

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Інститут інформатики та радіoeлектроніки,

Факультет комп'ютерних наук і технологій

(повне найменування інституту, назва факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

бакалавр

(ступінь вищої освіти)

на тему

ПРОГРАМНА РЕАЛІЗАЦІЯ ШАХОВОГО РУШІЯ З ВИКОРИСТАННЯМ
ТЕХНОЛОГІЇ ГЛИБОКОГО НАВЧАННЯ НЕЙРОМЕРЕЖЕВИХ МОДЕЛЕЙ
SOFTWARE IMPLEMENTATION OF A CHESS ENGINE USING DEEP
LEARNING TECHNOLOGY OF NEURAL NETWORK MODELS

Виконав: студент 4 курсу, групи КНТ-117

Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

Басанець М.І.

(прізвище та ініціали)

Керівник Олійник. А.О.

(прізвище та ініціали)

Рецензент Скрупський С.Ю.

(прізвище та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Інститут, факультет ФКНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
 (КОД І НАЙМЕНУВАННЯ)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
С.О. Субботін
 “ ” 20__ року

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

Басанця Михайла Івановича

(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Програмна реалізація шахового рушія з використанням технології глибокого навчання нейромережових моделей.

Software implementation of a Chess Engine Using Deep Learning Technology of Neural Network Models

керівник проєкту (роботи) Олійник Андрій Олександрович, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від “ 30 ” березня 2021 року № 103

2. Строк подання студентом проєкту (роботи) 10 травня 2021 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Розробка архітектури програми. 3. Основні рішення щодо реалізації компонентів системи. 4. Експлуатація, тестування та експериментальне дослідження програми. Висновки. Перелік джерел посилання. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	Олійник А.О., доцент		
Нормоконтролер	Андрєєв М.О., асистент		

7. Дата видачі завдання “ 28 ” лютого 2021 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області	2-3 тижні	Розділ 1
3	Розробка архітектури програми	4 тиждень	Розділ 2
4	Розробка програми	6-8 тижні	Розділ 3
5	Тестування та експериментальне дослідження програми	8 тиждень	Розділ 4
6	Оформлення пояснювальної записки та документів до неї. Нормоконтроль та рецензування	9 тиждень	Додатки
7	Захист роботи	10 тиждень	

Студент

(підпис) Басанець М.І.
(прізвище та ініціали)

Керівник проєкту (роботи)

(підпис) Олійник А.О.
(прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 188 с., 46 рис., 7 табл., 3 дод., 44 джерел.

БАГАТОШАРОВИЙ ПЕРСЕПТРОН, ГЛИБИННЕ НАВЧАННЯ, ШАХОВИЙ РУШІЙ, ШТУЧНИЙ ІНТЕЛЕКТ, ШТУЧНІ НЕЙРОННІ МЕРЕЖІ.

Об'єкт дослідження – процес реалізації шахових рухів.

Предмет дослідження – методи реалізації шахових рухів та застосування штучних нейронних мереж в шахових рушіях.

Мета роботи – програмна реалізація шахового рушія з використанням нейронної мережі.

Матеріали, методи та технічні засоби: структурне та об'єктно-орієнтоване програмування, мова програмування C++, мова програмування Python, фреймворк для глибокого навчання Tensorflow.

Результати. Створено програмний застосунок, який здатний аналізувати шахову позицію та вказувати найліпший хід у цій позиції за результатами аналізу за допомогою штучної нейронної мережі.

Висновки. Розроблено програмний застосунок для аналізу шахових позицій. Порівняно статичний аналіз позиції та аналіз позиції з використанням нейронної мережі.

Галузь використання – аналіз шахових позицій з використанням рушія або додавання шахового штучного інтелекту до власного додатку.

ABSTRACT

Explanatory note to the final qualifying work of the bachelor: 188 pages, 46 figures, 7 tables, 3 appendixes, 44 sources.

ARTIFICIAL INTELLIGENCE, ARTIFICIAL NEURAL NETWORKS, CHESS ENGINE, DEEP LEARNING, MULTILAYER PERCEPTRON.

Object of study is process of software implementation of chess engines.

Subject of study is methods of implementation of chess engines and usage of neural networks inside a chess engine.

The purpose of the work is implementation of a chess engine with a neural network.

Materials, methods and tools: structured and object-oriented programming, programming language C++, programming language Python, deep learning framework Tensorflow.

Results. Software for analysis of a chess position and evaluating the best move in the position using a neural network is created.

Conclusions. Software for chess position analysis is created. Static and neural network evaluations are compared.

The field of use is analysis of chess positions using an engine for educational purposes or adding a chess artificial intelligence to a chess app.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	7
Вступ.....	8
1 Аналіз предметної області.....	9
1.1 Основні поняття.....	9
1.2 Огляд існуючих методів реалізації шахових рушіїв	12
1.3 Порівняння існуючих методів реалізації шахових рушіїв	45
1.4 Постановка задачі до роботи	47
1.5 Висновок	48
2 Опис програми	49
2.1 Вибір технологій та мов програмування.....	49
2.2 Вибір датасету та його обробка.....	51
2.3 Архітектура компонентів системи	54
2.4 Висновки з розділу	59
3 Розробка програми.....	60
3.1 Розробка шахового рушія	60
3.2 Навчання нейронної мережі.....	70
3.3 Висновки з розділу	73
4 Експлуатація, тестування та експериментальне дослідження програми	74
4.1 Інструкція по експлуатації програми	74
4.2 Аналіз отриманих результатів	79
4.3 Практична цінність роботи	85
4.4 Висновки з розділу	85
Висновки.....	87
Перелік джерел посилання	89
Додаток А Технічне завдання.....	93
Додаток Б Текст програми.....	98
Додаток В Слайди презентації	180

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

CLI	– Command Line Interface;
FEN	– Forsyth-Edwards Notation;
NNUE	– Efficiently Updatable Neural Networks;
БП	– багатошаровий персептрон;
ПЗ	– програмне забезпечення;
ШНМ	– штучна нейронна мережа.

ВСТУП

В 2020-2021 роках через карантинні обмеження стрімко зросла зацікавленість широких мас у шахах та шахових сервісах. Зросла популярність як онлайн-платформ для ігор в шахи, так і перегляди стрімів та відео на шахову тематику.

У зв'язку з підвищенням популярності шахів зріс попит на програмне забезпечення для навчання людей базовим концептам в шахах та аналізу шахових позицій для підвищення рівню гри.

В той же час в останні роки стрімко розвиваються технології машинного та глибокого навчання. Різні види нейромережевих моделей дуже добре себе показують у задачах, в яких важко побудувати чіткий алгоритм дій для досягнення певного результату, проте існує велика кількість емпіричних даних.

Актуальність роботи полягає в дослідженні предметної області програмного забезпечення для шахів та аналіз методів реалізації окремих компонентів шахового рушія. Частина роботи призначена дослідженню застосуванню нейронних мереж в шахових рушіях, способам подання шахової позиції у вигляді вектору ознак для моделей глибокого навчання та порівнянню класичних методів оцінки позиції з методами, що використовують нейронні мережі.

У випускній кваліфікаційній роботі виконується спроба розробити шаховий рушій з класичним методом оцінки позиції та з методом, що використовує нейромережеву модель, та аналізуються отримані результати і способи їх покращення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

В даному розділі наведено аналіз предметної області, розбір основних понять предметної області, огляд існуючих методів реалізації шахових рушіїв та їх порівняння, існуючі шахові рушії та поставлено завдання до роботи.

1.1 Основні поняття

Шахове програмування – розділ програмування, що вивчає створення програмного забезпечення для виконання різноманітних маніпуляцій над шахами від найпростішого відображення шахової позиції до аналізу позицій, пошуку найкращих ходів та безпосередньо гри у шахи [1].

Шаховий рушій – комп'ютерна програма для представлення позиції, генерації можливих ходів у позиції, оцінки позиції та пошуку найкращих ходів та варіацій ходів. Шаховий рушій зазвичай не має користувацького інтерфейсу та використовується іншими програмами в якості програми з інтерфейсом командного рядку або в якості попередньо скомпільованої бібліотеки [2].

Представлення дошки – структура даних або їх сукупність для підтримки положення фігур на шаховій дошці та іншої інформації про гру (можливість рокірування, можливість взяття на проході, кількість ходів, повторів поточної позиції та ін.) зручним чином для генерації ходів, оцінки позиції та пошуку [3].

Генерація ходів – процес створення списку легальних чи псевдо-легальних ходів в даній позиції. Генерація ходів найбільше серед інших задач залежить від представлення дошки.

Пошук найкращих ходів та варіацій – процес побудови орієнтованого графу, в якому кожна вершина – шахова позиція, а кожне ребро графу – можливий хід. Пошук полягає в аналізі даного графу на певній глибині для оцінки позицій, що можуть бути досягнуті з поточної. Точність пошуку

На вхід одношарового нейрона надходить вхідний вектор – набір вхідних сигналів $x = \{x_j\}, j = 1, 2, \dots, N$, де N – кількість входів [4].

Кожний вхідний сигнал x_j зважується (масштабується певним чином) відносно зіставленої йому ваги зв'язку (вагового коефіцієнта) w_j , яка моделює перетворення сигналу у синапсі (міжнейронному контакті) [4].

Дискримінантна (вагова) функція нейрона ϕ поєднує зважені вхідні сигнали, отримуючи постсинаптичний потенціал та подає його значення до функції активації ψ , яка видає скалярне значення, що видається на виході нейрона [4].

Отже, математична модель функціонування штучного нейрона описується наступним співвідношенням (1.1):

$$y = \psi(\phi(w, x)), \quad (1.1)$$

де x – вектор вхідних аргументів (сигналів);

y – значення на виході нейрона;

ψ – функція активації;

ϕ – дискримінантна функція;

$w = \{w_j\}$ – вектор, що містить значення вагових w_j коефіцієнтів і значення зсуву (порогове значення) w_0 .

Глибинна нейронна мережа – це різновид ШНМ, що має багато шарів обробки даних, яка перетворює вхідні данні у вихідні, ієрархічно виділяючи та агрегуючи ознаки, підвищуючи рівень абстракції даних в напрямку від входів до виходів [4].

Нейронні мережі прямого поширення сигналу – це нейронні мережі, в яких формальні нейрони розділено на шари, а зв'язки між нейронами організовано таким чином, що сигнал від попереднього шару передається наступному (зворотній напрямок неможливий), та зв'язки між нейронами в

одному шарі відсутні. Значення з вихідного шару нейронів утворюють загальний вихідний сигнал ШНМ [4].

Багат шаровий персептрон – штучна нейронна мережа прямого поширення. Архітектурно характеризується кількістю шарів, кількістю нейронів у кожному з шарів та функцією активації на кожному з шарів [4].

1.2 Огляд існуючих методів реалізації шахових рухів

У даній роботі досліджується завдання реалізації шахового руху. Оскільки основними задачами шахового руху є представлення дошки, генерація можливих ходів, оцінка переваги у позиції, пошук найкращого ходу, розглянемо найпопулярніші методи та алгоритми реалізації переліченого функціоналу.

1.2.1 Методи представлення дошки

Під представленням дошки розуміють структуру даних або їх сукупність, яка дозволяє виконувати генерацію ходів, пошук та оцінку позиції.

Лише положення фігур на дошці недостатньо для повної інформації про позицію. Для дотримання усіх шахових правил [5], потрібно також зберігати інформацію про можливість рокірування, черговість ходу, можливість взяття на проході для конкретного поля. Представлення має надавати можливість перевірити стан гри (чи завершена гра та з яким результатом), а також робити та відмінити ходи.

В даному підрозділі розглянуті як нотації для непрограмного представлення позиції (FEN), так і програмні методи реалізації представлення дошки.

Програмні методи представлення дошки поділяються на поле-центровані та фігуро-центровані.

Поле-центровані представлення зазвичай зберігають інформацію про усі поля шахової дошки та надають можливість звернутися навіть до порожніх полів. Такий вид представлення дає можливість дуже швидко перевірити стан будь-якого поля, проте вимагає повного сканування дошки для пошуку конкретної фігури.

Фігуро-центровані представлення зберігають лише існуючі на дошці фігури та їх розташування на дошці. Це дозволяє відносно легко та швидко виконати пошук конкретної фігури, проте не дозволяє одразу перевірити зайнятість конкретного поля.

1.2.1.1 Нотація FEN

Через великий обсяг інформації для опису однієї шахової позиції та різноманітність способів представлення цієї інформації у різних шахових рушіях було винайдено декілька універсальних форматів представлення шахової позиції у вигляді послідовності символів. Один з таких форматів – FEN (Forsyth-Edwards Notation) [6].

FEN можна представити за допомогою наступної послідовності (1.2):

$$\langle FEN \rangle = \langle PP \rangle + \langle STM \rangle + \langle CA \rangle + \langle EPTS \rangle + \langle HC \rangle + \langle FC \rangle, \quad (1.2)$$

де $\langle FEN \rangle$ – символний рядок, що повністю кодує позицію;

$\langle PP \rangle$ – послідовність символів, що кодує розташування фігур (розшифровується як Piece Placement – Розташування фігур);

$\langle STM \rangle$ – послідовність символів, що кодує черговість ходу (розшифровується як Side To Move – Черга ходу);

$\langle CA \rangle$ – послідовність символів, що кодує можливість рокірування (розшифровується як Castling Abilities – Можливість рокірування);

$\langle EPTS \rangle$ – послідовність символів, що кодує можливість взяття на проході (розшифровується як En Passant Target Square – Поле взяття на проході);

$\langle HC \rangle$ – число, що позначає кількість полу-ходів (ходи однієї сторони), з останнього ходу пішаком чи взяття фігури (розшифровується як Halfmove Clock – лічильник полу-ходів);

$\langle FC \rangle$ – число, що позначає кількість повних ходів у грі (розшифровується як Fullmove Counter – лічильник повних ходів).

У якості роздільника між послідовностями використовується пробіл.

Розташування фігур представляється наступною послідовністю символів (1.3):

$$\langle PP \rangle = \langle r_8 \rangle / \langle r_7 \rangle / \langle r_6 \rangle / \langle r_5 \rangle / \langle r_4 \rangle / \langle r_3 \rangle / \langle r_2 \rangle / \langle r_1 \rangle, \quad (1.3)$$

де $\langle PP \rangle$ – послідовність символів, що кодує розташування фігур;

$\langle r_i \rangle$ – послідовність для представлення горизонталі шахової дошки.

Для кодування фігур використовуються літери латинського алфавіту («р» – пішак (pawn), «п» – кінь (knight), «б» – слон (bishop), «г» – тура (rook), «к» – ферзь (queen), «к» – король (king)). Для позначення білих фігур використовують великі літери латинського алфавіту, для позначення чорних – маленькі.

Для кодування порожніх полів використовують числа – кількість порожніх полів поспіль у горизонталі.

Для кодування черговості ходу використовують маленьку літеру латинського алфавіту «w» або «b», відповідно вказуючи, що зараз хід білих («w» – white), або чорних («b» – black).

Для кодування можливості рокірування використовують латинські символи «k» та «q», відповідно для позначення рокірування у королівську сторону («k» – king) та сторону ферзя («q» – queen). Для позначення можливостей рокірування для білих використовують великі латинські літери, для чорних – маленькі. За відсутності будь-яких можливостей рокірування використовують символ «-».

Для кодування поля для взяття на прохіді поле записують у форматі «вертикаль» «горизонталь», наприклад «a3» чи «h6». Потрібно звернути увагу, що можливі горизонталі лише 3 та 6 (за правилами шахів). За відсутності використовують символ «-».

На рисунку 1.2 зображено позицію, яка у форматі FEN виглядає як «rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1».

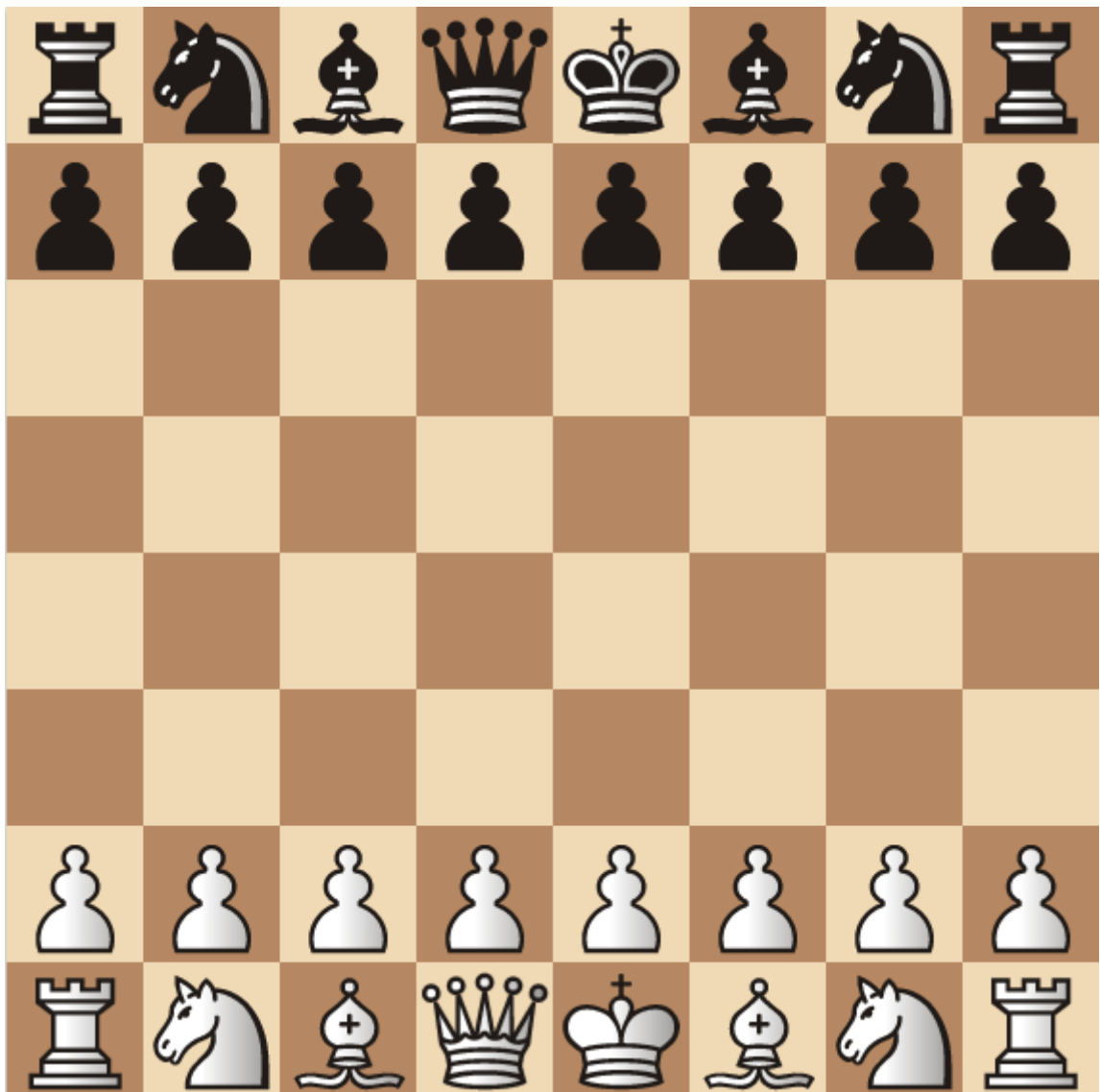


Рисунок 1.2 – Зображення шахової позиції [7]

Для створення зображень позицій було використано шаховий веб-додаток з відкритим кодом lichess.org [7].

1.2.1.2 Поштова скриня

Поштова скриня (mailbox) – поле-центроване представлення дошки, в якому до кожного поля можна звернутися знаючи конкретну адресу у пам'яті, через що метод і отримав свою назву [8].

Існують 3 найпоширеніші реалізації «поштової скрині»:

- Дошка 8x8;
- Дошка 10x12;
- Метод 0x88.

Дошка 8x8 – найпростіший метод реалізації поштової скрині. Реалізація полягає в створенні масиву з 64 змінних, кожна змінна представляє собою поле та фігуру, що його займає. Даний метод простий, проте має деякі недоліки з точки зору продуктивності в генерації ходів, через що є досить непопулярним [9].

Дошка 10x12 – вдосконалена дошка 8x8. Вдосконалення полягає в існуванні додаткових полів навколо основних, які використовуються в генерації ходів для перевірки валідності ходу [10].

Метод 0x88 – інше вдосконалення дошки 8x8. Представляє собою створення масиву з 128 змінних, лише 64 з яких представляють поля. Інші 64 використовуються для генерації ходів. Для перевірки валідності індексу поля потрібно накласти на нього бітову маску 0x88. Якщо результат ненульовий, то індекс поля не є дійсним, що дає значні переваги у швидкості при генерації ходів [11].

Переваги методу «поштова скриня»:

- дуже легка концепція для розуміння та програмній реалізації;
- може бути використана для неklasичних шахових варіантів;
- легка підтримка та пошук помилок, які дуже важко шукати в інших методах.

Недоліки методу «поштова скриня»:

- з точки зору швидкості роботи, процесору доведеться перевіряти багато умовних конструкцій (цикли, перевірки);
- важко перевіряти валідність ходу (через можливий шах королю);
- довго знаходити місцезнаходження конкретної фігури;
- довго генерувати ходи певного типу (лише взяття, лише тихі ходи).

Даний тип представлення використовується або в програмах, де не важлива швидкість виконання основних операцій, або в комбінаціях з іншими методами представлення.

1.2.1.3 Бітова дошка

Бітова дошка (англ. Bitboard) – структура даних для фігуро-центрованого представлення дошки. За своєю суттю бітові дошки – 64 розрядні змінні, кожен біт якої представляє собою одне поле дошки. Якщо біт рівний нулю, це означає, що поле дошки порожнє. Одиниця означає зайнятість поля. Також зустрічаються назви «бітовий набір» та «бітова мапа» [12].

Очевидно, що з одного біта інформації неможливо точно визначити тип фігури, що займає поле. Саме тому дошку неможливо представити одним бітовим набором, для цього використовується декілька бітових дошок та операції над ними. Зазвичай потрібно 1 бітова дошка для кожного типу фігури та для двох кольорів. За допомогою операції & (побітове AND) між бітовою дошкою «чорні фігури» та бітовою дошкою «пішаки» можна за одну операцію отримати бітову дошку «чорні пішаки». Приклад зображено на рисунку 1.3. Для спрощення сприйняття, нулі було замінено на крапки.

1 1 1 1 1 1 1 1		
1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
.		
.		
.	&	=
.		
.	1 1 1 1 1 1 1 1	
.		
чорні фігури	усі пішаки	чорні пішаки

Рисунок 1.3 – Приклад бітової операції над бітовими дошками

Оскільки кожна бітова дошка – скінченна множина з 64 елементів, над ними виконують більшість операцій, що і над множинами, а саме:

- об'єднання (наприклад для знаходження усіх фігур на дошці потрібно об'єднати 2 бітові дошки «чорні фігури» та «білі фігури»);
- доповнення (наприклад для знаходження множини усіх полів, на яких не розташовані ніякі фігури для генерації тихих ходів, потрібно знайти доповнення до бітової дошки «усі фігури»);
- перетин (наприклад, для знаходження полів взяття турою, потрібно знайти перетин між бітовими дошками «атаки тури» та «ворожі фігури»);
- різниця (наприклад, для знаходження усіх фігур, що не прив'язані до короля, потрібно знайти різницю між бітовими дошками «наші фігури» та «наші зв'язані фігури»);
- симетрична різниця (наприклад, для знаходження усіх полів певної вертикалі та горизонталі, окрім клітинки перетину, потрібно знайти симетричну різницю між бітовими дошками «горизонталь» та «вертикаль»).

Усі перелічені операції мають ті самі властивості, що і операції над множинами. В якості універсальної множини виступає бітова дошка з усіма

бітами встановленими в значення 1, в якості порожньої множини виступає бітова дошка з усіма бітами встановленими в значення 0.

Для повноцінного користування бітовими дошками потрібно вміти швидко знаходити наймолодший біт у бітовій множині та знаходити кількість встановлених бітів. Це потрібно для знаходження кількості певних фігур (наприклад фігур, які дають шах) та для знаходження індексу клітинки, на якій стоїть фігура.

Знаходження наймолодшого (чи найстаршого) біту в бітовій множині – одна з найчастіше використовуваних операцій в шахових рушіях, тому дуже важливо, щоб ця функція працювала швидко, а не перебирала кожний біт починаючи від наймолодшого (найстаршого). Було винайдено декілька способів, але розглянемо найшвидший та найрозповсюдженіший.

У 1997 році було створено алгоритм, який базується на послідовностях де Брейна. Алгоритм в перекладі звучить як «Використання послідовностей де Брейна для індексації одиниці в комп'ютерному слові» [13]. Під словом розуміється саме 64-бітне слово, що і використовується для бітових дошок. Алгоритм полягає в проведенні певних операцій над 64-бітним словом та використанні його 6 найстарших бітів як індекс для прорахованої заздалегідь послідовності, в якій збережено індекс наймолодшого біта. Розрахунки наведено у формулі (1.4).

$$lsb = S((B \& -B) * D_{64}) >> 58), \quad (1.4)$$

де lsb – індекс наймолодшого біта в 64-бітному слові;

B – вихідне 64-бітне слово;

D_{64} – константа, що дорівнює 285870213051386505, більш відома в шістнадцятковому представленні 0x03F79D71B4CB0A89;

S – ін'єктивне відображення, що для кожного з 64 значень старших 6 бітів перетвореного значення (від 0 до 63) повертає індекс наймолодшого біта в

64-бітному слові. Послідовність значень: {0, 1, 48, 2, 57, 49, 28, 3, 61, 58, 50, 42, 38, 29, 17, 4, 62, 55, 59, 36, 53, 51, 43, 22, 45, 39, 33, 30, 24, 18, 12, 5, 63, 47, 56, 27, 60, 41, 37, 16, 54, 35, 52, 21, 44, 32, 23, 11, 46, 26, 40, 15, 34, 20, 31, 10, 25, 14, 19, 9, 13, 8, 7, 6} [13].

В якості ін'єктивного відображення в програмуванні використовується масив з 64 чисел, індексування якого і відбувається старшими 6 бітами перетвореного значення [14].

Популяція бітової дошки – кількість бітів у ній зі значенням 1. Для знаходження популяції бітової дошки також існує багато способів. Дана операція не є часто використовуваною в шахових рушіях, проте навіть для неї бажано мати спосіб підрахування швидший за простий перебір бітів.

Алгоритм було запропоновано Дональдом Кнудом і полягає в розбитті 64-бітного слова на малі 2-бітні послідовності, підрахування кількості бітів в кожній за 3 операції, і подальше об'єднання послідовностей в 4-бітні, 8-бітні та назад у 64-бітні. Розрахунок може бути поданий у 4 перетворення за формулами (1.5) – (1.8) [15].

$$x_2 = x - ((x \gg 1) \& k_1), \quad (1.5)$$

де x_2 – перший проміжний результат, в якому послідовні 2 біти містять кількість активованих бітів у відповідній 2-бітній послідовності вихідного 64-бітного слова;

x – вихідне 64-бітне слово;

k_1 – константа, що дорівнює 0x5555555555555555 у шістнадцятковому вигляді (активовано старший 1 біт кожної підпослідовності довжиною 2);

$$x_4 = (x_2 \& k_2) + ((x_2 >> 2) \& k_2), \quad (1.6)$$

де x_4 – другий проміжний результат, в якому послідовні 4 біти містять кількість активованих бітів у відповідній 4-бітній послідовності вихідного 64-бітного слова;

x_2 – значення, вираховане за формулою 1.5;

k_2 – константа, що дорівнює 0x3333333333333333 у шістнадцятковому вигляді (активовано старші 2 біти кожної підпослідовності довжиною 4);

$$x_8 = (x_4 + (x_4 >> 4)) \& k_4, \quad (1.7)$$

де x_8 – третій проміжний результат, в якому послідовні 8 бітів містять кількість активованих бітів у відповідній послідовності вихідного 64-бітного слова;

x_4 – значення, вираховане за формулою 1.6;

k_4 – константа, що дорівнює 0x0f0f0f0f0f0f0f0f в шістнадцятковому вигляді (активовано старші 4 біти кожної підпослідовності довжиною 8);

$$P = (x_8 * k_f) >> 56, \quad (1.8)$$

де P – популяція (кількість бітів) вихідного 64-бітного слова;

x_8 – значення, вираховане за формулою 1.7;

k_f – константа, що дорівнює в шістнадцятковому вигляді (активовано старший біт кожної підпослідовності довжиною 8).

До переваг бітових дошок відносять:

- найчастіше висока щільність інформації;
- можливість оперувати полями шахової дошки як множинами;
- висока швидкість операцій над 64-бітовими словами на сучасних процесорах.

Недоліки бітових дошок:

- бітові дошки з низькою популяцією мають низьку щільність інформації;
- низька швидкість виконання на 32-бітних процесорах;
- відсутність можливості визначити зайнятість певного шахового поля.

Саме через останній недолік бітові дошки часто використовують в комбінації з представленням «поштова скриня».

1.2.1.4 Гібридне представлення

Гібридне представлення полягає у об'єднанні поле-центрованих та фігуро-центрованих методів представлення дошки. Зазвичай це робиться через те, що деякі операції зручно та швидко виконувати лише з одним типом представлення дошки (доступ до поля чи доступ до фігури).

Більшість сучасних шахових рушіїв містить у собі гібридне представлення дошки, об'єднуючи бітові дошки для швидкої генерації ходів та 8x8 дошки для швидкого доступу до будь-якого поля.

1.2.2 Методи генерації ходів

Генерація ходів – процес створення списку легальних або псевдо-легальних ходів у позиції. Легальним називається хід, який можна зробити у позиції без порушень правил. Псевдо-легальним називається хід, який не порушує правил пересувань фігур, проте може залишити гру в недійсному стані, адже псевдо-легальні ходи не перевіряють, чи залишається король у шаху [16].

Генератори ходів можуть створювати і інші види ходів для спеціальних випадків у побудові графу пошуку, такі як взяття, шахи,

ухилення від шаху, заміни пішака або лише тихі ходи. Тихим називається хід, який не змінює кількість фігур на дошці.

Генерація ходів посідає перше місце з використання у шаховому рушії, адже для побудови графа пошуку потрібно проводити генерацію ходів для кожної позиції. Саме тому генерація ходів має бути оптимізованою та швидкою.

Зазвичай фігури в генерації ходів діляться на ковзні та нековзні. Ковзними називають фігури, що можуть рухатися на будь-яку відстань за умови, що ніщо не перешкоджає їх руху. До ковзних фігур належать слони, тури та ферзі. До нековзних відповідно належать королі, пішаки та коні. Дане розділення відбувається через схожість обробки ситуацій для ковзних фігур. Нековзні фігури ж не піддаються узагальненню, тому генерацію ходів для коней, пішаків та королів зазвичай розглядають окремо.

Оскільки 2 найрозповсюдженіших представлення дошки це «Поштова скриня» та «Бітова дошка», розглянемо процес генерації ходів у комбінації з цими представленнями.

Оскільки дошка у людському уявленні – таблиця розмірності 8x8, а в пам'яті комп'ютера вони представляються як послідовності з 64 елементів, людські напрями на дошці зазвичай мають свої аналоги в програмах для спрощення сприйняття програмного коду та для зменшення кількості допущених помилок. Наприклад, для того, аби отримати першу клітинку у північному напрямку, для представлення «поштової скрині 8x8» потрібно додати до індексу клітинки 8, для «поштової скрині 10x12» потрібно додати 10, а для бітових дошок потрібно здійснити побітовий зсув вправо на 8 бітів. Зазвичай для кожного представлення заздалегідь визначаються значення для наступних напрямків:

- ортогональні напрями (північ, південь, захід, схід);
- діагональні напрями (південний схід, південний захід, північний схід, північний захід);

Для ходів коней та подвійного ходу пішаком вводяться або нові напрями, або використовується комбінація з існуючих.

1.2.2.1 Генерація ходів з представленням «Поштова скриня»

Для представлення «Поштова скриня» генерація ходів настільки ж проста в розумінні та реалізації як і саме представлення. Для кожної знайденої нековзної фігури потрібно обробити усі клітинки, в які вона може потрапити, а для ковзних фігур потрібно перевірити усі клітинки у заданому напрямленні до першої блокуючої фігури.

Очевидно, що такий метод дозволяє генерувати лише псевдо-легальні ходи. Для перевірки їх легальності витрачається додатковий час на здійснення ходу на дошці та знаходження усіх атакованих полів (що займає майже такий самий час як і генерація ходів супротивника). Як видно, для перевірки кожного згенерованого псевдо-легального ходу здійснюється багато операцій, яких хотілося б уникнути.

Переваги методу:

- легкість в розумінні та реалізації;
- легкість підтримки програмного коду та пошуку помилок;

Недоліки методу:

- довгий час роботи та висока алгоритмічна складність;
- відсутність підтримки концепції «зв'язаних» фігур та шаху.

1.2.2.2 Генерація ходів з представленням «Бітова дошка»

Загалом метод полягає в використанні заздалегідь підрахованих таблиць перетворення індексу поля в множини можливих полів для ходу для кожного типу фігури. Наприклад, на рисунку 1.4 зображено бітові множини кінцевих полів ходу для різних типів фігур, що знаходяться на полі c5.



Рисунок 1.4 – Генерація ходів для різних типів фігур

Для знаходження індексів клітинок розташування фігур або кінцевого поля ходу, потрібно швидко конвертувати бітову дошку в множину індексів полів на дошці, оскільки це найчастіше використовувана операція в генерації ходів з бітовими дошками. Оскільки відомо швидкий метод для знаходження наймолодшого біта в 64-бітному слові, використаємо його та встановимо наймолодший біт рівним нулю за формулою (1.9). Таким чином потрібно повторювати дану операцію, доки в бітова множина не стане порожньою:

$$B_0 = B \& (B - 1), \quad (1.9)$$

де B_0 – нова бітова дошка, в якій наймолодший біт встановлено в 0;

B – попередня бітова дошка, в якій потрібно деактивувати наймолодший біт.

За допомогою операції (1.9) можливо конвертувати положення власних фігур у індекси полів, використовувати ці індекси для отримання відповідного значення із заздалегідь підрахованих таблиць. Отримані значення знову перетворюються на індекси полів на дошці, отримуючи повний опис ходу – поле, з якого ходить фігура, та поле, до якого ходить фігура.

Обробку ходів пішаків бітовими дошками можливо робити одночасно для одразу усіх пішаків. Для цього використовуються операції побітового зсуву над бітовими множинами, але у випадку діагональних зсувів відбувається деактивація бітів на першій та останніх вертикалях дошки за допомогою бітових масок та операцій над множинами. На рисунку 1.5 зображено використання бітових дошок для генерації ходів пішаків.

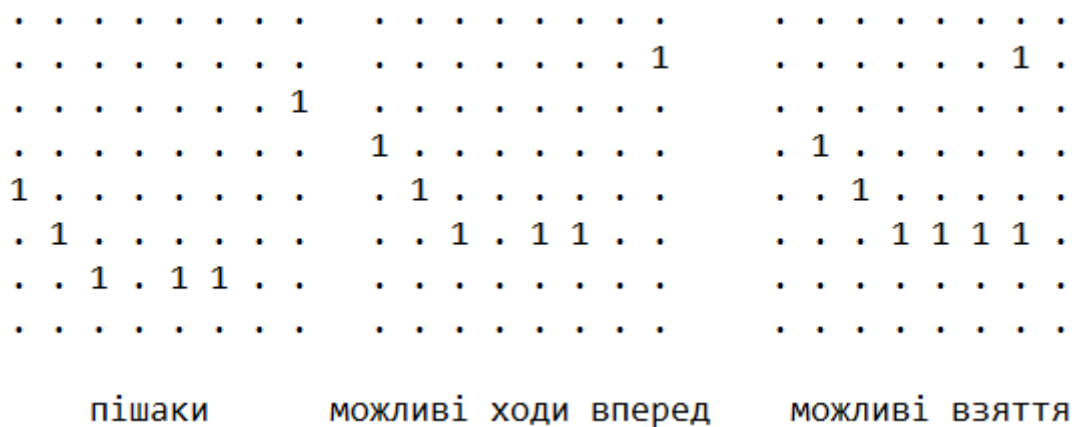


Рисунок 1.5 – Генерація ходів пішаків побітовим зсувом

Для генерації ходів ковзних фігур недостатньо отримати заздалегідь підраховане значення з таблиці за індексом, адже їх можливі ходи залежать від інших фігур на дошці. Наприклад, для тури важливе розташування фігур, що знаходяться на тій самій вертикалі та горизонталі, для слонів – розташування фігур на тій самій діагоналі та анти-діагоналі. Саме тому потрібно вдосконалити метод з підрахуванням значень [17].

Сучасна модифікація полягає в розширенні таблиці заздалегідь підрахованих значень. Метод називається магічні бітові дошки (Magic Bitboards), і полягає в побудові 1-1 відповідності між зайнятістю на дошці та можливими ходами фігури [18].

Очевидно, що заздалегідь підрахувати усі 2^{64} можливих розташувань неможливо, тому потрібно зменшити множини можливих розташування фігур.

По-перше, потрібно брати до уваги зайнятість клітинок на ортогональних чи діагональних лініях, на яких знаходиться фігура для тури та слонів відповідно. Таким чином, для ходів тури релевантні завжди лише 2^{14} значень, а для ходів слона релевантні максимум 2^{13} значень для кожного поля (рисунок 1.6). Для ходів ферзя використовується об'єднання можливих ходів тури та слона з заданого поля.

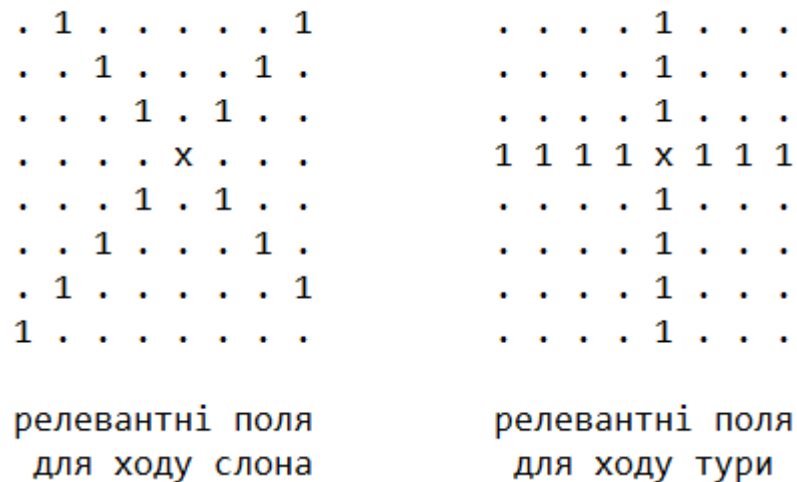


Рисунок 1.6 – Поля зайнятості для визначення можливих ходів

Як видно з рисунку, можливі ходи не залежать від розміщення фігур на полях, що позначені крапкою.

