НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
Інститут прикладного системного аналізу Кафедра
математичних методів системного аналізу

До захисту допущено:
Завідувач кафедри
_____ Олена ЧУМАЧЕНКО
«___»_____2023 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системи і методи штучного інтелекту»
спеціальності 122 «Комп'ютерні науки»
на тему: «Штучний інтелект у шахах з використанням методів теорії ігор»

Виконав: студент IV курсу, групи КІ-93
Ткаченко Максим Романович _____

Керівник: доцент кафедри ММСА,
кандидат фіз-мат. Наук,
Барановська Леся Валеріївна _____

Консультант з нормконтролю:
Гончарук Максим Миколайович _____

Консультант з економічного розділу:
доцент, к.е.н., Рощина Надія Василівна _____

Рецензент:
кандитат технічних наук, доцент
Харченко Костянтин Васильович _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Київ – 2023 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського» Інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олена ЧУМАЧЕНКО
«___»_____20__ р.

ЗАВДАННЯ

на дипломну роботу студенту

Ткаченко Максиму Романовичу

1. Тема роботи «Штучний інтелект у шахах з використанням методів теорії ігор», керівник роботи Барановська Леся Валеріївна, доцент кафедри ММСА, затверджені наказом по університету від «30» травня 2023 р. № 2065-с.

2. Термін подання студентом роботи: 08.06.2023 року.

3. Вихідні дані до роботи: C++, Chess Arena GUI.

4. Зміст роботи: аналіз предметної області, дослідження та огляд методів та алгоритмів теорії ігор для вирішення шахових задач, розробка програми обраними методами, оцінка отриманих результатів.

5. Перелік ілюстративного матеріалу: презентація з результатами виконаних обчислень.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Рощина Н.В., доцент		

7. Дата видачі завдання 21.02.2023 року

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Формулювання тематики дослідження	03.09.2022 – 30.09.2022	Виконано
2	Аналіз актуальності теми	01.10.2022 – 30.10.2022	Виконано
3	Формулювання задачі дослідження	01.11.2022 – 30.11.2022	Виконано
4	Збір інформації, дослідження алгоритмів вирішення задачі	01.02.2023 – 30.02.2023	Виконано
5	Написання програми	01.03.2023 – 30.03.2023	Виконано
6	Оформлення пояснювальної записки	01.04.2023 – 30.04.2023	Виконано
7	Підготовка презентації для захисту	1.05.2023 – 10.05.2023	Виконано
8	Попередній захист дипломної роботи		
9	Захист дипломної роботи		

Студент _____ Максим ТКАЧЕНКО

Керівник роботи _____ Леся БАРАНОВСЬКА

РЕФЕРАТ

Дипломна робота: 70 с., 15 табл., 31 рис., 1 додаток, 18 джерел.

Об'єкт дослідження - комп'ютерні шахові програми (Chess Enignes), які розробляються з метою гри в шахи та надання гравцям різних рівнів навичок можливість аналізувати варіанти шахових позицій.

Предмет дослідження - методи та алгоритми, які використовуються в шахових програмах для знаходження оптимальних ходів та оцінки позицій.

Мета роботи – детальний аналіз шахових програм, їх алгоритмів та евристичних методів, які використовуються для досягнення високого рівня гри в шахи. Створення власної шахової програми, здатної конкурувати з сильними шахистами та програмами аналогами.

Актуальність - дослідження шахових двигунів є актуальним завданням в сучасній комп'ютерній науці. Розробка шахових двигунів має практичну цінність для шахістів всіх рівнів. Програми здатні надати допомогу в аналізі власних партій, виявляючи помилки, що дозволяє шахістам покращувати свої навички та рівень гри.

ABSTRACT

Thesis: 70 p., 15 tabl., 31 fig., 1 appendice, 18 sources.

The research object is computer chess programs (Chess Engines) developed for playing chess and providing players of different skill levels with the ability to analyze chess positions. The research subject is the methods and algorithms used in chess programs to find optimal moves and evaluate positions.

The aim of this study is to conduct a detailed analysis of chess programs, their algorithms, and heuristic methods used to achieve a high level of play in chess. Additionally, the goal is to create a proprietary chess program capable of competing with strong chess players and analogous programs.

The relevance of studying chess engines is a current task in modern computer science. The development of chess engines has practical value for chess players at all levels. These programs can assist in analyzing their own games and identifying mistakes, allowing chess players to improve their skills and level of play.

ЗМІСТ

ВСТУП

Шахи вважаються однією з найстаріших настільних ігор, які до цього дня зберігають свою популярність та привабливість для мільйонів людей по всьому світу. Вони не лише є способом розваги, але й інтелектуальним викликом для тих, хто бажає розвивати свої аналітичні та стратегічні навички.

З розвитком комп'ютерних технологій виникла можливість створити програми, які здатні грати в шахи на рівні, недосяжному навіть для найсильніших шахістів використовуючи розрахункову потужність комп'ютерів для аналізу позицій та вибору найоптимальніших ходів.

Ця дипломна робота присвячена дослідженню таких шахових програм та розробці власної, яка буде здатна конкурувати з сильними шаховими гравцями.

Основною метою роботи є розуміння принципів та алгоритмів, які використовуються в шахових програмах, а також створення ефективного та конкурентоспроможного алгоритму для вирішення шахових задач.

У процесі виконання роботи буде проведений аналіз існуючих шахових двигунів, їх алгоритмів та методів оцінки позицій. Будуть розглянуті основні концепції та техніки, які лежать в основі роботи шахових двигунів, такі як пошук ходів, оцінка позицій, та визначення оптимального ходу у конкретній позиції. Особливу увагу буде приділено алгоритму *Min-Max*, який використовується для знаходження найбільшої вигоди в умовах конкуренції. Також будуть розглянуті методи оптимізації, такі як *Alpha-Beta* відсікання, транспозиційні таблиці та інші евристичні підходи для покращення швидкості обчислень.

Після проведення дослідження буде розроблено власну шахову програму, засновану на досліджених алгоритмах. У процесі розробки будуть враховані принципи оптимізації та швидкодії.

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Основні Поняття

У цьому розділі ми розглянемо основні визначення, пов'язані з шаховими двигунами, зробимо короткий огляду історії шахових програм, а також визначимо, як вони вплинули на саму гру в шахи.

1.1.1 Що таке шаховий двигун

Шаховий двигун (Chess Engine) — Комп’ютерна програма, яка має можливість генерувати легальні (відповідно до правил гри) ходи в конкретній позиції, оцінювати їх і, базуючись на цій оцінці, обирати найсильніше продовження для будь-якого з гравців. Термін "двигун" відноситься до потужної програми, яка здійснює багато пошукових і обчислювальних операцій.

Перша спроба винайти подібну програму була ще в 1912 році, коли створили машину, яка здатна виграти ендшпіль Короля з Турою проти Короля [1].

Але лише у 1951 році Алан Т'юрінг написав комп'ютерну програму [2], яка справді могла повноцінно грати у шахи. Протягом наступних 50 років програмісти працювали над створенням нових підходів для своїх шахових програм, а покращення обладнання дозволяли проводити обчислення глибше та швидше.

У 1997 році, коли найсильніших гравець того часу, гросмейстер Гаррі Каспаров, програв матч проти шахматної машини Deep Blue Expert System [3], світ шахів змінився назавжди.

1.1.2 Як шахові двигуни вплинули на гру людини

Один з найефективніших способів поліпшення своєї гри у шахи - аналізувати зіграні партії. Комп'ютери значно поліпшили цей процес, оскільки використання комп'ютерних програм дозволяє як аматорам, так і професіоналам проводити ретельний аналіз своїх ходів, виявляти помилки та недоліки, а також вивчати різноманітні стратегії та тактики, що значно збагачує їх розуміння гри в цілому.

Кращі гравці сьогодні широко використовують шахові програми для аналізу позицій і генерування ідей. На жаль, це також ввело шахрайство в шахи, де будь-який гравець, який використовує лише мобільний телефон і шахову програму, може грати краще, ніж будь-який гросмейстер.

Інтернет-сервіси для гри в шахи, такі як *chess.com*, щодня виловлюють сотні шахраїв, які не можуть протистояти бажанню виграти партію за допомогою комп'ютера. Тим не менш, шанувальники гри отримали вигоду від таких програм, оскільки вони допомогли гравцям, які бажають вдосконалюються, стати кращими завдяки аналізу партій, а також створили вид спорту для глядачів, де шахові програми борються за звання найсильнішої.

1.1.3 Як працює традиційний шаховий двигун

Шахові двигуни - складні у своїй природі. Проте, у спрощеному вигляді, можна сказати, що вони роблять лише дві речі:

1. Перебір варіантів позицій на задану глибину (Search). Подібно до висококваліфікованих шахістів, програми прагнуть передбачати наслідки своїх ходів наперед. Чим глибше вони можуть рахувати та оцінювати результат, тим кращий хід вони зможуть зробити в заданій позиції. У шахах кожен окремий хід називається "напівходом" (ply), а глибина оцінки позиції визначається пояснюється кількістю півходів. На глибині 20 (10 півходів за білих і 10 півходів за чорних) більшість шахових двигунів вже здатні оцінювати позиції значно точніше, ніж сильніші шахисти.

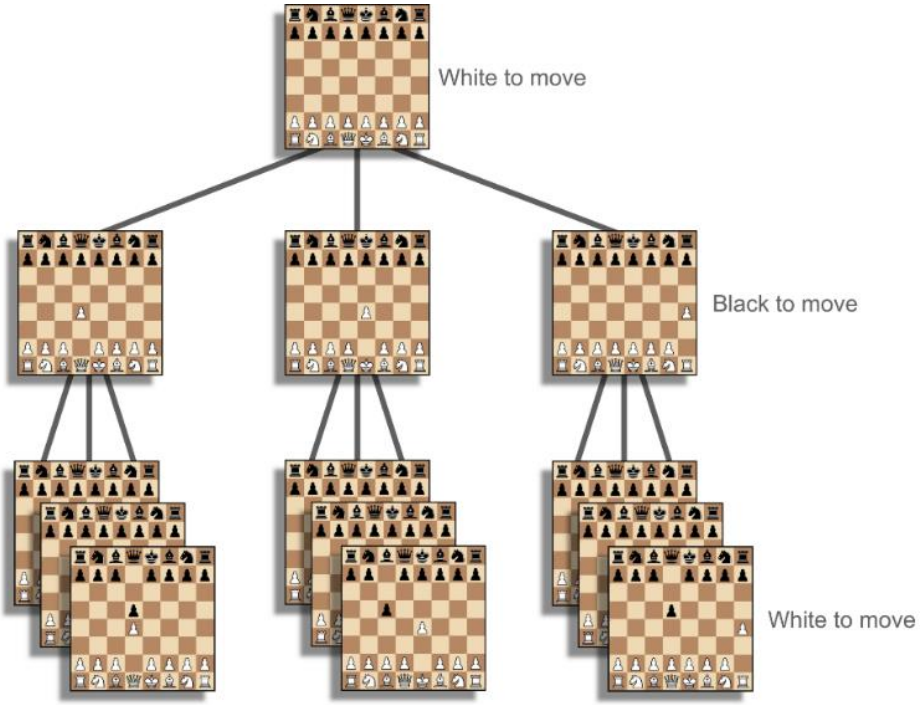


Рисунок 1.1 Ілюстрація дерева гри.

2. Оцінка позиції (Evaluation). Оцінкою займається окрема частина програми, метою якої є відобразити статичне уявлення дошки в числове значення. Результатом цієї оцінки є число, що зазвичай відповідає певному значенню в пішаковому еквіваленту (наприклад, Кінь - оцінюється в три Пішака, Тура – в чотири і т.д.). Кожен шаховий двигун реалізує цю функцію по-різному, але більшість програм звертають увагу на такі аспекти, як матеріальна перевага кожної сторони, загрози на дошці, безпека короля і структура пішачних ланцюгів.

Рисунок 1.2 Ілюстрація функції оцінки.

Використовуючи ці дві можливості програми, ми отримуємо інтуїтивно зрозумілий алгоритм для визначення оптимального ходу в конкретній позиції. У заданій позиції розглядаються всі легальні ходи, а з отриманої позиції аналізуються

всі можливі відповіді на попередній хід. Цей процес триває до досягнення заданої глибини. Після виконання кожного заключного ходу, викликається функція оцінки, щоб визначити, до яких результатів кожна послідовність зіграних ходів призводить. Коли цей процес завершиться, ми отримаємо оцінку для кожного ходу на кожному рівні пошуку в дереві. Оптимальним ходом буде той, який отримає найвищу оцінку, з урахуванням того факту, що після цього ходу суперник також обиратиме хід з найвищою оцінкою відосно себе для продовження гри.

1.1.4 Як комп'ютер представляє шахову дошку

Представлення позиції у грі - відповідальний крок у процесі розробки шахової програми. Від швидкості, з якою програма може вносити зміни на дошці, залежить кінцева сила такої програми. Хоча представлення позиції у вигляді масиву 8x8, де кожна клітина містить інформацію про присутність або відсутність певної фігури, може здатися найпростішим рішенням, яке само собою напрошується, насправді, такий підхід є надзвичайно повільним і неефективним, особливо під час генерації ходів.

Найбільш сильні шахові програми обирають використання бітових структур даних для представлення позиції. Замість масиву 8x8 використовуються бітові маски(BitBoards)[4], де кожен біт відповідає за наявність або відсутність фігури на клітці. Такий підхід дозволяє значно прискорити генерацію ходів, використовуючи бітові операції для виконання їх з масками, такими як об'єднання, перетин та зсув.

Головна причина використання BitBoards – швидкість. За щасливим збігом шахова дошка складається з 64 клітин. Це точно стільки, скільки може займати машинне слово в бітах на сучасних комп'ютерах. Це є головною причиною, чому бітборди використовуються у всіх передових шахових програмах. Використовуючи бітборди будь-яка зміна на дошці буде використовувати лише одну інструкцію на 64-розрядних процесорах, що природно є найшвидшим способом коригування позиції.

Друга причина використання BitBoards полягає в тому, що вони є найбільш ефективним способом зберігання дошки з точки зору використання пам’яті, що стає

актуальним у процесі пошуку, оскільки під час цього процесу в пам’яті потрібно зберігати багато позицій.

Таким чином, використовуючи бітборди, всю позицію можна описати дванадцятьма 64-бітовими числами - одне число на кожен тип фігури кожного кольору. Наприклад, білі Пішаки, що стоять на своїх домашніх клітинах, можуть бути описані числом 65280_{10}, або в двійковій системі числом 1111111100000000_2, значні біти якого відповідають наявності фігури на відповідній клітині на дошці.

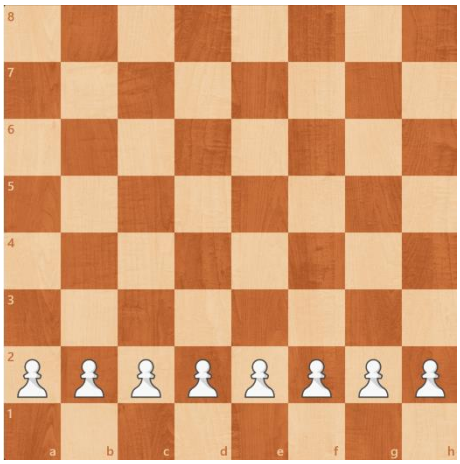


Рисунок 1.3 Білі пішаки на домашніх клітинах

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
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0

Рисунок 1.4 Представлення Пішаків у вигляді 64-бітного числа

1.2 Як визначити силу гравця

Онлайн-платформи та шахові організації використовують систему рейтингу ЕЛО (розроблену Александром Елорем) для надання числової оцінки рівня гри кожного гравця.

При грі гравця А проти гравця В обчислюється математичне сподівання кількості очок, які він має набрати в цій грі за формулою (1) [5]:

$$E_A = \frac{1}{1 + 10^{\frac{R_B - R_A}{400}}} \tag{1.1}$$

де:

E_A - математичне сподівання кількості очок, які набере гравець А в партії з В;

R_A - старий рейтинг гравця А;

R_B - старий рейтинг гравця В;

1.2.1 Класифікація

Приблизне співвідношення рейтингів ЕЛО та шахових звань і розрядів:

- ≥ 2500 — гросмейстер;
- 2400—2499 — міжнародний майстер;
- 2200—2399 — національний майстер;
- 2000—2199 — кандидат у майстри;
- 1800—1999 — 1-й розряд;
- 1600—1799 — 2-й розряд;
- 1400—1599 — 3-й розряд;
- 1200—1399 — середній любитель (4-й розряд);
- 1000—1199 — слабкий любитель (5-й розряд);
- < 1000 — новачок.

1.3 Існуючі шахові двигуни

-  Stockfish 15.1 [6] — шаховий двигун із відкритим кодом, розроблений великою спільнотою ентузіастів програмістів та шахистів. З 2016 року це найпотужніший традиційний шаховий двигун. Багато сучасних підходів до програмування шахових двигунів були впроваджені завдяки Stockfish. Він використовує складну формулу оцінки позиції та AlphaBeta пошук. Stockfish має високий рівень обчислювальної потужності та здатність аналізувати глибокі позиції. Рейтинг ЕЛО - більше 3500 [7].

-  Komodo 14 [8] є одним із найвідоміших шахових двигунів на ринку, перша версія була зроблена у 2010 році. Розробка програми включала співпрацю з великим гросмейстером Ларрі Кауфманном, який допоміг удосконалити оціночні можливості програми. Рейтинг ЕЛО – більше 3200 [7].

1.4 Вибір мови програмування

Мова програмування для реалізації власної шахової програми була обрана C++. Розробка шахової програми вимагає високої швидкодії для обробки великих обсягів даних та виконання значної кількості обчислень. В цьому контексті вибір мови програмування C++ є обґрунтованим, оскільки вона відома своєю ефективністю та можливістю оптимізувати код для досягнення максимальної швидкості. Більш того, C++ надає можливість працювати ближче до низькорівневих аспектів комп'ютера, що є надзвичайно корисним для роботи з бітовими операціями, які використовуються в шахових програмах для обробки позицій. Також C++ має високий ступінь портативності, що означає, що написаний на ньому шаховий двигун може бути легко адаптований і запущений на різних платформах і операційних системах.

1.5 Висновки до розділу 1

У цьому розділі було розглянуто різні аспекти шахових двигунів та їх роль у світі шахів. Були представлені деякі факти з історії розвитку шахових двигунів з моменту їх з'явлення в середині 20-го століття до сьогодення. Був згаданий матч між відомим гросмейстером і шаховою програмою, який привернув велику увагу і став віхою в розвитку шахових двигунів.

Також було обговорено основну функціональність шахових двигунів, включаючи їх здатність оцінювати ходи, а також вибирати оптимальні варіанти ходів для гравців. Були згадані деякі відомі шахові двигуни, такі як Stockfish, які відіграють важливу роль у сучасній шаховій практиці.

Крім того, були піднесені питання про вплив шахових двигунів на шахову культуру та навчання шахам. Шахові двигуни стали цінним інструментом для тренування та аналізу партій.

У заключній частині було обґрунтовано вибір мови програмування для реалізації подальшої прикладної частини даної роботи.

РОЗДІЛ 2. АЛГОРИТМИ ТА ЕВРИСТИЧНІ ПІДХОДИ ДЛЯ РОЗВ'ЯЗАННЯ ШАХОВИХ ЗАДАЧ

2.1 Задача Пошуку

Будь-яку шахову партію можна представити у вигляді гіганського n-арного дерева, а потім виконати пошук по цьому дереву, щоб знайти послідовність ходів (Principal Variation), які програма буде вважати найкращими. Найвідомішим і широко використовуваним алгоритмом для вирішення такої задачи є алгоритм *Minimax.*

2.1.1 Алгоритм Minimax.

Алгоритм *Minimax* — це рекурсивний алгоритм теорії ігор, який знаходить оптимальний хід для гравця за умови, що суперник також грає оптимально.

У основі алгоритму *Minimax* лежить ідея мінімізації можливих втрат і максимізації можливих виграшів. Він розглядає ходи як частину дерева гри, де на кожному рівні гравці максимізують свій виграш, а противники мінімізують його. Алгоритм *Minimax* просувається по дереву гри, розглядаючи різні можливі ходи та їх наслідки, і намагається знайти найкращий хід, який максимізує шанси на перемогу.

Припустимо, що гравець w вибирає хід m_w, а суперник b вибирає m_b. Тоді якщо E_w є функцією оцінки відносно гравця w маємо формулу [9] (2):

$$E_w = \max_w \min_b E_w(m_w, m_b) \tag{2.1}$$

Розглянемо як цей алгоритм працюватиме на практиці. Припустимо, що у позиції, яка розглядається хід білих. Викликається функція *Max()* в якій генеруються всі допустимі ходи білих. Після застосування кожного ходу по черзі, в отриманій позиції викликається функція *Min()*. Функція *Min()* оцінює позицію та повертає

значення. Оскільки зараз хід належить білим, вибирається хід із найбільшим *позитивним* значенням, і це значення повертається.

Функція *Min()* працює в "реверсному" режимі відносно функції *Max()*. Функція *Min()* викликається, коли хід належить чорним, тому обирається хід з найбільш *негативним* значенням.

Ці функції є взаємно рекурсивними, що означає, що вони викликають одна одну, поки не буде досягнута бажана глибина пошуку. Коли функції досягають "дна" пошукового дерева, вони повертають результат функції, яка оцінює позицію, що вийшла.

Отже, шляхом перебору всіх можливих ходів з обох сторін та після досягнення відповідної глибини та оцінки позиції, рекурсивно повертаючи оцінку вгору, програма може обирати найсильніші ходи за кожну зі сторін.

