КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

Кваліфікаційна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою "Інформатика"
спеціальності 122 Комп'ютерні науки на тему:

АНАЛІЗ ТА ПОРІВНЯННЯ СТРАТЕГІЙ ДЛЯ ГРИ В ШАХИ З НЕПОВНОЮ ІНФОРМАЦІЄЮ

Виконав студент 4-го курсу
Луценко Костянтин Олександрович _____

(підпис)

Науковий керівник:
доцент, доктор фізико-математичних наук
Завадський Ігор Олександрович _____

(підпис)

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

Студент _____

(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи: 40 сторінок, 16 ілюстрацій, 2 таблиці, 11 використаних джерел.

Ключові слова: ШАХИ З НЕПОВНОЮ ІНФОРМАЦІЄЮ, ШАХИ З ТУМАНОМ ВІЙНИ, АЛГОРИТМИ ДЛЯ ГРИ В ШАХИ, РЕЙТИНГ ЕЛО.

Об’єктом роботи є варіант шахів з неповною інформацією, який відомий як шахи з туманом війни або шахи втемну.

Метою кваліфікаційної роботи є створення, публікація, аналіз та порівняння алгоритмів для гри в шахи з туманом війни.

Інструментом створення є безкоштовний, вільно поширюваний редактор коду `Code::Blocks`, мова розмітки тексту `Typst`, мова програмування `C++` та графічна бібліотека `SFML`.

Результат роботи: розроблено та опубліковано 10 алгоритмів для гри в шахи з неповною інформацією, а також проведено їх порівняльну характеристику та розраховано орієнтовний рейтинг Ело цих алгоритмів.

ЗМІСТ

Вступ 5

Скорочення та умовні позначення 7

Розділ 1. Загальний огляд шахів з туманом війни 8

1.1. Правила гри 8

1.2. Огляд наявних платформ для гри 13

Розділ 2. Огляд розроблених алгоритмів 15

2.1. Реалізація правил гри 15

2.2. Інтерфейс розроблених алгоритмів 16

2.3. RandomMove 17

2.4. CaptureKing 18

2.5. CaptureLargest 19

2.6. CaptureWithLargestDifference 20

2.7. Supporter 21

2.8. PositionaryEvaluator 22

2.9. NeuralNetworkPlayer 25

2.10. Алгоритм для звичайних шахів MyStockfish 26

2.11. SimpleStockfish 28

2.12. PossibleBoardGenerator 29

2.13. PossibleBoardList 30

Розділ 3. Аналіз та порівняння алгоритмів 33

3.1. Реалізація гри алгоритмів один з одним 33

3.2. Розрахунок рейтингу Ело 33

3.3. Результати 35

Висновки 39

Перелік джерел посилання 40

ВСТУП

Шахи - це стратегічна гра, у яку люди грають вже протягом століть. Її складність і глибина зробили її предметом вивчення та аналізу, незліченна кількість книг, статей і комп’ютерних програм присвячена аналізу шахових ігор. Однак серед величезного масиву шахової літератури існує менш поширений варіант гри, відомий як «шахи з туманом війни», або «шахи втемну» (англ. Dark Chess), який вносить зміни у гру завдяки концепції туману війни.

Шахи з туманом війни ускладнюють аналіз позиції, оскільки вони позбавлені повного знання про положення фігур свого опонента. У цьому варіанті у традиційний чіткий вигляд шахівниці додається фактор невизначеності та обмеженої інформації, що вносить елемент несподіванки та напруги в кожен хід.

Шахи з туманом війни були винайдений Єнсом Беком Нільсеном і Торбеном Остедом у 1989 році [1]. Ця версія шахів натхненна версією «Крігшпіль» (Kriegsspiel), і, як і в тій версії, мета створення шахів з туманом війни — максимально наблизитися до справжньої війни. Для цього найголовніше, щоб ви точно не знали, де знаходяться фігури супротивника. Єнс Бек Нільсен і Торбен Остед назвали цей варіант «шахи втемну» (англ. Dark Chess), оскільки під час гри вони відчували, ніби рухались у темряві.

Класичні шахи, відомі своєю складністю та стратегічною глибиною, слугували плідним ґрунтом для алгоритмічних розробок

та досліджень у галузі штучного інтелекту. Протягом багатьох років було розроблено незліченну кількість алгоритмів і комп’ютерних програм для опанування традиційної гри в шахи, які розширюють межі комп’ютерного мислення та прийняття рішень. Однак шахи з туманом війни набагато менш досліджені: на просторах Інтернету було знайдено декілька блогів, які намагалися б дослідити оптимальні стратегії для перемоги [2] [3], проте інтерес до програмування алгоритмів для гри в такий варіант шахів є значно меншим.

У наступних розділах цієї роботи буде розглянуто правила і механіку темних шахів, а також декілька алгоритмів для гри у шахи з туманом війни; буде порівняно ці алгоритми шляхом проведення турніру між ними; а також розраховано приблизний рейтинг Ело кожного з цих алгоритмів.

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

CFoW Chess Fog of War, шахи з туманом війни.

ШНМ Штучна нейронна мережа

РОЗДІЛ 1. ЗАГАЛЬНИЙ ОГЛЯД ШАХІВ З ТУМАНОМ ВІЙНИ

1.1. Правила гри

В шахи з туманом війни грають два гравці — білі та чорні. В кожного є свій набір фігур, який складається з:

- короля;
- королеви;
- двох коней;
- двох слонів;
- двох тур;
- восьми пішаків.

Початкове розташувння фігур показано на рис.1.

