МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ПРИКЛАДНОЇ МАТЕМАТИКИ ТА МОДЕЛЮВАННЯ
СКЛАДНИХ СИСТЕМ

# КВАЛІФІКАЦІЙНА РОБОТА

тема роботи:

## МОДЕЛЮВАННЯ АНТАГОНІСТИЧНИХ ІГОР МЕТОДАМИ ПОБУДОВИ І ОПТИМІЗАЦІЇ «ШТУЧНОГО ІНТЕЛЕКТУ»

Завідувач кафедри — д.ф.-м.н., проф. Лисенко О. В.

Керівник роботи — к. ф.-м. н. Сушко Т. С.

Виконавець: студент факультету ЕлІТ,
гр. ПМ-41
В.В. Шурига

Затверджено на засіданні кафедри прикладної математики та моделювання складних систем від "25" жовтня 2017 року., протокол №3

Суми – 2018

# РЕФЕРАТ

Дипломна робота складається з 33с., 35 малюнків, 16 джерел, 1 додатку.

**Актуальність теми.** Пошук нових рішень у сфері штучного інтелекту, спрямований на досягнення необхідної якості прийняття рішень, мотивує до розширення спектру алгоритмів та їх комбінацій. Для вирішення даних питань найчастіше використовуються нейронні мережі, евристичні алгоритми, дерева можливих рішень. Евристичний алгоритм дозволяє більш швидко отримати бажаний результат, жертвуючи точністю, тому в даній роботі розглядується саме він. Найпопулярнішими сферами застосування є відеоігри, голосовий пошук, автономні авто, пропозиція товарів, тощо.

Таким чином, проблема побудови якісного алгоритму штучного інтелекту, є актуальною.

**Мета і задачі дослідження.**

Метою кваліфікаційної роботи є розробка мультиплатформенного додатку, а саме, антагоністичної гри на базі шахматної дошки, який повинен використовувати один з алгоритмів штучного інтелекту. Використовувати будемо мову програмування Java та бібліотеку Swing для побудови графічного інтерфейсу. Для досягнення поставленої мети потрібно:

- розглянути основні алгоритми побудови штучного інтелекту;
- визначити переваги та недоліки кожного з них, сферу застосування;
- здійснити вибір алгоритму більш відповідного до сфери ігор та адаптувати його під поставлену задачу;

КЛЮЧОВІ СЛОВА: АНТАГОНІСТИЧНА ГРА, ШТУЧНИЙ ІНТЕЛЕКТ, МІНІМАКС, ДЕРЕВО РІШЕНЬ, ЕВРИСТИКА.

# ЗМІСТ

# ВСТУП

Штучний інтелект захоплює уяву людей протягом багатьох років, але більшість з них не розуміють, що вони користуються їм майже кожний день. Більш за все, це відбувається через те, що уявляючи штучний інтелект, ми очікуємо побачити робота, який поводить себе як людина. Насправді, нас оточує незліченна кількість складних інструментів штучного інтелекту, котрі були створені для того щоб полегшити наше життя.

**Актуальність теми.** Пошук нових рішень у сфері штучного інтелекту, спрямований на досягнення необхідної якості прийняття рішень, мотивує до розширення спектру алгоритмів та їх комбінацій. Для вирішення даних питань найчастіше використовуються нейронні мережі, евристичні алгоритми, дерева можливих рішень. Евристичний алгоритм дозволяє більш швидко отримати бажаний результат, жертвуючи точністю, тому в даній роботі розглядується саме він. Гарними прикладами застосування штучного інтелекту є:

**Голосовий пошук.** Більшість сучасних мобільних операційних систем, мають своїх голосових асистентів. Наприклад у Android це Google Assistant, iOS – Siri, Windows – Cortana. Також гарним представником є голосовий асистент компанії Amazon – Alexa. Також, сучасним методом вводу та пошуку є голосовий ввід. Програма обробляє людський голос і на виході ми отримаємо текст який містить відповідь на запитання, що теж є інструментом штучного інтелекту.

**Відеоігри.** Майже кожна людина, хоча б раз у житті, грала у відеогру, яка мала суперника який непогано протистояв їй. Найвідомішим прикладом є шахи, алгоритми які вони використовують складають конкуренцію навіть гросмейстерам. Майже будь-яка гра має штучний інтелект. Деякі з них імітують поведінку людини дуже чітко, зацікавлюючи гравців.

**Автономні автівки.** Тенденції розвитку автомобілебудування сприяють до використання штучного інтелекту і в цій сфері. Водії

далекобійники будуть замінені на системи які будуть використовувати штучний інтелект, що дозволить використовувати їх 24 години на добу без загрози безпеці дорожнього руху навіть в ночі. Також системи можуть передбачати аварійні ситуації, такі як сонливість водія або неочікуваного пішохода на дорозі та рятувати людські життя, що є досить важливо.

**Пропозиція товарів.** Великі мережі магазинів мають дисконтні картки, за якими вони визначають покупця. Аналізуючи базу всіх покупців та їх вподобання, системи штучного інтелекту можуть рекомендувати або робити знижки на товари які дійсно їм потрібні.

Таким чином, проблема побудови якісного алгоритму штучного інтелекту, є актуальною.

**Мета і задачі дослідження.**

Метою кваліфікаційної роботи є розробка мультиплатформенного додатку, а саме, антагоністичної гри на базі шахматної дошки, який повинен використовувати один з алгоритмів штучного інтелекту. Використовувати будемо мову програмування Java та бібліотеку Swing для побудови графічного інтерфейсу. Для досягнення поставленої мети потрібно:

- розглянути основні алгоритми побудови штучного інтелекту;
- визначити переваги та недоліки кожного з них, сферу застосування;
- здійснити вибір алгоритму більш відповідного до сфери ігор та адаптувати його під поставлену задачу;

# 1. АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРИ

«Антогоністична гра - це гра з двома гравцями, які мають прямо протилежні інтереси. Формально, ця протилежність (антагоністичність), виявляється в тому, що при переході від однієї ситуації до іншої збільшення (зменшення) виграшу одного гравця тягне за собою зменшення (збільшення) виграшу іншого. Таким чином, сума виграшів гравців в будь-якій ситуації в антагоністичних іграх стала (зазвичай, можна вважати, що вона дорівнює нулю). Тому, антагоністичні ігри називають також іграми двох осіб з нульовою сумою (іноді — нульовими іграми)» [1]

Проаналізуємо сфери застосування та складність штучного інтелекту за такими критеріями:

- Інформація про ситуацію
- Навколишнє середовище
- Простір можливих дій
- Дії

Традиційні настільні ігри (легкий):

- Інформація про ситуацію - повна (знаємо де стоїть кожна фігура)
- Навколишнє середовище - статичне (фігури не рухаються самі по собі)
- Простір можливих дій - дискретне (правила визначені та зрозумілі)
- Дії – детерміновані - (результат ходу – фігура в потрібному місті)

Класичні відеоігри (середній):

- Інформація про ситуацію - обмежена (вороги не знають де ви)
- Навколишнє середовище - динамічне (все безперервно змінюється)
- Простір можливих дій - дискретне (правила визначені та зрозумілі)
- Дії – детерміновані - (результат ходу – фігура в потрібному місті)

Робототехніка (складний):

- Інформація про ситуацію - обмежена (сенсори)

- Навколишнє середовище - динамічне (все безперервно змінюється)
- Простір можливих дій - неперервне (неможливо передбачити що трапиться)
- Дії – імовірнісні - (дії можуть закінчитися невдачею)

Наразі можемо виділити 3 методи реалізації штучного інтелекту [2].

