку, точності оцінювання та часу відповіді. Дипломний проєкт демонструє цей баланс на практичному прикладі.

На прикладі досліджених позицій видно, що збільшення глибини змінює не тільки кількість вузлів, а й характер обраного ходу. Бот починає віддавати перевагу варіантам, які можуть не мати негайної вигоди, але краще витримують відповідь суперника.

Разом з тим глибина не може розглядатися ізольовано від порядку ходів. На однаковій глибині Alpha-Beta[10] може бути значно ефективнішим або менш ефективним залежно від того, наскільки вдало впорядковано кандидатні ходи.

У початковій позиції глибший пошук часто не створює драматичної різниці у виборі, оскільки багато ходів мають близьку оцінку. У тактичних позиціях різниця стає виразнішою, адже короткі комбінації можуть бути знайдені вже на глибині кількох півходів.

Для бакалаврського проєкту важливо показати не максимальну силу AI[7], а залежність між параметрами алгоритму та поведінкою системи. Саме тому в роботі аналізуються метрики вузлів, оцінювань і відсічень.

Отримані результати підтверджують класичну властивість пошукових алгоритмів: якість рішення підвищується разом із глибиною, але ціна такого підвищення зростає значно швидше. Практична система повинна знаходити компроміс.

4.4 Зменшення кількості аналізованих позицій

Зменшення кількості аналізованих позицій є основною перевагою Alpha-Beta Pruning[5] порівняно з повним Minimax[6]. У шахах це має особливе значення, оскільки кожна позиція зазвичай породжує десятки легальних ходів.

Ефективність відсікання залежить від порядку перегляду ходів. Якщо сильні або тактично важливі ходи аналізуються раніше, межі пошуку швидше звужуються, і слабші гілки можуть бути відкинуті без повного розгортання.

У реалізованому боті порядок ходів формується з урахуванням матових можливостей, взяття фігур, promotion і шахів. Це не гарантує ідеального порядку, але значно краще за випадковий перегляд у більшості практичних позицій.

Метрики показують, що на малих глибинах відсікання може бути незначним або відсутнім, оскільки дерево ще недостатньо розгорнуте.

Найпомітніший ефект проявляється на глибині 3 і вище, де з’являється багато альтернативних піддерев.

Скорочення аналізу не повинно порушувати коректність результату. Alpha-Beta[10] відсікає лише ті гілки, що вже не можуть покращити рішення сторони з урахуванням знайдених меж. Тому якість Minimax-рішення зберігається.

Практичні вимірювання підтверджують, що Alpha-Beta Pruning[5] найбільш корисний у позиціях із розгалуженим деревом. Що більше альтернатив має сторона, то більше потенційних гілок може бути відсічено після знаходження достатньо сильної відповіді.

Важливо, що ефективність оптимізації залежить від структури позиції. У спокійній позиції без очевидних тактичних ходів відсікання може бути менш вираженим, тоді як у позиціях із взяттями та шахами сильні ходи швидше формують корисні межі.

Порівняння метрик nodesVisited, evaluations[11] і prunedBranches дозволяє розділити фактичну кількість переглянутих вузлів і кількість позицій, які були пропущені завдяки межам. Це робить дослідження не описовим, а кількісним.

Для реалізованої системи зменшення кількості аналізованих позицій прямо впливає на відчуття швидкодії. Користувач не аналізує метрики під час звичайної гри, але помічає затримку між своїм ходом і відповіддю бота.

Таким чином Alpha-Beta Pruning[5] виконує роль не лише алгоритмічної оптимізації, а й засобу підтримання інтерактивності інтерфейсу. Це особливо важливо для JavaScript-застосунку, де пошук виконується в середовищі браузера.

Порівняння Minimax[6] і Alpha-Beta Pruning[5] доцільно виконувати не лише за кінцевим ходом, а й за шляхом його отримання. Якщо обидва алгоритми обирають той самий хід, але один переглядає суттєво менше позицій, оптимізація має практичну цінність.

Кількість відсічених гілок залежить від того, наскільки швидко алгоритм знаходить хорошу альтернативу. Тому move ordering у проєкті фактично

підсилює Alpha-Beta[10], хоча сам по собі не змінює математичної суті Minimax[6].

У шахових позиціях із багатьма взяттями впорядкування зазвичай корисніше, бо матеріально значущі ходи швидше формують високі або низькі оцінки. У спокійних позиціях ефект може бути скромнішим.