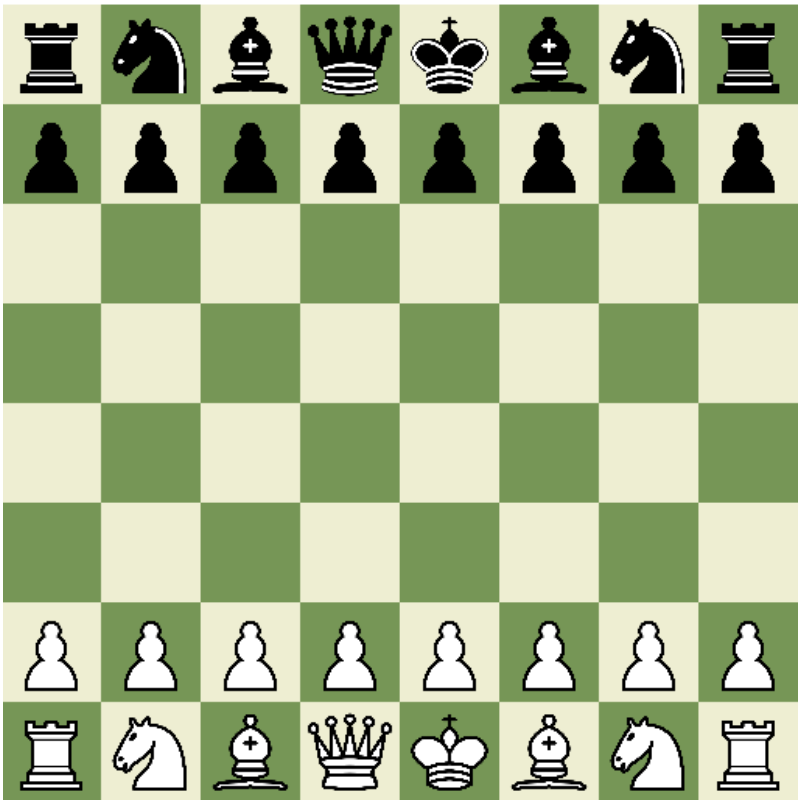

Рисунок 1: Початкове розташування фігур

Гравці ходять по черзі, розпочинають білі. У кожної фігури є правила, за якими кожна фігура може переміщатись по дошці. Фігури можуть закінчувати хід лише на пустих клітинках або клітинках, зайнятих фігурами суперника. В останньому випадку фігура супротивника знімається з дошки. Така операція називається взяттям.

Мета гри — взяти короля супротивника. Якщо гравці виконали по 75 ходів поспіль і за ці 150 ходів не було виконано жодного взяття або хода пішака, оголошується нічия. В турнірних партіях за перемогу гравець отримує 1 очко, за нічию — $\frac{1}{2}$ очка, за поразку — 0 очок.

Далі буде наведено можливі ходи для кожного типу фігур. Для зрозумілості введемо систему координат Oxy, як показано на рис. 2. Будемо вважати клітинку a1 клітинкою з координатами $(1, 1)$, а клітинку h8 — клітинкою з координатами $(8, 8)$.

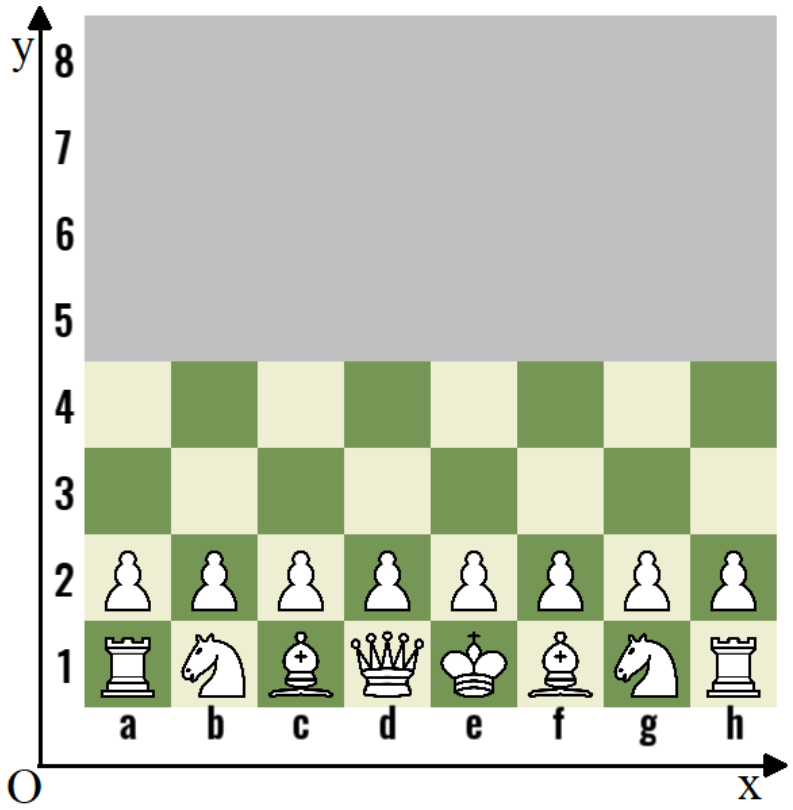

Рисунок 2: Система координат для пояснення можливих ходів

Пішак може виконати хід на одну клітинку вперед від гравця, якому він належить, тобто з клітинки (x, y) на клітинку $(x, y + 1)$ для білого та $(x, y - 1)$ для чорного пішака. Цей хід не може бути взяттям, тобто клітинка призначення має бути порожньою. Якщо пішак ще не робив жодного хода у грі (тобто знаходиться на 2-ій горизонталі для білого і на 6-й для чорного пішака), то він може походити на дві клітинки вперед, тобто з клітинки (x, y) на клітинку $(x, y + 2)$ для білого та $(x, y - 2)$ для чорного пішака. Також пішак може виконувати взяття, переміщаючись на одну клітинку вперед та на одну клітинку вбік тобто з клітинки (x, y) на клітинку $(x \pm 1, y + 1)$ для білого та $(x \pm 1, y - 1)$ для чорного пішака. В такому разі на клітинці призначення має бути розташована фігура суперника. Якщо

пішак опиняється на останній горизонталі (8-ій для білих та 1-ій для чорних), то він перетворюється на королеву.

Слон може виконати хід по діагоналі на довільну кількість клітинок, тобто з клітинки (x_1, y_1) на клітинку (x_2, y_2), якщо $|x_1 - x_2| = |y_1 - y_2|$. Слон не може «перестрибувати» через інші фігури, тобто всі клітинки між початковою та кінцевою клітинкою мають бути порожні.

Кінь може викоати хід на 2 клітинки по одному напрямку і на 1 клітинку по перпендикулярному напрямку, тобто з клітинки (x, y) на клітинки $(x \pm 1, y \pm 2)$ або $(x \pm 2, y \pm 1)$. Кінь може «перестрибувати» через інші фігури.

Тура може виконати хід вздовж горизонтальної або вертикальної прямої на довільну кількість клітинок, тобто з клітинки (x_1, y_1) на клітинку (x_2, y_2), якщо $x_1 = x_2$ або $y_1 = y_2$. Тура не може «перестрибувати» через інші фігури, тобто всі клітинки між початковою та кінцевою клітинкою мають бути порожні.

Королева може виконати довільний хід тури або слона, тобто з клітинки (x_1, y_1) на клітинку (x_2, y_2), якщо $|x_1 - x_2| = |y_1 - y_2|$, $x_1 = x_2$ або $y_1 = y_2$. Королева не може «перестрибувати» через інші фігури, тобто всі клітинки між початковою та кінцевою клітинкою мають бути порожні.

Король може виконати хід на одну клітинку в довільному напрямку, включаючи діагональні: тобто з клітинки (x_1, y_1) на клітинку (x_2, y_2), якщо $|x_1 - x_2| \leq 1$ і $|y_1 - y_2| \leq 1$.