## 1.1 Системи на основі правил

Найпростішою формою штучного інтелекту є система на основі правил. Така система найбільш віддалена від класичного поняття інтелекту. Набір заздалегідь заданих правил та алгоритмів імітує поведінку ігрових об'єктів. Наприклад, виконання якоїсь дії при певній умові (якщо гравець пішов вліво – ворог вимикає світло, або правило що говорить ігровому об'єкту рухатися по колу) Такі системи не є дуже складними та не складають конкуренції людині. Але досить широко застосовуються через простоту реалізації та досить непогано справляються з завданням [3].

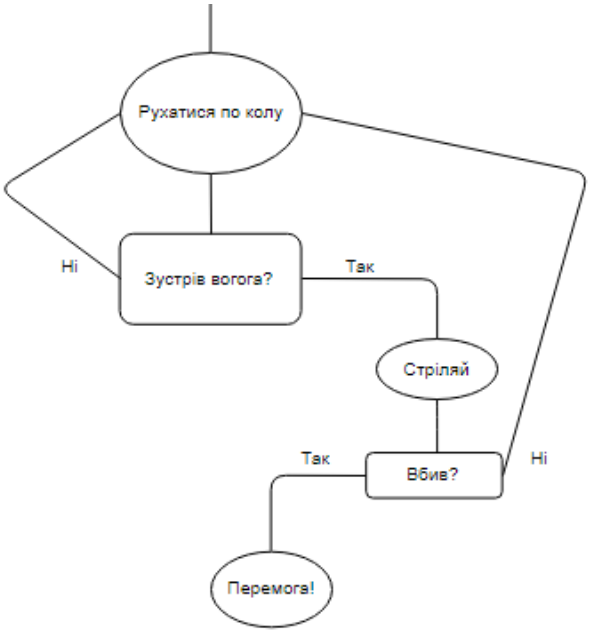


Рисунок 1.1 – Система на основі правил

Прикладами реалізації таких систем є всім відома гра «Packman», або система яка на основі певних даних про людину, вирішує можна видавати їй кредит чи ні.

## 1.2 Системи на основі дерев можливих рішень

Більш складною формою штучного інтелекту є системи які прораховують всі (або майже всі) можливі варіанти розвитку подій і роблять вибір на основі цієї інформації.

Для кожного можливого ходу прораховується оцінка. Вона відрізняє його від інших можливих ходів, чим вища оцінка – тим більший шанс на перемогу. Чим точніше ми опишемо функцію оцінки ходу, тим більш складним буде суперник в грі. Також ступінь складності можна регулювати глибиною розкриття дерева рішень [4].

Використовуючи ходи з найвищими оцінками система може складати досить серйозну конкуренцію людини у грі шашок або шах. Якщо любителя така система виграти зможе, то професіонала ні. Число можливих партій у шахах - $10^{123}$. Дуже складно прорахувати ціле дерево рішень для такої комбінації, до того ж це дуже довго, а ми хочемо грати в гру зараз і не чекати 2-3 години на хід суперника. Для часткового вирішення цієї проблеми використовуються метод альфа-бета відсічень. Головна ідея якого - припинити оцінювати хід, якщо є доказ того, що цей хід гірший, ніж оцінений раніше. Такі ходи не потребують подальшої оцінки. Тим самим ми зменшуємо кількість можливих вузлів та збільшуємо швидкість роботи.

Приклад структури дерева та роботу алгоритму можна побачити на Рисунку 1.2

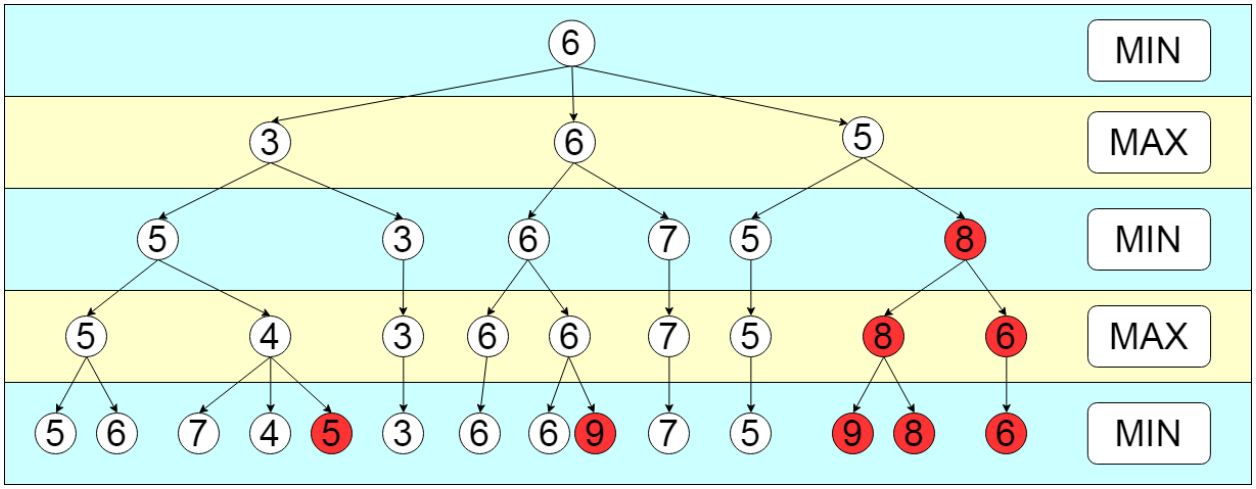


Рисунок 1.2 – Системи на основі дерева рішень

Червоним кольором визначені вузли дерева, які були відсічені від основного дерева пошуку завдяки Альфа-Бета відсіченням. Розглянемо роботу системи на прикладі алгоритму мінімакс. Задача якого, обрати оптимально-мінімальну оцінку для запропонованого дерева рішень на Рис.2 Алгоритм починає роботу з нижнього шару, обираючи для кожної гілки дерева мінімальних елемент, який відповідає найкращому варіанту розвитку подій [5]. Піднімаючись на шар вище, він вже обирає максимальний елемент, який вже відповідає найгіршому варіанту розвитку події. Тим самим, правильно підібравши оцінки для кожного вузла, можна досягти оптимального вибору майже для будь якої сфери діяльності [6].

### 1.3 Системи на основі нейронних мереж

Однією з найскладніших реалізацій штучного інтелекту є системи на основі нейронних мереж. Найчастіше використовують нейронну мережу яка використовує навчання з підкріпленням. Розглянемо приклади її застосування на прикладі ігор. Така система не потребує інформації про правила гри, тощо. Вона слідкує за тим що відбувається, пробує робити кроки та аналізує їх. Це може бути вдалий або невдалий хід в залежності від рахунку, місцезнаходження інших фігур та їх кількості (якщо мова йде про шахи). Мережа запам’ятовує ситуації які принести їй результат та

намагається повторити їх. Аналогічно, вона старається уникнути ситуацій які були не в її користь [7]. Таким чином, якщо залишити таку мережу грати саму з собою на 4-8 годин для кластеру, або 2-3 дні для звичайного ПК, вона навчиться на мільйонах помилок та зможе конкурувати з професіоналами.