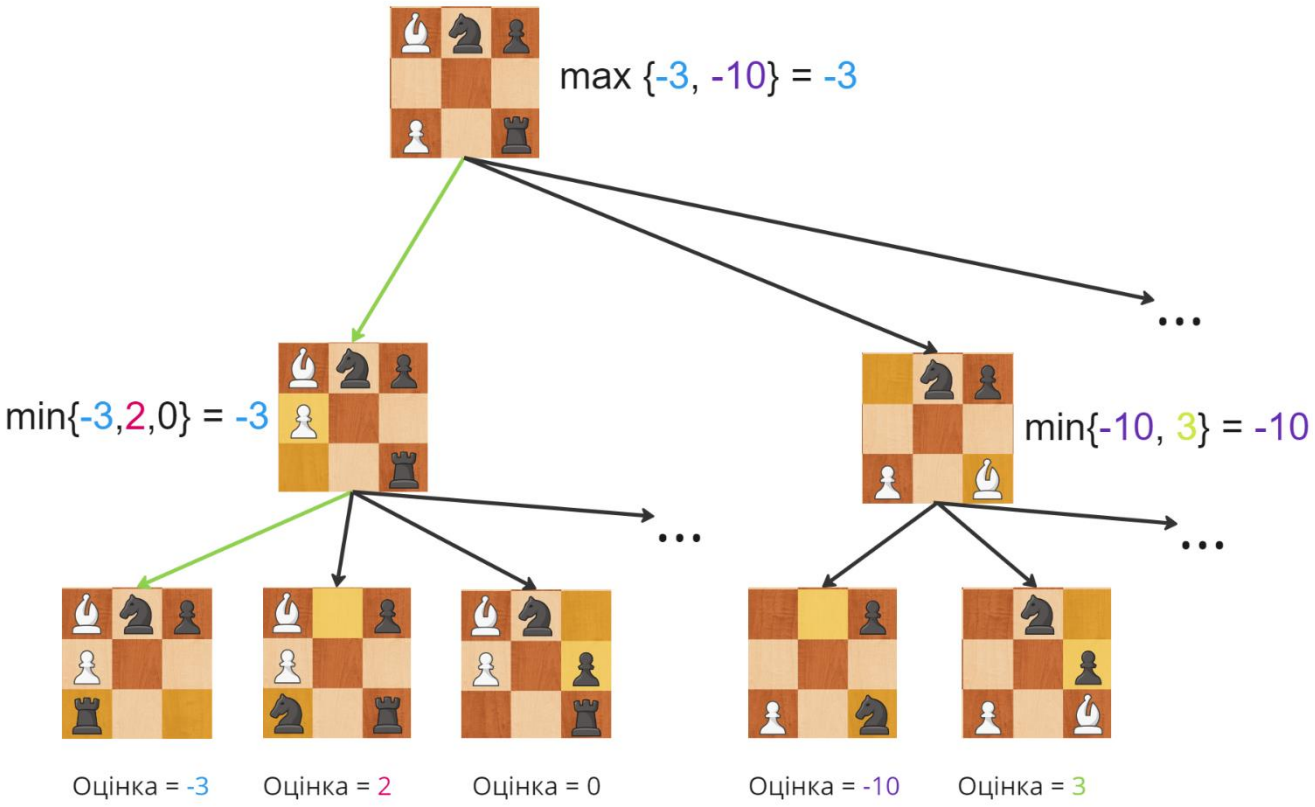


Рисунок 2.1 Приклад роботи алгоритму Minimax

```
Algorithm  Minimax
function MIN(depth)                          function MAX(depth)
    if depth == 0 then return Evaluate()         if depth == 0 then return Evaluate()
    best = +INF                                  best = -INF
    GenerateMoves(list)                          GenerateMoves(list)
    for move in list do                          for move in list do
        DoMove(move)                                 DoMove(move)
        score = Max(depth − 1)                       score = Min(depth − 1)
        UndoMove(move)                               UndoMove(move)

        if score < best then                         if score > best then
            best = score                                 best = score
            remember move                                remember move
    return best                                  return best
```

Рисунок 2.2 Псевдокод алгоритму Minimax

На практиці алгоритм *Minimax* є дуже повільним, оскільки кількість вузлів, які потрібно обробити, зростає експоненційно залежно від коефіцієнта розгалуження. Оскільки у шахах коефіцієнт розгалуження становить приблизно 35 [10], то для виконання пошуку на один напівхід потрібно обробити 35 вузлів, для пошуку на два напівходу - приблизно 1200 вузлів, для пошуку на три - приблизно 42 800 вузлів, для пошуку на чотири - приблизно півтора мільйона вузлів і так далі. Зазвичай, для сильної гри потрібно розглядати не менше 12 напівходів вперед, а отже використання класичного алгоритму мінімаксу для такої мети буде неоптимальним вибором через сильний зріст необхідних обчислень на кожному етапі пошуку.

2.1.2 Алгоритм Alpha-Beta

Алгоритм *Alpha-Beta* дуже схожий на алгоритм *Minimax*, і по суті він є лише його доповненням. Алгоритм *Minimax* вибираючи найкращий PV повністю досліджує ігрове дерево. Вибрана сторона намагається досягти найкращого результату з огляду на те, що суперник зробить усе можливе, щоб перешкодити цьому. Як ми вже зрозуміли, алгоритм *Minimax* неефективний, коли мета стоїть у розробці сильної шахової програми, яка здатна глибоко досліджувати позицію. Пошук на кожний додатковий напівхід призводить до експоненційного збільшення розміру ігрового дерева. Алгоритм *Alpha-Beta* допомагає вирішити цю проблему, зменшуючи фактор розгалуження шляхом відсічення свідомо поганих гілок

пошукового дерева, зберігаючи при цьому точну оцінку алгоритма *Minimax*, але не потребуючи проходити та оцінювати кожен вузол пошукового дерева [11].

Алгоритм базується на концепції, що якщо ми вже маємо гарантовану оцінку *X* у певній вершині, то ми можемо ефективно використовувати цю інформацію для зупинки пошуку в дочірніх вузлах, як тільки стає очевидним, що їх оцінки не змінять уже відому оцінку *X* для батьківської вершини.

У ситуації першого прикладу для алгоритму *Minimax*, при пошуку оптимального PV з використанням алгоритму *Alpha-Beta*, відбудеться принаймні два відсікання. При обході дерева в глибину після ходу білою пішки стає очевидним, що поточна сторона гарантовано отримає оцінку гіршу, ніж у вже обчисленому альтернативному продовженні. На цій підставі можна припинити подальшу оцінку поточного вузла, без будь-якого ризику упустити важливу інформацію.

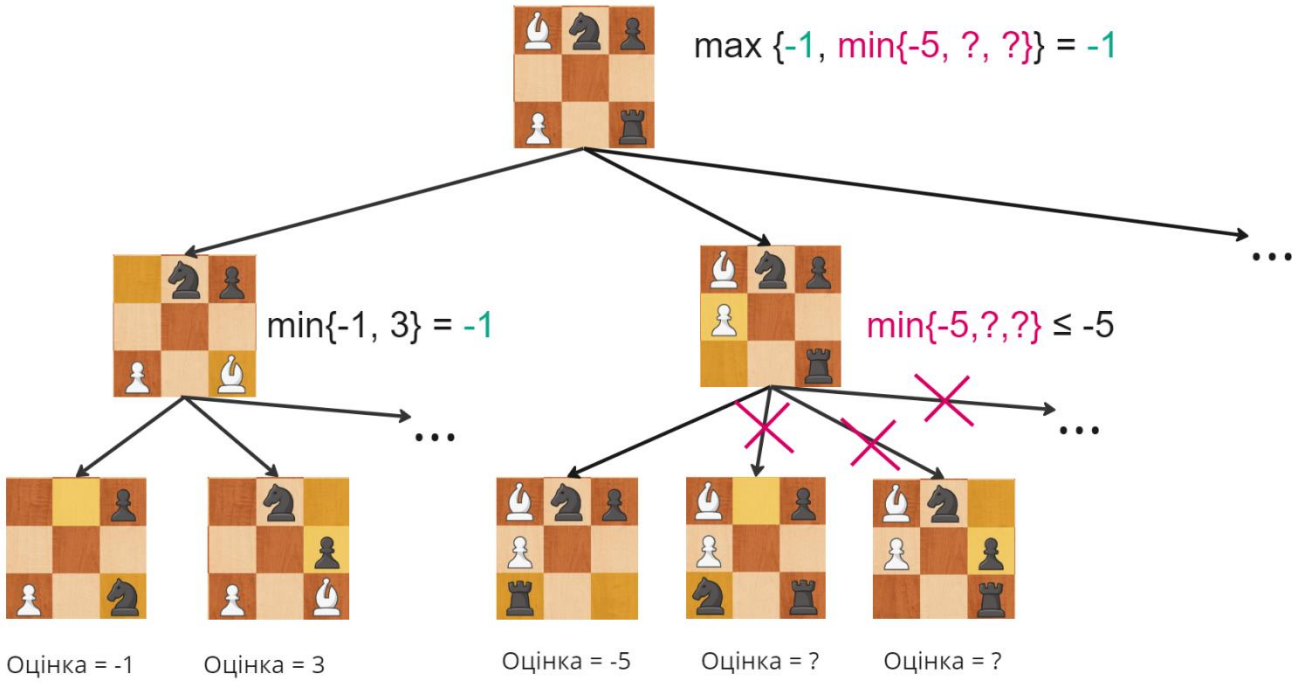


Рисунок 2.3 Приклад роботи алгоритму *Minimax* з відсіканнями

Ідея алгоритму полягає у передачі двох оцінок під час пошуку. Перша оцінка - альфа, представляє найкращий результат, який можна досягти з вже знайденої стратегії (досліджена PV). Усі ходи, що призводять до оцінки меншої за альфу є неоптимальними, оскільки існує стратегія, яка, як відомо, призводить до значення

альфи. Таким чином, будь-що, що менше або дорівнює альфі, не є поліпшенням поточної позиції.

Друга оцінка – бета. Бета - це гірше, на що може розраховувати супротивник, тому що відомо з вже обчислених PV, що противник має спосіб форсувати позицію не гірше за бету. Якщо пошук знаходить щось, що дає результат бета або більше, то це означатиме, що на попередній хід противника існує контрвідповідь, і відповідно розглядати цю позицію далі немає сенсу (бо вже відомо, що противник нічого не досягне своїм ходом, а значить, можна вважати, що він його і не зробить), тому решту легальних ходів не потрібно шукати та оцінювати.

Під час пошуку ходів кожен шуканий хід повертає оцінку, яка має певне відношення до значень альфа та бета, і саме на підставі цього співвідношення можна зробити висновок про перерву подальшого пошуку та повернути значення. Можна виділити три види таких співвідношень:

- $score \leq alpha$

  Якщо хід призводить до результату, який менший або дорівнює альфі, то хід є поганим, і про нього можна забути, оскільки, як я вже було сказано раніше, на цьому етапі відома стратегія, яка при оптимальній грі за обидві сторони приведе до позиції, яка оцінюється в альфу.

- $score \geq beta$

  Якщо хід призводить до результату, більшого або рівного бета, весь поточний вузол може вважатися " недосяжним", оскільки противник не дозволить стороні, яка здійснює хід, досягти цієї позиції, бо у противника є певний вибір, який дозволить уникнути цього. Отже, як тільки ми знаходимо що-небудь з оцінкою бета або більше - подальшу частину пошукового дерева, можна відсікати.

- $alpha \leq score \leq beta$

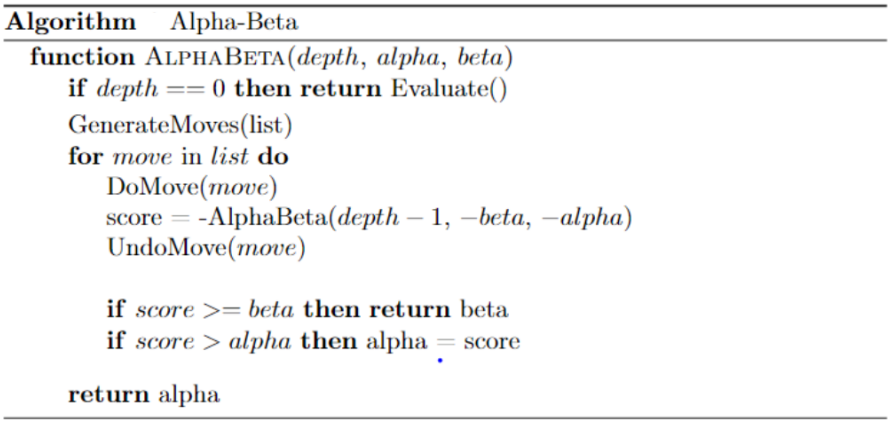
  Якщо хід призводить до результату, в якому оцінка знаходиться в межах альфа та бета, цей хід розглядається як кандидат на оптимальний хід. Значення оцінки стає новим альфа-значенням для поточної вершини. Пошук у цій

вершині продовжується, поки не будуть оцінені всі легальні ходи. Останній хід, який потрапляє в діапазон $[alpha;\ beta]$, вважається оптимальним.

Таким чином, простий додаток до алгоритму *Minimax* може суттєво зменшити коефіцієнт розгалуження пошукового дерева. Однак, це не є гарантованим. Алгоритм *Alpha-Beta* залежить від порядку оцінювання ходів, отже ефективність роботи алгоритму залежатиме наскільки добре ходи відсортовані. Якщо найгірші ходи будуть оцінюватися першими, бета-відсікання ніколи не станеться. В результаті алгоритму доведеться пройти все дерево пошуку так само як і алгоритму *Minimax*. Якщо програмі завжди буде вдаватися оцінювати найкращий хід для пошуку першим, то коефіцієнт розгалуження може буде зменьшений приблизно до квадратного кореню очікуваного коефіцієнта розгалуження [12], а саме до $\sqrt{35}$. Це значне поліпшення, і воно дозволяє шукати вдвічі глибше в тій же кількості вузлів за той же час.

Реалізація алгоритму в шаховій програмі виглядатиме приблизно наступним чином: функція AlphaBeta приймає значення глибини, на яку необхідно здійснити пошук, а також значення альфа і бета. Початкові значення альфа і бета встановлюються як найгірші для конкретної позиції. Тобто альфа встановлюється як найменше можливе значення, а бета - найбільше можливе значення. Потім у розглянутому вузлі генеруються всі легальні ходи, які застосовуються по черзі до поточної позиції. Після застосування кожного ходу викликається рекурсивно знов функція AlphaBeta зі значенням глибини зменшеним на одиницю, а також значеннями поточні альфа і бета відносно сторони противника. Коли виклик функції повертає результат, ми перевіряємо, чи була отримана оцінка більша за бета. Якщо так, відбувається відсічка, і функція повертає значення бета. Якщо ні - перевіряємо, чи оцінка є більшою за поточну альфа. Якщо так, оновлюємо поточне значення альфа отриманою оцінкою. Коли рекурсивне поглиблення досягає заданої глибини, $depth = 0$, повертається значення статичної функції оцінки для отриманої позиції. Процес пошуку в поточному вузлі триває до тих пір, поки не будуть досліджені всі згенеровані ходи, або не буде зроблено бета-відсікання. Якщо бета-

відсікання не сталося, повертається значення альфа. Псевдокод алгоритму наведено на рисунку.



```
Algorithm    Alpha-Beta
  function AlphaBeta(depth, alpha, beta)
    if depth == 0 then return Evaluate()
    GenerateMoves(list)
    for move in list do
      DoMove(move)
      score = -AlphaBeta(depth − 1, −beta, −alpha)
      UndoMove(move)

      if score >= beta then return beta
      if score > alpha then alpha = score

    return alpha
```

Рисунок 2.4 Псевдокод до алгоритма AlphaBeta.

2.1.3 Евристика нульового ходу

Одна з найефективніших технік для ще більшого зменшення коефіцієнта розгалуження під час пошуку є *Null-Move Pruning* [13].

Евристика *Null-Move* дозволяє шаховій програмі значно скоротити кількість досліджень у дереві пошуку з деяким керованим ризиком упустити щось важливе. Це призводить до збільшення глибини пошуку програми та у більшості випадків помітно скорочує кількість досліджуваних тактик, необхідних для прийняття рішення про оптимальний хід.

Евристика *Null-Move* — крок, який виконується перед початком пошуку. Ми запитуємо: «Якщо я в цій позиції нічого не зроблю, чи зможе противник щось зробити проти мене?». У звичайному випадку, коли ми описували алгоритм *Minimax*, ми не ставили це питання. Ми запитували: «Як найсильніше нашкодити противнику?». Запитувати, чи може противник нашкодити нам це зовсім інше.

Виявляється, існує багато випадків, коли противник не може нам зашкодити. Наприклад, припустимо, у нас є зайва Тура, і жодна з наших фігур не перебуває під атакою. В цьому випадку противник робить випадковий хід з доступних, але у нас все одно залишається перевага, через зайву Туру. Суть *Null-Move Pruning* полягає в

тому, щоб уникнути дослідження таких позицій і якомога раніше викликати бета-відсічку.

Досягається це, даючи противнику зробити два ходи поспіль. Якщо наша позиція все ще настільки хороша, що отриманий результат перевищує значення бета, ми припускаємо, що також перевищимо бета, якби досліджували всі доступні ходи в позиції. Псевдокод наведено на рисунку.

```
Algorithm  Alpha-Beta з Null-move pruning
function AlphaBeta(depth, alpha, beta)
    if depth == 0 then return Evaluate()
    DoNull()                              ▷ даємо противнику ще один хід
    score = -AlphaBeta(depth − 1,−beta, −beta + 1)
    UndoNull()
    if score >= beta then return beta

    GenerateMoves(list)
    for move in list do
        DoMove(move)
        score = -AlphaBeta(depth − 1,−beta, −alpha)
        UndoMove(move)

        if score >= beta then return beta
        if score > alpha then alpha = score

    return alpha
```

Рисунок 2.5 Псевдокод евристики *Null-Move Pruning* в алгоритмі *Alpha-Beta*

Загалом, данний підхід дозволяє швидше ідентифікувати неперспективні розвитки позиції, тим самим заощаджуючи час, який було б витрачено на їх більш глибший аналіз, який, ймовірно, привів би до тих самих висновків.

2.1.4 Ефект горизонту

У шахах виникає багато форсованих ситуацій, коли необхідно провести серію взяттів у відповідь на взяття противника. Наприклад, якщо хтось бере вашого Слона Ронем, вам краще забрати його Роня у відповідь.

Передаючи у функцію фіксоване значення глибини, на яку буде здійснюватися пошук, всі варіанти розвитку подій, які бачить алгоритм, будуть обмежені цим значенням. Через це, можуть виникати ситуації, коли на останньому півході пошуку сторона бачить можливість покращити свою оцінку шляхом взяття фігури, але не

бачить майбутньої відповіді про взаємне взяття на наступному півході. Таким чином, при використанни фіксованого значення глибини під час пошуку, є можливість пропустити деякі важливі тактичні комбінації, які можуть кардинально змінити значення оцінки позиції. Це явище отримало назву “Ефект Горизонту” [14].

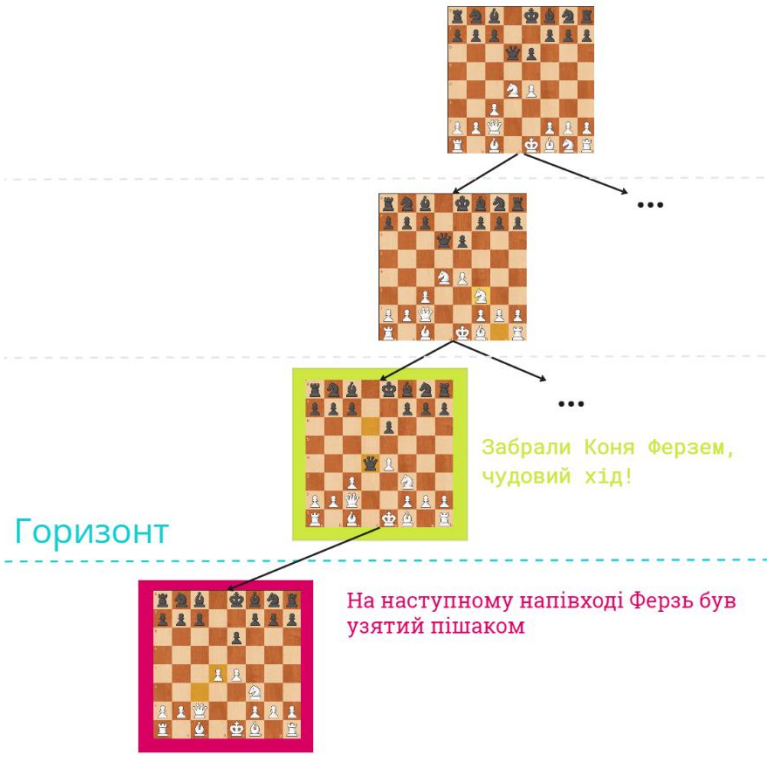


Рисунок 2.6 Приклад ефекту горизонта.

Метод боротьби з цим ефектом називається *Quiescence Search* [15]. Коли *Alpha-Beta* досягає глибини свого пошуку, замість виклику статичної функції оцінки викликається "неактивна" функція пошуку, яка не має фіксованої глибини пошуку. Натомість критерієм зупинки є відсутність доступних ходів, які роблять взяття. Алгоритм *Quiescence Search* дуже схожий на алгоритм *Alpha-Beta*, але з важливою відмінністю. У *Quiescence Search*, відразу викликається статична функція оцінки, і якщо оцінка достатньо хороша, щоб викликати відсічення без подальшого перебору ходів, то це одразу і робиться (повертається значення бета). Якщо оцінка не є настільки хорошою, щоб викликати відсічення, але краща, ніж альфа, то альфа оновлюється, щоб відобразити оцінку поточної позиції. Після цього алгоритм працює так само, як *Alpha-Beta*, з тією відмінністю, що в кожному вузлі генеруються

та досліджуються лише ходи, які атакують. Псевдокод алгоритму *Quiescence Search* наведено на наступному рисунку.

```
Algorithm   Quiescence Search
  function QSearch(alpha, beta)
    score = Evaluate()
    if score >= beta then return beta
    if score > alpha then alpha = score

    GenerateMoves(list)
    for move in list do
      DoMove(move)
      score = -QSearch(−beta, −alpha)
      UndoMove(move)

      if score >= beta then return beta
      if score > alpha then alpha = score

    return alpha
```

Рисунок 2.7 Псевдокод алгоритму Quiescence Search.