По-друге, для ортогональних і для діагональних напрямлень, блокувати подальше пересування можуть лише фігури, що знаходяться не на останньому полі променю атаки (рисунок 1.7). Виявляється, що для діагональних ковзних фігур важливі максимум 9 полів, а для ортогональних – 12, тобто необхідно заздалегідь підрахувати по 2^9 та 2^{12} значень відповідно, що значно менше за початкові 2^{64} та не займає багато пам'яті.

.	.	.	.	.	.	.	.
.	.	1	.	.	.	1	.
.	.	.	1	.	1	.	.
.	.	.	.	x	.	.	.
.	.	.	1	.	1	.	.
.	.	1	.	.	.	1	.
.	1	.	.	.	.	.	.
.	.	.	.	.	.	.	.
релевантні поля для ходу слона							

.	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.
x	1	1	1	1	1	1	.
релевантні поля для ходу тури							

Рисунок 1.7 – Релевантні поля зайнятості без останнього поля променю

Алгоритм магічних бітових дошок полягає в перетворенні відносної зайнятості релевантних полів на індекс для заздалегідь підрахованої таблиці. Робиться це за рахунок операцій множення та побітового зсуву з певними значеннями, які в деяких рушіях генеруються самотійно, а деякі використовують заздалегідь пораховані. За допомогою цих операцій утворюється ін'єктивне відношення між зайнятістю полів та індексами в таблиці. Загалом, отримання бітової дошки з можливими атаками можна описати наступним відношенням (1.10):

$$B_a = T(Sq, M(B_o)), \quad (1.10)$$

де B_a – бітова дошка з можливими ходами фігури;

T – ін'єктивна функція, що за клітинкою та перетвореним значенням повертає можливі атаки (зазвичай використовується матриця або масив);

Sq – клітинка, в якій знаходиться фігура;

M – сюр'єктивна функція перетворення зайнятості полів на індекс;

B_o – бітова дошка, що описує зайнятість полів.

Останнім аспектом в даному методі є попередній розрахунок усіх значень для таблиці. Виконується це за допомогою метода гіперболічна квінтесенція (Hyperbola Quintessence) [19], який здатний вирахувати атаки вздовж прямої лінії за формулою (1.11).

$$B_{LA} = (B_o \& B_L - 2 * B_{Sq}) \otimes R(R(B_o \& B_L) - 2 * R(B_{Sq})) \& B_L, \quad (1.11)$$

де B_{LA} – множина полів з усіма можливими атаками вздовж певної лінії;

B_o – множина полів з фігурами на дошці;

B_L – множина полів що належать певній лінії;

B_{Sq} – множина з одного елемента – поля, на якому стоїть задана фігура;

R – функція, що утворює обернену множину.

В якості оберненої використовується функція, що розгортає біти бітової дошки в зворотному напрямі.

Оскільки бітові дошки підтримують операції над множинами, генерація конкретного типу ходів стає дуже простою. Для генерації взяття чужих фігур знаючи атаки нашого ферзя, потрібно знайти перетин цих множин. Для генерації тихих ходів потрібно знайти перетин з множиною порожніх полів (рисунок 1.8).

За тією самою причиною стають простими і перевірки на шах та зв'язані фігури. Фактично, для генерації взяття потрібно знайти перетин атак фігури з множиною з 1 елемента – поля з фігурою, що дає шах (адже за взяття іншої фігури король залишиться атакованим), а для генерації тихих ходів потрібно знайти перетин атак фігури з полями між фігурою, що дає шах та королем (рисунок 1.9).

Переваги методу:

- швидкість роботи;
- легкість перевірки шаху та зв'язаних фігур.

Недоліки методу:

- складна концепція для розуміння та для реалізації;
- висока вірогідність помилок при реалізації та складність пошуку цих помилок.

1.2.3 Методи оцінки переваги у позиції

Оцінка переваги у позиції – визначення переваги будь-якої зі сторін для прогнозування результату гри.

На відміну від представлення дошки та генерації ходів, для оцінки переваги відсутні правила та обмеження, тому створення точної функції оцінки позиції – складне завдання, саме тому використовуються різноманітні евристичні операції та правила, які майже ніколи не збігаються в двох різних шахових рушіях [20].

Один з головних аспектів, на який потрібно звернути увагу при написанні функції оцінки позиції, це тип значення, що обчислює функція. Широкого застосування набули функції, що повертають одне ціле або дійсне число, яке і характеризує оцінку позиції. Одиниця вимірювання даної величини – сантипішак. Кожне ціле число переваги розцінюється як перевага у одну соту пішака. Знак цієї величини вказує на перевагу певної сторони, але інтерпретується по-різному та залежить від типу оцінки:

- абсолютна оцінка позиції передбачає, що позитивні значення результату функції вказують на перевагу білих, негативні значення вказують за перевагу чорних, модуль значення вказує на якість переваги (що більше модуль значення, тим більше перевага). Значення нуль свідчить про відсутність переваги для будь-якої зі сторін;

– відносна оцінка позиції передбачає, що позитивні значення результату функції вказують на перевагу сторони, яка наступна за чергою має зробити хід. Негативні вказують на перевагу протилежної сторони. Так само як і з абсолютною оцінкою позиції, модуль значення вказує на якість переваги, а нуль – на її відсутність.

Деякі шахові рушії використовують різні функції оцінки позиції на різних етапах гри:

– для дебютів використовуються «книги», які описують типові початки для шахових ігор та найкращі відповіді на ходи супротивника;

– для мітельшпілю (середнього етапу гри) використовується основна функція оцінки. Зазвичай її вирізняє піклування про безпеку короля та комплексні атаки групами фігур на різних кінцях дошки;

– для ендшпілю використовується допоміжна функція оцінки. Зазвичай її вирізняє активація короля як фігури через відсутність важких фігур та небезпеки, та атаки пішаками для проведення пішака на восьму горизонталь та його заміни на важку фігуру.

З недавніх часів існує 2 основні методи оцінки позиції:

- традиційні (статичні, класичні);
- з використанням нейронних мереж.

1.2.3.1 Класичні методи оцінки позиції

Класичні методи оцінки позицій зазвичай включають в себе набір певних правил, за допомогою яких і розраховується фінальне значення.

Основним правилом в більшості шахових рушіїв є сумування вартості фігур. Кожній фігурі надається певна вартість у сантипішаках. Фігури супротивника мають від’ємні значення. Типовий розподіл вартості фігур:

- пішак – 100 сантипішаків;

– кінь – 285 сантипішаків (прийнято вважати, що кінь дещо гірше за слона, хоча це залежить від позиції);

– слон – 300 сантипішаків;

– тура – 500 сантипішаків;

– ферзь – 900 сантипішаків;

– король – 0 сантипішаків (як фігура, яку не можна втратити).

Аналізуючи позицію, можна помітити, що деякі фігури розкривають увесь свій потенціал знаходячись на певних полях або контролюючи певні поля. Саме на цьому засновується друге основне правило оцінки позиції. Для кожної фігури існує додаткова вартість за знаходження на певному полі, причому вартість може бути як негативною, так і позитивною, наприклад:

– для пішаків знижена вартість у початкових клітинках, та вартість збільшується з наближенням до останньої горизонталі. Це змушує оцінювати вище позиції, в яких пішаки ближче до заміни на важкі фігури;

– для коней, слонів та ферзів підвищено вартість на полях в центрі дошки, де вони контролюють більше полів та мають більше простору для маневрів, і знижено вартість на полях на краю дошки;

– для ферзів та тур підвищено вартість за знаходження на полях у території супротивника, а саме на сьомій (для білих) та другій (для чорних) горизонталях. В цих полях фігури контролюють простір супротивника, лімітуючи його можливості та змушуючи захищатися;

– для королів знижено вартість знаходження у центрі дошки (через високу небезпеку), та підвищено вартість знаходження у на полях рокірування, тим самим заохочуючи знаходження короля в безпеці та під захистом інших фігур. За цією ж причиною деякі пішаки (на третій та шостій вертикалях) мають знижені вартості за знаходження не на початкових позиціях.

Даний перелік правил досить тривіальний, проте дозволяє досягнути відносно високої точності та оцінювати деякі позиції краще за початківців та

на рівні аматорів. Проте даний метод має значний недолік: такі оцінки будуть актуальні лише для мідельшпілю, адже для ендшпілю варто активувати короля та допомагати пішкам просуватися, а не залишатися за початкових позиціях, захищаючи короля.

До більш комплексних правил належать:

- мобільність фігур – кількість простору для ходів, кількість можливих ходів у позиції;

- контроль центру (або інших частин дошки) – стратегія для контролю певних полів фігурами, захоплюючи простір та надаючи можливість іншим фігурам розвиватися в позиції. Може вимірюватися кількістю атак на певне поле, кількістю захистів певного поля, кількістю пішаків в центрі та іншими значеннями;

- об'єднаність – стратегія для захисту власних фігур та заохочування синергії між фігурами одного кольору;

- фігури у пастці – стратегія для іммобілізації ворожих фігур, зменшення кількості ходів супротивника та блокування діагоналей, вертикалей чи горизонталей проти ковзних фігур;

- безпека короля – одна з найважливіх стратегій, що має забезпечувати безпеку короля від мату в довготривалій перспективі. Оцінюються можливі атаки супротивника, кількість ворожих та союзних фігур поруч з королем, положення короля на дошці, мобільність короля та інші характеристики.

Переваги класичної оцінки позиції:

- легкість розширення оцінки новими правилами та властивостями;
- легкість реалізації;
- швидкість роботи.

Недоліки класичної оцінки позиції:

- складність підтримки великої кількості правил;
- складність вигадування нових правил, пошуку закономірностей;
- складність регресивного тестування внесених змін.

1.2.3.2 Методи оцінки позиції з використанням нейронних мереж

Оцінка позиції – регресійна задача з великим списком складних правил. Зазвичай з задачами, де існує велика кількість складних для підтримки правил нейронні мережі справляються краще за людей. Прикладом можуть бути такі сфери як розпізнавання зображень, звуку, нелінійні регресії складних величин.

У 2020 році декілька передових шахових рушіїв винесли оцінку позиції з використанням нейронної мережі в основний функціонал рушія, тим самим започаткувавши новий прорив у сфері шахових рушіїв та значно збільшивши перевагу рушіїв над людьми. Найвідоміші рушії, що успішно користуються нейронними мережами – Stockfish та Komodo.

Для вирішення регресійної задачі оцінки позиції найчастіше [21] використовуються наступні типи нейронних мереж:

- багат шаровий перцептрон;
- нейронні мережі, що ефективно оновлюються (Efficiently Updatable Neural Networks).

Нейронні мережі, що ефективно оновлюються (NNUE) – неповно зв’язні нейронні мережі прямого поширення, головна відмінність полягає в швидкій роботі без графічного процесора та в швидкому розрахунку виходу нейронної мережі у випадку неповного оновлення вхідного шару. Як результат, через неповне оновлення вхідного шару відсутня потреба повністю оновлювати перший прихований шар, і саме в цьому і полягає ефективне оновлення нейронної мережі [22].

Якщо для реалізації багат шарового перцептрону існує велика кількість навчальних матеріалів та готових фреймворків, то для створення NNUE доведеться реалізовувати створення, навчання та оновлення самостійно. Нейронні мережі, що ефективно оновлюються, з’явилися лише в

2018 році [22], через що ще не встигли набути значного обігу та не мають реалізацій у найпопулярніших фреймворках для глибокого навчання.

Для навчання та тестування багат шарових перцептронів та NNUE використовуються набори даних, що були створені за допомогою класичної оцінки позиції з великою кількістю правил.

Створення вхідного шару для повного представлення позиції є також складною задачею. У Stockfish NNUE використовується перенасичення вхідного шару, за яких нейронна мережа працювала з більш високою точністю. На вхідному шарі розташовані 82048 нейронів, по 41024 нейрони для кожного кольору. Кожен нейрон – бінарна ознака, що позначає розташування короля певного кольору на певній клітині, та іншої фігури на іншій клітині та 1 додатковий нейрон, що і становить $64 * (64 * 10 + 1) = 41024$ нейрони. Нейронна мережа має 3 приховані шари з кількістю нейронів 512, 32 та 32 відповідно. Функція активації на прихованих шарах – ReLu. Архітектура нейронної мережі Stockfish NNUE [23] зображена на рисунку 1.10.

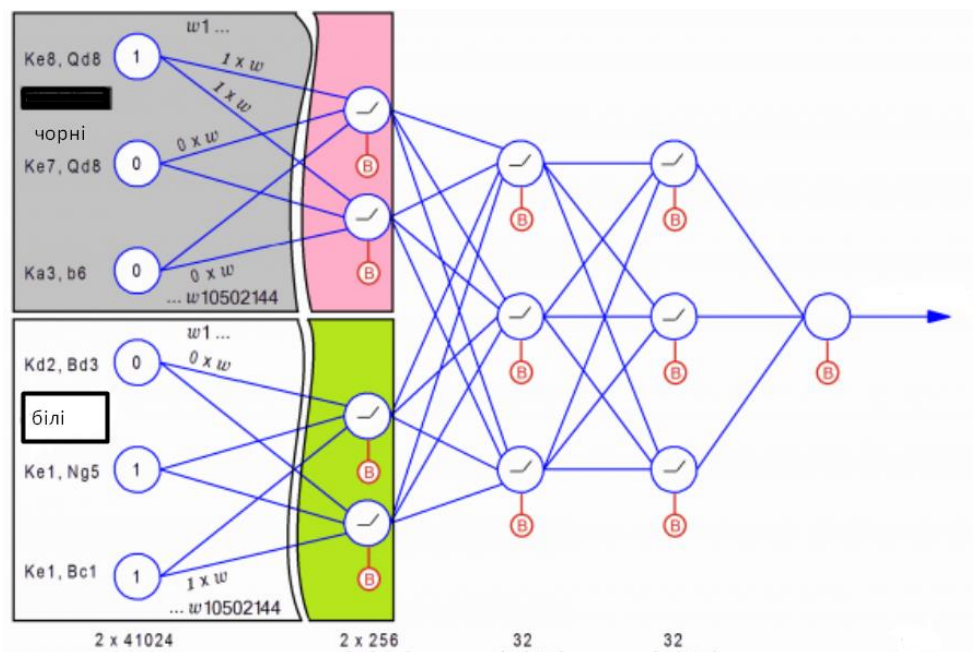


Рисунок 1.10 – Архітектура Stockfish NNUE [23]

Переваги оцінки позиції за допомогою нейронних мереж:

- відсутність необхідності вигадувати та підтримувати складні правила;
- можливість використовувати різні натреновані нейронні мережі для різних типів позицій.

Недоліки оцінки позиції за допомогою нейронних мереж:

- складність реалізації;
- необхідність наявності розміченого набору оцінених позицій;
- складність регресійного тестування та інтерпретації результатів.

1.2.4 Методи пошуку найкращого ходу

Оцінка позиції лише вказує перевагу у позиції, вона не дає уявлення про те, як саме цю перевагу потрібно реалізовувати, чи як потрібно захищатися, через що з'явилася потреба в пошуку найкращого ходу. Саме за допомогою побудови орієнтованого графу з позицій та переходами між ними в комбінації з функцією оцінки позиції можливо отримати більш точну оцінку, та точно вказати перевагу та обирати найкращий хід.

Орієнтований граф з позицій складається з вершин – позицій, та дуг – ходів, що дозволяють переходити з позиції у позицію.

Глибиною графу пошуку називається максимальна кількість переходів від поточної позиції до листка графу (позиції, з якої відсутні будь-які переходи).

Складність графу залежить від кількості можливих ходів у позиції, проте зі стартової позиції можливо досягти 400 позицій після двох пів-ходів, майже 5 мільйонів позицій після п'яти пів-ходів, та більш як 3 мільярди позицій за 7 пів-ходів. Через експоненціальний ріст розміру графу з його глибиною, шахові рушії не зберігають граф у явному вигляді, а найчастіше використовують depth-first-search та зберігати лише поточну позицію та

позиції, що їй передують. Для знаходження усіх дуг використовується генерація ходів.

Оскільки шахи належать до множини ігор з нульовою сумою (що означає, що перемога або перевага одного гравця, відповідно означає поразку або відставання другого гравця), до шах застосовується метод мінімакс та його оптимізації.

1.2.4.1 Мінімакс

Мінімакс (Minimax) – правило прийняття рішень, що використовується в теорії ігор, статистиці та штучному інтелекті для мінімізації можливих втрат за найгіршим сценарієм [24].

В якості оцінки позиції в даному методі використовується абсолютна оцінка позиції. Максимізатор (білі) намагається максимізувати оцінку позиції, в той час як мінімізатор (чорні) – навпаки, досягти найменшої оцінки позиції.

Наприклад, побудуємо граф можливих переходів на глибині 2 (це означає, що розглядається хід білих та відповідь суперника). Першими будуть проаналізовані позиції, що є листками графа. Після того, як для поточної позиції проаналізовані усі позиції, що з неї досягаються, оцінка даної позиції стає максимумом (у випадку білих) або мінімумом (у випадку чорних) з оцінок нащадків. Графічне представлення методу зображене на рисунку 1.11.

Незважаючи на те, що за допомогою другого ходу у білих є шанс досягти переваги у 7 очок, чорні можуть відповісти кращим для них ходом і перехопити перевагу, тому найкращим ходом для білих є третій хід, адже навіть за оптимальної гри чорних перевага залишається за білими.

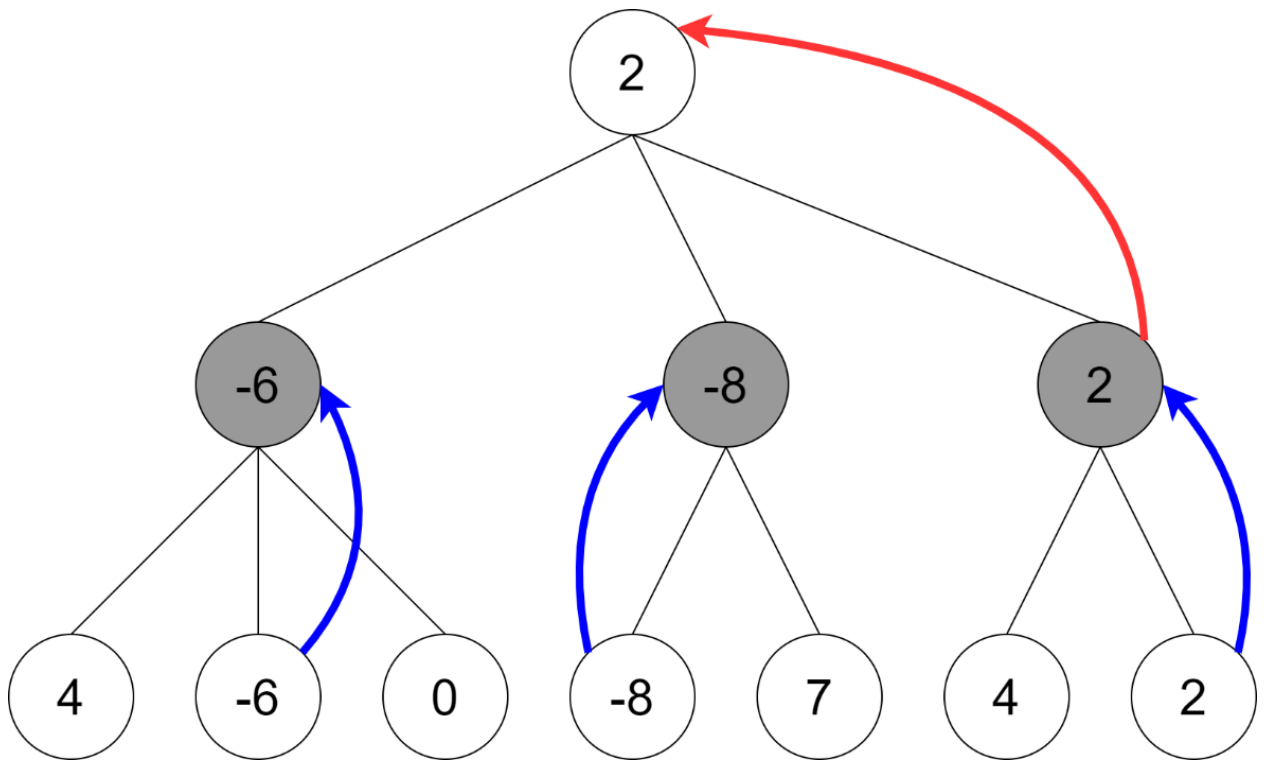


Рисунок 1.11 – Графічне представлення методу мінімакс

Переваги методу:

- простота сприйняття та реалізації;
- відсутність потреби побудови повного графу.

Недоліки методу:

- наявність потреби реалізації двох стратегій – максимізатора та мінімізатора;
- довгий час обходу графу на великій глибині.

Саме через останній недолік алгоритм було адаптовано та винайдено негамакс.

1.2.4.2 Негамакс

Одна з простих адаптацій мінімаксу, за якого існує лише стратегія максимізатора [25]. Дана модифікація можлива за рахунок рівності (1.12).

$$\max(a, b) = -\min(-a, -b), \quad (1.12)$$

де $\max$ – функція, що повертає максимальне значення;

$\min$ – функція, що повертає мінімальне значення;

a – перше дійсне число;

b – друге дійсне число.

Таким чином, серед усіх результатів, що може досягти суперник, повинен братися мінімальний з протилежним знаком. На рисунку 1.12 зображено графічне представлення методу негамакс.

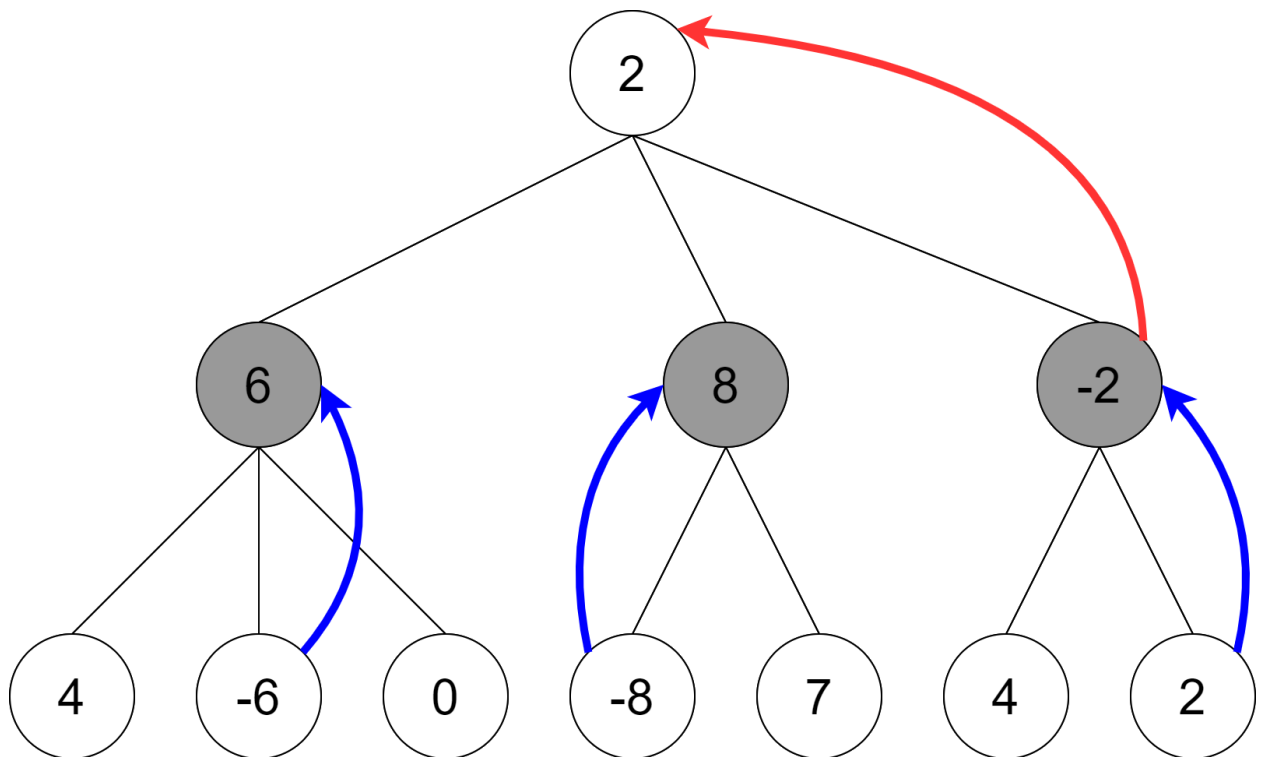


Рисунок 1.12 – Графічне представлення методу негамакс

Переваги даного методу зберігаються, і метод позбавляється одного з своїх недоліків – необхідності реалізації двох стратегій. Другий недолік – довгий час обходу усього графу – зберігається, для чого і було винайдено наступну модифікацію мінімаксу – відсічення альфа-бета.

1.2.4.3 Відсічення альфа-бета

Відсічення альфа-бета – модифікований метод пошуку мінімакс, що за певних умов значно зменшує кількість вершин, які необхідно оцінити в графі пошуку, при чому дозволяє отримати той же чисельний результат [26].

Відсічення неінформативних гілок графу досягається за рахунок підтримки нових двох значень – альфа та бета. Альфа представляє собою мінімальну кількість очок, яку максимізатор обов’язково отримає. Бета представляє собою максимальну кількість очок, яку мінімізатор обов’язково отримає. Таким чином, щоразу, коли максимізатор натрапляє на позицію, яка оцінена нижче за альфу, він може одразу відсікти дану позицію та не аналізувати її глибше. Графічне представлення методу зображено на рисунку 1.13.

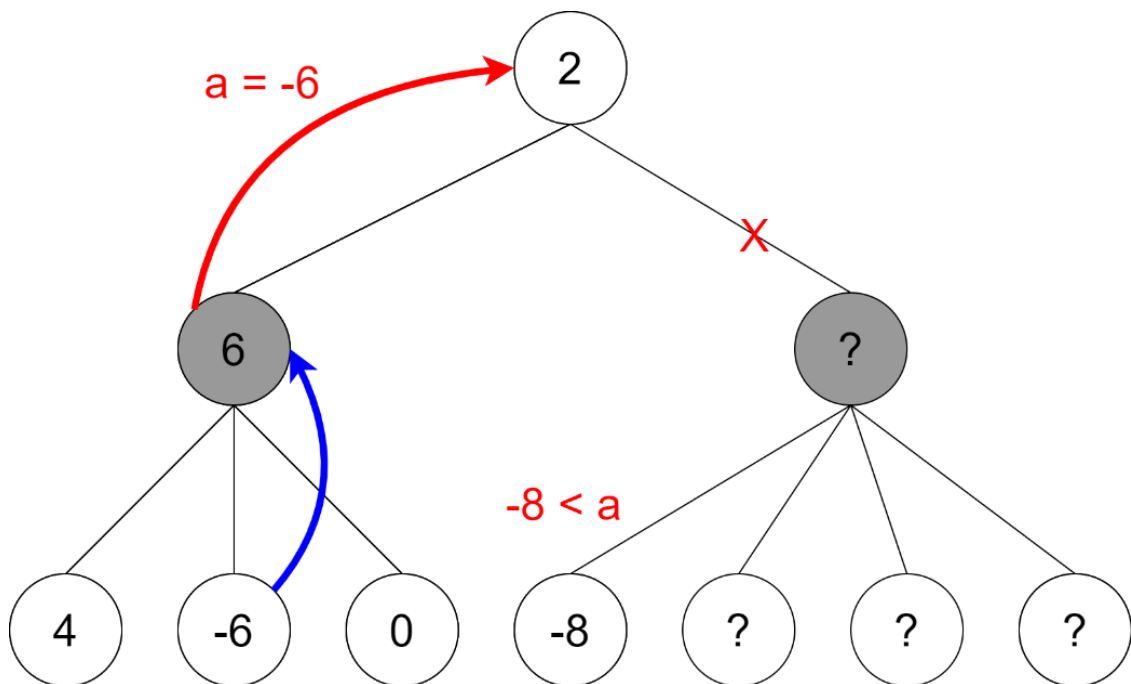


Рисунок 1.13 – Графічне представлення відсічення альфа-бета

Як видно з рисунку, після аналізу першого можливого ходу, білі гарантують собі щонайменше -6 очок. Очевидно, що перевага за чорними, і білим потрібно шукати кращий хід, проте у відповідь на другий можливий

хід у чорних є відповідь з оцінкою -8 на користь білих. На цьому етапі аналізувати інші відповіді чорних не має сенсу, адже в чорних вже є відповідь гірша для білих, ніж мінімально гарантована.

Оскільки ходи можуть бути проаналізовані в іншому порядку, відсічення альфа-бета не гарантує, що час обходу графу буде знижено. В найгіршому випадку, коли ходи будуть розташовані в порядку від найгіршого до найкращого, ніяких відсічень може зовсім не відбутися. Саме тому для відсічення альфа-бета дуже важливо впорядковувати ходи від найкращого до найгіршого. Звичайно, завчасно знати, який хід краще, а який гірше, неможливо, адже задля цього і робиться пошук. Проте можливо вигадати деякі правила, які дозволять впорядкувати ходи для прискорення пошуку, серед яких:

- хід у поле атаки противника – вірогідно поганий хід;
- хід, який забирає вартісною фігурою менш цінну захищену фігуру супротивника – вірогідно поганий хід;
- хід, який забирає менш цінною фігурою вартісну фігуру супротивника – вірогідно добрий хід;
- шахи, взяття та заміни пішака – вірогідно добрі ходи.

За допомогою даного методу в деяких випадках вдалося позбутися ще одного недоліку мінімаксу – довгої тривалості обходу графу з позиціями але навіть такий пошук не досить релевантний. Оцінки позицій у листках графу можуть значно змінитися на наступний хід – після взяття фігури. Щоб позбутися даного недоліку можна використати модифікацію алгоритму пошуку у шахах – тихий пошук.

1.2.4.4 Тихий пошук

Тихий пошук – модифікація алгоритму пошуку у шахах, яка полягає в тому, що при досягненні максимальної глибини у графі пошуку позиції не

оцінюються одразу, а оцінюються глибше з деякими обмеженнями. Обмеження полягає в тому, що для тихого пошуку не розглядаються усі ходи, а лише ті, які приводять позицію в «тихий» стан, коли відсутні взяття або тактичні ходи та неможливо різко перевернути оцінку позиції в чиюсь сторону [27].

Зазвичай в якості ходів, що розглядаються в тихому пошуку виступають взяття, проте деякі шахові рушії також розглядають шахи, вихід з шаху та напади.

Дана модифікація дозволяє підвищити точність оцінки позицій, проте значно підвищує час, який потрібен для обходу графа позицій (інколи у десятки разів).

1.2.4.5 Транспозиційні таблиці

Транспозиційні таблиці – модифікація пошуку в шахах, що вирішують проблему аналізу однакових позицій декілька разів [28].

Інколи можливо дійти до однієї позиції різними послідовностями ходів, наприклад зі стартової позиції: Кінь на f3 – пішак на e5 – пішак на e4 або пішак на e4 – пішак на e5 – кінь на f3. Приклад зображено на рисунку 1.14. Транспозиційні таблиці призначені для зберігання вже проаналізованих позицій та їх оцінки за деяким ключем та використовувати це в майбутньому.

Хоча на перший погляд метод простий для реалізації, існує дві основні проблеми з транспозиційними таблицями.

Перша з них це збереження великої позицій. Під час обходу дерева пошуку аналізуються мільярди позицій, що неможливо зберегти в оперативній пам'яті. Саме тому потрібно, по-перше, вміти зберігати лише ті позиції, які в теорії можуть зустрітися, а по-друге, зберігати лише невелику їх кількість у призначеній для цього пам'яті.

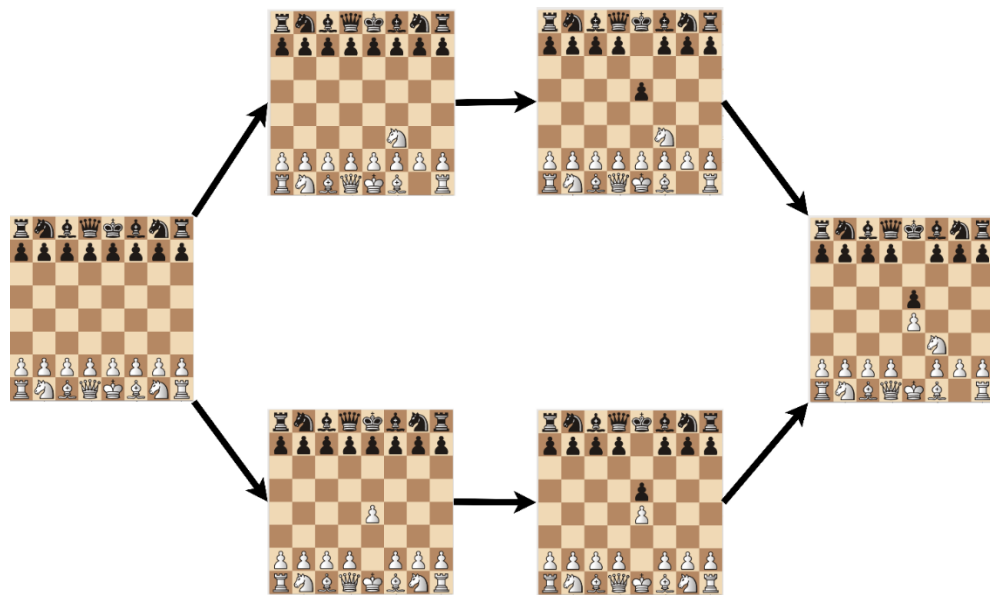


Рисунок 1.14 – Приклад транспозиції у шахах

Друга проблема стосується безпосередньо представлення позиції у пам'яті та знаходження її за ключем. Зберігати повну позицію у пам'яті досить витратно за ресурсами, а алгоритми хешування досить довго працюють. Для вирішення цієї проблеми застосовують інкрементальне оновлення хешів Zobrista.

Інкрементальне оновлення полягає у використанні минулого значення деякої характеристики для позиції та оновлення цього значення під час виконання та відміни ходу за константний час [29].

Хешування Zobrista – техніка для трансформування позиції у послідовність бітів довільної довжини з рівномірним розподілом серед усіх можливих значень. Зазвичай як ключ використовується 64-бітне значення. Хеш позиції рівний комбінації хешів усіх фігур на займаних ними полях (часто також комбінують хеші взяття на проході, хеш черговості ходу, хеші можливостей рокірування). Оскільки необхідна оборотна функція комбінування (для інкрементального оновлення), використовується побітове XOR. Хеші Zobrista також використовуються для виявлення повторень позиції [30].

1.2.4.6 Ітеративне заглиблення

Ітеративне заглиблення вирішує одразу дві проблеми – порядок обходу позицій (впорядкування згенерованих ходів) та вирішення, на якій глибині має проводитися пошук [31].

Що глибше пошук, то надійніше аналіз позиції, проте пошуки на велику глибину витрачають дуже багато часу. Існує можливість перервати пошук, проте в такому випадку ми втрачаємо будь-яку інформацію, адже пошук не завершено і саме остання гілка могла бути найінформативнішою. Для того, аби не виходити за рамки часу, концепція ітеративного пошуку пропонує провести пошук на низькій глибині, і далі поступово її збільшувати. Спершу здається, що це збільшить час аналізу, адже аналіз на низькій глибині не дає вагомих результатів та лише витрачає час. Проте, аналіз на невеликій глибині повертає найкращий хід у позиції, який надалі може використовуватися у пошуку в якості першого для кращого відсічення альфа-бета. Більш комплексний підхід вимагає збереження декількох найкращих ходів, а найкращі шахові рушії зберігають декілька найкращих послідовностей, які і стають опорними для пошуку на більшій глибині.

1.3 Порівняння існуючих методів реалізації шахових рушіїв

В даному підрозділі наведені порівняння методів представлення дошки, генерації ходів та оцінки позиції.

В таблиці 1.1 наведено порівняння методів представлення дошки та генерації ходів з відповідним представленням дошки.

Таблиця 1.1 – Порівняння методів представлення дошки та генерації ходів

Критерій порівняння \ Метод	Поштова скриня	Бітова дошка
Складність реалізації представлення	Низька	Висока
Ефективність застосування для основних операцій	Помірна	Висока
Складність реалізації генерації ходів	Помірна	Висока
Швидкість генерації ходів	Низька	Висока
Підтримка концепції шаху	Ні	Так
Підтримка генерації різних типів ходів	Ні	Так

Із порівняння в таблиці 1.1 видно, що бітові дошки програють лише за складністю реалізації, підтримки та пошуку помилок в реалізації.

В таблиці 1.2 наведено порівняння методів оцінки позиції.

Таблиця 1.2 – Порівняння методів оцінки позиції

Критерій порівняння \ Метод	Класична оцінка позиції	Оцінка позиції з використанням нейронних мереж
Складність реалізації	Помірна	Висока
Складність підтримки та розширення	Висока	Низька
Залежність від набору даних	Ні	Так
Швидкість роботи	Висока	Низька
Точність роботи	Висока	Висока
Здатність до узагальнення до різних етапів гри	Помірна	Помірна

Із порівняння в таблиці 1.2 видно, що не можна однозначно визначити, який з методів краще за інший, тому бажано в шаховому русії реалізовувати обидва методи та застосовувати в їх в залежності від обставин.

1.4 Постановка задачі до роботи

У даній роботі розглядається задача реалізації шахового русія.

Вимоги до реалізації шахового русія:

- розробити шаховий русій у вигляді бібліотеки та у вигляді бінарного виконуваного файлу з консольним інтерфейсом;
- реалізувати представлення дошки методом «бітові дошки»;
- реалізувати генерацію легальних ходів бітовими дошками;

- реалізувати аналіз позиції класичним методом та нейронною мережею;
- відібрати та підготувати дані для навчання і тестування нейронних мереж;
- реалізувати пошук найкращих ходів з відсіченням альфа-бета, ітеративним заглибленням, тихим пошуком.

1.5 Висновок

У даному розділі було досліджено предметну область та проаналізовано існуючі методи реалізації шахових рушіїв. Було порівняно існуючі методи представлення дошки, генерації ходів, аналізу позиції та пошуку найкращих ходів. Було визначено завдання до роботи та вимоги до шахового рушія.

2 ОПИС ПРОГРАМИ

В даному розділі наведено опис програми та розробку архітектури програми.

2.1 Вибір технологій та мов програмування

Проаналізувавши вимоги до програмного продукту, можна виокремити головні аспекти, що вплинули на вибір використовуваних технологій та мов програмування:

- висока швидкість роботи;
- висока алгоритмічна складність та як результат велика вірогідність допущення помилок в реалізації;
- застосування машинного навчання, а саме глибокого навчання;
- необхідність надати кінцеву програму як в якості бібліотеки (тобто як окремий модуль), так і в якості виконуваного файлу.