До мінусів даної реалізації можна віднести необхідність в обчислювальної потужності, складність та повільність реалізації. Перевагами є швидкість роботи та «розумність» інтелекту побудованого за такою системою [8].

В 2017 році підрозділ компанії Google під назвою DeepMind розробив штучний інтелект для гри «Го» - AlphaGo. Складність цієї гри в тому що вона нараховує $10^{360}$ можливих комбінацій. До 2017 року не існувало програм які могли б отримати перемогу у професійного гравця «Го». AlphaGo використовувала навчання з підкріпленням та грала сама з собою 3 дні. За цей було проведено близько 5-ти мільйонів партій, що дозволило програмі перший раз в історії отримати перемогу у лідера світового незалежного рейтингу Go Ratings гравця Кэ Цзе. [9]

# 2. ВИБІР АЛГОРИТМУ. АДАПТАЦІЯ ПІД ПОСТАВЛЕНУ ЗАДАЧУ.

Для початку опишемо правила для гри для якої будемо адаптувати алгоритм:

1. Ходити можна тільки по діагоналі (як в шашках).
2. Фігура лисиці може робити ходи назад та вперед (вгору або вниз).
3. Фігури вовків можуть робити ходи тільки вперед (вниз).
4. Фігура лисиці ходить першою.
5. Пропускати ходи неможна.
6. Фігура лисиці перемагає тоді, коли досягне однієї з верхніх клітин (там де розташовані вовки при старті гри).
7. Фігури вовків отримують перемогу тоді, коли вони оточили або притиснули синю фігуру так, що у неї немає змоги робити хід та дістатися верхніх клітин.

Для реалізації проставленої задачі кращим методом для реалізації є дерево можливих рішень на основі мінімаксу та оптимізацію за допомогою альфа-бета відсічень, тому що він є достатньо швидким та цікавим у реалізації, та забезпечує необхідну складність гри.

Для простоти викладки матеріалу, оцінювати пріоритетність ходу будемо цілими числами (округлення в меншу сторону). Визначимо, що за фінальний хід (коли фігура лисиці дістанеться однієї з верхніх клітин) даємо 50 балів.

Розробимо першу функцію оцінки ходу. Чим вище знаходиться фігура лисиці, тим більше у неї шансів на перемогу. Це буде спонукати її рухатися вгору. Функція буде вертати номер рядка на якому знаходиться червона фігура, та матиме вигляд:

$$f_1 = row\_number(Vi)$$

Де *(Vi)* – конкретне положення фігури. Складність O(1)

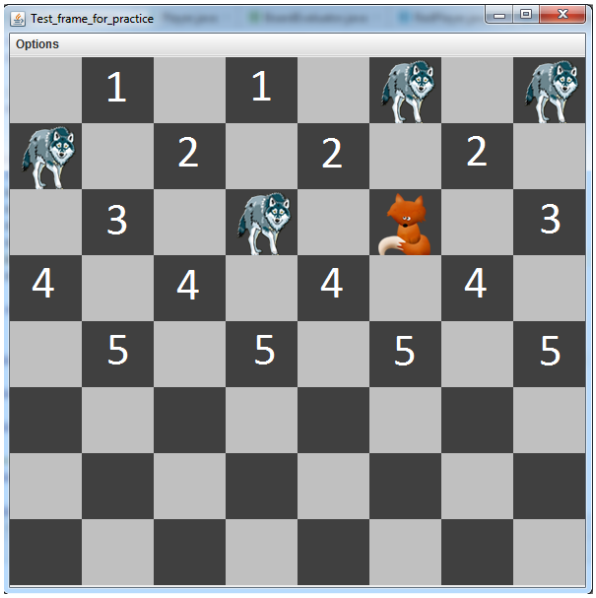


Рисунок 2.1 – Приклад функції оцінки $f_1$

Якщо обмежитися однією функцією, тоді фігура лисиці буде просто рухатися вгору, незважаючи на фігури вовків та потрапляти в пастки. Наступна функція оцінки вирішить цю проблему.

Розробимо другу функцію оцінки ходу. Ймовірність перемоги фігури лисиці вища тоді, коли їй потрібно зробити меншу кількість ходів до мети, при нерухомих червоних фігурах.

$$f_2 = distance\_to\_top(Vi)$$

Складність цієї функції оцінки – O(n), де n – кількість клітинок. *(Vi)* – конкретне положення фігури. Фактично, це пошук в ширину. Розглянемо приклад на Рисунку 2.2

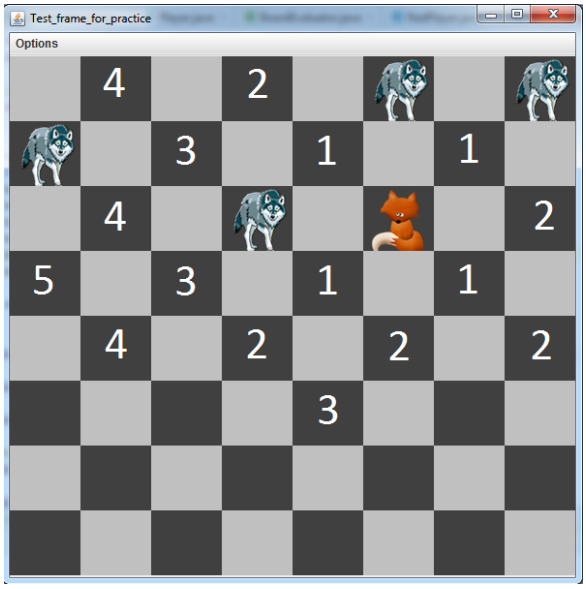


Рисунок 2.2 - Приклад функції оцінки $f_2$

Цифрами кількість ходів для перемоги. Зрозуміло що в даній ситуації мінімальна кількість ходів для перемоги – 2, саме цей хід отримає найвищу оцінку. Зауважимо, що переоцінка відбувається з кожним кроком, оскільки червоні фігури теж рухаються [10]. Ця оціночна функція надає можливість обходити сині фігури найкоротшим шляхом. Для синіх фігур все досить просто. Їх шанс на перемогу вищий тоді, коли він менших у червоної фігури, отже наші функції будуть працювати і з синіми фігурами також.

Фінальні оцінки будуть задаватися клітинці за наступною формулою:

$$f(Vi) = 0 - (f_2(Vi) - 0.5 * f_1(Vi))$$

Значення 0.5 виступає множником, який регулює значимість функції оцінки $f_1$. Для підвищення точності загальної оцінки, можна додати ще безліч функцій оцінок з множниками, які будуть регулювати їх значимість.

