Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра автоматизації, комп’ютерних наук і технологій

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеню вищої освіти – магістр
за освітньо-професійною програмою
«Комп’ютерні науки»

зі спеціальності
122 – Комп’ютерні науки

тема роботи:
«Розробка інтелектуальної системи прийняття рішень в умовах обмежених обчислювальних ресурсів»

Виконав студент гр. КН-24м. _____ Є. С. Костін

Керівник практики _____ Тронь В.В.

Нормоконтроль _____ Маринич І. А.

Завідувач кафедри _____ Рубан С. А.

Кривий Ріг – 2025

КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет: *інформаційних технологій*

Кафедра: *автоматизації, комп'ютерних наук і технологій*

Ступінь вищої освіти: *Магістр*

Спеціальність: *122 – Комп'ютерні науки*

ЗАТВЕРДЖУЮ

Зав. кафедрою: *к.т.н. Рубан С.А.*

_____

« ___»_____2025 р.

ЗАВДАННЯ

на кваліфікаційну роботу магістра

студентові групи КН-24м *Костіну Євгену Сергійовичу*

1. Тема кваліфікаційної роботи: *«Розробка інтелектуальної системи прийняття рішень в умовах обмежених обчислювальних ресурсів»*

затверджено наказом по університету № *257с* від *13.05.2025 р.*

2. Термін здачі кваліфікаційної роботи: *01.12.2025р.*

3. Склад кваліфікаційної роботи: *Пояснювальна записка обсягом 82с., додатки, презентація у Microsoft PowerPoint (15 слайдів) в електронному та друкованому вигляді*

4. Консультанти кваліфікаційної роботи:

Розділ 1-3 доц. Тронь В. В.

_____

Нормоконтроль доц. Маринич І. А.

5. Календарний план:

№	Етапи роботи	Термін виконання
1	*Вступ*	*14.05.25*
2	*Розділ 1*	*25.06.25*
3	*Розділ 2*	*18.07.25*
4	*Розділ 3*	*19.11.25*
5	*Висновки*	*20.11.25*
6	*Оформлення кваліфікаційної роботи*	*25.11.25*
7	*Підготовка презентації та графічного матеріалу*	*28.11.25*
8	*Підготовка доповіді до захисту*	*01.12.25*

6. Дата видачі завдання: *13.05.2025р.*

Керівник _____ /Тронь В. В./

7. Запевнення: *Я, Костін Євген Сергійович, запевняю, що ця кваліфікаційна робота виконана самостійно, не містить академічного плагіату, фабрикації, фальсифікації. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.*

Із чинним Положенням про академічну доброчесність Криворізького національного університету ознайомлений.

Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі умисних порушень робота не допускається до захисту або оцінюється незадовільно.

Студент _____ / Костін Є. С./

АНОТАЦІЯ

Костін Є. С. Розробка інтелектуальної системи прийняття рішень в умовах обмежених обчислювальних ресурсів : кваліфікаційна робота магістра : 122 – Комп’ютерні науки. Кривий Ріг. Криворізький національний університет, 2025. 83 с.

Робота складається зі вступу, трьох розділів, висновків, переліку використаної літератури з 30 позицій. Загальний обсяг роботи становить 82 сторінки, з яких основний зміст викладено на 75 сторінках та містить 56 рисунки.

Дослідження присвячене розробці інтелектуальної системи прийняття рішень, здатної ефективно працювати в умовах обмежених обчислювальних ресурсів. У роботі проведено комплексний аналіз сучасних алгоритмів пошуку. Особливу увагу приділено застосуванню гібридних нейромережевих моделей, що поєднують машинне навчання з деревопошуком для підвищення ефективності стратегічного аналізу.

У рамках практичної частини створено декілька варіантів шахового рушія. Експериментальні дослідження показали, що гібридний підхід забезпечує суттєве зростання ефективності порівняно з чистими нейромережевими моделями, хоча й поступається оптимізованому евристичному рушію. Отримані результати підтверджують, що саме поєднання легковагових нейромереж із класичними алгоритмами є найбільш перспективним напрямом створення продуктивних систем прийняття рішень для апаратно обмежених середовищ.

ІНТЕЛЕКТУАЛЬНА СИСТЕМА, СИСТЕМА ПРИЙНЯТТЯ РІШЕНЬ, НЕЙРОННІ МЕРЕЖІ, ШАХОВИЙ РУШІЙ, МАШИННЕ НАВЧАННЯ.

ABSTRACT

Kostin Y. S. Development of an Intelligent Decision-Making System under Limited Computational Resources: Master’s qualification thesis: 122 – Computer Science. Kryvyi Rih. Kryvyi Rih National University, 2025. 83 p.

The thesis consists of an introduction, three chapters, conclusions, and a list of references comprising 30 sources. The total volume of the work is 82 pages, of which 75 pages constitute the main content and include 56 figures.

The research is devoted to the development of an intelligent decision-making system capable of operating efficiently under conditions of limited computational resources. The thesis provides a comprehensive analysis of modern search algorithms. Particular attention is paid to the application of hybrid neural network models that combine machine learning with tree search techniques in order to improve the efficiency of strategic analysis.

Within the practical part of the study, several variants of a chess engine were developed. Experimental results demonstrated that the hybrid approach provides a significant increase in efficiency compared to purely neural network–based models, although it remains inferior to an optimized heuristic engine. The obtained results confirm that the combination of lightweight neural networks with classical search algorithms represents the most promising direction for the development of high-performance decision-making systems in hardware-constrained environments.

INTELLIGENT SYSTEM, DECISION-MAKING SYSTEM, NEURAL NETWORKS, CHESS ENGINE, MACHINE LEARNING.

ЗМІСТ

ВСТУП

Початок третього десятиліття, 21-го століття у науковому світі відзначився вибухом популярності штучного інтелекту (ШІ), та ростом і популяризацією ідеї сталого розвитку. Коли кожна ліпша компанія що працює у сфері послуг і не тільки, хоче мати персоналізований ШІ - дослідження в галузі тренування такої системи на обмежених ресурсах набувають стратегічного значення. Використання методів ефективного навчання дозволяє забезпечити високу продуктивність моделей без значного зростання обчислювальних витрат, що є ключовим фактором для застосування ШІ в реальному світі.

Одним із головних викликів є обмеження апаратних ресурсів, традиційні моделі глибокого навчання вимагають значних обчислювальних потужностей, що робить їх непридатними для роботи на пристроях із низьким енергоспоживанням. Для вирішення цієї проблеми активно розвиваються методи квантованого навчання, прунингу нейронних мереж, компресії моделей та застосування ефективних алгоритмів пошуку. Це дозволяє розширити сферу використання ШІ без необхідності у високопродуктивному обладнанні.

Крім технічних аспектів, питання тренування ШІ на обмежених ресурсах має важливий соціально-економічний вимір. Оптимізація процессу створення та використання цих моделей дозволить збільшити доступність ШІ для малих підприємств, освітніх закладів та країн із недостатнім технологічним потенціалом. Також це сприятиме зменшенню екологічного впливу ШІ, оскільки енергоефективні алгоритми дозволяють знизити вуглецевий слід дата-центрів.

Отже, дослідження у сфері тренування штучного інтелекту на обмежених ресурсах мають велике значення для подальшого розвитку технологій, їхньої доступності та екологічної стійкості. Подальше вдосконалення методів оптимізації навчання дозволить розширити можливості використання ШІ у різних сферах діяльності, забезпечуючи баланс між продуктивністю, витратами та стійкістю до ресурсних обмежень.

Основною метою даного дослідження є створення системи прийняття рішень у формі шахового рушія, що використовує для оптимізації пошуку нейронну мережу, це дозволить оцінити ефективність існуючих методів тренування вузькоспеціалізованого ШІ. Дослідження спрямоване на визначення оптимального підходу до розробки ШІ, який може ефективно працювати навіть при обмежених обчислювальних потужностях.

Робота передбачає вивчення сучасних шахових програм, аналіз їхньої архітектури та методів навчання. Особлива увага буде приділена алгоритмам пошуку найкращих ходів, ефективним методам оцінки позицій та можливостям компресії моделей без значної втрати їхньої продуктивності. Отримані результати дозволять зробити висновки про доцільність використання певних підходів до навчання ШІ для оптимізації дерев пошуку. Таким чином, дослідження сприятиме розвитку методології навчання вузькоспеціалізованого ШІ, розширенню розуміння його можливостей та обмежень.

РОЗДІЛ І. МЕТОДИ ПРИЙНЯТТЯ РІШЕНЬ ДЛЯ ПОШУКУ НАЙКРАЩОГО ХОДУ

Шахи - абстрактна стратегічна гра, яка не передбачає прихованої інформації та випадкових величин. Основна мета гри поставити шах і мат(загроза неминучого захоплення) королю противника. Також передбачається кілька сценаріїв при яких гра може закінчитися унічию.

1.1 МЕТОДИКА ОЦІНКИ ШАХОВОЇ ПОЗИЦІЇ

Оцінювання ходів у шахах є ключовим аспектом розробки алгоритмів для шахових програм і базується на методах пошуку та функціях оцінки позицій. Один із базових підходів до оцінки шахових ходів полягає у використанні повного перебору можливих варіантів із визначенням найкращого з них. Проте повний перебір усіх можливих ходів у шахах є обчислювально недосяжним через експоненційне зростання кількості позицій[1]. Замість цього використовуються обмежені за глибиною алгоритми пошуку, що дозволяють оцінювати позиції на кілька ходів уперед (наприклад, на 4-6 ходів).

Оскільки в більшості ситуацій неможливо передбачити результат гри на основі кількох ходів, для оцінки проміжних позицій використовується функція оцінки (evaluation function). Вона дозволяє визначити ймовірність виграшу однієї зі сторін на основі матеріальної та позиційної переваги. Найпростішим підходом є матеріальна оцінка, що базується на підрахунку кількості та значущості фігур у позиції. Кожній фігурі присвоюється певна вага у вигляді еквіваленту пішаків (наприклад, ферзь оцінюється у 9 пішаків, тура – у 5, слон та кінь – у 3). Якщо одна зі сторін має значну перевагу в матеріалі, її позиція вважається виграшною.

Рисунок 1.1 – Відносна вартість фігур.

Більш розвинені алгоритми доповнюють матеріальну оцінку позиційним аналізом, що враховує розташування фігур на дошці. Наприклад, розміщення пішаків і коней у центрі є вигіднішим, ніж на краях дошки, а відкриті лінії для тур сприяють активній грі[2]. Алгоритми також враховують структуру пішаків, зв’язок між турами, безпеку короля та інші стратегічні аспекти. Сучасні шахові програми поєднують матеріальні та позиційні критерії оцінки, а також використовують машинне навчання для оптимізації функцій оцінки. Наприклад, шаховий двигун Stockfish NNUE застосовує нейронну мережу для більш точної оцінки позицій.

1.1.1 МІНІМАКС ПОШУК

Одним із фундаментальних алгоритмів у розробці шахових програм є Мінімакс пошук, що використовується для прийняття рішень шляхом аналізу можливих ходів. Його основна ідея ґрунтується на принципі максимізації власних шансів на виграш та мінімізації шансів супротивника, що є ключовою особливістю гри з нульовою сумою в шахах. Алгоритм оцінює кожен можливий стан шахівниці з поточного ходу. Для кожної позиції використовується функція оцінки, яка визначає позиційну перевагу. Ходи оцінюються на певну глибину, після чого

відбувається рекурсивне повернення значень вгору по дереву. Гравець, що робить хід, вибирає найкращий варіант (максимізація), тоді як супротивник намагається зменшити його оцінку (мінімізація). На рисунку 1.2 сірі квадрати відповідають рішенням максимізуючого гравця, а білі мінімізуючого. Таким чином, алгоритм аналізує можливі сценарії розвитку гри, вибираючи найкращий хід з урахуванням опору супротивника.

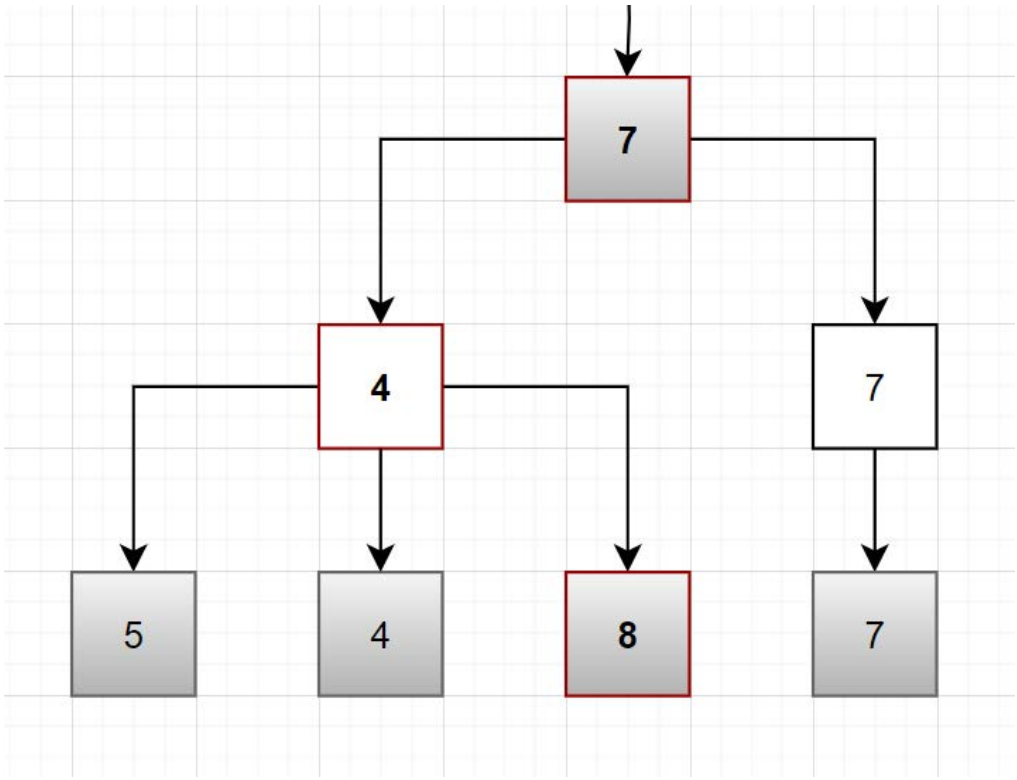

Рисунок 1.2 – Приклад дерева пошуку Мінімакс алгоритму

Незважаючи на те, що мінімакс дозволяє знайти ідеальні ходи при достатньому часі аналізу, його основний недолік – експоненційне зростання обчислювальної складності через велику кількість можливих ходів на кожному рівні. В шахах кількість варіантів розгалуження в середньому становить 10–20 ходів, що робить повний перебір неефективним. Щоб покращити ефективність, були розроблені різні методи оптимізації. «Shannon Type B мінімакс» обмежує розгляд ходів лише до невеликої кількості перспективних варіантів, що зменшує кількість вузлів у дереві пошуку. Однак такий підхід може призводити до помилкових оцінок через пропуск важливих тактичних можливостей[3].

Оптимальнішим рішенням стало використання альфа-бета відсікання, що дозволяє зменшити кількість розглянутих варіантів без втрати якості вибору ходів. Цей метод ефективно скорочує розмір дерева пошуку, відсікаючи неперспективні гілки, що робить мінімаксний алгоритм придатним для застосування у сучасних шахових програмах.

1.1.2 АЛЬФА-БЕТА ВІДСІКАННЯ

Альфа-бета відсікання є оптимізаційним методом для мінімаксного алгоритму, що дозволяє значно зменшити кількість розглянутих вузлів у дереві пошуку без втрати точності вибору найкращого ходу. Його основна ідея полягає в ігноруванні нерелевантних варіантів, які завідомо не можуть призвести до оптимального результату. Метод працює шляхом введення двох параметрів для кожного вузла пошуку: Альфа (α) – найкращий знайдений результат для гравця, що намагається максимізувати оцінку. Бета (β) – найкращий знайдений результат для гравця, що намагається мінімізувати оцінку[4, 5].

Алгоритм використовує ці значення для скорочення перебору варіантів. Якщо в процесі пошуку виявляється, що певний вузол не може покращити результат для гравця, він відсікається без подальшого аналізу, оскільки його розгляд не вплине на підсумкове рішення. У класичному алгоритмі мінімакс - кожен можливий хід розглядається з однаковою важливістю, що суттєво збільшує обчислювальні витрати. Альфа-бета відсікання усуває цю проблему, дозволяючи ігнорувати завідомо невигідні позиції. Це особливо ефективно при глибокому аналізі шахових ходів, оскільки в багатьох випадках не потрібно перевіряти всі можливі гілки дерева.

Альфа-бета відсікання дозволяє значно скоротити час обчислення. У найкращому випадку воно зменшує кількість оброблюваних вузлів до квадратного кореня від їхньої початкової кількості. Наприклад, якщо класичний мінімакс вимагає 100 секунд для вибору оптимального ходу, то альфа-бета відсікання може виконати цю ж задачу приблизно за 10 секунд. Однак ефективність цього методу

значною мірою залежить від порядку розгляду ходів. Якщо спочатку аналізуються найкращі ходи, то відсікання працює найбільш ефективно. В іншому випадку, якщо спершу розглядаються слабкі ходи, алгоритм не зможе швидко відсікти невигідні варіанти, що знизить його продуктивність.

Альфа-бета відсікання є стандартним методом у сучасних шахових двигунах. Воно дозволяє досягти рівня гросмейстерської гри на обмежених ресурсах, зменшуючи необхідність у повному переборі всіх варіантів. У поєднанні з іншими оптимізаціями, такими як сортування ходів або використання евристичних методів, альфа-бета відсікання забезпечує ефективний пошук найкращих рішень у складних шахових позиціях.

1.1.3 ЕВРИСТИЧНІ ПРИЙОМИ

Успіх пошуку з альфа-бета-відсіченням багато в чому залежить від порядку, в якому досліджуються ходи. Евристичні методи допомагають досліджувати найбільш перспективні варіанти першими, тим самим збільшуючи ефективність пошуку.