Розглядалися наступні мови програмування через популярність та наявність великої кількості навчальних ресурсів для даних мов програмування:

- C++;
- C#;
- Java;
- Python.

У таблиці 2.1 наведено порівняння мов програмування, що розглядалися як кандидати для реалізації шахового рушія.

Таблиця 2.1 – Порівняння мов програмування

Критерій \ Мова	C++	C#	Java	Python
Продуктивність	Висока	Середня	Середня	Низька
Автоматичний контроль пам'яті (Garbage Collector)	Ні	Так	Так	Так
Кросплатформерність модулів та бібліотек	Ні	Так	Так	Так
Наявність фреймворків для тестування	Так	Так	Так	Так
Наявність фреймворків для глибокого навчання	Так	Так	Так	Так
Підтримка фреймворку tensorflow для глибокого навчання	Так	Ні (неповна)	Ні (неповна)	Так

Незважаючи на автоматичний контроль пам'яті та кросплатформерність модулів та бібліотек, основною вимогою до шахового рушія є висока швидкість роботи основних модулів програми, через що для реалізації шахового рушія було обрано мову програмування C++. Проте програмні інтерфейси для роботи з нейронними мережами основних фреймворків мовою програмування є комплексними, тому для навчання моделі для економії часу доцільніше обрати іншу мову програмування.

Оскільки шахові рушії мають високу алгоритмічну складність, було обрано фреймворк для модульного тестування програм на мові програмування C++ googletest [32].

Для створення та тренування моделі було використано мову програмування Python з фреймворком Tensorflow [33], оскільки даний фреймворк має програмувальний інтерфейс як для мови програмування Python, так і для мови програмування C++, що дозволить використовувати моделі нейронних мереж у рушії. Було використано обгортку над фреймворком Tensorflow – crrflow [34], яка дозволяє виконувати базові операції з моделями (завантаження моделі з файлу та отримання передбачення).

Для тренування нейронної мережі через відсутність необхідного обладнання було обрано сервіс Google Colab, що дає обмежений доступ до графічного процесора та дозволяє тренувати моделі з високою швидкістю [35].

Для зручної роботи з даними в мові програмування Python було обрано сервіс для інтерактивних розрахунків IPython та Jupyter Notebook, що значно прискорюють роботу з даними через розбиття файлу на модулі, які можна виконувати незалежно. Через це зчитування великого об'єму даних відбувається лише 1 раз.

Для автоматизації збірки бібліотеки та виконуваного файлу рушія з вихідних файлів було обрано технологію CMake [36].

2.2 Вибір датасету та його обробка

Для тренування нейронної мережі необхідні розмічені дані. Зазвичай, дані беруться з класичних методів оцінки позицій. Було обрано датасет, знайдений на сервісі Kaggle [37], який містить 13 мільйонів шахових позицій з оцінкою найвідомішим шаховим рушієм Stockfish [38] на глибині пошуку 22.

Датасет складається з двох стовпців:

- позиція, що представлена у форматі FEN;

– оцінка позиції рушієм Stockfish у сантипішаках, або ж кількість ходів до мату позиції в разі його наявності.

Оскільки необхідно, щоб дані були одного формату, потрібно провести перетворення значень в даному датасеті для навчання моделі.

2.2.1 Підготовка оцінки позиції в датасеті

Задача підготовки оцінки (в даному випадку – вихідної ознаки) – перетворити кількість ходів до мату на оцінку позиції в сантипішаках.

Необхідно, щоб матова позиція оцінювалася вище за будь-яку іншу позицію, з якої неможливо поставити форсований мат за кілька ходів. Отже, першим кроком є встановлення для усіх матових позицій оцінки у довільне найбільше значення для білих та довільне найменше значення для чорних. Недолік такого підходу – однакова оцінка для матової позиції за 2 ходи та за 6 ходів.

Другим прийнятим рішенням було оцінювати кожен матову позицію в різницю довільного найбільшого числа та кількістю ходів до мату. Таким чином вдалося оцінити швидші мати вище за довгі матові атаки.

Останнім та найважливішим аспектом в перетворенні мату на оцінку було визначення довільного найбільшого та довільного найменшого числа. Не можна брати мільйон чи мільярд. Хоча це не порушить умови минулих двох спостережень, датасет стане дуже нерівномірним, що стане проблемою при навчанні моделі. Таким чином, необхідно визначити мінімальні абсолютні значення для матових позицій, для того, щоб вони були оцінені вище за позиції з великою перевагою, проте розподіл оцінок в датасеті не порушувався та не залишався на маленькому проміжку в нормалізованому датасеті.

Емпіричним шляхом було обрано величину 10000 сантипішаків для матових позицій. За кожен хід, що потрібно зробити до мату віднімається

додатково 100 сантипішаків. Для білих ця множина приймає додатні значення, для чорних – від’ємні.

2.2.2 Підготовка позиції в датасеті

Позиція у FEN форматі не представляє собою вдалий набір ознак для машинного навчання. Існує декілька способів представлення шахової позиції у вигляді вектору ознак для нейромережевої моделі [39], окремий випадок одного з яких вже було описано на прикладі архітектури NNUE у шаховому рушії Stockfish. Шахову позицію можна подати:

- у вигляді вектору бінарних ознак;
- у вигляді вектору числових ознак.

У випадку представлення позиції у вигляді вектору бінарних ознак, кожна клітинка шахової дошки представляється 12 (рідше 6) ознаками, для кожного типу фігури. Кожна ознака фактично означає, чи знаходиться фігура даного типу на даній клітинці. 12 ознак використовується у разі розглядання фігур різного кольору як фігур різного типу, 6 у випадку використання протилежних значень для різних кольорів фігури (наприклад -1 для білого коня і 1 для чорного). Також у вигляді бінарних ознак можуть представлятися можливості рокірування (4 ознаки), поле взяття на проході (16 ознак) та черговість ходу (1 ознака).

У випадку представлення позиції у вигляді вектору числових ознак, кожна клітинка шахової дошки представляється лише однією ознакою, а значення ознаки залежить від цінності фігури (наприклад, 1 для пішака, 3 для коня та слона, 5 для тури, 9 для ферзя та 10 для короля) та її кольору (від’ємні значення використовуються для протилежного кольору). Також так можна представити клітинку взяття на проході та можливості рокірування.

Можна використовувати об’єднання двох типів представлення, та представити клітинки дошки та розташування фігур у вигляді числових

ознак, та можливості рокирування та взяття на проході у вигляді бінарних ознак. Успіх даного представлення можна поставити під сумнів, адже не було знайдено ресурсів, в яких описувалися спробу навчати модель на такому наборі ознак.

Найпростішим та найлогічнішим є представлення дошки у вигляді бінарних ознак, саме це використовується в провідних шахових рушіях, тому для виконання поставленої задачі було обрано саме такий набір ознак.

2.3 Архітектура компонентів системи

В даному підрозділі описані архітектури розроблюваної програми та нейронної мережі.

2.3.1 Архітектура шахового рушія

Оскільки однією з вимог рушія є використання його як у вигляді бінарного файлу, так і у вигляді бібліотеки, було вирішено розділити рушій на 3 основні модулі:

- ядро рушія (в даний модуль входять усе, що стосується представлення дошки, генерації ходів, пошуку, класичної оцінки позиції та оцінки позиції за допомогою нейронної мережі);
- рушій (в даний модуль входять обгортки над бібліотекою для створення незалежного виконуваного файлу. Бібліотека приєднується до рушія статично);
- тестовий модуль (тестовий модуль створений для перевірки правильності роботи окремих компонентів ядра рушія).

Описана архітектура дозволить зменшити кількість компільованих файлів через винесення ядра рушія в окремий модуль, легко проводити тестування та задовольнити вимоги до програмного продукту.

Ядро рушія також буде використовувати бібліотеки Tensorflow та crrflow, які можливо приєднати статично та не залежати від цих бібліотек кінцевому користувачеві.

Навчання нейронної мережі відбуватиметься окремо від рушія з використанням мови програмування Python та сервісу Google Colab.

2.3.2 Архітектура нейронної мережі

За архітектуру нейронної мережі було обрано чотиришаровий перцептрон, останній шар якого містить 1 нейрон. Відомо, що тришаровий перцептрон є універсальним апроксиматором [4], проте 4 шари було обрано через успіх Stockfish NNUE, яка хоча і не є перцептроном, проте має 4 приховані шари та виходячи з того, що таким чином нейронна мережа легше узагальниться [23].

Потрібно зазначити, що незважаючи на успіх NNUE, використовувати їх досить складно та комплексно через відсутність фреймворків для побудови нейромереж з даною архітектурою. Саме тому довелося будувати архітектуру на основі перцептрона, а не більш успішної NNUE.

Для стабільної роботи шахового двигуна вимагається його швидка робота, а передбачення вихідної ознаки за вектором вхідних ознак відбувається досить довго (у порівнянні з класичною оцінкою позицій).

Через брак опису успішних типів архітектури нейронних мереж було обрано та навчено дві нейронні мережі з різними архітектурами для вибору точнішої та більш успішної.

Спільне в архітектурному плані для цих нейромереж – кількість шарів та функції активації на прихованих шарах. На прихованих шарах використовується функція активації ReLu, як і в Stockfish NNUE. Функція описана формулою (2.1). На вихідному шарі використовується функція активації гіперболічний тангенс, представлена формулою (2.2).

$$f(x) = \begin{cases} 0, & x \leq 0 \\ ax, & x > 0 \end{cases}, \quad (2.1)$$

де $f(x)$ – функція ReLu (rectified linear unit);

a – параметр функції;

x – аргумент функції;

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad (2.2)$$

де $\tanh(x)$ – гіперболічний тангенс;

e – експонента;

x – аргумент функції.

2.3.2.1 Архітектура нейронної мережі з повним поданням позиції

Архітектура першої нейронної мережі розроблена таким чином, що на вхідному шарі до нейронної мережі подається повне представлення позиції з розрахунком на узагальнюючі можливості нейронних мереж.

Вхідний шар – вектор бінарних ознак, який складається з 789 ознак:

– 768 бінарних ознак на опис фігур на дошці. Кожна з 64 клітинок описується 12 ознаками. Кожна з ознак вказує, яка саме фігура стоїть на клітинці. Перша ознака кожної клітинки відповідає за білого пішака, друга – за білого коня, третя – за білого слона, четверта – за білу туру, п'ята – за білого ферзя та шоста – за білого короля. Аналогічно до 12 ознаки відповідають за чорні фігури. Ознака приймає значення 1, якщо в даній клітинці знаходиться даний тип фігури, та 0 у протилежному випадку;

– 16 бінарних ознак на опис поля для взяття на проході. Перша ознака відповідає за поле А3, друга – В3, ..., дев'ята – А6, ..., шістнадцята – Н6. Ознака приймає значення 1 в випадку, якщо на дане поле можна виконати взяття на проході у поточній позиції, та 0 в протилежному випадку;

– 4 бінарні ознаки на опис можливостей рокірування. Перша ознака відповідає з можливості рокірування білими в сторону короля, друга – за можливість рокірування білими в сторону ферзя та аналогічно для чорних. Ознака приймає значення 1 в випадку, якщо така можливість рокірування існує та 0 в протилежному випадку;

– 1 бінарна ознака на опис черговості ходу. Ознака приймає значення 0 в випадку, якщо черга ходити білим та 1 – чорним.

На прихованих шарах розташовано 1024, 512, 128 та 1 нейрон відповідно. Останній шар є вихідним. На перших трьох прихованих шарах використовується функція активації ReLU, на вихідному – гіперболічний тангенс. Архітектуру нейронної мережі зображено на рисунку 2.1.

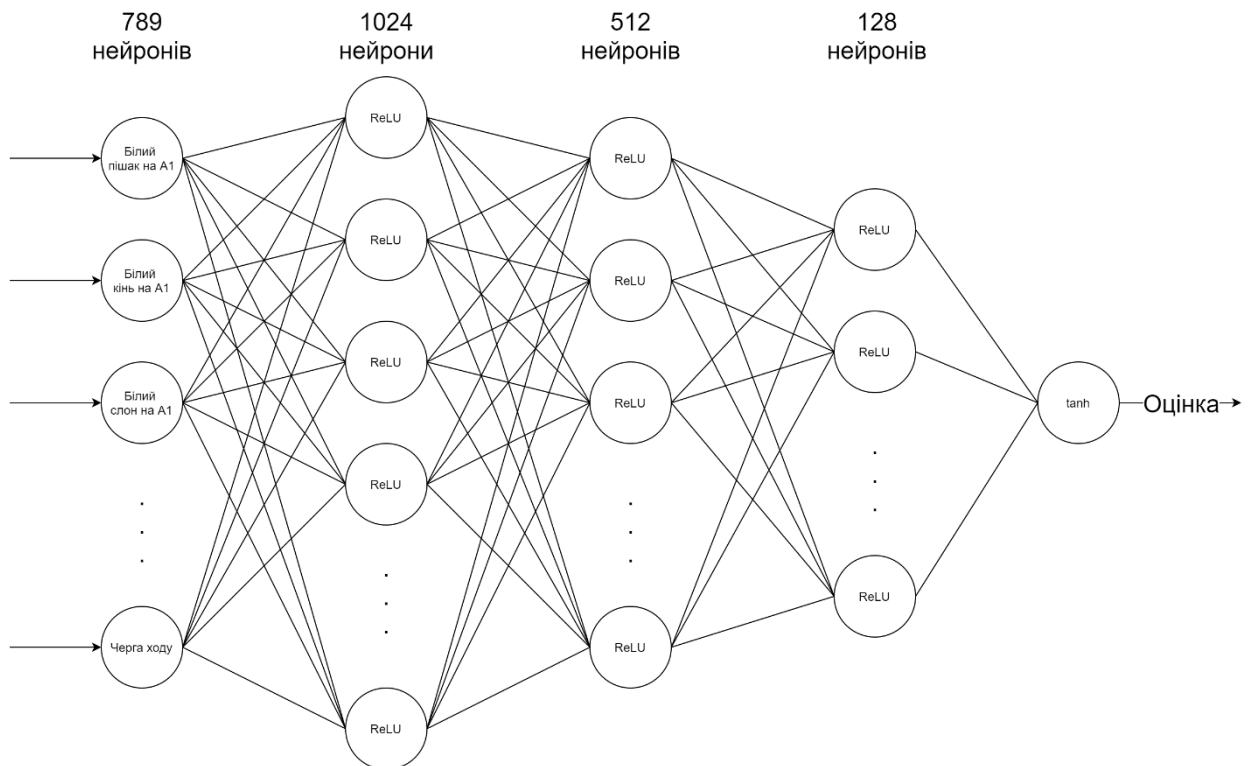


Рисунок 2.1 – Архітектура першої розробленої нейронної мережі

2.3.2.2 Архітектура нейронної мережі з неповним поданням позиції

Архітектура другої нейронної мережі розроблена таким чином, що на вхідному шарі до нейронної мережі подається неповне представлення позиції з розрахунком на брак узагальнюючих можливостей нейронної мережі.

Вхідний шар – вектор бінарних ознак, який складається з усього з 768 ознак на опис фігур на дошці. Модифікація полягає у вертикальному відображенні дошки у випадку черги ходу чорних для того, щоб фігури сторони, чия черга ходити, подавалася на вхід нейронної мережі однаково. Кожна з 64 клітинок описується 12 ознаками. Кожна з ознак вказує, яка саме фігура стоїть на клітинці. Перша ознака кожної клітинки відповідає за пішака того кольору, черга якого ходити, друга – за коня того кольору, черга якого ходити, третя – за слона того ж кольору, четверта – за туру того ж кольору, п'ята – за ферзя того ж кольору та шоста – за короля того ж кольору. Аналогічно до 12 ознаки відповідають за фігури супротивника. Ознака приймає значення 1, якщо в даній клітинці знаходиться даний тип фігури, та 0 у протилежному випадку.

Як видно, в архітектурі другої нейронної мережі було видалено ознаки, що можуть заважати узагальненню нейронної мережі. Більше того, з огляду на процес навчання першої нейронної мережі було зроблено висновок, що краще представляти черговість ходу не однією окремою ознакою, а кодувати це в розташуванні фігур. Таким чином, змінена черговість ходу та віддзеркалення позиції зовсім не змінить вектор вхідних ознак на відміну від першої нейронної мережі, а значить і не змінить оцінку позиції.

Оскільки після такої модифікації неможливо відновити колір фігури, що стоїть на певному полі, неможливо натренувати нейронну мережу давати абсолютну оцінку позиції. Тому для даної нейромережі потрібно провести конвертацію вихідних ознак з абсолютних на відносні.

На прихованих шарах розташовано 512, 128, 32 та 1 нейрон відповідно. Останній шар є вихідним. На перших трьох прихованих шарах використовується функція активації ReLU, на вихідному – гіперболічний тангенс. Архітектуру нейронної мережі зображено на рисунку 2.2.

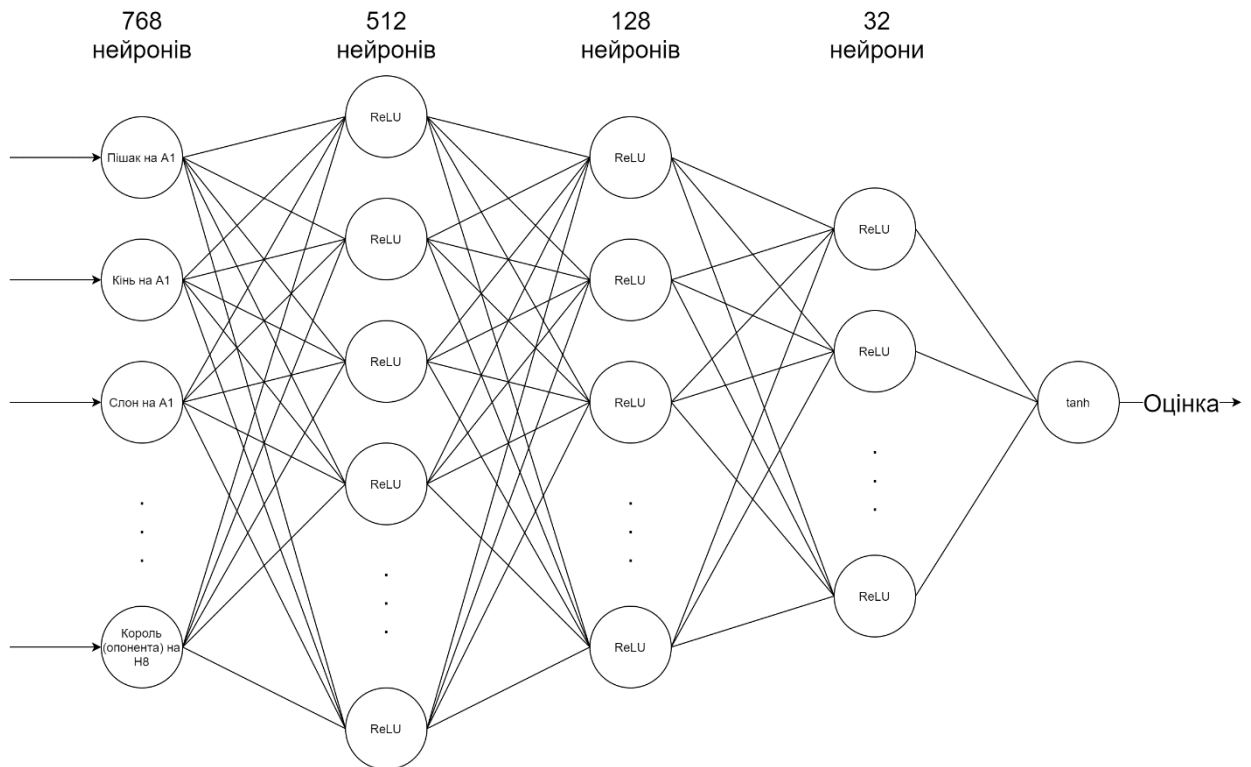


Рисунок 2.2 – Архітектура другої розробленої нейронної мережі

2.4 Висновки з розділу

У даному розділі було описано основні технології, які будуть використовуватися в процесі розробки, підготування датасету до процесу навчання нейромережі, архітектуру нейронної мережі та шахового рушія.

3 РОЗРОБКА ПРОГРАМИ

В даному розділі описано основні рішення щодо реалізації компонентів системи протягом розробки програми.

3.1 Розробка шахового рушія

В даному підрозділі описано спосіб поділу рушія на модулі, обрані методи реалізації методу представлення дошки, методу генерації ходів, методів оцінки позиції та пошуку найкращого ходу.

Код рушія було виконано мовою програмування C++. Незважаючи на те, що мова C++ підтримує об'єктно-орієнтоване програмування, головний модуль програми було побудовано без використання його основних концепцій, адже додаткові абстракції для динамічного поліморфізму витрачають додатковий час на виклик функцій, через що було зроблено акцент в бік продуктивності. Єдиний принцип ООП, що було використано при розробці – інкапсуляція, що полягає в пов'язуванні даних та методів, що над ними оперують.

Таким чином, при розробці було переважно зроблено акцент на конструкції структурного програмування, що є класичним підходом для алгоритмічних задач.

3.1.1 Поділ рушія на модулі

Оскільки рушій за технічним завданням має бути готовим до використання як у якості бібліотеки, так і у якості бінарного файлу, основні класи були винесені в ядро рушія – статичну бібліотеку. Рушій та його інтерфейс винесені в окремий модуль, який використовує ядро рушія як бібліотеку. Для тестування виконуваного коду було створено модуль з

тестами функціоналу ядра, що також використовує ядро рушія як бібліотеку. Взаємодія модулів проекту зображена на рисунку 3.1.

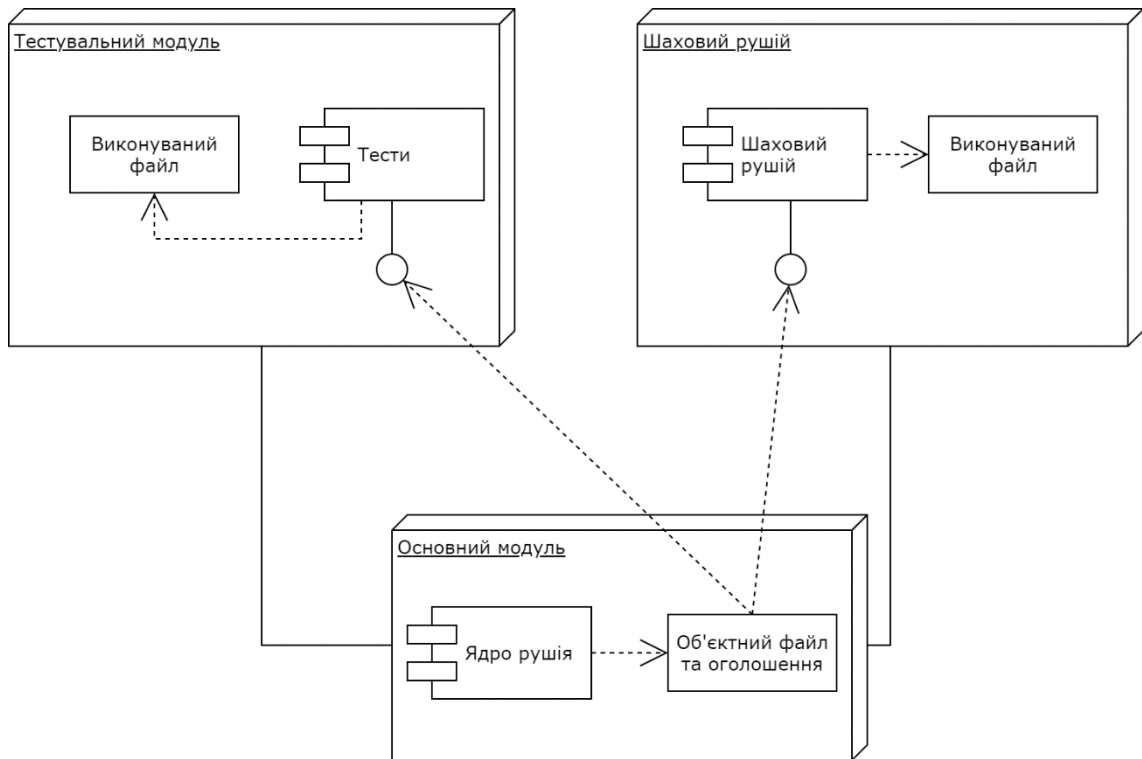


Рисунок 3.1 – Взаємодія модулів проекту

Кожен модуль має інтерфейс взаємодії та певні залежності.

Основний модуль залежить від бібліотек `cppflow` та `tensorflow`. Результатом компіляції коду бібліотеки є об'єктний файл, який разом з оголошеннями основних класів утворює інтерфейс, за допомогою якого можна використовувати бібліотеку.

Шаховий рушій залежить від основного модуля (ядра рушія). Результатом компіляції є виконуваний файл, інтерфейс взаємодії з яким детально описано в наступному розділі.

Тестовий модуль залежить від основного модуля (ядра рушія). Результатом компіляції є виконуваний файл, який виконує усі тести та повертає код 0 при успішному проходженні усіх тестів та інший код при неуспішному проходженні хоча б одного з них.

3.1.1.1 Розробка ядра рушія

До ядра рушія окрім основних складових рушія (представлення дошки, генерації ходів, пошуку найкращого ходу та оцінки позиції) прийнято рішення включити функціонал для роботи з FEN послідовностями та для тестування продуктивності двигуна. Основні класи, які наявні в ядрі рушія перелічені в таблиці 3.1

Таблиця 3.1 – Основні класи, наявні в ядрі рушія

Ім'я класу	Призначення класу
Position	Зберігання повного стану позиції, обробка зміни позиції, зберігання історії гри.
Bitboard	Зберігання бітової дошки та проведення основних операцій над нею.
Move	Зберігання інформації про тип ходу, стартове та кінцеве поля ходу.
Piece	Зберігання інформації про тип та колір фігури
MoveGenerator	Генерація ходів певного типу (статичний клас)
ClassicEvaluator	Класична оцінка позиції (статичний клас)
NeuralNetworkEvaluator	Зберігання нейронної мережі та оцінка позиції з її використанням.
Searcher	Зберігання інформації про дерево пошуку та пошук найкращого ходу
FenUtility	Конвертація FEN строки у позиції та навпаки (статичний клас)
PerfomanceTester	Пошук усіх позицій на певній глибині (статичний)

3.1.1.2 Розробка рушія

Рушій має отримувати команди шляхом стандартного вводу та надавати відповідь на запити через стандартний вивід. Для виконання користувацьких запитів рушій звертається до функцій ядра. Увесь функціонал з обробки запитів було поміщено в один клас.

Для дослідницьких цілей до складу рушія було включено класи для автономної гри в шахи. Класи, які входять до складу рушія перелічені в таблиці 3.2.

Таблиця 3.2 – Основні класи шахового рушія

Ім'я класу	Призначення класу
ChessEngine	Обробка користувацьких запитів
ChessGame	Підтримка стану гри та інформації про гравців
Player	Базовий клас гравця
ClassicEvaluationBot	Гра в шахи з використанням класичної оцінки позиції
NNEvaluationBot	Гра в шахи з використанням оцінки позиції за допомогою нейронної мережі

3.1.2 Вибір методу представлення дошки

Для представлення дошки в шаховому рушії було обрано гібридний метод представлення, що поєднує в собі переваги методів «Поштова скриня» та «Бітова дошка». Вибір було зроблено за наступними критеріями:

– швидкість роботи – перша й найвагоміша причина вибору. Гібридне представлення надає можливість швидко отримувати доступ як до конкретних фігур, так і до конкретних полів;

– операції над множинами полів – можливість виконувати основні операції над групами полів як над множинами;

– зручність використання у інших складових рушія. Деякі компоненти рушія зручніше реалізовувати саме з представленням «Поштова скриня» (наприклад, оцінці позиції), в той час як інші – з представленням «Бітова дошка» (наприклад, генерація ходів).

Класи та переліки, що використовуються в реалізації представлення позиції:

– Color – перелік, що використовується для представлення черговості ходу. Може приймати лише 2 значення (чорний та білий);

– Square – перелік, що використовується для ідентифікації поля та ітерації за полями дошки. Може приймати 64 значення (від A1 до H8). Дозволяє отримати горизонталь, вертикаль, діагональ та анти-діагональ поля;

– Direction – перелік, що використовується для позначення напрямів. Приймає 8 значень: 4 перпендикулярних (північ, південь, захід та схід) та 4 діагональних (північний захід, північний схід, південний захід та південний схід). Дозволяє отримувати відносний напрям для певного кольору;

– PieceType – перелік, що використовується для представлення типу фігури. Може приймати 7 значень (недійсний тип, пішак, кінь, слон, тура, ферзь та король);

– Piece – клас, що зберігає тип фігури та її колір;

– Bitboard – клас, що зберігає бітову множину полів (64-бітне слово). Містить бітові операції та основні операції для генерації ходів;

– Position – клас, що зберігає стан позиції (фігури, правила рокування, черговість ходу, історію позиції) та оброблює ходи і їх відміну;

– MoveType – перелік, що призначений для представлення типу ходу. Може примати декілька значень, серед яких (тихий, взяття, подвійний хід пішаком, рокування, взяття на проході, заміна пішака);

– Move – клас, що зберігає початкове та кінцеві поля ходу, а також зберігає тип ходу;

Для підтримки правил рокіровки використовується бітова дошка, що зберігає поля, на яких відбувалися рухи фігур. Ці поля порівнюються з відомими масками, таким чином визначаючи, чи можливо зробити конкретне рокірування.

3.1.3 Вибір методу генерації ходів

Для генерації ходів було обрано методи генерації ходів з представленням «Бітова дошка», який став де-факто стандартом в шахових рушіях за наступними причинами:

– продуктивність. Генерація ходів з представленням «поштова скриня» може наближатися до бітових дошок за продуктивністю лише з використанням додаткових структур даних, які так чи інакше можна замінити бітовими дошками;

– підтримка концепцій шаху та зв'язаних фігур при генерації ходів;

– проста генерація ходів різних типів.

Класи та переліки, що використовувалися в реалізації генерації ходів:

– більшість класів та переліків представлення дошки;

– MoveGenerator – статичний клас, що генерує легальні ходи певного типу;

– MoveList – клас, що зберігає згенеровані ходи. Створений для зручної ітерації за списком створених ходів та задля їх упорядкування.

Для генерації було заздалегідь підраховано наступні таблиці бітових дошок:

– масив бітових дошок для представлення горизонталей, вертикалей, діагоналей, анти-діагоналей;

- масив бітових дошок для представлення кожного поля (бітова дошка з одним встановленим бітом);
- масив бітових дошок для представлення полів між двома заданими полями;
- масив бітових дошок для представлення полів на лінії, заданій двома полями (ортогональні та діагональні).

Було реалізовано окремо генерацію ходів для ковзних фігур, пішаків, короля та коней.

Ходи короля мають оброблюватися першими, адже це дозволяє підрахувати кількість фігур, що дають шах. Якщо цих фігур 2, можливий тільки хід королем (адже не можна закритися від двох фігур чи взяти одразу 2 фігури).

Ковзні фігури не були розділені на ферзів, тур та слонів, а були розділені за іншим принципом, зручнішим для генерації: ортогональні ковзні фігури (тури та ферзі) та діагональні ковзні фігури (слони та ферзі). Завдяки цьому не потрібно було вирізняти ферзя як окрему фігуру та створювати для цього окремі обробки. Для генерації ходів ковзних фігур при запуску рушія потрібно заздалегідь підраховувати ходи для ковзних фігур (за методом магічних бітових дошок). Магічні числа для реалізації ідеального хешування між окупованими полями на прямих атак та бітовою дошкою полів атак було взято з шахового рушія SmallChess [40].

Ходи коней оброблюються та генеруються найлегше та не містять додаткових зауважень.

Ходи пішаків мають найбільше ситуацій для обробки, адже потрібно правильно згенерувати ходи для подвійного стартового ходу, взяття, взяття на проході, заміну пішака і навіть заміну пішака з взяттям. Для цього використовувалися бітові маски другої, четвертої та сьомої горизонталей (для виділення фігур на даних полях). Однією з найскладніших ситуацій для

обробки стала ситуація на рисунку 3.2. Взяття не є можливим через атаку тури, проте пішак не є зв'язаним.



Рисунок 3.2 – Одна з складних ситуацій для обробки

3.1.4 Вибір методу оцінки позиції

Для оцінки позиції було обрано як класичний метод, так і метод з використанням нейронних мереж для їх порівняння та висновків щодо їх ефективності.

Основні класи, що реалізують методи оцінки позиції:

- SimpleEvaluator – статичний клас, що оцінює позицію за матеріальним балансом фігур та за їх розташуванням у позиції;
- NeuralNetworkEvaluator – клас, що зберігає нейромережеву модель та обчислює оцінку позиції з її допомогою;

3.1.4.1 Реалізація класичного методу оцінки позиції

Для класичної оцінки позиції було завчасно призначено ціну в сантипішаках кожному типу фігури:

- пішак – 100 сантипішаків;
- кінь – 295 сантипішаків;
- слон – 300 сантипішаків;
- тура – 500 сантипішаків;
- ферзь – 900 сантипішаків;
- король – 0 сантипішаків.

Для кожного типу фігури створено таблиці нагороди за знаходження на конкретному полі. Значення для таблиць було взято з ресурсу chessprogramming.org [41] та дещо відредаговано емпіричним шляхом.

Реалізовано як абсолютну так і відносну оцінку позиції.

3.1.4.2 Реалізація методу оцінки позиції за допомогою нейронної мережі

Для оцінки позиції нейронною мережею було створено 2 нейронні мережі з різними архітектурами та навчено за допомогою датасету з майже 13 мільйонів позицій.

Позиція конвертується в масив бінарних ознак, який подається на вхід до нейронної мережі. Отриманий результат денормалізується та використовується як абсолютна оцінка позиції у випадку першої нейронної мережі та як відносна оцінка позиції у випадку другої нейронної мережі.

Після тестування нейронних мереж повинна залишитися та, яка дає кращі оцінки позиціям у більшості випадків.

3.1.5 Вибір методу пошуку найкращого ходу

Для вибору найкращого ходу було реалізовано негамакс пошук з наступними модифікаціями:

- альфа-бета відсічення;
- тихий пошук;
- ітеративне заглиблення;
- транспозиційні таблиці.

Для оптимізації роботи альфа-бета відсічення необхідно реалізувати упорядкування ходів, адже це впливає на кількість відсічених гілок у дереві пошуку. Для упорядкування ходів кожному ходу було призначено пріоритет, за яким ходи було відсортовано стандартним сортуванням в C++. Пріоритет вираховувався за наступними правилами:

- початковий пріоритет кожного ходу рівний нулю;
- якщо хід – взяття, пріоритет ходу має збільшитися на 10;
- якщо хід – заміна пішака, пріоритет ходу має збільшитися на 5;
- якщо хід – рокірування, пріоритет ходу має збільшитися на 1;
- якщо хід – тихий хід у зону атаки ворожого пішака, пріоритет ходу має зменшитися на 2;
- хід, визначений як найкращий минулою ітерацією ітеративного пошуку, має найвищий пріоритет.

У реалізації тихого пошуку розглядалися лише взяття.

Транспозиційні таблиці реалізовані за допомогою хешей Zobrista. Для вирахування хешей Zobrista для кожної позиції були згенеровані хеші для кожного типу фігури на кожній клітинці, для кожного поля взяття на проході, для кожного правила рокірування та для черговості ходу. Числа для таблиць згенеровані псевдовипадковим чином стандартним генератором mt19937 стандартної бібліотеки C++ при кожному старті програми.

3.2 Навчання нейронної мережі

Навчання нейронної мережі було здійснено за допомогою сервісу Google Colab [35], який надає доступ до хмарних обчислень за допомогою графічного процесора на 12 годин на добу.

3.2.1 Підготовка та поділ датасету

Датасет було взято на сервісі Kaggle [37]. Він складається з 12954834 оцінених позицій. Для конвертації FEN представлення дошки у бінарні ознаки для навчання нейромережі було використано модуль `python-chess` [42].

Конвертація кожного стовпця датасету описана в розділі 2.2.

Після конвертації ознак датасет було конвертовано у формат `.tfrecord`, який використовується для зберігання датасетів у фреймворці `tensorflow`.

Датасет було поділено на тестовий набір та перевірочний набір. До тестового набору увійшли 12 мільйонів позицій, до тренувального набору увійшли решта 954834 позицій.

Для навчання нейронної мережі датасет було перемішано та поділено на фрагменти, що складаються з 32 екземплярів кожен (для прискорення навчання).

Кожна епоха навчання займала від 30 хвилин до однієї години, через що не вдалося навчити нейронну мережу на великій кількості епох. Головними проблемами стали відсутність власного графічного процесора та вимога до постійної активності на сервісі Google Colab та відключення від хмарного середовища для обчислень у разі її відсутності за певний проміжок часу.

3.2.2 Результати навчання моделі

На рисунку 3.3 та рисунку 3.4 зображено графіки змін середньоквадратичної помилки за кількістю епох навчання першої та другої нейронних мереж.

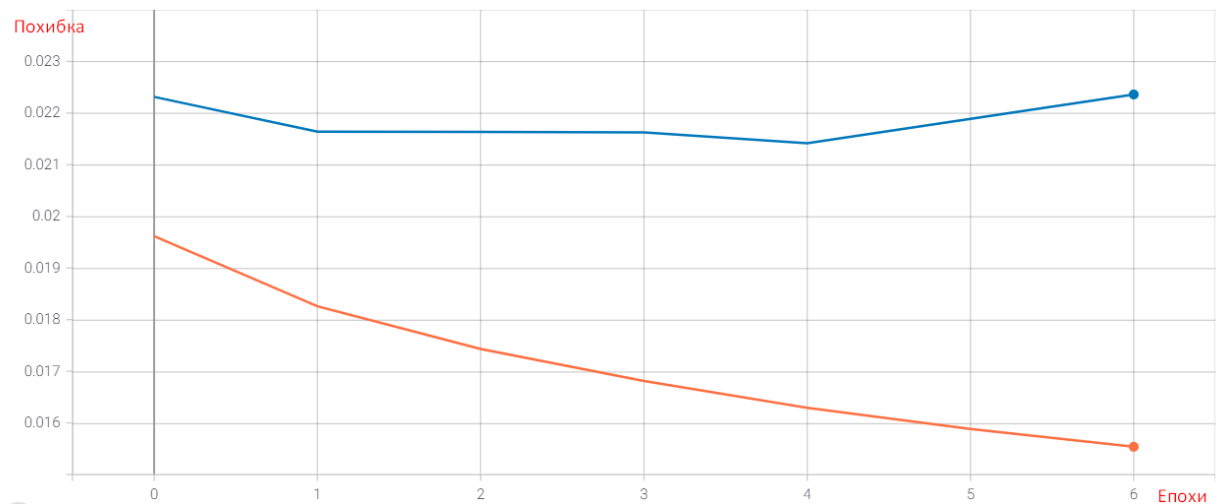


Рисунок 3.3 – Результати навчання першої нейронної мережі

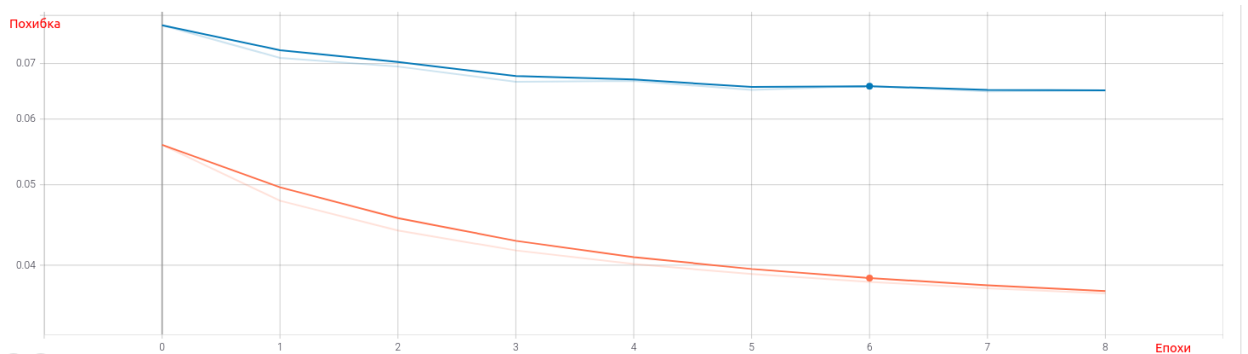


Рисунок 3.4 – Результати навчання другої нейронної мережі

Червоною лінією зображено значення середньоквадратичної похибки на тренувальному наборі, синьою – на тестувальному.