Хід буде обиратися за формулою:

$$\max(f(Vi_1), f(Vi_2) \dots f(Vi_k)$$

Фінальний результат оцінок на Рисунку 2.3

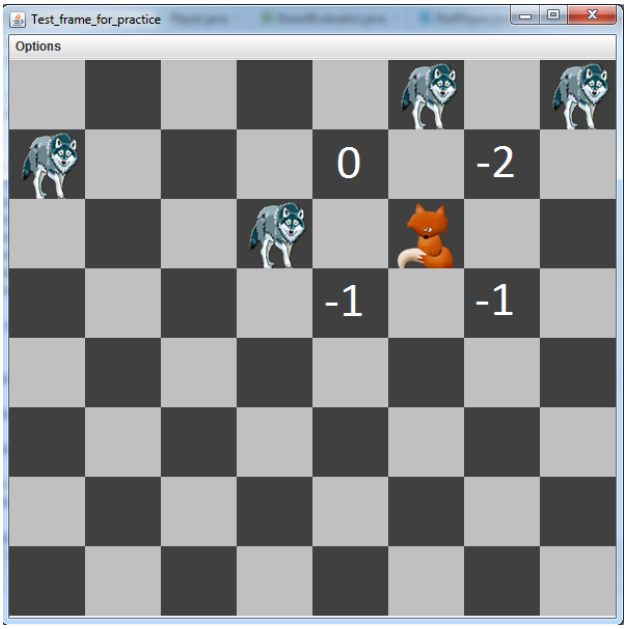


Рисунок 2.3 – Фінальні оцінки ходу

Ми проаналізувавши тему роботи, адаптували більш відповідний алгоритм до правил нашої гри, розробили 2 функції оцінки [11]. Дерево рішень для конкретного випадку на Рис.2.3 буде мати вигляд:

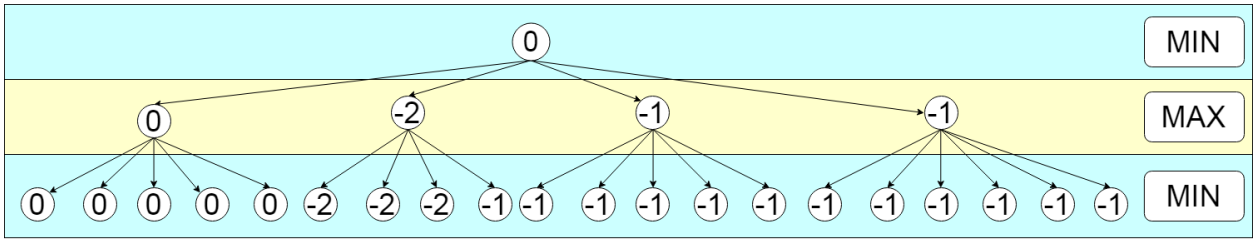


Рисунок 2.4 – Дерево рішень для ходу на Рис 2.3.

В теорії, лисиця має обрати хід з оцінкою 0. Наразі є вся необхідна інформація для написання програмного коду та впровадження алгоритму в програму.

# 3. ПРОГРАМНА РЕАЛІЗАЦІЯ

## 3.1 Створення графічного інтерфейсу

Розглянемо інтерфейс програми IntelliJ IDEA. Для того щоб створити проект перейдемо до налаштувань File > New > Project.

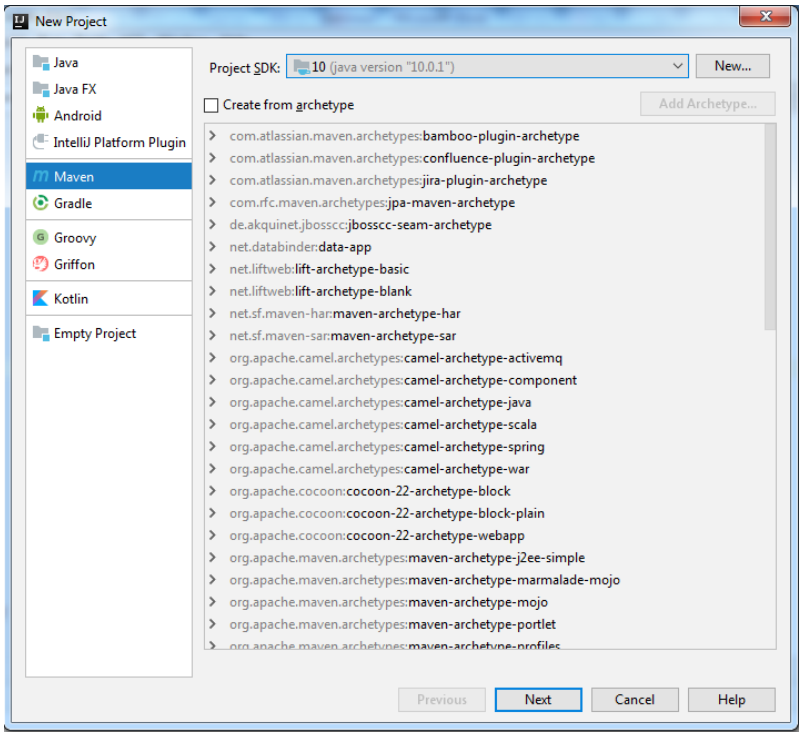


Рисунок 3.1 – Вікно створення нового проекту.

З списку вибираємо Maven проект, задаємо ім’я і тд [12]. Створимо клас MainFrame за допомогою якого будемо моделювати графічний інтерфейс додатку. Імпортуємо необхідні бібліотеки для роботи з GUI, а саме:

```
import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.*;
import java.util.List;
import java.util.concurrent.ExecutionException;
```

Рисунок 3.2 – Підключення бібліотек для створення GUI

Описуємо клас вкладений клас Board, який відповідає за представлення шахової дошки.

```
private class BoardPanel extends JPanel {

    final List<CellOfBoard> boardCells;

    public BoardPanel() {
        super(new GridLayout( rows: 8, cols: 8));
        this.boardCells = new ArrayList();
        for (int i = 0; i < 64; i++){
            final CellOfBoard cell = new CellOfBoard( board: this, i);
            this.boardCells.add(cell);
            add(cell);
            setPreferredSize(BOARD_PANEL_DIMENSION);
            validate();
        }
    }

    public void drawBoard(Board board){
        removeAll();
        for(CellOfBoard cellOfBoard : boardCells) {
            cellOfBoard.drawCell(board);
            add(cellOfBoard);
        }
        validate();
        repaint();
    }
}
```

Рисунок 3.3 – Реалізація класу BoardPanel

Конструктор класу викликається кожного разу, коли створюється новий об’єкт типу BoardPanel. В конструкції for створюємо 64 об’єктів типу CellOfBoard які будуть реалізувати кожну клітину на шаховій дошці. Константою MAIN_FRAME_DIMENTION задаємо розмір головного вікна 600 на 600 пікселів. Також клас містить метод drawBoard, який відображає створену дошку на екрані.

Абстрактний клас Cell містить 2 статичних класи EmptyCell та OccupiedCell, яким відповідають клітині без фігури та на якій стоїть якась фігура. Абстрактні класи допомагають уникнути дублювання коду, та є елементом об'єктно орієнтованого програмування.

```
public abstract class Cell {

    protected final int cellNumber;

    private static final Map<Integer, EmptyCell> emptyCellsMapCache = createAllEmptyCells();

    public abstract boolean isCellOccupied();

    public abstract Figure getFigure();

    public int getCellNumber() { return this.cellNumber; }

    public Cell(int cellNumber) { this.cellNumber = cellNumber; }

    private static Map<Integer,EmptyCell> createAllEmptyCells() {...}

    public static Cell createCell(final int cellNumber, final Figure figure){...}
```

Рисунок 3.4 – Реалізація абстрактного класу Cell

Також він містить методи isCellOccupied – вертає true або false в залежності від того чи стоїть фігура на клітині. Метод getFigure повертає об’єкт фігури яка стоїть на клітині. getCellNumber – вертає номер клітини на якій стоїть.

```
    public static final class EmptyCell extends Cell {

        private EmptyCell(int cellNumber) { super(cellNumber); }

        @Override
        public String toString() { return "-"; }

        @Override
        public boolean isCellOccupied() { return false; }

        @Override
        public Figure getFigure() { return null; }
    }

    public static final class OccupiedCell extends Cell {

        private Figure figureOnCell;

        private OccupiedCell(int cellNumber, Figure figureOnCell) {...}

        @Override
        public String toString() {...}

        @Override
        public boolean isCellOccupied() { return true; }

        @Override
        public Figure getFigure() { return figureOnCell; }
    }
}
```