Ефективність роботи даного алгоритму, так само як і алгоритму альфа-бета, залежить від послідовності вибору ходів. Для досягнення найбільшої кількості бета-відсічок і і внаслідок цього прискорення процесу пошуку найкращого ходу, необхідно намагатися оцінювати найкращі ходи першими. Існує кілька підходів для досягнення цього. Найпопулярнішим і найпростішим є *MVV-LVA (Most Valuable Victim - Least Valuable Aggressor)* [16], або якщо перекладати українською - *Найбільш цінна жертва - Найменш цінний агресор*, і сама назва відображає його суть. Цей підхід ґрунтується на припущенні, що найкращим ходом буде той, який дозволяє взяти фігуру, що має найвищу вартість. У випадку, коли кілька фігур можуть атакувати найціннішу фігуру, припускається, що кращим варіантом буде той, який використовує найменш цінну фігуру для взяття. Такий хід буде оцінюватися першим з надією, що він виявиться настільки вигідним, що можна відразу виконати бета-відсікання без подальшої оцінки всіх наступних ходів у списку. Це означає, що PxQ буде відсортовано першим. Далі йде NxQ або BxQ, потім RxQ, потім QxQ, потім KxQ, потім NxQ і так далі.

2.1.5 Таблиці Транспозицій

Шахове дерево можна розглядати як граф, в якому одна й та сама позиція може бути досягнута різною послідовністю ходів. Для виявлення таких транспозицій можна використовувати хеш-таблицю, що буде зберігати певну інформацію про вже досліджені позиції. Пізніше, у разі знаходження цієї самої позиції в іній частині дерева, ця інформація буде використовуватись, щоб уникнути повторення тієї ж роботи. Наприклад, якщо позицію після *1. e4 d6 2. d4* вже була розглянута у дереві пошуку, то немає сенсу заново оцінювати позицію після *1. d4 d6 2. e4.*

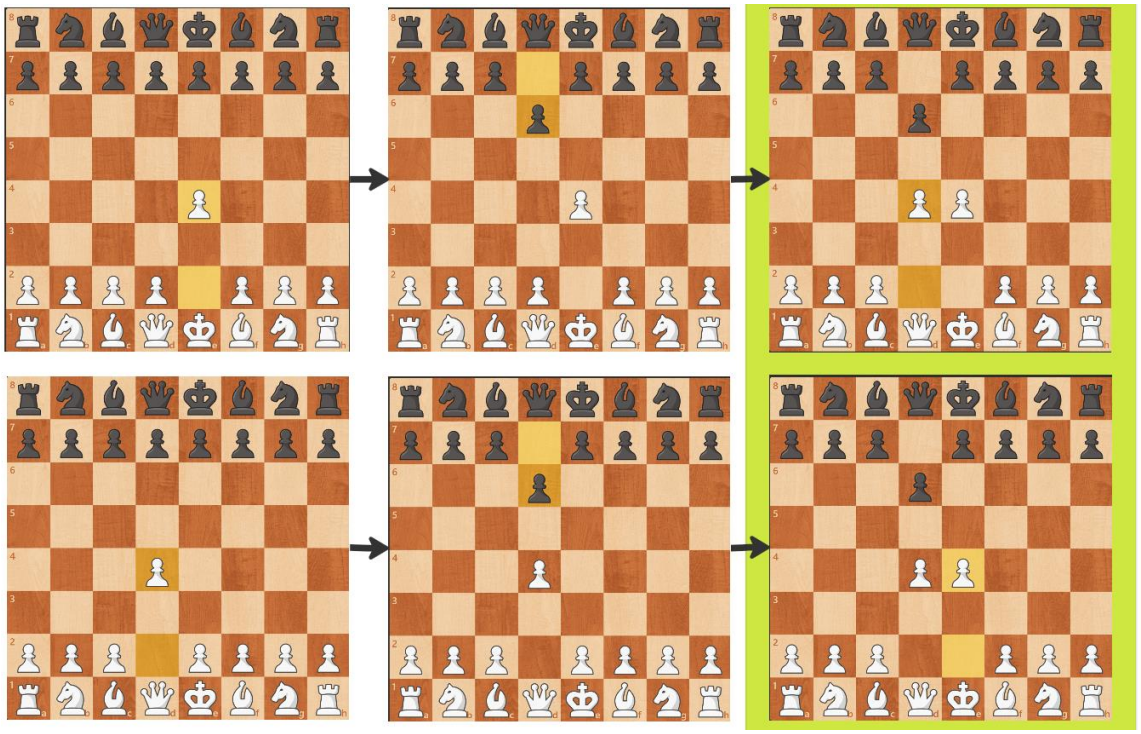


Рисунок 2.8 Приклад транспозиції.

У деяких позиціях, таких як завершення Король+Пішак з заблокованими Пішками у ендшпілі, кількість транспозицій настільки величезна, що їх виявлення є надзвичайно ефективною оптимізацією, завдяки якій глибокий аналіз позиції виконується за лічені секунди.

Використання наявної оцінки при виявленні транспозицій – лише одна функція хеш-таблиці. У звичайних міттельшпільових позиціях так само важливим є інше її застосування - використання збереженого найкращого ходу. Як вже було згадано в описі алгоритму *Alpha-Beta*, оцінка хороших ходів у першу чергу робить

пошук набагато ефективнішим за рахунок більшої кількості бета-відсікань. При дослідженні позиції, якщо не вдалося використати збережену оцінку для позиції, то хід, який для цієї позиції запам'ятався як "найкращий", має сенс оцінювати першим, сподіваючись, що це покращить порядок ходів, що на практиці майже завжди так і відбувається.

Хеш-массив індексується за допомогою ключа Zobrist. Ключ zobrist - це 64-бітне число, яке представляє закодоване представлення позиції. Воно обчислюється наступним чином:

- Ініціалізація: Для кожної клітинки шахової дошки і для кожної фігури кожного кольору генерується випадкове 64-бітне числове значення (маска). Також генерується додаткова маска для майбутнього кодування сторони, яка має зробити хід у позиції.
- Кодування: Для кожної присутньої фігури на дошці за допомогою операції XOR додаються відповідні маски, які були визначені на етапі ініціалізації. До Остаточного ключа zobrist також додається маска сторони, що має робити хід.
- Оновлення: При кожному зсуві фігури або зміні ходу гравця за допомогою операції XOR відбувається оновлення ключа шляхом вилучення попередньої маски і додавання нової.

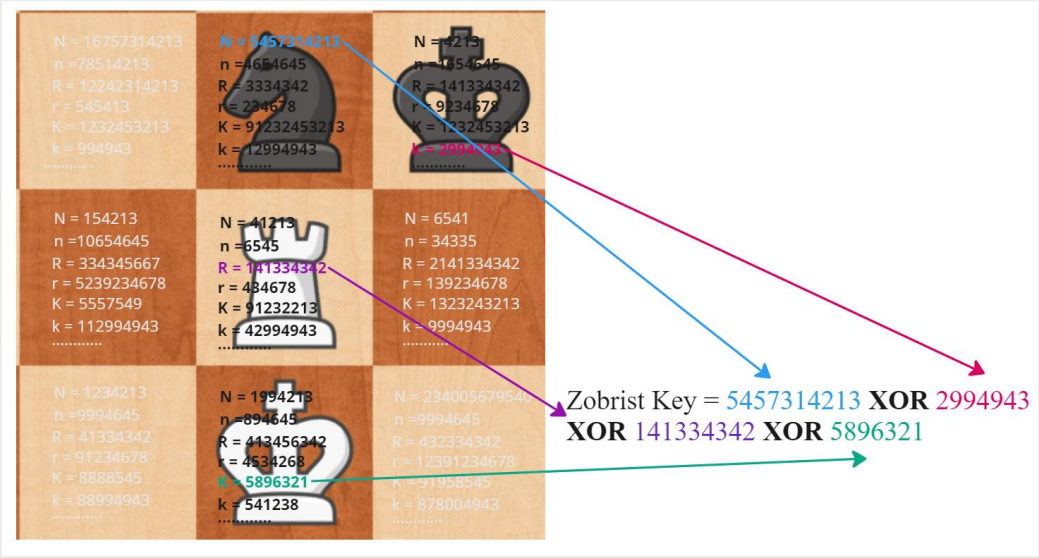


Рисунок 2.9 Приклад розрахунку zobrist-ключа.

Завдяки властивості XOR, при невеликих змінах позиції велика кількість бітів в zobrist-хеші залишається незмінною, що дозволяє ефективно виявляти транспозиції.

При збереженні інформації про позицію, для подальшого використання цих даних у випадку транспозиції, важливо також вказувати на якій глибині був проведений пошук. Це робиться для того, щоб в момент зчитування з таблиці визначити, наскільки ці дані будуть актуальні щодо поточної позиції. Якщо запис про позицію був досліджений на меншій глибині, ніж потрібно для поточного пошуку, то така інформація про оцінку з хеш-таблиці не може вважатися точною.

Невеликий недолік використання альфа-бета алгоритму полягає в тому, що ми часто не знаємо точно, наскільки поганий або хороший є вузол, ми просто знаємо, що він достатньо поганий або достатньо хороший, щоб більше не витрачати час на його подальше дослідження. Для коректного використання записаної оцінки позиції потрібно врахувати цю особливість алгоритму, а саме додавати до запису інформацію про те, до якого типу належала оцінка. Будь-який вузол в альфа-бета-пошуку належить до одного з трьох типів:

1. Alpha-вузол. Кожен хід в цьому вузлі має значення менше або рівне альфі. Це означає, що жоден з ходів в даному вузлі не є хорошим. Ймовірно, це відбулося через те, що позиція з самого початку була невигідна для сторони, яка має зробити хід.
2. Beta-вузол. Принаймні один із ходів поверне значення, більший або рівний бета.
3. PV-вузол. Один або кілька ходів поверне оцінку, більшу за альфа але жоден не поверне оцінку, більшу або рівну бета.

Таким чином, враховуючи все перераховане вище, приходимо до наступної оптимізації алгоритму *Alpha-Beta*. У кожному вузлі, перед генерацією та оцінкою ходів, перевіряється, чи була вже записана будь-яка інформація про поточну позицію, якщо так, і ця інформація актуальна для нас, то залежно від типу записаної оцінки перевіряється, чи отримана оцінка суперечить поточному пошуку. Якщо ні,

то повертаємо записану оцінку без подальшого дослідження вузла. Псевдокод наведено на рисунку.

Algorithm 1 Alpha-Beta з використанням TranspositionTable

```
function AlphaBeta(depth, alpha, beta)
    if ReadHashEentry() != Empty then
        if hashDepth >= depth then
            if hashType == PVnode then return hashScore
            if hashType == AlphaNode then
                if hashScore <= alpha then return alpha
            if hashType == BetaNode then
                if hashScore >= beta then return beta

    if depth == 0 then
        return Evaluate()

    hashType = AlphaNode
    GenerateMoves(list)
    for move in list do
        DoMove(move)
        score = -AlphaBeta(depth − 1,−beta, −alpha)
        UndoMove(move)

        if score >= beta then
            hashType = BetaNode
            WriteHashEntry(depth, hashType, bestMove, score);
            return beta
        if score > alpha then
            hashType = PVnode
            alpha = score
            bestMove = move

    WriteHashEntry(depth, hashType, bestMove, score);
    return alpha
```

Рисунок 2.10 Псевдокод використання таблиць транпозицій в алгоритмі *Alpha-Beta*

2.2 Задача оцінки позиції

Функція оцінки є серцем будь-якої шахової програми. Без сильної функції оцінки шаховий двигун не має жодних знань про шахи, окрім правил гри. Без функції оцінювання нам довелося б шукати всі можливі ходи в шахах, поки не буде знайдено виграшну позицію, що, звичайно, неможливо через надзвичайно велику кількість варіантів розвитку гри.

Сильна функція оцінки забезпечує баланс між точною оцінкою переваги позиції та швидкістю її обрахунку. Очевидно, ми хочемо, щоб наша функція оцінки була точною, щоб двигун робив ходи, які дійсно призводять до переваги. Проте, функція оцінки також повинна бути швидкою, оскільки чим швидше вона працює, тим більше можливих ходів у майбутнє може бути розглянуто за відведений час.

На жаль, на практиці швидкість і точність - дві протилежності. Досягнення високої точності вимагає ретельного вивчення безлічі характеристик аналізованої позиції з урахуванням багатьох деталей. Наприклад, одним з найважливіших компонентів у більшості функцій оцінювання є вартість кожної фігури. Коли людину навчають грати в шахи, зазвичай їй пояснюють, що Кінь та Слон - коштує три пішаки, Тура - п'ять пішаків, Ферзь - десять. Однак на більш глибокому рівні аналізу це не зовсім так. Вартість фігури також буде залежити від її розташування на дошці та від решти позиції в цілому. Тура на відкритій вертикалі коштує більше за звичайну, Слон, якого блокують свої ж Пішаки – менше. Здвоєні Пішаки і Кінь на краю дошки теж коштуватимуть менше. Існує безліч таких критеріїв, які застосовуватимуться до всіх фігур залежно від зіграного дебюту та етапу гри.

2.2.1 Спрощена функція оцінки

У спрощеному вигляді функція оцінки визначає перевагу однієї сторони над іншою, використовуючи лише статичну оцінку матеріалу на дошці та обмежену оцінку потенціала кожної з фігур, яке було наперед визначеною для кожної фігури в кожній клітині дошки.

Оцінка матеріалу полягає в обчисленні різниці у кількості конкретного типу фігур між гравцями і множенні цієї різниці на вартість цього типу фігури. Припустимо, що вартість кожної фігури відповідає значенням в таблиці.

Тип фігури	**Вартість**
Пішак	100
Кінь	300
Слон	300
Тура	500
Ферзь	900

Таблиця 2.1 Приблизна вартість фігури за її типом

Та маємо позицію, яку наведено на наступному рисунку.

Рисунок 2.11 Позиція, де білі мають зайвого коня

Тоді матеріальна оцінка відносно білих для позиції на наступному рисунку буде рахуватися наступним чином:

$$E_{\text{матеріал}} = \left(8_P - 8_p\right) * 100 + (2_N - 1_n) * 300 + (2_B - 2_b) * 300 + (2_R - 2_r) * 500 + \left(1_Q - 1_q\right) = \mathbf{300}$$

Розрахунок потенціалу кожної фігури включає в себе додавання певного бонусу або штрафу до попередньої оцінки в залежності від положення фігури на дошці. Розглянемо кожну фігуру окремо:

- Пішаків варто спонукати до просування вперед. Крім того, також має сенс заохочувати двигун, щоб він залишав Пішаки, які захищають Короля після рокірування на початкових позиціях. З таких міркувань, таблиця бонусів для білих Пішаків буде виглядати приблизно як на рисунку 2.12.
- Коней варто заохочувати йти до центру, бо саме там вони найефективніші. Стояти на краю – погана ідея. Стояти в кутку – ще гірше. З таких міркувань, отримаємо таблицю на рисунку 2.13.

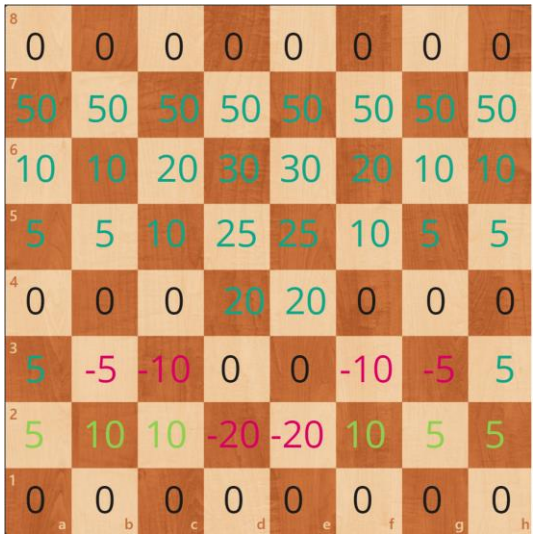


Рисунок 2.12 Оцінки потенціала білих Пішаків

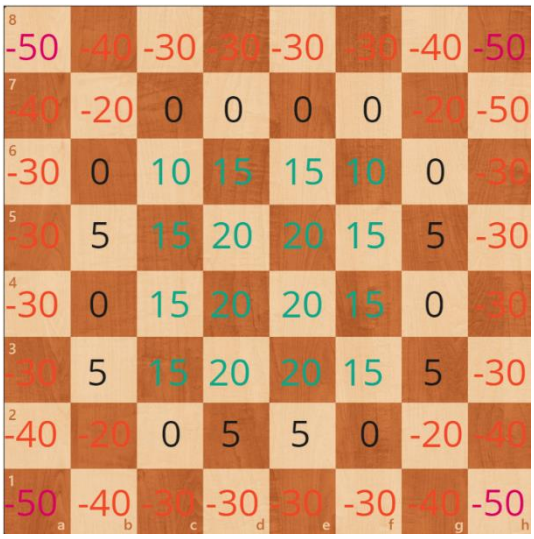


Рисунок 2.13 Оцінки потенціала коня

- Слонів варто заохочувати займати довгі діагоналі і перебувати здебільшого у центральних клітинах, уникаючи країв і кутів. З таких міркувань, таблиця бонусів для білих Слонів буде виглядати як на рисунку 2.14.
- Туру слід заохочувати займати центральні лінії і за можливості передостанній ряд противника, рисунок 2.15.

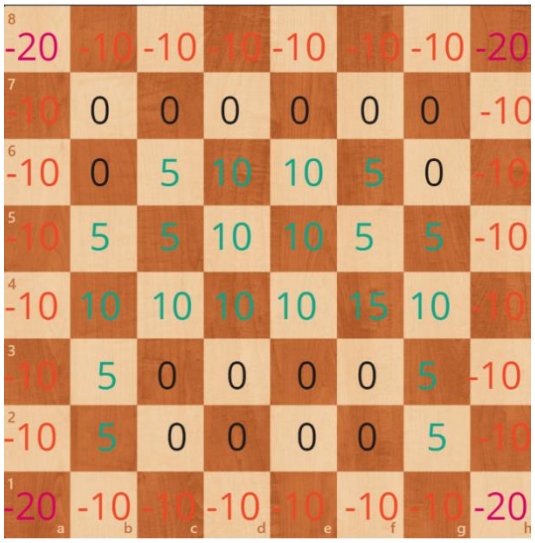


Рисунок 2.14 Оцінки потенціала білого Слона

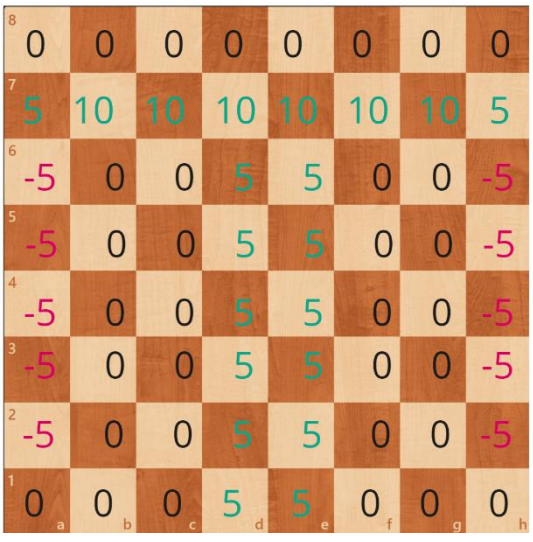


Рисунок 2.15 Оцінки потенціала білої Тури

- Ферзя, як правило, має сенс розміщувати в клітинах, де він буде найбільш мобільний. Тому заохочуємо його перебування в центрі і додаємо штраф, якщо він стоїть на краю або в кутку, рисунок 2.16.

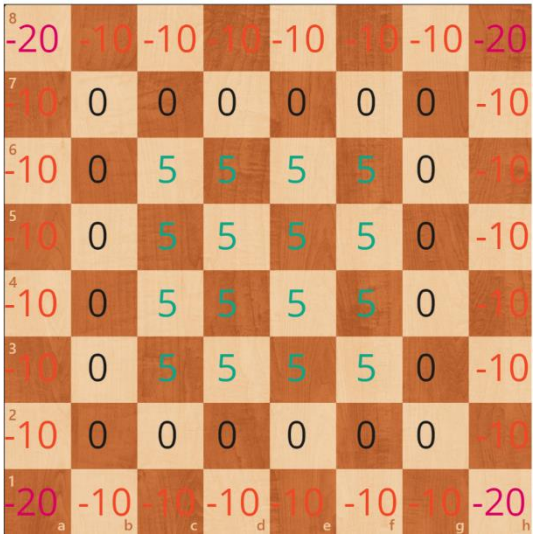


Рисунок 2.16 Оцінки потенціала Ферзя

- Що стосується Короля, тут можна виділити два шаблони його поведінки, які залежать від етапу гри. Якщо це початок або середина партії, варто заохочувати Короля залишатися на своєму домашньому ряді і надавати бонуси клітинам, на яких він опиняється після рокіровки і штрафи всім іншим. В кінці гри, навпаки, варто заохочувати його перебування в центрі, штрафуючи перебування на домашніх клітинах:

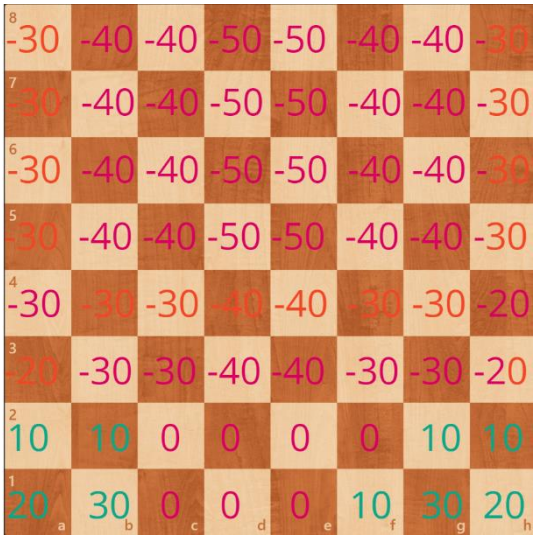


Рисунок 2.17 Оцінки потенціала білого Короля у середені гри.