Також, якщо король і одна з тур знаходяться на початкових позиціях, то можна виконати особливий хід — рокировку — яка полягає в переміщенні короля на дві клітинки в сторону тури і одночасно переміщення тури на клітинку, через яку пройшов король. У коді цей хід зберігається як хід короля на дві клітинки.

На відміну від звичайних шахів, шахи з туманом війни є грою з неповною інформацією. Таким чином, гравці в кожен момент часу бачать не всі клітинки дошки. Гравці мають інформацію лише про ті клітинки дошки, де може закінчити хід хоча б одна з їх фігур. Зокрема, якщо можливо виконати взяття, то гравець завжди бачить фігуру, яку можна взяти. На рис.2 зображений приклад, які клітинки будуть бачити гравці в конкретній позиції.

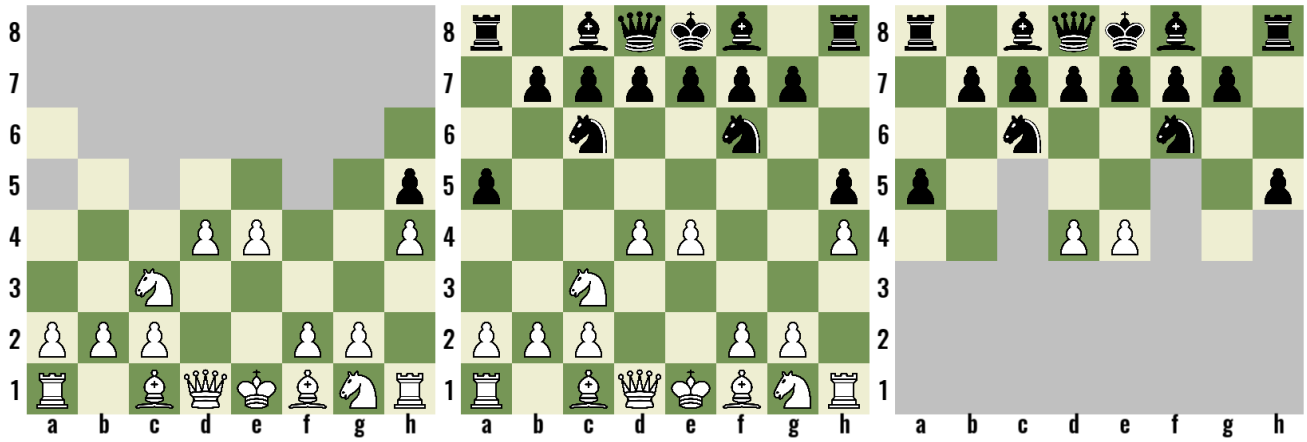

Рисунок 3: Приклад видимих клітинок (зліва направо): як бачать дошку білі, вся дошка, як бачать дошку чорні

1.2. Огляд наявних платформ для гри

В мережі Інтернет наявні декілька платформ для гри в шахи з туманом війни. Вони можуть містити незначні відмінності у правилах. Здебільшого відмінності полягають у ходах пішаків: по-перше, якщо перед пішаком стоїть фігура, то в деяких варіантах гравець бачить фігуру перед пішаком (хоча пішак і не може туди походити), також в деяких варіантах додано взяття на проході — якщо пішак суперника виконав хід на дві клітинки, то союзний пішак може виконати особливий хід, при якому взяття відбудеться не на клітинці, на якій опинився пішак суперника, а на тій клітинці, через яку він пройшов. Також в більшості платформ пішак, який дійшов до останньої горизонталі, може перетворюватись на довільну фігуру окрім короля; в деяких платформах при перетворенні суперник отримує сповіщення про нього.

Нижче наведено список платформ для гри в шахи з туманом війни та короткий опис відмінностей у правилах для кожної платформи:

- Chess.com [4] — найпопулярніша платформа для гри в шахи з туманом війни. Пішаки не бачать фігуру, якщо вона стоїть перед ними;
- BrainKing [5] — перетворення пішаків лишаються невидимими для суперника;

- ItsYourTurn [6] — опонент отримує сповіщення про перетворення пішаків;
- SchemingMind [7] — пішаки не бачать фігуру, якщо вона стоїть перед ними;
- AjaxPlay [8] — перетворення пішаків лишаються невидимими для суперника;
- GoldToken [9] — перетворення пішаків лишаються невидимими для суперника.

РОЗДІЛ 2. ОГЛЯД РОЗРОБЛЕНИХ АЛГОРИТМІВ

2.1. Реалізація правил гри

Перерахування `Color` містить два можливих значення — `White` та `Black`, і зображає колір фігури (білий або чорний відповідно).

Перерахування Type містить 8 можливих значень — `None`, `Fog`, `Pawn`, `Knight`, `Bishop`, `Rook`, `Queen`, `King`, і зображає тип клітинки або фігури на ній, тобто чи є клітинка порожньою, невидимою, або на ній знаходиться пішак, кінь, слон, тура, королева або король відповідно.

Структура `Piece` містить поля типів `Type` і `Color` і зображає одну клітинку дошки. Якщо в полі типу `Type` знаходиться `Fog` або `None` (тобто клітинка невидима або порожня), то поле типу Color не несе інформації.

Дошка — це квадратний масив 8×8 з структур `Piece`.

Структура `Square` містить поля `x` та `y` і зображає координати деякої клітинки.

Структура Move містить два поля типу Square (`from` і `to`) — координати полів, на яких відбувся хід (з якого та на яке). Також вона містить поле типу `Type`, яке зображає фігуру, яка виконала цей хід.

Наступні функції задають інтерфейс взаємодії з вище описаними структурами:

- `Board InitialBoard()` — повертає дошку, яка описує початкове положення фігур;
- `std::vector<Move> GetMoves(const Board& b, Color c)` — повертає список всіх ходів, які можна зробити у даній позиції гравцю c;
- `Board ApplyMove(const Board& b, Move m)` — повертає дошку b після виконання на ній ходу m;
- `Board GetFoggedBoard(const Board& b, Color c)` — повертає дошку b таку, як її би бачив гравець c (тобто всім невидимим клітинкам встановлює значення type = Fog).

2.2. Інтерфейс розроблених алгоритмів

Інтерфейс алгоритму для CFoW описаний у віртуальному класі Player та містить дві функції (див. рис. 3):

- `Player::Initialize(Color color)` — функція, яка ініціалізує алгоритм для початку гри за колір color.
- `Move Player::GetMove(const Board& b0, const Board& b)` — функція, яка має повертати обраний алгоритмом хід. Вона отримує на вхід дві дошки: дошку, яку бачив цей алгоритм до ходу супротивника, та після його ходу.

```
class Player
{
public:
    virtual void Initialize(Color color) = 0;
    virtual Move GetMove(const Board& b0, const Board& b) = 0;
};
```

Рисунок 4: Оголошення класу `Player` та його методів

Таким чином, кожен алгоритм буде являти собою клас (що вже не є віртуальним), який наслідується від `Player` та в якому визначені вище згадані функції.