1. Оцінка позиції: Першим кроком до впорядкування ходів є використання функції оцінки для їх попереднього ранжирування. Функція оцінки допомагає визначити, наскільки сильною є дана позиція для кожної сторони. Зазвичай, позиція оцінюється з точки зору матеріалу, рухливості, захисту короля та інших факторів.[6]

2. Історія ходів: Ідея полягає в тому, щоб запам'ятовувати, які ходи виявилися успішними в минулому, і надавати їм пріоритет при дослідженні. Для цього використовується таблиця, в якій для кожного можливого ходу зберігається кількість разів, коли хід призвів до кращого результату, ніж очікувалося.

3. Евристика вбивці: Цей метод передбачає запам'ятовування ходів, які викликали відсічення альфа-бета в інших варіантах. Ці ходи-«вбивці» вважаються перспективними і досліджуються в першу чергу.

4. Хід з попереднього пошуку: Оскільки пошук з альфа-бета-відсіченням зазвичай використовується в ітеративному поглибленні, можна використовувати хід з попереднього пошуку на меншій глибині в якості першого досліджуваного ходу. Цей хід часто є найкращим, оскільки він вже був ретельно досліджений на попередньому кроці.

Ще одним потужним методом оптимізації шахового пошуку є використання транспозиційних таблиць. У шахах одна і та ж позиція може виникнути різними шляхами. Транспозиційна таблиця дозволяє зберігати вже обчислені оцінки позицій, щоб уникнути їх повторного аналізу. Оскільки зберігання кожної позиції у пам’яті є неефективним, застосовується хешування методом Зобріста [7]. Кожному розташуванню фігури на дошці відповідає випадковий 32-бітний ключ. Повний хеш позиції отримується за допомогою бітової операції XOR над ключами всіх фігур. Завдяки цьому:

1. Оновлення хешу після ходу відбувається за два XOR-оператори, що значно швидше, ніж обчислення всього хешу заново.

2. Швидке порівняння позицій дає змогу миттєво відкидати повтори.

Метод упорядкування ходів значно підвищує продуктивність шахових алгоритмів, але має певні обмеження: Ефективність залежить від якості евристики – якщо погані ходи будуть проаналізовані першими, швидкість алгоритму знизиться. Колізії хешів – через обмежений розмір транспозиційних таблиць іноді різні позиції можуть мати однаковий хеш, що потребує використання додаткової пам’яті.

1.1.4 NULL MOVE PRUNING

Null Move Pruning (NMP) є однією з технік відсікання гілок у процесі пошуку шахового рушія. Основна ідея цього методу полягає в оцінці позиції шляхом симуляції ситуації, коли поточний гравець пропускає хід. Якщо, навіть без здійснення ходу, поточна оцінка позиції все ще перевищує β (параметр, що встановлює межу оцінки в алгоритмі альфа-бета відсікання), тоді подальший

пошук цієї гілки може бути припинений, оскільки справжній найкращий хід імовірно також приведе до β-відсікання.

Метод базується на так званому *спостереженні за нульовим ходом* (Null Move Observation), яке передбачає, що зазвичай існує принаймні один хід, який є кращим за відсутність ходу. Таким чином, якщо навіть пропуск ходу призводить до оцінки, що забезпечує β-відсікання, то справжній найкращий хід, ймовірно, зробить це ще швидше. Головна перевага NMP полягає у зменшенні глибини пошуку після симуляції нульового ходу. Це дозволяє скоротити час обчислень, оскільки пошук виконується на меншій глибині. Зазвичай зменшення глибини становить два або три рівні.

Існує кілька важливих аспектів, які необхідно враховувати при реалізації NMP:

1. Рекурсивне NMP – деякі шахові рушії дозволяють послідовні нульові ходи (recursive null move pruning). Однак немає єдиного стандарту щодо того, чи слід це використовувати.

2. Перевірка на шах – якщо нульовий хід виконується в позиції, де гравець уже перебуває під шахом, то така ситуація є нелегальною.

3. Проблема цугцвангу – NMP може давати помилкові результати у позиціях цугцвангу, коли будь-який хід погіршує становище гравця. Оскільки пропуск ходу у таких позиціях був би вигіднішим, ніж будь-який допустимий хід, він порушує основну логіку NMP.

4. Застосування у quiescence search – у розширеному пошуку (quiescence search) не всі ходи розглядаються, тому зменшення глибини може не дати достатньої інформації про силу позиції.

У тестах, проведених для рушія Tesseract, було встановлено, що: використання рекурсивного NMP не дало покращення швидкості, а рейтинг ELO навіть знизився; відсікання у позиціях цугцвангу виявилося складним і мало незначний ефект; у quiescence search застосування NMP не дало переваг, оскільки оцінювальна функція рушія була недостатньо сильною. Null Move Pruning дозволяє скоротити кількість розглянутих позицій, забезпечуючи швидшу обробку

варіантів[8, 9]. Однак його ефективність залежить від точності евристик та коректного налаштування параметрів, особливо в ситуаціях цугцвангу чи під час застосування в quiescence search.

1.1.5 LATE MOVE REDUCTIONS

Late Move Reductions (LMR) є технікою обрізання дерева пошуку, що спрямована на зменшення глибини аналізу менш значущих ходів. Вона базується на припущенні, що початкові ходи в упорядкованому списку є більш важливими та перспективними, тоді як пізні ходи менш ймовірно впливають на результат партії. LMR використовує принцип сортування ходів, при якому перші кілька (зазвичай три або чотири) найкращі ходи досліджуються повною мірою, а всі наступні перевіряються на можливість скорочення глибини пошуку. Зменшення глибини для пізніх ходів залежить від певних умов, що дозволяють уникнути зниження точності пошуку.

Основні кроки роботи LMR:

1. Упорядкування ходів за ймовірною значущістю.

2. Повний аналіз перших кількох ходів.

3. Для решти ходів перевіряються обмеження, які визначають можливість скорочення глибини пошуку.

4. Якщо скорочений пошук все ще дає високу оцінку, хід може бути перевірений повторно з повною глибиною, щоб уникнути пропуску важливих варіантів.

Для запобігання помилковому скороченню важливих ходів використовуються наступні обмеження: Взяття фігури (captures) не підлягають скороченню. Перетворення пішака (promotions) завжди аналізуються повністю. "Killer moves" (ходи, що призвели до обрізання β-значення у попередніх пошуках) не скорочуються. Ходи, що ведуть до шаху, завжди аналізуються з повною глибиною. Позиції, в яких гравець уже перебуває під шахом, не піддаються скороченню. Глибина пошуку менше 3 півходів не допускає скорочень.

Основна складність LMR полягає у правильному балансуванні між швидкістю та точністю пошуку. Занадто агресивне скорочення може призводити до серйозних помилок, включаючи пропуск критичних ходів. Якщо пізній хід, який було скорочено, фактично є найкращим, його доведеться аналізувати заново, що може знизити ефективність методу. У процесі тестування LMR в рушії Tesseract було встановлено, що: хоча LMR дало значний приріст швидкості пошуку, це відбулося за рахунок зниження рейтингу ELO. Часто хороші ходи скорочувалися замість поганих, що призводило до помилок[10]. Менш агресивні налаштування LMR також не дали очікуваного покращення, що може вказувати на необхідність вдосконалення алгоритмів сортування ходів та оцінки позицій. Успішне впровадження LMR вимагає тонкого налаштування, оскільки надмірне скорочення може призводити до помилок у грі.

1.1.6 ЕФЕКТ ГОРИЗОНТУ ТА КВАЗІСТАЦІОНАРНИЙ ПОШУК

Всі наведені вище алгоритми – є алгоритмами із фіксованою глибиною пошуку. Для таких алгоритмів властива проблема горизонт-ефекту. Вона полягає в тому, що програма може не враховувати послідовності ходів, які відбуваються за межами встановленої глибини, що призводить до некоректної оцінки позиції. Це трапляється, коли алгоритм не бачить майбутніх вигідних ходів, оскільки вони знаходяться за межами його поточного аналізу, або коли програма неправильно оцінює розмін фігур, оскільки у встановленій глибині вона бачить лише втрату матеріалу без подальшого відновлення позиції. Наприклад, алгоритм може відмовитися від розміну фігурами через те, що перевага буде отримана на ходу що виходить за обмеження глибини пошуку.

Для зменшення впливу горизонт-ефекту застосовується методика квазістаціонарного (quiescence) пошуку. Її суть полягає у виконанні додаткового локального аналізу в кінцевих вузлах основного алгоритму, що дозволяє стабілізувати оцінку позиції перед її завершальним підрахунком. Такий підхід працює завдяки обмеженню лише нестабільними позиціями, де можливі

захоплення фігур або критичні тактичні загрози. Оскільки розглядаються лише шахи, взяття та інші важливі ходи, фактор розгалуження суттєво зменшується, що, у свою чергу, дозволяє виконувати аналіз на більшій глибині, ніж стандартні алгоритми мінімаксу або альфа-бета відсікання. Використання квазістаціонарного пошуку дає змогу підвищити точність оцінки позиції, уникнути некоректних рішень через обмежену глибину аналізу та значно знизити вплив горизонт-ефекту, що дозволяє алгоритму приймати стратегічно вигідні рішення. Водночас цей метод має і певні недоліки, зокрема збільшення обчислювальної складності, оскільки додатковий аналіз може уповільнювати роботу алгоритму. Крім того, правильний вибір критеріїв для визначення нестабільності позицій безпосередньо впливає на ефективність пошуку.

1.1.7 ДЕРЕВО ПОШУКУ МОНТЕ КАРЛО

Monte Carlo Tree Search (MCTS) є одним із найефективніших алгоритмів для оцінки ходів у шахах та інших стратегічних іграх. Його унікальність полягає у використанні випадкових імітацій гри для визначення найкращих можливих ходів, що дозволяє алгоритму працювати без попередньо заданої функції оцінки. Це робить MCTS особливо корисним у складних іграх із великим фактором розгалуження, де класичні методи, такі як альфа-бета відсікання, можуть бути неефективними.

MCTS складається з чотирьох основних етапів:

1. Вибір (Selection) – алгоритм обирає найбільш перспективний вузол у дереві пошуку відповідно до певної стратегії вибору. Найпоширенішим підходом є використання UCT (Upper Confidence Bound for Trees), що балансує дослідження нових ходів і експлуатацію вже відомих.

2. Розширення (Expansion) – коли досягнуто листовий вузол, створюється один або більше нових вузлів для розширення дерева пошуку.

3. Симуляція (Simulation) – з нового вузла проводяться випадкові ігри (плейаути), які тривають до завершення партії. Це дозволяє алгоритму отримати статистичні оцінки можливих результатів гри.

4. Зворотне поширення (Backpropagation) – результати симуляцій передаються вгору по дереву, оновлюючи значення вузлів і визначаючи ймовірність виграшу для кожного ходу.

Однією з ключових переваг MCTS є його незалежність від функції оцінки, що дозволяє використовувати алгоритм у будь-якій грі з повною інформацією. Проте існують певні недоліки, зокрема велика кількість випадкових ігор, які можуть пропустити критичні ходи. Для оптимізації MCTS у шахах використовуються такі підходи: Гібридні методи – поєднання MCTS з глибокими нейронними мережами, наприклад, у AlphaZero, де симуляції проводяться не випадково, а з використанням навченої нейромережі для оцінки позицій; Вибір ходів за історичними даними – врахування ходів, які часто приводили до успішних результатів у попередніх симуляціях[11]. Обмеження глибини симуляцій - використання правил скорочення гри для уникнення надмірного пошуку у вже відомих позиціях.

MCTS є обчислювально інтенсивним алгоритмом, особливо в шахах, де можливі мільйони варіантів ходів. Для ефективної роботи алгоритму потрібні потужні процесори (CPU) та графічні процесори (GPU), якщо використовується гібридний підхід з нейромережами. Наприклад, у AlphaZero MCTS був інтегрований із глибоким навчанням, що вимагало використання тисяч TPU (Tensor Processing Unit) для самонавчання моделі. Наразі MCTS використовується в провідних шахових двигунах, таких як Leela Chess Zero, демонструючи високу ефективність у пошуку найкращих ходів. Алгоритм дозволяє знаходити стратегічно вигідні рішення, які можуть не бути очевидними при традиційному аналізі. Попри великі обчислювальні витрати, MCTS продовжує залишатися одним із найбільш перспективних методів для оцінки шахових позицій.

1.1.7 СПОСОБИ ПРЕДСТАВЛЕННЯ ШАХІВ

Для реалізації шахів використовуються різні підходи, зокрема, бітові представлення (bitboards), псевдозаконні ходи (pseudo-legal moves) та спеціальні алгоритми для обробки унікальних правил шахів, таких як рокірування або взяття на проході. Для представлення дошки використовують два основних методи:

1. Квадратно-центричне представлення (Square-centric representation) – дошка представлена у вигляді масиву із 64 осередків, де кожен осередок містить інформацію про тип фігури та її колір. Реалізація цього підходу полягає у використанні одновимірного або двовимірного масиву, де кожен індекс відповідає певній клітинці на дошці. Наприклад, у 64-елементному масиві кожен індекс (від 0 до 63) може містити значення, що відповідає певній фігурі (наприклад, 0 – порожня клітинка, 1 – білий пішак, 2 – чорний пішак тощо). Такий метод є простим у реалізації та дозволяє легко отримувати інформацію про фігури, але для перевірки ходів може знадобитися додатковий обхід масиву

2. Фігурно-центричне представлення (Piece-centric representation) – кожна фігура зберігається у списку разом із координатами розташування. Це означає, що замість того, щоб зберігати всю шахівницю у масиві, програма веде окремі списки для кожного типу фігур. Наприклад, для білих пішаків може бути створений масив із координатами всіх активних пішаків. Перевагою цього підходу є швидкий доступ до кожної фігури та її позиції, що особливо корисно при генерації можливих ходів. Однак цей метод потребує додаткової обробки для визначення загальної ситуації на шахівниці, наприклад, при перевірці атак чи загроз [12].

Щоб підвищити ефективність обробки, сучасні рушії використовують гібридний підхід, що поєднує обидва методи. Один з найефективніших способів – битові дошки (bitboards), що дозволяють зберігати інформацію про розташування фігур за допомогою 64-бітних чисел. Це забезпечує швидку обробку ходів за допомогою побітових операцій. Така реалізація передбачає використання двох рівнів представлення. Під час аналізу позиції рушій може звертатися як до списку фігур для швидкого доступу до їхніх місць розташування, так і до масиву клітин

для перевірки загального стану дошки. Наприклад, для визначення атакованих клітин рушій може використовувати бітборди, а для оцінки матеріальної переваги – списки фігур. Це дозволяє об'єднати переваги обох підходів, зменшуючи необхідність зайвих обчислень. Крім того, такі структури даних ефективно використовуються у поєднанні з транспозиційними таблицями для оптимізації повторних обчислень, що значно прискорює аналіз партії.

Генерація ходів у шахових рушіях відбувається наступним чином. Спочатку генеруються таблиці пошуку (lookup tables): використовуються для швидкого отримання можливих варіантів руху для кожної фігури. Цей підхід передбачає попереднє обчислення всіх можливих ходів для кожної клітинки дошки та збереження цих даних у масиві, що дозволяє миттєво отримати відповідь під час гри. При цьому фігури поділяються на два типи: Фігури з обмеженими траєкторіями (пішаки, королі, коні) – їхні ходи визначаються статичними таблицями переміщень, оскільки вони не залежать від інших фігур на дошці; Фігури з ковзною атакою (тури, слони, ферзі) – вони використовують магічні бітові дошки (magic bitboards) або PEXT-таблиці (Parallel Bits Extract), що забезпечують швидке обчислення доступних ходів. На Рисунок 1.3 показаний механізм роботи магічної дошки, ліва картинка – це маска фігур що блокують ходи, вона помножується на «магічне число», якому відповідає середній малюнок. Результат такої дії зображено на правому малюнку – він відповідає можливим ходам фігури.

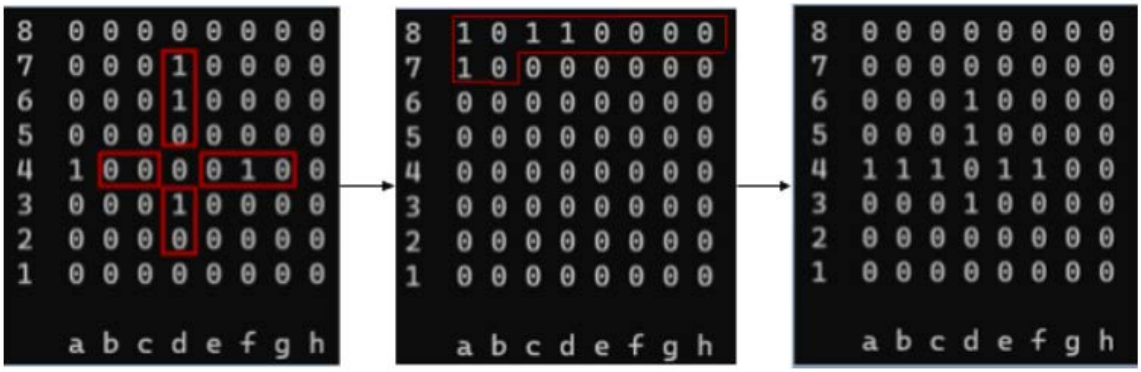

Рисунок 1.3. Методика визначення ходів для фігур з ковзною атакою

Магічні бітові дошки використовують попередньо розраховані множники та бітові маски для миттєвого визначення доступних ходів у будь-якій позиції. Фільтрація псевдозаконних ходів: Після первинної генерації необхідно перевірити, чи призводить хід до шаху власному королю. Це особливо важливо при розрахунку ходів у глибоких варіантах аналізу. Також треба враховувати спеціальні випадки генерації ходів: Рокірування – перевіряється наявність загроз уздовж траєкторії короля та відсутність перешкод між королем і турою(в обидва боки); Взяття на проході (en passant) – вимагає додаткової перевірки можливості взяття пішака та коректної зміни стану дошки; Перетворення пішаків – при досягненні останньої горизонталі додаються альтернативні варіанти трансформації пішаків у ферзя, туру, слона або коня.