Як видно з графіку, перша нейронна мережа показує погані результати навчання. Середньоквадратична похибка для тренувального набору падає дуже повільно, а для тестувального не змінюється. Це свідчить

про те, що перша нейронна мережа не здатна до узагальнення, або здатна після великої кількості епох. Проте незмінний графік похибки на тестувальному наборі свідчить про те, що нейронна мережа перенавчається, тобто просто запам'ятовує екземпляри тренувального датасету.

Друга нейронна мережа показує кращі результати навчання. Як видно, середньоквадратична похибка зменшується як для тестувального, так і для тренувального наборів. Єдиним недоліком є замала швидкість навчання та зменшення середньоквадратичної похибки. Бажано залишити нейромережу навчатися на декілька десятків епох, проте через брак обчислювальних ресурсів цього зробити не вдалось.

Причинами великої різниці у успіху навчанні нейронних мереж виокремлено 2 фактори.

По-перше, уявимо будь-яку позицію, в якій від черговості ходу залежить перевага. Тобто, якщо хід чорних, то перевага на боці чорних, якщо хід білих, то перевага на боці білих. Для першої нейронної мережі різниця у черговості ходу виражається лише у одній ознаці, в той час як для другої нейронної мережі зміняться усі ознаки, що рівні 1.

Це саме твердження працює і для оберненого випадку: для дзеркальних позицій (позицій, у яких змінено колір фігур та черговість ходу на протилежний) оцінка має бути рівною. У випадку першої нейронної мережі, дзеркальні позиції означають зміну усіх ознак, що відповідають за розташування фігур, в той час як для другої нейронної мережі дзеркальні позиції представлені абсолютно однаковим вектором ознак.

По-друге, спроба надати нейронній мережі повну інформацію про позицію, а саме інформацію про рокірування та поля для взяття на проході, може заважати нейронній мережі узагальнюватися, адже іноді взяття на проході є лише шумом і ходом, який у позиції лише втратить перевагу, а іноді взяття на проході є єдиним ходом, що не програє. Через цю варіативність нейронній мережі краще надавати інформацію лише про розташування фігур, що і зроблено у другій нейронній мережі.

3.3 Висновки з розділу

У даному розділі було розглянуто основні рішення, прийняті при розробці програми. Розглянуто розробку рушія та його модулів, навчання нейронної мережі, проблеми, що виникли при навчанні нейронної мережі та зроблені висновки.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

В даному розділі описане застосування програми, її експлуатація, методика та результати тестування а також практичне застосування програми для вирішення поставленої проблеми.

4.1 Інструкція по експлуатації програми

Як вже було сказано, рушій зазвичай використовується в серверній частині або ж шаховими графічними інтерфейсами через інтерфейс командної строки. За технічним завданням, рушій має оброблювати наступні запити:

- встановлення певної позиції у форматі FEN;
- генерація можливих ходів;
- пошук найкращого ходу в заданій позиції на заданій глибині використовуючи класичну оцінку позиції;
- пошук найкращого ходу в заданій позиції на заданій глибині використовуючи оцінку позиції за допомогою нейронної мережі;
- підрахунок позицій, що можуть бути досягнені з заданої позиції на певній глибині пошуку;
- підрахунок позицій, що можуть бути досягнені з заданої позиції на певній глибині пошуку за кожним ходом поточної позиції.

Для тестування рушія було додано запит, що візуалізує поточну позицію в консолі. Текстова команда «d» (така сама, як і в більшості шахових рушіїв) візуалізує позицію в консолі (рисунок 4.1).

```
Engine started!

d
  A B C D E F G H
+---+---+---+---+---+---+
8 | r | n | b | q | k | b | n | r | 8
+---+---+---+---+---+---+
7 | p | p | p | p | p | p | p | p | 7
+---+---+---+---+---+---+
6 | - | - | - | - | - | - | - | - | 6
+---+---+---+---+---+---+
5 | - | - | - | - | - | - | - | - | 5
+---+---+---+---+---+---+
4 | - | - | - | - | - | - | - | - | 4
+---+---+---+---+---+---+
3 | - | - | - | - | - | - | - | - | 3
+---+---+---+---+---+---+
2 | P | P | P | P | P | P | P | P | 2
+---+---+---+---+---+---+
1 | R | N | B | Q | K | B | N | R | 1
+---+---+---+---+---+---+
  A B C D E F G H

FEN: rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
Hash: 0x7d1ad1f45cac80aa

Time taken: 0.000395
```

Рисунок 4.1 – Демонстрація обробки запиту «d»

Встановлення певної позиції у форматі FEN виконується за допомогою команди «position FEN», де замість FEN має йти представлення дошки в форматі FEN. На рисунку 4.2 продемонстровано обробку запиту «position r4rk1/ppqbbpp1/2p1nn1p/P1Ppp3/BP2P3/3PBN1P/3NQPP1/R4RK1 w Qq - 0 1»

```
position r4rk1/ppqbbpp1/2p1nn1p/P1Ppp3/BP2P3/3PBN1P/3NQPP1/R4RK1 w Qq - 0 1
Time taken: 8e-06

d
  A B C D E F G H
+---+---+---+---+---+---+
8 | r | - | - | - | - | r | k | - | 8
+---+---+---+---+---+---+
7 | p | p | q | b | b | p | p | - | 7
+---+---+---+---+---+---+
6 | - | - | p | - | n | n | - | p | 6
+---+---+---+---+---+---+
5 | P | - | P | p | p | - | - | - | 5
+---+---+---+---+---+---+
4 | B | P | - | - | P | - | - | - | 4
+---+---+---+---+---+---+
3 | - | - | - | P | B | N | - | P | 3
+---+---+---+---+---+---+
2 | - | - | - | N | Q | P | P | - | 2
+---+---+---+---+---+---+
1 | R | - | - | - | - | R | K | - | 1
+---+---+---+---+---+---+
  A B C D E F G H

FEN: r4rk1/ppqbbpp1/2p1nn1p/P1Ppp3/BP2P3/3PBN1P/3NQPP1/R4RK1 w Qq - 0 1
Hash: 0x901b21743d7b86f1

Time taken: 0.000128
```

Рисунок 4.2 – Демонстрація обробки запиту «position ...»

Генерація можливих ходів у встановленій позиції відбувається за допомогою команди «generate». Кожний знайдений легальний хід у позиції виведено у форматі «Початкова клітинка Кінцева клітинка». У випадку заміни пішака, п'ятий додатковий символ показує, на яку фігуру потрібно замінити пішака. На рисунку 4.3 продемонстровано обробку запиту «generate».

```
generate
g1h1
g1h2
e2d1
e3h6
e3d4
e3f4
e3g5
a4c6
a4d1
a4c2
a4b3
a4b5
a1b1
a1c1
a1d1
a1e1
a1a2
a1a3
f1b1
f1c1
f1d1
f1e1
e2e1
d2b1
d2b3
d2c4
f3e5
f3e1
f3h2
f3d4
f3h4
f3g5
g2g3
d3d4
h3h4
b4b5
e4d5
a5a6
g2g4
Time taken: 0.000144
```

Рисунок 4.3 – Демонстрація обробки запиту «generate»

Пошук найкращого ходу у встановленій позиції на заданій глибині використовуючи класичну оцінку позиції відбувається за допомогою команди «find N». Замість N потрібно зазначити невід’ємне ціле, що задає глибину пошуку. Результатом обробки запиту є найкращий хід та відносна оцінка позиції за умови виконання знайденого ходу. На рисунку 4.4 продемонстровано обробку запиту «find 6».

```
find 6
Best move: e2e4
Evaluation: 0
Max depth with quiet search: 30
Time taken: 0.504374
```

Рисунок 4.4 – Демонстрація обробки запиту «find ...»

Пошук найкращого ходу у встановленій позиції на заданій глибині використовуючи оцінку позиції за допомогою нейронної мережі відбувається за допомогою команди «findnn N». Замість N потрібно зазначити невід’ємне ціле, що задає глибину пошуку. Результатом обробки запиту є найкращий хід та відносна оцінка позиції за умови виконання знайденого ходу. На рисунку 4.5 продемонстровано обробку запиту «findnn 3».

```
findnn 3
Best move: c2c3
Evaluation: -1362
Max depth with quiet search: 6
Time taken: 0.887992
```

Рисунок 4.5 – Демонстрація обробки запиту «findnn ...»

Підрахунок позицій, що можуть бути досягнені з встановленої позиції на певній глибині пошуку відбувається за допомогою команди «perft N» (так само як і в більшості шахових рушіїв). Замість N потрібно зазначити невід’ємне ціле, що задає глибину пошуку. Загалом дана команда використовується для тестування продуктивності рушія. Результатом обробки запиту є кількість вузлів (вершин у графі пошуку, або ж позицій), що досягаються з поточної позиції. На рисунку 4.6 продемонстровано обробку запиту «perft 6» з початкової позиції.

```
perft 6
Total nodes: 119060324
Time taken: 2.40625
```

Рисунок 4.6 – Демонстрація обробки запиту «perft ...»

Підрахунок позицій, що можуть бути досягнені з заданої позиції на певній глибині пошуку за кожним ходом поточної позиції відбувається за допомогою команди «perftdiv N». Замість N потрібно зазначити невід’ємне ціле, що задає глибину пошуку. Результатом обробки запиту є список ходів у позиції та кількість вузлів, що досягається після виконання певного ходу. На рисунку 4.7 продемонстровано обробку запиту «perftdiv 6» з початкової позиції.

```
perftdiv 6
b1a3: 4856835
b1c3: 5708064
g1f3: 5723523
g1h3: 4877234
a2a3: 4463267
b2b3: 5310358
c2c3: 5417640
d2d3: 8073082
e2e3: 9726018
f2f3: 4404141
g2g3: 5346260
h2h3: 4463070
a2a4: 5363555
b2b4: 5293555
c2c4: 5866666
d2d4: 8879566
e2e4: 9771632
f2f4: 4890429
g2g4: 5239875
h2h4: 5385554
Total nodes: 119060324
Time taken: 2.52819
```

Рисунок 4.7 – Демонстрація обробки запиту «perftdiv ...»

Для завершення взаємодії з рушієм та закінчення роботи програми потрібно ввести команду «exit». На рисунку 4.8 продемонстровано обробку запиту «exit»

```
exit
Shutting down
Press <RETURN> to close this window...
```

Рисунок 4.8 – Демонстрація обробки запиту «exit»

4.2 Аналіз отриманих результатів

В даному підрозділі проаналізовано швидкість та точність роботи рушія.

4.2.1 Тестування коректності реалізації основних модулів

Було проведено тестування представлення позиції (а також виконання ходу та його відміну в позиції), генерація ходів та побудова графу пошуку. Точний аналіз оцінки позиції провести неможливо, адже оцінка позиції – евристична функція.

Тестування відбувається наступним чином: задається глибина графу пошуку, в позиції генеруються легальні ходи, модифікується представлення дошки виконанням ходу, рекурсивно будується граф пошуку на глибину 1 менше, ніж для поточної позиції, модифікується представлення дошки відміною ходу. Таким чином тестуються одразу 3 модулі. Отримані результати у вигляді кількості досягнутих позицій порівнюються з результатами виконання `perft` команд на проблематичних позиціях іншими рушіями. Проблематичні позиції та підраховані для них кількості досягнутих вузлів були знайдені на ресурсі chessprogramming.org [43].

Для усіх невідповідностей було знайдено проблематичний хід за допомогою команди `perftdiv` та порівняння результатів розробленого рушія з результатами рушія Stockfish. Таким чином було знайдено та виправлено більшість помилок у генерації ходів та оновленні представлення позиції в рушії.

Оскільки кожна зміна у шаховому рушії може призвести до помилки, у тестувальний модуль було додано тестування на шести позиціях на різній глибині. Дане тестування виконувалося кожного разу, коли було змінено або ж генерацію ходів, або ж представлення дошки. На рисунку 4.9 зображено результати тестування на шести позиціях на різній глибині.

```

Terminal
[ OK ] Position3/PerftPosition3.AtDepth/6 (279 ms)
[ RUN ] Position3/PerftPosition3.AtDepth/7
[ OK ] Position3/PerftPosition3.AtDepth/7 (4088 ms)
[-----] 8 tests from Position3/PerftPosition3 (4392 ms total)

[-----] 6 tests from Position4/PerftPosition4
[ RUN ] Position4/PerftPosition4.AtDepth/0
[ OK ] Position4/PerftPosition4.AtDepth/0 (0 ms)
[ RUN ] Position4/PerftPosition4.AtDepth/1
[ OK ] Position4/PerftPosition4.AtDepth/1 (0 ms)
[ RUN ] Position4/PerftPosition4.AtDepth/2
[ OK ] Position4/PerftPosition4.AtDepth/2 (0 ms)
[ RUN ] Position4/PerftPosition4.AtDepth/3
[ OK ] Position4/PerftPosition4.AtDepth/3 (0 ms)
[ RUN ] Position4/PerftPosition4.AtDepth/4
[ OK ] Position4/PerftPosition4.AtDepth/4 (5 ms)
[ RUN ] Position4/PerftPosition4.AtDepth/5
[ OK ] Position4/PerftPosition4.AtDepth/5 (240 ms)
[-----] 6 tests from Position4/PerftPosition4 (245 ms total)

[-----] 6 tests from Position5/PerftPosition5
[ RUN ] Position5/PerftPosition5.AtDepth/0
[ OK ] Position5/PerftPosition5.AtDepth/0 (0 ms)
[ RUN ] Position5/PerftPosition5.AtDepth/1
[ OK ] Position5/PerftPosition5.AtDepth/1 (0 ms)
[ RUN ] Position5/PerftPosition5.AtDepth/2
[ OK ] Position5/PerftPosition5.AtDepth/2 (0 ms)
[ RUN ] Position5/PerftPosition5.AtDepth/3
[ OK ] Position5/PerftPosition5.AtDepth/3 (1 ms)
[ RUN ] Position5/PerftPosition5.AtDepth/4
[ OK ] Position5/PerftPosition5.AtDepth/4 (33 ms)
[ RUN ] Position5/PerftPosition5.AtDepth/5
[ OK ] Position5/PerftPosition5.AtDepth/5 (1124 ms)
[-----] 6 tests from Position5/PerftPosition5 (1158 ms total)

[-----] 6 tests from Position6/PerftPosition6
[ RUN ] Position6/PerftPosition6.AtDepth/0
[ OK ] Position6/PerftPosition6.AtDepth/0 (0 ms)
[ RUN ] Position6/PerftPosition6.AtDepth/1
[ OK ] Position6/PerftPosition6.AtDepth/1 (0 ms)
[ RUN ] Position6/PerftPosition6.AtDepth/2
[ OK ] Position6/PerftPosition6.AtDepth/2 (0 ms)
[ RUN ] Position6/PerftPosition6.AtDepth/3
[ OK ] Position6/PerftPosition6.AtDepth/3 (1 ms)
[ RUN ] Position6/PerftPosition6.AtDepth/4
[ OK ] Position6/PerftPosition6.AtDepth/4 (49 ms)
[ RUN ] Position6/PerftPosition6.AtDepth/5
[ OK ] Position6/PerftPosition6.AtDepth/5 (2068 ms)
[-----] 6 tests from Position6/PerftPosition6 (2118 ms total)

[-----] Global test environment tear-down
[=====] 39 tests from 6 test suites ran. (13578 ms total)
[ PASSED ] 39 tests.

```

Рисунок 4.9 – Результати тестування програми

Таким чином також перевіряється продуктивність шахового рушія. На рисунку 4.10 зображено час виконання команди `perft` (скорочення від Performance Test) на стартовій позиції на ноутбуці з процесором Intel Core I3-6006 на різних глибинах.

```
[-----] 7 tests from StartingPosition/PerftStartingPosition
[ RUN      ] StartingPosition/PerftStartingPosition.AtDepth/0
[ OK       ] StartingPosition/PerftStartingPosition.AtDepth/0 (0 ms)
[ RUN      ] StartingPosition/PerftStartingPosition.AtDepth/1
[ OK       ] StartingPosition/PerftStartingPosition.AtDepth/1 (0 ms)
[ RUN      ] StartingPosition/PerftStartingPosition.AtDepth/2
[ OK       ] StartingPosition/PerftStartingPosition.AtDepth/2 (0 ms)
[ RUN      ] StartingPosition/PerftStartingPosition.AtDepth/3
[ OK       ] StartingPosition/PerftStartingPosition.AtDepth/3 (0 ms)
[ RUN      ] StartingPosition/PerftStartingPosition.AtDepth/4
[ OK       ] StartingPosition/PerftStartingPosition.AtDepth/4 (12 ms)
[ RUN      ] StartingPosition/PerftStartingPosition.AtDepth/5
[ OK       ] StartingPosition/PerftStartingPosition.AtDepth/5 (120 ms)
[ RUN      ] StartingPosition/PerftStartingPosition.AtDepth/6
[ OK       ] StartingPosition/PerftStartingPosition.AtDepth/6 (2553 ms)
```

Рисунок 4.10 – Результати тестування продуктивності рушія

4.2.2 Порівняння розробленого рушія із готовими рішеннями

Головними показниками якості роботи рушія є точність оцінки позиції та швидкість пошуку. Оскільки оцінка позиції – евристична функція і точні критерії для її порівняння виділити важно, порівняймо час пошуку розробленого рушія (з різними методами оцінки позиції) з рушієм Stockfish.

В таблиці 4.1 представлене порівняння часу витраченого на різні глибини пошуку зі стартової позиції розробленого шахового рушія із шаховим рушієм Stockfish.

Таблиця 4.1 – Порівняння глибини пошуку різних рушіїв

Рухий Глибина пошуку	Розроблений рушій (класична оцінка позиції)	Розроблений рушій (нейронна оцінка позиції)	Stockfish
1	2	3	4
1	< 1с	< 1с	< 1с
2	< 1с	< 1с	< 1с
3	< 1с	1.01 с	< 1с
4	< 1с	4.19 с	< 1с

Продовження таблиці 4.1

1	2	3	4
5	< 1с	42.06 с	< 1с
6	< 1с	-	< 1с
7	1.26 с	-	< 1с
8	10.6 с	-	< 1с
9	126.984 с	-	< 1с
10	-	-	< 1с
12	-	-	< 1с
16	-	-	< 1с
17	-	-	1.01 с
18	-	-	1.17 с
19	-	-	1.64 с
20	-	-	2.25 с
21	-	-	3.19 с

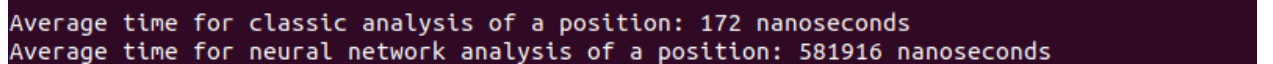
Загалом розроблений рушій дуже сильно програє готовим рішенням через брак реалізацій деяких концепцій, серед яких:

- довгий пошук через відсутність комплексного евристичного алгоритму з упорядкування ходів;
- довгий пошук через відсутність концепції варіацій, що дозволяють зберігати не лише найкращий хід з минулої ітерації пошуку, а повну найкращу варіацію;
- відсутність концепції паралельного пошуку на декількох ядрах процесора;
- відсутність окремих методів оцінки позиції для дебюту, мітельшпілю, ендшпілю;
- відсутність підтримки бази дебютів в рушії;
- слабка підтримка траснпозиційних таблиць;

- незавершене навчання нейронної мережі в рушії;
- неповне покриття усіх розрахунків, що можуть бути зроблені на етапі компіляції (магічні бітові дошки, хеші Zobrista).

4.2.3 Порівняння класичного методу оцінки позиції та за допомогою нейронної мережі

Порівняймо час оцінки позиції двома методами. На ноутбучі з процесором Intel Core I3-6006 середній час класичним методом оцінки позиції становить 172 наносекунди, методом з нейронною мережею – 581916 наносекунд. Можна зробити висновок, що аналіз позиції за допомогою нейронної мережі займає в 3350 разів більше часу, що значно зменшує глибину графу пошуку за той же час (в порівнянні з класичним методом оцінки позиції) та кількість оцінених позицій (рисунок 4.11).



```
Average time for classic analysis of a position: 172 nanoseconds
Average time for neural network analysis of a position: 581916 nanoseconds
```

Рисунок 4.11 – Середній час оцінки позиції двома методами

Для порівняння роботи рушія з використанням класичного методу оцінки позиції та з використанням нейронної мережі було створено 2 агенти для гри в шахи. Було проведено по одній грі між агентами на різній глибині пошуку. Немає сенсу проводити більше однієї гри між однаковими агентами, адже результат ігор не зміниться. Результати проведених ігор між агентами представлено в таблиці 4.2

Також було проведено декілька ігор між агентами на людину. Людині з аматорським рівнем гри в шахи (близько 1500 ELO на онлайн платформах) майже завжди вдається виграти агенту з оцінкою позиції нейронною мережею, в той час як агент з класичною оцінкою позиції створює хоча б якусь конкуренцію. Приблизно у чверті випадків людині

вдається обіграти останнього агента, половина ігор закінчується поразкою людини, та залишок припадає на нічії. Статистичні дані дуже приблизні оскільки не було проведено більше 50 ігор між людиною та рушієм.

Таблиця 4.2 – Результати ігор між агентами

Параметри гри Номер гри	Агент, що грає за білих	Агент, що грає за чорних	Результат гри
1	Класичний на глибині 1	Нейронний на глибині 1	Перемога білих
2	Нейронний на глибині 1	Класичний на глибині 1	Нічия
3	Класичний на глибині 2	Нейронний на глибині 2	Перемога білих
4	Нейронний на глибині 2	Класичний на глибині 2	Перемога чорних
5	Класичний на глибині 2	Нейронний на глибині 3	Перемога білих
6	Класичний на глибині 1	Нейронний на глибині 3	Нічия
7	Нейронний на глибині 3	Класичний на глибині 2	Нічия
8	Нейронний на глибині 3	Класичний на глибині 1	Нічия

Агент з класичною оцінкою позиції завжди виграв агента з оцінкою позиції нейронною мережею, якщо глибина пошуку у класичного агента більше. Це свідчить про те, що нейронна мережа не натренована в повній мірі.

4.3 Практична цінність роботи

За результатами роботи програми зрозуміло, що розроблений програмний продукт програє існуючим аналогам за швидкістю пошуку та точністю оцінки у випадку використання оцінки позиції за допомогою нейронної мережі. Це пов'язано як з алгоритмічною складністю рішень існуючих аналогів, так і з кількістю розробників, що працюють над ними. Попри такі значні недоліки у розробленій програмі існує і ряд переваг:

- можливість змінювати архітектуру нейронної мережі (окрім структур вхідного та вихідного шарів) без зміни програмного коду або повторної компіляції виконуваного файлу;
- можливість обирати метод оцінки позиції.

Також, практична цінність роботи полягає в дослідженні методів представлення шахової позиції у вигляді вектору ознак. Було винайдено спосіб з відображенням черговості ходу для покращення узагальнюючих можливостей нейронної мережі.

4.4 Висновки з розділу

З останніх підрозділів очевидно, що класична оцінка позиції дає точніші результати за оцінку позиції за допомогою нейронної мережі. Це може відбуватися за наступними причинами:

- нестача ресурсів для навчання нейронної мережі. Для навчання нейронної мережі рушію Stockfish більше тисячі людей з усього світу тестують різні методики навчання та надають доступ до своїх процесорів та графічних процесорів для прискорення навчання;
- репрезентативність датасету. Для навчання було використано як дебютні позиції, так і мітельшпілі і ендшпілі. Можливо, потрібно було

навчати нейронну мережу лише для оцінки позицій певного етапу гри, адже в просунутих рушійх класичні методи оцінки різних етапів гри відрізняються;

- замалий вхідний шар. Вхідний шар у Stockfish NNUE містить більше 80000 тисяч нейронів. Подібний багатошаровий персептрон вимагав би великої кількості ресурсів для навчання та оцінки позиції, через що швидкість пошуку в рушії зменшилась би на декілька порядків;

- завелика точність оцінки позицій в датасеті. Для тренування розробленої нейронної мережі використовувались оцінки позицій рушієм Stockfish на глибині 22, в той час як для тренування Stockfish NNUE використовувалися позиції на помірній глибині пошуку [44].

ВИСНОВКИ

Під час виконання кваліфікаційної роботи бакалавра було розроблено програмну реалізацію шахового рушія, який використовує представлення «бітова дошка», генерує легальні ходи використовуючи це представлення, виконує пошук з модифікаціями алгоритму негамакс та альфа-бета відсічення та реалізує аналіз шахової позиції як класичним методом, так і методом, що використовує нейронну мережу.

Програмний продукт дозволяє виконати наступні операції:

- знайти найкращий хід у заданій позиції використовуючи класичний метод оцінки позиції;
- знайти найкращий хід у заданій позиції використовуючи метод оцінки позиції з нейронною мережею;
- згенерувати можливі ходи у заданій позиції;

Були успішно виконані наступні завдання:

- реалізовано представлення дошки;
- реалізовано генерацію ходів;
- реалізовано класичну оцінку позиції;
- реалізовано алгоритм пошуку;
- досліджено способи представлення дошки у вигляді вектору ознак;
- створено нейронну мережу для аналізу цінності позиції.

Розроблений програмний продукт відповідає усім умовам технічного завдання.

У розробленому програмному продукті є недоліки, які було проаналізовано. Реалізації пошуку найкращого ходу та оцінки позиції з використанням нейронної мережі не показали себе конкурентоспроможними з відомими аналогами за наступними причинами:

- відсутність реалізації паралельного пошуку;
- відсутність різних методів оцінки позиції для різних етапів гри;

- відсутність обчислювальних ресурсів для повного навчання різних нейронних мереж та їх порівняння;

- відсутність підтримки нейронних мереж, що ефективно оновлюються в основних фреймворках для роботи з глибоким навчанням через їх новизну.

Попри такі значні недоліки у розробленій програмі існує і ряд переваг:

- можливість змінювати архітектуру нейронної мережі (окрім структур вхідного та вихідного шарів) без зміни програмного коду або повторної компіляції виконуваного файлу;

- можливість обирати метод оцінки позиції.

Також, практична цінність роботи полягає в дослідженні методів представлення шахової позиції у вигляді вектору ознак. Було винайдено спосіб з відображенням черговості ходу для покращення узагальнюючих можливостей нейронної мережі.

Можливими подальшими шляхами розвитку програмного продукту є:

- тренування нейронної мережі на більшій кількості епох;
- реалізація та оптимізація паралельного пошуку найкращого ходу;
- оптимізація роботи з транспозиційними таблицями;
- зменшення точності оцінки позиції в датасеті (провести оцінку на нижчій глибині);
- застосування різних методів оцінки позиції для різних етапів гри;
- використання нейронної мережі, що ефективно оновлюється замість персептрону.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Computer chess [Electronic resource]. – Access mode : https://en.wikipedia.org/wiki/Computer_chess
2. Chess engine [Electronic resource]. – Access mode : https://en.wikipedia.org/wiki/Chess_engine
3. Board representation (computer chess) [Electronic resource]. – Access mode : [https://en.wikipedia.org/wiki/Board_representation_\(computer_chess\)](https://en.wikipedia.org/wiki/Board_representation_(computer_chess))
4. Нейронні мережі : теорія та практика: навч. посіб. / С. О. Субботін. – Житомир : Вид. О. О. Євенок, 2020. – 184 с.
5. Rules of chess [Electronic resource]. – Access mode : https://en.wikipedia.org/wiki/Rules_of_chess
6. Forsyth-Edwards Notation [Electronic resource]. – Access mode : https://www.chessprogramming.org/Forsyth-Edwards_Notation
7. Редактор дошки для моделювання позицій [Електронний ресурс]. – Режим доступу : <https://lichess.org/editor>
8. Mailbox board representation technique [Electronic resource]. – Access mode : <https://www.chessprogramming.org/Mailbox>
9. 8x8 board mailbox representation technique [Electronic resource]. – Access mode : https://www.chessprogramming.org/8x8_Board
10. 10x12 board mailbox representation technique [Electronic resource]. – Access mode : https://www.chessprogramming.org/10x12_Board
11. 0x88 board mailbox representation technique [Electronic resource]. – Access mode : <https://www.chessprogramming.org/0x88>
12. Bitboards (board representation technique) [Electronic resource]. – Access mode : <https://www.chessprogramming.org/Bitboards>
13. Leiserson C. E. Using de Bruijn Sequences to Index a 1 in a Computer Word [Text] / C. E. Leiserson, H. Prokop, K. H. Randall. – MIT Laboratory for Computer Science, Cambridge, 1998. – 8 p.

14. Least significant bit index search [Electronic resource]. – Access mode : <https://www.chessprogramming.org/BitScan>
15. Population count in a 64-bit computer word [Electronic resource]. – Access mode : https://www.chessprogramming.org/Population_Count
16. Move generation techniques in chess engines [Electronic resource]. – Access mode : https://www.chessprogramming.org/Move_Generation
17. Sliding piece attacks generation using bitboards [Electronic resource]. – Access mode : https://www.chessprogramming.org/Sliding_Piece_Attacks
18. Magic bitboards technique for sliding piece attacks generation [Electronic resource]. – Access mode : https://www.chessprogramming.org/Magic_Bitboards
19. Hyperbola quintessence technique for sliding piece attacks generation [Electronic resource]. – Access mode : https://www.chessprogramming.org/Hyperbola_Quintessence
20. Chess position evaluation techniques [Electronic resource]. – Access mode : <https://www.chessprogramming.org/Evaluation>
21. Neural networks in chess engines [Electronic resource]. – Access mode : https://www.chessprogramming.org/Neural_Networks
22. Efficiently updatable neural network [Electronic resource]. – Access mode : https://en.wikipedia.org/wiki/Efficiently_updatable_neural_network
23. Stockfish efficiently updatable neural network [Electronic resource]. – Access mode : https://www.chessprogramming.org/Stockfish_NNUE
24. Minimax algorithm in game theory [Electronic resource]. – Access mode : <https://www.chessprogramming.org/Minimax>
25. Negamax algorithm in chess search [Electronic resource]. – Access mode : <https://www.chessprogramming.org/Negamax>
26. Alpha-Beta pruning algorithm in chess search [Electronic resource]. – Access mode : <https://www.chessprogramming.org/Alpha-Beta>
27. Quiescence search enhancement of chess search [Electronic resource]. – Access mode : https://www.chessprogramming.org/Quiescence_Search

28. Transposition table technique in chess search [Electronic resource]. – Access mode : https://www.chessprogramming.org/Transposition_Table
29. Incremental updates in chess position [Electronic resource]. – Access mode : https://www.chessprogramming.org/Incremental_Updates
30. Zobrist hashing for transposition tables [Electronic resource]. -Access mode : https://www.chessprogramming.org/Zobrist_Hashing
31. Iterative deepening technique in chess search [Electronic resource]. – Access mode : https://www.chessprogramming.org/Iterative_Deepening
32. googletest C++ testing and mocking framework [Electronic resource]. – Access mode : <https://github.com/google/googletest>
33. Tensorflow – framework for deep learning [Electronic resource]. – Access mode : <https://www.tensorflow.org/>
34. Cppflow header-only library for tensorflow model running [Electronic resource]. – Access mode : <https://github.com/serizba/cppflow>
35. Google Colab [Electronic resource]. – Access mode : <https://colab.research.google.com/>
36. CMake – an open-source builder tool [Electronic resource]. – Access mode : <https://cmake.org/>
37. Dataset of evaluated chess positions [Electronic resource]. – Access mode : <https://www.kaggle.com/ronakbadhe/chess-evaluations>
38. Stockfish – open-source chess engine [Electronic resource]. – Access mode : <https://stockfishchess.org/>
39. Sabatelli M. Learning to Evaluate Chess Positions with Deep Neural Networks and Limited Lookahead [Text] / M. Sabatelli, F. Bidoia, V. Codreanu. – Montefiore Institute, 2018. – 9 p.
40. Magic bitboard precalculated magic numbers from SmallChess engine [Electronic resource]. – Access mode : <http://smallchess.com/MagicMoves.zip>
41. Simplified evaluation function [Electronic resource]. – Access mode : https://www.chessprogramming.org/Simplified_Evaluation_Function

42. Python-chess library for Python [Electronic resource]. – Access mode : <https://python-chess.readthedocs.io/en/latest/index.html>

43. Precalculated perft results for move generation testing [Electronic resource]. – Access mode : https://www.chessprogramming.org/Perft_Results

44. Introducing NNUE Evaluation in Stockfish [Electronic resource]. – Access mode : <https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/>

ДОДАТОК А

Технічне завдання

A.1 Вступ

В даному розділі описано назву програми, її призначення та область застосування.

A.1.1 Назва програми

Програма називається «SimpleChessEngine».

A.1.2 Призначення та область застосування

Програма призначена для підтримування стану позиції, генерації ходів в заданій позиції, аналізу і позиції та пошуку найкращого ходу.

Програма представляє консольний інтерфейс і застосовується як допоміжний модуль на серверній частині чи як рушій для шахового графічного користувацького інтерфейсу.

A.2 Підстави для розробки

Підставою для розробки є завдання на диплому роботу на тему «Програмна реалізація шахового рушія з використанням технології глибокого навчання нейромережових моделей».

Актуальність роботи полягає в дослідженні шахових рушіїв та застосування нейронних мереж у їх складі для аналізу цінності позиції, оскільки створення власної евристичної функції з великою кількістю параметрів не є простою задачею.

Також актуальність полягає в дослідженні способів представлення шахової позиції у вигляді вектору ознак.

А.3 Призначення розробки

Програмний продукт реалізовує представлення шахової позиції, надає можливості для її зміни, генерації легальних ходів, оцінки позиції класичним методом та з використанням нейронної мережі а також аналізує позицію і знаходить найкращий хід.

А.4 Вимоги до програми

В даному розділі описані вимоги до програмного продукту.

А.4.1 Вимоги до функціональних характеристик

Програмний продукт повинен забезпечувати виконання наступних функцій:

- користувач повинен мати можливість встановлювати певну позицію в шаховому рушії;
- користувач повинен мати можливість згенерувати легальні ходи в позиції;
- користувач повинен мати можливість знайти найкращий хід у позиції на заданій глибині пошуку з використанням класичного методу оцінки позиції та за допомогою нейронної мережі за вибором;
- робота програми повинна виконуватись без доступу до мережі.

А.4.2 Вимоги до надійності

Програма призначена для роботи на серверній частині, через що має забезпечувати безперебійну роботу протягом усього часу, що працює сервер.

A.4.3 Вимоги до технічних та програмних засобів

Для повноцінного функціонування програми повинні виконуватися наступні умови:

- наявність процесору з тактовою частотою 2 ГГц та більше;
- об'єм оперативної пам'яті більший за 128МБ;
- вільний простір на диску в об'ємі не менше 64 МБ;
- операційна система Linux;

Умови до програмної частини:

- наявність бібліотеки tensorflow на комп'ютері (для використання аналізу за допомогою нейронної мережі);
- наявність нейромережевої моделі у форматі tensorflow на комп'ютері.

A.5 Вимоги до програмної документації

Склад програмної документації до програмного продукту:

- технічне завдання;
- опис розробки програми;
- опис тестування та експлуатації програми.

A.6 Стадії і етапи розробки

Розробка має бути проведена в наступні стадії:

- створення та узгодження технічного завдання;
- проектування системи;
- аналіз можливих методів реалізації рушія;
- розробка рушія;
- пошук датасету для нейронної мережі;

- проектування та навчання нейронної мережі;
- інтеграція рушія з нейронною мережею;
- створення супровідної документації;
- тестування працездатності програмного продукту.

A.7 Порядок контролю та приймання

Для контролю якості розробленого продукту, продукт та супутня документація повинні пройти наступні стадії:

- тестування програмного продукту;
- затвердження працездатності продукту та достовірності супутньої документації науковим керівником;
- нормоконтроль документації до програмного продукту.