2.3. RandomMove

Алгоритм `RandomMove` приймає рішення робити випадковий хід з усіх можливих. Алгоритм використовує функцію `GetMoves` для отримання всіх можливих ходів з поточної позиції, після чого генерує випадкове число по модулю кількості поточних ходів та обирає хід з номером, рівним згенерованому числу.

Цей алгоритм наявний здебільшого для порівняння з іншими алгоритмами, адже по відсотку перемог над ним можна зрозуміти, чи є якийсь інший алгоритм кращим за випадковий вибір ходу.

Код методу `Initialize` класу `RandomMove` містить лише один рядок, в якому запам’ятовується колір, за який грає алгоритм. Код методу `GetMove` цього класу наведено на рис.5.

```
Move RandomMove::GetMove(const Board& b0, const Board& b)
{
    auto v = GetMoves(b, c);
    int r = rand()%v.size();
    return v[r];
}
```

Рисунок 5: Код методу GetMove класу RandomMove

2.4. CaptureKing

Алгоритм `CaptureKing` приймає рішення брати короля суперника, якщо такий хід є в наявності. У іншому випадку алгоритм аналогічний RandomMove. Алгоритм використовує функцію `GetMoves` для отримання всіх можливих ходів з поточної позиції, після чого ітерується по всім можливим ходам. Якщо для якогось із ходів в клітинці призначення на поточній дошці стоїть король, то алгоритм повертає цей хід. Інакше, так само як і в алгоритмі `RandomMove`, генерується випадкове число по модулю кількості ходів та обирається хід з номером, рівним згенерованому числу.

Цей алгоритм має велику перевагу над алгоритмом `RandomMove`, але також гарно показує силу інших алгоритмів, адже по відсотку перемог над ним можна зрозуміти, чи інший алгоритм робить щось краще, аніж спроба виграти в один хід (взяти короля).

Код методу `Initialize` класу `CaptureKing` містить лише один рядок, в якому запам’ятовується колір, за який грає алгоритм. Код методу `GetMove` цього класу наведено на рис.6.

```
Move CaptureKing::GetMove(const Board& b0, const Board& b)
{
    auto v = GetMoves(b, c);
    for (auto m : v)
        if (b[m.to.x][m.to.y].type == Type::King)
            return m;
    return v[rand()%v.size()];
}
```

Рисунок 6: Код методу `GetMove` класу `CaptureKing`

2.5. CaptureLargest

Алгоритм `CaptureLargest` приймає рішення брати найбільшу фігуру суперника, або випадкову з них, якщо таких фігур декілька. Формально, якість ходу з клітинки A на клітинку B оцінюється як вага фігури, що стояла на клітинці B, збільшена на випадкове мале число (для того, щоб при рівних значеннях якості ходу обирався випадковий з таких ходів).

Цей алгоритм реалізує такий метод програмування, як жадібний алгоритм, адже якщо оцінкою позиції вважати сумарну вагу своїх фігур мінус сумарну вагу фігур суперника, то цей алгоритм обирає хід, що максимізує оцінку позиції після свого ходу.

Алгоритм використовує функцію `GetMoves` для отримання всіх можливих ходів з поточної позиції, після чого ітерується по всім можливим ходам. Якщо для якогось із ходів його якість перевищує найкращу якість ходу, знайдену на даний момент, то кращий хід та його якість замінюються поточним ходом та його якістю. Таким

чином, в кінці алгоритму в змінній, що зберігає кращий хід, буде зберігатись хід з найкращою якістю.

Код методу `Initialize` класу `CaptureLargest` містить лише один рядок, в якому запам'ятовується колір, за який грає алгоритм. Код методу `GetMove` цього класу наведено на рис.7.

```
Move CaptureLargest::GetMove(const Board& b0, const Board& b)
{
    double res = -1e9;
    Move best;
    for (Move m : GetMoves(b, c))
    {
        double score = clvalue(b[m.to.x][m.to.y].type) + rand()*1e-6/RAND_MAX;

        if (score > res)
        {
            res = score;
            best = m;
        }
    }
    return best;
}
```

Рисунок 7: Код методу GetMove класу CaptureLargest

Вага фігур тут і далі оцінюється так: пішак — 1, кінь та слон — 3, тура — 5, королева — 10, король — 1000.

2.6. CaptureWithLargestDifference

Алгоритм `CaptureWithLargestDifference` обирає хід, який максимізує вагу взятої фігури супротивника, зменшену на вагу фігури, що виконує взяття. Формально, якість ходу з клітинки A на клітинку B оцінюється як вага фігури, що стояла на клітинці B мінус вага фігури, що стояла на клітинці A, збільшена на випадкове мале число (для того, щоб при рівних значеннях якості ходу обирався випадковий з таких ходів).

Алгоритм використовує функцію `GetMoves` для отримання всіх можливих ходів з поточної позиції, після чого ітерується по всім можливим ходам. Якщо для якогось із ходів його якість перевищує найкращу якість ходу, знайдену на даний момент, то кращий хід та його якість замінюються поточним ходом та його якістю. Таким чином, в кінці алгоритму в змінній, що зберігає кращий хід, буде зберігатись хід з найкращою якістю.

Код методу `Initialize` класу `CaptureWithLargestDifference` містить лише один рядок, в якому запам'ятовується колір, за який грає алгоритм. Код методу `GetMove` цього класу наведено на рис.7.

```
Move CaptureWithLargestDifference::GetMove(const Board& b0, const Board& b)
{
    double res = -1e9;
    Move best;
    for (Move m : GetMoves(b, c))
    {
        double score = cwldvalue(b[m.to.x][m.to.y].type)
                     - cwldvalue(b[m.from.x][m.from.y].type) + rand()*1e-6/RAND_MAX;

        if (score > res)
        {
            res = score;
            best = m;
        }
    }
    return best;
}
```

Рисунок 8: Код методу GetMove класу CaptureWithLargestDifference

2.7. Supporter

Алгоритм `Supporter` оцінює якість ходу так само, як і алгоритм `CaptureLargest`, тобто ціною взятої фігури, але після ходу до оцінки якості хода за кожну прикриту фігуру до оцінки додається одиниця. Прикрита фігура — це фігура, яка не є пішаком або королем, та

яку можна було б взяти, якби вона була кольору супротивника. Таким чином, алгоритм намагається розташовувати фігури на таких позиціях, на яких не можна втратити фігуру «задарма».

Алгоритм використовує функцію `GetMoves` для отримання всіх можливих ходів з поточної позиції, після чого ітерується по всім можливим ходам. Якщо для якогось із ходів його якість перевищує найкращу якість ходу, знайдену на даний момент, то кращий хід та його якість замінюються поточним ходом та його якістю. Таким чином, в кінці алгоритму в змінній, що зберігає кращий хід, буде зберігатись хід з найкращою якістю.