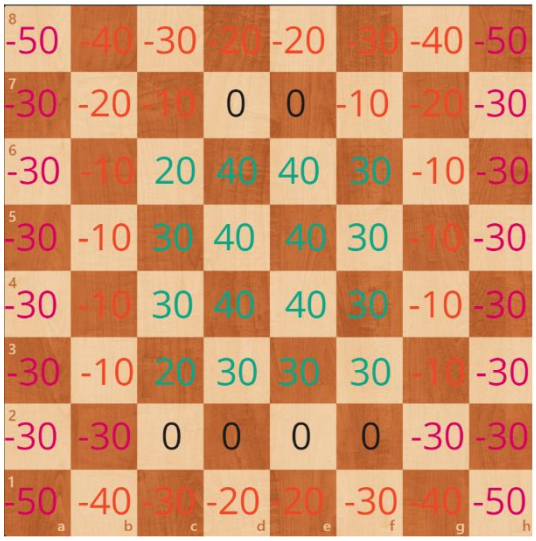


Рисунок 2.18 Оцінки потенціала Короля у кінці гри.

Незважаючи на те, що така проста функція оцінки, яка враховує тільки два спостереження за дошкою - матеріал та обмежену стратегічний потенціал кожної

фігури, вона все ж таки може дозволити програмі грати на досить високому рівні, за умови, що пошук проводиться досить глибоко.

Оціночна функція буде тим сильнішою, чим більше вона здатна зробити спостережень за дошкою. Коротко опишемо деякі з них:

- Мобільність – доступність вільних клітин для пересування. Чим більше клітин може зайняти кожна з фігур, тим більша оцінка.
- Захищеність короля - наскільки Король уразливий до теоретичних атак. На на оцінку можуть вплинути розташування Короля на дошці, його оточення своїми Пішаками та рентгенівські атаки супротивника.
- Структура Пішаків - наскільки добре розташовані Пішаки. Вважають, що якщо пішаки утворюють послідовність зв'язуючи одне одного в єдиний ланцюг, це є перевагою. Якщо пішаки здвоєні – недоліком.

Рисунок 2.19 Приклад гарної Пішачної структури за білих та поганої за чорних.

- Сильні Тури - якщо дві Тури "пов'язані" одна з одною по загальному ряду або лінії, то це вважається перевагою. Так само якщо тура зі свого домашнього ряду стоїть на відкритій вертикалі.
- Сильні Слони - чим менше Пішаків заважають Слону пересуватися по дошці, тим оцінка буде вищою.
- Прохідні Пішаки - якщо є Пішак, на одній вертикалі перед яким відсутні Пішаки супротивника, а на сусідніх вертикалях нема пішаків супротивника

або вони не тримають під боєм поля, через які пішак повинен пройти до поля перетворення, то оцінка такої позиції вища.

2.2.2 Tapered Evaluation

Як вже згадувалося раніше, вартість фігур та їх потенційно "хороші" клітини для знаходження змінюватимуться залежно від фази гри. Техніка, яка дозволяє перераховувати значення пріоритетних полів та вартості фігур у процесі гри, називається Tapered Evaluation. Вона передбачає наявність двох заздалегідь визначених значень бонусів для кожного критерію оцінки, що відповідають початку та кінцю гри. Під час самої гри, спираючись на поточну фазу, що базується на кількості фігур та їх цінності, проводиться інтерполяція між цими значеннями. Це дозволяє гнучко адаптувати стратегію гри до змінюючихся умов, забезпечуючи більш точну оцінку.

Як правило, у шахах виділяють три фази гри – *Початок*, *Мітельшпіль* та *Ендшпіль*. Значення фази визначається кількістю фігур, що залишилися на дошці. Чим більше цінних фігур на дошці, тим ближча фаза до *Початкової*. В процесі гри, за мірою зменшення кількості фігур, значення фази наближається до *Ендшпілю*. Для визначення *Мітельшпілью* використовуються попередньо встановлені порогові значення, які вказують на закінчення *Початкової* фази і початок *Ендшпілю*. Підсумовуючи все вищесказане, отримуємо формулу для підрахунку значення поточної фази гри [17]:

$$PhaseScore = \sum_{color}^{2} \sum_{figure}^{6} E_{\text{матеріал}}(figure, color) \tag{2.3}$$

Порахувавши фазове значення, можемо визначити "масштабоване" значення для кожного критерію по наступній формулі (2.4) [18]:

$$E_{k_{tapered}} = \frac{\left(E_{k_{opening}} * PhaseScore\right) + \left(E_{k_{endgame}} * (K - PhaseScore)\right)}{K} \quad (2.4)$$

де:

$E_{k_{opening}}$ – оцінка критерію k у Початковій фазі

$E_{k_{engame}}$ – оцінка критерію k у фазі Ендшпілю

$PhaseScore$ – значення поточної фази

K – поріг початкової фази гри

2.3 Висновки до розділу 2

У даному розділі було детально розглянуто дві ключові складові будь-якої шахової програми - процес пошуку оптимального ходу та оцінку позиції.

Було продемонстровано, що шахові партії можна представити у вигляді дерева, по якому здійснюється пошук оптимальної серії ходів за допомогою алгоритма *Minimax*. Також розглянуто алгоритм *Alpha-Beta*, який забезпечує отримання тих же результатів, але з меншою кількістю необхідних обчислень. Визначено, що для досягнення найвищої ефективності цього алгоритму потрібні методи сортування ходів. Було згадано про деякі недоліки алгоритмів пошуку та проблеми, які можуть виникати під час їх застосування. В результаті було описано кілька підходів до вирішення цих проблем.

При цьому була розглянута задача оцінки шахової позиції, використовуючи методи, які враховують різні аспекти, такі як матеріальна перевага, розташування фігур на дошці, а також інші фактори, значення яких можуть залежити від поточної фази гри.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Інтерфейс взаємодії з користувачем

Оскільки шахові двигуни за своєю суттю є консольними програмами без власного інтерфейсу, для зручної взаємодії з ними необхідно використовувати стандартизовані протоколи комунікації. Універсальний протокол Universal Chess Interface (UCI) є найбільш популярним та широко використовуваним на сьогоднішній день.

UCI протокол забезпечує зручний та універсальний спосіб обміну даними між шаховим інтерфейсом та програмою. Він передбачає використання текстового вводу та виводу, що дозволяє зручно взаємодіяти з програмою через команди та отримувати від неї відповіді. За допомогою UCI протоколу можна виконувати різноманітні дії з шаховою програмою, такі як встановлення позиції на дошці, виконання ходів, отримання стану гри та отримання найкращого ходу. Крім цього, протокол підтримує інші корисні команди, які спрощують роботу з програмою та дозволяють контролювати її поведінку.

Набагато зручніше використовувати UCI протокол разом із графічними інтерфейсами (GUI) для шахів. Це дозволяє користувачам обирати з різних програм, які надають зручний і естетичний інтерфейс для гри в шахи з використанням шахового двигуна. Такий підхід надає широкий спектр можливостей, включаючи відображення шахової дошки, введення ходів за допомогою миші або клавіатури, відображення варіантів ходів, аналіз партій та багато іншого.

3.2 Встановлення позиції для аналізу

Для роботи з шаховими програмами необхідно якимось чином вказувати позицію, яку потрібно дослідити. Додавання кожної фігури по черзі на пусту дошку було б досить стомлюючим завданням, яке ще й до того забирало б багато часу. Більш ефективним способом є використання нотації FEN.

FEN (Forsyth-Edwards Notation) - нотація, що використовується для опису стану дошки. Вона дозволяє передати всю необхідну інформацію про позицію, включаючи розташування фігур, можливість рокіровки, право ходу, наявність пішаків, які можуть здійснювати хід "взяття на проході" та кількість зроблених ходів. FEN є зручним тим, що дозволяє легко перетворити будь-яку шахову позицію в один рядок тексту. Це спрощує процес відтворення позицій за допомогою комп'ютерів та дозволяє гравцям ділитися ними та перезапускати гру з будь-якої точки, яку вони побажають.

Наприклад, рядок FEN для початкового стану дошки буде виглядати наступним чином:

= ***rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1***

Рисунок 3.1 Початкова позиція та її FEN нотація

Розглянемо рядок FEN більш детально. Рядок складається з шести значень розділених пробілами:

- ***rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR*** *w KQkq - 0 1*

  Перше значення описує розташування фігур на дошці. Буквенні значення відповідають фігурі, розташованій у відповідній клітинці на дошці, а цифри вказують на кількість порожніх клітин.
- *rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR* ***w*** *KQkq - 0 1*

  Друге значення вказує на колір сторони, яка має право ходу.
- *rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w* ***KQkq*** *- 0 1*

Третє значення вказує на можливість рокіровки для кожної зі сторін.

- *rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1*

  Четверте значення зберігає клітину, на яку Пішак зробив подвійний хід, для визначення можливості взяття "на проході".

- *rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1*

  П'яте значення відображає загальну кількість зроблених напівходів.

- *rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1*

  Шосте значення - лічильник, який відстежує кількість ходів без безповоротних змін позиції (взяття фігури або ходу пішки) для застосування правила про 50 ходів.

3.3 Формулювання вимог до програми

Опишемо головні вимоги до програми, звертаючи увагу на різноманітні аспекти, які включають архітектурні рішення та оптимізацію швидкодії:

- Для досягнення максимальної швидкодії розроблюваної програми необхідно використовувати мову програмування C++, з особливим акцентом на оптимізацію та використання низькорівневих структур даних. Перевагу слід менш надійним, але швидкішим стандартним C++ масивам, замість інших структур з бібліотеки std.
- Для представлення шахової позиції в програмі повинно використовуватись бітові структури, що дозволить використовувати швидкі низькорівневі інструкції для процесора для обробки інформації про шахову дошку найшвидшим способом.
- Функціонал програми повинен включати можливість генерацію ходів з використанням різних технік для прискорення цього процесу таких як хешування ковзаючих ходів (Magic BitBoards) для оптимізації генерації можливих ходів у шаховій позиції.
- Програма повинна мати функціонал оцінки шахової позиції та пошуку найкращого ходу з використанням описаних методів у попередньому

розділі. Робота функції оцінки має бути швидкою та достатньо ефективною для сильної гри в шахи. Пошук найкращого ходу повинен враховувати оптимізацію для досягнення кращої продуктивності.

- Комунікація користувача з шаховою програмою повинна відбуватися безпосередньо за допомогою протоколу UCI (Universal Chess Interface). Програма повинна правильно обробляти команди UCI, такі як "uci", "isready", "position", "go" та інші. Користувачу слід надати можливість задавати позицію у форматі нотації FEN для зручного визначення шахової позиції для аналізу.

3.4 Використання сторонніх GUI

Для найбільш комфортного використання шахового двигуна рекомендується використовувати сторонні графічні інтерфейси користувача (GUI), які підтримують протокол UCI. Такі інтерфейси забезпечують зручний та інтуїтивно зрозумілий спосіб взаємодії з шаховим двигуном, дозволяючи користувачам легко встановлювати позиції, виконувати обчислення та аналізувати варіанти гри.

З метою подальшого тестування розробленого шахового двигуна та гри з аналогічними програмами, було вирішено використовувати Arena Chess GUI. Цей GUI є одним із найпопулярніших інтерфейсів для шахових двигунів та надає широкий спектр функціональності, включаючи візуалізацію шахової дошки, управління параметрами обчислень та турнірний режим, що дозволяє автоматизовано проводити серію ігор з іншими двигунами.

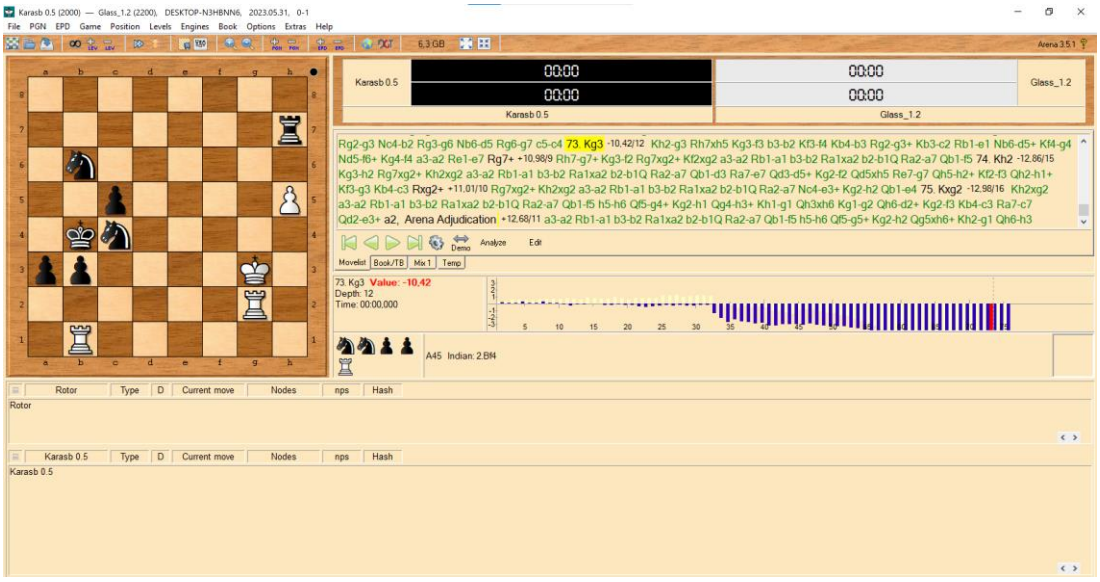


Рисунок 3.2 Arena Chess GUI.

3.5 Результати роботи

Під час тестування програми будуть перевірятися два основних критерії, які в кінцевому підсумку визначають силу і точність гри шахового двигуна. Перший критерій - швидкість перебору позицій, який буде залежати від часу необхідному для генерації легальних ходів, їх оцінки та застосування цих ходів до поточної шахової позиції. В якості одиниці виміру візьмемо кількість досліджених вузлів грального дерева за секунду (NPS).

Другий критерій - кількість пройдених вузлів у гральному дереві для вибору оптимального ходу. Чим менше вузлів потрібно обійти, тим глибше можна проводити пошук за той самий час, що закономірно призводить до збільшення сили гри програми.

Усі наведені нижче результати були отримані використовуючи персональний комп'ютер із процесором Intel® Core™ i7-4702HQ 2.20Ghz.

3.5.1 Тест продуктивності

Для визначення швидкості роботи програми розглянемо дві основні складові, які впливають на її продуктивність - генерацію ходів і функцію оцінки позиції. Суть

наступного тесту полягатиме у переборі всіх можливих позицій на задану глибину, підрахунку їх кількості та фіксації часу який знадобився на ці операції. Буде окремо розглянута швидкість генерації ходів і швидкість оцінки позиції.

Алгоритм підрахунку позицій суттєво нагадуватиме реалізацію алгоритму *Alpha-Beta* з тією відмінністю, що у ньому не будуть проводитись бета-відсікання. Псевдокод наведено на рисунку 3.3.

Algorithm Підрахунок кількості вузлів в ігровому дереві

```
function PerformanceTest(depth)
    if depth == 0 then
        return 1

    GenerateLegalMoves(list)
    for move in list do
        DoMove(move)
        node_count += PerformanceTest(depth − 1)
        UndoMove(move)

    return node_count
```

Рисунок 3.3. Псевдокод алгоритму, який підраховує кількість вузлів в ігровому дереві на задану глибину.

Результати тестування наведено на наступних двох таблицях. Обрана початкова позиція: $rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR\ w\ KQkq\ -\ 0\ 1$.

Глибина	Кількість позицій	Час (в секундах)
4	197,281	0,034
5	4,865,609	0,421
6	119,060,324	8,049

Таблиця 3.1 Результат обходу дерева гри **без** оцінки вузлів.

Глибина	Кількість позицій	Час (в секундах)
4	197,281	0,103
5	4,865,609	1,772
6	119,060,324	38,225

Таблиця 3.2 Результат обходу дерева гри **з** оцінкою вузлів.

Підрахувавши середнє арифметичне отриманих значень з кожної глибини, приходимо до висновку, що розроблена програма може досліджувати близько трьох мільйонів позицій за секунду, що є досить непоганим результатом.

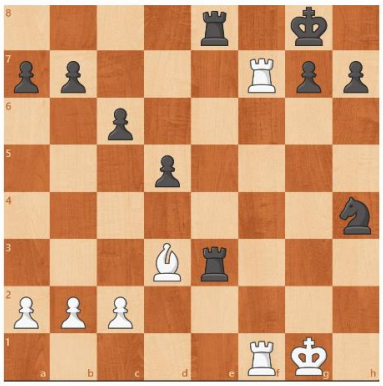
Швидкість обходу дерева без використання функції оцінки досягає більше 14,5 мільйонів позицій за секунду, що свідчить про досить оптимізований підхід до генерації легальних ходів.

3.5.2 Тест ефективності

Для оцінки ефективності роботи евристичних алгоритмів, які зменшують кількість обчислень шляхом скорочення необхідного дерева пошуку, розглянемо ряд позицій на різних етапах гри та порівняємо кількість досліджених вузлів при повному обході дерева без використання евристичних методів та з їх застосуванням. Також обчислимо коефіцієнт розгалуження дерева та порівняємо його з відомим середнім коефіцієнтом розгалуження в шахах.

Для тестування візьмемо кілька позицій зі зіграних раніше партій програмою, зображених на рисунку 3.2.

Позиція №1



4r1k1/pp3Rpp/2p5/3p4/7n/3Br3/PPP5/5RK1 w - - 2 28

Позиція №2

rn1qr1k1/pp4pp/2p5/3pP1n1/6b1/3B1N2/PPPN2PP/R2Q1K1R w - - 2 13

Позиція №3

4rrk1/1bq3pp/p3p1p1/1p1pB1P1/3P3P/P1P5/1P2QP2/R3K2R b KQ - 0 22

Позиція №4

r2qk1nr/pppb1ppp/4p3/2bpP3/5B2/3BP3/PPPN1PPP/R2QK2R b KQkq - 4 8

Рисунок 3.4 Вибрані позиції для тестування

Результати дослідження обраних позицій з використанням тільки стандартного *Alpha-Beta* алгоритму пошуку дерева гри на глибину 6 без використання методів оптимізації наведено в таблиці 3.3:

depth=6	Кількість досліджених вузлів	Коеф. розгалуження
Позиція №1	1,220,769	5
Позиція №2	2,070,715	6
Позиція №3	2,264,206	8
Позиція №4	5,958,607	9

Таблиця 3.3 Результат дослідження позицій **без** використання евристичних методів.

Результати дослідження обраних позицій з використанням різноманітних технік прискорення відсічок у дереві пошуку наведено в таблиці 3.4:

depth=6	Кількість досліджених вузлів	Коеф. розгалуження
Позиція №1	26,309	3
Позиція №2	41,560	2
Позиція №3	17,558	4
Позиція №4	19,102	4

Таблиця 3.4 Результат дослідження позицій **з** використанням евристичних методів.

З даних результатів видно, що для оцінки позиції на глибину 6 без використання реалізованих методів оптимізації в середньому потрібно оцінити 2.8 мільйони позицій. У той же час, в оптимізованому варіанті в середньому потрібно всього 26 тисяч позицій для отримання того ж результату, що є дуже суттєвою різницею. Середнє значення коефіцієнта розгалуження зменшилася удвічі до значення 3. Це означає, що в кожній позиції в середньому достатньо було дослідити лише перші 3 ходи з 30 [10], щоб зробити висновок про найсильніший хід у цій позиції.

3.6 Приклади роботи програми

В якості прикладу роботи програми, розв'яжемо за допомогою неї кілька так званих "шахових етюдів", у яких потрібно знайти мат за задану кількість ходів.

r1k4r/ppp1bq1p/2n1N3/6B1/3p2Q1/8/PPP2PPP/R5K1 w - - 0 1

Рисунок 3.5 Етюд “Мат в 5 ходів”

Результат роботи програми наведено на Рисунку 3.6.

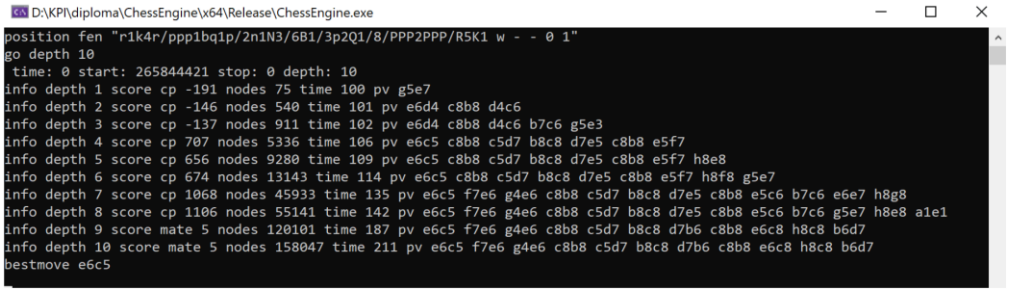


Рисунок 3.6 Output програми.

Розглянемо більш детально, що було зроблено на рисунку 3.6:

1. `position fen "r1k4r/ppp1bq1p/2n1N3/6B1/3p2Q1/8/PPP2PPP/R5K1 w - - 0 1"`

Спочатку користувачем була викликана команда *position* з використанням флагу *fen* та наступним аргументом значення FEN позиції, яку потрібно дослідити.

2. `go depth 10`

Після визначення позиції була використана команда пошуку "go" з аргументом, який вказує глибину, на яку необхідно здійснити пошук.

3.