ДОДАТОК Б

Текст програми

Б.1 Код проєкту ядра рушія

Код файлу CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.19)
project(ChessEngineCore LANGUAGES CXX)

set(CMAKE_INCLUDE_CURRENT_DIR ON)
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

#option(USE_NN_EVALUATION "Compile the neural network evaluation
functionality" ON)
configure_file(CoreConfig.h.in CoreConfig.h)

add_library(ChessEngineCore
    BoardRepresentation/Bitboard.cpp
    BoardRepresentation/Bitboard.h
    BoardRepresentation/Color.h
    BoardRepresentation/Direction.h
    BoardRepresentation/Piece.h
    BoardRepresentation/Square.h
    BoardRepresentation/Position.cpp
    BoardRepresentation/Position.h
    Utility/Zobrist.cpp
    Utility/Zobrist.h
    BoardRepresentation/Move.cpp
    BoardRepresentation/Move.h
    Utility/FenUtility.cpp
    Utility/FenUtility.h
    BoardRepresentation/BB.cpp
    BoardRepresentation/BB.h
    MoveGeneration/MoveGenerator.cpp
    MoveGeneration/MoveGenerator.h
    MoveGeneration/MoveList.h
    MoveGeneration/MoveList.cpp
    Search/PerformanceTester.cpp
    Search/PerformanceTester.h
    Evaluation/SimpleEvaluator.cpp
    Evaluation/SimpleEvaluator.h
    Search/DefaultSearcher.cpp
    Search/DefaultSearcher.h
    Search/PrincipalVariationSearcher.cpp
    Search/PrincipalVariationSearcher.h
    Search/TranspositionTable.cpp
    Search/TranspositionTable.h
    Evaluation/NeuralNetworkEvaluator.cpp
    Evaluation/NeuralNetworkEvaluator.h)

#if (USE_NN_EVALUATION)
    find_library(TENSORFLOW_LIB tensorflow HINT
$ENV{HOME}/libtensorflow2/lib)
    target_link_libraries (ChessEngineCore "${TENSORFLOW_LIB}")
#endif()

target_compile_definitions(ChessEngineCore PRIVATE CHESSENGINECORE_LIBRARY)
target_include_directories(ChessEngineCore PUBLIC
    "${PROJECT_BINARY_DIR}"
    ${EXTRA_INCLUDES})
```

)

```
target_link_options(ChessEngineCore PUBLIC -fuse-ld=gold)
```

Код файла ВВ.h:

```
#ifndef CHESSENGINE_BB_H
#define CHESSENGINE_BB_H

#include "Bitboard.h"
#include "Square.h"
#include "Color.h"

class BB
{
    // https://www.chessprogramming.org/Magic_Bitboards
    // https://www.chessprogramming.org/Hyperbola_Quintessence
public:
    static void initializeAll();

public:
    static constexpr Bitboard EMPTY = 0ull;
    static constexpr Bitboard FULL = ~EMPTY;

    static constexpr Bitboard RANK_1 = 0xFFull;
    static constexpr Bitboard RANK_2 = RANK_1 << NORTH;
    static constexpr Bitboard RANK_3 = RANK_2 << NORTH;
    static constexpr Bitboard RANK_4 = RANK_3 << NORTH;
    static constexpr Bitboard RANK_5 = RANK_4 << NORTH;
    static constexpr Bitboard RANK_6 = RANK_5 << NORTH;
    static constexpr Bitboard RANK_7 = RANK_6 << NORTH;
    static constexpr Bitboard RANK_8 = RANK_7 << NORTH;

    static constexpr Bitboard FILE_A = 0x01010101010101ull;
    static constexpr Bitboard FILE_B = FILE_A << EAST;
    static constexpr Bitboard FILE_C = FILE_B << EAST;
    static constexpr Bitboard FILE_D = FILE_C << EAST;
    static constexpr Bitboard FILE_E = FILE_D << EAST;
    static constexpr Bitboard FILE_F = FILE_E << EAST;
    static constexpr Bitboard FILE_G = FILE_F << EAST;
    static constexpr Bitboard FILE_H = FILE_G << EAST;

    static constexpr Bitboard WHITE_CASTLING_00_MASK = 0x90ull;
    static constexpr Bitboard WHITE_CASTLING_000_MASK = 0x11ull;
    static constexpr Bitboard WHITE_CASTLING_MASK = WHITE_CASTLING_00_MASK |
WHITE_CASTLING_000_MASK;
    static constexpr Bitboard BLACK_CASTLING_00_MASK =
0x9000000000000000ull;
    static constexpr Bitboard BLACK_CASTLING_000_MASK =
0x1100000000000000ull;
    static constexpr Bitboard BLACK_CASTLING_MASK = BLACK_CASTLING_00_MASK |
BLACK_CASTLING_000_MASK;
    static constexpr Bitboard ALL_CASTLING_MASK = WHITE_CASTLING_MASK |
BLACK_CASTLING_MASK;

    static constexpr Bitboard WHITE_00_BLOCKERS_MASK = 0x70ull;
    static constexpr Bitboard WHITE_000_BLOCKERS_MASK = 0x1Cull;
    static constexpr Bitboard BLACK_00_BLOCKERS_MASK = 0x7000000000000000ull;
    static constexpr Bitboard BLACK_000_BLOCKERS_MASK =
0x1C00000000000000ull;
```



```

        0x1000000000000000ull,    0x2000000000000000ull,    0x4000000000000000ull,
0x8000000000000000ull,
};

```

```

Bitboard BB::RANKS[8] =
{
    RANK_1, RANK_2, RANK_3, RANK_4, RANK_5, RANK_6, RANK_7, RANK_8
};

```

```

Bitboard BB::FILES[8] =
{
    FILE_A, FILE_B, FILE_C, FILE_D, FILE_E, FILE_F, FILE_G, FILE_H
};

```

```

Bitboard BB::DIAGONALS[15] =
{
    0x80ull,
    0x8040ull,
    0x804020ull,
    0x80402010ull,
    0x8040201008ull,
    0x804020100804ull,
    0x80402010080402ull,
    0x8040201008040201ull,
    0x4020100804020100ull,
    0x2010080402010000ull,
    0x1008040201000000ull,
    0x8040201000000000ull,
    0x4020100000000000ull,
    0x2010000000000000ull,
    0x1000000000000000ull,
};

```

```

Bitboard BB::ANTI_DIAGONALS[15] =
{
    0x1ull,
    0x102ull,
    0x10204ull,
    0x1020408ull,
    0x102040810ull,
    0x10204081020ull,
    0x1020408102040ull,
    0x102040810204080ull,
    0x204081020408000ull,
    0x408102040800000ull,
    0x810204080000000ull,
    0x1020408000000000ull,
    0x2040800000000000ull,
    0x4080000000000000ull,
    0x8000000000000000ull,
};

```

```

Bitboard BB::SQUARES_BETWEEN[NUMBER_OF_SQUARES][NUMBER_OF_SQUARES];
Bitboard BB::LINE[NUMBER_OF_SQUARES][NUMBER_OF_SQUARES];

```

```

Bitboard BB::PAWN_ATTACKS[NUMBER_OF_COLORS][NUMBER_OF_SQUARES] =
{
    { // BLACK PAWNS ATTACKS TO WHITE

```

```

    0x200ull, 0x500ull, 0xa00ull, 0x1400ull,
    0x2800ull, 0x5000ull, 0xa000ull, 0x4000ull,
    0x20000ull, 0x50000ull, 0xa0000ull, 0x140000ull,
    0x280000ull, 0x500000ull, 0xa00000ull, 0x400000ull,
    0x2000000ull, 0x5000000ull, 0xa000000ull, 0x14000000ull,
    0x28000000ull, 0x50000000ull, 0xa0000000ull, 0x40000000ull,
    0x200000000ull, 0x500000000ull, 0xa00000000ull, 0x1400000000ull,
    0x2800000000ull, 0x5000000000ull, 0xa000000000ull, 0x4000000000ull,
    0x20000000000ull,      0x50000000000ull,      0xa0000000000ull,
0x140000000000ull,
    0x280000000000ull,      0x500000000000ull,      0xa00000000000ull,
0x400000000000ull,
    0x2000000000000ull,      0x5000000000000ull,      0xa000000000000ull,
0x14000000000000ull,
    0x28000000000000ull,      0x50000000000000ull,      0xa0000000000000ull,
0x40000000000000ull,
    0x200000000000000ull,      0x500000000000000ull,      0xa00000000000000ull,
0x1400000000000000ull,
    0x2800000000000000ull, 0x5000000000000000ull, 0xa000000000000000ull,
0x4000000000000000ull,
    0x0ull, 0x0ull, 0x0ull, 0x0ull,
    0x0ull, 0x0ull, 0x0ull, 0x0ull,
}, // BLACK PAWNS ATTACKS TO WHITE

{ // WHITE PAWNS ATTACKS TO BLACK
    0x0ull, 0x0ull, 0x0ull, 0x0ull,
    0x0ull, 0x0ull, 0x0ull, 0x0ull,
    0x2ull, 0x5ull, 0xaull, 0x14ull,
    0x28ull, 0x50ull, 0xa0ull, 0x40ull,
    0x200ull, 0x500ull, 0xa00ull, 0x1400ull,
    0x2800ull, 0x5000ull, 0xa000ull, 0x4000ull,
    0x20000ull, 0x50000ull, 0xa0000ull, 0x140000ull,
    0x280000ull, 0x500000ull, 0xa00000ull, 0x400000ull,
    0x2000000ull, 0x5000000ull, 0xa000000ull, 0x14000000ull,
    0x28000000ull, 0x50000000ull, 0xa0000000ull, 0x40000000ull,
    0x200000000ull, 0x500000000ull, 0xa00000000ull, 0x1400000000ull,
    0x2800000000ull, 0x5000000000ull, 0xa000000000ull, 0x4000000000ull,
    0x20000000000ull,      0x50000000000ull,      0xa0000000000ull,
0x140000000000ull,
    0x280000000000ull,      0x500000000000ull,      0xa00000000000ull,
0x400000000000ull,
    0x2000000000000ull,      0x5000000000000ull,      0xa000000000000ull,
0x14000000000000ull,
    0x28000000000000ull,      0x50000000000000ull,      0xa0000000000000ull,
0x40000000000000ull,
    } // WHITE PAWNS ATTACKS TO BLACK
};

Bitboard BB::KING_ATTACKS[NUMBER_OF_SQUARES] =
{
    0x302ull, 0x705ull, 0xe0aull, 0x1c14ull,
    0x3828ull, 0x7050ull, 0xe0a0ull, 0xc040ull,
    0x30203ull, 0x70507ull, 0xe0a0eull, 0x1c141cull,
    0x382838ull, 0x705070ull, 0xe0a0e0ull, 0xc040c0ull,
    0x3020300ull, 0x7050700ull, 0xe0a0e00ull, 0x1c141c00ull,
    0x38283800ull, 0x70507000ull, 0xe0a0e000ull, 0xc040c000ull,
    0x302030000ull, 0x705070000ull, 0xe0a0e0000ull, 0x1c141c0000ull,
    0x3828380000ull, 0x7050700000ull, 0xe0a0e00000ull, 0xc040c00000ull,
    0x30203000000ull, 0x70507000000ull, 0xe0a0e000000ull, 0x1c141c000000ull,
    0x382838000000ull,      0x705070000000ull,      0xe0a0e0000000ull,
0xc040c0000000ull,

```

```

    0x3020300000000ull,    0x7050700000000ull,    0xe0a0e00000000ull,
0x1c141c000000000ull,    0x7050700000000ull,    0xe0a0e00000000ull,
    0x382838000000000ull,    0x7050700000000ull,    0xe0a0e00000000ull,
0xc040c0000000000ull,    0x7050700000000ull,    0xe0a0e00000000ull,
    0x302030000000000ull,    0x7050700000000ull,    0xe0a0e00000000ull,
0x1c141c000000000ull,    0x7050700000000ull,    0xe0a0e00000000ull,
    0x382838000000000ull,    0x7050700000000ull,    0xe0a0e00000000ull,
0xc040c0000000000ull,    0x507000000000000ull,    0xa0e000000000000ull,
    0x203000000000000ull,    0x507000000000000ull,    0xa0e000000000000ull,
0x141c00000000000ull,    0x507000000000000ull,    0xa0e000000000000ull,
    0x283800000000000ull,    0x507000000000000ull,    0xa0e000000000000ull,
0x40c000000000000ull,
};

```

```
Bitboard BB::KNIGHT_ATTACKS[NUMBER_OF_SQUARES] =
```

```

{
    0x20400ull, 0x50800ull, 0xa1100ull, 0x142200ull,
    0x284400ull, 0x508800ull, 0xa01000ull, 0x402000ull,
    0x2040004ull, 0x5080008ull, 0xa110011ull, 0x14220022ull,
    0x28440044ull, 0x50880088ull, 0xa0100010ull, 0x40200020ull,
    0x204000402ull, 0x508000805ull, 0xa1100110aull, 0x1422002214ull,
    0x2844004428ull, 0x5088008850ull, 0xa0100010a0ull, 0x4020002040ull,
    0x20400040200ull, 0x50800080500ull, 0xa1100110a00ull, 0x142200221400ull,
    0x284400442800ull,    0x508800885000ull,    0xa0100010a000ull,
0x402000204000ull,
    0x2040004020000ull,    0x5080008050000ull,    0xa1100110a0000ull,
0x14220022140000ull,
    0x28440044280000ull,    0x5088008850000ull,    0xa0100010a00000ull,
0x40200020400000ull,
    0x204000402000000ull,    0x508000805000000ull,    0xa1100110a000000ull,
0x1422002214000000ull,
    0x2844004428000000ull,    0x5088008850000000ull,    0xa0100010a0000000ull,
0x4020002040000000ull,
    0x4000402000000000ull,    0x8000805000000000ull,    0x1100110a00000000ull,
0x2200221400000000ull,
    0x4400442800000000ull,    0x8800885000000000ull,    0x100010a000000000ull,
0x2000204000000000ull,
    0x402000000000000ull,    0x805000000000000ull,    0x110a00000000000ull,
0x221400000000000ull,
    0x442800000000000ull,    0x0088500000000000ull,    0x0010a00000000000ull,
0x204000000000000ull,
};

```

```
Bitboard BB::BISHOP_ATTACKS[NUMBER_OF_SQUARES][512];
```

```
Bitboard BB::BISHOP_ATTACK_MASKS[NUMBER_OF_SQUARES];
```

```
int BB::BISHOP_ATTACK_SHIFTS[NUMBER_OF_SQUARES];
```

```
// THANK YOU SMALLCHESS ENGINE
```

```
Bitboard BB::BISHOP_MAGICS[NUMBER_OF_SQUARES] =
```

```

{
    0x0002020202020200ull,    0x0002020202020000ull,    0x0004010202000000ull,
0x00040400800000000ull,
    0x00011040000000000ull,    0x00008210400000000ull,    0x0000410410400000ull,
0x0000104104104000ull,
    0x0000040404040400ull,    0x0000020202020200ull,    0x0000040102020000ull,
0x00000404008000000ull,
    0x00000110400000000ull,    0x0000008210400000ull,    0x0000004104104000ull,
0x0000002082082000ull,
    0x0004000808080800ull,    0x0002000404040400ull,    0x0001000202020200ull,
0x0000800802004000ull,
    0x0000800400A00000ull,    0x0000200100884000ull,    0x0000400082082000ull,
0x0000200041041000ull,
};

```

```

0x0002080010101000ull, 0x0001040008080800ull, 0x0000208004010400ull,
0x0000404004010200ull,
0x0000840000802000ull, 0x0000404002011000ull, 0x0000808001041000ull,
0x0000404000820800ull,
0x0001041000202000ull, 0x0000820800101000ull, 0x0000104400080800ull,
0x0000020080080080ull,
0x00004040040040100ull, 0x0000808100020100ull, 0x0001010100020800ull,
0x0000808080010400ull,
0x0000820820004000ull, 0x0000410410002000ull, 0x0000082088001000ull,
0x0000002011000800ull,
0x0000080100400400ull, 0x0001010101000200ull, 0x0002020202000400ull,
0x0001010101000200ull,
0x0000410410400000ull, 0x0000208208200000ull, 0x0000002084100000ull,
0x0000000020880000ull,
0x0000000100202000ull, 0x0000040408020000ull, 0x0004040404040000ull,
0x0002020202020000ull,
0x0000104104104000ull, 0x0000002082082000ull, 0x0000000020841000ull,
0x0000000000208800ull,
0x0000000010020200ull, 0x00000000404080200ull, 0x0000040404040400ull,
0x0002020202020200ull,
};

```

```

Bitboard BB::ROOK_ATTACKS[NUMBER_OF_SQUARES][4096];
Bitboard BB::ROOK_ATTACK_MASKS[NUMBER_OF_SQUARES];
int BB::ROOK_ATTACK_SHIFTS[NUMBER_OF_SQUARES];
// THANK YOU SMALLCHESS ENGINE
Bitboard BB::ROOK_MAGICS[NUMBER_OF_SQUARES] =
{
    0x0080001020400080ull,    0x0040001000200040ull,    0x00800081000200080ull,
    0x0080040800100080ull,    0x0080010200040080ull,    0x0080008001000200ull,
    0x0080020400080080ull,    0x0080010200040080ull,    0x0080008001000200ull,
    0x0000800020400080ull,    0x0000400020005000ull,    0x0000801000200080ull,
    0x0000800800100080ull,    0x0000800400080080ull,    0x0000800100020080ull,
    0x0000800040800100ull,    0x0000208000400080ull,    0x0000808010002000ull,
    0x000080808001000ull,    0x0000404000201000ull,    0x0000808010002000ull,
    0x0000808004000800ull,    0x0000808002000400ull,    0x0000010100020004ull,
    0x0000020000408104ull,    0x0000208080004000ull,    0x00000100080200080ull,
    0x0000080080100080ull,    0x0000040080080080ull,    0x0000010080800200ull,
    0x0000800080004100ull,    0x0000204000800080ull,    0x0000010008080200ull,
    0x0000204000800080ull,    0x0000200040401000ull,    0x0000010008080200ull,
    0x0000080080801000ull,    0x0000040080800800ull,    0x0000020001010004ull,
    0x0000800040800100ull,    0x0000200040008080ull,    0x00000100020008080ull,
    0x0000080010008080ull,    0x0000040008008080ull,    0x0000010002008080ull,
    0x0000004081020004ull,    0x0000204000800080ull,    0x00000100020008080ull,
    0x0000080010008080ull,    0x0000040008008080ull,    0x00000800100020080ull,
    0x0000800041000080ull,    0x00FFFCDDFCED714Aull,    0x003FFFCDDFFD88096ull,
    0x0000040810002101ull,    0x0001000204080011ull,    0x0001000082000401ull,
    0x0001FFFAABFAD1A2ull,
};

```

```

void BB::initializeAll()
{
    // initialize rook tables
    for (Square sq = A1; sq <= H8; ++sq)
    {
        Bitboard boardEdges = ((FILE_A | FILE_H) & ~FILES[fileOf(sq)])
                               | ((RANK_1 | RANK_8) & ~RANKS[rankOf(sq)]);
        ROOK_ATTACK_MASKS[sq] = (RANKS[rankOf(sq)] ^ FILES[fileOf(sq)]) &
~boardEdges;
        ROOK_ATTACK_SHIFTS[sq] = 64 -
ROOK_ATTACK_MASKS[sq].populationCount();

        Bitboard subset = 0;
        do
        {
            uint64_t index = static_cast<uint64_t>(subset);
            index = index * static_cast<uint64_t>(ROOK_MAGICS[sq]);
            index = index >> ROOK_ATTACK_SHIFTS[sq];
            ROOK_ATTACKS[sq][index] = Bitboard::slidingAttacks(sq,
Bitboard(subset), FILES[fileOf(sq)])
| Bitboard::slidingAttacks(sq,
Bitboard(subset), RANKS[rankOf(sq)]);
            subset = (subset - ROOK_ATTACK_MASKS[sq]) &
ROOK_ATTACK_MASKS[sq];
        }
        while (subset);
    }

    // initialize bishop tables
    for (Square sq = A1; sq <= H8; ++sq)
    {
        Bitboard boardEdges = ((FILE_A | FILE_H) & ~FILES[fileOf(sq)])
                               | ((RANK_1 | RANK_8) & ~RANKS[rankOf(sq)]);
        BISHOP_ATTACK_MASKS[sq] = (DIAGONALS[diagonalOf(sq)]
^ ANTI_DIAGONALS[antiDiagonalOf(sq)]) &
~boardEdges;
        BISHOP_ATTACK_SHIFTS[sq] = 64 -
BISHOP_ATTACK_MASKS[sq].populationCount();

        Bitboard subset = 0;
        do
        {
            uint64_t index = static_cast<uint64_t>(subset);
            index = index * static_cast<uint64_t>(BISHOP_MAGICS[sq]);
            index = index >> BISHOP_ATTACK_SHIFTS[sq];
            BISHOP_ATTACKS[sq][index] = Bitboard::slidingAttacks(sq, subset,
DIAGONALS[diagonalOf(sq)])
| Bitboard::slidingAttacks(sq,
subset, ANTI_DIAGONALS[antiDiagonalOf(sq)]);
            subset = (subset - BISHOP_ATTACK_MASKS[sq]) &
BISHOP_ATTACK_MASKS[sq];
        }
        while (subset);
    }

    // initialize squares between
    for (Square sI = A1; sI <= H8; ++sI)
    {
        for (Square sJ = A1; sJ <= H8; ++sJ)
        {

```

```

        Bitboard squares = SQUARES[sI] | SQUARES[sJ];
        if (sI == sJ)
        {
            SQUARES_BETWEEN[sI][sJ] = EMPTY;
        }
        else if (fileOf(sI) == fileOf(sJ) || rankOf(sI) == rankOf(sJ))
        {
            SQUARES_BETWEEN[sI][sJ] = Bitboard::rookAttacks(sI, squares)
& Bitboard::rookAttacks(sJ, squares);
        }
        else if (diagonalOf(sI) == diagonalOf(sJ) || antiDiagonalOf(sI)
== antiDiagonalOf(sJ))
        {
            SQUARES_BETWEEN[sI][sJ] = Bitboard::bishopAttacks(sI,
squares) & Bitboard::bishopAttacks(sJ, squares);
        }
        else
        {
            SQUARES_BETWEEN[sI][sJ] = EMPTY;
        }
    }
}

// initialize lines
for (Square sI = A1; sI <= H8; ++sI)
{
    for (Square sJ = A1; sJ <= H8; ++sJ)
    {
        Bitboard squares = SQUARES[sI] | SQUARES[sJ];
        if (sI == sJ)
        {
            LINE[sI][sJ] = EMPTY;
        }
        else if (fileOf(sI) == fileOf(sJ) || rankOf(sI) == rankOf(sJ))
        {
            LINE[sI][sJ] = (Bitboard::rookAttacks(sI, EMPTY) &
Bitboard::rookAttacks(sJ, EMPTY)) | squares;
        }
        else if (diagonalOf(sI) == diagonalOf(sJ) || antiDiagonalOf(sI)
== antiDiagonalOf(sJ))
        {
            LINE[sI][sJ] = (Bitboard::bishopAttacks(sI, EMPTY) &
Bitboard::bishopAttacks(sJ, EMPTY)) | squares;
        }
        else
        {
            LINE[sI][sJ] = EMPTY;
        }
    }
}
}

```

Код файла Bitboard.h:

```

#ifndef CHESSENGINE_BITBOARD_H
#define CHESSENGINE_BITBOARD_H

#include <stdint>
#include "Square.h"
#include "Color.h"

```

```

#include "Direction.h"

class Bitboard
{
public:
    constexpr Bitboard(uint64_t bitboard = 0ull)
        : bitboard(bitboard) {}
    constexpr Bitboard(const Bitboard & other) = default;
    constexpr Bitboard(Bitboard && other) = default;
    constexpr Bitboard & operator = (const Bitboard & other) = default;
    constexpr Bitboard & operator = (Bitboard && other) = default;
    constexpr ~Bitboard() = default;

public:
    constexpr Bitboard operator ~ () const
    {
        return Bitboard(~bitboard);
    }

    constexpr Bitboard operator - (const Bitboard & b) const
    {
        return Bitboard(bitboard - b.bitboard);
    }

    constexpr Bitboard operator * (const Bitboard & b) const
    {
        return Bitboard(bitboard * b.bitboard);
    }

    constexpr Bitboard operator & (const Bitboard & b) const
    {
        return Bitboard(bitboard & b.bitboard);
    }

    constexpr Bitboard & operator &= (const Bitboard & b)
    {
        this->bitboard &= b.bitboard;
        return *this;
    }

    constexpr Bitboard operator | (const Bitboard & b) const
    {
        return Bitboard(bitboard | b.bitboard);
    }

    constexpr Bitboard & operator |= (const Bitboard & b)
    {
        this->bitboard |= b.bitboard;
        return *this;
    }

    constexpr Bitboard operator ^ (const Bitboard & b) const
    {
        return Bitboard(bitboard ^ b.bitboard);
    }

    constexpr Bitboard & operator ^= (const Bitboard & b)
    {
        this->bitboard ^= b.bitboard;
        return *this;
    }
}

```

```

constexpr Bitboard operator << (int shift) const
{
    return Bitboard(bitboard << shift);
}

constexpr bool operator == (const Bitboard & b) const
{
    return bitboard == b.bitboard;
}

constexpr bool operator != (const Bitboard & b) const
{
    return bitboard != b.bitboard;
}

constexpr Bitboard reversed() const
{
    //https://www.chessprogramming.org/Reverse_Bitboards
    uint64_t bb = bitboard;
    bb = (bb & 0x5555555555555555ull) << 1 | (bb >> 1) &
0x5555555555555555ull;
    bb = (bb & 0x3333333333333333ull) << 2 | (bb >> 2) &
0x3333333333333333ull;
    bb = (bb & 0x0f0f0f0f0f0f0f0full) << 4 | (bb >> 4) &
0x0f0f0f0f0f0f0f0full;
    bb = (bb & 0x00ff00ff00ff00ffull) << 8 | (bb >> 8) &
0x00ff00ff00ff00ffull;

    return (bb << 48) | ((bb & 0xffff0000ull) << 16) |
((bb >> 16) & 0xffff0000ull) | (bb >> 48);
}

Bitboard shifted(Direction dir) const;

constexpr explicit operator bool() const
{
    return static_cast<bool>(this->bitboard);
}

constexpr explicit operator uint64_t() const
{
    return bitboard;
}

static constexpr int DEBRUNIJN_INDEX[64] = {
    0, 1, 48, 2, 57, 49, 28, 3,
    61, 58, 50, 42, 38, 29, 17, 4,
    62, 55, 59, 36, 53, 51, 43, 22,
    45, 39, 33, 30, 24, 18, 12, 5,
    63, 47, 56, 27, 60, 41, 37, 16,
    54, 35, 52, 21, 44, 32, 23, 11,
    46, 26, 40, 15, 34, 20, 31, 10,
    25, 14, 19, 9, 13, 8, 7, 6
};

constexpr Square lsb() const
{
    //https://www.chessprogramming.org/BitScan
    constexpr uint64_t debruijn64 = 0x03f79d71b4cb0a89;

```

```

        return static_cast<Square>(DEBRUNIJN_INDEX[((bitboard & -bitboard) *
debruijn64) >> 58]);
    }

    constexpr Square popLsb()
    {
        Square lsb = this->lsb();
        bitboard &= (bitboard - 1);
        return lsb;
    }

    constexpr int populationCount() const
    {
        // https://www.chessprogramming.org/Population_Count
        constexpr Bitboard K1 = 0x5555555555555555ull;
        constexpr Bitboard K2 = 0x3333333333333333ull;
        constexpr Bitboard K4 = 0x0f0f0f0f0f0f0f0full;
        constexpr Bitboard KF = 0x0101010101010101ull;

        uint64_t x = bitboard;
        x = x - ((x >> 1) & K1.bitboard);
        x = (x & K2.bitboard) + ((x >> 2) & K2.bitboard);
        x = (x + (x >> 4)) & K4.bitboard;
        x = (x * KF.bitboard) >> 56;
        return static_cast<int>(x);
    }

    friend std::ostream & operator << (std::ostream & os, const Bitboard &
bb);

public:
    static Bitboard square(Square square);

    static Bitboard kingCastlingMask(Color color);
    static Bitboard queenCastlingMask(Color color);
    static Bitboard kingCastlingBlockers(Color color);
    static Bitboard queenCastlingBlockers(Color color);
    static Bitboard slidingAttacks(Square square, Bitboard occupancy,
Bitboard mask);

    static uint32_t castleRightsFromMovedPieces(Bitboard movedPieces);
    static Bitboard pawnAttacksTo(Color color, Square square);
    static Bitboard pawnAttacks(Color color, Bitboard pawns);
    static Bitboard kingAttacks(Square square);
    static Bitboard knightAttacks(Square square);
    static Bitboard bishopAttacks(Square square, Bitboard occupancy);
    static Bitboard rookAttacks(Square square, Bitboard occupancy);
    static Bitboard queenAttacks(Square square, Bitboard occupancy);
    static Bitboard line(Square sq1, Square sq2);
    static Bitboard squaresBetween(Square sq1, Square sq2);
    static Bitboard rank(Square sq);
    static Bitboard file(Square sq);

private:
    uint64_t bitboard;
};

//const

#endif //CHESENGINE_BITBOARD_H

```

Код файла Bitboard.cpp:

```
#include "Bitboard.h"
#include "BB.h"
#include <ostream>

Bitboard Bitboard::square(Square square)
{
    return BB::SQUARES[square];
}

Bitboard Bitboard::pawnAttacksTo(Color color, Square square)
{
    return BB::PAWN_ATTACKS[color][square];
}

Bitboard Bitboard::pawnAttacks(Color color, Bitboard pawns)
{
    if (color == WHITE)
    {
        return pawns.shifted(NORTH_WEST) | pawns.shifted(NORTH_EAST);
    }
    else
    {
        return pawns.shifted(SOUTH_WEST) | pawns.shifted(SOUTH_EAST);
    }
}

Bitboard Bitboard::shifted(Direction dir) const
{
    if (dir == NORTH)        return bitboard << 8;
    if (dir == SOUTH)        return bitboard >> 8;
    if (dir == EAST)         return (bitboard & ~BB::FILE_H.bitboard) << 1;
    if (dir == WEST)         return (bitboard & ~BB::FILE_A.bitboard) >> 1;
    if (dir == NORTH_EAST)   return (bitboard & ~BB::FILE_H.bitboard) << 9;
    if (dir == NORTH_WEST)   return (bitboard & ~BB::FILE_A.bitboard) << 7;
    if (dir == SOUTH_EAST)   return (bitboard & ~BB::FILE_H.bitboard) >> 7;
    if (dir == SOUTH_WEST)   return (bitboard & ~BB::FILE_A.bitboard) >> 9;
    return 0;
}

Bitboard Bitboard::kingAttacks(Square square)
{
    return BB::KING_ATTACKS[square];
}

Bitboard Bitboard::knightAttacks(Square square)
{
    return BB::KNIGHT_ATTACKS[square];
}

Bitboard Bitboard::bishopAttacks(Square square, Bitboard occupancy)
{
    const      uint32_t      index      =      ((occupancy      &
BB::BISHOP_ATTACK_MASKS[square]).bitboard
BB::BISHOP_MAGICS[square].bitboard) >> BB::BISHOP_ATTACK_SHIFTS[square];
    return BB::BISHOP_ATTACKS[square][index];
}
```

```

Bitboard Bitboard::rookAttacks(Square square, Bitboard occupancy)
{
    const uint32_t index = ((occupancy &
BB::ROOK_ATTACK_MASKS[square]).bitboard * BB::ROOK_MAGICS[square].bitboard)
>> BB::ROOK_ATTACK_SHIFTS[square];
    return BB::ROOK_ATTACKS[square][index];
}

Bitboard Bitboard::queenAttacks(Square square, Bitboard occupancy)
{
    return bishopAttacks(square, occupancy) | rookAttacks(square, occupancy);
}

Bitboard Bitboard::slidingAttacks(Square square, Bitboard occupancy, Bitboard
mask)
{
    // https://www.chessprogramming.org/Hyperbola_Quintessence
    const Bitboard maskedOccupancy = occupancy & mask;
    const Bitboard firstPart = maskedOccupancy - BB::SQUARES[square] * 2;
    const Bitboard secondPart = (maskedOccupancy.reversed() -
BB::SQUARES[square].reversed() * 2).reversed();
    return (firstPart ^ secondPart) & mask;
}

uint32_t Bitboard::castleRightsFromMovedPieces(Bitboard movedPieces)
{
    // generate 4-bit castleRights for indexing the zobrist table
    uint32_t mask = 0;

    if (!(movedPieces & Bitboard::queenCastlingMask(WHITE)))
        mask |= 0b0001;

    if (!(movedPieces & Bitboard::queenCastlingMask(BLACK)))
        mask |= 0b0010;

    if (!(movedPieces & Bitboard::kingCastlingMask(WHITE)))
        mask |= 0b0100;

    if (!(movedPieces & Bitboard::kingCastlingMask(BLACK)))
        mask |= 0b1000;

    return mask;
}

std::ostream &operator<<(std::ostream &os, const Bitboard &bb)
{
    for (int i = 56; i >= 0; i -= 8)
    {
        for (int j = 0; j < 8; ++j)
        {
            os << (bb.bitboard & (1ull << (i + j)) ? 1 : 0);
        }
        os << '\n';
    }
    return os;
}

Bitboard Bitboard::line(Square sq1, Square sq2)
{
    return BB::LINE[sq1][sq2];
}

```

```

Bitboard Bitboard::squaresBetween(Square sq1, Square sq2)
{
    return BB::SQUARES_BETWEEN[sq1][sq2];
}

Bitboard Bitboard::kingCastlingMask(Color color)
{
    return color == WHITE ? BB::WHITE_CASTLING_00_MASK :
    BB::BLACK_CASTLING_00_MASK;
}

Bitboard Bitboard::queenCastlingMask(Color color)
{
    return color == WHITE ? BB::WHITE_CASTLING_000_MASK :
    BB::BLACK_CASTLING_000_MASK;
}

Bitboard Bitboard::kingCastlingBlockers(Color color)
{
    return color == WHITE ? BB::WHITE_00_BLOCKERS_MASK :
    BB::BLACK_00_BLOCKERS_MASK;
}

Bitboard Bitboard::queenCastlingBlockers(Color color)
{
    return color == WHITE ? BB::WHITE_000_BLOCKERS_MASK :
    BB::BLACK_000_BLOCKERS_MASK;
}

Bitboard Bitboard::rank(Square square)
{
    return BB::RANKS[rankOf(square)];
}

Bitboard Bitboard::file(Square square)
{
    return BB::FILES[fileOf(square)];
}

```

Код файла Color.h:

```

#ifndef CHESENGINE_COLOR_H
#define CHESENGINE_COLOR_H

enum Color : int // derive from int, cause this should be faster
{
    WHITE = 0,
    BLACK = 1,
    NUMBER_OF_COLORS = 2
};

constexpr Color operator~(Color c)
{
    return static_cast<Color>(c ^ BLACK);
}

#endif //CHESENGINE_COLOR_H

```

Код файлу Direction.h:

```

#ifndef CHESENGINE_DIRECTION_H
#define CHESENGINE_DIRECTION_H

enum Direction: int
{
    NORTH_WEST = 7, NORTH          = 8, NORTH_EAST = 9,
    WEST        = -1, NULL_DIRECTION = 0, EAST      = 1,
    SOUTH_WEST = -9, SOUTH          = -8, SOUTH_EAST = -7
};

constexpr Direction relativeDirection(Color color, Direction dir)
{
    return static_cast<Direction>(color == WHITE ? dir : -dir);
}

#endif //CHESENGINE_DIRECTION_H

```

Код файлу Move.h:

```

#ifndef CHESENGINE_MOVE_H
#define CHESENGINE_MOVE_H

#include <cstdint>
#include "Square.h"

class Move
{
public:
    enum Type: int
    {
        QUIET ,
        CAPTURE,
        DOUBLE_PUSH ,
        CASTLE_00,
        CASTLE_000,
        EN_PASSANT,

        PROMOTION,
        PROMOTION_KNIGHT,
        PROMOTION_BISHOP,
        PROMOTION_ROOK,
        PROMOTION_QUEEN,

        PROMOTION_CAPTURE,
        PROMOTION_CAPTURE_KNIGHT,
        PROMOTION_CAPTURE_BISHOP,
        PROMOTION_CAPTURE_ROOK,
        PROMOTION_CAPTURE_QUEEN
    };

private:
    uint16_t move;

public:
    constexpr explicit Move()
        : move(0) {}

```

```

constexpr Move(Square from, Square to, Type type = Type::QUIET)
    : move((type << 12) | (from << 6) | to) {}

constexpr Move(const Move & other) = default;
constexpr Move(Move && other) = default;
constexpr Move & operator = (const Move & other) = default;
constexpr Move & operator = (Move && other) = default;

constexpr ~Move() = default;

public:
    constexpr Square to() const
    {
        return static_cast<Square>(move & 0b111111);
    }

    constexpr Square from() const
    {
        return static_cast<Square>((move >> 6) & 0b111111);
    }

    constexpr bool isValid() const
    {
        return to() != NULL_SQUARE && from() != NULL_SQUARE;
    }

    constexpr Type type() const
    {
        return static_cast<Type>(move >> 12);
    }

    constexpr bool isPromotion() const
    {
        Type type = this->type();
        return
            type == PROMOTION_KNIGHT
            || type == PROMOTION_BISHOP
            || type == PROMOTION_ROOK
            || type == PROMOTION_QUEEN
            || type == PROMOTION_CAPTURE_KNIGHT
            || type == PROMOTION_CAPTURE_BISHOP
            || type == PROMOTION_CAPTURE_ROOK
            || type == PROMOTION_CAPTURE_QUEEN;
    }

    constexpr bool isCapture() const
    {
        Type type = this->type();
        return
            type == CAPTURE
            || type == EN_PASSANT
            || type == PROMOTION_CAPTURE_KNIGHT
            || type == PROMOTION_CAPTURE_BISHOP
            || type == PROMOTION_CAPTURE_ROOK
            || type == PROMOTION_CAPTURE_QUEEN;
    }

    constexpr bool isCastle() const
    {
        Type type = this->type();
        return type == CASTLE_00 || type == CASTLE_000;
    }

```

```

constexpr bool isQuiet() const
{
    Type type = this->type();
    return type == QUIET
        || type == DOUBLE_PUSH
        || type == PROMOTION_KNIGHT
        || type == PROMOTION_BISHOP
        || type == PROMOTION_ROOK
        || type == PROMOTION_QUEEN
        || type == CASTLE_00
        || type == CASTLE_000;
}

constexpr bool operator == (const Move & move) const
{
    return this->move == move.move;
}

constexpr bool operator != (const Move & move) const
{
    return !(*this == move);
}

std::string toString() const;
};

#endif //CHESENGINE_MOVE_H

```

Код файла Move.cpp:

```

#include "Move.h"

std::string Move::toString() const
{
    std::string promotionPiece = "";
    if (isPromotion())
    {
        if (type() == PROMOTION_KNIGHT || type() == PROMOTION_CAPTURE_KNIGHT)
            promotionPiece = "n";
        if (type() == PROMOTION_BISHOP || type() == PROMOTION_CAPTURE_BISHOP)
            promotionPiece = "b";
        if (type() == PROMOTION_ROOK || type() == PROMOTION_CAPTURE_ROOK)
            promotionPiece = "r";
        if (type() == PROMOTION_QUEEN || type() == PROMOTION_CAPTURE_QUEEN)
            promotionPiece = "q";
    }

    return std::string(SQUARE_STRING[from()]) +
std::string(SQUARE_STRING[to()] + promotionPiece);
}

```

Код файла Piece.h:

```

#ifndef CHESENGINE_PIECE_H
#define CHESENGINE_PIECE_H

#include "Color.h"

```

```

#include <string>

// To do: create a class that holds both enums
enum PieceType : int
{
    NULL_PIECE_TYPE    = 0,
    PAWN               = 1,
    KNIGHT              = 2,
    BISHOP              = 3,
    ROOK                = 4,
    QUEEN              = 5,
    KING               = 6,
    NUMBER_OF_PIECE_TYPES = 7
};

const std::string PIECE_CHAR = "-PNBRQK--pnbrqk-";

enum Piece : int
{
    NULL_PIECE    = 0,

    WHITE_PAWN    = 1,
    WHITE_KNIGHT  = 2,
    WHITE_BISHOP  = 3,
    WHITE_ROOK    = 4,
    WHITE_QUEEN   = 5,
    WHITE_KING    = 6,