Код методу `Initialize` класу `Supporter` містить лише один рядок, в якому запам’ятовується колір, за який грає алгоритм. Код методу `GetMove` цього класу наведено на рис.7.

2.8. PositionaryEvaluator

Алгоритм PositionaryEvaluator зберігає набір параметрів, за якими він оцінює хід в даній позиції. Якість ходу фігури S з клітинки A на клітинку B оцінюється як вага фігури, що стояла на клітинці B, до чого додається якість фігури S на позиції B і віднімається вага фігури S на позиції A. Таким чином, для кожної фігури супротивника в масиві параметрів є відповідна вага, а також для кожної пари «фігура» – «клітинка» також є відповідна вага, що впливає на якість ходу.

Для того, щоб знайти оптимальний (або близький до оптимального) масив параметрів, було використано генетичний алгоритм, а саме: була створена популяція з 64 об’єктів класу PositionaryEvaluator з випадковими параметрами, а була проведена операція імітації природного відбору, а саме: кожен елемент популяції зіграв з кожним по партії в CFoW, далі по результатам турніру кращі 8 об’єктів лишаються і породжують інші 56 об’єктів. Кожен з нових об’єктів копіює масив параметр випадкового з кращих 8 об’єктів, після чого відбувається випадкова мутація: кожен з параметрів змінюється на випадкове мале число. Після кожної операції кращий об’єкт грає 1000 партій з алгоритмом CaptureLargest.

Генетичний алгоритм виконує функція `RunGeneticAlgorithm`, яка приймає:

- Масив вказівників на об’єкти класу `ParametrizedPlayer` або його нащадків — це вказівники на гравців, які власне й будуть еволюціонувати. Віртуальний клас `ParametrizedPlayer` в свою чергу наслідує клас `Player` з єдиною доданою функцією `SetParameters`, що приймає масив параметрів і встановлює значення поля `params` рівним цьому масиву. Далі значення цього поля буде використовуватись при оцінці позиції;
- Масив параметрів, який буде встановлено гравцям на етапі ініціалізації;
- Розмір популяції;

- Кількість лідерів, які «виживуть», тобто будуть перенесені до наступної популяції;
- Кількість ігор, що кожної епохи буде грати кожен гравець з кожним;
- Крок зміни масиву параметрів, тобто максимальне число, на яке може змінитися деякий параметр в межах однієї мутації.

Таким чином, для запуску генетичного алгоритму з параметрами, наведеними вище, було виконано наступний виклик функції `RunGeneticAlgorithm`:

```
RunGeneticAlgorithm(v, vector<double>(m, 0.0),
64, 8, 1, 1.0)
```

, де `v` — масив вказівників на гравців, а `m` — загальна кількість параметрів у гравців.

Нижче наведено, як змінювався графік кількості виграшів проти CaptureLargest з плином часу. Таким чином, після деякого моменту фактична сила гри алгоритму змінюється не сильно, тому після 440 епох еволюцію було прийнято рішення припинити.

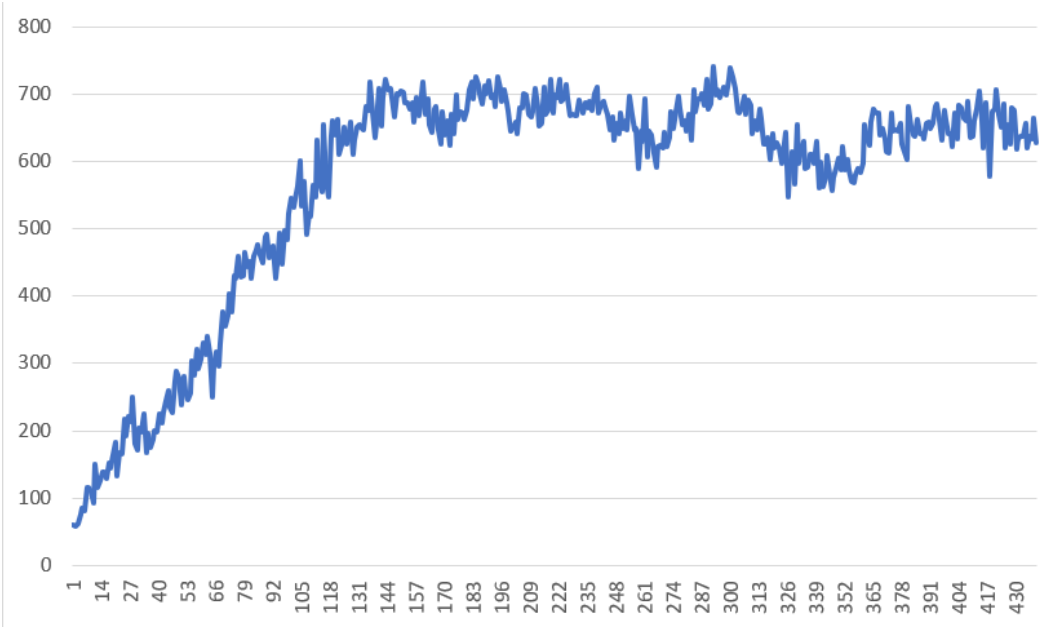

Рисунок 9: Графік кількості перемог над алгоритмом CaptureLargest

2.9. NeuralNetworkPlayer

Алгоритм `NeuralNetworkPlayer` використовує штучну нейронну мережу для прийняття рішень стосовно того, який хід робити. З сайту chess.com [4] було завантажено 100 ігор в шахи з туманом війни. На основі цих ігор нейронна мережа натренована приймати на вхід позицію шахів з туманом війни, і на виході повертає для кожної клітинки ймовірність, чи є ця клітинка тою, з якої був зроблений хід, та чи є ця клітинка тою, на яку був зроблений хід. Також в алгоритм вбудована перевірка на те, чи можна взяти короля суперника. В такому разі король береться, інакше викликається нейронна мережа.

Нейронна мережа має наступний вигляд: 14×64 вхідних нейронів (по 14 нейронів на кожну клітинку, один з яких дорівнює одиниці в залежності від того, чи є ця клітинка порожньою, туманом, білими або чорними пішаком, конем, слоном, турою, королевою або королем), 2 прихованих шари по 64 нейрони та вихідний шар з 2×64 нейронів. На кожну клітинку у вихідному шарі відводиться два нейрони. Рівно два нейрони вихідного шару будуть дорівнювати одиниці, тому тривіальна ШНМ (така, що значення всіх вихідних нейронів завжди дорівнюють нулеві) буде мати середню похибку, рівну двом.

На рис.10 наведено, як змінювалася середня похибка ШНМ протягом тренування.