Скорочення кількості позицій також зменшує кількість викликів clone і applyMove. Це важливо, оскільки саме створення копій рушія й симуляція ходів становлять значну частину обчислювальних витрат.

У браузері обчислювальні ресурси обмежені не лише процесором, а й потребою зберігати плавність інтерфейсу. Тому зменшення дерева пошуку має прямий вплив на відчуття якості застосунку.

Alpha-Beta Pruning[5] у цьому проєкті можна розглядати як приклад алгоритмічної оптимізації, що зберігає результат, але змінює вартість його отримання. Це робить його особливо придатним для дослідження в бакалаврській роботі.

4.5 Обмеження оцінювальної функції

Оцінювальна функція є наближенням, а не повним розумінням шахової позиції. Вона перетворює складний стан дошки на числове значення, тому неминуче втрачає частину стратегічного контексту.

Матеріальний компонент добре відображає грубі переваги, але не пояснює якість розташування фігур. Позиція з рівним матеріалом може бути виграшною або програшною через активність, слабкість короля чи контроль ключових клітин.

Позиційні таблиці допомагають врахувати типові принципи, однак вони є статичними. Одна й та сама клітина може бути сильною в одній позиції та слабкою в іншій залежно від структури пішаків і планів сторін.

Мобільність є корисним показником свободи дій, але її також потрібно обмежувати. Велика кількість ходів не завжди означає перевагу, якщо ці ходи не створюють загроз або не поліпшують захист короля.

Безпека короля у спрощеній функції оцінюється за обмеженою кількістю ознак. Повноцінний шаховий рушій міг би враховувати відкриті лінії, координацію атакуючих фігур і довгострокові загрози, але це виходить за межі дипломного прототипу.

Через обмеження оцінювання бот може іноді обирати ходи, які виглядають логічними для функції, але не є найкращими з шахової точки зору. Це особливо помітно на малих глибинах, коли алгоритм не бачить віддалених наслідків.

Разом із тим обрана оцінювальна функція достатня для дослідницької мети роботи. Вона дозволяє порівнювати стратегії, аналізувати вплив глибини та демонструвати практичну користь Alpha-Beta Pruning[5].

Однією з типових проблем спрощених оцінювальних функцій є так званий горизонт пошуку. Якщо негативний наслідок ходу виникає за межами заданої глибини, бот може не побачити його й обрати зовні привабливий варіант.

Інша проблема полягає у співвідношенні ваг. Якщо мобільність або позиційний бонус мають занадто велику вагу, бот може переоцінити активність і недооцінити матеріальні втрати. Якщо вага занадто мала, функція майже зводиться до підрахунку фігур.

У професійних рушіях оцінювання часто залежить від фази гри: дебют, мітельшпіль і ендшпіль мають різні пріоритети. У дипломному проєкті використано універсальну функцію, що спрощує реалізацію, але не враховує всіх фазових особливостей.

Попри ці обмеження, функція evaluateBoard[11] дає достатньо стабільну основу для порівняння алгоритмів. Оскільки одна й та сама функція використовується в усіх вимірюваннях, різниця в метриках насамперед відображає роботу пошуку та відсікання.

Для подальшого розвитку доцільно було б додати окремі ваги для дебюту й ендшпілю, а також більш детальний аналіз безпеки короля. Це могло б підвищити якість гри без радикальної зміни архітектури AI[7].

Оцінка мобільності може бути неоднозначною. Наприклад, сторона може мати багато ходів, але більшість із них не поліпшує позицію. Тому мобільність у функції оцінювання повинна залишатися допоміжним, а не головним критерієм.

Безпека короля також має контекстну природу. Король у центрі дошки зазвичай небезпечний у дебюті й мітельшпілі, але в ендшпілі активний король може бути перевагою. Універсальна функція не повністю відображає цю зміну ролі.

Матеріальна оцінка не враховує якість фігур. Пасивна тура може мати ту саму номінальну вартість, що й активна, однак її реальний вплив на партію буде іншим. Позиційні таблиці частково компенсують це, але не повністю.

Для дослідження важливо, що обмеження функції не приховуються. Навпаки, вони пояснюють, чому AI-суперник є демонстраційним і дослідницьким, а не професійним шаховим рушієм.

Подальше вдосконалення функції оцінювання може виконуватися поступово. Завдяки модульності AI[7] можна додавати нові компоненти оцінки без зміни ChessEngine або мережевої архітектури.