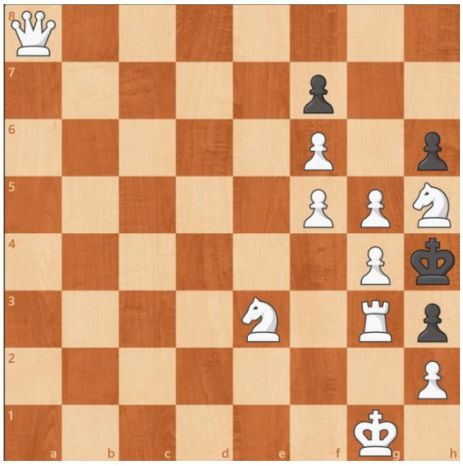
```
time: 0 start: 265844421 stop: 0 depth: 10
info depth 1 score cp -191 nodes 75 time 100 pv g5e7
info depth 2 score cp -146 nodes 540 time 101 pv e6d4 c8b8 d4c6
info depth 3 score cp -137 nodes 911 time 102 pv e6d4 c8b8 d4c6 b7c6 g5e3
info depth 4 score cp 707 nodes 5336 time 106 pv e6c5 c8b8 c5d7 b8c8 d7e5 c8b8 e5f7
info depth 5 score cp 656 nodes 9280 time 109 pv e6c5 c8b8 c5d7 b8c8 d7e5 c8b8 e5f7 h8e8
info depth 6 score cp 674 nodes 13143 time 114 pv e6c5 c8b8 c5d7 b8c8 d7e5 c8b8 e5f7 h8f8 g5e7
info depth 7 score cp 1068 nodes 45933 time 135 pv e6c5 f7e6 g4e6 c8b8 c5d7 b8c8 d7e5 c8b8 e5c6 b7c6 e6e7 h8g8
info depth 8 score cp 1106 nodes 55141 time 142 pv e6c5 f7e6 g4e6 c8b8 c5d7 b8c8 d7e5 c8b8 e5c6 b7c6 g5e7 h8e8 a1e1
info depth 9 score mate 5 nodes 120101 time 187 pv e6c5 f7e6 g4e6 c8b8 c5d7 b8c8 d7b6 c8b8 e6c8 h8c8 b6d7
info depth 10 score mate 5 nodes 158047 time 211 pv e6c5 f7e6 g4e6 c8b8 c5d7 b8c8 d7b6 c8b8 e6c8 h8c8 b6d7
bestmove e6c5
```

Далі було отримано результат від програми. На першому рядку була відображена деяка інформація про умови пошуку: обмеження часу на пошук, час початку пошуку, час, коли пошук має завершитися (за умови, що було задано обмеження на час пошуку) і глибина, на яку виконується пошук. Потім йде інформація, отримана на кожній глибині пошуку, вказуючи оцінку поточної позиції (score) у cp (centipawns, 100 cp = 1 Пішак), кількість пройдених вузлів (nodes), час у мілісекундах (time) і послідовність ходів (pv) в алгебраїчній нотації, які програма вважає найкращим продовженням.

```
info depth 9 score mate 5 nodes 120101 time 187 pv e6c5 f7e6 g4e6 c8b8 c5d7 b8c8 d7b6 c8b8 e6c8 h8c8 b6d7
```

Як ми бачимо на глибині 9, замість оцінки вказується значення *mate 5*, що означає, що мат за п'ять ходів гарантовано, якщо дотримуватися ходів у варіанті, який пропонує програма.

Розглянемо ще один етюд на рисунку 3.7:



Q7/5p2/5P1p/5PPN/6Pk/4N1Rp/7P/6K1 w - - 0 1

Рисунок 3.7 Етюд “Мат в 3 ходи”

```
D:\KPI\diploma\ChessEngine\x64\Release\ChessEngine.exe
position fen "Q7/5p2/5P1p/5PPN/6Pk/4N1Rp/7P/6K1 w - - 0 1"
go depth 6
 time: 0 start: 269652006 stop: 0 depth: 6
info depth 1 score cp 2474 nodes 37 time 83 pv g5h6
info depth 2 score cp 2543 nodes 194 time 84 pv g5g6 f7g6
info depth 3 score cp 2604 nodes 328 time 85 pv g5h6 h4g5 h6h7
info depth 4 score cp 2837 nodes 852 time 86 pv g5h6 h4g5 a8f8 g5h4
info depth 5 score cp 2837 nodes 1037 time 86 pv g5h6 h4g5 a8f8 g5h4 f8f7
info depth 6 score mate 3 nodes 2915 time 88 pv g3h3 h4h3 a8g2 h3h4 g2f2 h4g5 h2h4
bestmove g3h3
```

Рисунок 3.8 Результат роботи програми вирішуючи етюд “Мат в 3 ходи”

Як ми бачимо розроблена програма успішно впоралася з пошуком послідовності ходів, які гарантовано призводять до мату за задану кількість ходів в обох досліджуваних позиціях, що свідчить про її високу працеспроможність та точність у вирішенні шахових завдань.

3.7 Результати зіграних партій із аналогічними програмами

Для визначення рейтингу розробленої програми (Karasb) було зіграно понад чотирьохсот партій із різними відомими програмами з різним рейтингом у проміжку з 2000 до 2550 ЕЛО згідно даних CCRL.

Результати турнірів 1 на 1 з різними супротивниками наведені у наступних таблицях 3.5 – 3.8. Для автоматизованої гри шахових двигунів був використаний Arena Chess GUI.

Engine	Score	Win-Loss-Draw	Новий рейтинг
1. Наш двигун	54,4/100	45-36-19	+53
2. Dragon 4.6	45,5/100	36-45-19	-48

Таблиця 3.5. Результат 100 ігор проти Dragon 4.6 (2450 ЕЛО). Time control – 1 секунда/хід.

Engine	Score	Win-Loss-Draw	Новий рейтинг
1. Наш двигун	69,0/100	64-30-6	+128
2. Dragon 4.6	31,0/100	30-67-6	-114

Таблиця 3.6 Результат 100 ігор проти Dragon 4.6 (2450 ЕЛО). Time control – 10 секунд/хід.

Engine	Score	Win-Loss-Draw	Новий рейтинг
1. AnMon 5.75	73,0/100	67-21-12	+143
2. Наш двигун	27,0/100	21-67-12	-143

Таблиця 3.7. Результат 100 ігор проти AnMon 5.75 (2547 ЕЛО). Time control – 1 секунда/хід.

Engine	Score	Win-Loss-Draw	Новий рейтинг
1. AnMon 5.75	56,5/100	57-31-12	+45
2. Наш двигун	43,5/100	31-57-12	-58

Таблиця 3.8. Результат 100 ігор проти AnMon 5.75 (2547 ЕЛО). Time control – 10 секунд/хід.

Як ми бачимо за результатами 400 зіграних партій, розроблена програма грає десь у проміжку 2400 – 2500 ЕЛО, що еквівалентно приблизно рівню міжнародного майстра.

3.8 Висновки до розділу 3

В даному розділі було досліджено інтерфейс взаємодії з користувачем, метод встановлення позиції для аналізу, формулювання вимог до програми та результати тестування розробленого програмного продукту.

Було встановлено, що інтерфейс взаємодії з користувачем за допомогою UCI забезпечує зручний підхід до роботи з шахматним двигуном. Використання формату FEN дозволяє швидко та точно визначати позицію для аналізу. Формулювання вимог до програми сприяло створенню функціональності, що задовольняє потреби користувачів. Результати тестування і зіграних партій, з аналогічними програмами, показали ефективність розробленого програмного продукту та його конкурентоспроможність.

РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У цьому розділі оцінюються основні характеристики програмного продукту, призначеного для вирішення шахових задач.

Також в даному дослідженні показано різні варіанти реалізації для забезпечення найбільш коректної та оптимальної стратегії вибору, що має вплив на економічні фактори та сумісність з майбутнім програмним продуктом. Для цього застосовувався апарат функціонально-вартісного аналізу.

ФВА передбачає собою технологію, що дозволяє оцінити реальну вартість продукту або послуги незалежно від організаційної структури компанії. ФВА проводиться з метою виявлення резервів зниження витрат за рахунок ефективніших варіантів виробництва, кращого співвідношення між споживчою вартістю виробу та витратами на його виготовлення. Для проведення аналізу використовується економічна, технічна та конструкторська інформація.

Мета ФВА полягає у забезпеченні найбільш оптимального розподілу ресурсів, що виділені на виробництво продукції або надання послуг.

Алгоритм функціонально-вартісного аналізу включає в себе визначення послідовності етапів розробки продукту, визначення повних витрат (річних) та кількості робочих часів, визначення джерел витрат та кінцевий розрахунок вартості програмного продукту.

4.1 Постановка задачі проектування

У роботі застосовується метод ФВА для проведення техніко-економічного аналізу розробки шахового двигуна. Оскільки рішення стосовно проектування та реалізації компонентів, що розробляються, впливають на всю систему, кожна окрема підсистема має задовольняти свої власні функціональні вимоги. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, призначеного для оцінки шахової позиції та пошуку оптимального ходу в ній.

Технічні вимоги до програмного продукту є наступні:

- функціонування на персональних комп’ютерах із стандартним набором компонентів;
- зручність та зрозумілість для користувача;
- швидкість обробки роботи програми;
- можливість зручного масштабування;
- мінімальні витрати на впровадження програмного продукту.

4.2 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка програмного продукту, який вирішує задачу пошуку та вибору оптимального ходу в шаховій позиції. Беручи за основу цю функцію, можна виділити наступні:

F_1 – вибір мови програмування.

F_2 – вибір підхідоду до оцінки позиції.

F_3– вибір середовища програмуванн.

Кожна з цих функцій має декілька варіантів реалізації:

Функція F_1:

а) C++

б) Python

Функція F_2:

а) Традиційний підхід

б) NNUE підхід

Функція F_3:

а) Visual Studio 2019

б) Visual Studio Code

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 4.1).

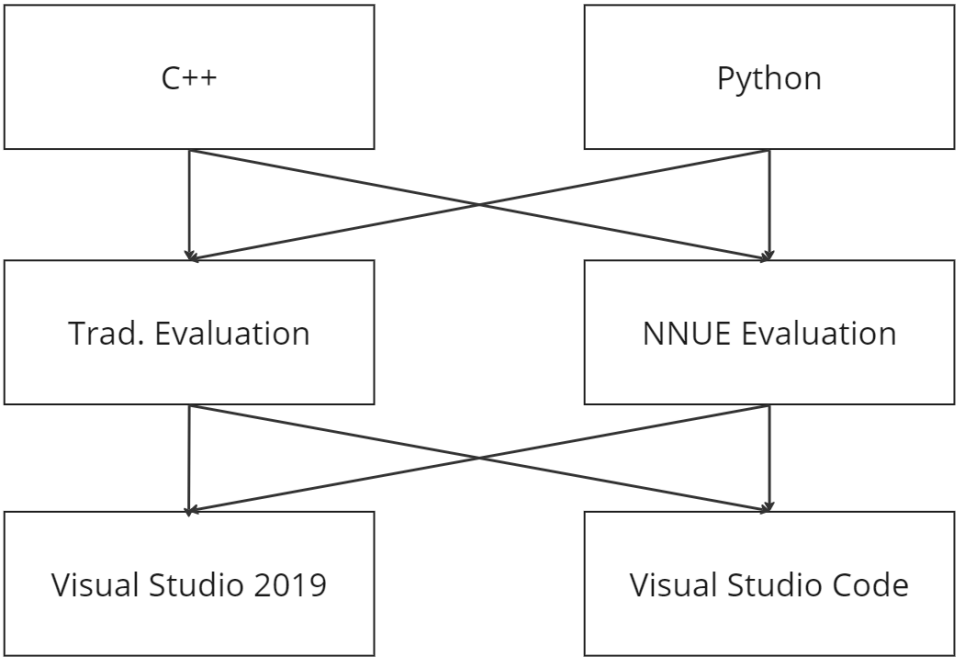


Рисунок 4.1 Морфологічна карта

Морфологічна карта відображає множину всіх можливих варіантиів основних функцій. Позитивно-негативна матриця показана в таблиці 4.1:

Функції	Варіанти реалізації	Переваги	Недоліки
F_1	*А*	Висока швидкодія, доступ до оптимізаційних можливостей процесора.	Складність використання, вимога до вміння ручного керування пам'яттю.
	Б	Зручність, багатофункціональність, широкий вибір бібліотек.	Швидкодія, немає можливості безпосередньо маніпулювати пам'яттю.
F_2	*А*	Швидкодія, відносно проста в реалізації та розумінні.	Схильний до “ефекту горизонта”.
	Б	Гнучкість в адаптації до різних ігрових ситуацій та стратегій без необхідності ручного налаштування функції оцінки.	Необхідність у великих обсягах даних для навчання та високих обчислювальних потужностей для навчання та оцінки позицій.

F_3	*А*	Повноцінне IDE, багаті можливості дебагу.	Підходить переважно для розробки програм під Windows, і може бути менш зручним для розробки для інших платформ.
	Б	Легкий та швидкий редактор коду, що підходить для розробки на різних платформах.	менше можливостей дебагу та профілювання в порівнянні з повноцінною IDE.

Таблиця 4.1 Позитивно-негативна матриця

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція F_1: Перевагу надаємо більш ефективному варіанту з можливістю працювати найшвидшим способом та з більш широкими можливостями роботи з пам'яттю. Для досягнення максимальної швидкодіївариант Б має бути відкинутий.

Функція F_2: Програма допускає обрання обох варіантів. Можливо використати варіанти А чи Б.

Функція F_3: Варіант А є більш сприятливим для задачі, оскільки пропонує ширші можливості для відлагодження програми.

Таким чином, будемо розглядати такий варіанти реалізації ПП:

$$F_1\text{а} - F_2\text{а} - F_3\text{а}$$

$$F_1\text{а} - F_2\text{б} - F_3\text{а}$$

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

4.3 Обґрунтування системи параметрів програмного продукту

На основі даних, розглянутих вище, визначаються основні параметри вибору, які будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

- X1 – швидкодія роботи програми;
- X2 – об’єм пам’яті для обчислень та збереження даних;
- X3 – потенційний об’єм програмного коду.

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію програмного продукту, як показано у таблиці 4.2.

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Швидкодія роботи програми	X1	оп./мс.	7000	11000	15000
Об’єм пам’яті	X2	мб.	256	128	64
Потенційний об’єм програмного коду	X3	кількість рядків коду	10000	5000	3000

Таблиця 4.2 Основні параметри програмного продукту

За даними таблиці 4.2 побудовано графічні характеристики параметрів – рисунок 4.2 – 4.4.

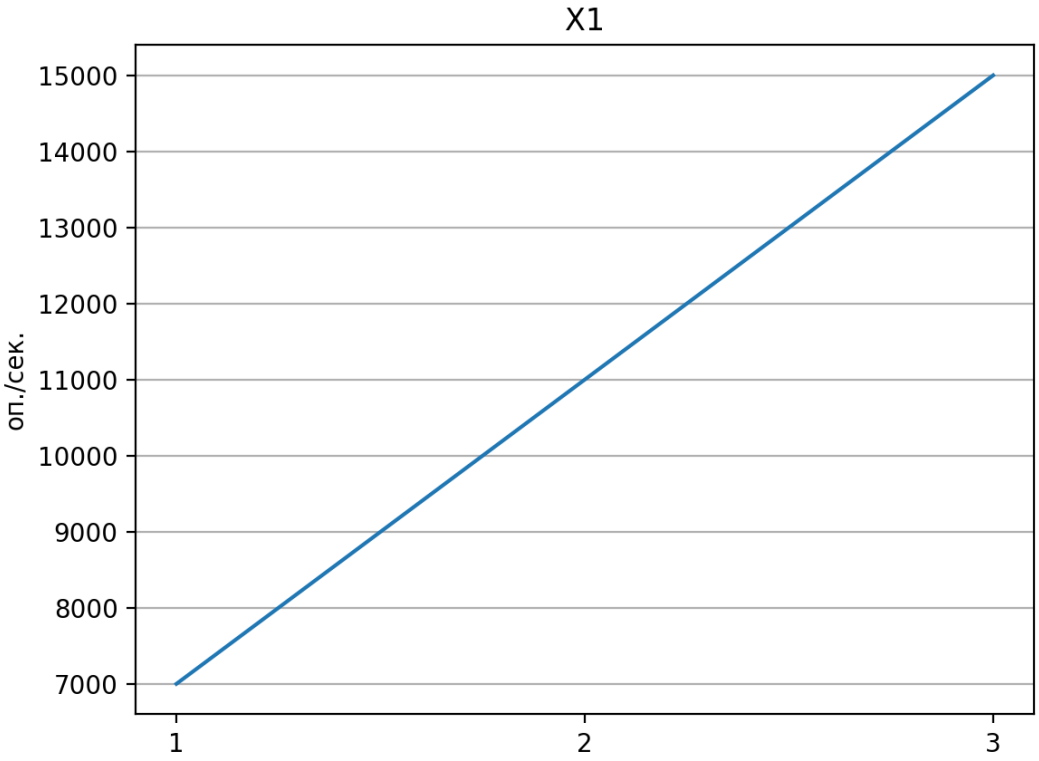


Рисунок 4.2 Графік X1, швидкодія роботи програми.

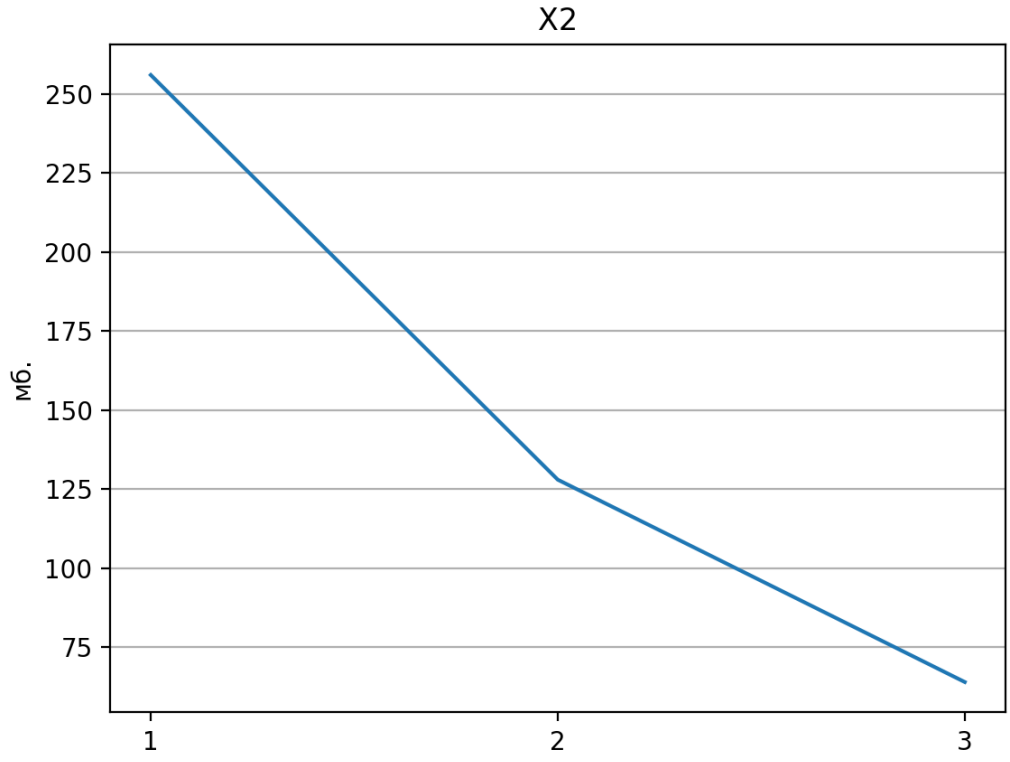


Рисунок 4.3 Графік X2, Об’єм пам’яті.

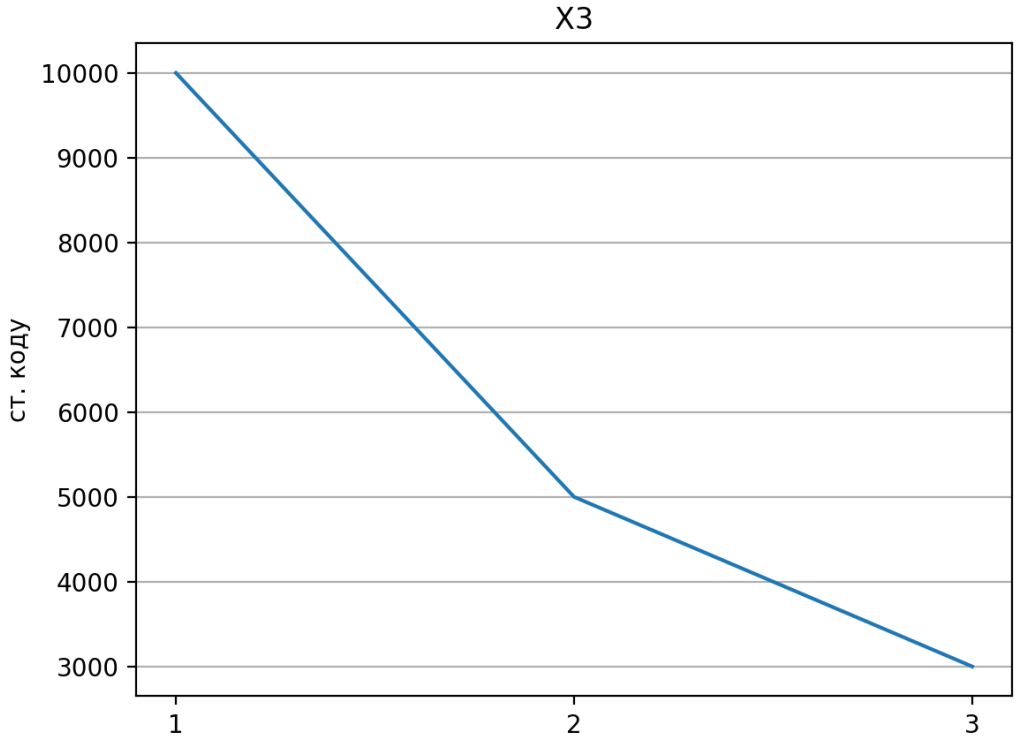


Рисунок 4.4 Графік X3, Потенційний об’єм програмного коду.