Рисунок 3.5 – Реалізація класів EmptyCell та OccupiedCell

Метод isRedWin перевіряє стан кінця гри, та вертає true у разі перемоги, або false якщо гра ще не зкінчена.

```
public boolean isRedWin() {
    for (Figure figure : getRedFigures()){
        if (figure.getPosition() == 1 ||
                figure.getPosition() == 3 ||
                figure.getPosition() == 5 ||
                figure.getPosition() == 7){
            return true;
        } else return false;
    }
    return false;
}
```

Рисунок 3.6 – Метод isRedWin

Далі створимо класи фігур. Гра містить сині фігури (фігури вовків) та червону фігуру (фігуру лисиці). Створимо 2 класи які будуть відтворювати їх поведінку. Використаємо механізм ООП — наслідування. Класи BlueFigure та RedFigure реалізовують абстрактрий клас Figure разом з методом calculateLegalMoves. Важливою змінною є масив MOVE_COORDINATER, він задає на скільки клітинок потрібно додати для того щоб отримати можливе переміщення фігури. Так як наше поле представлене масивом чисел з 0 до 63, то якщо ми додамо 7 або 9 до поточного значення, то отримаємо координату наступної нижньої діагональної клітини [13].

```
public abstract class Figure {

    protected final FigureType figureType;
    protected final int position;
    protected final Alliance figureAlliance;
    protected final boolean isFirstMove;

    public Figure(FigureType figureType, int position, Alliance figureAlliance) {
        this.figureType = figureType;
        this.position = position;
        this.figureAlliance = figureAlliance;
        this.isFirstMove = false;
    }
```

Рисунок 3.7 – Клас Figure

```java
public class BlueFigure extends Figure {

    private static int[] MOVE_COORDINATES = {7, 9};

    public BlueFigure(int position, Alliance figureAlliance) { super(position, figureAlliance); }

    @Override
    public Collection<Move> calculateLegalMoves(Board board) {

        final List<Move> legalMoves = new ArrayList<~>();

        for (final int valueToMove: MOVE_COORDINATES) {
            final int candidateToMove = this.position + valueToMove;

            if (Utility.isValidCell(candidateToMove)) {
                if(isFirstColumnTrouble(this.position, valueToMove)
                        || isEigthColumnTrouble(this.position, valueToMove)
                        || isEightRowTrouble(this.position, valueToMove)) {
                    continue;
                }
                final Cell candidateDestinationCell = Board.getCell(candidateToMove);

                if (candidateDestinationCell.isCellOccupied()){
                    // Nothing to do
                } else {
                    legalMoves.add(new Move(board,  movedFigure: this, candidateToMove));
                }
            }
        }
        return Collections.unmodifiableCollection(legalMoves);
    }
```

Рисунок 3.8 – Реалізація класу фігури вовків.

```java
public class RedFigure extends Figure {

    private static int[] MOVE_COORDINATES = {-9, -7, 7, 9};


    public RedFigure(final int position, final Alliance figureAlliance) {
        super(position, figureAlliance);
    }

    @Override
    public Collection<Move> calculateLegalMoves(Board board) {

        final List<Move> legalMoves = new ArrayList<~>();

        for (final int valueToMove: MOVE_COORDINATES) {
            final int candidateToMove = this.position + valueToMove;

            if (Utility.isValidCell(candidateToMove)) {

                if(isFirstColumnTrouble(this.position, valueToMove)
                        || isFirstColumnAndEightRowTrouble(this.position, valueToMove)
                        || isEigthColumnTrouble(this.position, valueToMove)){
                    continue;
                }

                final Cell candidateDestinationCell = Board.getCell(candidateToMove);
```

Рисунок 3.9 – Реалізація класу фігури  лисиці.

Оскільки лисиця має рухатись вгору и вниз, до координат руху добавимо -7 та -9 Методи calculateLegaleMoves визначають можливі ходи для кожної фігури, враховуючи положення суперника.

Реалізуємо метод assignImage, для того щоб придати фігурі картинку. Потрібен формат png, оскільки він має прозорість, яка потрібна.

```
private void assignImage(final Board board){
    this.removeAll();
    if(board.getCell(this.cellId).isCellOccupied()) {
        if(board.getCell(this.cellId).getFigure().getFigureAlliance().toString() == "RED") {
            try {
                BufferedImage image = ImageIO.read(new File( pathname: "src/main/java/gui/img/red.png"));
                add(new JLabel(new ImageIcon(image)));
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else if (board.getCell(this.cellId).getFigure().getFigureAlliance().toString() == "BLUE") {
            try {
                BufferedImage image = ImageIO.read(new File( pathname: "src/main/java/gui/img/blue.png"));
                add(new JLabel(new ImageIcon(image)));
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Рисунок 3.10 – Реалізація методу assignImage

Розставимо фігури на наше віртуальне поле в методі createStandartBoard, та перевіримо результат.

```
public static Board createStandartBoard(){
    final Builder builder = new Builder();
    builder.setFigure(new BlueFigure( position: 1, Alliance.BLUE));
    builder.setFigure(new BlueFigure( position: 3, Alliance.BLUE));
    builder.setFigure(new BlueFigure( position: 5, Alliance.BLUE));
    builder.setFigure(new BlueFigure( position: 7, Alliance.BLUE));
    builder.setFigure(new RedFigure( position: 60, Alliance.RED));
    builder.setMoveMaker(Alliance.RED);

    return builder.build();
}
```

Рисунок 3.11 – Заповнення дошки фігурами

На виході отримаємо наступний результат:

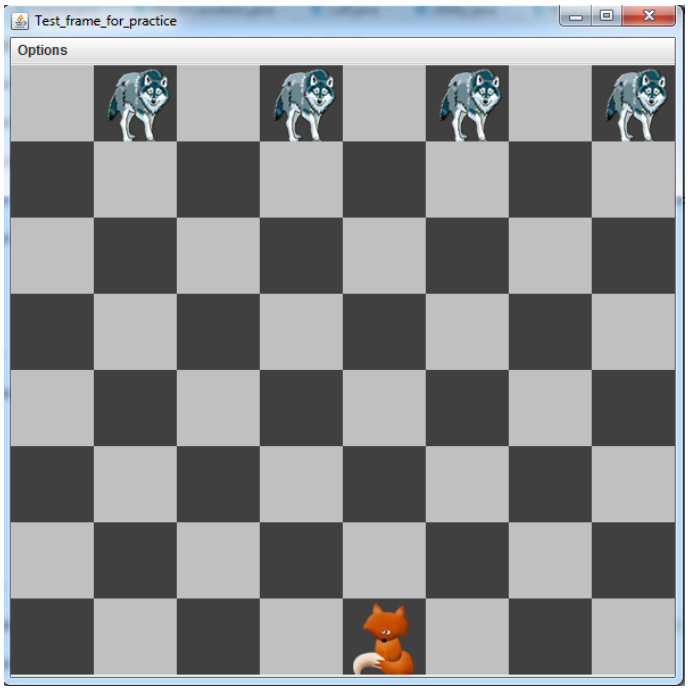


Рисунок 3.12 – Головне вікно програми

## 3.2 Написання алгоритму

Тепер реалізуємо наш адаптований алгоритм до додатку. Для цього створимо клас MinMax, який буде описувати роботу алгоритму мінімакс та побудови дерева рішень. Він буде імплементувати інтерфейс MoveStraregy, для того, щоб у майбутньому можна будо дуже легко змінити реалізацію методу штучного інтелекту, або перемикати програму на один з них, якщо їх буде декілька [14].