1.1.8 ВІКНА ОЧІКУВАНОЇ ОЦІНКИ

Aspiration Windows (вікна очікуваної оцінки) є оптимізаційним методом, що використовується в алгоритмі iterative deepening (ітеративного заглиблення) у шахових рушіях. Основна ідея полягає у встановленні вузького діапазону значень (вікна) для альфа-бета відсікання на основі оцінки попереднього проходу пошуку. Це дозволяє прискорити процес знаходження найкращого ходу, оскільки обмежений діапазон значень зменшує кількість розглянутих вузлів дерева пошуку. У традиційному шаховому пошуку значення α та β ініціалізуються відповідно як -∞ і +∞. Це означає, що будь-який варіант може бути прийнятним. Проте після виконання першої ітерації ітеративного заглиблення отримана оцінка позиції може використовуватися як центральна точка для встановлення вузького діапазону (aspiration window), що дозволяє наступним ітераціям більш ефективно відсіювати гілки дерева пошуку.

Процес роботи Aspiration Windows включає наступні кроки:

1. Використання оцінки з попередньої ітерації як центрального значення (evaluation score).

2. Встановлення початкового вікна, наприклад [evaluation - margin, evaluation + margin], де margin — це деяке фіксоване значення (наприклад, 0.5 пішака).

3. Виконання нового пошуку в межах цього вікна.

4. Якщо знайдена оцінка виходить за межі встановленого діапазону, то виконується розширення вікна (window widening).

Aspiration Windows використовує дві основні стратегії у випадку виходу знайденого значення за межі встановленого діапазону: Повне скидання вікна (Complete Window Reset) – алгоритм повертається до стандартного альфа-бета пошуку з необмеженим діапазоном оцінок; Поступове розширення (Gradual Widening) – розширює поточний діапазон, додаючи до меж вікна фіксовану величину (наприклад, ще 0.5 пішака). Такий підхід дозволяє зменшити кількість перевірених вузлів – у вузькому вікні більше відсічень (beta-cutoffs), що зменшує обсяг обчислень, покращення швидкодії у випадках, коли оцінка стабільна між ітераціями. Проте у такого алгоритму є проблеми з нестабільною оцінкою позицій – якщо оцінка змінюється між ітераціями, алгоритм часто виконує повне скидання вікна, що може зменшити ефективність. Він залежний від якості функції оцінки – якщо функція оцінки нестабільна, Aspiration Windows може не дати приросту швидкості або навіть погіршити продуктивність. Aspiration Windows широко використовується у провідних шахових рушіях, таких як Stockfishi Komodo. Проте його ефективність залежить від стабільності оцінки позицій, що вказує на важливість оптимізації функції оцінки та правильного вибору початкових параметрів вікна. У деяких рушіях, таких як Tesseract, впровадження Aspiration Windows не дало суттєвого приросту продуктивності, що може вказувати на слабку взаємодію з іншими оптимізаційними методами[13].

1.2 ОГЛЯД ІСНУЮЧИХ СИСТЕМ ПРИЙНЯТТЯ РІШЕНЬ.

1.2.1 ALPHA-ZERO

AlphaZero є одним із найбільш значущих досягнень у сфері штучного інтелекту для шахів, розробленим компанією DeepMind. Його унікальність полягає у використанні підходу навчання "з нуля" (tabula rasa) без попередньої експертної оцінки позицій та наборів шахових дебютів. Дана методика дозволяє AlphaZero самостійно знаходити оптимальні стратегії гри виключно через самонавчання та взаємодію з середовищем за допомогою підкріпленого навчання [14].

Основним алгоритмічним підходом у роботі AlphaZero є комбінація глибоких нейронних мереж та підкріпленого навчання. На відміну від класичних шахових рушіїв, таких як Stockfish, які використовують метод альфа-бета відсікання та евристики, AlphaZero застосовує алгоритм Монте-Карло Tree Search (MCTS) для пошуку ходів. Нейронна мережа AlphaZero отримує на вхід позицію на дошці та видає: вектор ймовірностей можливих ходів, оцінку позиції (очікуваний результат партії). Навчання здійснюється повністю за допомогою самогри (self-play). На кожному етапі алгоритм грає сам із собою, використовуючи поточні параметри нейромережі для вибору ходів. Після завершення партії отриманий результат використовується для оновлення параметрів моделі за допомогою градієнтного спуску.

Результати тестування AlphaZero проти інших провідних шахових програм показали його домінування:

1. У матчі зі Stockfish (чемпіоном TCEC 2016) AlphaZero виграло 28 партій, 72 завершилися внічию, жодної поразки.

2. AlphaZero використовує меншу кількість обчислень, але досягає високого рівня гри завдяки оптимізованому алгоритму пошуку.

3. Методика навчання дозволила алгоритму відкривати нові стратегічні концепції, що раніше не використовувалися в шахах.

Навчання AlphaZero вимагало значних обчислювальних ресурсів. Для генерації ігор було використано 5 000 TPU першого покоління. Для навчання нейронної мережі - 64 TPU другого покоління. Було зіграно 44 мільйони партій, на що було витрачено 9 годин. AlphaZero аналізувало близько 80 тисяч позицій за секунду, що значно менше, ніж у Stockfish (70 мільйонів), однак ефективність навченої моделі дозволяла компенсувати цю різницю за рахунок більш точного вибору ходів.

1.2.2 LC0

Leela Chess Zero (Lc0) — це відкрите програмне забезпечення для гри у шахи, що базується на технологіях глибокого навчання. Його розвиток розпочався у 2017 році після публікації статті DeepMind про AlphaZero, що демонстрував надзвичайну ефективність самонавчання нейромереж у грі в шахи. Відмінністю Lc0 є її відкритий код та розподілена обчислювальна платформа, що дозволяє ентузіастам з усього світу брати участь у тренуванні моделі. Lc0 використовує глибоку нейромережу, схожу на AlphaZero, але з певними відмінностями. В основі її архітектури лежать блоки Squeeze-and-Excitation, які дозволяють покращити обробку позицій шляхом зміни ваг нейронів залежно від важливості особливостей позиції. Вхідний шар нейромережі кодує позицію у 112 площинах розміром 8×8, що включають розташування фігур, можливість рокіровки, історію ходів тощо [15].

Останні версії Lc0 мають три вихідні голови (heads):

1. Політична голова (policy head) — передбачає ймовірності всіх можливих ходів.

2. Оцінювальна голова (value head) — обчислює ймовірність перемоги, нічиєї або поразки.

3. Голова прогнозу тривалості партії (moves left head) — прогнозує, скільки ходів залишилось до завершення гри.

Lc0 використовує навчання з підкріпленням із самогрою (self-play reinforcement learning). Процес навчання розподілений серед численних добровольців, які запускають клієнти Lc0 на своїх комп’ютерах для генерації ігрових даних. Ці дані передаються на центральний сервер, де модель оновлюється та розсилається для подальшого навчання. Для оцінки позицій використовується алгоритм Монте-Карло з обрізанням альфа-бета (Monte Carlo Tree Search, MCTS). Цей підхід дозволяє ефективніше досліджувати глибину позицій та приймати стратегічні рішення. На відміну від класичних шахових двигунів, які використовують алгоритми альфа-бета, Lc0 робить ставку на оцінку позиції, а не на глибокий пошук усіх можливих варіантів.

Однією з основних відмінностей між AlphaZero та Lc0 є доступність обчислювальних ресурсів. DeepMind використовував спеціалізовані тензорні процесори (TPU) для тренування AlphaZero, тоді як Lc0 спирається на розподілені обчислення через добровольців. Це дозволяє спільноті розвивати модель, проте накладає певні обмеження на її продуктивність та швидкість навчання. Слава Україні! Lc0 демонструє високий рівень гри та конкурує з найсильнішими шаховими рушіями, такими як Stockfish. Головна її перевага — здатність до позиційного розуміння гри та виявлення стратегічних ідей, що часто недоступні класичним шаховим рушіям. Її використовують шахові гросмейстери для аналізу партій та пошуку нових дебютних ідей. Leela Chess Zero є яскравим прикладом успішного застосування глибокого навчання для гри у шахи. Вона об'єднала шахову та комп'ютерну спільноту у спільному проекті, що дозволяє експериментувати з методами навчання та відкриває нові можливості у дослідженні штучного інтелекту.

1.2.3 NNUE

Efficiently Updatable Neural Networks (NNUE) є однією з найважливіших інновацій у шахових алгоритмах, що поєднує класичний альфа-бета пошук із глибоким навчанням. NNUE вперше було розроблено у 2018 році для японської гри сьогі, а пізніше адаптовано для шахових рушіїв, включаючи Stockfish. Основна перевага цього підходу полягає у можливості використовувати нейромережу для оцінки позицій без значного уповільнення роботи двигуна, що раніше було основною проблемою шахових програм на базі нейромереж. NNUE побудовано на основі компактної нейронної мережі, яка здійснює оцінку шахових позицій. Основний внесок цієї технології полягає у використанні бінарного кодування HalfKP (Half-King-Piece), що ефективно відображає взаємозв’язок між королем та іншими фігурами на дошці. Вхідні дані представлені у вигляді бінарних векторів, що суттєво зменшує обчислювальні витрати.

Основним алгоритм пошуку, що використовується в NNUE, залишається альфа-бета відсікання, що дозволяє ефективно досліджувати глибину позицій. Оцінка позицій відбувається шляхом інкрементного оновлення ваг нейронної мережі, що значно пришвидшує обчислення у порівнянні з повним перерахунком значень. Навчання NNUE ґрунтується на використанні позиційних оцінок, отриманих від самонавчання або шляхом аналізу партій інших рушіїв. Спочатку використовується набір шахових позицій з відповідними оцінками, які подаються до нейромережі. Потім застосовується метод градієнтного спуску, що дозволяє нейромережі покращувати точність оцінки. У деяких випадках використовується підкріплене навчання, коли рушій повторно аналізує позиції та коригує свої оцінки на основі нових результатів.[16, 17]

Цікавим фактом є те, що NNUE у Stockfish використовує навчальні дані від Leela Chess Zero (Lc0), що демонструє ефективність співпраці між різними підходами до шахового ШІ. NNUE було розроблено з урахуванням можливості роботи на звичайних центральних процесорах (CPU) без потреби у графічних процесорах (GPU) або тензорних процесорах (TPU). Це робить його доступним для

широкого кола користувачів, оскільки більшість сучасних комп’ютерів можуть запускати двигуни з NNUE без значних витрат на обчислювальні ресурси [18].

Інтеграція NNUE до Stockfish 12 призвела до збільшення продуктивності рушія приблизно на 80 Elo балів, що дозволило йому значно перевершити Leela Chess Zero за силою гри. Подальша оптимізація алгоритму у Stockfish 14 додала ще 20 Elo, завдяки комбінованому підходу: NNUE використовується для оцінки позицій у статичних ситуаціях, тоді як у тактичних позиціях застосовується класична евристична оцінка. Таким чином, NNUE стало однією з найефективніших технологій для шахових двигунів, поєднуючи точність нейромереж із швидкістю класичних алгоритмів[19].

РОЗДІЛ ІІ. РОЗРОБКА ПРОТОТИПУ СИСТЕМИ ОЦІНКИ ШАХОВОЇ ПОЗИЦІЇ

У цьому розділі сформульовано вимоги до програмного забезпечення, яке моделюватиме процес прийняття рішень шляхом пошуку вглиб у графі на прикладі шахової гри. Головною метою створення моделі є дослідження та порівняння ефективності існуючих методів оптимізації задачі прийняття рішень.

2.1 НЕФУНКЦІОНАЛЬНІ ВИМОГИ

Для розробки обрано мову Python, що пояснюється її високим рівнем абстракції, наявністю великої кількості готових бібліотек для роботи з шахами, алгоритмами та графами, а також широким використанням у наукових дослідженнях. Альтернативи на кшталт C++ або Java були відхилені з огляду на більш тривалий час розробки та складність у швидкій інтеграції нових компонентів. Python, хоча й поступається у швидкодії мовам нижчого рівня, дозволяє зосередитись саме на дослідницькому аспекті, а не на деталях оптимізації реалізації. Для керування версіями та організації колективної роботи використовується GitHub. Середовище розробки Visual Studio Code надає можливості інтеграції з системами контролю версій, візуалізації коду та зручного налагодження. Керування зовнішніми бібліотеками здійснюється через стандартний пакетний менеджер pip (версія 25.2), що гарантує сумісність залежностей.

Ключовою сторонньою бібліотекою є python-chess, яка реалізує шаховий рушій, генерацію ходів, підтримку правил та зручний інтерфейс для роботи з шаховими позиціями. Дана бібліотека є де-факто стандартом у дослідженнях комп’ютерних шахів на Python і має стабільну підтримку. Архітектурно програма буде побудована модульно, із використанням парадигми об’єктно орієнтованого програмування, що дозволить спростити тестування і модифікацію різних версій алгоритму.

Також одним із ключових інструментів у дослідженні стала бібліотека PyTorch, яка виступає базовим фреймворком для побудови та навчання глибинних нейронних мереж. Завдяки динамічному обчислювальному графу PyTorch дозволяє оперативно модифікувати структуру мережі, що особливо важливо при дослідженні архітектурних рішень, подібних до ResNet-блоків чи трансформерних механізмів, що застосовуються у сучасних розробках у сфері нейронних мереж. Підсистема автоматичного диференціювання (autograd) забезпечує коректний і швидкий розрахунок похідних, необхідних для алгоритмів оптимізації, а вбудована підтримка апаратного прискорення на GPU робить PyTorch придатним для реалізації складних моделей у межах прийнятного часу навчання.

Допоміжну роль у підготовці даних та виконанні низькорівневих обчислень відіграє бібліотека NumPy. Вона використовується для роботи з багатовимірними масивами, обробки шахових позицій, формування тензорів та виконання базових лінійно-алгебраїчних операцій, що передують їх передачі у PyTorch. Оскільки NumPy оптимізовано для швидкої роботи на рівні низькорівневих обчислювальних процедур, він виступає важливим елементом у підготовчому етапі тренування нейронної мережі.

Для моніторингу процесу виконання тривалих обчислювальних операцій використовувався пакет tqdm, який забезпечує формування інформативних індикаторів прогресу. Цей інструмент не впливає на алгоритмічний чи математичний аспекти роботи, однак значно покращує контроль та відтворюваність експериментів, дозволяючи оперативно оцінювати швидкість проходження епох, генерації самонавчальних партій або обробки великих обсягів даних.

Ще одним ключовим компонентом, що забезпечує практичну можливість тренування великих нейронних моделей, є апаратно-орієнтована технологія NVIDIA CUDA. CUDA надає інтерфейси для ефективного виконання паралельних обчислень на графічних процесорах, які здатні одночасно обробляти тисячі потоків. Це дозволяє прискорити виконання операцій, пов’язаних з матричною алгеброю, згортками та механізмами уваги, які становлять основу сучасних

архітектур глибинного навчання. Взаємодія PyTorch із драйверами CUDA забезпечує ефективне розподілення обчислювальних ресурсів, що є критично важливим у завданнях, пов’язаних із тренуванням великих, або складних моделей. Використання CUDA істотно скорочує час навчання та дозволяє проводити масштабні експерименти, недосяжні при суто CPU-орієнтованих обчисленнях.

2.2 ФУНКЦІОНАЛЬНІ ВИМОГИ

Основною функцією програми-рушія є аналіз шахової позиції із застосуванням алгоритмів пошуку вглиб. Передбачено два режими роботи: аналіз на заздалегідь визначену глибину, встановлену користувачем, або аналіз до максимально можливої глибини за обмеженням часу. Такий підхід відображає реальні сценарії використання шахових програм: або дослідник хоче отримати контрольований результат із заданою точністю, або обмежений ресурсами часу та прагне досягнути оптимальної глибини у межах доступного ліміту.

Результатом роботи алгоритму є не лише вибір найкращого ходу, а й більш комплексний вихідний набір даних. Зокрема, програма повинна виводити комбіновану оцінку позиції, що базується на матеріальному балансі та позиційних факторах. Такий підхід дозволяє уникнути спрощення оцінки лише до кількості фігур, що часто критикується у класичних евристичних системах. Додатково виводиться оптимальна послідовність ходів, яка ілюструє, як саме алгоритм прийшов до вибраного рішення, що важливо для перевірки якості розрахунку. Третім компонентом є оцінка позиції після реалізації цієї послідовності, що дозволяє порівняти короткострокові та середньострокові наслідки прийнятих рішень. Нарешті, передбачається вивід метрик, які використовуються для оцінки ефективності роботи алгоритму. Така багатокомпонентна структура виходу робить програму не лише функціональним інструментом, а й дослідницькою платформою.

2.3 ОПИС РОЗРОБЛЕНОГО ПРОГРАМНОГО КОДУ

В рамках кваліфікаційної роботи було розроблено комплекс скриптів Скрипт-арена - для симуляції і оцінювання програм-рушіїв. Еталонний евристичний рушій – виключно алгоритмічний скрипт, що не використовує нейронну мережу, реалізує основні алгоритми винайдені програмістами, що займаються шаховими рушіями, та аналітикою у сфері шахів. Та кілька скриптів-тренерів – що були використані для тренування, та проміжного оцінювання нейронних мереж.

2.3.1 СКРИПТ-АРЕНА

Цей скрипт реалізує комплексну систему порівняльного тестування двох шахових рушіїв — евристичного та нейронного — з автоматичним формуванням PGN-записів, збором продуктивних метрик та залученням зовнішнього еталонного рушія Stockfish для оцінювання якості ходів. Структура програми є модульною, компонентною і відповідає сучасним практикам побудови експериментальних платформ для аналізу ігрового штучного інтелекту.