4.4 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при знаходженні параметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

– визначення рівня значимості параметра шляхом присвоєння різних рангів;

– перевірку придатності експертних оцінок для подальшого використання;

– визначення оцінки попарного пріоритету параметрів;

– обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 4.3.

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхи-лення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія роботи програми	Оп./мс.	6	4	5	3	4	8	5	35	12	144
X2	Об'єм пам'яті	Mb	3	4	4	3	4	1	2	21	-2	4
X3	Потенційний об'єм програмного коду	Кількість рядків коду	1	2	1	4	2	1	3	14	-9	81
	Разом		10	10	10	10	10	10	10	70	0	229

Таблиця 4.3 Результати ранжування параметрів.

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 70, \tag{4.1}$$

де N – число експертів,

n – кількість параметрів;

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} \cong 23 \tag{4.2}$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T. \tag{4.3}$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 229. \tag{4.4}$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 229}{7^2(3^3 - 3)} = 2{,}34 > W_k = 0{,}67. \tag{4.5}$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 4.4.

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	>	=	>	=	>	>	>	>	1,5
X1 і X3	>	>	>	>	>	>	>	>	1,5
X2 і X3	>	>	>	<	>	=	<	>	1,5

Таблиця 4.4 Попарне порівняння параметрів.

Числове значення, що визначає ступінь переваги *i*–го параметра над *j*–тим, a_{ij} визначається по формулі:

$$a_{ij} = \begin{cases} 1.5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases}. \tag{4.6}$$

З отриманих числових оцінок переваги складемо матрицю А=$\| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості $K_{вi}$ за наступними формулами:

$$K_{\text{вi}} = \frac{b_i}{\sum_{i=1}^{n} b_i} \tag{4.7}$$

$$b_i = \sum_{i=1}^{N} a_{ij} \tag{4.8}$$

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{\text{вi}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}, \tag{4.9}$$

$$b_i' = \sum_{i=1}^{N} a_{ij} b_j \tag{4.10}$$

Параметри x_i				Перша ітер.		Друга ітер.		Третя ітер.	
	X1	X2	X3	b_i	$K_{\text{вi}}$	b_i^1	$K_{\text{вi}}^1$	b_i^2	$K_{\text{вi}}^2$
X1	1	1,5	1,5	4	0,44	11,5	0,5	29,5	0,46
X2	0,5	1	1,5	3	0,33	6	0,26	20	0,31
X3	0,5	0,5	1	2	0,22	5,5	0,23	14,25	0,22
Всього:				9	1	23	1	63,75	1

Таблиця 4.5 - Розрахунок вагомості параметрів

Як видно з таблиці 4.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

4.5 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів *X1* (Швидкодія мови програмування), *X2* (Об'єм пам'яті) та *X3* (потенційний об'єм програмного коду) відповідають технічним вимогам умов функціонування даного ПП.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 4.6):

$$K_K(j) = \sum_{i=1}^{n} K_{вi,j} B_{i,j}\,, \tag{4.11}$$

де n – кількість параметрів;

$K_{вi}$– коефіцієнт вагомості i–го параметра;

B_i – оцінка i–го параметра в балах.

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	А	X1	12000	21	0,46	9,66
F2	А	X2	64	17	0,31	5,27
	Б	X2	256	14	0,31	4,34
F3	А	X4	1000	10	0,22	2,2

Таблиця 4.6 - Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

За даними з таблиці 4.6 за формулою:

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}], \tag{4.12}$$

визначаємо рівень якості кожного з варіантів:

$$K_{K1} = 9{,}66 + 5{,}27 + 2{,}2 = 17{,}13$$
$$K_{K2} = 9{,}66 + 4{,}34 + 2{,}2 = 16{,}2$$

Як видно з розрахунків, кращим є 1 варіант, для якого коефіцієнт технічного рівня має найбільше значення.

4.6 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;

2. Розробка програмної оболонки;

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Загальна трудомісткість обчислюється як:

$$\mathrm{Т_О} = \mathrm{Т_Р} \cdot \mathrm{К_П} \cdot \mathrm{К_{СК}} \cdot \mathrm{К_М} \cdot \mathrm{К_{СТ}} \cdot \mathrm{К_{СТ.М}}, \qquad (4.13)$$

де $\mathrm{Т_Р}$ – трудомісткість розробки ПП;

$\mathrm{К_П}$ – поправочний коефіцієнт;

$\mathrm{К_{СК}}$ – коефіцієнт на складність вхідної інформації;

$\mathrm{К_М}$ – коефіцієнт рівня мови програмування;

$К_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$К_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: $Т_Р$ =31 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $К_П$ = 1.6. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: $К_{СК}$ = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $К_{СТ}$ = 0.8. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$Т_1 = 31 \cdot 1.6 \cdot 0.8 = 39,68 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто $Т_Р$ =25 людино-днів, $К_П$ =1.2, $К_{СК}$ = 1, $К_{СТ}$ =0.9:

$$Т_2 = 25 \cdot 1.2 \cdot 0.9 = 27 \text{ людино-днів.}$$

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$$Т_I = (39,68 + 27 + 5.4 + 27) \cdot 8 = 792,64 \text{ людино-годин.}$$

$$Т_{II} = (39,68 + 27 + 7.22 + 27) \cdot 8 = 807,2 \text{ людино-годин.}$$

Найбільш високу трудомісткість має варіант II.

В розробці один два програмісти з окладом 36200 грн.. Визначимо середню зарплату за годину за формулою:

$$С_{ч} = \frac{М}{T_m \cdot t} \text{ грн.}, \quad (4.14)$$

де М – місячний оклад працівників;

T_m – кількість робочих днів тиждень;

t – кількість робочих годин в день.

$$С_{ч} = \frac{36200 + 36200}{3 \cdot 21 \cdot 8} = 143{,}65 \text{ грн.} \quad (4.15)$$

Тоді, розрахуємо заробітну плату за формулою:

$$С_{ЗП} = С_{ч} \cdot Т_i \cdot К_{Д}, \quad (4.16)$$

де $С_{ч}$– величина погодинної оплати праці програміста;

$Т_i$ – трудомісткість відповідного завдання;

$К_{Д}$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 143{,}65 \cdot 792{,}64 \cdot 1.2 = 136636{,}03$ грн.

II. $С_{ЗП} = 143{,}65 \cdot 807{,}2 \cdot 1.2 = 139145{,}9$ грн.

Відрахування на єдиний соціальний внесок становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 136636{,}03 \cdot 0.22 = 30059{,}92$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 139145{,}9 \cdot 0.22 = 30612{,}09$ грн.

Тепер визначимо витрати на оплату однієї машино-години. ($С_{М}$)

Так як одна ЕОМ обслуговує одного програміста з окладом 36200 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 36200 \cdot 0,2 = 86880 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_Г \cdot (1 + К_З) = 86880 \cdot (1 + 0.2) = 104256 \text{ грн.}$$

Відрахування на соціальний внесок:

$$С_{ВІД} = С_{ЗП} \cdot 0.22 = 104256 \cdot 0,22 = 22936,32 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 27000 грн.

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.2 \cdot 0.25 \cdot 27000 = 8100 \text{ грн.,}$$

де $К_{ТМ}$– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

$К_А$– річна норма амортизації;

$Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.2 \cdot 27000 \cdot 0.05 = 1620 \text{ грн.,}$$

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 8 – 11) \cdot 8 \cdot 0.7 =$$

$$= \quad 1355{,}2 \text{ години},$$

де $Д_К$ – календарна кількість днів у році;

$Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів;

$Д_Р$ – кількість днів планових ремонтів устаткування;

t –кількість робочих годин в день;

$К_В$– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 1355{,}2 \cdot 0{,}5 \cdot 0{,}2 \cdot 4{,}86 = 658{,}62 \text{ грн.},$$

де $N_С$ – середньо-споживча потужність приладу;

$К_З$– коефіцієнтом зайнятості приладу;

$Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} \cdot 0.67 = 27000 \cdot 0{,}67 = 18090 \quad \text{грн.}$$

Тоді, річні експлуатаційні витрати будуть:

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н, \qquad (4.17)$$

$$С_{ЕКС} = 104256 + 22936{,}32 + 8100 + 1620 + 658{,}62 + 18090 = 155660{,}62 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$C_{М\text{-}Г} = C_{ЕКС} / T_{ЕФ} = 132211{,}17 \ / \ 1355{,}2 \ = 97{,}55 \ \text{грн/год.}$$

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$C_М = C_{М-Г} \cdot T, \qquad (4.18)$$

I. $C_М = 97{,}55 \cdot 792{,}64 = 77322{,}03$ грн.

II. $C_М = 97{,}55 \cdot 807{,}2 = 78742{,}36$ грн.

Накладні витрати складають 67% від заробітної плати:

$$C_Н = C_{ЗП} \cdot 0{,}67, \qquad (4.19)$$

I. $C_Н = 136636{,}03 \cdot 0{,}67 = 91546{,}14$ грн.

II. $C_Н = 139145{,}9 \cdot 0{,}67 = 93227{,}75$ грн.

Отже, вартість розробки ПП за варіантами становить:

$$C_{ПП} = C_{ЗП} + C_{ВІД} + C_М + C_Н, \qquad (4.20)$$

I. $C_{ПП} = 91546{,}14 + 30059{,}9 + 77322{,}03 + 91546{,}14 = 290474{,}21$ грн.

II. $C_{ПП} = 93227{,}75 + 30612{,}09 + 78742{,}36 + 93227{,}75 = 295809{,}95$ грн.

4.7 Вибір кращого варіанту ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕР j} = К_{Кj} / С_{Фj}, \quad (4.21)$$

$$К_{ТЕР1} = 13,64 / 290474,21 = 4,6957 \cdot 10^{-5},$$

$$К_{ТЕР2} = 11,28 / 295809,95 = 3.8132 \cdot 10^{-5}.$$

Як бачимо, найбільш ефективним є перший варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1}= 4,6957 \cdot 10^{-5}$,.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є перший варіант реалізації програмного продукту. У нього виявився найкращий показник техніко-економічного рівня якості $К_{ТЕР} = 4,6957 \cdot 10^{-5}$.

4.8 Висновки до розділу 4

В даній частині було проведено повний функціонально-вартісний аналіз програмного продукту. Також було знайдено оцінку основних функцій програмного продукту.

В результаті виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, було визначено та проведено оцінку основних функцій програмного продукту, а також знайдено параметри, які його характеризують.

На основі аналізу вибрано варіант реалізації програмного продукту.

ВИСНОВКИ

У даній роботі було проведено дослідження методів та евристик, які дозволяють комп'ютеру грати у шахи на високому рівні та була розроблена програма здатна успішно обігравати гравців з рейтингом до 2400-2500 ЕЛО.

У першому розділі була розглянута актуальність задачі. Описано основні складові будь-якої шахової програми, а також наведено приклади вже існуючих аналогів. Сформульовано основний принцип пошуку оптимального ходу у шаховій позиції.

У другому розділі були розглянуті конкретні методи вирішення шахових задач, а саме задачі перебору варіантів позицій та задача оцінки позиції. Детально описані алгоритми обходу грального дерева та показані принципи та критерії, за якими комп'ютер може проводити оцінку конкретної позиції.

У третьому розділі увага була зосереджена на тестуванні програми та як їй користуватися. Також наведені результати зіграних партій з іншими аналогічними програмами.

У четвертому розділі було виконано комплексний аналіз функціональності та вартості програмного продукту, а також було надано оцінку основних його функцій. В результаті проведення оцінки були визначені параметри, що характеризують програмний продукт, та обраний варіант його реалізації.

СПИСОК ДЖЕРЕЛ

1. *El Ajedrecista*. Доступно за посиланням: https://en.wikipedia.org/wiki/El_Ajedrecista#:~:text=El%20Ajedrecista%20(English%3A%20The%20Chess,play%20chess%20without%20human%20guidance
2. *Alan Turing. Turochamp*. Доступно за посиланням: https://en.wikipedia.org/wiki/Turochamp
3. *Deep Blue проти Garry Kasparov* Доступно за посиланням: https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov
4. *Using Bitboards for Move Generation in Shogi*. ICGA Journal, Vol. 30, No. 1. Доступно за посиланням: https://www2.teu.ac.jp/gamelab/RESEARCH/ICGAJournal2007.pdf
5. *Рейтинг ЕЛО*. Доступно за посиланням: https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B9%D1%82%D0%B8%D0%BD%D0%B3_%D0%95%D0%BB%D0%BE
6. *Stockfish (chess)*. Доступно за посиланням: https://en.wikipedia.org/wiki/Stockfish_(chess)
7. *Computer Chess Rating List (CCRL).* Доступно за посиланням: https://ccrl.chessdom.com/ccrl/4040/
8. *Komodo (chess)*. Доступно за посиланням: https://en.wikipedia.org/wiki/Komodo_(chess)
9. *Алгоритм Minimax*. Доступно за посиланням: https://en.wikipedia.org/wiki/Minimax
10. *Фактор розгалуження у шахах*. Доступно за посиланням: https://en.wikipedia.org/wiki/Branching_factor
11. *Алгоритм Alpha-Beta*. Доступно за посиланням: https://www.chessprogramming.org/Alpha-Beta
12. *Корнилов Е.Н. Програмування шахів, 2005.*
13. *Jonathan Schaeffer. ICCA Journal paper Speculative Computing, 1987.*
14. *Berliner, Hans J. Some Necessary Conditions for a Master Chess Program, August 20–23, 1973.*

15. *Beal Don. A generalised quiescence search algorithm, April 1990.*
16. *Marc Boulé. An FPGA Move Generator for the Game of Chess, 2002*
17. *Tapered Evaluation*. Доступно за посиланням: http://mediocrechess.blogspot.com/2011/10/guide-tapered-eval.html
18. *Michael Hoffmann. Tapered Evaluation and MSE, January 10, 2021.*

ДОДАТОК А ЛІСТИНГ ПРОГРАМИ

Файл Position.h

```
#pragma once
#ifndef CPOSITION_H_
#define CPOSITION_H_
#include <string>
#include "MoveList.h"
#include "Misc.h"
#include "Transposition.h"

class Position
{
public:
    Position() : Position("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1") {}
    Position(std::string fen);
    friend std::ostream& operator<<(std::ostream& out, const Position& pos);
    void setFEN(const std::string& fen);

    U64 figures(EFigureType type = EFigureType::ALL_FIGURE_TYPE) const;
    U64 figures(EColor color) const;
    U64 figures(EColor color, EFigureType fig_type) const;
    int figure_count(EFigure fig) const;
    EFigure on_square(int sq) const;
    EFigureType on_square_type(int sq) const;
    ESquare get_en_passant() const;
    EColor side_to_move() const;
    U64 get_zobrist() const;
    int get_castling_rights() const;
    bool is_castling_impeded(CastlingRights cr) const;
    bool is_move_legal(Move move) const;
    bool is_king_attacked(EColor color) const;
    bool is_capture_move(Move move) const;
    bool can_castling(CastlingRights cr) const;
    void set_side2move(EColor color);
    void set_en_passant(ESquare sq);
    U64 attack_to(ESquare square) const;
    EGamePhase phase() const;
    void recalc_zobrist();
    int calculate_phase_score() const;
    EGamePhase calculate_phase() const;
    int phase_score() const;

    Position make_move(Move m) const;
    Position make_move(std::string algNotation) const;

    template<EGenType>
    void generate(MoveList&) const;

private:
    void make_castling(EColor color, ESquare from, ESquare to);
    void move_figure(ESquare from, ESquare to);
    void remove_figure(ESquare from);
    void create_figure(ESquare sq, EFigure fig);

private:
    EFigure board[SQ_NB];
    U64 byTypeBB[FIGURE_TYPE_NB];
    U64 byColorBB[COLOR_NB];

    int castlingRights;
    ESquare enPassant;
    EColor sideToMove;
    EGamePhase gp;
    int gpScore;

    U64 zobrist;

    int figureCount[FIGURE_NB];
};

inline EGamePhase Position::phase() const {
    return gp;
}
```

```
inline void Position::recalc_zobrist()
{
    zobrist = Transposition::GetZobristHash(*this);
}

inline int Position::calculate_phase_score() const
{
    using namespace evaluate;
    int score = 0;
    for (int f = KNIGHT; f <= QUEEN; ++f) {
           score +=
            misc::countBits(figures(WHITE, EFigureType(f))) * material_score[OPENNING][f]
            + misc::countBits(figures(BLACK, EFigureType(f))) * material_score[OPENNING][f];
    }
    return score;
}

inline U64 Position::figures(EFigureType type) const {
    return byTypeBB[type];
}

inline U64 Position::figures(EColor c) const {
    return byColorBB[c];
}

inline U64 Position::figures(EColor c, EFigureType fig_type) const {
    return byColorBB[c] & byTypeBB[fig_type];
}

inline int Position::figure_count(EFigure fig) const
{
    return figureCount[fig];
}

inline EFigure Position::on_square(int sq) const {
    return board[sq];
}

inline EFigureType Position::on_square_type(int sq) const {
    return type_of(board[sq]);
}

inline ESquare Position::get_en_passant() const {
    return enPassant;
}

inline EColor Position::side_to_move() const
{
    return sideToMove;
}

inline U64 Position::get_zobrist() const
{
    return zobrist;
}

inline int Position::get_castling_rights() const
{
    return castlingRights;
}

inline bool Position::is_castling_impeded(CastlingRights cr) const {
    return byTypeBB[ALL_FIGURE_TYPE] & castling::pathBlockers[cr];
}

inline bool Position::can_castling(CastlingRights cr) const {
    return castlingRights & cr;
}

inline void Position::set_side2move(EColor color)
{
    sideToMove = color;
}

inline void Position::set_en_passant(ESquare sq)
{
    enPassant = sq;
}
```

```
inline bool Position::is_king_attacked(EColor color) const {
    return attack_to(ESquare(misc::lsb(figures(color, KING)))) & byColorBB[~color];
}

inline bool Position::is_capture_move(Move move) const
{
    return board[READ_TO(move)] != NO_FIGURE;
}

inline void Position::move_figure(ESquare from, ESquare to)
{
    EFigure fig = board[from];
    U64 fromTo = TO_BITBOARD(from) | TO_BITBOARD(to);
    byTypeBB[ALL_FIGURE_TYPE] ^= fromTo;
    byTypeBB[type_of(fig)] ^= fromTo;
    byColorBB[color_of(fig)] ^= fromTo;
    board[from] = NO_FIGURE;
    board[to] = fig;
}

inline void Position::remove_figure(ESquare from)
{
    EFigure fig = board[from];
    U64 fromBB = TO_BITBOARD(from);
    byTypeBB[ALL_FIGURE_TYPE] ^= fromBB;
    byTypeBB[type_of(fig)] ^= fromBB;
    byColorBB[color_of(fig)] ^= fromBB;
    board[from] = NO_FIGURE;
    figureCount[fig]--;

}

inline void Position::create_figure(ESquare sq, EFigure fig)
{
    U64 sqBB = TO_BITBOARD(sq);
    byTypeBB[ALL_FIGURE_TYPE] |= sqBB;
    byTypeBB[type_of(fig)] |= sqBB;
    byColorBB[color_of(fig)] |= sqBB;
    board[sq] = fig;
    figureCount[fig]++;
}

#endif //CPOSITION_H_
```