```
public interface MoveStrategy {
    Move execute(Board board);
}
```

Рисунок 3.13 – Інтерфейс MoveStrategy

Сам клас має вигляд:

```
public class MinMax implements MoveStrategy {

    private final BoardEvaluator boardEvaluetor;
    private final int depth;

    public MinMax(int depth) {
        this.boardEvaluetor = new Evaluator();
        this.depth = depth;
    }
```

Рисунок 3.14 – Реалізація класу MinMax

Імплементований метод execute,  з якого і починається будова дерева та оцінка його вузлів (кожного можливого ходу). Він буде вертати найкращий з усіх можливих ходів для фігури. Його реалізацію можна побачити на Рис 22.

```
@Override
public Move execute(Board board) {

    Move bestMove = null;

    int hieghestValue = Integer.MIN_VALUE;
    int lowestValue = Integer.MAX_VALUE;
    int currentValue;

    for(final Move move : board.getCurrentPlayer().getLegalMoves()){
        final MoveTransition moveTransition = board.getCurrentPlayer().makeMove(move);
        if(moveTransition.getMoveStatus().isDone()){
            currentValue = board.getCurrentPlayer().getAlliance().isRed() ?
                    min(moveTransition.getTransitionBoard(), depth: depth - 1) :
                    max(moveTransition.getTransitionBoard(), depth: depth - 1);

            if(board.getCurrentPlayer().getAlliance().isRed() && currentValue >= hieghestValue){
                hieghestValue = currentValue;
                bestMove = move;
            } else if (board.getCurrentPlayer().getAlliance().isBlue() && currentValue <= lowestValue) {
                lowestValue = currentValue;
                bestMove = move;
            }
        }
    }
    return bestMove;
}
```

Рисунок 3.15 – Метод execute

Метод min відповідає за шар дерева, де обирається мінімальний елемент, та передається до методу max, який в свою чергу рекурсивно викликає метод min. Параметром функції є змінна depth, яка визначає глибину розкриття дерева. Таким чином ми маємо можливість регулювати складність гри. При виборі глибини більшої за 5, робота алгоритму може займати деякий час.

```
public int min(final Board board, final int depth){
    if (depth == 0 || isEndGame(board)){
        return this.boardEvaluetor.evaluete(board, depth);
    }

    int lowestValue = Integer.MAX_VALUE;
    for(final Move move : board.getCurrentPlayer().getLegalMoves()){
        final MoveTransition moveTransition = board.getCurrentPlayer().makeMove(move);
        if(moveTransition.getMoveStatus().isDone()){
            final int currentValue = max(moveTransition.getTransitionBoard(), depth: depth - 1);
            if(currentValue <= lowestValue){
                lowestValue = currentValue;
            }
        }
    }
    return lowestValue;
}
```

Рисунок 3.16 – Реалізація методу min

```
public int max(final Board board, final int depth){
    if (depth == 0 || isEndGame(board)){
        System.out.println("DEPTH = " + depth);
        return this.boardEvaluetor.evaluete(board, depth);
    }

    int highestValue = Integer.MIN_VALUE;
    for(final Move move : board.getCurrentPlayer().getLegalMoves()){
        final MoveTransition moveTransition = board.getCurrentPlayer().makeMove(move);
        if(moveTransition.getMoveStatus().isDone()){
            final int currentValue = min(moveTransition.getTransitionBoard(), depth: depth - 1);
            if(currentValue >= highestValue){
                highestValue = currentValue;
            }
        }
    }
    return highestValue;
}
```

Рисунок 3.17 – Реалізація методу max

Наступним етап – створити клас та методи для оцінки ходів за нашими формулами. Клас Evaluator  містить у собі функції оцінки [14]. Метод row_height містить умовний оператор if який перевіряє умову перемоги фігури лисиці, у і разі позитивної відповіді виставляє 50 балів.

```
public final class Evaluator implements BoardEvaluator {

    @Override
    public int evaluete(final Board board, final int depth) {
        int result = 0 - scorePlayer(board, board.redPlayer(), depth);
        System.out.println("Evaluator             : " + result);
        return result;
    }

    private int scorePlayer(final Board board, final Player player, final int depth) {
        return distanceToWin(board) - (row_height(board) / 2);
    }

    private int row_height(final Board board){

        ArrayList redFigure = new ArrayList();

        for (Figure figure : board.getRedFigures()){
            redFigure.add(figure.getPosition());
        }

        if ((int)redFigure.get(0) == 1 || (int)redFigure.get(0) == 3 || (int)redFigure.get(0) == 5 || (int)redFigure.get(0) == 7){
            return 50;
        }

        return (int)(((int)redFigure.get(0) / 8)) + 1;
    }
```

Рисунок 3.18 – Реалізація класу Evaluator

Для роботи додатку залишилося зв'язати графічний інтерфейс з алгоритмом. Зробимо це за допомогою класів AIThink та GameAI, які будуть передавати керування грою нашому алгоритму одразу після ходу людини, завдяки SwingWorker [16].

```
private static class AIThink extends SwingWorker<Move, String> {

    private AIThink(){}

    @Override
    protected Move doInBackground() throws Exception {
        final MoveStrategy minimax = new MinMax( depth: 2);
        final Move bestMove = minimax.execute(MainFrame.get().getGameBoard());
        return bestMove;
    }

    @Override
    public void done() {
        try {
            final Move bestMove = get();
            MainFrame.get().updateComputerMove(bestMove);
            MainFrame.get().updateGameBoard(MainFrame.get().getGameBoard().getCurrentPlayer().makeMove(bestMove).getTransitionBoard());
            MainFrame.get().getBoardPanel().drawBoard(MainFrame.get().getGameBoard());
            MainFrame.get().moveMadeUpdate(PlayerType.COMPUTER);

            if(MainFrame.get().getGameBoard().isRedWin()){
                JOptionPane.showMessageDialog(MainFrame.get().board,  message: "FOX WIN!");
            }

        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

Рисунок 3.19 – Реалізація класу AIThink

```
private static class GameAI implements Observer{

    @Override
    public void update(Observable o, Object arg) {

        if (MainFrame.get().getSetupPanel().isAIPlayer(MainFrame.get().getGameBoard().getCurrentPlayer())){
            final AIThink think = new AIThink();
            think.execute();
        }
    }
}
```

Рисунок 3.20 – Реалізація класу GameAI

# РЕЗУЛЬТАТИ

Спробуємо зіграти в гру, та проаналізуємо поведінку штучного інтелекту. Точкою входу є клас Test.

```
public class Test {

    public static void main(String[] args) {

        MainFrame.get().show();
        Board board = Board.createStandartBoard();
        System.out.println(board);

        Evaluator evaluator = new Evaluator();

        evaluator.evaluete(board, depth: 4);
    }
}
```

Рисунок 3.21 – Клас Test

Додаток грає за лисицю, людина за вовків. Нижче приведено скріншоти роботи програми для кожного ходу.