4.6 Висновки за розділом 4

У четвертому розділі досліджено роботу Minimax[6] і Alpha-Beta Pruning[5] у контексті реалізованої шахової системи. Встановлено, що збільшення глибини підвищує якість рішень, але різко збільшує кількість позицій, а Alpha-Beta Pruning[5] разом з упорядкуванням ходів істотно зменшує обсяг пошуку без зміни логіки Minimax[6].

Таблиця 4.4 — Якісне порівняння стратегій вибору ходу

Стратегія	**Принцип роботи**	**Переваги**	**Обмеження**
RandomBot	Випадковий вибір серед легальних ходів.	Мінімальні обчислювальні витрати.	Відсутність позиційного аналізу.
GreedyBot	Пріоритет матеріально вигідних захоплень.	Краща тактична поведінка у простих ситуаціях.	Не враховує відповіді суперника.
Minimax	Пошук у дереві ходів до заданої глибини.	Прогнозування наслідків ходів.	Висока обчислювальна складність.
Minimax з Alpha-Beta	Пошук Minimax з відсіканням неперспективних гілок.	Менше аналізованих позицій за тієї самої глибини.	Ефективність залежить від порядку ходів.

РОЗДІЛ 5 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ ТА НАПРЯМИ РОЗВИТКУ

5.1 Оцінювання відповідності поставленим завданням

Отримані результати слід оцінювати, порівнюючи їх з початковими завданнями дипломного проекту. Розроблена нами система містить усі ключові елементи, необхідні для гри в інтелектуальні багатокористувацькі шахи в Інтернеті.

Шаховий двигун генерує легальні ходи та перевіряє на шах, мат, пат, спеціальні правила та управління станом гри. Він слугує ідеальною платформою для клієнта, сервера та штучного інтелекту. Противник зі штучним інтелектом представлений кількома стратегіями, і ми можемо порівняти простий випадковий вибір, жадібну тактику та пошук Minimax[6] з оптимізацією обрізанням альфа-бета.

Набір відповідає дослідницькій меті роботи.

Адаптивний інтерфейс був розроблений шляхом поєднання React[1] та PixiJS[2]. Дошка масштабується до різних розмірів екранів, а функції меню та мережевого режиму забезпечують доступ до основних користувацьких сценаріїв.

Дослідження Minimax[6] та обрізання альфа-бета було проведено за допомогою пошукового бота та збору метрик. Це відкрило шлях від теоретичного опису алгоритмів до аналізу їх поведінки в створеній системі. В цілому, реалізація відповідає набору завдань не лише формально, але й у функціональному сенсі: модулі взаємодіють один з одним, підтримують повний цикл шахової гри та демонструють вивчені алгоритмічні властивості.

Таблиця 6.1 — Відповідність реалізації поставленим завданням

Завдання	**Стан виконання**	**Підтвердження у системі**
Розробка шахового рушія	Виконано	ChessEngine, генерація ходів, спеціальні правила, статуси партії.
Реалізація AI-суперника	Виконано	RandomBot, GreedyBot, MinimaxBot, оцінювальна функція.
Багатокористувацький режим	Виконано	Кімнати Socket.IO, create_room, join_room, move_applied.
Серверна валідація	Виконано	Перевірка activeSide, належності до кімнати і legality через ChessEngine.
Адаптивний інтерфейс	Виконано	React-екрани, масштабування PixiJS-сцени, responsive UI.
Дослідження алгоритмів	Виконано	Метрики MinimaxBot для різних глибин і позицій.

5.2 Переваги розробленого рішення

Основною перевагою є спільний шаховий двигун для різних частин проекту. Це забезпечує однакове тлумачення правил у локальній грі, пошуку ШІ та перевірці сервером мережі.

Другою перевагою є авторитетна серверна модель багатокористувацького режиму. Клієнт не змінює гру самостійно, але сервер має підтвердити правильний стан гри, тому менш ймовірно, що клієнт буде десинхронізований.

Третьою перевагою є наявність багатьох стратегій ШІ та метрик пошуку. Це перетворює розробку на гру, а також на дослідницький інструмент для вивчення впливу глибини та оптимізації на рішення. Існують також переваги структури програмного забезпечення. Всі ключові модулі мають конкретну роль, і реалізація чітко пояснена в дисертації. Рішення застосовне до обох сценаріїв гри: комп'ютерного гравця та іншого користувача. Це тому, що архітектура різноманітна, оскільки обидві ці гри грають у шахи.

Система має дослідницьку цінність завдяки метрикам ШІ. Користувач (або розробник) може бачити не лише фінальний хід, але й обсяг пошуку та аналізувати ефективність алгоритмів.