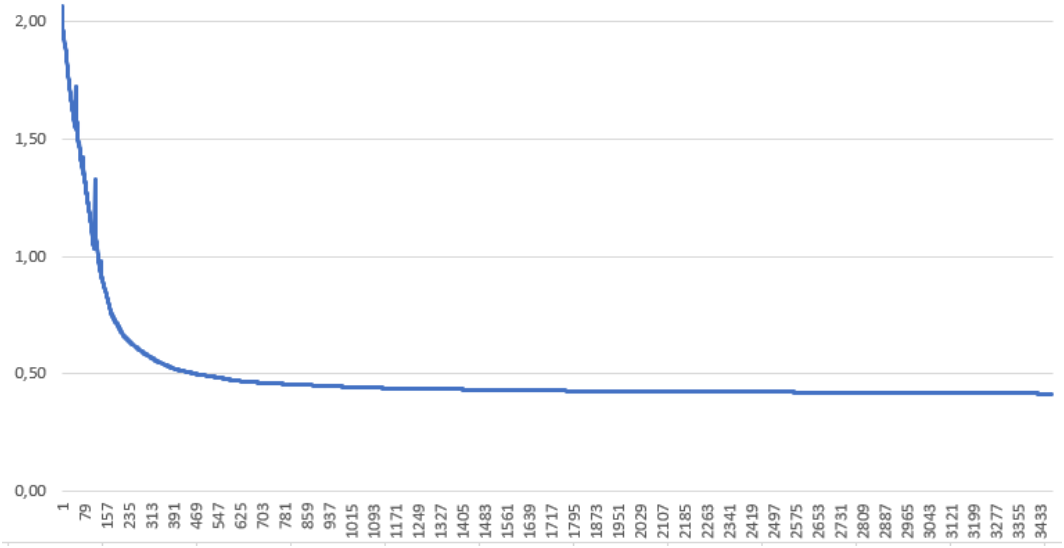

Рисунок 10: Графік середньої помилки ШНМ від номеру епохи

2.10. Алгоритм для звичайних шахів MyStockfish

Подальші стратегії використовують алгоритм, який грав би в класичні шахи (з повною інформацією). Цей алгоритм будує дерево пошуку певної глибини і приймає рішення про оцінку позиції й певних ходів на основі методу прийняття рішень «мінімакс», тобто при виборі ходу для білих обирає максимальне значення оцінки позиції, а при виборі ходу для чорних обирає мінімальне значення оцінки позиції. Оцінка позиції ж розраховується рекурсивним викликом тої ж функції, але з меншою глибиною.

```
double ans = (c == Color::White) ? -1e9 : 1e9;

for (Move m : GetMoves(b, c))
{
    double r = Evaluate(ApplyMove(b, m), (c == Color::White) ? Color::Black : Color::White, depth-1, m.to);
    if (c == Color::White) ans = max(ans, r);
    else ans = min(ans, r);
}
return ans;
```

Рисунок 11: Основна частина шахового алгоритму: при `c==Color::White` обирається максимум, інакше — мінімум

В кінцевих точках побудованого дерева пошуку (в тих точках, де глибина дорівнює нулеві) оцінка позиції розраховується тривіально: кожна фігура вносить свій вклад в залежності від типу фігури, її кольору та позиції. Колір фігури впливає на знак виразу, який вона додає до позиції, «+» для білих та «-» для чорних фігур.

Базові значення ваг кожної фігури співпадають зі значеннями, використаними в алгоритмах CaptureLargest та CaptureLargestDifference: пішак — 1, кінь та слон — 3, тура — 5, королева — 10, король — 1000, проте для того, щоб алгоритм обирав більш розвиваючі ходи (а не лише слідкував за кількістю фігур) вага фігури в конкретній позиції коригується її положенням. Наприклад, пішак важить тим більше, чим далі він продвинутий, коней і королеву краще тримати більше до центру, а короля — навпаки, тримати більше до свого краю дошки.

```
double EvaluatePiece(Type t, int x, int y)
{
    // y == 0 => white side
    if (t == Type::Pawn) return 1+0.01*(y-1);
    if (t == Type::Knight) return 3-0.1*sqrt(pow(abs(x-3.5), 2) + pow(abs(y-3.5), 2));
    if (t == Type::Bishop) return 3-0.05*sqrt(pow(abs(x-3.5), 2) + pow(abs(y-3.5), 2));
    if (t == Type::Rook) return 5-0.01*abs(x-3.5);
    if (t == Type::Queen) return 10-0.1*sqrt(pow(abs(x-3.5), 2) + pow(abs(y-3.5), 2));
    if (t == Type::King) return 1000-0.02*y;
    return 0;
}
```

Рисунок 12: Оцінка фігури t на клітинці поля (x, y)

Цей алгоритм було названо `MyStockfish`. Він має інтерфейс, що складається з двох основних функцій:

- `std::pair<double, Move> Run(Board b, Color c, int depth)` — повертає кращий хід і оцінку позиції після нього;
- `std::vector<std::pair<double, Move> > All(Board b, Color c, int depth)` — повертає список всіх ходів разом з оцінками позиції після них.

На табл.1 наведений час роботи функції `Run(InitialBoard(), Color::White, d)` за різних значень глибини `d`.

Глибина	1	2	3	4	5
Час роботи	<1мс	1мс	0.02с	0.56с	13.61с

Таблиця 1: Час роботи MyStockfish на початковій дошці.

2.11. SimpleStockfish

Алгоритм `SimpleStockfish` викликає `MyStockfish` від дошки, яку бачить гравець на момент свого ходу. На основі цього робиться вибір кращого ходу. При виклику `MyStockfish` від дошки, у якій деякі клітинки є туманом, ходи на територію туману в дереві

пошуку є забороненими, тобто дерево пошуку будується лише для тих клітинок, які були видимі на момент ходу, і ця множина не змінюється при виконанні пошуку.

Глибина пошуку може бути встановлена різною для різних цілей. Для подальшого тестування вона була рівна трьом.

```
Move SimpleStockfish::GetMove(const Board& b0, const Board& b)
{
    return MyStockfish::Run(b, c, 3).second;
}
```

Рисунок 13: Код методу GetMove класу SimpleStockfish

2.12. PossibleBoardGenerator

Алгоритм `PossibleBoardGenerator` генерує дошку, яка підходить під поточну ситуацію, яку бачить алгоритм. Ця дошка зберігається у змінній possibleBoard. Дошка генерується в залежності від значення possibleBoard на попередньому ході. При ініціалізації можлива дошка дорівнює початковому розташуванню фігур.

```
Move PossibleBoardGenerator::GetMove(const Board& b0, const Board& b)
{
    possibleBoard = GenerateBoard(possibleBoard, b, c);

    Move m = MyStockfish::Run(possibleBoard, c, 3).second;
    possibleBoard = ApplyMove(possibleBoard, m);
    return m;
}
```

Рисунок 14: Код методу GetMove класу PossibleBoardGenerator

Генерація дошки відбувається в 3 етапи:

- Спочатку на дошку розставляються всі фігури, які гравець бачить. Щоб це виконати, вони перебираються і для кожної такої фігури

знаходиться найближча фігура на попередній дошці, яка дорівнює розглянутій. Ця найближча фігура і ставиться на місце видимої.