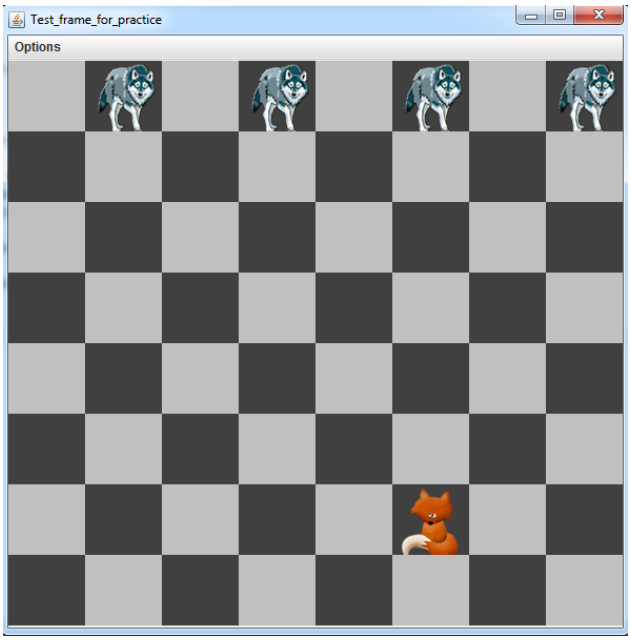


Рисунок 3.22 – Перший хід

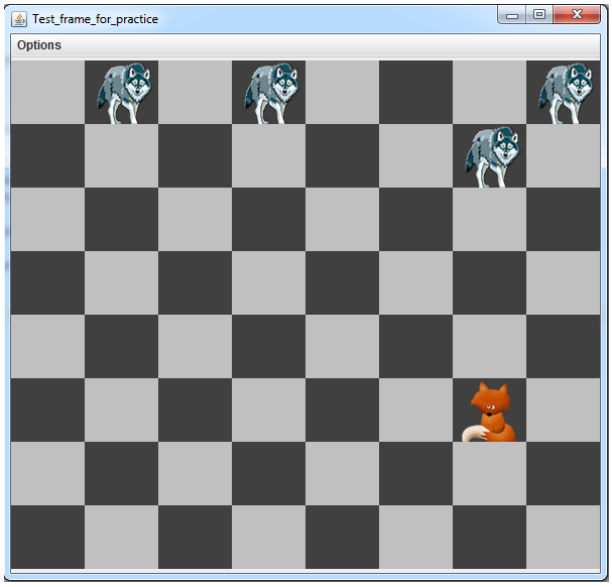


Рисунок 3.23 – Третій хід

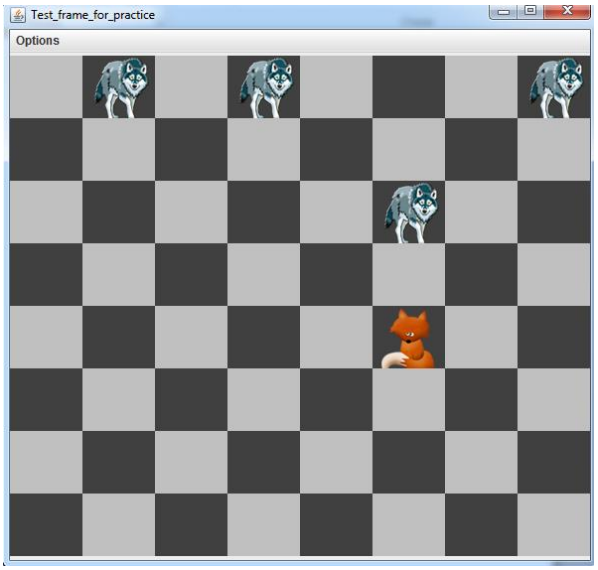


Рисунок 3.24 – П'ятий хід

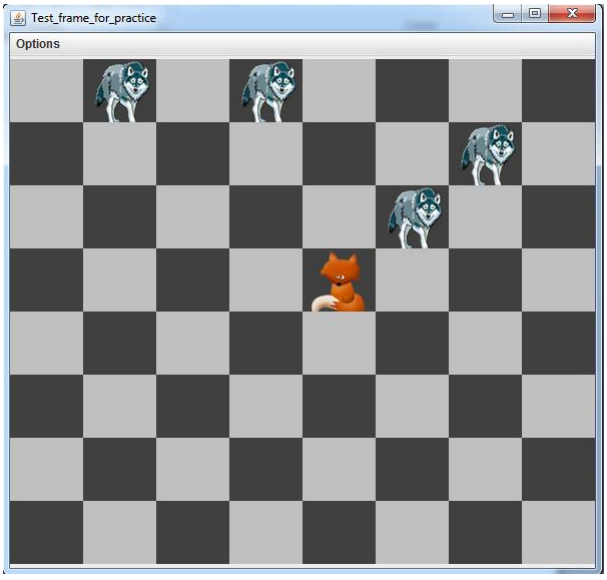


Рисунок 3.25 – Сьомий хід

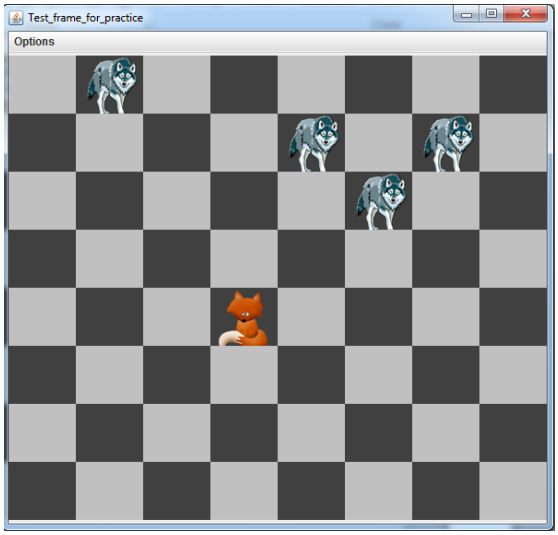


Рисунок 3.26 – Дев'ятий хід

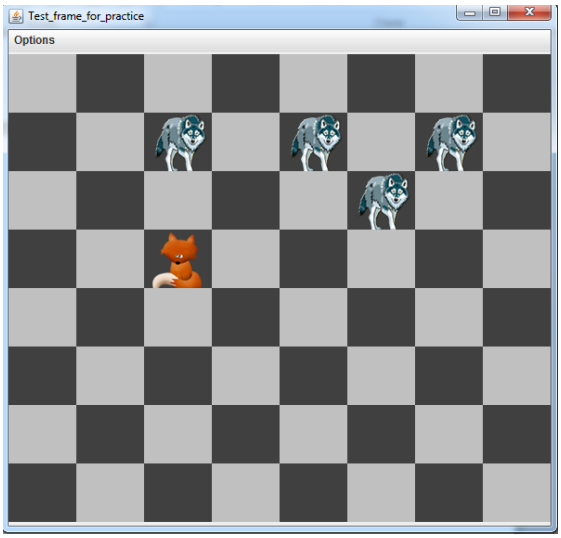


Рисунок 3.27 – Одинадцятий хід

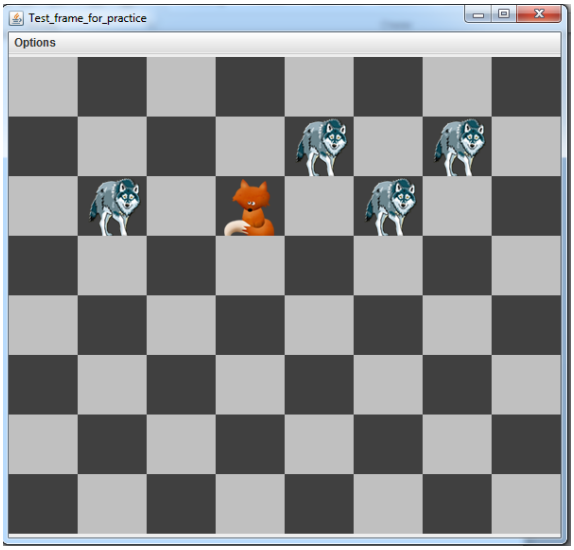


Рисунок 3.28 – Тринадцятий хід

Після 13-го ходу очевидна перемога лисиці, так як вовки можуть ходити тільки вниз. На 17-му ході штучний інтелект отримав перемогу.