    BLACK_PAWN    = 9,
    BLACK_KNIGHT  = 10,
    BLACK_BISHOP  = 11,
    BLACK_ROOK    = 12,
    BLACK_QUEEN   = 13,
    BLACK_KING    = 14,

    NUMBER_OF_PIECES = 16
};

constexpr Piece makePiece(Color color, PieceType pieceType)
{
    return static_cast<Piece>((color << 3) | pieceType);
}

constexpr PieceType typeOf(Piece piece)
{
    constexpr int pieceTypeMask = 0b111;
    return static_cast<PieceType>(piece & pieceTypeMask);
}

constexpr Color colorOf(Piece piece)
{
    return static_cast<Color>((piece >> 3) & 0b1);
}

#endif //CHESENGINE_PIECE_H

```

Код файла Position.h:

```

#ifndef CHESENGINE_POSITION_H
#define CHESENGINE_POSITION_H

```

```

#include <string>
#include <vector>
#include "Bitboard.h"
#include "Piece.h"
#include "Square.h"
#include "Move.h"
#include "Direction.h"
#include "../Utility/FenUtility.h"

class Position
{
public:
    enum Status
    {
        WHITE_WIN,
        BLACK_WIN,
        DRAW,
        STALEMATE,
        NOT_FINISHED
    };

    struct UnrecoverableState
    {
        constexpr UnrecoverableState()
            : movedPieces(0ull)
            , checkers(0)
            , pinned(0)
            , lastCaptured(NULL_PIECE)
            , enPassantSquare(NULL_SQUARE)
            , plyFiftyMoveRule(0)
        {}

        constexpr UnrecoverableState(const UnrecoverableState & other)
            : movedPieces(other.movedPieces)
            , checkers(0)
            , pinned(0)
            , lastCaptured(NULL_PIECE)
            , enPassantSquare(NULL_SQUARE)
            , plyFiftyMoveRule(other.plyFiftyMoveRule + 1)
        {}

        Bitboard movedPieces;
        Bitboard checkers;
        Bitboard pinned;
        Piece lastCaptured;
        Square enPassantSquare;
        int plyFiftyMoveRule;
    };

public:
    UnrecoverableState history[512];
    Bitboard pieceBB[NUMBER_OF_PIECES];
    Bitboard colorBB[NUMBER_OF_COLORS];

    Piece mailbox[NUMBER_OF_SQUARES];
    uint64_t hash;
    uint32_t gamePly;
    Color sideToPlay;

```

```

public:
    explicit Position(const std::string & fen = FenUtility::DEFAULT_FEN);
    Position(const Position & pos);
    Position(Position && pos) = delete;
    Position & operator = (const Position & pos) = delete;
    Position & operator = (Position && pos) = delete;
    ~Position() = default;

public:
    std::vector<float> binaryFeatures() const;
    void clear();
    Status status() const;
    void set(const std::string & fen);
    std::string fen() const;
    Move moveFromString(const std::string & move);

    void putPiece(Piece piece, Square to);
    void removePiece(Square from);

    void movePieceQuiet(Square from, Square to);
    void updateNonPieceHashing(int updateGamePly);

    void makeMove(Move move);
    void undoMove(Move move);

    Bitboard getDiagonalSliders(Color color) const;
    Bitboard getOrthogonalSliders(Color color) const;
    void recalculateCheckersAndPinned();
    Bitboard getCheckers() const;
    Bitboard getPinnedPieces() const;

    friend std::ostream & operator << (std::ostream & os, const Position &
pos);
};

#endif //CHESENGINE_POSITION_H

```

Код файла Position.cpp:

```

#include "Position.h"
#include "../Utility/Zobrist.h"
#include "../Utility/FenUtility.h"
#include <ostream>
#include "../Evaluation/SimpleEvaluator.h"
#include <MoveGeneration/MoveList.h>

Position::Position(const std::string &fen)
{
    clear();
    set(fen);
}

Position::Position(const Position & pos)
{
    std::copy(pos.mailbox, pos.mailbox + NUMBER_OF_SQUARES, mailbox);
    std::copy(pos.piecesBB, pos.piecesBB + NUMBER_OF_PIECES, piecesBB);
    std::copy(pos.colorsBB, pos.colorsBB + NUMBER_OF_PIECES, colorsBB);
    std::copy(pos.history, pos.history + 512, history);
    hash = pos.hash;
}

```

```

        gamePly = pos.gamePly;
        sideToPlay = pos.sideToPlay;
    }

std::vector<float> Position::binaryFeatures() const
{
    std::vector<float> features(768, 0.0f);

    for (Square sq = A1; sq < NUMBER_OF_SQUARES; ++sq)
    {
        if (mailbox[sq] != NULL_PIECE)
        {
            Square relSquare = ((sideToPlay == WHITE) ? sq : mirrored(sq));
            const size_t featureIndex = relSquare * 12 + (typeOf(mailbox[sq])
- 1)
+ (colorOf(mailbox[sq]) != sideToPlay
? 6 : 0);
            features[featureIndex] = 1.0f;
        }
    }

    // if (history[gamePly].enPassantSquare != NULL_SQUARE)
    // {
    //     const size_t featureIndex = 64 * 12
    //                                     +
    fileOf(history[gamePly].enPassantSquare)
    //                                     +
    (rankOf(history[gamePly].enPassantSquare) != 2 ? 8 : 0);
    //     features[featureIndex] = 1.0f;
    // }

    // const Bitboard movedPieces = history[gamePly].movedPieces;
    // if (!(movedPieces & Bitboard::kingCastlingMask(WHITE)))
    // {
    //     constexpr size_t featureIndex = 64 * 12 + 16 + 0;
    //     features[featureIndex] = 1.0f;
    // }
    // if (!(movedPieces & Bitboard::queenCastlingMask(WHITE)))
    // {
    //     constexpr size_t featureIndex = 64 * 12 + 16 + 1;
    //     features[featureIndex] = 1.0f;
    // }

    // if (!(movedPieces & Bitboard::kingCastlingMask(BLACK)))
    // {
    //     constexpr size_t featureIndex = 64 * 12 + 16 + 2;
    //     features[featureIndex] = 1.0f;
    // }
    // if (!(movedPieces & Bitboard::queenCastlingMask(BLACK)))
    // {
    //     constexpr size_t featureIndex = 64 * 12 + 16 + 3;
    //     features[featureIndex] = 1.0f;
    // }

    return features;
}

void Position::clear()
{
    std::fill(mailbox, mailbox + NUMBER_OF_SQUARES, NULL_PIECE);
    std::fill(pieceBB, pieceBB + NUMBER_OF_PIECES, Bitboard());
}

```

```

        std::fill(colorBB, colorBB + NUMBER_OF_PIECES, Bitboard());
        std::fill(history, history + 256, UnrecoverableState());
        hash = 0;
        gamePly = 0;
        sideToPlay = WHITE;
    }

    Position::Status Position::status() const
    {
        MoveList list(*this);
        if (list.size() == 0)
        {
            if (getCheckers() == 0)
                return STALEMATE;
            else
            {
                if (sideToPlay == WHITE) return BLACK_WIN;
                else return WHITE_WIN;
            }
        }

        // add threefold repetition by hash
        if (history[gamePly].plyFiftyMoveRule >= 50)
            return DRAW;

        return NOT_FINISHED;
    }

    void Position::set(const std::string &fen)
    {
        FenUtility::set(*this, fen);

        if (sideToPlay == BLACK)
            hash ^= Zobrist::sideToMove;

        updateNonPieceHashing(gamePly);
        recalculateCheckersAndPinned();
    }

    std::string Position::fen() const
    {
        return FenUtility::fen(*this);
    }

    Move Position::moveFromString(const std::string & move)
    {
        std::string fromStr = move.substr(0, 2);
        std::string toStr = move.substr(2, 2);
        std::string promotionPiece = "";
        if (move.size() > 4)
            promotionPiece = move.substr(4);

        Square from = static_cast<Square>(std::find(SQUARE_STRING, SQUARE_STRING
+ NUMBER_OF_SQUARES, fromStr) - SQUARE_STRING);
        Square to = static_cast<Square>(std::find(SQUARE_STRING, SQUARE_STRING +
NUMBER_OF_SQUARES, toStr) - SQUARE_STRING);
        Move::Type type = Move::Type::QUIET;

        switch (typeOf(mailbox[from])) {
        case PAWN:
            if (fileOf(from) == fileOf(to))

```

```

    {
        // non-capture
        if (std::abs(rankOf(from) - rankOf(to)) == 2) type =
Move::Type::DOUBLE_PUSH;
        else if (promotionPiece == "n") type =
Move::Type::PROMOTION_KNIGHT;
        else if (promotionPiece == "b") type =
Move::Type::PROMOTION_BISHOP;
        else if (promotionPiece == "r") type =
Move::Type::PROMOTION_ROOK;
        else if (promotionPiece == "q") type =
Move::Type::PROMOTION_QUEEN;
    }
    else
    {
        // capture
        if (mailbox[to] == NULL_PIECE) type = Move::Type::EN_PASSANT;
        else if (promotionPiece == "") type = Move::Type::CAPTURE;
        else if (promotionPiece == "n") type =
Move::Type::PROMOTION_CAPTURE_KNIGHT;
        else if (promotionPiece == "b") type =
Move::Type::PROMOTION_CAPTURE_BISHOP;
        else if (promotionPiece == "r") type =
Move::Type::PROMOTION_CAPTURE_ROOK;
        else if (promotionPiece == "q") type =
Move::Type::PROMOTION_CAPTURE_QUEEN;
    }
    break;

case KING:
    if (fileOf(from) == fileOf(E1) && fileOf(to) == fileOf(G1))
    {
        type = Move::Type::CASTLE_00;
    }
    else if (fileOf(from) == fileOf(E1) && fileOf(to) == fileOf(C1))
    {
        type = Move::Type::CASTLE_000;
    }
    else if (mailbox[to] != NULL_PIECE)
    {
        type = Move::Type::CAPTURE;
    }
    break;

case KNIGHT:
case BISHOP:
case ROOK:
case QUEEN:
    if (mailbox[to] != NULL_PIECE)
    {
        type = Move::Type::CAPTURE;
    }

    break;

    // to avoid warnings
case NULL_PIECE_TYPE:
case NUMBER_OF_PIECE_TYPES:
    break;
}

```

```

return Move(from, to, type);
}

void Position::putPiece(Piece piece, Square to)
{
    hash ^= Zobrist::pieceTable[piece][to];
    pieceBB[piece] |= Bitboard::square(to);
    colorBB[colorOf(piece)] |= Bitboard::square(to);

    mailbox[to] = piece;
}

void Position::removePiece(Square from)
{
    const Piece piece = mailbox[from];

    hash ^= Zobrist::pieceTable[piece][from];
    pieceBB[piece] &= ~Bitboard::square(from);
    colorBB[colorOf(piece)] &= ~Bitboard::square(from);

    mailbox[from] = NULL_PIECE;
}

void Position::movePieceQuiet(Square from, Square to)
{
    putPiece(mailbox[from], to);
    removePiece(from);
}

// parameter for make-unmake
void Position::updateNonPieceHashing(int updateGamePly)
{
    hash ^=
Zobrist::castlingRightsTable[Bitboard::castleRightsFromMovedPieces(history[updateGamePly].movedPieces)];

    if (history[updateGamePly].enPassantSquare != NULL_SQUARE)
        hash ^=
Zobrist::fileTable[fileOf(history[updateGamePly].enPassantSquare)];
}

std::ostream &operator<<(std::ostream &os, const Position &pos)
{
    const char* s = "  +---+---+---+---+---+---+---+---+---+\n";
    const char* t = "      A   B   C   D   E   F   G   H\n";
    os << t;
    for (int i = 56; i >= 0; i -= 8)
    {
        os << s << " " << i / 8 + 1 << " ";
        for (int j = 0; j < 8; ++j)
        {
            os << "| " << PIECE_CHAR[pos.mailbox[i + j]] << " ";
        }
        os << "| " << i / 8 + 1 << "\n";
    }
    os << s;
    os << t << "\n";

    os << "FEN: " << pos.fen() << "\n";
    os << "Hash: 0x" << std::hex << pos.hash << std::dec << "\n";
}

```

```

    return os;
}

void Position::makeMove(Move move)
{
    const Color us = sideToPlay;

    sideToPlay = ~sideToPlay;
    ++gamePly;

    const Square from = move.from();
    const Square to = move.to();
    history[gamePly] = UnrecoverableState(history[gamePly - 1]);
    history[gamePly].movedPieces |= (Bitboard::square(to) |
    Bitboard::square(from));

    if (typeOf(mailbox[from]) == PAWN)
    {
        history[gamePly].plyFiftyMoveRule = 0;
    }

    switch (move.type())
    {
        case Move::QUIET:
            movePieceQuiet(from, to);
            break;

        case Move::CAPTURE:
            history[gamePly].plyFiftyMoveRule = 0;
            history[gamePly].lastCaptured = mailbox[to];
            removePiece(to);
            putPiece(mailbox[from], to);
            removePiece(from);
            break;

        case Move::DOUBLE_PUSH:
            movePieceQuiet(from, to);
            history[gamePly].enPassantSquare = static_cast<Square>(to +
relativeDirection(us, SOUTH));
            break;

        case Move::CASTLE_00:
            if (us == WHITE)
            {
                movePieceQuiet(E1, G1);
                movePieceQuiet(H1, F1);
            }
            else
            {
                movePieceQuiet(E8, G8);
                movePieceQuiet(H8, F8);
            }
            break;

        case Move::CASTLE_000:
            if (us == WHITE)
            {
                movePieceQuiet(E1, C1);
                movePieceQuiet(A1, D1);
            }
            else
    
```

```

    {
        movePieceQuiet(E8, C8);
        movePieceQuiet(A8, D8);
    }
    break;

case Move::EN_PASSANT:
    movePieceQuiet(from, to);
    history[gamePly].lastCaptured = mailbox[static_cast<Square>(to +
relativeDirection(us, SOUTH))];
    removePiece(static_cast<Square>(to + relativeDirection(us,
SOUTH)));
    break;

case Move::PROMOTION_KNIGHT:
    removePiece(from);
    putPiece(makePiece(us, KNIGHT), to);
    break;

case Move::PROMOTION_BISHOP:
    removePiece(from);
    putPiece(makePiece(us, BISHOP), to);
    break;

case Move::PROMOTION_ROOK:
    removePiece(from);
    putPiece(makePiece(us, ROOK), to);
    break;

case Move::PROMOTION_QUEEN:
    removePiece(from);
    putPiece(makePiece(us, QUEEN), to);
    break;

case Move::PROMOTION_CAPTURE_KNIGHT:
    history[gamePly].plyFiftyMoveRule = 0;
    history[gamePly].lastCaptured = mailbox[to];
    removePiece(from);
    removePiece(to);
    putPiece(makePiece(us, KNIGHT), to);
    break;

case Move::PROMOTION_CAPTURE_BISHOP:
    history[gamePly].plyFiftyMoveRule = 0;
    history[gamePly].lastCaptured = mailbox[to];
    removePiece(from);
    removePiece(to);
    putPiece(makePiece(us, BISHOP), to);
    break;

case Move::PROMOTION_CAPTURE_ROOK:
    history[gamePly].plyFiftyMoveRule = 0;
    history[gamePly].lastCaptured = mailbox[to];
    removePiece(from);
    removePiece(to);
    putPiece(makePiece(us, ROOK), to);
    break;

case Move::PROMOTION_CAPTURE_QUEEN:
    history[gamePly].plyFiftyMoveRule = 0;
    history[gamePly].lastCaptured = mailbox[to];

```

```

        removePiece(from);
        removePiece(to);
        putPiece(makePiece(us, QUEEN), to);
        break;
    }

    hash ^= Zobrist::sideToMove;
    updateNonPieceHashing(gamePly);
    // still positive cause we made a move
    updateNonPieceHashing(gamePly - 1);
    recalculateCheckersAndPinned();
}

void Position::undoMove(Move move)
{
    const Color us = ~sideToPlay; // we moved before, we undo
    const Square from = move.from();
    const Square to = move.to();

    switch (move.type())
    {
        case Move::QUIET:
        case Move::DOUBLE_PUSH:
            movePieceQuiet(to, from);
            break;

        case Move::CAPTURE:
            movePieceQuiet(to, from);
            putPiece(history[gamePly].lastCaptured, to);
            break;

        case Move::CASTLE_00:
            if (us == WHITE)
            {
                movePieceQuiet(G1, E1);
                movePieceQuiet(F1, H1);
            }
            else
            {
                movePieceQuiet(G8, E8);
                movePieceQuiet(F8, H8);
            }
            break;

        case Move::CASTLE_000:
            if (us == WHITE)
            {
                movePieceQuiet(C1, E1);
                movePieceQuiet(D1, A1);
            }
            else
            {
                movePieceQuiet(C8, E8);
                movePieceQuiet(D8, A8);
            }
            break;

        case Move::EN_PASSANT:
            movePieceQuiet(to, from);
            putPiece(makePiece(~us, PAWN), static_cast<Square>(to +
relativeDirection(us, SOUTH)));

```

```

        break;

        case Move::PROMOTION_KNIGHT:
        case Move::PROMOTION_BISHOP:
        case Move::PROMOTION_ROOK:
        case Move::PROMOTION_QUEEN:
            removePiece(to);
            putPiece(makePiece(us, PAWN), from);
            break;

        case Move::PROMOTION_CAPTURE_KNIGHT:
        case Move::PROMOTION_CAPTURE_BISHOP:
        case Move::PROMOTION_CAPTURE_ROOK:
        case Move::PROMOTION_CAPTURE_QUEEN:
            removePiece(to);
            putPiece(makePiece(us, PAWN), from);
            putPiece(history[gamePly].lastCaptured, to);
            break;
    }

    hash ^= Zobrist::sideToMove;
    updateNonPieceHashing(gamePly);
    // still positive cause couldn't have made an undo otherwise
    updateNonPieceHashing(gamePly - 1);

    sideToPlay = ~sideToPlay;
    --gamePly;
}

Bitboard Position::getDiagonalSliders(Color color) const
{
    // need to make template and constexpr
    return pieceBB[makePiece(color, BISHOP)]
        | pieceBB[makePiece(color, QUEEN)];
}

Bitboard Position::getOrthogonalSliders(Color color) const
{
    // need to make template and constexpr
    return pieceBB[makePiece(color, ROOK)]
        | pieceBB[makePiece(color, QUEEN)];
}

void Position::recalculateCheckersAndPinned()
{
    const Color us = sideToPlay;
    const Color them = ~us;
    const Bitboard bbUs = colorBB[us];
    const Bitboard bbThem = colorBB[them];

    const Square ourKing = pieceBB[makePiece(us, KING)].lsb();

    Bitboard pinned = 0;
    Bitboard checkers = 0;

    checkers |= Bitboard::knightAttacks(ourKing) & pieceBB[makePiece(them,
    KNIGHT)];
    checkers |= Bitboard::pawnAttacksTo(us, ourKing) &
    pieceBB[makePiece(them, PAWN)];

    Bitboard xRaySliders = 0;

```

```

        xRaySliders |= Bitboard::rookAttacks(ourKing, bbThem) &
getOrthogonalSliders(them);
        xRaySliders |= Bitboard::bishopAttacks(ourKing, bbThem) &
getDiagonalSliders(them);

        while (xRaySliders)
        {
            Square slider = xRaySliders.popLsb();
            Bitboard ourPiecesBetween = Bitboard::squaresBetween(ourKing, slider)
& bbUs;
            if (ourPiecesBetween == 0)
            {
                checkers |= Bitboard::square(slider);
            }
            else if ((ourPiecesBetween & (ourPiecesBetween - 1)) == 0)
            {
                pinned |= ourPiecesBetween;
            }
        }

        history[gamePly].checkers = checkers;
        history[gamePly].pinned = pinned;
    }

    Bitboard Position::getCheckers() const
    {
        return history[gamePly].checkers;
    }

    Bitboard Position::getPinnedPieces() const
    {
        return history[gamePly].pinned;
    }

```

Код файла Square.h:

```

#ifndef CHESENGINE_SQUARE_H
#define CHESENGINE_SQUARE_H

#include <string>

enum Square: int
{
    NULL_SQUARE = -1,

    A1, B1, C1, D1, E1, F1, G1, H1,
    A2, B2, C2, D2, E2, F2, G2, H2,
    A3, B3, C3, D3, E3, F3, G3, H3,
    A4, B4, C4, D4, E4, F4, G4, H4,
    A5, B5, C5, D5, E5, F5, G5, H5,
    A6, B6, C6, D6, E6, F6, G6, H6,
    A7, B7, C7, D7, E7, F7, G7, H7,
    A8, B8, C8, D8, E8, F8, G8, H8,

    NUMBER_OF_SQUARES = 64
};

constexpr Square & operator++(Square & square)
{
    return square = static_cast<Square>(static_cast<int>(square) + 1);
}

```

```

}

constexpr int rankOf(Square square)
{
    return static_cast<int>(square >> 3);
}

constexpr int fileOf(Square square)
{
    return static_cast<int>(square & 0b111);
}

constexpr int diagonalOf(Square square)
{
    return 7 + rankOf(square) - fileOf(square);
}

constexpr int antiDiagonalOf(Square square)
{
    return rankOf(square) + fileOf(square);
}

constexpr Square mirrored(Square square)
{
    Square result = static_cast<Square>(square + 8 * (7 - rankOf(square) *
2));
    return result;
}

const std::string SQUARE_STRING[NUMBER_OF_SQUARES] =
{
    "a1", "b1", "c1", "d1", "e1", "f1", "g1", "h1",
    "a2", "b2", "c2", "d2", "e2", "f2", "g2", "h2",
    "a3", "b3", "c3", "d3", "e3", "f3", "g3", "h3",
    "a4", "b4", "c4", "d4", "e4", "f4", "g4", "h4",
    "a5", "b5", "c5", "d5", "e5", "f5", "g5", "h5",
    "a6", "b6", "c6", "d6", "e6", "f6", "g6", "h6",
    "a7", "b7", "c7", "d7", "e7", "f7", "g7", "h7",
    "a8", "b8", "c8", "d8", "e8", "f8", "g8", "h8"
};

#endif //CHESENGINE_SQUARE_H

```

Код файла NeuralNetworkEvaluator.h:

```

#ifndef NEURALNETWORKEVALUATOR_H
#define NEURALNETWORKEVALUATOR_H

#include <memory>
#include <BoardRepresentation/Color.h>
#include <cppflow/model.h>

class Position;

class NeuralNetworkEvaluator
{
public:
    NeuralNetworkEvaluator(const std::string & modelFilePath);

```

```

        int32_t evaluate(const Position & position);
        int32_t evaluate(const Position & position, Color color);

private:
    std::unique_ptr<cppflow::model> m_model;
};

#endif // NEURALNETWORKEVALUATOR_H

```

Код файла NeuralNetworkEvaluator.cpp:

```

#include "NeuralNetworkEvaluator.h"

#include <BoardRepresentation/Position.h>

NeuralNetworkEvaluator::NeuralNetworkEvaluator(const std::string &
modelFilePath)
    : m_model(new cppflow::model(modelFilePath))
{
}

int32_t NeuralNetworkEvaluator::evaluate(const Position & position)
{
    const static std::vector<int64_t> shape = {1, 1, 768};
    float normalizeFactor = 1000.0;

    cppflow::tensor input = cppflow::tensor(position.binaryFeatures(),
shape);
    cppflow::tensor output = m_model->operator()(input);

    int32_t evaluation = output.get_data<float>().front() * normalizeFactor;
    return evaluation;
}

int32_t NeuralNetworkEvaluator::evaluate(const Position & position, Color
color)
{
    return (color == WHITE) ? evaluate(position) : -evaluate(position);
}

```

Код файла SimpleEvaluator.h:

```

#ifndef SIMPLIEEVALUATOR_H
#define SIMPLIEEVALUATOR_H

class Position;
#include <inttypes.h>
#include "../BoardRepresentation/Piece.h"
#include "../BoardRepresentation/Square.h"

// https://www.chessprogramming.org/Simplified_Evaluation_Function
class SimpleEvaluator
{
public:
    // color independent eval. negative means good for black, positive - good
    for white
    static int32_t evaluate(const Position & position);
}

```

```

        // color dependent eval. Positive means good for side (param)
        static int32_t evaluate(const Position & position, Color side);
};

#endif // SIMPLIEVALUATOR_H

```

Код файла SimpleEvaluator.cpp:

```

#include "SimpleEvaluator.h"

#include <array>

#include <BoardRepresentation/Position.h>

static constexpr std::array<int32_t, NUMBER_OF_PIECE_TYPES> PIECE_VALUES =
{
    0, // null piece
    100, // pawn
    300, // knight
    300, // bishop
    500, // rook
    900, // queen
    0, // king is unvaluable
};

// location bonuses for pieces BLACK (need to mirror for white)
static constexpr std::array<std::array<int32_t, NUMBER_OF_SQUARES>,
NUMBER_OF_PIECE_TYPES> SQUARE_VALUES =
{
    {
        // null piece
        {
        },
        // pawns
        {
            0, 0, 0, 0, 0, 0, 0, 0,
            50, 50, 50, 50, 50, 50, 50, 50,
            10, 10, 20, 30, 30, 20, 10, 10,
            5, 5, 10, 25, 25, 10, 5, 5,
            0, 0, 0, 20, 20, 0, 0, 0,
            5, -5, -10, 0, 0, -10, -5, 5,
            5, 10, 10, -40, -40, 10, 10, 5,
            0, 0, 0, 0, 0, 0, 0, 0,
        },
        // knights
        {
            -50, -40, -30, -30, -30, -30, -40, -50,
            -40, -20, 0, 0, 0, 0, -20, -40,
            -30, 0, 7, 15, 15, 7, 0, -30,
            -30, 5, 15, 20, 20, 15, 5, -30,
            -30, 0, 15, 20, 20, 15, 0, -30,
            -30, 5, 7, 15, 15, 7, 5, -30,
            -40, -20, 0, 5, 5, 0, -20, -40,
            -50, -40, -30, -30, -30, -30, -40, -50,
        },
        // bishops

```

```

{
    -20,-10,-10,-10,-10,-10,-10,-20,
    -10, 0, 0, 0, 0, 0, 0,-10,
    -10, 0, 5, 10, 10, 5, 0,-10,
    -10, 5, 5, 10, 10, 5, 5,-10,
    -10, 0, 10, 10, 10, 10, 0,-10,
    -10, 10, 10, 10, 10, 10, 10,-10,
    -10, 5, 0, 0, 0, 0, 5,-10,
    -20,-10,-10,-10,-10,-10,-10,-20,
},

// rooks
{
    0, 0, 0, 0, 0, 0, 0, 0,
    5, 10, 10, 10, 10, 10, 10, 5,
    -5, 0, 0, 0, 0, 0, 0, -5,
    -5, 0, 0, 0, 0, 0, 0, -5,
    -5, 0, 0, 0, 0, 0, 0, -5,
    -5, 0, 0, 0, 0, 0, 0, -5,
    -5, 0, 0, 0, 0, 0, 0, -5,
    0, 0, 0, 5, 5, 0, 0, 0,
},

// queens
{
    -20,-10,-10, -5, -5,-10,-10,-20,
    -10, 0, 0, 0, 0, 0, 0,-10,
    -10, 0, 5, 5, 5, 5, 0,-10,
    -5, 0, 5, 5, 5, 5, 0, -5,
    0, 0, 5, 5, 5, 5, 0, -5,
    -10, 5, 5, 5, 5, 5, 0,-10,
    -10, 0, 5, 0, 0, 0, 0,-10,
    -20,-10,-10, -5, -5,-10,-10,-20,
},

// king
{
//      -30,-40,-40,-50,-50,-40,-40,-30,
//      -30,-40,-40,-50,-50,-40,-40,-30,
//      -30,-40,-40,-50,-50,-40,-40,-30,
//      -30,-40,-40,-50,-50,-40,-40,-30,
//      -20,-30,-30,-40,-40,-30,-30,-20,
//      -10,-20,-20,-20,-20,-20,-20,-10,
//      20, 20, 0, 0, 0, 0, 20, 20,
//      20, 30, 10, 0, 0, 10, 30, 20,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 20, 0, -1, -1, 0, 20, 0,
}
}
};

```

```

static constexpr auto generatePieceSquareValues()
{

```

```

        std::array<std::array<std::array<int32_t,          NUMBER_OF_SQUARES>,
NUMBER_OF_PIECE_TYPES>, NUMBER_OF_COLORS> result = {};

        for (PieceType pieceType = PieceType::PAWN; pieceType <
PieceType::NUMBER_OF_PIECE_TYPES; pieceType =
static_cast<PieceType>(pieceType + 1))
        {
            for (Square square = A1; square < NUMBER_OF_SQUARES; ++square)
            {
                result[WHITE][pieceType][square] = PIECE_VALUES[pieceType] +
SQUARE_VALUES[pieceType][mirrored(square)];
                result[BLACK][pieceType][square] = -(PIECE_VALUES[pieceType] +
SQUARE_VALUES[pieceType][square]);
            }
        }

        return result;
    }

    static constexpr std::array<std::array<std::array<int32_t,
NUMBER_OF_SQUARES>,          NUMBER_OF_PIECE_TYPES>,          NUMBER_OF_COLORS>
PIECE_SQUARE_VALUES = generatePieceSquareValues();
//static constexpr int32_t
PIECE_SQUARE_VALUES[NUMBER_OF_COLORS][NUMBER_OF_PIECE_TYPES][NUMBER_OF_SQUARE
S]

int32_t SimpleEvaluator::evaluate(const Position & position)
{
    int32_t evaluation = 0;

    for (Square sq = A1; sq < NUMBER_OF_SQUARES; ++sq)
    {
        if (position.mailbox[sq] != NULL_PIECE)
        {
            const Piece piece = position.mailbox[sq];
            evaluation +=
PIECE_SQUARE_VALUES[colorOf(piece)][typeOf(piece)][sq];
        }
    }

    return evaluation;
}

int32_t SimpleEvaluator::evaluate(const Position & position, Color side)
{
    return (side == WHITE) ? evaluate(position) : -evaluate(position);
}

```

Код файла MoveGenerator.h:

```

#ifndef CHESENGINE_MOVEGENERATOR_H
#define CHESENGINE_MOVEGENERATOR_H

#include "../BoardRepresentation/Move.h"
#include "../BoardRepresentation/Bitboard.h"

class MoveList;
class Position;

```

```

class MoveGenerator
{
public:
    static void generateLegalMoves(MoveList & list, const Position &
position);

private:
    static void generateKingMoves(MoveList & list, const Position &
position);
    static void generatePawnMoves(MoveList & list, const Position & position,
Bitboard quietMask, Bitboard captureMask, Bitboard pinned);
    static void generateKnightMoves(MoveList & list, const Position &
position, Bitboard quietMask, Bitboard captureMask, Bitboard pinned);
    static void generateSlidingMoves(MoveList & list, const Position &
position, Bitboard quietMask, Bitboard captureMask, Bitboard pinned);

    template<Move::Type type>
    static void generateMoves(MoveList & list, Square from, Bitboard to);
};

#endif //CHESENGINE_MOVEGENERATOR_H

```

Код файла MoveGenerator.cpp:

```

#include <cassert>
#include "MoveGenerator.h"
#include "../BoardRepresentation/Position.h"
#include "MoveList.h"

void MoveGenerator::generateLegalMoves(MoveList &list, const Position
&position)
{
    const Color us = position.sideToPlay;
    const Color them = ~us;

    const Bitboard bbUs = position.colorBB[us];
    const Bitboard bbThem = position.colorBB[them];
    const Bitboard bbAll = bbUs | bbThem;

    const Bitboard checkers = position.getCheckers();

    Bitboard captureMask = bbThem;
    Bitboard quietMask = ~bbAll;

    generateKingMoves(list, position);
    int numberOfCheckers = checkers.populationCount();
    if (numberOfCheckers == 2)
    {
        return; // only king moves
    }
    else if (numberOfCheckers == 1)
    {
        captureMask = checkers;

        Square checker = checkers.lsb();

        if (position.mailbox[checker] == makePiece(them, PAWN) ||
            position.mailbox[checker] == makePiece(them, KNIGHT))
        {

```

```

        quietMask = 0;
    }
    else
    {
        Square ourKing = position.pieceBB[makePiece(us, KING)].lsb();
        quietMask = Bitboard::squaresBetween(ourKing, checker);
    }
}
generateSlidingMoves(list, position, quietMask, captureMask,
position.getPinnedPieces());
generateKnightMoves(list, position, quietMask, captureMask,
position.getPinnedPieces());
generatePawnMoves(list, position, quietMask, captureMask,
position.getPinnedPieces());
}

```

```

template<Move::Type type>
void MoveGenerator::generateMoves(MoveList &list, Square from, Bitboard to)
{
    while (to)
    {
        if constexpr (type == Move::PROMOTION)
        {
            Square sq = to.popLsb();
            list.push_back(Move(from, sq, Move::PROMOTION_QUEEN));
            list.push_back(Move(from, sq, Move::PROMOTION_KNIGHT));
            list.push_back(Move(from, sq, Move::PROMOTION_BISHOP));
            list.push_back(Move(from, sq, Move::PROMOTION_ROOK));
        }
        else if constexpr (type == Move::PROMOTION_CAPTURE)
        {
            Square sq = to.popLsb();
            list.push_back(Move(from, sq, Move::PROMOTION_CAPTURE_QUEEN));
            list.push_back(Move(from, sq, Move::PROMOTION_CAPTURE_KNIGHT));
            list.push_back(Move(from, sq, Move::PROMOTION_CAPTURE_BISHOP));
            list.push_back(Move(from, sq, Move::PROMOTION_CAPTURE_ROOK));
        }
        else
        {
            list.push_back(Move(from, to.popLsb(), type));
        }
    }
}

```

```

void MoveGenerator::generateKingMoves(MoveList &list, const Position
&position)
{
    const Color us = position.sideToPlay;
    const Color them = ~us;

    const Bitboard bbUs = position.colorBB[us];
    const Bitboard bbThem = position.colorBB[them];
    const Bitboard bbAll = bbUs | bbThem;

    const Square ourKing = position.pieceBB[makePiece(us, KING)].lsb();
    const Square theirKing = position.pieceBB[makePiece(them, KING)].lsb();

    // squares attacked by enemy pieces
    Bitboard attackedSquares = 0ull;
    attackedSquares |= Bitboard::pawnAttacks(them,
position.pieceBB[makePiece(them, PAWN)]);
}

```

```

    attackedSquares |= Bitboard::kingAttacks(theirKing);
    Bitboard theirKnights = position.pieceBB[makePiece(them, KNIGHT)];
    while (theirKnights) attackedSquares |=
Bitboard::knightAttacks(theirKnights.popLsb());
    // exclude the king to prevent it from moving behind itself (which still
will be attacked)
    Bitboard theirDiagSliders = position.getDiagonalSliders(them);
    while (theirDiagSliders) attackedSquares |=
Bitboard::bishopAttacks(theirDiagSliders.popLsb(), bbAll ^
Bitboard::square(ourKing));
    Bitboard theirOrthogonalSliders = position.getOrthogonalSliders(them);
    while (theirOrthogonalSliders) attackedSquares |=
Bitboard::rookAttacks(theirOrthogonalSliders.popLsb(), bbAll ^
Bitboard::square(ourKing));

    Bitboard kingMoves = Bitboard::kingAttacks(ourKing) & ~(bbUs |
attackedSquares); // we cannot move to attacked squares and our squares;
    generateMoves<Move::Type::QUIET>(list, ourKing, kingMoves & (~bbThem));
    generateMoves<Move::Type::CAPTURE>(list, ourKing, kingMoves & bbThem);

    // generate castling
    const Bitboard movedPieces =
position.history[position.gamePly].movedPieces;

    if ( !(movedPieces & Bitboard::kingCastlingMask(us)) &&
        !(((bbAll & (~Bitboard::square(ourKing))) | attackedSquares) &
Bitboard::kingCastlingBlockers(us)))
    {
        if (us == WHITE) list.push_back(Move(E1, G1, Move::CASTLE_00));
        else list.push_back(Move(E8, G8, Move::CASTLE_00));
    }

    if ( !(movedPieces & Bitboard::queenCastlingMask(us)) &&
        !(attackedSquares & Bitboard::queenCastlingBlockers(us)) &&
        !(bbAll & Bitboard::squaresBetween(ourKing, us == WHITE ? A1 : A8)))
    {
        if (us == WHITE) list.push_back(Move(E1, C1, Move::CASTLE_000));
        else list.push_back(Move(E8, C8, Move::CASTLE_000));
    }
}

void MoveGenerator::generatePawnMoves(MoveList &list, const Position
&position, Bitboard quietMask, Bitboard captureMask, Bitboard pinned)
{
    const Color us = position.sideToPlay;
    const Color them = ~us;
    const Square ourKing = position.pieceBB[makePiece(us, KING)].lsb();

    const Bitboard relativeRank2 = (us == WHITE ? Bitboard::rank(A2) :
Bitboard::rank(A7));
    const Bitboard relativeRank4 = (us == WHITE ? Bitboard::rank(A4) :
Bitboard::rank(A5));
    const Bitboard relativeRank7 = (us == WHITE ? Bitboard::rank(A7) :
Bitboard::rank(A2));

    const Bitboard bbUs = position.colorBB[us];
    const Bitboard bbThem = position.colorBB[them];
    const Bitboard bbAll = bbUs | bbThem;

    const Bitboard ourPawns = position.pieceBB[makePiece(us, PAWN)];
    const Bitboard pinnedPawns = ourPawns & pinned;

```

```

const Bitboard nonPinnedPawns = ourPawns & (~pinned);

// check en-passant moves first
const Square enPassantSq =
position.history[position.gamePly].enPassantSquare;
if ((enPassantSq != NULL_SQUARE) &&
    (captureMask &
    Bitboard::square(enPassantSq).shifted(relativeDirection(them, NORTH))))
{
    const Bitboard theirOrthogonalSliders =
position.getOrthogonalSliders(them);

    Bitboard nonPinnedCandidate = Bitboard::pawnAttacksTo(them,
enPassantSq) & nonPinnedPawns;
    while (nonPinnedCandidate)
    {
        Square square = nonPinnedCandidate.popLsb();

        Bitboard occupancyAfterCapture = bbAll &
(~Bitboard::square(square)) &
(~Bitboard::square(enPassantSq).shifted(relativeDirection(them, NORTH)));
        if ((Bitboard::slidingAttacks(ourKing, occupancyAfterCapture,
Bitboard::rank(ourKing)) &
            theirOrthogonalSliders) == 0)
            // when the pawn is not pinned, but en-passant discovers an
orthogonal slider check
            {
                list.push_back(Move(square, enPassantSq, Move::EN_PASSANT));
            }
    }