Файл Position.cpp

```
#include "Position.h"
#include "Misc.h"
#include "Magic.h"
#include "Rays.h"
#include "Transposition.h"
#include <stdexcept>
#include <cassert>
#include <iomanip>


namespace {
    constexpr char figureSymbol[12] = { 'P','N','B','R','Q','K','p','n','b','r','q','k' };

    constexpr int MVV LVA BONUS[7][7] = {
        {0,0,0,0,0,0,0},
        {0,1500,2500,3500,4500,5500,6500},
        {0,1300,2300,3300,4300,5300,6300},
        {0,1100,2100,3100,4100,5100,6100},
        {0,900,1900,2900,3900,4900,5900},
        {0,700,1700,2700,3700,4700,5700},
        {0,500,1500,2500,3500,4500,5500},
    };

    constexpr int promotionBonus = 5500;


    template<EColor color>
    U64 GetPawnAttackBB(U64 pawnsBB) {
        return color == WHITE ?
            misc::shift_bb<NORTH_EAST>(pawnsBB) | misc::shift_bb<NORTH_WEST>(pawnsBB) :
```

```
            misc::shift bb<SOUTH EAST>(pawnsBB) | misc::shift bb<SOUTH WEST>(pawnsBB);
    }

    template<EGenType gentype, EDir dir>
    void MakePromotions(MoveList& list, ESquare to) {
        const ESquare from = ESquare(to - misc::shift value<dir>());
        list.add(EncodeMove<PROMOTION>(from, to, KNIGHT), promotionBonus);
        list.add(EncodeMove<PROMOTION>(from, to, QUEEN), promotionBonus+100);
    }

    template<EColor color, EGenType gentype>
    void GeneratePawnMoves(const Position& pos, MoveList& list)
    {
        using namespace misc;

        constexpr EColor opColor = ~color;
        constexpr EDir upRight = (color == WHITE) ? NORTH EAST : SOUTH WEST;
        constexpr EDir upLeft = (color == WHITE) ? NORTH WEST : SOUTH EAST;
        constexpr EDir up = (color == WHITE) ? NORTH : SOUTH;
        constexpr U64 transformLine = (color == WHITE) ? bitboards::RANK_7 : bitboards::RANK_2;
        constexpr U64 doublePushLine = (color == WHITE) ? bitboards::RANK 4 : bitboards::RANK 5;

        const U64 occ = pos.figures(ALL FIGURE TYPE);
        const U64 pawns bb = pos.figures(color, PAWN);
        const U64 transformPawns = pawns_bb & transformLine;

        //Capture Moves
        if constexpr (gentype != EGenType::QUIETS) {
            //Right pawns attack
            U64 rightHook = shift bb<upRight>(pawns bb & ~transformLine);
            rightHook &= pos.figures(opColor);
            while (rightHook) {
                ESquare to = pop lsb(rightHook);
                ESquare from = ESquare(to - shift value<upRight>());
                list.add(EncodeMove<NORMAL>(from, to, PAWN),
MVV LVA BONUS[PAWN][type of(pos.on square(to))]);
            }

            //Left pawns attack
            U64 leftHook = shift bb<upLeft>(pawns bb & ~transformLine);
            leftHook &= pos.figures(opColor);
            while (leftHook) {
                ESquare to = pop_lsb(leftHook);
                ESquare from = ESquare(to - shift value<upLeft>());
                list.add(EncodeMove<NORMAL>(from, to, PAWN),
MVV LVA BONUS[PAWN][type of(pos.on square(to))]);
            }

            //EnPassant
            constexpr U64 epLine = (color == WHITE) ? bitboards::RANK 5 : bitboards::RANK 4;
            if (pos.get en passant() != SQ NONE) {
                const U64 epBB = TO BITBOARD(pos.get en passant());
                U64 epCandidates =
                    pawns_bb & (shift_bb<EAST>(epBB) | shift_bb<WEST>(epBB));

                ESquare to = ESquare(lsb(shift bb<up>(epBB)));
                while (epCandidates) {
                    ESquare from = pop lsb(epCandidates);
                    list.add(EncodeMove<EN_PASSANT>(from, ESquare(lsb(shift_bb<up>(epBB))), PAWN),
MVV_LVA_BONUS[PAWN][PAWN]);
                }
            }

            // Capture promotions
            if (transformPawns) {
                U64 prom2 = shift bb<upRight>(transformPawns) & pos.figures(opColor);
                U64 prom3 = shift bb<upLeft>(transformPawns) & pos.figures(opColor);
                while (prom2) {
                    MakePromotions<gentype, upRight>(list, pop lsb(prom2));
                }
                while (prom3) {
                    MakePromotions<gentype, upLeft>(list, pop lsb(prom3));
                }
            }
        }

        //Quiet Moves
        if constexpr (gentype != CAPTURES) {
            //One-Push Moves
            U64 onePushBoard = shift_bb<up>(pawns_bb & ~transformLine);
```

```
            onePushBoard &= ~occ;

            U64 doublePushBoard = onePushBoard;
            while (onePushBoard) {
                ESquare to = pop lsb(onePushBoard);
                ESquare from = ESquare(to - shift value<up>());
                list.add(EncodeMove<NORMAL>(from, to, PAWN));
            }

            //Double-Push Moves
            doublePushBoard = shift bb<up>(doublePushBoard);
            doublePushBoard &= doublePushLine & ~occ;

            while (doublePushBoard) {
                ESquare to = pop lsb(doublePushBoard);
                ESquare from = ESquare(to - 2 * shift value<up>());
                list.add(EncodeMove<DOUBLE PUSH>(from, to, PAWN));
            }

            // Quiet promotions
            if (transformPawns) {
                U64 prom1 = shift bb<up>(transformPawns) & ~occ;
                while (prom1) {
                    MakePromotions<gentype, up>(list, pop lsb(prom1));
                }
            }
        }
    }

    template<EColor color, EFigureType fig type>
    void GenerateMove(const Position& pos, MoveList& list, U64 target_squares) {
        assert(fig_type != PAWN);

        U64 figuresBB = pos.figures(color, fig type);

        while (figuresBB) {
            ESquare from = misc::pop_lsb(figuresBB);
            U64 attacks = GetAttackBB<fig type>(from, pos.figures(EFigureType::ALL FIGURE TYPE)) &
target squares;
            while (attacks) {
                ESquare to = misc::pop lsb(attacks);
                list.add(EncodeMove<NORMAL>(from, to, fig_type),
MVV_LVA_BONUS[fig_type][type_of(pos.on_square(to))]);
            }
        }
    }

    template<EColor color, EGenType gentype>
    void GenerateAll(const Position& pos, MoveList& list) {

        constexpr EColor opColor = ~color;

        //Mask to use with attack bitboard of each figure
        U64 targetSq = gentype == QUIETS ? ~pos.figures(opColor) :
            gentype == CAPTURES ? pos.figures(opColor) :
            ~pos.figures(color) | pos.figures(opColor);

        GeneratePawnMoves<color, gentype>(pos, list);
        GenerateMove<color, KNIGHT>(pos, list, targetSq);
        GenerateMove<color, BISHOP>(pos, list, targetSq);
        GenerateMove<color, ROOK>(pos, list, targetSq);
        GenerateMove<color, QUEEN>(pos, list, targetSq);
        GenerateMove<color, KING>(pos, list, targetSq);

        //castling
        if constexpr (gentype != CAPTURES) {
            for (CastlingRights cr : castling::byColor[color]) {
                bool rookExist = TO BITBOARD(castling::rookSquare[cr]) & pos.figures(color, ROOK);
                if (pos.can castling(cr) && !pos.is castling impeded(cr) && rookExist) {
                    ESquare ksq = color == WHITE ? E1 : E8;

                    if (!(pos.attack to(ksq) & pos.figures(opColor))) {
                        if (!(pos.attack to(castling::kingSafetySq[cr]) & pos.figures(opColor))) {
                            list.add(EncodeMove<CASTLING>(ksq, castling::rookSquare[cr], KING));
                        }
                    }
                }
            }
        }
    }
```

```
}

Position::Position(std::string fen)
{
    this->setFEN(fen);
}

EGamePhase Position::calculate phase() const
{
    int gamePhaseScore = calculate phase score();

    EGamePhase gp =
        gamePhaseScore > evaluate::opening bounder score ? EGamePhase::OPENNING :
        gamePhaseScore < evaluate::endgame_bounder_score ? EGamePhase::END_GAME :
        EGamePhase::MIDDLE GAME;

    return gp;
}

int Position::phase_score() const
{
    return gpScore;
}

void Position::setFEN(const std::string& fen)
{
    //attackedSquares = 0;
    enPassant = ESquare::SQ NONE;

    //Clear representation of the board
    for (int fig = W_PAWN; fig <= B_KING; ++fig) {
        figureCount[fig] = 0;
    }
    for (int color = 0; color < COLOR NB; ++color) {
        byColorBB[color] = 0ULL;
    }
    for (int figType = 0; figType < FIGURE_TYPE_NB; ++figType) {
        byTypeBB[figType] = 0ULL;
    }
    for (int i = 0; i < 64; ++i) {
        board[i] = NO FIGURE;
    }

    int squareInt = ESquare::A8;
    int cur = 0;
    for (; fen[cur] != ' '; ++cur) {
        if ((int)fen[cur] >= '1' && (int)fen[cur] <= '8') {
            squareInt += fen[cur] - '0';
        }
        else if (fen[cur] == '/') {
            squareInt -= 16;
        }
        else
        {
            EFigureType type = NO_FIGURE_TYPE;
            EColor color = WHITE;
            switch (fen[cur])
            {
            case 'K':
                type = KING;
                break;
            case 'k':
                type = KING;
                color = BLACK;
                break;
            case 'Q':
                type = QUEEN;
                break;
            case 'q':
                type = QUEEN;
                color = BLACK;
                break;
            case 'B':
                type = BISHOP;
                break;
            case 'b':
                type = BISHOP;
                color = BLACK;
                break;
            case 'N':
```

```
                type = KNIGHT;
                break;
            case 'n':
                type = KNIGHT;
                color = BLACK;
                break;
            case 'P':
                type = PAWN;
                break;
            case 'p':
                type = PAWN;
                color = BLACK;
                break;
            case 'R':
                type = ROOK;
                break;
            case 'r':
                type = ROOK;
                color = BLACK;
                break;
            }

            board[squareInt] = add color(color, type);
            byTypeBB[type] |= TO BITBOARD(squareInt);
            byTypeBB[ALL_FIGURE_TYPE] |= TO_BITBOARD(squareInt);
            byColorBB[color] |= TO_BITBOARD(squareInt);
            figureCount[board[squareInt]]++;
            squareInt++;
        }
    }

    // w or b
    if (++cur < fen.size()) {
        char side = fen[cur++];
        if (side == 'w') {
            sideToMove = EColor::WHITE;
        }
        else if (side == 'b') {
            sideToMove = EColor::BLACK;
        }
        else {
            throw std::invalid_argument("Invalid FEN");
        }
    }

    castlingRights = 0;
    for (cur++; fen[cur] != ' ' && cur < fen.size(); ++cur) {
        switch (fen[cur]) {
        case 'K':
            castlingRights |= WHITE 00;
            break;
        case 'k':
            castlingRights |= BLACK_00;
            break;
        case 'Q':
            castlingRights |= WHITE 000;
            break;
        case 'q':
            castlingRights |= BLACK_000;
            break;
        }
    }

    char epFile = fen[++cur];
    if (epFile != '-') {
        char epRank = fen[++cur];
        enPassant = ESquare(8 * (epRank - '1') + epFile - 'a');
        std::cout << "";
    }

    gp = calculate phase();
    gpScore = calculate phase score();

    zobrist = Transposition::GetZobristHash(*this);
}

std::ostream& operator<<(std::ostream& out, const Position& pos)
{
    out << "+---+---+---+---+---+---+---+---+\n";
    for (int sq = A8; sq >= 0; ++sq) {
```

```
        EFigure fig = pos.board[sq];
        char symbol = (fig == NO_FIGURE) ? ' ' : figureSymbol[(color_of(fig) == WHITE) ? type_of(fig) - 1
: type_of(fig) + 5];
        out << "| " << symbol << " ";

        if ((sq + 1) % 8 == 0) {
            out << "| " << (sq / 8) + 1 << std::endl;
            out << "+---+---+---+---+---+---+---+---+\n";
            sq -= 16;
        }
    }
    out << "  a   b   c   d   e   f   g   h\n";

    out << std::endl;

    out << std::setw(15) << "Side: " << std::setw(5) << misc::ToString(pos.side to move()) << std::endl;
    out << std::setw(15) << "Enpassant: " << std::setw(5) << misc::ToString(pos.get en passant()) <<
std::endl;

    std::string castling(4, ' ');
    int cr = pos.get castling rights();
    castling[0] = (cr & WHITE 00) ? 'K' : '-';
    castling[1] = (cr & WHITE 000) ? 'Q' : '-';
    castling[2] = (cr & BLACK_00) ? 'k' : '-';
    castling[3] = (cr & BLACK_000) ? 'q' : '-';

    out << std::setw(15) << "Castling: " << std::setw(5) << castling << std::endl << std::endl;
    return out;
}

U64 Position::attack_to(ESquare square) const {
    return
        (GetPawnAttackBB<WHITE>(TO BITBOARD(square)) & figures(BLACK, PAWN)) |
        (GetPawnAttackBB<BLACK>(TO BITBOARD(square)) & figures(WHITE, PAWN)) |
        (GetAttackBB<KNIGHT>(square) & byTypeBB[KNIGHT]) |
        (GetAttackBB<BISHOP>(square, byTypeBB[ALL_FIGURE_TYPE]) & (byTypeBB[BISHOP] | byTypeBB[QUEEN])) |
        (GetAttackBB<ROOK>(square, byTypeBB[ALL FIGURE TYPE]) & (byTypeBB[ROOK] | byTypeBB[QUEEN])) |
        (GetAttackBB<KING>(square) & byTypeBB[KING]);
}

bool Position::is_move_legal(Move move) const
{
    Position tempPos = make move(move);
    return !tempPos.is king attacked(sideToMove);
}

Position Position::make_move(Move m) const
{
    ESquare from = ESquare(READ FROM(m));
    ESquare to = ESquare(READ TO(m));
    EFigureType fig = EFigureType(READ FIGURE(m));
    EMoveType move_type = EMoveType(READ_MOVE_TYPE(m));

    EColor color = sideToMove;
    EColor opColor = ~color;
    EFigure capture = move type == CASTLING ? NO FIGURE : on square(to);

    Position newPos(*this);
    newPos.enPassant = SQ_NONE;
    newPos.sideToMove = opColor;

    if (move type == CASTLING) {
        newPos.make_castling(color, from, to);
    }
    else if (move type == DOUBLE PUSH) {
        newPos.enPassant = to;
        newPos.move figure(from, to);
    }

    int gphase = newPos.gpScore;
    if (capture != NO FIGURE) {
        newPos.remove figure(to);

        //Update game-phase if needed
        if (type_of(capture) >= KNIGHT && type_of(capture) <= QUEEN) {
            newPos.gpScore -= evaluate::material score[OPENNING][type of(capture)];
            assert(newPos.calculate phase score() == newPos.gpScore);
        }
    }
```

```
    if (move type == NORMAL) {
        newPos.move_figure(from, to);

        if (fig == KING) {
            //newPos.castling rights &= ~(3 << (color * 2));
            newPos.castlingRights &= color == WHITE ? ~(WHITE 00 | WHITE 000) : ~(BLACK 00 | BLACK 000);
        }

        if (fig == ROOK) {
            if (from == castling::rookFromTo00[color][0]) {
                newPos.castlingRights &= color == WHITE ? ~WHITE 00 : ~BLACK 00;
            }
            else if (from == castling::rookFromTo000[color][0]) {
                newPos.castlingRights &= color == WHITE ? ~WHITE 000 : ~BLACK 000;
            }
        }
    }
    else if (move type == PROMOTION) {
        newPos.remove_figure(from);
        newPos.create figure(to, add color(color, fig));

        //Update game phase score
        newPos.gpScore += evaluate::material score[OPENNING][fig];
    }
    else if (move_type == EN_PASSANT) {
        newPos.remove figure(ESquare(color == WHITE ? to - 8 : to + 8));
        newPos.move figure(from, to);
    }

    assert(newPos.calculate_phase_score() == newPos.gpScore);

    //Set new game phase
    newPos.gp =
        newPos.gpScore > evaluate::opening bounder score ? EGamePhase::OPENNING :
        newPos.gpScore < evaluate::endgame bounder score ? EGamePhase::END GAME :
        EGamePhase::MIDDLE_GAME;

    Transposition::AmmendHash(newPos.zobrist, m, *this);

    return newPos;
}

void Position::make castling(EColor color, ESquare from, ESquare to)
{
    ESquare rf = castling::rookFromTo00[color][0];
    ESquare rto = castling::rookFromTo00[color][1];
    ESquare kto = castling::kingTo00[color];

    //if long castling
    if (from > to) {
        rf = castling::rookFromTo000[color][0];
        rto = castling::rookFromTo000[color][1];
        kto = castling::kingTo000[color];
    }

    //move king
    move figure(from, kto);

    //move rook
    move figure(rf, rto);

    //delete rights for castling
    castlingRights &= color == WHITE ? ~(WHITE_00 | WHITE_000) : ~(BLACK_00 | BLACK_000);
}

template<EGenType gentype>
void Position::generate(MoveList& list) const {

    sideToMove == WHITE ?
        GenerateAll<WHITE, gentype>(*this, list) :
        GenerateAll<BLACK, gentype>(*this, list);
}

template<>
void Position::generate<LEGAL>(MoveList& legalMoves) const {
    MoveList pseudoMoves;
    generate<PSEUDO>(pseudoMoves);

    for (int i = 0; i < pseudoMoves.size(); ++i) {
```

```
        if (EMoveType(READ_MOVE_TYPE(pseudoMoves[i]) == CASTLING)) {
            std::cout << "";
        }
        if (is_move_legal(pseudoMoves[i])) {
            legalMoves.add(pseudoMoves[i], pseudoMoves.get_score(i));
        }
    }
}

Position Position::make_move(std::string algNotation) const
{
    MoveList list;
    generate<LEGAL>(list);

    for (int i = 0; i < list.size(); ++i) {
        if (algNotation == misc::ToString(list[i])) {
            return make_move(list[i]);
        }
    }

    std::cout << "move " << algNotation << " not found\n";
    throw std::invalid_argument("There is no [" + algNotation + "] move in current position");
}

template void Position::generate<PSEUDO>(MoveList&) const;
template void Position::generate<CAPTURES>(MoveList&) const;
template void Position::generate<QUIETS>(MoveList&) const;
```

Файл Search.h

```
#pragma once
#ifndef SEARCH_H_
#define SEARCH_H_
#include "Constants.h"
#include <vector>
#include <iostream>

class Position;
namespace search {
    struct s_search {
        friend std::ostream& operator<<(std::ostream& out, const s_search& s);
        s_search() {}
        s_search(const std::vector<Move>& pv, long long time, U64 nodes, int score) :
            pv(pv),
            time(time),
            nNodes(nodes),
            score(score)
        {}
        std::vector<Move> pv;
        long long time;
        U64 nNodes;
        int score;
    };

    s_search search(int depth, const Position& pos);
    s_search iterative_deepening(int depth, const Position& pos);


    U64 perft(const Position& pos, int depth, PerftStat* stat = nullptr);
}

#endif // SEARCH_H_
```