У рядках 1–14 виконується підключення стандартних бібліотек для роботи з шахами (python-chess), системними ресурсами (psutil, os), обробкою даних (pandas, numpy), діагностикою (traceback) та генерацією випадкових чисел (random). Зокрема, імпорт модуля chess.engine (рядок 2) забезпечує взаємодію з рушієм Stockfish через UCI-протокол, а chess.pgn (рядок 3) дає змогу формувати дерево ходів у форматі PGN. Додатково рядок 14 підключає компонент datetime, необхідний для генерації унікальних імен файлів під час збереження партій. У блоці конфігурації рядки 17–21 визначають ключові параметри експерименту: шлях до рушія Stockfish, кількість запланованих партій, часовий контроль, обмеження на кількість ходів, а також кількість PGN-файлів, які зберігатимуться після завершення матчу.

```
1  import chess
2  import chess.engine
3  import chess.pgn
4  import time
5  import psutil
6  import os
7  import pandas as pd
8  import numpy as np
9  import traceback
10 import random
11 from datetime import datetime # Added for unique filenames
12
13 # --- CONFIGURATION ---
14 STOCKFISH_PATH = "../stockfish-windows-x86-64-avx2/stockfish/stockfish-windows-x86-64-avx2.exe"
15 MODEL_FILE = "chess_nn_best.pt" # Points to your .pt file
16 GAMES_TO_PLAY = 2
17 TIME_LIMIT = 15.0 # S on move
18 MAX_MOVES = 100
19 # --- NEW CONFIGURATION ---
20 PGN_GAMES_TO_SAVE = 2 # Number of PGN files to save at the end
21
```

Рисунок 2.1 – скрипт-арена, рядки 1-21

Блок рядків 22–35 відповідає за імпорт двох користувацьких рушіїв: евристичного (HeuristicEngine), якщо файл Pure_Heur_bot.py доступний; нейронного (NNEngine), який інкапсулює модель глибинного навчання та адаптер для оцінки шахових позицій. При недоступності будь-якого модуля система переходить у режим обробки помилок, ініціюючи відповідні попередження (рядки 26–28 та 33–35). Таким чином забезпечуються обробка відсутності залежностей та стійкість роботи програми.

```
22 # Import your engines
23 # 1. Pure Heuristic
24 try:
25     from Pure_Heur_bot import HeuristicEngine
26 except ImportError:
27     print("WARNING: Pure_Heur_bot.py not found.")
28     HeuristicEngine = None
29
30 # 2. Neural Network Engine (Imports from File 1)
31 try:
32     from NN_Reinforced_adapter import NNEngine
33 except ImportError:
34     print("CRITICAL WARNING: nn_engine.py not found. NNEngine will fail.")
35     NNEngine = None
36
```

Рисунок 2.2 – скрипт-арена, рядки 22-36

У рядках 40–90 визначено класи-адаптери, які виконують роль уніфікованого інтерфейсу між рушіями, серед них:

1) Базовий клас Adapter (рядки 40–47) оголошує стандартні поля (name, nodes, depth) та метод get_move(), який перевизначається у похідних класах.

```
40  class Adapter:
41      def __init__(self, name):
42          self.name = name
43          self.nodes = 0
44          self.depth = 0
45
46      def get_move(self, board):
47          pass
48
```

Рисунок 2.3 – скрипт-арена, рядки 40-48

2) HeuristicAdapter: У рядках 49–66 реалізовано адаптер евристичного рушія. Основний метод get_move() (рядки 57–66): виклик select_move в евристичному рушії; збір статистики кількості відвіданих вузлів (nodes_visited); обробку помилок із fallback на випадковий хід, що підвищує стійкість експерименту.

```
49  class HeuristicAdapter(Adapter):
50      def __init__(self):
51          super().__init__("Heuristic Bot")
52          if HeuristicEngine:
53              self.engine = HeuristicEngine()
54          else:
55              self.engine = None
56
57      def get_move(self, board):
58          if not self.engine: return random.choice(list(board.legal_moves))
59          try:
60              move = self.engine.select_move(board, time_limit=TIME_LIMIT)
61              self.nodes = getattr(self.engine, 'nodes_visited', 0)
62              self.depth = 0
63              return move
64          except Exception as e:
65              print(f"Heuristic Crash: {e}")
66              return None
```

Рисунок 2.4 – скрипт-арена, рядки 49-66

3) У рядках 68–88 представлено адаптер нейромережевого рушія, який завантажує модель з файлу (MODEL_FILE). Метод get_move() виконує: пошук найкращого ходу із поверненням оцінки (move, score); фіксацію глибини пошуку (self.depth); діагностику помилок з виведенням трасування (traceback.print_exc()).

```
68  class NNEngineAdapter(Adapter):
69      def __init__(self):
70          super().__init__("NN")
71          if NNEngine and os.path.exists(MODEL_FILE):
72              self.engine = NNEngine(MODEL_FILE)
73          else:
74              print("WARNING: Model file missing or NNEngine class not found.")
75              self.engine = None
76
77      def get_move(self, board):
78          if not self.engine: return random.choice(list(board.legal_moves))
79          try:
80              move, score = self.engine.get_best_move(board)
81              self.nodes = getattr(self.engine, 'nodes_visited', 0)
82              self.depth = self.engine.depth
83              return move
84          except Exception as e:
85              print(f"NN Crash: {e}")
86              traceback.print_exc()
87              return None
88
```

Рисунок 2.5 – скрипт-арена, рядки 68-88

В рядках 92-142 імплементовано клас Tracker, що виконує функції збору метрик і відіграє роль централізованого моніторингового компоненту. Його призначення полягає у вимірюванні ресурсоспоживання, якості ходів та ефективності алгоритмів під час шахової партії між двома різними підходами до побудови ігрового рушія — евристичною моделлю та нейронною мережею. У конструкторі класу (рядки 95–101) оголошено словник stats, який містить структури даних для двох типів агентів: "Heuristic Bot" та "NN". Для кожного з агентів зберігаються такі параметри: Time — кількість часу, витраченого на вибір кожного ходу; Nodes — кількість проаналізованих позицій (рядки 47, 74); Mem — оперативна пам’ять, спожита процесом під час обчислення; ACPL (Average Centipawn Loss) — середня втрата точності ходів у сантипешках, що є стандартною шаховою метрикою якості; Wins — кількість виграних партій.

Важливим елементом підсистеми є взаємодія зі стороннім рушієм Stockfish, який у цьому контексті виступає незалежним експертом-оцінювачем позицій. Функція start_sf() (рядки 100–109) ініціалізує рушій через протокол UCI. Якщо рушій відсутній або некоректно встановлений, система інформує про це

користувача. Завершення роботи рушія реалізовано методом stop_sf() (рядки 110–112). Оцінка позиції здійснюється методом get_eval() (рядки 114–120). Запускається аналіз позиції з обмеженням глибини (depth=12), що дозволяє отримати числову оцінку у Centipawn. Ця інформація використовується під час обчислення ACPL — метрики, що відображає точність гри порівняно з оптимальними ходами Stockfish.

```
92  class Tracker:
93      def __init__(self):
94          self.stats = {
95              "Heuristic Bot": {"Time": [], "Nodes": [], "Mem": [], "ACPL": [], "Wins": 0},
96              "NN":     {"Time": [], "Nodes": [], "Mem": [], "ACPL": [], "Wins": 0}
97          }
98          self.sf_engine = None
99
100     def start_sf(self):
101         if os.path.exists(STOCKFISH_PATH):
102             try:
103                 self.sf_engine = chess.engine.SimpleEngine.popen_uci(STOCKFISH_PATH)
104             except Exception as e:
105                 print(f"Stockfish Init Failed: {e}")
106         else:
107             if "stockfish" in STOCKFISH_PATH.lower():
108                 print(f"WARNING: Stockfish not found at {STOCKFISH_PATH}")
109
110     def stop_sf(self):
111         if self.sf_engine:
112             self.sf_engine.quit()
113
114     def get_eval(self, board):
115         if not self.sf_engine: return 0
116         try:
117             info = self.sf_engine.analyse(board, chess.engine.Limit(depth=12))
118             return info["score"].relative.score(mate_score=10000)
119         except: return 0
120
```

Рисунок 2.6 – скрипт-арена, рядки 92-120

Функція record_move() (рядки 127–136) зберігає статистику обчислювальних витрат, фіксуючи: час виконання (аргумент time_taken); кількість переглянутих вузлів пошуку (nodes); використання RAM, отримане через модуль psutil (рядок 133). Завершальною частиною метрик є обчислення ACPL, реалізоване у методі calculate_acpl() (рядки 138–155). Алгоритм функціонує таким чином: Оцінюється поточна позиція перед ходом (best_eval). Після виконання ходу (board.push(move)), позиція оцінюється повторно (actual_eval). Різниця між оцінками визначає втрату

точності. Значення обрізається зверху (мінімізація артефактів), відповідно до загальноприйнятого обмеження ACPL ≤ 900.

```
121     def record_move(self, engine_name, time_taken, nodes, depth):
122         if engine_name not in self.stats: return
123         self.stats[engine_name]["Time"].append(time_taken)
124         self.stats[engine_name]["Nodes"].append(nodes)
125 
126         process = psutil.Process(os.getpid())
127         mem_mb = process.memory_info().rss / 1024 / 1024
128         self.stats[engine_name]["Mem"].append(mem_mb)
129 
130     def calculate_acpl(self, engine_name, board, move):
131         if not self.sf_engine: return
132 
133         best_eval = self.get_eval(board)
134         board.push(move)
135         actual_eval = -self.get_eval(board)
136         board.pop()
137 
138         loss = max(0, best_eval - actual_eval)
139         loss = min(loss, 900)
140 
141         if engine_name in self.stats:
142             self.stats[engine_name]["ACPL"].append(loss)
143 
```

Рисунок 2.7 – скрипт-арена, рядки 121-143

Основне тіло скрипта - рядки 164–266: Основний контроль за виконанням шахових партій здійснюється функцією run_match() (рядок 164). Цей модуль відповідає за організацію матчу, почерговий виклик ботів, логіку завершення гри та фіксацію результатів. На початку (рядки 166–172) відбувається ініціалізація адаптерів для обох агентів: евристичного (HeuristicAdapter) та нейромережевого (NNEngineAdapter).

Обидва адаптери є реалізаціями патерну “обгортка”, що дозволяє стандартизувати інтерфейс між різними алгоритмами прийняття рішень. Після цього створюється об’єкт Tracker, який активує рушій Stockfish для подальшої оцінки ходів.

```
144  # ======================================================
145  #  Match Logic (Modified)
146  # ======================================================
147  def run_match():
148      # Initialize Adapters
149      white_engine_adapter = HeuristicAdapter()
150      black_engine_adapter = NNEngineAdapter()
151
152      tracker = Tracker()
153      tracker.start_sf()
154
155      # NEW: List to hold completed game objects
156      completed_games = []
157
158      print(f"--- Starting {GAMES_TO_PLAY} Game Match ---")
159
```

Рисунок 2.8 - скрипт-арена, рядки 144-159

Ініціалізується список completed_games, що зберігатиме готові PGN-об’єкти партій. Основний цикл матчу (рядок 177) виконується кількість разів, визначену параметром GAMES_TO_PLAY (рядок 10). Для кожної партії:

1. Створюється нова шахова дошка (chess.Board()), а також PGN-об’єкт (chess.pgn.Game()) — рядки 161–164.

2. Встановлюються заголовки PGN: подія, локація, імена гравців (рядки 166–181). Реалізується механізм зміни кольорів після кожної партії, що забезпечує баланс між умовами гри.

```
158     print(f"--- Starting {GAMES_TO_PLAY} Game Match ---")
159
160     for game_idx in range(GAMES_TO_PLAY):
161         board = chess.Board()
162         game = chess.pgn.Game() # NEW: Initialize PGN Game object
163         game.headers["Event"] = "Arena Match"
164         game.headers["Site"] = "Local"
165
166         # Swap colors
167         if game_idx % 2 == 1:
168             p1, p2 = black_engine_adapter, white_engine_adapter
169             game.headers["White"] = p2.name
170             game.headers["Black"] = p1.name
171         else:
172             p1, p2 = white_engine_adapter, black_engine_adapter
173             game.headers["White"] = p1.name
174             game.headers["Black"] = p2.name
175
176         node = game # Start PGN node at the root of the game
177
178         print(f"\nGame {game_idx+1}: White={p1.name}, Black={p2.name}")
179
180         moves_count = 0
181         while not board.is_game_over() and moves_count < MAX_MOVES:
182             current_player = p1 if board.turn == chess.WHITE else p2
183
```

Рисунок 2.9 - скрипт-арена, рядки 158-183

3. Внутрішній цикл ходів (рядки 181–224) виконується до моменту: завершення партії за шаховими правилами (board.is_game_over()), або досягнення ліміту ходів MAX_MOVES (рядок 12). На кожному кроці: Визначається активний гравець залежно від кольору (board.turn). Вимірюється час обчислення ходу (рядки 206–208). Викликається метод get_move() відповідного адаптера. Якщо хід некоректний або відсутній — агент автоматично програє (рядки 210–214). Оновлюються метрики через виклики record_move() та calculate_acpl() (рядки 216–219). Хід додається до PGN-дерева (node.add_variation(move), рядок 222).

```
200         while not board.is_game_over() and moves_count < MAX_MOVES:
201             current_player = p1 if board.turn == chess.WHITE else p2
202 
203             # Print status
204             print(f"\rMove {moves_count+1}: {current_player.name} thinking...", end="")
205 
206             t0 = time.time()
207             move = current_player.get_move(board)
208             dt = time.time() - t0
209 
210             if move is None or move not in board.legal_moves:
211                 print(
212                     f"\nCRITICAL: {current_player.name} returned illegal/no move. Resigning."
213                 )
214                 break
215 
216             tracker.record_move(
217                 current_player.name, dt, current_player.nodes, current_player.depth
218             )
219             tracker.calculate_acpl(current_player.name, board, move)
220 
221             board.push(move)
222             node = node.add_variation(move)  # NEW: Add move to PGN tree
223             moves_count += 1
224 
225         res = board.result(claim_draw=True)
226         game.headers["Result"] = res  # Set the final result header
227         print(f"\nGame Over. Result: {res}")
```

Рисунок 2.10 - скрипт-арена, рядки 215-227

Система підраховує кількість перемог для кожного агента (рядки 229–237), а після завершення всього матчу викликається логіка збереження PGN-файлів (рядки 241–262). Файли маркуються часовою позначкою (datetime.now(), рядок 251), що забезпечує їх унікальність.

```
229         # NEW: Store the completed game object
230         completed_games.append(game)
231 
232         if res == "1-0":
233             tracker.stats[p1.name]["Wins"] += 1
234         elif res == "0-1":
235             tracker.stats[p2.name]["Wins"] += 1
236         elif res == "1/2-1/2":
237             pass
238 
239     tracker.stop_sf()
```

Рисунок 2.11 - скрипт-арена, рядки 229-239

```
241     # --- PGN SAVING LOGIC ---
242     print(f"\n--- SAVING LAST {PGN_GAMES_TO_SAVE} GAMES ---")
243
244     # Determine which games to save (the last N games)
245     games_to_save = completed_games[-PGN_GAMES_TO_SAVE:]
246
247     for i, game in enumerate(games_to_save):
248         # Calculate the original index to make the filename informative
249         original_game_idx = GAMES_TO_PLAY - len(games_to_save) + i + 1
250
251         timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
252         filename = f"game_{original_game_idx:02d}_{game.headers['White']}_vs_{game.headers['Black']}_{timestamp}.pgn"
253
254         try:
255             with open(filename, "w") as f:
256                 f.write(str(game))
257             print(f"Saved: {filename}")
258         except Exception as e:
259             print(f"Error saving PGN file {filename}: {e}")
260
```

Рисунок 2.12 - скрипт-арена, рядки 241-260

Заключним етапом є підготовка підсумкової таблиці результатів за допомогою pandas (рядки 265–266). Таблиця включає середні значення часових витрат, кількості оброблених вузлів, RAM, ACPL і загальний рахунок перемог — параметри, які становлять основу для подальшого аналітичного інтерпретування гри агентів.

```
261     # --- TOURNAMENT RESULTS DISPLAY ---
262     print("\n\n=== TOURNAMENT RESULTS ===")
263
264     results = []
265     for name, data in tracker.stats.items():
266         results.append(
267             {
268                 "Engine": name,
269                 "Wins": data["Wins"],
270                 "Avg Time (s)": f"{np.mean(data['Time']) if data['Time'] else 0:.2f}",
271                 "Avg Nodes": int(np.mean(data["Nodes"])) if data["Nodes"] else 0,
272                 "Avg RAM (MB)": int(np.mean(data["Mem"])) if data["Mem"] else 0,
273                 "Avg ACPL": int(np.mean(data["ACPL"])) if data["ACPL"] else 0,
274             }
275         )
276
277     df = pd.DataFrame(results)
278     print(df.to_string(index=False))
279
280
281 if __name__ == "__main__":
282     run_match()
283
```

Рисунок 2.13 - скрипт-арена, рядки 261-283

2.3.2 ЕТАЛОННИЙ ШАХОВИЙ РУШІЙ

Скрипт реалізує евристичний еталонний шаховий рушій, який використовується як базовий алгоритмічний компонент для порівняння з іншими моделями. Код має класичну структуру, притаманну традиційним шаховим двигунам доепохи глибинного навчання, і включає модулі оцінювання позиції, порядкування ходів, пошуку з альфа-бета відсіканням, квазістаціонарного пошуку та позиційного інтерполювання між фазами гри.