5.3 Обмеження розробленої системи

Розроблена система є дипломним прототипом, тому має певні обмеження. Вона не претендує на той самий рівень, що й провідні шахові платформи, які підтримують рейтинги, турніри, архіви ігор та розширену аналітику. Поточний сервер зберігає кімнати в оперативній пам'яті. Це зручніше для реалізації та розгортання системи, але не дозволяє відновити гру після перезапуску процесу сервера.

Штучний інтелект обмежений глибиною пошуку, яка підходить для виконання в браузері. Якщо ми заглибимося без оптимізацій, можемо виявити, що вибір ходів є повільним. Функція оцінки враховує матеріал, позиційні таблиці, мобільність і безпеку короля, але не складні стратегічні фактори: повну структуру пішаків, довгострокові плани, бази даних ендшпілю та книги дебютів.

Мережевий режим працює лише для двох гравців у кімнаті і не підтримує глядачів. Це відповідає встановленим вимогам, але ускладнює демонстрацію застосунку школам. Інтерфейс гнучкий, але не має повного набору додаткових налаштувань, які очікують користувачі двох основних платформ: вибір теми дошки, історія ходів у довшій формі або аналіз гри після її завершення.

Згадані обмеження не суперечать меті роботи, оскільки основна увага зосереджена на шаховій логіці, алгоритмах штучного інтелекту та синхронізації в мультиплеєрі. Водночас вони є напрямками для подальшого дослідження. Ще одне обмеження - відсутність повної шахової нотації гри у формі, придатній для експорту. Це не впливає на основні завдання, але обмежує використання системи для аналізу зіграних партій. Архітектура сервера не підтримує балансування навантаження між кількома процесами.

Для дипломного прототипу це прийнятно, але для великої кількості одночасних кімнат потрібно масштабувати окремо. Штучний інтелект не має книги дебютів, тому на першому етапі гри його ходи здійснюються лише за допомогою функції оцінки та пошуку. Це може призвести до менш природних рішень на початку гри порівняно з професійними движками. Таблиці ендшпілю, які використовуються в реальних шахових движках для точного оцінювання фінальних позицій з невеликою кількістю фігур, у нашому випадку не використовуються. У проекті ми оцінюємо такі позиції за допомогою загальної функції.

5.4 Напрями подальшого розвитку

Розробка системи може також включати збереження ігор у базі даних. Відновлення гри після перезапуску сервера, історія ходів та архів зіграних ігор можуть бути реалізовані.

Для штучного інтелекту потрібно додати ітеративне поглиблення, обмеження часу на ходи та таблиці транспозицій. Такі механізми дозволять краще використовувати час і не аналізувати повторно позиції, що виникають через різні шляхи. Оцінку можна розширити, аналізуючи структуру пішаків, відкриті лінії, активність короля в ендшпілі та стадії гри. Це покращить якість рішень без збільшення глибини пошуку.

Для багатокористувацького режиму спостерігачі, таймер гри та система завершення гри на основі часу були б корисними. Це наблизить нас до повноцінного онлайн-сервісу. Інтерфейс можна розширити журналом ходів у шаховій нотації та налаштуваннями теми дошки, а також режимом перегляду завершених ігор. Усі ці функції покращать користувацький досвід без зміни базової архітектури.

Другий напрямок — розширення тестового набору для двигуна. Він може включати стандартизовані позиції для перевірки кількості легальних ходів, спеціальних правил та складних шахових випадків. У сфері розгортання корисно забезпечити більш стабільне зберігання стану кімнати та моніторинг на стороні сервера. Це покращить надійність системи для довгострокового використання.

Врешті-решт, архітектура проекту повинна бути збережена: правила гри знаходяться в ChessEngine, сервер відповідає за авторитетний стан, а клієнт підтримує дружній користувацький досвід з грою. Ще один перспективний крок — аналіз після гри. Критичні помилки, найкращий вибір та/або зміна оцінки позиції від гри до гри будуть показані системою.

У навчальному середовищі слід ввести режим головоломок, де гравець повинен вирішувати тактичні позиції. Цей режим може бути реалізований у ChessEngine для перевірки гри та модулі штучного інтелекту для аналізу, які відповіді є правильними. У мережі можна реалізувати рейтингову або умовно рейтингову систему. Це вимагатиме ідентифікації користувачів, бази даних та зберігання результатів, що виходить за межі поточного прототипу диплома.