- Далі заповнюються обов’язкові клітинки, тобто ті, які не видно, але на яких точно знаходиться ворожа фігура. Таке можливо, коли пішак не може виконати хід вперед через туман. Для таких клітинок також шукається найближча ворожа фігура на попередній дошці. Ця найближча фігура і ставиться на розглянуте місце.
- Далі потрібно розташувати у тумані всі фігури супротивника, які ще не були розташовані. Для цього перебираються всі такі фігури і для кожної такої фігури знаходиться найближча клітинка, яка є туманом і яка ще є порожньою. На цю клітинку і ставиться розглянута фігура.

Щоб уникнути детермінізму, при знаходженні довжин від однієї клітинки до іншої до шуканої довжини додається випадкове число, що за модулем менше одиниці. Таким чином алгоритм не буде повторювати одні і ті самі ходи.

2.13. PossibleBoardList

Алгоритм `PossibleBoardList` підтримує масив дошок possibleBoards, які можливі на даний момент. Кожен хід, як супротивника, так і власний, цей масив оновлюється так, щоб задовольняти поточним умовам. Спочатку масив складається з єдиної дошки, що зберігає початкову позицію. Таким чином, кожен хід

(кожен виклик функції `GetMove(b0, b)`) обробка масиву складається з трьох етапів:

- Спочатку з масиву видаляються ті дошки, які при погляду з точки зору алгоритму не перетворюються на b0;
- Потім шукаються всі можливі ходи для дошок з масиву, які перетворюють їх на дошки, які при погляді з точки зору алгоритму видимі як b. Всі отримані дошки записуються в масив, з якого потім видаляються дублікати дошок. Якщо на цьому етапі не знайдено жодної дошки, яка може виявитися справжньою, то така дошка генерується згідно з алгоритмом, описаним у попередньому розділі.
- Після цього обирається хід, який буде робити алгоритм. Далі на всіх дошках, що зберігаються в масиві, робиться цей хід. Таким чином, в масиві знову будуть зберігатися дошки, що теоретично можуть виявитися справжніми.

Для того, щоб визначити хід, для дошок з масиву викликається алгоритм для класичних шахів `MyStockfish`. Таким чином, для кожного ходу будується список оцінок позиції після цього ходу. Далі оцінкою ходу вважається найгірша з отриманих оцінок. Обирається хід, для якого ця найгірша оцінка буде найкращою.

Для цього алгоритму можна вручну встановлювати обмеження по часу на хід; причому половина цього часу буде йти на пошук позицій (на другий з наведених вище етапів), а інша половина — на аналіз позицій за допомогою `MyStockfish`. В турнірі, описаному в

наступному розділі, часовому обмеженню було встановлене значення півсекунди.

РОЗДІЛ 3. АНАЛІЗ ТА ПОРІВНЯННЯ АЛГОРИТМІВ

3.1. Реалізація гри алгоритмів один з одним

Для того, щоб симулювати гру двох (можливо, однакових) алгоритмів було розроблено функцію PlayGame, яка приймає на вхід вказівники на два об’єкти класу Player і повертає результат гри:

- `-2`, якщо білі спробували виконати неможливий хід;
- `-1`, якщо чорні взяли короля білих;
- `0`, якщо гра завершилася внічию;
- `1`, якщо білі взяли короля чорних;
- `2`, якщо чорні спробували виконати неможливий хід.

Таким чином, результат додатній, якщо партія завершилася перемогою білих, від’ємний, якщо перемогли чорні, та нульовий, якщо сталася нічия.

Також доступна можливість зберегти партію в файл, для цього існує варіант функції PlayGame з додатковим рядковим параметром, у який можна передати назву вихідного файлу.

3.2. Розрахунок рейтингу Ело

Рейтинг Ело — це широко використовувана система вимірювання та порівняння відносних рівнів майстерності гравців у змагальних іграх, зокрема в шахах, але також застосовується і в інших видах спорту та іграх. Названа на честь свого винахідника, Арпада

Ело, угорсько-американського професора фізики і шахового майстра, рейтингова система Ело надає кількісну оцінку результативності гравця і дозволяє проводити змістовні порівняння між гравцями різних рівнів майстерності.

Рейтингова система Ело базується на концепції очікуваної та фактичної результативності. Кожному гравцеві присвоюється початковий рейтинг, який зазвичай базується на його попередніх результатах або на попередньо визначеному стартовому значенні. Коли два гравці з різними рейтингами змагаються один проти одного, система розраховує очікувану ефективність кожного гравця на основі їхніх відповідних рейтингів. Розрахунок враховує різницю в рейтингах між двома гравцями і призначає прогнозовану ймовірність перемоги для кожного гравця.

Після гри фактичний результат (перемога, поразка або нічия) порівнюється з очікуваним. Якщо гравець перевершує очікуваний результат, його рейтинг зростає. І навпаки, якщо гравець не досягає очікуваного результату, його рейтинг знижується. Величина зміни рейтингу залежить від різниці в рейтингах і результату гри. Більші коригування рейтингу відбуваються, коли фактичний результат значно відхиляється від очікуваного.

Формально, нехай грають два гравці, A і B, що мають рейтинги R_A і R_B відповідно. Щоб оновити рейтинг гравця A, обчислюється очікувана кількість очок, яку він би набрав у цій грі, за формулою $E_A = \frac{1}{1+10^{\frac{R_B - R_A}{400}}}$. Після цього розглядається S_A — фактично набрана

кількість очок, яку набрав A. Новий рейтинг гравця A буде рівний $R_A + K \cdot (S_A - E_A)$. Аналогічно оновлюється рейтинг гравця B. Коефіцієнт K може бути різним для різних видів спорту та відрізків рейтингу, наприклад, в шахах він рівний 10, 20 або 40 в залежності від рейтингу гравця, його віку та кількості проведених ним ігор.

У випадку комп'ютерних алгоритмів можна змусити алгоритми зіграти круговий турнір (тобто кожен алгоритм зіграє з кожним деяку кількість партій). Далі рейтинг можна розраховувати таким чином:

- Покласти значення рейтингу всіх алгоритмів рівним нулеві;
- Застосувати результати турніру для оновлення рейтинга: перерахувати рейтинг за формулою, наведеною вище, для всіх партій турніру одночасно;
- Повторювати попередній крок, поки сумарна зміна рейтингу перевищує наперед задане мале число;
- Покласти рейтинг одного з алгоритмів рівним нулеві, і додати його рейтинг до рейтингу всіх інших алгоритмів.