Евристичні бази знань (рядки 6–147). Першим елементом рушія є система евристично налаштованих знань, яка складається з двох типів даних:

1) Матеріальні оцінки фігур (PIECE_VALS, рядки 14–23), представлені як пари значень для середньої гри (MG) та ендшпілю (EG).

```
1   import chess
2   import time
3
4   # ================================================================================
5   #  PeSTO Evaluation Tables (The "Knowledge")
6   #  These are tuned values used by strong engines.
7   #  Format: [MiddleGame Value, EndGame Value]
8   # ================================================================================
9
10  # Piece values (MG, EG)
11  PIECE_VALS = {
12      chess.PAWN: (82, 94),
13      chess.KNIGHT: (337, 281),
14      chess.BISHOP: (365, 297),
15      chess.ROOK: (477, 512),
16      chess.QUEEN: (1025, 936),
17      chess.KING: (0, 0),
18  }
19
```

Рисунок 2.14 - скрипт-еталонний рушій, рядки 1-20

2) Таблиці «фігура-клітина» (PST), що містять 64-елементні матриці позиційної цінності для кожного типу фігур (рядки 27–147). Ці таблиці відображають стандартний підхід так званої PeSTO evaluation, відомий у класичних рушіях, зокрема у Stockfish, Komodo та інших двигунах покоління до появи нейромережевих оцінювачів. Значення PST дозволяють оцінювати: контроль центру; оптимальні квадрати для розвитку фігур; ефективність структури пішаків;

безпеку короля Формат таблиць відповідає числовому діапазону, сталому для «класичних» оцінювачів (від –150 до +200), що забезпечує швидкість обчислень і простоту подальшої оптимізації.

```
20  # Piece-Square Tables (Flipped for Black automatically in logic)
21  # Tables are 64 values, from A1 to H8.
22  # These tables encourage central control, king safety, and pawn advancement.
23
24  # (Shortened for brevity, these are simplified PeSTO-style tables)
25  PST = {
26      chess.PAWN: (
27          [ 0,  0,  0,  0,  0,  0,  0,  0,
28           98, 134, 61, 95, 68, 126, 34, -11,
29           -6,  7, 26, 31, 65, 56, 25, -20,
30          -14, 13, 6, 21, 23, 12, 17, -23,
31          -27, -2, -5, 12, 17, 6, 10, -25,
32          -26, -4, -4, -10, 3, 3, 33, -12,
33          -35, -1, -20, -23, -15, 24, 38, -22,
34            0,  0,  0,  0,  0,  0,  0,  0], # MG
35          [ 0,  0,  0,  0,  0,  0,  0,  0,
36          178, 173, 158, 134, 147, 132, 165, 187,
37           94, 100, 85, 67, 56, 53, 82, 84,
38           32, 24, 13, 5, -2, 4, 17, 17,
39           13, 9, -3, -7, -7, -8, 3, -1,
40            4, 7, -6, 1, 0, -5, -1, -8,
41           13, 8, 8, 10, 13, 0, 2, -7,
42            0,  0,  0,  0,  0,  0,  0,  0]  # EG
43      ),
44      chess.KNIGHT: (
45          [-167, -89, -34, -49, 61, -97, -15, -107,
```

Рисунок 2.15 - скрипт-еталонний рушій, рядки 20-45

Структура класу рушія (рядки 136–150). Клас HeuristicEngine ініціалізує базові структури: транспозиційну таблицю (transposition_table), що слугує кешем позицій; лічильник відвіданих вузлів (nodes_visited); механізм контролю часу (time_limit, start_time); змінну stop_search для екстреного завершення пошуку. Така архітектура є типовою для компактних рушіїв і забезпечує достатню ефективність для швидких локальних симуляцій.

```
136  # ================================================================================
137  #  The Engine Class
138  # ================================================================================
139
140
141  class HeuristicEngine:
142      def __init__(self):
143          self.transposition_table = {}
144          self.nodes_visited = 0
145          self.start_time = 0
146          self.time_limit = 0
147          self.stop_search = False
148
149      # --- 1. TAPERED EVALUATION ---
150      def evaluate(self, board):
```

Рисунок 2.16 - скрипт-еталонний рушій, рядки 136-150

Модуль оцінювання (evaluate) — перемикання між фазами гри (рядки 150–200). Метод evaluate() реалізує класичний підхід tapered evaluation, у якому оцінка позиції є інтерполяцією між двома моделями: оцінкою середньої гри (Middle Game Score), оцінкою ендшпілю (End Game Score). Рядки 156–176 реалізують обчислення фази гри, що враховує:

1) Підрахунок кіль-сті матеріалу: 1 за коня, 1 за слона, 2 за туру, 4 за ферзя — загалом максимум 24 одиниці (коли на дошці присутні всі важливі фігури).

```
149      # --- 1. TAPERED EVALUATION ---
150      def evaluate(self, board):
151          if board.is_checkmate():
152              return -99999 if board.turn else 99999
153          if board.is_game_over():
154              return 0
155
156          # Calculate Game Phase (24 = Full material without kings/pawns)
157          # Knight=1, Bishop=1, Rook=2, Queen=4
158          mg_score = 0
159          eg_score = 0
160          game_phase = 0
```

Рисунок 2.17 - скрипт-еталонний рушій, рядки 148-160

2) Рядки 180–200 забезпечують формування MG та EG суми. Після цього (рядок 199) масштабування відбувається відносно сторони, яка ходить, що є стандартом у рушіях на основі negamax.

```
162         for square, piece in board.piece_map().items():
163             pt = piece.piece_type
164             color = piece.color
165 
166             # Calculate Phase
167             if pt == chess.KNIGHT:
168                 game_phase += 1
169             elif pt == chess.BISHOP:
170                 game_phase += 1
171             elif pt == chess.ROOK:
172                 game_phase += 2
173             elif pt == chess.QUEEN:
174                 game_phase += 4
175 
176             # Calculate Scores
177             # If White, index is flipped (63 - square) because our tables are 0..63 from A1
178             idx = square if color == chess.WHITE else chess.square_mirror(square)
179 
180             # Material + PST
181             mg_val = PIECE_VALS[pt][0] + PST[pt][0][idx]
182             eg_val = PIECE_VALS[pt][1] + PST[pt][1][idx]
183 
184             if color == chess.WHITE:
185                 mg_score += mg_val
186                 eg_score += eg_val
187             else:
188                 mg_score -= mg_val
189                 eg_score -= eg_val
190
```

Рисунок 2.18 - скрипт-еталонний рушій, рядки 162-190

```
190 
191         # Interpolate between Middle Game and End Game
192         # Max phase is 24.
193         mg_phase = min(game_phase, 24)
194         eg_phase = 24 - mg_phase
195 
196         evaluation = ((mg_score * mg_phase) + (eg_score * eg_phase)) / 24
197 
198         # Return score relative to side to move
199         return evaluation if board.turn == chess.WHITE else -evaluation
200 
```

Рисунок 2.19 - скрипт-еталонний рушій, рядки 190-200

Порядкування ходів (order_moves) — Пріоритет захоплень та хеш-ходу (рядки 200–230). Метод order_moves() виконує критично важливе оптимізаційне завдання — визначення порядку обчислення ходів для мінімізації глибини дерева пошуку. Реалізовано такі евристики:

1) Hash move first — найцінніший хід з транспозиційної таблиці отримує фіксовану високу оцінку (10000), рядки 207–208.

2) MVV–LVA (Most Valuable Victim – Least Valuable Aggressor), рядки 209–221.

Сортування виконується у спадаючому порядку (рядок 226), що істотно підвищує ефективність альфа-бета відсікання.

```
201     # --- 2. MOVE ORDERING ---
202     def order_moves(self, board, moves, best_move=None):
203         """Sorts moves to improve pruning: Captures first."""
204         scored_moves = []
205         for move in moves:
206             score = 0
207             if move == best_move:
208                 score = 10000  # Try hash move first
209             elif board.is_capture(move):
210                 # MVV-LVA (Most Valuable Victim - Least Valuable Aggressor)
211                 # Attacker: P=1, N=3... Victim: P=1, Q=9
212                 # Score = Victim * 10 - Attacker
213                 attacker = board.piece_at(move.from_square).piece_type
214                 victim_p = board.piece_at(move.to_square)
215                 victim = (
216                     victim_p.piece_type if victim_p else chess.PAWN
217                 )  # En passant assumption
218 
219                 # Approximate values
220                 vals = {1: 1, 2: 3, 3: 3, 4: 5, 5: 9, 6: 0}
221                 score = vals.get(victim, 1) * 10 - vals.get(attacker, 1) + 100
222 
223             scored_moves.append((score, move))
224 
225         # Sort descending
226         scored_moves.sort(key=lambda x: x[0], reverse=True)
227         return [m[1] for m in scored_moves]
```

Рисунок 2.20 - скрипт-еталонний рушій, рядки 201-230

Квазістаціонарний пошук (quiescence) — боротьба з ефектом горизонту (рядки 230–250). Метод quiescence() обмежує ефект горизонту, розглядаючи лише ходи-взяття після досягнення глибини 0. Структура компонента:

1) Рядки 234–238 — стоячі оцінки (stand-pat), які визначають базову якість позиції.

2) Рядки 245–246 — перевірка «cutoff» за β-межами.

3) Рядки 243–252 — рекурсивний аналіз усіх можливих захоплень з використанням negamax-симетрії.

```
229     # --- 3. QUIESCENCE SEARCH ---
230     def quiescence(self, board, alpha, beta):
231         """Searches captures only to prevent horizon effect."""
232         self.nodes_visited += 1
233 
234         stand_pat = self.evaluate(board)
235         if stand_pat >= beta:
236             return beta
237         if alpha < stand_pat:
238             alpha = stand_pat
239 
240         moves = list(board.generate_legal_captures())
241         moves = self.order_moves(board, moves)
242 
243         for move in moves:
244             board.push(move)
245             score = -self.quiescence(board, -beta, -alpha)
246             board.pop()
247 
248             if score >= beta:
249                 return beta
250             if score > alpha:
251                 alpha = score
252         return alpha
```

Рисунок 2.21 - скрипт-еталонний рушій, рядки 230-250

Основний алгоритм пошуку — Negamax з альфа-бета відсіканням (рядки 254–326). Метод negamax() є ядром усього рушія. Він реалізує симетризовану форму minimax, у якій оцінка позиції для опонента зводиться до зміни знаку. Основні компоненти:

1) Перевірка ліміту часу (рядки 258–263) — кожні 2048 вузлів (а не кожен хід) перевіряється, чи вичерпано час.

2) Транспозиційна таблиця (рядки 265–275) — реалізовано типове TT-кешування з прапорцями exact/lower/upper, що знижує кількість повторних аналізів.

```
254     # --- 4. NEGAMAX (ALPHA-BETA) ---
255     def negamax(self, board, depth, alpha, beta):
256         self.nodes_visited += 1
257 
258         # Check Time
259         if self.nodes_visited % 2048 == 0:
260             if time.time() - self.start_time > self.time_limit:
261                 self.stop_search = True
262         if self.stop_search:
263             return 0
264 
265         # Transposition Table Lookup
266         board_key = board.fen()  # Simple key
267         if board_key in self.transposition_table:
268             entry = self.transposition_table[board_key]
269             if entry["depth"] >= depth:
270                 # Can use stored value?
271                 if entry["flag"] == "exact":
272                     return entry["score"]
273                 if entry["flag"] == "lower" and entry["score"] >= beta:
274                     return beta
275                 if entry["flag"] == "upper" and entry["score"] <= alpha:
276                     return alpha
277 
```

Рисунок 2.22 - скрипт-еталонний рушій, рядки 254-277

3) Перевірка базових умов (рядки 278–283) — якщо глибина 0, провести квазістаціонарний пошук; перевірити дошку на кінцеву позицію, якщо гру завершену – перейти до оцінювання.

4) Move Ordering (рядки 285–291) — виклик методу класу, для сортування легальних позицій.

5) Основний цикл пошуку (рядки 296–310) — рекурсія з оновленням α та β, виконання відсікань.

```
278         # Base Case
279         if depth == 0:
280             return self.quiescence(board, alpha, beta)
281 
282         if board.is_game_over():
283             return self.evaluate(board)
284 
285         # Move Ordering
286         best_move_hash = None
287         if board_key in self.transposition_table:
288             best_move_hash = self.transposition_table[board_key].get("move")
289 
290         legal_moves = list(board.legal_moves)
291         ordered_moves = self.order_moves(board, legal_moves, best_move_hash)
292 
293         best_val = -999999
294         move_made = False
295 
296         for move in ordered_moves:
297             board.push(move)
298             val = -self.negamax(board, depth - 1, -beta, -alpha)
299             board.pop()
300 
301             if self.stop_search:
302                 return 0
303 
304             if val > best_val:
305                 best_val = val
306                 best_move_hash = move
307 
308             alpha = max(alpha, best_val)
309             if alpha >= beta:
310                 break  # Pruning
```

Рисунок 2.23 - скрипт-еталонний рушій, рядки 278-310

6) Оновлення транспозиційної таблиці (рядки 312–326) на основі результатів пошуку.

```
312         # Store in TT
313         tt_flag = "exact"
314         if best_val <= alpha:
315             tt_flag = "upper"
316         elif best_val >= beta:
317             tt_flag = "lower"
318 
319         self.transposition_table[board_key] = {
320             "score": best_val,
321             "depth": depth,
322             "flag": tt_flag,
323             "move": best_move_hash,
324         }
325 
326         return best_val
```

Рисунок 2.24 - скрипт-еталонний рушій, рядки 312-326

Ітеративне поглиблення та API (select_move) — публічний інтерфейс рушія (рядки 328–370). Метод select_move() виконує керування пошуком за обмеженням часу. Алгоритм:

1) Скидання статистики та очищення кешу (рядки 338–340).

2) Запуск ітераційного заглиблення (рядки 345–370). На кожному кроці: здійснюється пошук до глибини d, отримується найкращий хід з TT, глибина інкрементується.

Алгоритм зупиняється коли закінчується час, або буде досягнуто максимальну глибину (depth > 10).

```
328     # --- 5. PUBLIC API ---
329     def select_move(self, board, time_limit=1.0):
330         """
331         Iterative Deepening Search.
332         Returns the best move found within time_limit.
333         """
334         self.start_time = time.time()
335         self.time_limit = time_limit
336         self.stop_search = False
337         self.nodes_visited = 0
338         self.transposition_table = (
339             {}
340         )  # Clear TT for new move or keep it? Clearing is safer for simple engines.
341 
342         best_move = None
343         depth = 1
344 
345         # Iterative Deepening Loop
346         while True:
347             # Check if we are out of time before starting next depth
348             if time.time() - self.start_time > self.time_limit:
349                 break
350 
351             # Search
352             score = self.negamax(board, depth, -999999, 999999)
353 
354             # If search was aborted mid-depth, don't use result
355             if self.stop_search:
356                 break
```

Рисунок 2.25 - скрипт-еталонний рушій, рядки 328-356

```
358             # Retrieve best move from TT
359             entry = self.transposition_table.get(board.fen())
360             if entry and entry.get("move"):
361                 best_move = entry["move"]
362                 # Optional: Print info for debugging
363                 # print(f"Depth {depth} | Score: {score} | Move: {best_move} | Nodes: {self.nodes_visited}")
364 
365             depth += 1
366             # Hard cap depth to prevent infinite loops on super-fast positions
367             if depth > 10:
368                 break
369 
370         return best_move
371 
372 
373 # Example Usage if run directly
374 if __name__ == "__main__":
375     engine = HeuristicEngine()
376     board = chess.Board()
377     print("Thinking...")
378     move = engine.select_move(board, time_limit=2.0)
379     print("Best Move:", move)
380 
```

Рисунок 2.26 - скрипт-еталонний рушій, рядки 358-380

2.3.3 СКРИПТ-ОБГОРТКА ДЛЯ НЕЙРОННОЇ МЕРЕЖІ

Цей скрипт було реалізовано в декількох варіаціях в залежності від розміру мережі, розглянуто буде лише один. Представлений скрипт реалізує легковаговий нейромережевий шаховий рушій, який виконує функцію оціночного модуля на основі навченої згорткової нейронної мережі. Його структура об’єднує три ключові компоненти:

1) систему перетворення шахової позиції у тензор ознак;

2) архітектуру нейронної мережі для оцінки позиції;

3) механізм вибору найкращого ходу через групові обчислення вартості всіх кандидатів.

На відміну від класичних детермінованих алгоритмів оцінювання, цей модуль застосовує підхід, аналогічний до Lc0-подібних систем, де рішення формується статистичною моделлю, а не набором евристик.

Перетворення шахової позиції у вхідний тензор (рядки 1–38). Першим етапом роботи рушія є функція board_to_tensor_optimized(), яка формує 12-канальний тензор розмірності (12, 8, 8). В основу закладено принцип, поширений у нейромережах для ігрових задач (зокрема в AlphaZero та Leela Chess Zero): 6 каналів для білих фігур, 6 каналів для чорних фігур, кожен канал є бінарною картою

8×8, де 1.0 означає присутність фігури певного типу на конкретній клітині Тензор створюється методом прямого заповнення на рівні NumPy, що підвищує швидкість підготовки груп у порівнянні з поклітинною обробкою. Для узгодженості з навчальною процедурою використано попередньо сформовану таблицю SQ_TO_IDX, яка забезпечує інверсію нумерації ранґів відповідно до прийнятого формату навчальних даних. Це представлення відповідає раннім версіям Lc0 на основі 2D convolutional boards, забезпечуючи однорідність із процедурою генерації навчальних прикладів.

```
1   import torch
2   import torch.nn as nn
3   import numpy as np
4   import chess
5   import random
6
7   # ==================================================
8   #  Helpers (Must match Training Script)
9   # ==================================================
10  SQ_TO_IDX = np.zeros((64, 2), dtype=int)
11  for sq in range(64):
12      r, f = divmod(sq, 8)
13      SQ_TO_IDX[sq] = [7 - r, f]
14
15  def board_to_tensor_optimized(board):
16      planes = np.zeros((12, 8, 8), dtype=np.float32)
17      # White pieces (Channels 0-5)
18      for pt in range(1, 7):
19          for sq in board.pieces(pt, chess.WHITE):
20              r, f = SQ_TO_IDX[sq]
21              planes[pt-1, r, f] = 1.0
22      # Black pieces (Channels 6-11)
23      for pt in range(1, 7):
24          for sq in board.pieces(pt, chess.BLACK):
25              r, f = SQ_TO_IDX[sq]
26              planes[pt+5, r, f] = 1.0
27      return planes
28
```

Рисунок 2.27 - скрипт-обгортка для НН, рядки 1-30

Архітектура оціночної нейронної мережі ValueNet (рядки 30–60). Модель ValueNet реалізує компактну згорткову архітектуру, призначену для обчислення скалярної оцінки позиції у діапазоні [-1, 1] (білі виграють / чорні виграють). Структурно вона складається з двох основних частин:

1. Згорткове тіло (Conv Body) Послідовність із трьох згорткових блоків: Conv2d → BatchNorm → ReLU (64 каналів), Conv2d → BatchNorm → ReLU (128 каналів), Conv2d → BatchNorm → ReLU (128 каналів), завершених 1×1-згорткою, яка стискає простір каналів до 32. Цей дизайн зберігає просторову структуру шахової дошки, аналогічно тому, як це робиться у великих моделях AlphaZero/Lc0, але в меншому масштабі.