З технічного погляду можливим є винесення обчислень AI[7] у Web Worker. Це дозволило б виконувати глибший пошук без блокування основного потоку інтерфейсу й покращило б плавність взаємодії.

Для серверної частини перспективним є горизонтальне масштабування, однак воно потребує спільного сховища станів або механізму маршрутизації користувачів до процесу, у якому зберігається їхня кімната. Це завдання актуальне для переходу від прототипу до сервісу з багатьма одночасними партіями.

Для підвищення стабільності мережевого режиму можна додати збереження кімнат у зовнішньому сховищі. Це дозволило б відновлювати партії після перезапуску сервера або перенесення процесу на інший вузол.

Корисним розвитком була б система аналізу ходів у післяпартійному режимі. Вона могла б повторно проходити позиції, порівнювати ходи користувача з вибором MinimaxBot і показувати критичні моменти партії.

У майбутньому можна реалізувати декілька рівнів складності AI[7] не тільки через глибину, а й через різні оцінювальні профілі. Наприклад, один бот міг би грати агресивніше, інший — обережніше щодо безпеки короля.

Для інтерфейсу перспективним є додавання доступності: кращої підтримки клавіатурного керування, контрастних тем і текстових повідомлень про стан дошки. Це розширило б коло користувачів системи.

Також доцільно додати журнал подій мережевої партії. Він допоміг би аналізувати проблеми синхронізації, відключення та повторного підключення під час тестування або експлуатації.

У напрямі AI[7] перспективним є використання транспозиційних таблиць. У шахах одна й та сама позиція може виникати різними послідовностями ходів, тому збереження вже оцінених станів може зменшити повторні обчислення.

Ще одним корисним удосконаленням є ітеративне поглиблення. Алгоритм міг би спершу швидко знаходити хід на малій глибині, а потім уточнювати рішення, доки не вичерпано доступний час.

Для мережевого режиму важливим майбутнім кроком є додавання таймерів. Шахові партії часто проводяться з контролем часу, тому сервер повинен буде авторитетно відстежувати час кожної сторони.

З боку інтерфейсу можна реалізувати перегляд варіантів після партії. Користувач міг би крокувати історією ходів, повертатися до конкретної позиції та порівнювати власний хід із рекомендацією AI[7].

У перспективі система може бути доповнена навчальними сценаріями. Наприклад, бот може пояснювати, чому певний хід оцінено краще, або підсвічувати тактичні мотиви, знайдені під час пошуку.

5.5 Висновки до розділу 5

У шостому розділі оцінено відповідність реалізації поставленим задачам, визначено переваги й обмеження системи та окреслено напрями розвитку. Результати підтверджують, що створений застосунок виконує основні функції інтелектуальної багатокористувацької шахової вебсистеми.

ВИСНОВКИ

У дипломній роботі розроблено та досліджено інтелектуальну багатокористувацьку систему для гри в шахи на основі вебтехнологій. Реалізоване рішення поєднує клієнтську частину, серверну взаємодію, спільний шаховий рушій і модулі штучного інтелекту.

Під час виконання роботи проаналізовано особливості шахових вебсистем і визначено вимоги до рушія, мережевого режиму та інтерфейсу. Обґрунтовано вибір JavaScript, React[1], PixiJS[2], Node.js[3] і Socket.IO[4] як технологій, що відповідають подієвій та інтерактивній природі проєкту.

Розроблено ChessEngine, який підтримує генерацію легальних ходів, перевірку шаху, мату й пату, рокірування, взяття на проході, перетворення пішака, клонування позиції та формування snapshot стану. Рушій використовується у клієнті, сервері та AI[7], що забезпечує узгодженість правил.

Реалізовано AI-суперника у вигляді кількох стратегій: RandomBot, GreedyBot і MinimaxBot. Для пошукового бота застосовано Alpha-Beta Pruning[5], функцію оцінювання позиції та впорядкування ходів, що дозволило дослідити вплив оптимізації на кількість аналізованих позицій.

Створено багатокористувацький режим із кімнатами, серверною перевіркою ходів, синхронізацією стану та підтримкою повторного підключення. Авторитетна серверна архітектура забезпечує однаковий стан партії для обох клієнтів.

Дослідження показало, що збільшення глибини Minimax[6] покращує здатність бота враховувати наслідки ходів, але суттєво збільшує обчислювальні витрати. Alpha-Beta Pruning[5] разом з упорядкуванням ходів зменшує кількість переглянутих позицій і робить пошук практичнішим для вебзастосунку.