В нашому випадку алгоритми будуть грати один з одним по 100 партій кожним кольором, після чого рейтинг алгоритму RandomMove буде покладений рівним нулеві та розраховані рейтинги інших алгоритмів.

3.3. Результати

На рис. 14 наведено таблицю, що відображає результати турніру. Зліва можна побачити список алгоритмів, аналогічний список наведено зверху. У клітинці на перетині i-го рядку та j-го стовпця основної частини таблиці міститься інформація про результати партій i-го алгоритму проти j-го: світла частина клітинки відповідає за 100 партій, у яких алгоритм i грав білими, а темна частина клітинки відповідає за 100 партій, у яких алгоритм i грав чорними. Числа в цих клітинках — це сумарна кількість очок, які набрав алгоритм i за відповідні партії. Зокрема, так як у кожній партії розігрується одне очко, то сума значень білої частини клітинки (i, j) та темної частини клітинки (j, i) завжди дорівнюватиме 100.

N	Ім'я	1	2	3	4	5	6	7	8	9	10	Сума
1	RandomMove		16	1	0	2	0	4	0	1	0	36
			8	1	0	1	0	2	0	0	0	
2	CaptureKing	92		3	13	8	1	22	1	8	0	306
		84		7	12	8	1	39	1	5	1	
3	CaptureLargest	99	93		51	43	37	87	27	22	2	921
		99	97		50	30	39	94	37	14	0	
4	CaptureWithLargestDifference	100	88	50		41	6	84	31	6	0	793
		100	87	49		30	7	80	27	6	1	
5	Supporter	99	92	70	70		47	88	50	23	3	1062
		98	92	57	59		54	83	52	23	2	
6	PositionaryEvaluator	100	99	61	93	46		99	39	37	2,5	1187
		100	99	63	94	53		100	53	48,5	0	
7	NeuralNetworkPlayer	98	61	6	20	17	0		4	2	4	440,5
		96	78	13	16	12	1		5	4	3,5	
8	SimpleStockfish	100	99	63	73	48	47	95		34	2	1140,5
		100	99	73	69	50	61	96		28	3,5	
9	PossibleBoardGenerator	100	95	86	94	77	51,5	96	72		8	1359,5
		99	92	78	94	77	63	98	66		13	
10	PossibleBoardList	100	99	100	99	98	100	96,5	96,5	87		1754,5
		100	100	98	100	97	97,5	96	98	92		

Рисунок 15: Повна турнірна таблиця

На рис. 15 наведено код для розрахунку рейтингу алгоритмів. Варто зауважити, що для великих значень параметру K значення рейтингу можнуть утворювати незбіжну послідовність [10], в такому разі параметр K потрібно зменшити.

```
double K = 1.0;
double rating[nmax];
double new_rating[nmax];
double Iteration() // returns total change
{
    for (int i=0; i<n; i++)
        new_rating[i] = rating[i];
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            if (i!=j)
    {
        double ea = 1.0/(1+pow(10, (rating[j]-rating[i])/400));
        new_rating[i] += K*(mr[i][j].wins*(1-ea) + mr[i][j].draws*(0.5-ea) + mr[i][j].loses*(0-ea));

        double eb = 1.0/(1+pow(10, (rating[i]-rating[j])/400));
        new_rating[j] += K*(mr[i][j].loses*(1-eb) + mr[i][j].draws*(0.5-eb) + mr[i][j].wins*(0-eb));
    }
    double res = 0;
    for (int i=0; i<n; i++)
    {
        res += abs(rating[i] - new_rating[i]);
        rating[i] = new_rating[i];
    }
    return res;
}
void CalcRating()
{
    for (int i=0; i<n; i++)
        rating[i] = 0;
    while (Iteration() > 1e-9){}
}
```

Рисунок 16: Код для обчислення рейтингу

У таблиці 2 наведена узагальнена інформація про алгоритми. Підрахована сумарна кількість очок та виміряно середній час на хід.

Алгоритм	Кількість очок	Оцінка рейтингу	Середній час на хід, мс
RandomMove	36	0	<1
CaptureKing	306	388,39	<1
CaptureLargest	921	839,62	<1
CaptureWithLargestDifference	793	764,03	<1
Supporter	1062	919,93	<1
PositionaryEvaluator	1187	992,05	<1
NeuralNetworkPlayer	440,5	514,8	10
SimpleStockfish	1140,5	964,88	10
PossibleBoardGenerator	1359,5	1100,75	12
PossibleBoardList	1754,5	1525,73	475

Таблиця 2: Узагальнена інформація про алгоритми

ВИСНОВКИ

У роботі досліджено варіацію шахів з неповною інформацією, також відому як шахи втемну або шахи з туманом війни.

Реалізовано 10 алгоритмів для гри в такий модифікований варіант шахів, кожен з яких реалізує спільний інтерфейс на мові програмування C++.

Проведено турнір між алгоритмами для їх порівняння і визначення відсотку перемог кожного алгоритму з кожним.

За формулами обчислення рейтингу Ело було обчислено орієнтовний рейтинг Ело всіх реалізованих алгоритмів.

Застосовано наступні методи програмування:

- Жадібний алгоритм (алгоритм `CaptureLargest`);
- Генетичний алгоритм (алгоритм `PositionaryEvaluator`);
- Використання штучної нейронної мережи (алгоритм `NeuralNetworkPlayer`);
- Дерево пошуку і правило прийняття рішень «мінімакс» (алгоритм для гри в класичні шахи `MyStockfish`).

Код всіх алгоритмів можна знайти у відкритому GitHub репозиторії: https://github.com/KostasKostil/Chess-Fog-of-War.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

[1] Jens Bæk Nielsen, “Darkness chess.” https://www.chessvariants.com/incinf.dir/darkness.html

[2] “Fog of war chess - grandmaster strategy!.” https://www.chess.com/blog/Illingworth/fog-of-war-chess-grandmaster-strategy

[3] “Dark chess: The art of war strategy guide.” https://docs.google.com/document/d/1hzyB6aoexikb3RnC1RU53m44-YldHfct8Belce_gyjg/pub

[4] [Online]. Available: https://www.chess.com/variants/fog-of-war

[5] [Online]. Available: http://www.brainking.com/

[6] [Online]. Available: http://www.itsyourturn.com/

[7] [Online]. Available: https://www.schemingmind.com/

[8] [Online]. Available: http://ajaxplay.com/

[9] [Online]. Available: http://www.goldtoken.com/

[10] S. J. Adrien Krifa Florian Spinelli, “On the convergence of the elo rating system for a bernoulli model and round-robin tournaments.” [Online]. Available: https://hal.science/hal-03286065/document