2. Головний прогностичний блок (Value Head). Flatten (вектор довжиною 32×8×8), Dense(256) + ReLU, Dense(1) + Tanh. Останній шар Tanh() застосовується для нормування виходу до діапазону [-1, 1], що робить оцінки інтерпретуваними як ймовірність виграшу, подібно до value-head у Lc0.

```
29  # ======================================================
30  #  Neural Network Architecture
31  # ======================================================
32  class ValueNet(nn.Module):
33      def __init__(self):
34          super().__init__()
35          self.body = nn.Sequential(
36              nn.Conv2d(12, 64, 3, padding=1),
37              nn.BatchNorm2d(64),
38              nn.ReLU(),
39              nn.Conv2d(64, 128, 3, padding=1),
40              nn.BatchNorm2d(128),
41              nn.ReLU(),
42              nn.Conv2d(128, 128, 3, padding=1),
43              nn.BatchNorm2d(128),
44              nn.ReLU(),
45              nn.Conv2d(128, 32, 1),
46              nn.ReLU(),
47          )
48          self.head = nn.Sequential(
49              nn.Flatten(),
50              nn.Linear(32 * 8 * 8, 256),
51              nn.ReLU(),
52              nn.Linear(256, 1),
53              nn.Tanh()
54          )
55
56      def forward(self, x):
57          return self.head(self.body(x))
58
```

Рисунок 2.28 - скрипт-обгортка для НН, рядки 30-60

Завантаження моделі и управління пристроєм (рядки 60–90). Конструктор класу NNEngine виконує: автоматичний вибір обчислювального пристрою (cuda, якщо доступно), створення екземпляра ValueNet, завантаження контрольної точки навчання. Реалізовано універсальний механізм завантаження: якщо структура відповідає формату тренувального скрипту (checkpoint["m"]), то використовується

словник ваг «моделі», а також виводиться метаінформація про покоління (generation) та найкращий win-rate; якщо файл містить лише state_dict, він завантажується без модифікацій. При помилці завантаження рушій переходить у «псевдонормальний» режим та виконує випадковий вибір ходу.

```
59  # ==================================================
60  #  Engine Class
61  # ==================================================
62  class NNEngine:
63      def __init__(self, model_path, device=None):
64          if device is None:
65              self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
66          else:
67              self.device = torch.device(device)
68
69          self.model = ValueNet().to(self.device)
70          self.nodes_visited = 0
71          self.depth = 1  # Standard 1-ply search
72
73          try:
74              # Load checkpoint; map_location ensures CPU/GPU compatibility
75              checkpoint = torch.load(model_path, map_location=self.device)
76
77              # Handle the dictionary structure from the training script
78              if isinstance(checkpoint, dict) and 'm' in checkpoint:
79                  self.model.load_state_dict(checkpoint['m'])
80                  print(f"   [NNEngine] Loaded Generation {checkpoint.get('g', '?')}")
81                  print(f"   (Best WR: {checkpoint.get('best_wr', 0):.1%})")
82              else:
83                  self.model.load_state_dict(checkpoint)
84                  print("   [NNEngine] Loaded raw state dictionary.")
85
86              self.model.eval()
87          except Exception as e:
88              print(f"   [NNEngine] ERROR Loading Model: {e}")
89              self.model = None
```

Рисунок 2.29 - скрипт-обгортка для НН, рядки 60-90

Механізм вибору ходу через групове оцінювання (рядки 90–125). Метод get_best_move() реалізує характерний для сучасних нейромережевих шахових агентів підхід: усі можливі ходи формують батч(группу), який обробляється одноразовою інференцією. Основні етапи:

1) Генерація усіх можливих ходів - Перебираються усі legal_moves, і для кожного виконується: зробити хід (push), конвертувати нову позицію у тензор, повернути попередній стан (pop()).

2) Формування батчу - Усі 12×8×8 представлення структуруються у тривимірний тензор: (N,12,8,8) де N — кількість легальних ходів.

3) Інференція моделі - Мережа повертає масив оцінок (N×1), після чого: білі обирають хід з максимальною оцінкою, чорні — з мінімальною, що відтворює zero-sum природу гри.

4) Підрахунок кількості відвіданих вузлів - Значення nodes_visited = len(legal_moves) дозволяє використовувати рушій у системах порівняльного аналізу ефективності.

```
91      def get_best_move(self, board):
92          """
93          Evaluates all legal moves and returns the best one.
94          Returns: (move, score)
95          """
96          if not self.model:
97              return random.choice(list(board.legal_moves)), 0.0
98
99          legal_moves = list(board.legal_moves)
100         if not legal_moves:
101             return None, 0.0
102
103         # Pre-process all resulting boards into a single batch
104         inputs = []
105         for move in legal_moves:
106             board.push(move)
107             inputs.append(board_to_tensor_optimized(board))
108             board.pop()
109
```

Рисунок 2.30 - скрипт-обгортка для НН, рядки 91-110

```
110         # Convert to tensor and run inference
111         input_tensor = torch.tensor(np.stack(inputs), dtype=torch.float32, device=self.device)
112
113         with torch.no_grad():
114             # Squeeze to remove batch dimension (N, 1) -> (N,)
115             scores = self.model(input_tensor).squeeze(1).cpu().numpy()
116
117         self.nodes_visited = len(legal_moves)
118
119         # White wants positive scores (1.0), Black wants negative (-1.0)
120         if board.turn == chess.WHITE:
121             best_idx = np.argmax(scores)
122         else:
123             best_idx = np.argmin(scores)
124
125         return legal_moves[best_idx], float(scores[best_idx])
```

Рисунок 2.31 - скрипт-обгортка для НН, рядки 110-125

2.3.4 ГІБРИДНИЙ РУШІЙ. MCTS ТА НЕЙРОННА МЕРЕЖА

Скрипт MCTS, що використовує для пошуку оцінки нейронної мережі. Скрипт складається з двох основних частин: підсистеми завантаження та інференсу нейронної мережі (рядки 67–97) та реалізації логіки MCTS (рядки 101–194).

Підсистема завантаження та інференсу нейронної мережі (рядки 67–97). Клас NNEvaluator (рядок 67) виконує роль обгортки для завантаження параметрів моделі та обчислення оцінки позиції. У конструкторі класу (рядки 68–82) ініціалізується пристрій виконання (cuda або cpu) та створюється екземпляр архітектури ValueNet(), що переміщується на відповідний пристрій. Модель переводиться у режим інференсу методом eval(). На рядках 74–81 реалізовано механізм завантаження ваг нейронної мережі з файлу model_path. Тут скрипт перевіряє існування файлу ваг, а також можливі варіанти їх збереження: або у вигляді "чистого" state_dict, або вкладеного у словник з ключем 'm'. У випадку успіху завантаження встановлюється відповідний прапорець self.loaded = True. Якщо файл відсутній або пошкоджений, на рядках 82–83 виводиться попереджувальне повідомлення. Метод get_value() (рядки 87–97) реалізує обчислення оціночної функції нейронної мережі. Якщо модель не завантажена (рядок 90), повертається нейтральне значення 0.0. В іншому випадку вхідна шахова позиція попередньо перетворюється у тензор методом board_to_tensor_optimized() (рядок 92) та подається на модель. Обчислена мережею оцінка — значення в інтервалі [-1; 1], що відображає перевагу білих, — повертається як результат (рядок 96).

```
62  # Wrapper to load the weights
63  class NNEvaluator:
64      def __init__(self, model_path="chess_nn_best.pt"):
65          self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
66          self.model = ValueNet().to(self.device)
67          self.model.eval()
68          self.loaded = False
69
70          if os.path.exists(model_path):
71              try:
72                  checkpoint = torch.load(model_path, map_location=self.device)
73                  if isinstance(checkpoint, dict) and 'm' in checkpoint:
74                      self.model.load_state_dict(checkpoint['m'])
75                  else:
76                      self.model.load_state_dict(checkpoint)
77                  print(f"[MCTS] NN Loaded on {self.device}")
78                  self.loaded = True
79              except Exception as e:
80                  print(f"[MCTS] NN Load Failed: {e}")
81          else:
82              print("[MCTS] Warning: Model file not found.")
83
```

Рисунок 2.32 – Скрипт для Мережі із алгоритмом MCTS рядки 62-83

```
84      def get_value(self, board):
85          """
86          Returns a value [-1, 1] representing White's advantage.
87          """
88          if not self.loaded: return 0.0
89
90          tensor = board_to_tensor_optimized(board)
91          input_tensor = torch.tensor(np.stack([tensor]), dtype=torch.float32, device=self.device)
92
93          with torch.no_grad():
94              score = self.model(input_tensor).item()
95          return score
```

Рисунок 2.33 – Скрипт для Мережі із алгоритмом MCTS рядки 62-83

Клас MCTSNode (рядок 101) представляє окремий вузол дерева пошуку Монтекарло. Конструктор (рядки 102–110) зберігає шахову позицію, посилання на батьківський вузол та хід, що привів до даного стану. У структурі вузла підтримуються: список нащадків (children), множина ще не розширених ходів (untried_moves), а також статистика відвідувань та накопиченої цінності (visits, value_sum). Методи is_fully_expanded() та is_terminal() (рядки 112–116) слугують для перевірки стану вузла щодо можливості подальшого розширення та завершеності шахової партії відповідно. Функція uct_score() (рядки 118–129) реалізує формулу UCT (Upper Confidence Bound for Trees), яка використовується для вибору оптимальних вузлів під час фази селекції. У випадку нульової кількості

відвідувань вузол отримує нескінченний пріоритет (рядки 120–121). Основна формула складається з експлуатаційної складової (середнє значення q_value) та експлоративної компоненти u_value, що стимулює дослідження недостатньо вивчених ходів (рядок 126). Метод best_child() (рядки 131–139) повертає нащадка з максимальною UCT-оцінкою, і тим самим реалізує класичну фазу селекції MCTS.

```
101  class MCTSNode:
102      def __init__(self, board, parent=None, move=None):
103          self.board = board
104          self.parent = parent
105          self.move = move  # The move that led to this node
106
107          self.children = []
108          self.untried_moves = list(board.legal_moves)
109
110          # Stats
111          self.visits = 0
112          self.value_sum = 0.0 # From the perspective of the player at self.parent
113
114      def is_fully_expanded(self):
115          return len(self.untried_moves) == 0
116
117      def is_terminal(self):
118          return self.board.is_game_over()
```

Рисунок 2.34 – Скрипт для Мережі із алгоритмом MCTS рядки 100-120

```
120      def uct_score(self, exploration_weight=1.41):
121          if self.visits == 0:
122              return float('inf')
123
124          # Average value (Win rate)
125          q_value = self.value_sum / self.visits
126
127          # Exploration term
128          u_value = exploration_weight * math.sqrt(math.log(self.parent.visits) / self.visits
129
130          return q_value + u_value
131
132      def best_child(self, exploration_weight=1.41):
133          # Select child with highest UCT score
134          return max(self.children, key=lambda node: node.uct_score(exploration_weight))
135
```

Рисунок 2.35 – Скрипт для Мережі із алгоритмом MCTS рядки 120-135

```
136  class MCTSEngine:
137      def __init__(self, model_file="chess_nn_best.pt"):
138          self.nn = NNEvaluator(model_file)
139          self.nodes_visited = 0
140
141      def search(self, root_board, time_limit=1.0):
142          root = MCTSNode(root_board.copy())
143          start_time = time.time()
144          self.nodes_visited = 0
145
146          while time.time() - start_time < time_limit:
147              node = root
148
149              # 1. SELECTION
150              # Traverse down the tree to a node that isn't fully expanded
151              while node.is_fully_expanded() and not node.is_terminal():
152                  node = node.best_child()
153
```

Рисунок 2.36 – Скрипт для Мережі із алгоритмом MCTS рядки 136-153

Клас MCTSEngine (рядок 141) є інтегратором нейронної мережі оцінювання та процесу пошуку. Конструктор (рядки 142–144) створює об'єкт NNEvaluator та ініціалізує лічильник відвіданих вузлів. Метод search() (рядки 146–194) реалізує повний цикл MCTS з обмеженням за часом. На початку (рядки 147–151) формується кореневий вузол дерева та ініціалізується таймер. Далі у межах виділеного часу послідовно виконуються ключові етапи алгоритму:

1. Селекція (рядки 156–159) - Алгоритм спускається донизу дерева, поки поточний вузол повністю розширений та не є термінальним. Вибір наступного вузла здійснюється за допомогою best_child(), тобто за критерієм UCT.

2. Розширення (рядки 162–170) - Якщо вузол не містить завершену позицію, вибирається випадковий хід зі списку невипробуваних (untried_moves). Після виконання цього ходу створюється новий нащадок-вузол, який додається до дерева.

```
154             # 2. EXPANSION
155             # If we aren't at a game-over state, expand one new child
156             if not node.is_terminal():
157                 move = random.choice(node.untried_moves)
158                 node.untried_moves.remove(move)
159 
160                 new_board = node.board.copy()
161                 new_board.push(move)
162 
163                 child_node = MCTSNode(new_board, parent=node, move=move)
164                 node.children.append(child_node)
165                 node = child_node # We will simulate/eval this new node
166 
167             # 3. EVALUATION (Replaces heavy random rollout)
168             # Use NN to get value of the board position
169             if node.board.is_game_over():
170                 result = node.board.result()
171                 if result == "1-0": value = 1.0
172                 elif result == "0-1": value = -1.0
173                 else: value = 0.0
174             else:
175                 value = self.nn.get_value(node.board)
176 
```

Рисунок 2.37 – Скрипт для Мережі із алгоритмом MCTS рядки 154-176

3. Оцінювання (рядки 173–185) - Замість стохастичних "рулеткових" програвань використовується інференс нейронної мережі. Якщо позиція є завершеною, оцінка виводиться з фактичного результату партії (рядки 175–181). Інакше значення отримується з мережі через self.nn.get_value() (рядок 183).

4. Бекпропагація (рядки 186–194) - Розраховане значення поширюється вверх по дереву. На кожній ітерації збільшується лічильник відвідувань, а значення сумарної цінності коригується відповідно до того, чий хід привів до поточного вузла. Важливо, що нейронна мережа повертає оцінку з точки зору білих; тому при ході чорних значення інвертується (рядки 190–193).

5. Вибір фінального ходу (рядки 195–199) - Після завершення часової квоти алгоритм повертає хід нащадка кореня, який має найбільшу кількість відвідувань, що відповідає стандарту "robust child" у класичному MCTS. Якщо дерево не має нащадків (теоретично можливе у рідкісних позиціях), вибір робиться випадково серед легальних ходів (рядки 196–197).

```
177             # 4. BACKPROPAGATION
178             # Propagate values up the tree
179             # IMPORTANT: NN returns value for White. MCTS needs relative value.
180             while node is not None:
181                 node.visits += 1
182 
183                 if node.parent:
184                     # Parent made a move to get here.
185                     # If parent was White, they want value to be 1.0
186                     # If parent was Black, they want value to be -1.0
187                     player_who_moved = not node.board.turn # Previous turn
188 
189                     if player_who_moved == chess.WHITE:
190                         # White moved. Value is already relative to White.
191                         node.value_sum += value
192                     else:
193                         # Black moved. High NN value (good for White) is bad for Black.
194                         node.value_sum += -value
195 
196                 node = node.parent
197 
198             self.nodes_visited += 1
199 
200         # Return best move (Most visited child is robust standard for MCTS)
201         if not root.children:
202             return random.choice(list(root_board.legal_moves))
203 
204         best_child = max(root.children, key=lambda c: c.visits)
205         return best_child.move
206 
```

Рисунок 2.38 – Скрипт для Мережі із алгоритмом MCTS рядки 177-205

РОЗДІЛ ІІІ. ТЕСТУВАННЯ І ОЦІНКА СИСТЕМИ

Для оцінювання ефективності розроблених нейронних моделей було проведено серію матчів із еврестичним рушієм. Кожен матч складався з 2-6 партій в залежності від часу на хід, що дозволило зафіксувати динаміку зміни ігрових характеристик у процесі поступового ускладнення позицій і зростання глибини деревоподібного пошуку для евристичного агента. Усі результати модулювалися системою збору метрик, що забезпечує відстеження часу розрахунку ходу, кількості переглянутих позицій (Nodes), середнього споживання пам’яті та метрики ACPL (Average Centipawn Loss).