    Bitboard pinnedCandidates = Bitboard::pawnAttacksTo(them,
enPassantSq) & pinnedPawns
                                & Bitboard::line(enPassantSq, ourKing);
    while (pinnedCandidates) // there can only be 1 such candidate -
optimize it
    {
        list.push_back(Move(pinnedCandidates.popLsb(), enPassantSq,
Move::EN_PASSANT));
    }
}

// generate promotions
{
    {
        Bitboard nonPinnedCandidates = nonPinnedPawns & relativeRank7;
        while (nonPinnedCandidates)
        {
            Square sq = nonPinnedCandidates.popLsb();

            Bitboard attacks = Bitboard::pawnAttacksTo(us, sq) &
captureMask;
            generateMoves<Move::Type::PROMOTION_CAPTURE>(list, sq,
attacks);

            Bitboard push =
Bitboard::square(sq).shifted(relativeDirection(us, NORTH)) & quietMask;
            generateMoves<Move::Type::PROMOTION>(list, sq, push);
        }
    }
}

```

```

    {
        Bitboard pinnedCandidates = pinnedPawns & relativeRank7;
        while (pinnedCandidates)
        {
            Square sq = pinnedCandidates.popLsb();

            Bitboard attacks = Bitboard::pawnAttacksTo(us, sq) &
captureMask & Bitboard::line(sq, ourKing);
            generateMoves<Move::Type::PROMOTION_CAPTURE>(list, sq,
attacks);

            Bitboard push =
Bitboard::square(sq).shifted(relativeDirection(us, NORTH)) & quietMask &
Bitboard::line(sq, ourKing);
            generateMoves<Move::Type::PROMOTION>(list, sq, push);
        }
    }

    // generate non-promotion captures and pushes
    {
        {
            Bitboard nonPinnedCandidates = nonPinnedPawns & (~relativeRank7);
            while (nonPinnedCandidates)
            {
                Square sq = nonPinnedCandidates.popLsb();

                Bitboard attacks = Bitboard::pawnAttacksTo(us, sq) &
captureMask;
                generateMoves<Move::Type::CAPTURE>(list, sq, attacks);

                Bitboard push =
Bitboard::square(sq).shifted(relativeDirection(us, NORTH)) & quietMask;
                generateMoves<Move::Type::QUIET>(list, sq, push);
            }
        }

        {
            Bitboard pinnedCandidates = pinnedPawns & (~relativeRank7);
            while (pinnedCandidates)
            {
                Square sq = pinnedCandidates.popLsb();

                Bitboard attacks = Bitboard::pawnAttacksTo(us, sq) &
captureMask & Bitboard::line(ourKing, sq);
                generateMoves<Move::Type::CAPTURE>(list, sq, attacks);

                Bitboard push =
Bitboard::square(sq).shifted(relativeDirection(us, NORTH)) & quietMask &
Bitboard::line(ourKing, sq);
                generateMoves<Move::Type::QUIET>(list, sq, push);
            }
        }
    }

    // generate double pushes
    {
        {
            Bitboard pinnedSecondRank = pinnedPawns & relativeRank2;
            while (pinnedSecondRank)

```

```

        {
            Square sq = pinnedSecondRank.popLsb();
            if (position.mailbox[sq + relativeDirection(us, NORTH)] ==
NULL_PIECE)
                generateMoves<Move::DOUBLE_PUSH>(list, sq, relativeRank4
& quietMask & Bitboard::line(sq, ourKing) & Bitboard::file(sq));
        }
    }

    {
        Bitboard nonPinnedSecondRank = nonPinnedPawns & relativeRank2;
        while (nonPinnedSecondRank)
        {
            Square sq = nonPinnedSecondRank.popLsb();
            if (position.mailbox[sq + relativeDirection(us, NORTH)] ==
NULL_PIECE)
                generateMoves<Move::DOUBLE_PUSH>(list, sq, relativeRank4
& quietMask & Bitboard::file(sq));
        }
    }
}

void MoveGenerator::generateKnightMoves(MoveList &list, const Position
&position, Bitboard quietMask, Bitboard captureMask, Bitboard pinned)
{
    const Color us = position.sideToPlay;

    Bitboard ourKnights = position.pieceBB[makePiece(us, KNIGHT)] &
(~pinned);

    while (ourKnights)
    {
        Square knight = ourKnights.popLsb();
        Bitboard knightAttacks = Bitboard::knightAttacks(knight);

        generateMoves<Move::Type::CAPTURE>(list, knight, knightAttacks &
captureMask);
        generateMoves<Move::Type::QUIET>(list, knight, knightAttacks &
quietMask);
    }
}

void MoveGenerator::generateSlidingMoves(MoveList &list, const Position
&position, Bitboard quietMask, Bitboard captureMask, Bitboard pinned)
{
    const Color us = position.sideToPlay;
    const Color them = ~us;

    const Bitboard bbUs = position.colorBB[us];
    const Bitboard bbThem = position.colorBB[them];
    const Bitboard bbAll = bbUs | bbThem;

    const Square ourKing = position.pieceBB[makePiece(us, KING)].lsb();

    const Bitboard ourOrthogonalSliders = position.getOrthogonalSliders(us);
    const Bitboard ourDiagonalSliders = position.getDiagonalSliders(us);

    {
        Bitboard pinnedDiagonal = ourDiagonalSliders & pinned;
        while (pinnedDiagonal)

```

```

    {
        Square pieceSq = pinnedDiagonal.popLsb();
        // only in the LINE direction
        Bitboard diagAttacks = Bitboard::bishopAttacks(pieceSq, bbAll) &
        Bitboard::line(ourKing, pieceSq);

        generateMoves<Move::Type::CAPTURE>(list, pieceSq, diagAttacks &
        captureMask);
        generateMoves<Move::Type::QUIET>(list, pieceSq, diagAttacks &
        quietMask);
    }
}

{
    Bitboard nonPinnedDiagonal = ourDiagonalSliders & (~pinned);
    while (nonPinnedDiagonal)
    {
        Square pieceSq = nonPinnedDiagonal.popLsb();
        Bitboard diagAttacks = Bitboard::bishopAttacks(pieceSq, bbAll);

        generateMoves<Move::Type::CAPTURE>(list, pieceSq, diagAttacks &
        captureMask);
        generateMoves<Move::Type::QUIET>(list, pieceSq, diagAttacks &
        quietMask);
    }
}

{
    Bitboard pinnedOrthogonal = ourOrthogonalSliders & pinned;
    while (pinnedOrthogonal)
    {
        Square pieceSq = pinnedOrthogonal.popLsb();
        // only in the LINE direction
        Bitboard orthAttacks = Bitboard::rookAttacks(pieceSq, bbAll) &
        Bitboard::line(ourKing, pieceSq);

        generateMoves<Move::Type::CAPTURE>(list, pieceSq, orthAttacks &
        captureMask);
        generateMoves<Move::Type::QUIET>(list, pieceSq, orthAttacks &
        quietMask);
    }
}

{
    Bitboard nonPinnedOrthogonal = ourOrthogonalSliders & (~pinned);
    while (nonPinnedOrthogonal)
    {
        Square pieceSq = nonPinnedOrthogonal.popLsb();
        Bitboard orthAttacks = Bitboard::rookAttacks(pieceSq, bbAll);

        generateMoves<Move::Type::CAPTURE>(list, pieceSq, orthAttacks &
        captureMask);
        generateMoves<Move::Type::QUIET>(list, pieceSq, orthAttacks &
        quietMask);
    }
}
}

```

Код файлу MoveList.h:

```

#ifndef CHESENGINE_MOVELIST_H
#define CHESENGINE_MOVELIST_H

#include "../BoardRepresentation/Move.h"
class Position;

class MoveList
{
public:
    explicit MoveList (const Position & position);

    Move * begin()
    {
        return moveList;
    }

    Move * end()
    {
        return last;
    }

    uint32_t size() const
    {
        return last - moveList;
    }

    void push_back(Move move)
    {
        *last++ = move;
    }

    void orderMoves(const Position & position);

private:
    static constexpr uint32_t MAX_MOVES = 256; // from stockfish
    Move moveList[MAX_MOVES];
    Move * last;
};

#endif //CHESENGINE_MOVELIST_H

```

Код файла MoveList.cpp:

```

#include "MoveList.h"
#include "../BoardRepresentation/Position.h"
#include "MoveGenerator.h"

#include <vector>

MoveList::MoveList(const Position &position)
    : last(moveList)
{
    MoveGenerator::generateLegalMoves(*this, position);
}

void MoveList::orderMoves(const Position & position)
{
    std::vector<std::pair<int32_t, Move>> prioritizedMoves;

```

```

        const Color them = ~position.sideToPlay;
        const Bitboard theirPawnAttacks = Bitboard::pawnAttacks(them,
position.pieceBB[makePiece(them, PAWN)]);

        for (Move move: *this)
        {
            int32_t priority = 0;

            if (move.isCapture()) priority += 10;
            if (move.isPromotion()) priority += 5;
            if (move.isCastle()) priority += 1;

            if (move.isQuiet())
            {
                // if we move a piece to a square that is attacked by enemy pawns
                if (Bitboard::square(move.to()) & theirPawnAttacks)
                {
                    priority -= 2;
                }
            }

            prioritizedMoves.push_back({priority, move});
        }

        std::sort(prioritizedMoves.begin(), prioritizedMoves.end(),
        rhs) [(const std::pair<int32_t, Move> & lhs, const std::pair<int32_t, Move> &
        {
            return lhs.first > rhs.first;
        }));
        last = moveList;

        for (const auto &[p, move]: prioritizedMoves)
        {
            (void) p;
            this->push_back(move);
        }
    }
}

```

Код файла DefaultSearcher.h:

```

#ifndef DEFAULTSEARCHER_H
#define DEFAULTSEARCHER_H

class Position;
class Move;

#include "CoreConfig.h"

#include <vector>
#include <inttypes.h>
#include <tuple>

#include <BoardRepresentation/Color.h>
#include <MoveGeneration/MoveList.h>

class NeuralNetworkEvaluator;

class DefaultSearcher

```

```

{
public:
    uint64_t timeTaken;
    uint32_t nodesReached;
    uint32_t nodesEvaluated;
    uint32_t nodesCutOff;

public:
    explicit DefaultSearcher(NeuralNetworkEvaluator * nnEvaluator = nullptr);
private:
    NeuralNetworkEvaluator * nnEvaluator;

public:
    std::tuple<Move, int32_t> iterativeDeepeningSearch(const Position &
position, uint32_t maxDepth);
    std::tuple<Move, int32_t> findBestMove(const Position & position,
uint32_t depth);
    int32_t search(Position & position, uint32_t depth, uint32_t plyFromRoot,
int32_t alpha, int32_t beta);
    int32_t quiescenceSearch(Position & position, uint32_t plyFromRoot,
int32_t alpha, int32_t beta);
    int32_t evaluatePosition(const Position & position);

private:
    Move knownBestMove = Move(NULL_SQUARE, NULL_SQUARE);
    void orderMoves(MoveList & moveList, const Position & pos, Move bestMove
= Move(NULL_SQUARE, NULL_SQUARE));
};

#endif // DEFAULTSEARCHER_H

```

Код файлу DefaultSearcher.cpp:

```

#include "DefaultSearcher.h"

#include <MoveGeneration/MoveList.h>
#include <BoardRepresentation/Position.h>
#include <Evaluation/SimpleEvaluator.h>
#include <chrono>
#include <stdexcept>

static constexpr int32_t positiveInfinity = 100000;
static constexpr int32_t negativeInfinity = -positiveInfinity;
static constexpr int32_t checkmateScore = 10000;

#include <Evaluation/NeuralNetworkEvaluator.h>

DefaultSearcher::DefaultSearcher(NeuralNetworkEvaluator * nnEvaluator)
    : nnEvaluator(nnEvaluator)
{
}

std::tuple<Move, int32_t> DefaultSearcher::iterativeDeepeningSearch(const
Position & position, uint32_t maxDepth)
{
    auto start = std::chrono::high_resolution_clock::now();

    std::tuple<Move, int32_t> result;
    for (uint32_t depth = 1; depth <= maxDepth; ++depth)

```

```

    {
        result = findBestMove(position, depth);
        std::tie(knownBestMove, std::ignore) = result;
    }

    auto finish = std::chrono::high_resolution_clock::now();
    timeTaken = std::chrono::duration_cast<std::chrono::microseconds>(finish
- start).count();

    return result;
}

std::tuple<Move, int32_t> DefaultSearcher::findBestMove(const Position &
position, uint32_t depth)
{
    if (depth == 0)
        throw std::runtime_error("Cannot find any moves at depth 0");

    std::tuple<Move, int32_t> bestMove = {Move(NULL_SQUARE, NULL_SQUARE),
negativeInfinity};

    Position copiedPosition = position;
    MoveList moveList(copiedPosition);
    orderMoves(moveList, position, knownBestMove);

    int32_t alpha = negativeInfinity;
    for (Move move: moveList)
    {
        copiedPosition.makeMove(move);
        // since we don't know other beta's here yet
        int32_t evaluation = -search(copiedPosition, depth - 1, 1,
negativeInfinity, -alpha);
        copiedPosition.undoMove(move);

        if (evaluation > alpha)
        {
            alpha = evaluation;
            bestMove = {move, evaluation};
        }
    }

    return bestMove;
}

int32_t DefaultSearcher::search(Position & position, uint32_t depth, uint32_t
plyFromRoot, int32_t alpha, int32_t beta)
{
    if (depth == 0)
    {
        return quiescenceSearch(position, plyFromRoot, alpha, beta);
    }

    ++nodesReached;

    if (position.history[position.gamePly].plyFiftyMoveRule == 50)
    {
        // draw;
        return 0;
    }

    MoveList moveList(position);

```

```

        moveList.orderMoves(position);
        if (moveList.size() == 0)
        {
            if (position.getCheckers() != 0)
            {
                // checkmate, subtract the ply because the earlier checkmate the
                better
                return -(checkmateScore - plyFromRoot * 10);
            }

            // stalemate
            return 0;
        }

        for (Move move: moveList)
        {
            position.makeMove(move);
            int32_t evaluation = -search(position, depth - 1, plyFromRoot + 1, -
            beta, -alpha);
            position.undoMove(move);

            // move is TOO good
            if (evaluation >= beta)
            {
                ++nodesCutoff;
                return beta;
            }

            if (evaluation > alpha)
            {
                alpha = evaluation;
            }
        }

        return alpha;
    }

    // simplified quiescence search
    int32_t DefaultSearcher::quiescenceSearch(Position & position, uint32_t
    plyFromRoot, int32_t alpha, int32_t beta)
    {
        ++nodesEvaluated;
        ++nodesReached;

        int evaluation = this->evaluatePosition(position);
        if (evaluation >= beta)
        {
            // cut off
            ++nodesCutoff;
            return beta;
        }

        if (evaluation > alpha)
        {
            alpha = evaluation;
        }

        MoveList moveList(position);
        if (moveList.size() == 0)
        {
            if (position.getCheckers() != 0)

```

```

        {
            // checkmate, subtract the ply because the earlier checkmate the
better        return -(checkmateScore - plyFromRoot * 10);
        }

        // stalemate
        return 0;
    }

    for (Move move: moveList)
    {
        if (!move.isCapture()) continue;

        position.makeMove(move);
        evaluation = -quiescenceSearch(position, plyFromRoot + 1, -beta, -
alpha);
        position.undoMove(move);

        if (evaluation >= beta)
        {
            ++nodesCutoff;
            return beta;
        }

        if (evaluation > alpha)
        {
            alpha = evaluation;
        }
    }

    return alpha;
}

int32_t DefaultSearcher::evaluatePosition(const Position & position)
{
    if (nnEvaluator == nullptr)
    {
        return SimpleEvaluator::evaluate(position, position.sideToPlay);
    }
    else
    {
        return nnEvaluator->evaluate(position, position.sideToPlay);
    }
}

void DefaultSearcher::orderMoves(MoveList & moveList, const Position & pos,
Move bestMove)
{
    moveList.orderMoves(pos);

    // set the move we know is best to this move
    if (bestMove.isValid())
    {
        Move * move = std::find(moveList.begin(), moveList.end(), bestMove);
        if (move != moveList.end())
        {
            std::iter_swap(moveList.begin(), move);
        }
    }
}

```

Код файла PerformanceTester.h:

```
#ifndef PERFOMANCETESTER_H
#define PERFOMANCETESTER_H

class Position;
#include <inttypes.h>

class PerfomanceTester
{
public:
    static uint64_t perft(Position & pos, uint32_t depth);
    static uint64_t perftdiv(Position & pos, uint32_t depth);
};

#endif // PERFOMANCETESTER_H
```

Код файла PerformanceTester.cpp:

```
#include "PerformanceTester.h"
#include "../BoardRepresentation/Position.h"
#include "../MoveGeneration/MoveList.h"

#include <iostream>

uint64_t PerfomanceTester::perft(Position & pos, uint32_t depth)
{
    if (depth == 0) return 1;

    MoveList moveList(pos);
    if (depth == 1) return moveList.size();

    uint64_t nodes = 0;
    for (Move move: moveList)
    {
        pos.makeMove(move);
        nodes += perft(pos, depth - 1);
        pos.undoMove(move);
    }

    return nodes;
}

uint64_t PerfomanceTester::perftdiv(Position & pos, uint32_t depth)
{
    if (depth == 0)
        return 1;

    MoveList moveList(pos);
    uint64_t totalNodes = 0;

    for (Move move: moveList)
    {
        pos.makeMove(move);
        uint64_t nodes = perft(pos, depth - 1);
        pos.undoMove(move);
    }
}
```

```

        std::cout << move.toString() << ": " << nodes << std::endl;
        totalNodes += nodes;
    }

    return totalNodes;
}

```

Код файла PrincipalVariationSearcher.h:

```

#ifndef PRINCIPALVARIATIONSEARCHER_H
#define PRINCIPALVARIATIONSEARCHER_H

#include <tuple>
#include <BoardRepresentation/Position.h>
#include <MoveGeneration/MoveList.h>
class NeuralNetworkEvaluator;

class PrincipalVariationSearcher
{
public:
    explicit PrincipalVariationSearcher(NeuralNetworkEvaluator * nnEvaluator
= nullptr);

public:
    std::tuple<Move, int32_t> iterativeDeepeningSearch(const Position &
position, uint32_t maxDepth);
    std::tuple<Move, int32_t> findBestMove(const Position & position,
uint32_t depth);

    int32_t search(Position & position, uint32_t depth, uint32_t plyFromRoot,
int32_t alpha, int32_t beta);
    int32_t quiescenceSearch(Position & position, uint32_t plyFromRoot,
int32_t alpha, int32_t beta);
    int32_t evaluatePosition(const Position & position);

    uint32_t getMaxDepth() const;

private:
    void orderMoves(MoveList & moveList, const Position & pos, uint32_t
plyFromRoot);

    NeuralNetworkEvaluator * nnEvaluator;
    std::vector<Move> principalVariation;
    uint32_t maxReachedDepth = 0;
};

#endif // PRINCIPALVARIATIONSEARCHER_H

```

Код файла PrincipalVariationSearcher.cpp:

```

#include "PrincipalVariationSearcher.h"

#include <Evaluation/NeuralNetworkEvaluator.h>
#include <Evaluation/SimpleEvaluator.h>

static constexpr int32_t positiveInfinity = 100000;
static constexpr int32_t negativeInfinity = -positiveInfinity;
static constexpr int32_t checkmateScore = 10000;

```

```

PrincipalVariationSearcher::PrincipalVariationSearcher(NeuralNetworkEvaluator
* nnEvaluator)
    : nnEvaluator(nnEvaluator)
{
    constexpr uint32_t MAX_SEARCH_DEPTH = 256;
    principalVariation.resize(MAX_SEARCH_DEPTH, Move());
}

std::tuple<Move, int32_t>
PrincipalVariationSearcher::iterativeDeepeningSearch(const Position &
position, uint32_t maxDepth)
{
    maxReachedDepth = 0;
    std::tuple<Move, int32_t> result;
    for (uint32_t depth = 1; depth <= maxDepth; ++depth)
    {
        result = findBestMove(position, depth);
    }
    return result;
}

std::tuple<Move, int32_t> PrincipalVariationSearcher::findBestMove(const
Position & position, uint32_t depth)
{
    if (depth == 0)
        throw std::runtime_error("Cannot find any moves at depth 0");

    std::tuple<Move, int32_t> bestMove = {Move(NULL_SQUARE, NULL_SQUARE),
negativeInfinity};

    Position copiedPosition = position;
    MoveList moveList(copiedPosition);
    orderMoves(moveList, copiedPosition, 0);

    int32_t alpha = negativeInfinity;
    for (Move move: moveList)
    {
        copiedPosition.makeMove(move);
        int32_t evaluation = -search(copiedPosition, depth - 1, 1,
negativeInfinity, -alpha);
        copiedPosition.undoMove(move);

        if (evaluation > alpha)
        {
            alpha = evaluation;
            bestMove = {move, evaluation};
            principalVariation[0] = move;
        }
    }

    return bestMove;
}

int32_t PrincipalVariationSearcher::search(Position & position, uint32_t
depth, uint32_t plyFromRoot, int32_t alpha, int32_t beta)
{
    maxReachedDepth = std::max(maxReachedDepth, plyFromRoot);
    if (depth == 0)
    {
        return quiescenceSearch(position, plyFromRoot, alpha, beta);
    }

```

```

    }

    if (position.history[position.gamePly].plyFiftyMoveRule == 50)
    {
        // draw;
        return 0;
    }

    MoveList moveList(position);
    orderMoves(moveList, position, plyFromRoot);
    if (moveList.size() == 0)
    {
        if (position.getCheckers() != 0)
        {
            // checkmate, subtract the ply because the earlier checkmate the
better
            return -(checkmateScore - plyFromRoot * 10);
        }

        // stalemate
        return 0;
    }

    for (Move move: moveList)
    {
        position.makeMove(move);
        int32_t evaluation = -search(position, depth - 1, plyFromRoot + 1, -
beta, -alpha);
        position.undoMove(move);

        if (evaluation >= beta)
        {
            // move is TOO good
            return beta;
        }

        if (evaluation > alpha)
        {
            alpha = evaluation;
            principalVariation[plyFromRoot] = move;
        }
    }

    return alpha;
}

int32_t PrincipalVariationSearcher::quiescenceSearch(Position & position,
uint32_t plyFromRoot, int32_t alpha, int32_t beta)
{
    maxReachedDepth = std::max(maxReachedDepth, plyFromRoot);
    int evaluation = this->evaluatePosition(position);
    if (evaluation >= beta)
    {
        return beta;
    }

    if (evaluation > alpha)
    {
        alpha = evaluation;
    }
}

```

```

MoveList moveList(position);
orderMoves(moveList, position, plyFromRoot);
if (moveList.size() == 0)
{
    if (position.getCheckers() != 0)
    {
        // checkmate, subtract the ply because the earlier checkmate the
better
        return -(checkmateScore - plyFromRoot * 10);
    }

    // stalemate
    return 0;
}

for (Move move: moveList)
{
    if (!move.isCapture()) continue;

    position.makeMove(move);
    evaluation = -quiescenceSearch(position, plyFromRoot + 1, -beta, -
alpha);
    position.undoMove(move);

    if (evaluation >= beta)
    {
        return beta;
    }

    if (evaluation > alpha)
    {
        alpha = evaluation;
        principalVariation[plyFromRoot] = move;
    }
}

return alpha;
}

int32_t PrincipalVariationSearcher::evaluatePosition(const Position &
position)
{
    if (nnEvaluator == nullptr)
    {
        return SimpleEvaluator::evaluate(position, position.sideToPlay);
    }
    else
    {
        return nnEvaluator->evaluate(position, position.sideToPlay);
    }
}

void PrincipalVariationSearcher::orderMoves(MoveList & moveList, const
Position & pos, uint32_t plyFromRoot)
{
    moveList.orderMoves(pos);

    // set the move we know is best to this move
    Move bestMove = principalVariation[plyFromRoot];
    if (bestMove.isValid())
    {

```

```

        Move * move = std::find(moveList.begin(), moveList.end(), bestMove);
        if (move != moveList.end())
        {
            std::iter_swap(moveList.begin(), move);
        }
    }
}

uint32_t PrincipalVariationSearcher::getMaxDepth() const
{
    return maxReachedDepth;
}

```

Код файла TranspositionTable.h:

```

#ifndef TRANSPOSITIONTABLE_H
#define TRANSPOSITIONTABLE_H

#include <unordered_map>
#include <inttypes.h>

class TranspositionTable
{
public:
    class Entry
    {
    public:
        enum EvalType: uint16_t
        {
            Exact,
            UpperBound,
            LowerBound
        };

        uint64_t fullHash;
        int32_t evaluation;
        uint16_t plyFromRoot;
        EvalType evalType;
    };

    void clear();
    void put(const Entry & entry);
    Entry * find(uint64_t hash);

private:
    std::unordered_map<uint16_t, Entry> tt;
};

#endif // TRANSPOSITIONTABLE_H

```

Код файла TranspositionTable.cpp:

```

#include "TranspositionTable.h"

void TranspositionTable::clear()
{
    tt.clear();
}

```

```

void TranspositionTable::put(const TranspositionTable::Entry & entry)
{
    uint16_t key = (uint16_t) entry.fullHash;
    tt[key] = entry;
}

TranspositionTable::Entry * TranspositionTable::find(uint64_t hash)
{
    uint16_t key = (uint16_t) hash;
    auto it = tt.find(key);

    if (it != tt.end())
        return &it->second;
    else
        return nullptr;
}

```

Код файла FenUtility.h:

```

#ifndef CHESENGINE_FENUTILITY_H
#define CHESENGINE_FENUTILITY_H

#include <string>
class Position;

class FenUtility
{
public:
    static void set(Position & pos, const std::string & fen);
    static std::string fen(const Position & position);

    // https://www.chessprogramming.org/Perft_Results
    static const std::string DEFAULT_FEN;
    static const std::string KIWIPETE_FEN;
    static const std::string TEST3_FEN;
    static const std::string TEST4_FEN;
    static const std::string TEST5_FEN;
    static const std::string TEST6_FEN;
};

#endif //CHESENGINE_FENUTILITY_H

```

Код файла FenUtility.cpp:

```

#include <sstream>
#include "../BoardRepresentation/Position.h"
#include "FenUtility.h"
#include "../BoardRepresentation/Direction.h"
#include "../BoardRepresentation/BB.h"

const std::string FenUtility::DEFAULT_FEN =
    "rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1";
const std::string FenUtility::KIWIPETE_FEN =
    "r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBPPPP/R3K2R w KQkq - 0 1";
const std::string FenUtility::TEST3_FEN = "8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w
- - 0 1";

```

```

const          std::string          FenUtility::TEST4_FEN          =
"r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq - 0 1";
const          std::string          FenUtility::TEST5_FEN          =
"rnbq1k1r/pp1Pbppp/2p5/8/2B5/8/PPP1NnPP/RNBQK2R w KQ - 1 8";
const          std::string          FenUtility::TEST6_FEN          =
"r4rk1/1pp1qppp/p1np1n2/2b1p1B1/2B1P1b1/P1NP1N2/1PP1QPPP/R4RK1 w - - 0 10";

void FenUtility::set(Position &pos, const std::string &fen)
{
    pos.clear();

    size_t spaceChar = 0;
    size_t nextSpaceChar = fen.find(' ', spaceChar);
    std::string pieces = fen.substr(spaceChar, nextSpaceChar - spaceChar);

    spaceChar = nextSpaceChar;
    nextSpaceChar = fen.find(' ', spaceChar + 1);
    std::string sideToPlay = fen.substr(spaceChar + 1, nextSpaceChar -
spaceChar - 1);

    spaceChar = nextSpaceChar;
    nextSpaceChar = fen.find(' ', spaceChar + 1);
    std::string castleRights = fen.substr(spaceChar + 1, nextSpaceChar -
spaceChar - 1);

    spaceChar = nextSpaceChar;
    nextSpaceChar = fen.find(' ', spaceChar + 1);
    std::string enPassantSquare = fen.substr(spaceChar + 1, nextSpaceChar -
spaceChar - 1);

    spaceChar = nextSpaceChar;
    nextSpaceChar = fen.find(' ', spaceChar + 1);
    std::string halfMoveClock = fen.substr(spaceChar + 1, nextSpaceChar -
spaceChar - 1);

    spaceChar = nextSpaceChar;
    std::string fullMoves = fen.substr(spaceChar + 1);

    // set side to play
    pos.sideToPlay = (sideToPlay == "w" ? WHITE : BLACK);

    // set game ply
    pos.gamePly = (std::stoi(fullMoves) - 1) * 2 +
static_cast<int>(pos.sideToPlay);

    // set EP square
    {
        Square square = static_cast<Square>(std::find(SQUARE_STRING,
SQUARE_STRING + NUMBER_OF_SQUARES, enPassantSquare) - SQUARE_STRING);
        pos.history[pos.gamePly].enPassantSquare = square !=
NUMBER_OF_SQUARES ? square : NULL_SQUARE;
    }

    // set 50 move rule counter
    pos.history[pos.gamePly].plyFiftyMoveRule = std::stoi(halfMoveClock);

    // set pieces
    Square square = A8;
    for (char ch: pieces)
    {
        if (isdigit(ch))

```

```

        {
            square = static_cast<Square>(square + static_cast<int>(ch - '0')
* static_cast<int>(EAST));
        }
        else if (ch == '/')
        {
            square = static_cast<Square>(square + 2 *
static_cast<int>(SOUTH));
        }
        else
        {
            pos.putPiece(static_cast<Piece>(PIECE_CHAR.find(ch)), square);
            ++square;
        }
    }

    // set castle rights
    pos.history[pos.gamePly].movedPieces = BB::ALL_CASTLING_MASK;
    for (char token: castleRights)
    {
        switch (token)
        {
            case 'K':
                pos.history[pos.gamePly].movedPieces &=
~BB::WHITE_CASTLING_00_MASK;
                break;
            case 'Q':
                pos.history[pos.gamePly].movedPieces &=
~BB::WHITE_CASTLING_000_MASK;
                break;
            case 'k':
                pos.history[pos.gamePly].movedPieces &=
~BB::BLACK_CASTLING_00_MASK;
                break;
            case 'q':
                pos.history[pos.gamePly].movedPieces &=
~BB::BLACK_CASTLING_000_MASK;
                break;
        }
    }
}

std::string FenUtility::fen(const Position &position)
{
    std::string fen;

    for (int i = 56; i >= 0; i -= 8)
    {
        uint32_t empty = 0;
        for (int j = 0; j < 8; ++j)
        {
            Piece piece = position.mailbox[i + j];
            if (piece == NULL_PIECE)
            {
                ++empty;
            }
            else
            {
                fen += (empty == 0 ? "" : std::to_string(empty)) +
PIECE_CHAR[piece];
                empty = 0;
            }
        }
    }
}

```

```

    }
}

if (empty != 0)
{
    fen += std::to_string(empty);
}
if (i > 0)
{
    fen += '/';
}
}

fen += (position.sideToPlay == WHITE ? " w " : " b ");
int castlings = 0;
if (!position.history[position.gamePly].movedPieces &
BB::WHITE_CASTLING_00_MASK)) { fen += "K"; ++castlings; }
if (!position.history[position.gamePly].movedPieces &
BB::WHITE_CASTLING_000_MASK)) { fen += "Q"; ++castlings; }
if (!position.history[position.gamePly].movedPieces &
BB::BLACK_CASTLING_00_MASK)) { fen += "k"; ++castlings; }
if (!position.history[position.gamePly].movedPieces &
BB::BLACK_CASTLING_000_MASK)) { fen += "q"; ++castlings; }
if (castlings == 0) fen += "-";

fen += (position.history[position.gamePly].enPassantSquare == NULL_SQUARE
? " _" : " " +
SQUARE_STRING[position.history[position.gamePly].enPassantSquare]);
fen += " " +
std::to_string(position.history[position.gamePly].plyFiftyMoveRule);
fen += " " + std::to_string(position.gamePly / 2 + 1);

return fen;
}

```

Код файла Zobrist.h:

```

#ifndef CHESENGINE_ZOBRIST_H
#define CHESENGINE_ZOBRIST_H

#include <stdint>
#include "../BoardRepresentation/Piece.h"
#include "../BoardRepresentation/Square.h"

class Zobrist
{
public:
    static void initializeTable();
    // used to hash pieces
    static uint64_t pieceTable[NUMBER_OF_PIECES][NUMBER_OF_SQUARES];

    // used to hash EP square
    static uint64_t fileTable[8];

    // to hash castling rights
    static uint64_t castlingRightsTable[4];

    static uint64_t sideToMove;
};

```

```
#endif //CHESENGINE_ZOBRIST_H
```

Код файлу Zobrist.cpp:

```
#include "Zobrist.h"
#include <random>

uint64_t Zobrist::pieceTable[NUMBER_OF_PIECES][NUMBER_OF_SQUARES];
// used to hash EP square
uint64_t Zobrist::fileTable[8];

// to hash castling rights
uint64_t Zobrist::castlingRightsTable[4];

uint64_t Zobrist::sideToMove;

void Zobrist::initializeTable()
{
    std::mt19937 prng(0);
    std::uniform_int_distribution<uint64_t> distribution;
    for (int i = 0; i < static_cast<int>(NUMBER_OF_PIECES); ++i)
    {
        for (int j = 0; j < static_cast<int>(NUMBER_OF_SQUARES); ++j)
        {
            pieceTable[i][j] = distribution(prng);
        }
    }

    for (int i = 0; i < 8; ++i)
    {
        fileTable[i] = distribution(prng);
    }

    for (int i = 0; i < 4; ++i)
    {
        castlingRightsTable[i] = distribution(prng);
    }

    sideToMove = distribution(prng);
}
```

Б.2 Код проєкту шахового рушія

Код проєктного файлу CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.19)
project(ChessEngine LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(ChessEngine
    main.cpp
```

```

        ChessEngine.cpp
        ChessEngine.h

#           Player.h
#           NNEvaluationBot.h
#           ChessGame.h
#           StaticEvaluationBot.h

#           NNEvaluationBot.cpp
#           ChessGame.cpp
#           StaticEvaluationBot.cpp

    )

target_link_libraries(ChessEngine PRIVATE ChessEngineCore)
target_include_directories(ChessEngine PRIVATE
    ../ChessEngineCore
    "${PROJECT_BINARY_DIR}")

configure_file(EngineConfig.h.in EngineConfig.h)

```

Код файла main.cpp:

```

#include "ChessEngine.h"

int main(int argc, char** argv)
{
    std::string modelPath =
"/home/mikhail/university/SlowChessEngine/nn/evaluate_position_simplified";

    if (argc > 1)
    {
        modelPath = std::string(argv[1]);
        std::cout << "Setting model path to " << modelPath;
    }

    ChessEngine engine(modelPath);
    return engine.exec();
}

```

Код файла ChessEngine.h:

```

#ifndef CHESSENGINE_H
#define CHESSENGINE_H

#include <iostream>
#include <BoardRepresentation/Position.h>
#include <memory>
#include <Evaluation/NeuralNetworkEvaluator.h>

class ChessEngine
{
public:
    explicit ChessEngine(const std::string & pathToModel);

public:
    int exec();
    void executeCommand(const std::string & command);

```

```

public:
    std::string waitForUserInput(std::istream & is);
    Position chessPosition;

private:
    std::unique_ptr<NeuralNetworkEvaluator> nnEvaluator;
};

#endif // CHESEENGINE_H

```

Код файла ChessEngine.cpp:

```

#include "ChessEngine.h"

#include <Utility/Zobrist.h>
#include <BoardRepresentation/BB.h>
#include <Evaluation/NeuralNetworkEvaluator.h>
#include <Search/DefaultSearcher.h>
#include <Search/PerfomanceTester.h>
#include <Search/PrincipalVariationSearcher.h>
#include <chrono>

ChessEngine::ChessEngine(const std::string & pathToModel)
{
    Zobrist::initializeTable();
    BB::initializeAll();
    chessPosition.set(FenUtility::DEFAULT_FEN);
    nnEvaluator = std::make_unique<NeuralNetworkEvaluator>(pathToModel);
    std::cout << "\n\nEngine started!\n\n" << std::endl;
}

int ChessEngine::exec()
{
    std::string command = "";
    while (command != "exit")
    {
        command = waitForUserInput(std::cin);
        if (command == "exit")
        {
            std::cout << "Shutting down" << std::endl;
        }
        else
        {
            auto start = std::chrono::system_clock::now();
            executeCommand(command);
            auto finish = std::chrono::system_clock::now();
            double timeTaken =
std::chrono::duration_cast<std::chrono::microseconds>(finish - start).count()
/ 1000000.0;
            std::cout << "Time taken: " << timeTaken << "\n\n" << std::endl;
        }
    }
    return 0;
}

void ChessEngine::executeCommand(const std::string & command)
{
    if (command.starts_with("position"))
    {
        const std::string fen = command.substr(command.find(' ') + 1);
    }
}

```

```

        chessPosition.set(fen);
    }
    else if (command == "d")
    {
        std::cout << chessPosition << std::endl;
    }
    else if (command == "generate")
    {
        MoveList moveList(chessPosition);
        for (Move move: moveList)
            std::cout << move.toString() << std::endl;
    }
    else if (command.starts_with("perftdiv"))
    {
        uint32_t depth = std::stoi(command.substr(command.find(' ')));
        uint64_t totalNodes = PerfomanceTester::perftdiv(chessPosition,
depth);
        std::cout << "Total nodes: " << totalNodes << std::endl;
    }
    else if (command.starts_with("perft"))
    {
        uint32_t depth = std::stoi(command.substr(command.find(' ')));
        uint64_t totalNodes = PerfomanceTester::perft(chessPosition, depth);
        std::cout << "Total nodes: " << totalNodes << std::endl;
    }
    else if (command.starts_with("findnn"))
    {
        uint32_t depth = std::stoi(command.substr(command.find(' ')));
        PrincipalVariationSearcher searcher(nnEvaluator.get());
        const auto [move, eval] =
searcher.iterativeDeepeningSearch(chessPosition, depth);
        std::cout << "Best move: " << move.toString() << std::endl;
        std::cout << "Evaluation: " << eval << std::endl;
        std::cout << "Max depth with quiet search: " <<
searcher.getMaxDepth() << std::endl;
    }
    else if (command.starts_with("find"))
    {
        uint32_t depth = std::stoi(command.substr(command.find(' ')));
        PrincipalVariationSearcher searcher(nullptr);
        const auto [move, eval] =
searcher.iterativeDeepeningSearch(chessPosition, depth);
        std::cout << "Best move: " << move.toString() << std::endl;
        std::cout << "Evaluation: " << eval << std::endl;
        std::cout << "Max depth with quiet search: " <<
searcher.getMaxDepth() << std::endl;
    }
    else
    {
        std::cout << "Unknown command" << std::endl;
    }
}

std::string ChessEngine::waitForUserInput(std::istream & is)
{
    std::string command;
    std::getline(is, command);
    return command;
}