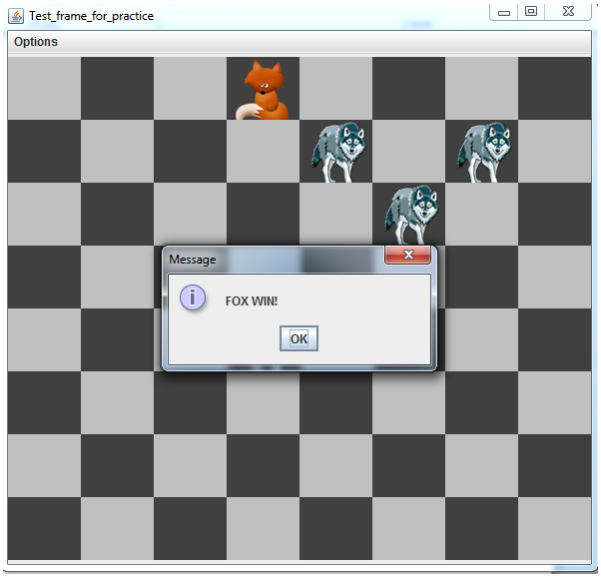


Рисунок 3.29 – Сімнадцятий хід

# ВИСНОВКИ

В дипломній роботі розроблено додаток, а саме гра, на мові програмування Java з системою штучного інтелекту, яка може протистояти людині. Був використаний патерн програмування MVC, задля спрощення подальшої модифікації або удосконалення проекту. Досліджені основні методи побудови штучного інтелекту, сфери де вони застосовуються та наведені приклади застосування. На основі цих знань реалізовано алгоритм мінімаксу на базі дерев можливих рішень. Також досліджена предметна сфера застосування, і розроблено дві евристичні функції, які оцінюють ходи у грі.

З результатів роботи можна зробити висновок що метод підібрано вдало, оскільки з поставленими задачами він добре справляється. Для унеможливлення перемоги вовків слід обробити виняткові ситуації, такі як поступове наближення вовків в одні лінію. Також є можливість реалізації даного додатку за допомогою нейронних мереж, але це значно ускладнює алгоритм та швидкість розробки. Очевидно, вибір залежить від необхідної точності прийняття рішень.

Загалом викладено практичну реалізацію та пояснення, за якими можна відтворити систему до будь-якої предметної сфери, щоб задовольнити свої інтереси.

# ПЕРЕЛІК ПОСИЛАНЬ


1. https://uk.wikipedia.org/wiki/Антагоністична_гра
2. Ясницкий Л. Н. Введение в искусственный интеллект. — М.: Издат. центр «Академия», 2005. — 176 с.
3. Компьютер учится и рассуждает (ч. 1) // Компьютер обретает разум = Artificial Intelligence Computer Images / под ред. В. Л. Стефанюка. — Москва: Мир, 1990. — 240 с.
4. Ананій В. Левітін. Глава 10. Обмеження потужності алгоритмів: Дерева прийняття рішення // Алгоритми: введення в розробку й аналіз = Introduction to The Design and Analysis of Aigorithms.
5. https://uk.wikipedia.org/wiki/Мінімакс
6. Демьянов В. Ф., Малоземов В. Н. Введение в минимакс. — М.: Наука, 1972. — 368 с.
7. Тадеусевич Рышард, Боровик Барбара, Гончаж Томаш, Леппер Бартош. Элементарное введение в технологию нейронных сетей с примерами программ / Перевод И. Д. Рудинского. — М.: Горячая линия — Телеком, 2011. — 408 с.
8. Лорьер Ж.-Л. Системы искусственного интеллекта. — М.: Мир, 1991. — 568 с.
9. https://uk.wikipedia.org/wiki/AlphaGo
10. https://habr.com/post/163681/
11. Васин А. А., Морозов В. В. Теория игр и модели математической экономики. — М., 2005.
12. http://maven.apache.org/plugins/index.html
13. Иван Портянкин. Swing: Эффектные пользовательские интерфейсы, 2-е издание. — 2-е. — Санкт-Петербург: «Лори», 2011. — С. 600.
14. Иан Грэхем. Объектно-ориентированные методы. Принципы и практика = Object-Oriented Methods: Principles & Practice. — 3-е изд. — М.: «Вильямс», 2004. — С. 880

15. http://wjournal.com.ua/evristichnij-metod-navchannja.html
16. Антони Синтес. Освой самостоятельно объектно-ориентированное программирование за 21 день = Sams Teach Yourself Object-Oriented Programming in 21 Days. — М.: «Вильямс», 2002. — С. 672

# Додаток А

Програмна реалізація класу Utility, який відповідає за ініціалізацію змінних додатку.

```
public class Utility {

    public static final boolean[] FIRST_COLUMN = initColumn(0);
    public static final boolean[] EIGHT_COLUMN = initColumn(7);

    public static final boolean[] FIRST_ROW = initRow(0);
    public static final boolean[] SECOND_ROW = initRow(8);
    public static final boolean[] THIRD_ROW = initRow(16);
    public static final boolean[] FOURTH_ROW = initRow(24);
    public static final boolean[] FIFTH_ROW = initRow(32);
    public static final boolean[] SIXTH_ROW = initRow(40);
    public static final boolean[] SEVENTH_ROW = initRow(48);
    public static final boolean[] EIGHT_ROW = initRow(56);

    public static final int NUMBER_OF_CELLS = 64;
    public static final int NUMBER_OF_CELLS_IN_ROW = 8;

    private static boolean[] initColumn(int columnNumber) {
        final boolean[] column = new boolean[NUMBER_OF_CELLS];
        for(int i = 0; i < column.length; i++) {
            column[i] = false;
        }
        do {
            column[columnNumber] = true;
            columnNumber += NUMBER_OF_CELLS_IN_ROW;
        } while (columnNumber < NUMBER_OF_CELLS);

        return column;
    }

    private static boolean[] initRow(int rowNunber) {
        final boolean[] row = new boolean[NUMBER_OF_CELLS];
        for(int i = 0; i < row.length; i++) {
            row[i] = false;
        }
        do {
            row[rowNunber] = true;
            rowNunber++;

        } while (rowNunber % NUMBER_OF_CELLS_IN_ROW != 0);

        return row;
    }

    public static boolean isValidCell(int coordinate) {
        return coordinate >= 0 && coordinate < 64;
    }
}
```