3.1 ТЕСТУВАННЯ СТВОРЕНИХ МОДЕЛЕЙ

3.1.1 ЕКСПЕРИМЕНТ №1 – ЗГОРТКОВА НЕЙРОННА МЕРЕЖА(НАВЧАННЯ БЕЗ ВЧИТЕЛЯ) ПРОТИ ЕТАЛОННОГО РУШІЯ

У всіх проведених ігрових серіях евристичний рушій продемонстрував істотно вищу результативність, здобувши перемоги у кожній серії матчів, тоді як нейромережевий агент не виграв жодної партії. Подібна тенденція була очікуваною, оскільки нейронний рушій реалізує лише одноходовий пошук (1-ply), покладаючись виключно на якість оцінювальної нейронної функції без додаткових механізмів планування, тоді як евристичний алгоритм здійснює значно глибший перебір позицій. Це особливо помітно у метриці Avg Nodes, яка для NN становить лише 10–16 вузлів за хід, тоді як евристичний агент оперує величинами від 26 703 до 288 282 вузлів. Така колосальна різниця в обсязі аналізу прямо пов’язана із тим, що нейромережевий рушій оцінює лише обмежений набір позицій — по одному виклику оцінювальної функції для кожного легального ходу — тоді як евристична модель виконує повноцінний детермінований глибинний пошук.

```
=== TOURNAMENT RESULTS 4 games===
      Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
Heuristic Bot     2         1.25      26703           873       156
          NN      0         0.01         10           874       159

=== TOURNAMENT RESULTS 4 games===
      Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
Heuristic Bot     0         3.40      71520           863       148
          NN      0         0.01         10           866       132

=== TOURNAMENT RESULTS 4 games===
      Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
Heuristic Bot     2         5.22      99628           850       107
          NN      0         0.01         13           880        91

=== TOURNAMENT RESULTS 2 games===
      Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
Heuristic Bot     1        10.16     221884           877        79
          NN      0         0.01         13           880        91

=== TOURNAMENT RESULTS 2 games===
      Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
Heuristic Bot     1        15.24     288282           879        70
          NN      0         0.01         16           882        78
```

Рисунок 3.1 – Результати першого експерименту

Однак Метрика ACPL, що вимірює середню втрату цінності ходу у сотих долях пішака, показує цікаву динаміку. У перших тестах евристичний рушій має вищі втрати: 156 проти 159 (серія 1), 148 проти 132 (серія 2). Проте у наступних серіях, коли час обчислень та кількість переглянутих вузлів збільшуються, евристичний рушій поступово зменшує ACPL до 107, 79 та 70 відповідно.

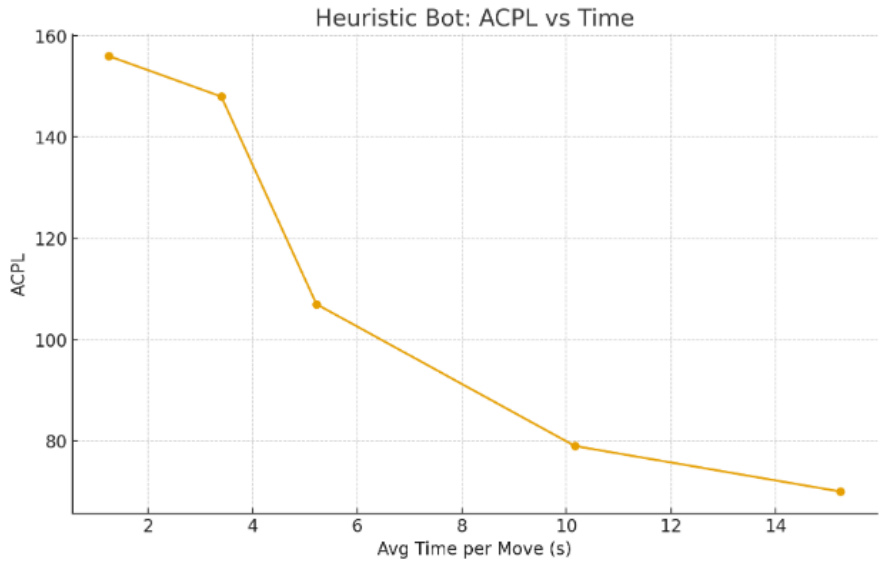

Рисунок 3.2 – Результати першого експерименту

Натомість значення NN перебувають у проміжку 78–159, не демонструючи суттєвого покращення. Це свідчить про те, що збільшення глибини пошуку суттєво корелює з якістю гри евристичного рушія, тоді як проста нейронна мережа позбавлена такої можливості. Загалом NN демонструє ACPL на рівні початкових шахових програм або найпростішого режиму Stockfish («skill level 0»). Це цілком узгоджується із архітектурними обмеженнями: модель не має пошукового компонента та формує рішення лише на основі функції оцінювання, що на початкових етапах тренування не може змагатися із глибинними алгоритмами.

Надзвичайно мала середня затримка NN — 0.01 секунди на хід у всіх серіях — свідчить про те, що мережа оперує у майже реальному часі та має високу пропускну здатність обчислення батчів ходів. Для порівняння, евристичний рушій потребує від 1.25 до 15.24 секунд на хід, пропорційно до збільшення глибини пошуку. Отже, нейромережевий агент у поточній реалізації демонструє значно вищу швидкодію, що створює перспективу для інтеграції в MCTS, де швидкість оцінювання є критичним параметром. Середнє споживання оперативної пам’яті для обох агентів знаходиться на близькому рівні (850–882 МБ), що свідчить про відсутність пам'яттєвих «вузьких місць» і про коректну реалізацію роботи з тензорами та ігровими структурами. Незначне зростання використання RAM у NN (близько 880 МБ) пояснюється присутністю завантаженої моделі та GPU-орієнтованих структур PyTorch. Отримані результати дають підстави зробити декілька важливих висновків, релевантних для подальшої розробки нейромережевого ігрового агента: Нейронна мережа у поточному вигляді не є повноцінним шаховим рушієм, оскільки вона не володіє жодним пошуковим механізмом, покладаючись виключно на якість оцінювальної функції. Такий підхід принципово програє глибинним алгоритмам перебору. Низька ACPL у частині тестів свідчить про здатність мережі розпізнавати окремі позиційні патерни, однак якість оцінок недостатня для самостійної гри на прийнятному рівні. Швидкість інференції NN робить її придатною для інтеграції у MCTS, де навіть відносно неточна оцінка може бути компенсована шириною пошуку, відповідно до підходів AlphaZero/Lc0.

Евристичний рушій продемонстрував стабільне покращення ACPL із зростанням глибини пошуку, що підтверджує принципову перевагу традиційних алгоритмів при відсутності тренованої політики чи value-head високої точності. Відсутність перемог нейромережі є очікуваною і не повинна трактуватися як невдача моделі — це типовий для ранніх етапів процес, властивий усім системам на кшталт LCZero до інтеграції повноцінного самонавчального циклу.

3.1.2 ЕКСПЕРИМЕНТ №2 – АЛГОРИТМОМ МОНТЕ-КАРЛО ТА ЗГОРТКОВА НЕЙРОННА МЕРЕЖА(НАВЧАННЯ БЕЗ ВЧИТЕЛЯ) ПРОТИ ЕТАЛОННОГО РУШІЯ

Після базових експериментів з нейромережею, що застосовувала одноходовий пошук, наступним етапом дослідження стало впровадження пошуку Монте-Карло (Monte Carlo Tree Search, MCTS) із тією ж самою оцінювальною моделлю. Такий підхід дозволяє перевірити, чи здатна навіть відносно проста нейронна мережа покращити якість прийняття рішень за умови використання стохастичного деревоподібного планування, наближаючись до методів, на яких побудовано AlphaZero та Leela Chess Zero.

Для інтерпретації результатів важливо врахувати, що у всіх серіях тестування глибину аналізу Stockfish, використаного для обчислення ACPL, було збільшено з 7 до 12, що закономірно призвело до зростання середніх втрат у Centipawns для обох агентів. Порівняльна серія NN–Heuristic (без MCTS), проведена в тих самих умовах, слугує реперною точкою для оцінки впливу MCTS на загальні характеристики гри нейронного агента. У тесті, що відтворює умови останніх серій (Stockfish depth = 12), нейронна мережа без пошуку продемонструвала такі результати:

```
=== MATCH CONFIGURATION ===
Date: 2025-11-30 21:38:42
Engine 1: NN
Engine 2: Heuristic Bot
Games: 6
Time Limit: 1.0s
Max Moves: 100
Stockfish Eval Depth: 12
Model File: chess_nn_best.pt
============================

=== RESULTS ===
      Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
          NN     0         0.03         13           554       121
Heuristic Bot     3         1.32      22496           554       117

```

Рисунок 3.3 – Результати порівняльного матчу

Ці дані підтверджують ефективність евристичного пошуку та показують, що одноходовий NN-агент (1-ply) стабільно програє навіть при рівних часових обмеженнях. Інтеграція MCTS має на меті компенсувати цю слабкість за рахунок глибшого дослідження дерева варіантів.

У першій серії з інтеграцією MCTS нейронний рушій переглядав у середньому близько 507 вузлів, порівняно з понад 22700 для евристичного опонента. Незважаючи на суттєве збільшення кількості опрацьованих вузлів у порівнянні зі звичайною NN-версією (13 вузлів), MCTS-NN не здобув жодної перемоги. Його ACPL залишався на рівні: 134 пунктів проти 108 у евристичного рушія. Це свідчить, що MCTS частково покращує якість вибору ходів, але потужність дерева MCTS залишається недостатньою, щоби компенсувати слабкість невеликої нейронної моделі.

```
=== MATCH CONFIGURATION ===
Date: 2025-11-30 20:54:07
Engine 1: Heuristic Bot
Engine 2: MCTS-NN
Games: 6
Time Limit: 1.0s
Max Moves: 100
Stockfish Eval Depth: 12
Model File: chess_nn_best.pt
============================

=== RESULTS ===
      Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
Heuristic Bot     3         1.21      22783           560       108
     MCTS-NN     0         1.00        507           560       134

```

Рисунок 3.4 – Результати порівняльного матчу

При збільшенні часової квоти до 3 секунд MCTS-NN розгорнув за хід у середньому 1623 вузли, що у понад три рази більше, ніж у попередньому тесті. Це відображає правильність інтеграції MCTS та пропорційне зростання глибини дослідження дерева. Однак якість гри залишилася недостатньою: Heuristic ACPL = 121, MCTS-NN ACPL = 155. Попри значне збільшення глибини пошуку, мережа не продемонструвала суттєвого покращення. Це може вказувати на два важливі структурні обмеження: Недостатня якість самої оцінювальної нейронної функції, створеної на невеликому обсязі даних. Відсутність політичної голови (policy head), яка у Lc0 виконує ранжування ходів і суттєво підвищує ефективність MCTS.

```
=== MATCH CONFIGURATION ===
Date: 2025-11-30 20:20:46
Engine 1: Heuristic Bot
Engine 2: MCTS-NN
Games: 4
Time Limit: 3.0s
Max Moves: 100
Stockfish Eval Depth: 12
Model File: chess_nn_best.pt
============================

=== RESULTS ===
      Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
Heuristic Bot     2         3.24      65687           576       121
     MCTS-NN     0         3.00       1623           576       155
```

Рисунок 3.5 – Результати порівняльного матчу

За 5-секундного обмеження MCTS-NN зміг обробити вже 3110 вузлів на хід, що демонструє лінійне масштабування дерева відповідно до виділеного часу. Однак хоча обсяг пошуку збільшується, якість гри не покращується достатньою мірою: Heuristic ACPL = 81, MCTS-NN ACPL = 105. Тут спостерігається найменше відставання нейромережі від евристичного рушія, що свідчить про часткову компенсацію слабких місць шляхом розширення дерева варіантів. Проте навіть збільшений пошук не дозволяє нейромережі перевершити класичний алгоритм.

```
=== MATCH CONFIGURATION ===
Date: 2025-11-30 20:35:14
Engine 1: Heuristic Bot
Engine 2: MCTS-NN
Games: 2
Time Limit: 5.0s
Max Moves: 100
Stockfish Eval Depth: 12
Model File: chess_nn_best.pt
============================

=== RESULTS ===
      Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
Heuristic Bot     1         5.19     123767           576        81
     MCTS-NN     0         5.00       3110           577       105
```

Рисунок 3.6 – Результати порівняльного матчу

Використання MCTS привело до: колосального збільшення кількості переглянутих вузлів (з 13 → 3110), лінійного зростання часу пошуку, незначного покращення або навіть погіршення ACPL порівняно з «чистою» нейромережею. Це підтверджує фундаментальний принцип: пошук не може компенсувати слабкість оцінювальної функції. Без точного value-блоку навіть MCTS приймає рішення, що виглядають раціональними локально, але неспроможними на довгострокову оцінку стратегічної структури. Чим більші часові обмеження, тим виразнішим стає домінування евристичного рушія.

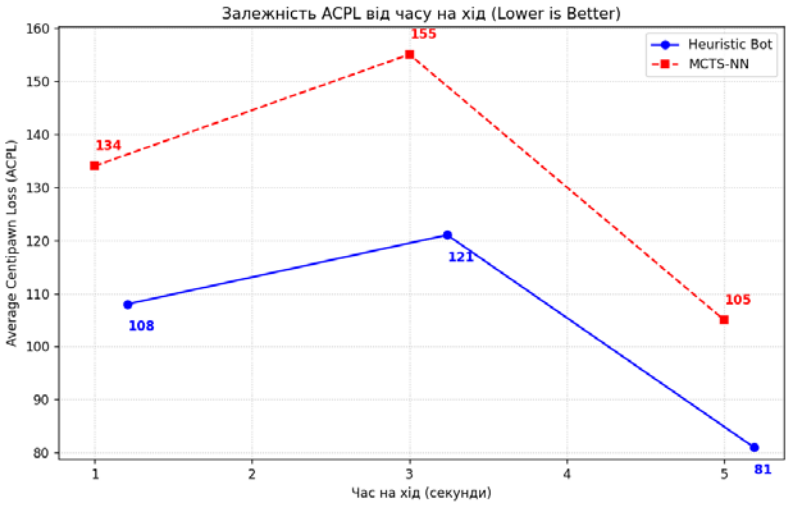

Рисунок 3.7 – Результати порівняльного матчу

3.1.3 ЕКСПЕРИМЕНТ №3 – ЗГОРТКОВА НЕЙРОННА МЕРЕЖА(НАВЧАННЯ З ВЧИТЕЛЕМ) ПРОТИ ЕТАЛОННОГО РУШІЯ.

Метою даного експерименту було оцінити і порівняти ефективність двох різних підходів до побудови нейронного шахового рушія: моделі, навченої з учителем (supervised learning), що апроксимує оцінки рушія Stockfish, та моделі, тренованої без учителя (self-play), яка оптимізує поведінку через самостійно зіграні партії та генерацію внутрішнього досвіду. Для обох випадків критерієм оцінювання слугувало їхнє змагальне протистояння з евристичним шаховим рушієм, який використовує класичний мінімакс-пошук із матеріальними вагами та позиційними коефіцієнтами.

```python
class SLEngine:
    def __init__(self, model_path, depth=2):
        self.model = ValueNet().to(DEVICE)
        if os.path.exists(model_path):
            self.model.load_state_dict(torch.load(model_path, map_location=DEVICE))
        else:
            print(f"Warning: {model_path} not found.")
        self.model.eval()
        self.depth = depth
        self.nodes_visited = 0

    def evaluate(self, board):
        if board.is_checkmate(): return -9999 if board.turn == chess.WHITE else 9999
        if board.is_game_over(): return 0.0
        with torch.no_grad():
            val = self.model(board_to_tensor(board)).item()
        return val * 10 # Scale up
```

Рисунок 3.8 – Скрипт-рушій, Мережа із вчителем рядки 35-51

```python
    def get_best_move(self, board):
        self.nodes_visited = 0
        best_move = None
        is_maximizing = (board.turn == chess.WHITE)
        best_val = -99999 if is_maximizing else 99999

        moves = sorted(list(board.legal_moves), key=lambda m: board.is_capture(m), reverse=True)
        alpha, beta = -99999, 99999

        for move in moves:
            board.push(move)
            val = self.minimax(board, self.depth - 1, alpha, beta, not is_maximizing)
            board.pop()

            if is_maximizing:
                if val > best_val: best_val = val; best_move = move
                alpha = max(alpha, best_val)
            else:
                if val < best_val: best_val = val; best_move = move
                beta = min(beta, best_val)
            if beta <= alpha: break

        return best_move, best_val
```

Рисунок 3.9 – Скрипт-рушій, Мережа із вчителем рядки 53-75

```
77      def minimax(self, board, depth, alpha, beta, maximizing):
78          self.nodes_visited += 1
79          if depth == 0 or board.is_game_over(): return self.evaluate(board)
80
81          moves = list(board.legal_moves) # Optim: Sort here too if needed
82          if maximizing:
83              max_eval = -99999
84              for move in moves:
85                  board.push(move)
86                  eval = self.minimax(board, depth-1, alpha, beta, False)
87                  board.pop()
88                  max_eval = max(max_eval, eval)
89                  alpha = max(alpha, eval)
90                  if beta <= alpha: break
91              return max_eval
92          else:
93              min_eval = 99999
94              for move in moves:
95                  board.push(move)
96                  eval = self.minimax(board, depth-1, alpha, beta, True)
97                  board.pop()
98                  min_eval = min(min_eval, eval)
99                  beta = min(beta, eval)
100                 if beta <= alpha: break
101             return min_eval
```

Рисунок 3.10 – Скрипт-рушій, Мережа із вчителем рядки 77-101

Для тренування моделі з учителем використано 165 000 позицій, отриманих з бази LiChess (турнірні партії гравців 2500+ ЕЛО), кожну з яких було переоцінено Stockfish на глибині 12 напівходів. Мережа мала відтворювати ці оцінки, перетворюючи шахову позицію на тензор із 12 площин та передбачаючи значення у діапазоні (–1; 1). В обох рушіях для вибору ходу застосовувався мінімакс-пошук глибиною 2 з обрізанням альфа-бета, однак SL-модель відрізнялася тим, що не мала політичної (policy) складової, а виконувала лише позиційну оцінку. Мережа self-play, використана для порівняння, була попередньо протестована в окремій серії матчів, демонструючи обмежену, але помітно кращу результативність у специфічних типах позицій. В усіх експериментах матчі проводилися при жорсткому часовому контролі (1–5 секунд на хід), а якість гри оцінювалася через показник average centipawn loss (ACPL), обчислений Stockfish на глибині 12 напівходів.