Файл Search.cpp

```
#include "Search.h"
#include "Position.h"
#include "Evaluate.h"
#include "Transposition.h"
#include "UCI.h"
#include <iostream>
#include <memory>

U64 repetitionTable[1000];
int repetitionIndex;

namespace {
    struct Stats {
        Stats() :
```

```
        nNodes(0),
        killers(),
        hist(),
        counterHist(),
        table(new Transposition(256)) {}

    U64 nNodes;
    KillerHeuristic killers;
    HistoryHeuristic hist;
    CounterMoveHeuristic counterHist;
    std::unique_ptr<Transposition> table;
} ss;

int pvLength[64];
Move pvTable[64][64];

bool followPv = false;
bool scorePv = false;

void EnablePvScoring(const MoveList& list, int ply) {
    followPv = false;

    for (int i = 0; i < list.size(); ++i) {
        //make sure we hit pv line
        if (pvTable[0][ply] == list[i]) {
            followPv = true;
            scorePv = true;
            return;
        }
    }
}

//UCI time control
bool isStopped = false;

void communicate() {
    if (UCI::flags.timeset && UCI::get_time_ms() > UCI::flags.stoptime) {
        isStopped = true;
    }
}

//Quiescence Search
int quies(int alpha, int beta, const Position& pos) {
    if ((ss.nNodes & 2047) == 0) {
        communicate();
    }

    int eval = evaluate::eval(pos, pos.side_to_move());
    if (eval >= beta) {
        return beta;
    }
    if (eval > alpha) {
        alpha = eval;
    }

    //generate only captures
    MoveList forcedMoves;
    pos.generate<CAPTURES>(forcedMoves);

    for (int i = 0; i < forcedMoves.size(); ++i) {
        forcedMoves.pick(i);

        Position newPos = pos.make move(forcedMoves[i]);

        if (newPos.is king attacked(pos.side to move())) {
            continue;
        }

        ss.nNodes++;
        eval = -quies(-beta, -alpha, newPos);

        if (eval >= beta) {
            return beta;
        }
        if (eval > alpha) {
            alpha = eval;
        }
    }

    return alpha;
```

```
}

bool IsRepetition(const Position& currentPos) {
    for (int i = 0; i < repetitionIndex; ++i) {
        if (repetitionTable[i] == currentPos.get zobrist()) {
            return true;
        }
    }
    return false;
}

//std::vector<U64> bfArr;
Move prevMove = 0;
int alpha_beta(int depth, int ply, int alpha, int beta, const Position& pos, bool doNull = true) {

    pvLength[ply] = ply;

    Move bestMove = 0;

    EHashFlag hashf = HASH_ALPHA;

    //get information about current position from the transposition table
    const s node* const cache = ss.table->get(pos.get zobrist());

    //3 fold repetition rule
    //check if current position has already been before in the current search tree
    if (ply && IsRepetition(pos)) {
        return 0;
    }

    //check if currently in pv node
    int pv_node = beta - alpha > 1;

    if (ply && cache->flag != HASH EMPTY && cache->depth >= depth && pv node == 0) {
        int score = cache->score;

        if (cache->flag == HASH_EXACT) {
            return score;
        }
        if (cache->flag == HASH ALPHA && score <= alpha) {
            return alpha;
        }
        if (cache->flag == HASH_BETA && score >= beta) {
            return beta;
        }
    }

    if ((ss.nNodes & 2047) == 0) {
        communicate();
    }

    if (depth == 0) {
        return quies(alpha, beta, pos);
    }

    bool inCheck = pos.is king attacked(pos.side to move());

    //if the king is under attack, search one depth deeper
    if (inCheck) {
        depth++;
    }

    //Null-move heuristic
    //The idea is to give the opponent a free shot,
    //and if current position is still so good that exceed beta,
    //we assume that we'd also exceed beta if we went and searched all of the moves.
    if (!followPv && depth >= 3 && ply && !inCheck) {
        Position nullPos(pos);

        //delete en passant
        nullPos.set en passant(SQ NONE);

        //literally give the opponent a second move in a row
        nullPos.set side2move(~pos.side to move());

        //update hash
        nullPos.recalc zobrist();

        //do search with reduced depth
        int score = -alpha_beta(depth - 1 - 2, ply + 1, -beta, -beta + 1, nullPos, doNull);
```

```
    if (isStopped) return 0;

    //if even after two moves of opponent our score
    //is still >= beta, then make cut off
    if (score >= beta) {
        return beta;
    }
}

//Razoring
if (!pv node && !inCheck && depth <= 3) {
    //static evaluation + bonus
    int score = evaluate::eval(pos, pos.side_to_move()) + 125;

    int newScore;
    //Fail low
    //This position was not good enough for us
    //even after adding the bonus value to the score
    if (score < beta) {
        if (depth == 1) {
            newScore = quies(alpha, beta, pos);
            //max(score, newScore)
            return score > newScore ? score : newScore;
        }
    }

    //Add second bonus
    score += 175;

    if (score < beta && depth <= 2) {
        newScore = quies(alpha, beta, pos);
        if (newScore < beta) {
            //max(score, newScore)
            return score > newScore ? score : newScore;
        }
    }
}

MoveList moves;
pos.generate<PSEUDO>(moves);

//check if we are following the pv line and
//if we should put pv move at the top of the list
if (followPv) {
    EnablePvScoring(moves, ply);
}
Move pvMove = 0;
if (scorePv) {
    pvMove = pvTable[0][ply];
    scorePv = false;
}

//updating the scores for the moves
moves.sort(ss.killers,
    ss.hist,
    ss.counterHist,
    cache->bestMove,
    prevMove,
    pvMove,
    ply,
    pos.side to move());

int nLegal = 0;
for (int i = 0; i < moves.size(); ++i) {
    //lift the best move to the list[i]
    moves.pick(i);

    Position newPos = pos.make move(moves[i]);

    //make sure move is legal
    if (newPos.is king attacked(pos.side to move())) {
        continue;
    }

    //save hash of the current position to determine 3fold repetition later.
    repetitionTable[repetitionIndex++] = pos.get zobrist();

    ss.nNodes++;
    nLegal++;
```

```
            prevMove = moves[i];

            //Full window search for the first move in the move-list
            int score;
            if (nLegal < 2) {
                score = -alpha_beta(depth - 1, ply + 1, -beta, -alpha, newPos, doNull);
            }
            else {
                //Late move reductions. here we rely on the fact we have a good enough sorting of
                //moves to be sure that the first 4 moves will most often be the best, so we count
                //all other moves at a reduced depth and a smaller window to try speed up the search

                //Conditions to LMR
                if (nLegal >= 4 && depth >= 3 && !inCheck
                    && !pos.is_capture_move(moves[i]) && EMoveType(READ_MOVE_TYPE(moves[i])) != PROMOTION
                    && !newPos.is_king_attacked(newPos.side_to_move())) {
                    int r = nLegal <= 6 ?
                        newPos.phase_score() < 2500 ? 2 : 1
                        : depth / 3;

                    score = -alpha_beta(depth - 1 - r, ply + 1, -alpha - 1, -alpha, newPos, doNull);
                }
                else {
                    score = alpha + 1;
                }

                //Principal Variation Search.
                //The the technique makes the assumption that if
                //you find one PV move when you are searching a node, you definitely have a PV node.
                //It again assumes that move ordering will be good enough that we won't find a better PV
move
                //Once we've found a move with a score that is between alpha and beta, the rest of the
moves
                //are searched with the goal of proving that they are all bad
                if (score > alpha)
                {
                    score = -alpha_beta(depth - 1, ply + 1, -alpha - 1, -alpha, newPos, doNull);
                    //Check for failure, re-search full window and full depth.
                    if ((score > alpha) && (score < beta)) {
                        score = -alpha_beta(depth - 1, ply + 1, -beta, -alpha, newPos, doNull);
                    }
                }
            }

            //int score = score = -alpha_beta(depth - 1, ply + 1, -beta, -alpha, newPos, doNull);

            repetitionIndex--;

            if (isStopped) return 0;

            //current move improves score
            if (score > alpha) {
                alpha = score;

                hashf = HASH_EXACT;
                bestMove = moves[i];

                //History heuristic
                if (!pos.is_capture_move(bestMove)) {
                    ss.hist[pos.side_to_move()][READ_FROM(bestMove)][READ_TO(bestMove)] += depth;
                }

                //Reset triangular PV table with new best move
                pvTable[ply][ply] = bestMove;
                for (int nextPly = ply + 1; nextPly < pvLength[ply + 1]; ++nextPly) {
                    pvTable[ply][nextPly] = pvTable[ply + 1][nextPly];
                }
                pvLength[ply] = pvLength[ply + 1];
            }

            //Fails High
            if (score >= beta) {
                if (!pos.is_capture_move(moves[i])) {
                    //Save move for Hiller Heuristic
                    ss.killers[1][ply] = ss.killers[0][ply];
                    ss.killers[0][ply] = moves[i];

                    //History Heuristic
                    ss.hist[pos.side_to_move()][READ_FROM(moves[i])][READ_TO(moves[i])] = 1 << depth;
```

```
                    //Countermove Heuristic
                    ss.counterHist[READ FROM(prevMove)][READ TO(prevMove)] = moves[i]; //!
                }

                //save position to hash table
                ss.table->add(pos.get zobrist(), bestMove, depth, beta, HASH BETA);

                //bfArr.push back(nLegal);

                return beta;
            }
        }

        //if there were no legal moves, check for mate or stalemate
        if (nLegal == 0) {
            return inCheck ? -figure::KING COST + ply : 0;
        }

        //bfArr.push back(nLegal);

        //save position to hash table
        ss.table->add(pos.get zobrist(), bestMove, depth, alpha, hashf);

        //Fails low
        //meaning that none of the moves in here will be any good
        return alpha;
    }
}

namespace search {
    std::ostream& operator<<(std::ostream& out, const s_search& s) {
        std::string score = abs(s.score) > figure::KING_COST - MAX_PLY ?
            "mate " + std::to string((figure::KING COST - abs(s.score)) / 2) :
            "cp " + std::to string(s.score);

        out << "score " << score <<
            " nodes " << s.nNodes <<
            " time " << s.time <<
            " pv ";
        for (const Move& m : s.pv) {
            out << GetAlgebraicNotation(m) << " ";
        }
        out << std::endl;
        return out;
    }

    s_search search(int depth, const Position& pos) {
        ss = Stats();
        std::vector<Move> pv;
        isStopped = false;

        //auto start = std::chrono::high resolution clock::now();
        int score = alpha_beta(depth, 0, -INF, INF, pos);
        //auto end = std::chrono::high_resolution_clock::now();

        //auto timeMS = std::chrono::duration cast<std::chrono::milliseconds>(end - start).count();
        //int sum = 0;
        //for (auto el : bfArr) {
        //    sum += el;
        //}
        //std::cout << "Branch Factor: " << sum / bfArr.size()<<std::endl;
        //bfArr.clear();
        return { pv, UCI::get time ms() - UCI::flags.starttime, ss.nNodes, score };
    }

    bool initHashTableFlag = true;
    s search iterative deepening(int depth, const Position& pos) {
        if (initHashTableFlag) {
            ss = Stats();
            initHashTableFlag = false;
        }

        ss.killers = KillerHeuristic();
        ss.hist = HistoryHeuristic();
        ss.counterHist = CounterMoveHeuristic();

        ss.nNodes = 0;
        scorePv = false;
        followPv = false;
        isStopped = false;
```

```
    std::vector<Move> pv;
    s search res;

    //set aspiration window size
    int alpha = -INF;
    int beta = INF;

    //Iterative deepenig
    for (int d = 1; d <= depth; ++d) {

        pv.clear();
        followPv = true;

        //search
        int score = alpha beta(d, 0, alpha, beta, pos);

        if (isStopped) {
            break;
        }

        //if we failed with current aspiration window, re-search with full
        if (score <= alpha || score >= beta) {
            alpha = -INF;
            beta = INF;
            d--;
            continue;
        }

        //update aspiration window
        alpha = score - 50;
        beta = score + 50;

        //save pv line
        for (int i = 0; i < pvLength[0]; ++i) {
            pv.push back(pvTable[0][i]);
        }

        res = { pv, UCI::get time ms() - UCI::flags.starttime, ss.nNodes, score };

        std::cout << "info depth " << d << " " << res;
    }
    repetitionIndex = 0;
    return res;
}

std::ostream& operator<<(std::ostream& out, const PerftStat& s)
{
    out << "Captures: " << s.nCapture << std::endl;
    out << "EnPassant: " << s.nEnpassant << std::endl;
    out << "Castles: " << s.nCastling << std::endl;
    out << "Promotions: " << s.nPromotion << std::endl;
    return out;
}

std::vector<U64> legals;
U64  BranchFactor(const Position& pos, int depth) {
    U64 nodes = 0;
    U64 nLegal = 0;

    if (depth == 0) return 1;

    MoveList moves;
    pos.generate<PSEUDO>(moves);
    for (int i = 0; i < moves.size(); ++i) {
        Position newPos = pos.make move(moves[i]);
        if (newPos.is king attacked(~newPos.side to move())) continue;

        nodes +=  BranchFactor(newPos, depth - 1);
        nLegal++;
    }

    if (nLegal)
        legals.push back(nLegal);
    return nodes;
}
void BranchFactor(const Position& pos, int depth) {
    legals.clear();
    U64 nodes =  BranchFactor(pos, depth);

    U64 sum = 0;
```

```
        for (auto el : legals) {
            sum += el;
        }

        std::cout << "nodes: " << nodes<<std::endl;
        std::cout << "branch factor: " << sum / legals.size() << std::endl;
    }



    U64 perft(const Position& pos, int depth, PerftStat* stat)
    {
        U64 nodes = 0;
        if (depth == 0) {
            int eval = evaluate::eval(pos, WHITE);
            return 1;
        }

        MoveList moves;
        //pos.generate<LEGAL>(moves);
        pos.generate<PSEUDO>(moves);
        for (int i = 0; i < moves.size(); ++i) {
            Position newPos = pos.make move(moves[i]);

            if (newPos.is_king_attacked(~newPos.side_to_move())) {
                continue;
            }
             nodes += perft(newPos, depth - 1, stat);
        }

        return nodes;
    }
}
```

Файл Evaluate.cpp

```
#include "Evaluate.h"
#include "Position.h"
#include "Magic.h"

namespace {

    // positional piece scores [game phase][piece][square]
    constexpr int positional score[2][7][64] =
        // opening positional piece scores //
    {
        //EFigureType::NO FIGURE
        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,   0,   0,   0,   0,   0,  0,   0,

        //pawn
        0,   0,   0,   0,   0,   0,  0,   0,
        98, 134,  61,  95,  68, 126, 34, -11,
        -6,   7,  26,  31,  65,  56, 25, -20,
        -14,  13,   6,  21,  23,  12, 17, -23,
        -27,  -2,  -5,  12,  17,   6, 10, -25,
        -26,  -4,  -4, -10,   3,   3, 33, -12,
        -35,  -1, -20, -23, -15,  24, 38, -22,
        0,   0,   0,   0,   0,   0,  0,   0,

        // knight
        -167, -89, -34, -49,  61, -97, -15, -107,
        -73, -41,  72,  36,  23,  62,   7,  -17,
        -47,  60,  37,  65,  84, 129,  73,   44,
        -9,  17,  19,  53,  37,  69,  18,   22,
        -13,   4,  16,  13,  28,  19,  21,   -8,
        -23,  -9,  12,  10,  19,  17,  25,  -16,
        -29, -53, -12,  -3,  -1,  18, -14,  -19,
        -105, -21, -58, -33, -17, -28, -19,  -23,

        // bishop
        -29,   4, -82, -37, -25, -42,   7,  -8,
```

```
-26,  16, -18, -13,  30,  59,  18, -47,
-16,  37,  43,  40,  35,  50,  37,  -2,
-4,   5,  19,  50,  37,  37,   7,  -2,
-6,  13,  13,  26,  34,  12,  10,   4,
0,  15,  15,  15,  14,  27,  18,  10,
4,  15,  16,   0,   7,  21,  33,   1,
-33,  -3, -14, -21, -13, -12, -39, -21,

// rook
32,  42,  32,  51, 63,  9,  31,  43,
27,  32,  58,  62, 80, 67,  26,  44,
-5,  19,  26,  36, 17, 45,  61,  16,
-24, -11,   7,  26, 24, 35,  -8, -20,
-36, -26, -12,  -1,  9, -7,   6, -23,
-45, -25, -16, -17,  3,  0,  -5, -33,
-44, -16, -20,  -9, -1, 11,  -6, -71,
-19, -13,   1,  17, 16,  7, -37, -26,

// queen
-28,   0,  29,  12,  59,  44,  43,  45,
-24, -39,  -5,   1, -16,  57,  28,  54,
-13, -17,   7,   8,  29,  56,  47,  57,
-27, -27, -16, -16,  -1,  17,  -2,   1,
-9, -26,  -9, -10,  -2,  -4,   3,  -3,
-14,   2, -11,  -2,  -5,   2,  14,   5,
-35,  -8,  11,   2,   8,  15,  -3,   1,
-1, -18,  -9,  10, -15, -25, -31, -50,

// king
-65,  23,  16, -15, -56, -34,   2,  13,
29,  -1, -20,  -7,  -8,  -4, -38, -29,
-9,  24,   2, -16, -20,   6,  22, -22,
-17, -20, -12, -27, -30, -25, -14, -36,
-49,  -1, -27, -39, -46, -44, -33, -51,
-14, -14, -22, -46, -44, -30, -15, -27,
1,   7,  -8, -64, -43, -16,   9,   8,
-15,  36,  12, -54,   8, -28,  24,  14,


// Endgame positional piece scores //

//EFigureType::NO_FIGURE
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,   0,   0,   0,   0,   0,  0,   0,

//pawn
0,   0,   0,   0,   0,   0,   0,   0,
178, 173, 158, 134, 147, 132, 165, 187,
94, 100,  85,  67,  56,  53,  82,  84,
32,  24,  13,   5,  -2,   4,  17,  17,
13,   9,  -3,  -7,  -7,  -8,   3,  -1,
4,   7,  -6,   1,   0,  -5,  -1,  -8,
13,   8,   8,  10,  13,   0,   2,  -7,
0,   0,   0,   0,   0,   0,   0,   0,

// knight
-58, -38, -13, -28, -31, -27, -63, -99,
-25,  -8, -25,  -2,  -9, -25, -24, -52,
-24, -20,  10,   9,  -1,  -9, -19, -41,
-17,   3,  22,  22,  22,  11,   8, -18,
-18,  -6,  16,  25,  16,  17,   4, -18,
-23,  -3,  -1,  15,  10,  -3, -20, -22,
-42, -20, -10,  -5,  -2, -20, -23, -44,
-29, -51, -23, -15, -22, -18, -50, -64,

// bishop
-14, -21, -11,  -8, -7,  -9, -17, -24,
-8,  -4,   7, -12, -3, -13,  -4, -14,
2,  -8,   0,  -1, -2,   6,   0,   4,
-3,   9,  12,   9, 14,  10,   3,   2,
-6,   3,  13,  19,  7,  10,  -3,  -9,
-12,  -3,   8,  10, 13,   3,  -7, -15,
-14, -18,  -7,  -1,  4,  -9, -15, -27,
-23,  -9, -23,  -5, -9, -16,  -5, -17,
```

```
        // rook
        13, 10, 18, 15, 12,  12,   8,   5,
        11, 13, 13, 11, -3,   3,   8,   3,
        7,  7,  7,  5,  4,  -3,  -5,  -3,
        4,  3, 13,  1,  2,   1,  -1,   2,
        3,  5,  8,  4, -5,  -6,  -8, -11,
        -4,  0, -5, -1, -7, -12,  -8, -16,
        -6, -6,  0,  2, -9,  -9, -11,  -3,
        -9,  2,  3, -1, -5, -13,   4, -20,

        // queen
        -9,  22,  22,  27,  27,  19,  10,  20,
        -17,  20,  32,  41,  58,  25,  30,   0,
        -20,   6,   9,  49,  47,  35,  19,   9,
        3,  22,  24,  45,  57,  40,  57,  36,
        -18,  28,  19,  47,  31,  34,  39,  23,
        -16, -27,  15,   6,   9,  17,  10,   5,
        -22, -23, -30, -16, -16, -23, -36, -32,
        -33, -28, -22, -43,  -5, -32, -20, -41,

        // king
        -74, -35, -18, -18, -11,  15,   4, -17,
        -12,  17,  14,  17,  17,  38,  23,  11,
        10,  17,  23,  15,  20,  45,  44,  13,
        -8,  22,  24,  27,  26,  33,  26,   3,
        -18,  -4,  21,  24,  27,  23,   9, -11,
        -19,  -3,  11,  21,  23,  16,   7,  -9,
        -27, -11,   4,  13,  14,   4,  -5, -17,
        -53, -34, -21, -11, -28, -14, -24, -43
    };

    constexpr ESquare mirror square[64] =
    {
        A8, B8, C8, D8, E8, F8, G8, H8,
        A7, B7, C7, D7, E7, F7, G7, H7,
        A6, B6, C6, D6, E6, F6, G6, H6,
        A5, B5, C5, D5, E5, F5, G5, H5,
        A4, B4, C4, D4, E4, F4, G4, H4,
        A3, B3, C3, D3, E3, F3, G3, H3,
        A2, B2, C2, D2, E2, F2, G2, H2,
        A1, B1, C1, D1, E1, F1, G1, H1
    };


    template<EColor color, EFigureType type>
    inline void EvalType(const Position& pos, int& opening_score, int& endgame_score) {
        using namespace evaluate;
        constexpr int sideCoef = color == WHITE ? 1 : -1;

        const auto& fileMask = Rays::Get().getFileMask();
        const auto& isolatedPawnMask = Rays::Get().getIsolatedPawnMask();
        const auto& passedPawnMask = Rays::Get().getPassedPawnMask();

        U64 bb = pos.figures(color, type);
        while (bb) {

            ESquare sq = misc::pop lsb(bb);
            int mirroredSq = mirror_square[sq];

            opening score += sideCoef * positional score[OPENNING][type][color == WHITE ? mirroredSq :
sq];
            opening score += sideCoef * material score[OPENNING][type];

            endgame score += sideCoef * positional score[END GAME][type][color == WHITE ? mirroredSq :
sq];
            endgame score += sideCoef * material score[END GAME][type];

            if constexpr (type == PAWN) {
                //doubled penalty
                int nDoubled = misc::countBits(pos.figures(color, PAWN) & fileMask[sq]);
                if (nDoubled > 1) {
                    opening score += sideCoef * (nDoubled - 1) * evaluate::penalty::doubledPawnOpening;
                    endgame score += sideCoef * (nDoubled - 1) * evaluate::penalty::doubledPawnEndgame;
                }

                //passed pawn bonus
                if ((passedPawnMask[color][sq] & pos.figures(~color, PAWN)) == 0) {
                    int bonus = bonus::passed pawn[misc::square to rank[color == WHITE ? mirroredSq :
sq]];
```

```
                    opening score += sideCoef * bonus;
                    endgame score += sideCoef * bonus;
                }

                //isolated pawn penalty
                if ((isolatedPawnMask[sq] & pos.figures(color, PAWN)) == 0) {
                    opening score += sideCoef * penalty::isolatedPawnOpening;
                    endgame score += sideCoef * penalty::isolatedPawnEndgame;
                }

            }

            if constexpr (type == BISHOP) {
                //mobility
                int mobilityBonus = misc::countBits(GetAttackBB<type>(sq, pos.figures(ALL FIGURE TYPE))) -
4;
                opening score += sideCoef * mobilityBonus * 5;
                endgame score += sideCoef * mobilityBonus * 5;
            }

            if constexpr (type == ROOK) {
                //semi open file bonus
                if ((pos.figures(color, PAWN) & fileMask[sq]) == 0) {
                    opening score += sideCoef * bonus::semiOpenFile;
                    endgame_score += sideCoef * bonus::semiOpenFile;

                    //open file penalty
                    if ((pos.figures(PAWN) & fileMask[sq]) == 0) {
                        opening score += sideCoef * bonus::openFile;
                        endgame score += sideCoef * bonus::openFile;
                    }
                }
            }

            if constexpr (type == QUEEN) {
                int mobilityBonus = misc::countBits(GetAttackBB<type>(sq, pos.figures(ALL FIGURE TYPE))) -
9;
                opening score += sideCoef * mobilityBonus;
                endgame score += sideCoef * mobilityBonus * 2;
            }

            if constexpr (type == KING) {
                //king shield bonus
                int kingRays = Rays::Get().getPseudoAttacks()[KING][sq];
                int bonus = sideCoef
                    * misc::countBits(kingRays & pos.figures(color))
                    * bonus::kingShield;
                opening_score += bonus;
                endgame_score += bonus;


                //semi open file penalty
                const U64 usPawnBB = pos.figures(color, PAWN);
                if ((usPawnBB & fileMask[sq]) == 0) {
                    opening_score += sideCoef * penalty::semiOpenFile;
                    endgame score += sideCoef * penalty::semiOpenFile;

                    //open file penalty
                    if ((pos.figures(PAWN) & fileMask[sq]) == 0) {
                        opening_score += sideCoef * penalty::openFile;
                        endgame score += sideCoef * penalty::openFile;
                    }
                }
            }
        }
    }
}

namespace evaluate {

    int eval(const Position& pos, EColor relativeTo) {
        int gamePhaseScore = pos.phase score();
        EGamePhase gp = pos.phase();

        int openingScore = 0;
        int endgameScore = 0;
        EvalType<WHITE, PAWN>(pos, openingScore, endgameScore);
        EvalType<BLACK, PAWN>(pos, openingScore, endgameScore);

        EvalType<WHITE, KNIGHT>(pos, openingScore, endgameScore);
```

```
        EvalType<BLACK, KNIGHT>(pos, openingScore, endgameScore);

        EvalType<WHITE, BISHOP>(pos, openingScore, endgameScore);
        EvalType<BLACK, BISHOP>(pos, openingScore, endgameScore);

        EvalType<WHITE, ROOK>(pos, openingScore, endgameScore);
        EvalType<BLACK, ROOK>(pos, openingScore, endgameScore);

        EvalType<WHITE, QUEEN>(pos, openingScore, endgameScore);
        EvalType<BLACK, QUEEN>(pos, openingScore, endgameScore);

        EvalType<WHITE, KING>(pos, openingScore, endgameScore);
        EvalType<BLACK, KING>(pos, openingScore, endgameScore);

        //interpolation between opening and endgame scores
        int score = gp == MIDDLE GAME ?
            (openingScore * gamePhaseScore +
                endgameScore * (opening bounder score - gamePhaseScore)) / opening bounder score
            :
            gp == OPENNING ? openingScore : endgameScore;

        //score relative to the given side
        return relativeTo == WHITE ? score : -score;
    }
}
```

Весь код можна знайти за посиланням:
https://github.com/tkchmax/ChessEngine/tree/master/ChessEngine