Тестування підтвердило працездатність основних модулів системи: шахової логіки, AI[7], мережевої синхронізації, серверної валідації та

адаптивного інтерфейсу. Виявлені обмеження пов’язані переважно з відсутністю довготривалого зберігання партій і професійних шахових оптимізацій.

Отже, поставлену мету досягнуто: створено працездатну інтелектуальну багатокористувацьку шахову систему та виконано дослідження алгоритмів Minimax[6] і Alpha-Beta Pruning[5] у контексті її практичної реалізації.

Під час роботи особливо важливим стало рішення про винесення шахової логіки у спільний модуль. Воно спростило архітектуру, забезпечило однакову поведінку різних режимів і дало можливість використовувати один рушій для перевірки ходів та AI-симуляцій.

Практична реалізація підтвердила, що вебтехнології є придатною основою для створення шахової системи з реальним часом. React[1] і PixiJS[2] забезпечили інтерфейс, Socket.IO[4] — подієвий обмін, а Node.js[3] — серверну координацію кімнат.

Науково-практичний результат роботи полягає в демонстрації того, як класичні алгоритми пошуку можуть бути інтегровані у сучасний вебзастосунок. Minimax[6] і Alpha-Beta Pruning[5] розглянуті не ізольовано, а через їхній вплив на конкретну поведінку AI-суперника.

Розроблена система має потенціал подальшого розвитку, зокрема через додавання збереження партій, глибших методів AI-оптимізації, розширених налаштувань інтерфейсу та навчальних режимів. При цьому базова архітектура дозволяє розширювати функціональність без зміни основних правил гри.

Виконана робота підтверджує, що навіть навчальний шаховий застосунок потребує комплексного підходу. Неможливо окремо реалізувати лише інтерфейс або лише алгоритм AI[7]: коректна система виникає з узгодженої взаємодії рушія, клієнта, сервера й методів оцінювання.

Запропонована архітектура показує перевагу спільної логіки у вебпроєктах, де одна й та сама предметна модель потрібна в різних середовищах. Це особливо помітно в шахах, де найменша розбіжність правил може зруйнувати партію.

Дослідження Minimax[6] і Alpha-Beta Pruning[5] дало змогу оцінити не лише теоретичну складність алгоритмів, а й їхню поведінку у створеному застосунку. Отримані метрики підтверджують, що оптимізація пошуку є необхідною умовою практичного використання AI[7] у браузерній грі.

Таким чином дипломний проєкт поєднує програмну реалізацію та дослідницький компонент. Розроблена система може слугувати базою для подальшого вдосконалення, а отримані висновки — для розуміння принципів побудови інтелектуальних багатокористувацьких вебзастосунків.

ПЕРЕЛІК ПОСИЛАНЬ

1. React Documentation. URL: https://react.dev/ (дата звернення: 07.06.2026).

2. PixiJS Documentation. URL: https://pixijs.com/ (дата звернення: 07.06.2026).

3. Node.js Documentation. URL: https://nodejs.org/docs/ (дата звернення: 07.06.2026).

4. Socket.IO Documentation. URL: https://socket.io/docs/ (дата звернення: 07.06.2026).

5. Wikipedia, Alpha Beta pruning. URL: https://en.wikipedia.org/wiki/Alphabeta_pruning (дата звернення: 07.06.2026).

6. Chess Programming Wiki. Minimax. URL: https://www.chessprogramming.org/Minimax (дата звернення: 07.06.2026).

7. Chess Programming Wiki. AI. URL: https://www.chessprogramming.org/Artificial_Intelligence (дата звернення: 07.06.2026).

8. GitHub Pages Documentation. URL: https://docs.github.com/pages (дата звернення: 07.06.2026).

9. Railway Documentation. URL: https://docs.railway.com/ (дата звернення: 07.06.2026).

10. Chess Programming Wiki. Alpha-Beta. URL: https://www.chessprogramming.org/Alpha-Beta (дата звернення: 07.06.2026).

11. Chess Programming Wiki. Evaluation. URL: https://www.chessprogramming.org/Evaluation (дата звернення: 07.06.2026).

ДОДАТОК А

Основні події багатокористувацького режиму

Подія	**Призначення**
create_room	Створення нової кімнати та призначення гравцю сторони білих.
join_room	Приєднання другого гравця до наявної кімнати.
make_move	Передавання наміру виконати хід для серверної перевірки.
move_applied	Розсилка підтвердженого ходу та оновленого стану гри.
rejoin_room	Відновлення участі у партії після тимчасового відключення.