```

Код файла ChessGame.h:

```
#ifndef CHESSGAME_H
#define CHESSGAME_H

#include "Player.h"

#include <BoardRepresentation/Position.h>

class ChessGame
{
public:
    ChessGame(Player * white, Player * black);

public:
    void play(const std::string & fen = FenUtility::DEFAULT_FEN);
    int result() const;

private:
    int m_result;
    Player * whitePlayer;
    Player * blackPlayer;
};

#endif // CHESSGAME_H
```

Код файла ChessGame.cpp:

```
#include "ChessGame.h"
#include <iostream>
#include <Utility/Zobrist.h>
#include <BoardRepresentation/BB.h>

ChessGame::ChessGame(Player * white, Player * black)
    : whitePlayer(white)
    , blackPlayer(black)
    , m_result(0)
{
}

void ChessGame::play(const std::string & fen)
{
    Position pos(fen);

    while (pos.status() == Position::NOT_FINISHED)
    {
        //      std::cout << pos << std::endl;
        if (pos.sideToPlay == WHITE)
        {
            whitePlayer->makeMove(pos);
        }
        else
        {
            blackPlayer->makeMove(pos);
        }
    }

    //      std::cout << pos << std::endl;
```

```

        switch (pos.status())
        {
        case Position::DRAW:
            m_result = 0;
//            std::cout << "Game finished with draw" << std::endl;
            break;

        case Position::STALEMATE:
            m_result = 0;
//            std::cout << "Game finished with stalemate" << std::endl;
            break;

        case Position::WHITE_WIN:
            m_result = 1;
//            std::cout << "White won the game" << std::endl;
            break;

        case Position::BLACK_WIN:
            m_result = -1;
//            std::cout << "Black won the game" << std::endl;
            break;

        case Position::NOT_FINISHED:
            throw std::runtime_error("AAAA");
//            std::cout << "Game wasn't finished" << std::endl;
            break;
        }
    }

    int ChessGame::result() const
    {
        return m_result;
    }

```

Код файлу HumanPlayer.h:

```

#ifndef HUMANPLAYER_H
#define HUMANPLAYER_H

#include "Player.h"

class HumanPlayer : public Player
{
public:
    HumanPlayer() = default;
    ~HumanPlayer() override = default;

public:
    Move makeMove(Position & position) override;
};

#endif // HUMANPLAYER_H

```

Код файлу HumanPlayer.cpp:

```

#include "HumanPlayer.h"
#include <iostream>
#include <BoardRepresentation/Position.h>

```

```

Move HumanPlayer::makeMove(Position & position)
{
    if (position.status() != Position::NOT_FINISHED)
    {
        throw std::runtime_error("Cannot make the move - the game is
finished");
    }

    std::string moveString;
    std::cout << "Enter your move (pure coord notation): ";
    std::cin >> moveString;

    Move move = position.moveFromString(moveString);
    position.makeMove(move);

    std::cout << "[HUMAN]: Played a move: " << move.toString() << std::endl;

    return move;
}

```

Код файла NNEvaluationBot.h:

```

#ifndef NNEVALUATIONBOT_H
#define NNEVALUATIONBOT_H

#include <memory>
#include "Player.h"
#include <Evaluation/NeuralNetworkEvaluator.h>

class NNEvaluationBot : public Player
{
public:
    NNEvaluationBot(NeuralNetworkEvaluator * nnEvaluator, uint32_t
maxSearchDepth = 3);
    ~NNEvaluationBot() override = default;

public:
    Move makeMove(Position & position) override;

private:
    NeuralNetworkEvaluator * neuralEvaluator;
    uint32_t maxSearchDepth;
};

#endif // NNEVALUATIONBOT_H

```

Код файла NNEvaluationBot.cpp:

```

#include "NNEvaluationBot.h"

#include <iostream>
#include <BoardRepresentation/Position.h>
#include <Search/DefaultSearcher.h>
#include <Search/PrincipalVariationSearcher.h>

NNEvaluationBot::NNEvaluationBot(NeuralNetworkEvaluator * nnEvaluator,
uint32_t maxSearchDepth)

```

```

        : neuralEvaluator(nnEvaluator)
        , maxSearchDepth(maxSearchDepth)
    {
    }

Move NNEvaluationBot::makeMove(Position & position)
{
    if (position.status() != Position::NOT_FINISHED)
    {
        throw std::runtime_error("Cannot make the move - the game is
finished");
    }

    PrincipalVariationSearcher moveSearcher(neuralEvaluator);
    const auto & [move, evaluation] =
moveSearcher.iterativeDeepeningSearch(position, maxSearchDepth);
    // std::cout << "[NNEvaluationBot]: Best move found: " << move.toString()
<< " with evaluation: " << evaluation << std::endl;
    position.makeMove(move);
    return move;
}

```

Код файла Player.h:

```

#ifndef PLAYER_H
#define PLAYER_H

class Move;
class Position;

class Player
{
public:
    virtual ~Player() = default;

public:
    virtual Move makeMove(Position & position) = 0;
};

#endif // PLAYER_H

```

Код файла StaticEvaluationBot.h:

```

#ifndef STATICEVALUATIONBOT_H
#define STATICEVALUATIONBOT_H

#include "Player.h"
#include <cstdint>

class StaticEvaluationBot: public Player
{
public:
    StaticEvaluationBot(uint32_t maxDepth);
    ~StaticEvaluationBot() override = default;

public:
    Move makeMove(Position & position) override;
}

```

```
private:
    uint32_t maxSearchDepth;
};

#endif // STATICEVALUATIONBOT_H
```

Код файла StaticEvaluationBot.cpp:

```
#include "StaticEvaluationBot.h"

#include <BoardRepresentation/Position.h>
#include <iostream>
#include <Search/DefaultSearcher.h>
#include <Search/PrincipalVariationSearcher.h>

StaticEvaluationBot::StaticEvaluationBot(uint32_t maxDepth)
    : maxSearchDepth(maxDepth)
{
}

Move StaticEvaluationBot::makeMove(Position & position)
{
    if (position.status() != Position::NOT_FINISHED)
    {
        throw std::runtime_error("Cannot make the move - the game is finished");
    }

    PrincipalVariationSearcher moveSearcher;
    const auto & [move, evaluation] =
moveSearcher.iterativeDeepeningSearch(position, maxSearchDepth);
    // std::cout << "[StaticEvaluationBot]: Best move found: " <<
move.toString() << " with evaluation: " << evaluation << std::endl;
    position.makeMove(move);
    return move;
}
```

Б.3 Код тестуального модуля (проект)

Код проектного файла CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.19)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
project(ChessEngineTest LANGUAGES CXX)

add_definitions(-DGTEST_LANGUAGE_CXX11)

find_package(Threads REQUIRED)

set(GOOGLETEST_DIR /usr/src/googletest)
set(GTestSrc ${GOOGLETEST_DIR}/googletest)
set(GMockSrc ${GOOGLETEST_DIR}/googlemock)
```

```

set(GTestFiles ${GTestSrc}/src/gtest-all.cc)
set(GTestIncludes ${GTestSrc} ${GTestSrc}/include)
if (NOT ${GMockSrc} STREQUAL "")
    list(APPEND GTestFiles ${GMockSrc}/src/gmock-all.cc)
    list(APPEND GTestIncludes ${GMockSrc} ${GMockSrc}/include)
endif ()

include_directories(${GTestIncludes})

add_executable(ChessEngineTest
    main.cpp
    Perft/tst_PerftStartingPosition.h
    Perft/tst_PerftKiwipete.h
    Perft/tst_PerftPosition3.h
    Perft/tst_PerftPosition4.h
    Perft/tst_PerftPosition5.h
    Perft/tst_PerftPosition6.h
    ${GTestFiles})
add_test(NAME perft COMMAND perft)

target_link_libraries(ChessEngineTest PRIVATE Threads::Threads)
target_link_libraries(ChessEngineTest PRIVATE ChessEngineCore)
target_include_directories(ChessEngineTest PRIVATE ../ChessEngineCore)

```

Код файла main.cpp:

```

#include "Perft/tst_PerftStartingPosition.h"
#include "Perft/tst_PerftKiwipete.h"
#include "Perft/tst_PerftPosition3.h"
#include "Perft/tst_PerftPosition4.h"
#include "Perft/tst_PerftPosition5.h"
#include "Perft/tst_PerftPosition6.h"
#include <gtest/gtest.h>

int main(int argc, char *argv[])
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```

Код файла tst_PerftStartingPosition.h:

```

#ifndef TST_PERFTSTARTINGPOSITION_H
#define TST_PERFTSTARTINGPOSITION_H

#include <gtest/gtest.h>

#include <Utility/Zobrist.h>
#include <BoardRepresentation/BB.h>
#include <BoardRepresentation/Position.h>
#include <Search/PerformanceTester.h>

class PerftStartingPosition:
    public ::testing::TestWithParam<std::pair<uint32_t, uint64_t>>
{
public:

```

```

        static void SetUpTestSuite()
        {
            Zobrist::initializeTable();
            BB::initializeAll();
        }
};

INSTANTIATE_TEST_SUITE_P(StartingPosition,
                          PerftStartingPosition,
                          ::testing::Values(
                              std::make_pair<uint32_t, uint64_t>(0, 1ull)
                              , std::make_pair<uint32_t, uint64_t>(1, 20ull)
                              , std::make_pair<uint32_t, uint64_t>(2, 400ull)
                              , std::make_pair<uint32_t, uint64_t>(3, 8902ull)
                              , std::make_pair<uint32_t, uint64_t>(4,
197281ull)
                              , std::make_pair<uint32_t, uint64_t>(5,
4865609ull)
                              , std::make_pair<uint32_t, uint64_t>(6,
119060324ull)
                              // , std::make_pair<uint32_t, uint64_t>(7,
3195901860ull)
                              ));

TEST_P(PerftStartingPosition, AtDepth)
{
    Position position(FenUtility::DEFAULT_FEN);

    const auto & [perftDepth, expectedResult] = GetParam();

    ASSERT_EQ(PerfomanceTester::perft(position, perftDepth), expectedResult);
}

#endif // TST_PERFTSTARTINGPOSITION_H

```

Код файлу tst_PerftKiwipete.h:

```

#ifndef TST_PERFTKIWIPE_H
#define TST_PERFTKIWIPE_H

#include <gtest/gtest.h>

#include <Utility/Zobrist.h>
#include <BoardRepresentation/BB.h>
#include <BoardRepresentation/Position.h>
#include <Search/PerfomanceTester.h>

class PerftKiwipete:
    public ::testing::TestWithParam<std::pair<uint32_t, uint64_t>>
{
public:
    static void SetUpTestSuite()
    {
        Zobrist::initializeTable();
        BB::initializeAll();
    }
};

INSTANTIATE_TEST_SUITE_P(Kiwipete,
                          PerftKiwipete,

```

```

        ::testing::Values(
            std::make_pair<uint32_t, uint64_t>(0, 1ull)
            , std::make_pair<uint32_t, uint64_t>(1, 48ull)
            , std::make_pair<uint32_t, uint64_t>(2, 2039ull)
            , std::make_pair<uint32_t, uint64_t>(3, 97862ull)
            , std::make_pair<uint32_t, uint64_t>(4,
4085603ull)
            , std::make_pair<uint32_t, uint64_t>(5,
193690690ull)
            // , std::make_pair<uint32_t, uint64_t>(6,
8031647685ull)
        ));

TEST_P(PerftKiwipete, AtDepth)
{
    Position position(FenUtility::KIWIPETE_FEN);

    const auto & [perftDepth, expectedResult] = GetParam();

    ASSERT_EQ(PerfomanceTester::perft(position, perftDepth), expectedResult);
}

#endif // TST_PERFTKIWIPETE_H

```

Код файла tst_PerftPosition3.h:

```

#ifndef TST_PERFTPOSITION3_H
#define TST_PERFTPOSITION3_H

#include <gtest/gtest.h>

#include <Utility/Zobrist.h>
#include <BoardRepresentation/BB.h>
#include <BoardRepresentation/Position.h>
#include <Search/PerfomanceTester.h>

class PerftPosition3:
    public ::testing::TestWithParam<std::pair<uint32_t, uint64_t>>
{
public:
    static void SetUpTestSuite()
    {
        Zobrist::initializeTable();
        BB::initializeAll();
    }
};

INSTANTIATE_TEST_SUITE_P(Position3,
    PerftPosition3,
    ::testing::Values(
        std::make_pair<uint32_t, uint64_t>(0, 1)
        , std::make_pair<uint32_t, uint64_t>(1, 14)
        , std::make_pair<uint32_t, uint64_t>(2, 191)
        , std::make_pair<uint32_t, uint64_t>(3, 2812)
        , std::make_pair<uint32_t, uint64_t>(4, 43238)
        , std::make_pair<uint32_t, uint64_t>(5, 674624)
        , std::make_pair<uint32_t, uint64_t>(6, 11030083)
        , std::make_pair<uint32_t, uint64_t>(7, 178633661)
        // , std::make_pair<uint32_t, uint64_t>(8,
3009794393)

```

```

    ));

TEST_P(PerftPosition3, AtDepth)
{
    Position position(FenUtility::TEST3_FEN);

    const auto & [perftDepth, expectedResult] = GetParam();

    ASSERT_EQ(PerfomanceTester::perft(position, perftDepth), expectedResult);
}

#endif // TST_PERFTPOSITION3_H

```

Код файла tst_PerftPosition4.h:

```

#ifndef TST_PERFTPOSITION4_H
#define TST_PERFTPOSITION4_H

#include <gtest/gtest.h>

#include <Utility/Zobrist.h>
#include <BoardRepresentation/BB.h>
#include <BoardRepresentation/Position.h>
#include <Search/PerfomanceTester.h>

class PerftPosition4:
    public ::testing::TestWithParam<std::pair<uint32_t, uint64_t>>
{
public:
    static void SetUpTestSuite()
    {
        Zobrist::initializeTable();
        BB::initializeAll();
    }
};

INSTANTIATE_TEST_SUITE_P(Position4,
    PerftPosition4,
    ::testing::Values(
        std::make_pair<uint32_t, uint64_t>(0, 1)
        , std::make_pair<uint32_t, uint64_t>(1, 6)
        , std::make_pair<uint32_t, uint64_t>(2, 264)
        , std::make_pair<uint32_t, uint64_t>(3, 9467)
        , std::make_pair<uint32_t, uint64_t>(4, 422333)
        , std::make_pair<uint32_t, uint64_t>(5, 15833292)
        , std::make_pair<uint32_t, uint64_t>(6,
//
706045033)
    ));

TEST_P(PerftPosition4, AtDepth)
{
    Position position(FenUtility::TEST4_FEN);

    const auto & [perftDepth, expectedResult] = GetParam();

    ASSERT_EQ(PerfomanceTester::perft(position, perftDepth), expectedResult);
}

#endif // TST_PERFTPOSITION4_H

```

Код файла tst_PerftPosition5.h:

```
#ifndef TST_PERFTPOSITION5_H
#define TST_PERFTPOSITION5_H

#include <gtest/gtest.h>

#include <Utility/Zobrist.h>
#include <BoardRepresentation/BB.h>
#include <BoardRepresentation/Position.h>
#include <Search/PerfomanceTester.h>

class PerftPosition5:
    public ::testing::TestWithParam<std::pair<uint32_t, uint64_t>>
{
public:
    static void SetUpTestSuite()
    {
        Zobrist::initializeTable();
        BB::initializeAll();
    }
};

INSTANTIATE_TEST_SUITE_P(Position5,
    PerftPosition5,
    ::testing::Values(
        std::make_pair<uint32_t, uint64_t>(0, 1)
        , std::make_pair<uint32_t, uint64_t>(1, 44)
        , std::make_pair<uint32_t, uint64_t>(2, 1486)
        , std::make_pair<uint32_t, uint64_t>(3, 62379)
        , std::make_pair<uint32_t, uint64_t>(4, 2103487)
        , std::make_pair<uint32_t, uint64_t>(5, 89941194)
    ));

TEST_P(PerftPosition5, AtDepth)
{
    Position position(FenUtility::TEST5_FEN);

    const auto & [perftDepth, expectedResult] = GetParam();

    ASSERT_EQ(PerfomanceTester::perft(position, perftDepth), expectedResult);
}

#endif // TST_PERFTPOSITION5_H
```

Код файла tst_PerftPosition6.h:

```
#ifndef TST_PERFTPOSITION6_H
#define TST_PERFTPOSITION6_H

#include <gtest/gtest.h>

#include <Utility/Zobrist.h>
#include <BoardRepresentation/BB.h>
#include <BoardRepresentation/Position.h>
#include <Search/PerfomanceTester.h>
```

```

class PerftPosition6:
    public ::testing::TestWithParam<std::pair<uint32_t, uint64_t>>
{
public:
    static void SetUpTestSuite()
    {
        Zobrist::initializeTable();
        BB::initializeAll();
    }
};

INSTANTIATE_TEST_SUITE_P(Position6,
    PerftPosition6,
    ::testing::Values(
        std::make_pair<uint32_t, uint64_t>(0, 1)
        , std::make_pair<uint32_t, uint64_t>(1, 46)
        , std::make_pair<uint32_t, uint64_t>(2, 2079)
        , std::make_pair<uint32_t, uint64_t>(3, 89890)
        , std::make_pair<uint32_t, uint64_t>(4, 3894594)
        , std::make_pair<uint32_t, uint64_t>(5, 164075551)
        , std::make_pair<uint32_t, uint64_t>(6,
//
6923051137)
    ));

TEST_P(PerftPosition6, AtDepth)
{
    Position position(FenUtility::TEST6_FEN);

    const auto & [perftDepth, expectedResult] = GetParam();

    ASSERT_EQ(PerfomanceTester::perft(position, perftDepth), expectedResult);
}

#endif // TST_PERFTPOSITION6_H

```

Б.4 Код батьківського проєкту

Текст файлу CMakeLists.txt:

```

cmake_minimum_required(VERSION 3.19)

project(SimpleChessEngine LANGUAGES CXX)

add_subdirectory(ChessEngine)
add_subdirectory(ChessEngineTest)
add_subdirectory(ChessEngineCore)

```

Б.5 Код для підготовки датасету та навчання нейронної мережі

Код файлу prepare_dataset.ipynb:

```

{

```



```

        "    if (piece.color == chess.BLACK):\n",
        "        index += 6\n",
        "    return index"
    ]
},
{
    "cell_type": "code",
    "execution_count": null,
    "id": "b34a06c5",
    "metadata": {},
    "outputs": [],
    "source": [
        "def en_pasant_square_index(square):\n",
        "    index = chess.square_file(square)\n",
        "    if (chess.square_rank(square) != 2):\n",
        "        index += 8\n",
        "    return index"
    ]
},
{
    "cell_type": "code",
    "execution_count": null,
    "id": "1b860b28",
    "metadata": {},
    "outputs": [],
    "source": [
        "def row_from_fen(fen):\n",
        "    board = chess.Board(fen)\n",
        "    row = [0 for i in range(total_number_of_features)]\n",
        "    \n",
        "    if (board.turn == chess.BLACK):\n",
        "        board = board.mirror()\n",
        "    \n",
        "    for square in chess.SQUARES:\n",
        "        if board.piece_at(square) != None:\n",
        "            feature_index = int(square) * features_per_square \\\n",
        "chess_piece_index(board.piece_at(square))\n",
        "            row[feature_index] = 1\n",
        "    \n",
        "    return np.asarray(row, dtype=bool)"
    ]
},
{
    "cell_type": "code",
    "execution_count": null,
    "id": "ac8a9004",
    "metadata": {},
    "outputs": [],
    "source": [
        "def transform_evaluation(eval, board):\n",
        "    checkmate_score = 1000\n",
        "    loss_per_move_to_checkmate = 1\n",
        "    evaluation = 0\n",
        "    \n",
        "    if eval[0] == '\\\\u00ff':\n",
        "        eval = eval[1:]\n",
        "    \n",
        "    if eval[0] == '#':\n",
        "        moves_to_checkmate = int(eval[2:])\n",
        "        sign = 1 if eval[1] == '+' else -1\n",

```

```

        "        evaluation = sign * (checkmate_score - (abs(moves_to_checkmate)
- 1) * loss_per_move_to_checkmate)\n",
        "        else:\n",
        "            evaluation = int(eval)\n",
        "            \n",
        "            # make relative instead of absolute\n",
        "            if (board.turn == chess.BLACK):\n",
        "                evaluation = -evaluation\n",
        "            \n",
        "            # normalize \n",
        "            evaluation = np.clip(evaluation, -checkmate_score,
checkmate_score)\n",
        "            return evaluation / checkmate_score"
    ]
},
{
    "cell_type": "code",
    "execution_count": null,
    "id": "027839bb",
    "metadata": {},
    "outputs": [],
    "source": [
        "def bytes_list_feature(value):\n",
        "
return
tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))"
    ],
},
{
    "cell_type": "code",
    "execution_count": null,
    "id": "299cf3d6",
    "metadata": {},
    "outputs": [],
    "source": [
        "def float32_feature(value):\n",
        "
return
tf.train.Feature(float_list=tf.train.FloatList(value=[value]))"
    ],
},
{
    "cell_type": "code",
    "execution_count": null,
    "id": "096f1061",
    "metadata": {},
    "outputs": [],
    "source": [
        "def position_example(position, y):\n",
        "    feature = {'feature': bytes_list_feature(position)}\n",
        "    feature['y'] = float32_feature(y)\n",
        "
return
tf.train.Example(features=tf.train.Features(feature=feature))"
    ],
},
{
    "cell_type": "code",
    "execution_count": null,
    "id": "16f2bd67",
    "metadata": {},
    "outputs": [],
    "source": [
        "def parse_dataset(record):\n",

```

```

        features_names = {'f'+str(i): tf.io.FixedLenFeature([], tf.byte) for
i in range(total_number_of_features)}\n",
        output_name = {'y' : tf.io.FixedLenFeature([], tf.float32)}\n",
        features = tf.io.parse_example([record],
features=features_names)\n",
        output = tf.io.parse_example([record], features=output_name)\n",
        return features, output"
    ]
},
{
    "cell_type": "code",
    "execution_count": null,
    "id": "b737de44",
    "metadata": {},
    "outputs": [],
    "source": [
        "filename = \"~/university/SlowChessEngine/nn/chessData.csv\"\n",
        "df = read_dataset(filename)"
    ]
},
{
    "cell_type": "code",
    "execution_count": null,
    "id": "e19048ea",
    "metadata": {},
    "outputs": [],
    "source": [
        "batch_size = 10000\n",
        "batch_index = 1\n",
        "\n",
        "for index in df.index:\n",
        "    \n",
        "    if index % batch_size == 0:\n",
        "        if batch_index != 1:\n",
        "            writer.close()\n",
        "            break\n",
        "            \n",
        "            writer =
tf.io.TFRecordWriter(f'/home/mikhail/university/SlowChessEngine/nn/chess_data
set{batch_index}.tfrecord')\n",
        "        batch_index += 1\n",
        "        \n",
        "        evaluation = transform_evaluation(df["Evaluation"][index],
chess.Board(df["FEN"][index]))\n",
        "        features = row_from_fen(df["FEN"][index]).tobytes()\n",
        "        #iterate through csv\n",
        "        ex = position_example(features, evaluation)\n",
        "        writer.write(ex.SerializeToString())\n",
        "\n",
        "writer.close()\n"
    ]
},
{
    "cell_type": "code",
    "execution_count": null,
    "id": "283a59a5",
    "metadata": {},
    "outputs": [],
    "source": []
}
],
"metadata": {

```

```

"kernel_spec": {
  "display_name": "Python 3",
  "language": "python",
  "name": "python3"
},
"language_info": {
  "codemirror_mode": {
    "name": "ipython",
    "version": 3
  },
  "file_extension": ".py",
  "mimetype": "text/x-python",
  "name": "python",
  "nbconvert_exporter": "python",
  "pygments_lexer": "ipython3",
  "version": "3.8.6"
}
},
"nbformat": 4,
"nbformat_minor": 5
}

```

Код файла train_model.ipynb:

```

{
  "cells": [
    {
      "cell_type": "code",
      "execution_count": null,
      "id": "ddc19a75",
      "metadata": {},
      "outputs": [],
      "source": [
        "import numpy as np\n",
        "import tensorflow as tf\n",
        "import glob"
      ]
    },
    {
      "cell_type": "code",
      "execution_count": null,
      "id": "f8abc047",
      "metadata": {},
      "outputs": [],
      "source": [
        "def get_dataset(records_path, shuffle_size, batch_size):\n",
        "    tf_records = glob.glob(records_path)\n",
        "    dataset = tf.data.TFRecordDataset(tf_records)\n",
        "    dataset = dataset.map(parse_dataset)\n",
        "    #dataset = dataset.shuffle(shuffle_size).batch(batch_size)\n",
        "    dataset = dataset.batch(batch_size)\n",
        "    return dataset"
      ]
    },
    {
      "cell_type": "code",
      "execution_count": null,
      "id": "85e49c95",
      "metadata": {},
      "outputs": []
    }
  ]
}

```

```

"source": [
  "def parse_dataset(record):\n",
  "    name_to_features = {\n",
  "        'feature': tf.io.FixedLenFeature([], tf.string),\n",
  "        'y' : tf.io.FixedLenFeature([], tf.float32)\n",
  "    }\n",
  "    features = tf.io.parse_example([record],
features=name_to_features)\n",
  "    X = tf.io.decode_raw(features['feature'], bool)\n",
  "    return X, features['y']"
],
},
{
  "cell_type": "code",
  "execution_count": null,
  "id": "e581413b",
  "metadata": {},
  "outputs": [],
  "source": [
    "def create_model():\n",
    "    model = tf.keras.Sequential()\n",
    "    model.add(tf.keras.layers.Input((1, 768)))\n",
    "    model.add(tf.keras.layers.Dense(512, activation='relu'))\n",
    "    model.add(tf.keras.layers.Dense(128, activation='relu'))\n",
    "    model.add(tf.keras.layers.Dense(32, activation='relu'))\n",
    "    model.add(tf.keras.layers.Dense(1, activation='tanh'))\n",
    "    model.compile(optimizer = 'adam', loss = 'mse', metrics =
['mse'])\n",
    "    return model"
  ],
},
{
  "cell_type": "code",
  "execution_count": null,
  "id": "44c3159e",
  "metadata": {},
  "outputs": [],
  "source": [
    "model = create_model()"
  ],
},
{
  "cell_type": "code",
  "execution_count": null,
  "id": "1e4f37b7",
  "metadata": {},
  "outputs": [],
  "source": [
    "model.summary()"
  ],
},
{
  "cell_type": "code",
  "execution_count": null,
  "id": "779293c6",
  "metadata": {},
  "outputs": [],
  "source": [
    "dataset = get_dataset(\"./*.tfrecord\", 1000, 32)"
  ],
},
},

```

```

{
  "cell_type": "code",
  "execution_count": null,
  "id": "728e0098",
  "metadata": {},
  "outputs": [],
  "source": [
    "callback = tf.keras.callbacks.TensorBoard(\n",
    "    log_dir='logs', update_freq = 1000\n",
    ")\n",
    "model.fit(dataset, epochs = 1, callbacks = [callback])"
  ]
},
{
  "cell_type": "code",
  "execution_count": null,
  "id": "120ffbe2",
  "metadata": {},
  "outputs": [],
  "source": [
    "model.save(\"evaluate_position_simplified\")"
  ]
},
{
  "cell_type": "code",
  "execution_count": null,
  "id": "dead9e00",
  "metadata": {},
  "outputs": [],
  "source": [
    "for batch in dataset:\n",
    "    print((batch[0][0]))\n",
    "    print(batch[1])\n",
    "    break"
  ]
},
{
  "cell_type": "code",
  "execution_count": null,
  "id": "f9fe1524",
  "metadata": {},
  "outputs": [],
  "source": []
}
],
"metadata": {
  "kernelspec": {
    "display_name": "Python 3",
    "language": "python",
    "name": "python3"
  },
  "language_info": {
    "codemirror_mode": {
      "name": "ipython",
      "version": 3
    },
    "file_extension": ".py",
    "mimetype": "text/x-python",
    "name": "python",
    "nbconvert_exporter": "python",
    "pygments_lexer": "ipython3",

```

```
    "version": "3.8.6"  
  }  
},  
"nbformat": 4,  
"nbformat_minor": 5  
}
```

ДОДАТОК В

Слайди презентації

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ

Випускна робота бакалавра
за темою «Програмна реалізація шахового рушія з
використанням технології глибокого навчання
нейромережових моделей»

Підготував: Басанець Михайло Іванович, ст. гр. КНТ-117
Керівник: Олійник Андрій Олександрович, к.т.н, доцент НУЗП

1

Рисунок В.1 – Слайд №1

Об'єкт та предмет дослідження Мета роботи

Об'єкт дослідження – процес реалізації шахових рушіїв.

Предмет дослідження – методи реалізації шахових рушіїв та застосування штучних нейронних мереж в шахових рушіях.

Мета роботи – розробка шахового рушія з двома методами оцінки позиції: класичного та з використанням нейронної мережі.

2

Рисунок В.2 – Слайд №2

Огляд існуючих методів

Основні компоненти рушія:

- представлення дошки;
- генерація ходів;
- оцінка позиції;
- пошук (найкращого ходу або найкращих варіацій).

3

Рисунок В.3 – Слайд №3

Методи представлення дошки та генерації ходів

Критерій \ Метод	Поштова скриня (Mailbox)	Бітова дошка (Bitboard)
Складність реалізації представлення	Низька	Помірна
Складність реалізації генерації	Помірна	Висока
Швидкість роботи	Низька	Висока
Підтримка концепції шаху	Ні	Так
Генерація різних типів ходів	Ні (неповна)	Так
Операції над множинами	Ні	Так

4

Рисунок В.4 – Слайд №4

Генерація ходів з бітовими дошками

<pre> 1 . 1 . . 1 . 1 . 1 . . . 1 1 1 . . 1 1 1 . 1 1 1 . . . 1 1 1 1 . 1 1 . . . </pre>	<pre> 1 . . 1 1 . . 1 1 . 1 1 1 1 1 . . 1 1 1 . </pre>	=	<pre> 1 . 1 . . 1 . 1 . 1 . . . 1 1 1 . . 1 1 1 . 1 1 1 . . . 1 1 1 1 . 1 1 . . . </pre>
атаки ферзя	& ворожі фігури	=	взяття ферзем

<pre> 1 . 1 . . 1 . 1 . 1 . . . 1 1 1 . . 1 1 1 . 1 1 1 . . . 1 1 1 1 . 1 1 . . . </pre>	<pre> . 1 1 . . 1 1 . . 1 1 1 . 1 1 1 1 1 . 1 1 1 1 1 1 1 1 1 1 1 . 1 1 . 1 1 1 1 1 1 1 1 1 1 . . 1 1 1 . 1 </pre>	=	<pre> 1 . 1 . . 1 . 1 . 1 . . . 1 1 1 . . 1 1 1 . 1 1 1 . . . 1 1 1 1 . 1 1 . . . </pre>
атаки ферзя	& порожні поля	=	тихі ходи ферзем

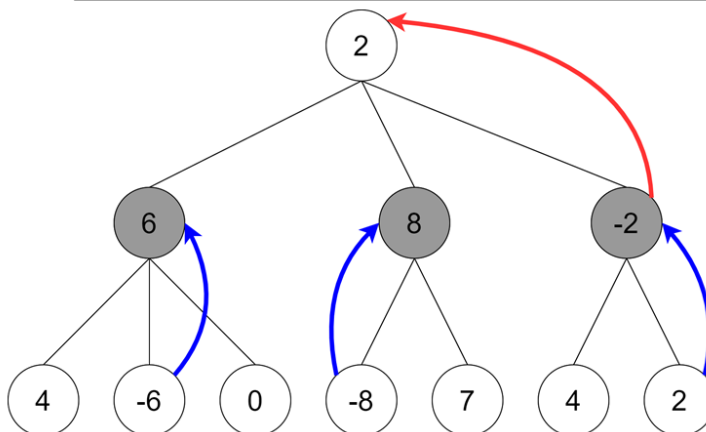
Додаткові можливості:

- завчасний розрахунок основних бітових масок (вертикалі, горизонталі);
- завчасний розрахунок атак певних фігур з певних клітинок;
- використання завчасно підрахованих бітових дошок для пришвидшення операцій.

5

Рисунок В.5 – Слайд №5

Пошук варіацій та його модифікації



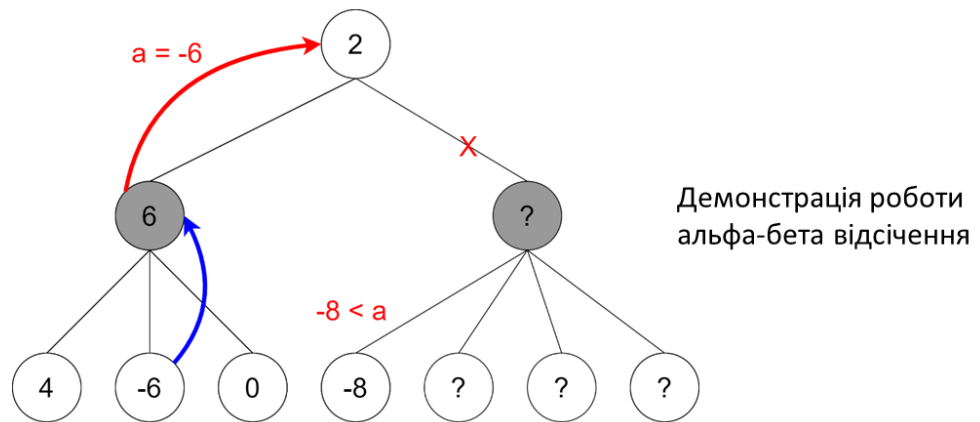
Модифікації:

- альфа-бета відсічення;
- транспозиційні таблиці;
- «тихий» пошук;
- ітеративне заглиблення;
- первинна варіація;
- упорядкування ходів.

6

Рисунок В.6 – Слайд №6

Пошук варіацій та його модифікації



7

Рисунок В.7 – Слайд №7

Методи оцінки позиції

Критерій \ Метод	Класичний	З використанням нейронної мережі
Складність реалізації	Помірна	Висока
Складність підтримки і розширення	Висока	Низька
Швидкість роботи	Висока	Низька
Залежність від набору даних	Ні	Так
Точність роботи	Висока	Висока (при успішному тренуванні)
Здатність до узагальнення до різних етапів гри	Помірна	Помірна

8

Рисунок В.8 – Слайд №8

Вибір технологій для реалізації рушія

Критерій \ МП	C++	C#	Java	Python
Продуктивність	Висока	Середня	Середня	Низька
Garbage Collection	Ні	Так	Так	Так
Кросплатформерність бібліотек	Ні	Так	Так	Так
Підтримка фреймворку tensorflow	Так	Неповна	Неповна	Так

Було обрано:

- C++ для реалізації шахового рушія;
- Python для створення та тренування моделі.

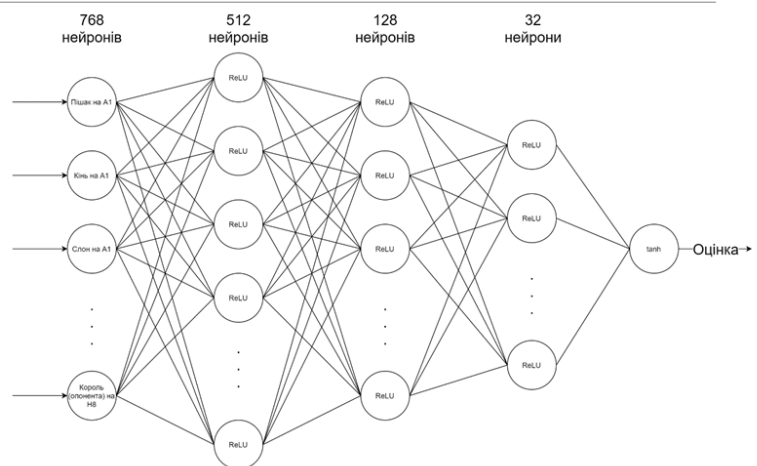
9

Рисунок В.9 – Слайд №9

Архітектура розробленої нейронної мережі

Особливості:

- неповне представлення позиції;
- очікування однакових векторів ознак для дзеркальних позицій;
- відносна оцінка позиції



10

Рисунок В.10 – Слайд №10

Результати роботи програми

Рушій здатен оброблювати наступні запити:

- встановлення заданої позиції;
- генерація ходів;
- пошук найкращого ходу на заданій глибині з класичною оцінкою позиції та з використанням нейронної мережі;
- підрахунок кількості позицій на певній глибині пошуку (2 варіанти запиту).

```
find 5
Best move: e4f5
Evaluation: 346
Max depth with quiet search: 28
Time taken: 0.195699
```

```
findnn 3
Best move: e4f5
Evaluation: 189
Max depth with quiet search: 5
Time taken: 0.400489
```

11

Рисунок В.11 – Слайд №11

Порівняння методів оцінки позиції

Глибина пошуку \ Метод	Класична оцінка позиції	Оцінка позиції нейронною мережею
1	< 1 с	< 1 с
2	< 1 с	< 1 с
3	< 1 с	1.01 с
4	< 1 с	4.19 с
5	< 1 с	42.06 с
6	< 1 с	-
7	1.26 с	-
8	10.6 с	-
9	126.9 с	-
10	-	-

12

Рисунок В.12 – Слайд №12

Переваги та недоліки розробленого продукту

Переваги:

- повністю реалізовані основні компоненти шахового рушія;
- можливість обирати метод оцінки позиції;
- можливість змінювати архітектуру нейронної мережі з певними обмеженнями без повторної компіляції коду;
- наявність як бінарного файлу, так і бібліотеки.

Недоліки:

- нейронна мережа не натренована в достатній мірі;
- низька швидкість пошуку порівняно з існуючими аналогами;
- залежність від додаткових бібліотек (tensorflow).

13

Рисунок В.13 – Слайд №13

Висновки

- проаналізовано та порівняно існуючі методи реалізації шахових рушіїв;
- реалізовано шаховий рушій з двома методами оцінки позиції: класичним та з використанням нейронної мережі;
- створено програмний продукт, який може підтримувати шахову позицію, генерувати ходи та знаходити найкращий хід в позиції;
- досліджено методи представлення дошки у вигляді вектору ознак та запропоновано власний метод представлення.

14

Рисунок В.14 – Слайд №14

Шляхи для вдосконалення

- реалізувати та оптимізувати паралельний пошук;
- оптимізувати роботу з транспозиційними таблицями;
- зменшити глибину оцінки позиції в датасеті;
- натренувати нейронну мережу на більшій кількості епох;
- застосувати різні методи оцінки позиції для різних етапів гри;
- використовувати нейронну мережу, що ефективно оновлюється замість персептрону.