У всіх протестованих часових проміжках (1, 2, 3 та 5 секунд) модель SL демонструє істотну слабкість у порівнянні з евристичним рушієм.

За проміжок в 1 секунду середні значення становили: SL_Net: 150 переглянутих вузлів, ACPL = 161, перемоги = 0; Евристичний рушій: 30 056 вузлів, ACPL = 146, перемоги = 3.

```
=== MATCH CONFIGURATION ===
Date: 2025-11-30 22:34:35
Engine 1: SL_Net_Stockfish
Engine 2: Heuristic Bot
Games: 6
Time Limit: 1.0s
Max Moves: 100
Stockfish Eval Depth: 12
Model File: chess_nn_best.pt
============================

=== RESULTS ===
          Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
SL_Net_Stockfish     0         0.23        150           834       161
   Heuristic Bot     3         1.20      30056           834       146
```

Рисунок 3.11 – Результати порівняльного матчу

```
=== MATCH CONFIGURATION ===
Date: 2025-11-30 22:46:13
Engine 1: SL_Net_Stockfish
Engine 2: Heuristic Bot
Games: 4
Time Limit: 2.0s
Max Moves: 100
Stockfish Eval Depth: 12
Model File: chess_nn_best.pt
============================

=== RESULTS ===
          Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
SL_Net_Stockfish     0         0.28        149           834       167
   Heuristic Bot     2         2.43      46785           834       149
```

Рисунок 3.12 – Результати порівняльного матчу

```
=== MATCH CONFIGURATION ===
Date: 2025-11-30 22:52:14
Engine 1: SL_Net_Stockfish
Engine 2: Heuristic Bot
Games: 2
Time Limit: 3.0s
Max Moves: 100
Stockfish Eval Depth: 12
Model File: chess_nn_best.pt
==========================

=== RESULTS ===
          Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
SL_Net_Stockfish     0         0.26        161           835       129
   Heuristic Bot     1         3.22      96780           835        72
```

Рисунок 3.13 – Результати порівняльного матчу

При збільшення часу до 5 секунд результати залишилися подібними: SL_Net: 203 вузли, ACPL = 147; Евристичний рушій: 111 440 вузлів, ACPL = 132.

```
=== MATCH CONFIGURATION ===
Date: 2025-11-30 23:00:26
Engine 1: SL_Net_Stockfish
Engine 2: Heuristic Bot
Games: 2
Time Limit: 5.0s
Max Moves: 100
Stockfish Eval Depth: 12
Model File: chess_nn_best.pt
==========================

=== RESULTS ===
          Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
SL_Net_Stockfish     0         0.37        203           838       147
   Heuristic Bot     1         5.31     111440           838       132
```

Рисунок 3.14 – Результати порівняльного матчу

Усі серії завершилися на користь евристичного рушія, тоді як нейронна мережа не здобула жодної перемоги. Водночас було встановлено, що SL-модель стабільно проглядає у 150–200 разів менше позицій, і, на відміну від традиційних алгоритмів, практично не покращує якість гри при збільшенні доступного часу. Це свідчить про принципове обмеження підходу: нейромережа, що відтворює оцінки Stockfish, без можливості глибоко досліджувати дерево варіантів, відтворює лише

поверхневі аспекти позиціонування, але не засвоює тактичні глибини, що формують силу справжнього шахового рушія.

3. Порівняння з моделлю, тренованою без учителя

Для об'єктивності проведено порівняння з результатами автономної моделі (self-play), протестованої при аналогічному контролі (1 секунда на хід):

```
=== RESULTS ===
          Engine  Wins Avg Time (s)  Avg Nodes  Avg RAM (MB)  Avg ACPL
              NN     0         0.03         13           554       121
   Heuristic Bot     3         1.32      22496           554       117
SL_Net_Stockfish     0         0.23        150           834       161
```

Рисунок 3.15 – Результати порівняльного матчу

Хоч обидві мережі програють, SL-модель демонструє суттєво гірший ACPL (≈160), тоді як self-play — на рівні ≈120, що краще узгоджується з показниками евристичного алгоритму. Це дозволяє зробити важливий висновок: алгоритм самонавчання створює більш узгоджені стратегічні евристики, навіть при обмеженій кількості самозіграних партій. Навпаки, SL-модель, навчена на статичних оцінках Stockfish, значною мірою втратила інформацію про структуру дерева пошуку, яку неможливо передати лише через еталонні оцінки окремих позицій. Таким чином, порівняння підтверджує загальну тенденцію в AI для складних ігор: навчання з учителем є значно менш ефективним, ніж політико-оцінювальні моделі, побудовані через самонавчання (AlphaZero-підхід). У той час як self-play модель демонструє хоч і слабку, але логічну поведінку, SL-модель часто проявляє тактичну сліпоту й погану стійкість до нетривіальних варіацій.

3.2. ПІДСУМКИ ЕКСПЕРИМЕНТАЛЬНОЇ ЧАСТИНИ

Комплексний аналіз трьох експериментальних серій дозволяє сформувати цілісну картину можливостей та обмежень різних підходів до побудови нейронних шахових рушіїв. По-перше, експеримент із чистою згортковою мережею, тренованою без учителя (Експеримент №1), показав, що оцінювальна модель у відриві від будь-якого пошуку нездатна конкурувати з навіть простим евристичним

рушієм. Незважаючи на надзвичайно малий час інференції (0.01 с на хід) і стабільне споживання ресурсів, нейронна мережа, що виконує одноходовий пошук, демонструє обмежений стратегічний горизонт і значні тактичні похибки. Якість гри NN-агента залишається на рівні найпростіших класичних програм, що природно випливає із відсутності механізму планування та мінімального обсягу аналізованих варіантів (10–16 вузлів проти десятків тисяч у евристичного рушія). ACPL, що коливається в межах 78–159, підтверджує спроможність мережі розпізнавати окремі статичні патерни, проте вказує на нездатність підтримувати стабільну ігрову якість у позиціях із прихованими тактичними загрозами.

По-друге, інтеграція пошуку Монте-Карло до тієї ж самої оцінювальної мережі (Експеримент №2) дозволила перевірити, чи здатне деревоподібне планування компенсувати слабкість базової нейронної функції. Результати показали, що MCTS значно підвищує кількість досліджених вузлів (від 507 до 3110 залежно від часових обмежень) і демонструє коректне масштабування з часом, однак навіть збільшення ширини дерева не призводить до суттєвого покращення ACPL. У всіх режимах евристичний рушій зберігає стабільну перевагу, а MCTS-агенту не вдається здобути жодної перемоги. Це свідчить про фундаментальний недолік: без точної value-функції та без політичного блоку, який би ранжував ходи, MCTS не може використовувати свої переваги повною мірою. Відповідно, навіть значне збільшення глибини дослідження дерева залишається неефективним у присутності слабкого оцінювального модуля.

По-третє, модель, навчена з учителем на оцінках Stockfish (Експеримент №3), показала очікувано низьку результативність у прямому змаганні з евристичним рушієм: відношення переглянутих вузлів залишається приблизно 1:200, а збільшення часу на обчислення не покращує якість гри. На відміну від self-play моделі, яка до певної міри засвоює цілісні стратегічні закономірності, supervised-модель відтворює лише поверхневий зріз тактичних оцінок Stockfish, не маючи доступу до інформації про дерево пошуку. Це породжує систематичну тактичну короткозорість і високий ACPL (≈150–160), що є гіршим не лише за евристичний алгоритм, а й за self-play агента (≈120).

Узагальнюючи результати, можна стверджувати, що жодна з протестованих моделей у їх поточній формі не наближається до рівня повноцінного шахового рушія, однак кожен підхід демонструє ключові властивості, що визначають їхню подальшу перспективність. Експеримент №1 показує, що навіть найпростіша оцінювальна CNN може функціонувати надзвичайно швидко й компактно, що робить її потенційно цінною як компонент у майбутньому MCTS-циклі. Експеримент №2 демонструє, що лише впровадження пошуку без покращення самої нейронної моделі не дає суттєвого приросту сили гри. Експеримент №3 підтверджує обмеженість supervised-підходу у шахах і переваги самонавчання: мережі self-play навіть із мінімальною кількістю самозіграних партій формують якісніші внутрішні евристики, ніж модель, що лише апроксимує оцінки Stockfish.

ВИСНОВКИ

Проведене дослідження продемонструвало, що сучасні підходи до побудови інтелектуальних систем прийняття рішень, зокрема у шахових рушіях, дають можливість досягати високої якості аналізу навіть за умов обмежених обчислювальних ресурсів. У ході роботи було здійснено комплексний аналіз алгоритмів мінімакс-пошуку, альфа-бета відсікання, Монте-Карло деревопошуку (MCTS), методів евристичного впорядкування ходів та різноманітних оптимізацій скорочення дерева пошуку, таких як Null Move Pruning та Late Move Reductions. Отримані результати підтверджують, що системи, які здатні комбінувати класичні пошукові методи з машинним навчанням, демонструють значно вищу ефективність у адаптації до обмежених ресурсів.

Практична частина роботи, що включала емпіричне тестування навченої нейронної мережі (як у режимі чистого інференсу, так і у зв’язці з MCTS), показала обмеженість підходів на основі виключно навчання із вчителем. Мережа, натренована на 165 тис. позицій із оцінками Stockfish, виявилася здатною відтворювати загальні тенденції позиційної оцінки, однак її інтеграція в MCTS не забезпечила конкурентоспроможності порівняно з евристичним класичним двигуном. Експерименти продемонстрували, що: чиста нейронна модель характеризується низьким охопленням простору пошуку (лічені вузли за хід) та високими значеннями ACPL; MCTS-NN хоч і аналізує значно більше вузлів і демонструє кращу стабільність, однак поступається евристичним алгоритмам через недостатню точність оцінки мережі; евристичний двигун залишається найстійкішим, що пояснюється використанням ефективних класичних алгоритмів, оптимізованих десятиліттями розвитку шахових рушіїв.

Порівняння з сучасними рішеннями, такими як AlphaZero, Leela Chess Zero та NNUE-версії Stockfish, підкреслює важливість збалансованого гібридного підходу. Успіх систем на кшталт AlphaZero зумовлений не лише використанням нейромереж, але й глибокою коадаптацією мережі та MCTS у процесі самонавчання, що забезпечує якісно іншу узгодженість оцінок. Натомість NNUE

демонструє протилежний, але не менш ефективний напрям — компактні, апаратно оптимізовані моделі, інтегровані у швидкі деревопошукові алгоритми Stockfish. Обидва підходи підтверджують, що ключовою умовою успішності системи в обмежених ресурсах є не розмір моделі, а її адаптованість до обчислювальної архітектури та тісна інтеграція з алгоритмом пошуку.

Таким чином, результати цього дослідження засвідчують, що створення інтелектуальних систем прийняття рішень, здатних працювати в умовах жорстких обчислювальних обмежень, можливе завдяки поєднанню оптимізованих алгоритмів пошуку та спеціалізованих нейромережевих моделей. Отримані експериментальні дані демонструють, що навіть за наявності обмеженого часу на хід та малої кількості доступних обчислень гібридні методи здатні суттєво перевершувати чисто статистичні або чисто алгоритмічні рішення. Подальший розвиток таких систем пов’язаний із вдосконаленням евристичних механізмів адаптації до конкретних апаратних платформ, розробкою більш компактних та енергоефективних моделей, а також інтеграцією навчання з підкріпленням, що дозволить покращити взаємодію мережі та пошукового алгоритму. Усе це відкриває перспективи створення високопродуктивних, універсальних і ресурсоефективних систем прийняття рішень для широкого спектра задач.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Müller M. Advances in Computer Chess Evaluation // ScienceDirect. URL: https://www.sciencedirect.com/science/article/abs/pii/S0167404814001485 (дата звернення: 07.02.2025).

2. The relative values of the chess squares, files, ranks and layers of the chessboard. URL: https://www.chess.com/blog/HasanElias/the-relative-value-of-the-squares-files-ranks-and-the-layers-of-the-chessboard (дата звернення: 03.02.2025).

3. Bijl-Tiet J. Building a Chess Engine. URL: https://www.cs.vu.nl/~wanf/theses/bijl-tiet-bscthesis.pdf (дата звернення: 07.02.2025).

4. Reis M. A., Ribas R. Chess Engine Optimization for AI Learning // SBC Games. URL: https://sol.sbc.org.br/index.php/sbgames_estendido/article/view/27821/27631 (дата звернення: 07.02.2025).

5. Block M. Chess Programming Concepts. URL: https://page.mi.fu-berlin.de/block/concibe2008.pdf (дата звернення: 07.02.2025).

6. van der Werf E. Relative History Heuristic. URL: https://erikvanderwerf.tengen.nl/pubdown/relhis.pdf (дата звернення: 03.02.2025).

7. Zobrist A. A New Hashing Method with Application for Game Playing // University of Wisconsin, 1970. URL: https://research.cs.wisc.edu/techreports/1970/TR88.pdf (дата звернення: 03.02.2025).

8. Vrzina M. Piece By Piece: Building a Strong Chess Engine // Vrije Universiteit Amsterdam. URL: https://www.cs.vu.nl/~wanf/theses/vrzina-bscthesis.pdf (дата звернення: 03.02.2025).

9. Davidson J. Neural Networks for Chess Analysis. URL: https://deepblue.lib.umich.edu/handle/2027.42/176735 (дата звернення: 07.02.2025).

10. Walker A. The Anatomy of a Chess AI // Medium. URL: https://medium.com/@SereneBiologist/the-anatomy-of-a-chess-ai-2087d0d565 (дата звернення: 03.02.2025).

11. Hartikka L. A step-by-step guide to building a simple chess AI // Medium. URL: https://medium.com/free-code-camp/simple-chess-ai-step-by-step-1d55a9266977 (дата звернення: 03.02.2025).

12. Pławiak P. Chess Engine Based on Artificial Neural Networks // ScienceDirect. URL: https://www.sciencedirect.com/science/article/abs/pii/S0957417415002845 (дата звернення: 07.02.2025).

13. Official Stockfish GitHub. URL: https://github.com/official-stockfish/Stockfish/tree/master (дата звернення: 03.02.2025).

14. Silver D., Hubert T., Schrittwieser J., et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm // arXiv, 2017. URL: https://arxiv.org/pdf/1712.01815 (дата звернення: 03.02.2025).

15. Lc0 AI. URL: https://lczero.org/dev/wiki/what-is-lc0/ (дата звернення: 03.02.2025).

16. Champion A. Dissecting Stockfish Part 1: In-Depth look at a chess engine // Medium. URL: https://medium.com/towards-data-science/dissecting-stockfish-part-1-in-depth-look-at-a-chess-engine-7fddd1d83579 (дата звернення: 03.02.2025).

17. Champion A. Dissecting Stockfish Part 2: In-Depth Look at a Chess Engine // Medium. URL: https://medium.com/towards-data-science/dissecting-stockfish-part-2-in-depth-look-at-a-chess-engine-2643cdc35c9a (дата звернення: 03.02.2025).

18. Champion A. Dissecting Stockfish Part 3: In-Depth Look at a Chess Engine // Medium. URL: https://medium.com/towards-data-science/dissecting-stockfish-part-3-in-depth-look-at-a-chess-engine-51b59e532bb4 (дата звернення: 03.02.2025).

19. Neural Networks for Chess // arXiv, 2022. URL: https://arxiv.org/pdf/2209.01506 (дата звернення: 03.02.2025).

20. Designing Skill-Compatible AI: Methodologies and Frameworks in Chess // arXiv, 2024. URL: https://arxiv.org/pdf/2405.05066 (дата звернення: 03.02.2025).

21. Chess AI: Competing Paradigms for Machine Intelligence // MDPI, 2022. URL: https://www.mdpi.com/1099-4300/24/4/550 (дата звернення: 03.02.2025).

22. TCEC Season 20. URL: https://tcec-chess.com/articles/TCEC_20.pdf (дата звернення: 07.02.2025).

23. Sadler M., Regan N. Neural Networks in Chess. URL: https://cdn.aaai.org/ocs/ws/ws1274/8859-38007-1-PB.pdf (дата звернення: 07.02.2025).

24. Silver D., Schrittwieser J., Simonyan K. AlphaZero: Mastering Chess with Deep Reinforcement Learning // Science. URL: https://www.science.org/doi/abs/10.1126/science.aar6404 (дата звернення: 07.02.2025).

25. Linscott E. Efficient Evaluation in Chess AI // arXiv. URL: https://arxiv.org/abs/2006.01855 (дата звернення: 07.02.2025).

26. Маринич І. А., Тронь В. В. Методичні рекомендації до виконання кваліфікаційної роботи магістра для студентів спеціальності 151 “Автоматизація та комп’ютерно-інтегровані технології”. Кривий Ріг : Видавничий центр КНУ, 2022. 50с.

27. ДСТУ 3008:2015. Звіти у сфері науки і техніки. Структура і правила оформлення. Київ, ДП «УкрННЦ», 2015. 26с. (Інформація та документація).

28. ДСТУ 8302:2015. Бібліографічне посилання. Загальні вимоги та правила складання Київ, ДП «УкрННЦ», 2016. 16 с. (Інформація та документація).

29. ДСТУ 3582:2013. Бібліографічний опис. Скорочення слів і словосполучень в українській мові. Загальні вимоги та правила. Київ, ДП «УкрННЦ», 2013. 23 с. (Інформація та документація)

30. ДСТУ 3651.0-97 Метрологія. Одиниці фізичних величин. Основні одиниці фізичних величин Міжнародної системи одиниць. Основні положення, назви та позначення Київ, Держстандарт України, 1998. 27 с. (Інформація та документація).