КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра теорії та технології програмування

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

ЗАСТОСУВАННЯ ТЕОРІЇ ІГОР У ЗАДАЧАХ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ МОДЕЛЮВАННЯ ПЕРЕМОЖНОЇ ГРИ В ШАШКИ

Виконав студент 4-го курсу

Дмитро ГОЛІЙЧУК _____

Науковий керівник :

професор, доктор фізико-математичних наук

Микола НІКІТЧЕНКО _____

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

Студент _____

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування

«__»_____ 2023 р., протокол №__

Завідувач кафедри

Микола НІКІТЧЕНКО _____

Київ – 2023

РЕФЕРАТ

Обсяг роботи 43 сторінки, 33 ілюстрації, 7 джерел посилань, 1 додаток.

ТЕОРІЯ ІГОР, МІНІМАКС, АЛЬФА-БЕТА ВІДСІЧЕННЯ, ГЕНЕТИЧНИЙ АЛГОРИТМ, МАШИННЕ НАВЧАННЯ, НАВЧАННЯ З ПІДКРІПЛЕННЯМ, ГРА «МІЖНАРОДНІ ШАШКИ», JMESPATH

Об’єктом роботи є алгоритми теорії ігор та машинного навчання, гра «Міжнародні шашки».

Метою роботи є дослідження алгоритмів теорії ігор для моделювання переможної гри у шашках.

Методи розроблення: математичне моделювання, алгоритми теорії ігор. Інструменти розроблення: мова програмування Python, середовище програмування PyCharm.

Результати роботи: проаналізовано алгоритми теорії ігор, які застосовні для моделювання переможної гри у шашках, розроблено застосунок з використанням вибраного алгоритму, проаналізовано роботу застосунку.

Розроблений застосунок може використовуватися гравцями у шашки для створення більш складної для них гри, у теорії ігор та машинному навчанні для розвитку наявних способів моделювання виграшних ходів у шашках.

ЗМІСТ

ВСТУП

Оцінка сучасного стану об’єкта дослідження. Гра «Міжнародні шашки» є однією з найбільш популярних настільних ігор з великою кількістю можливих ходів та стратегій [7]. Ця гра досліджується як гравцями з метою розвитку їхньої здатності грати на більш високому рівні, так і галуззю штучного інтелекту з метою розвитку методів моделювання виграшних ходів.

Актуальність роботи та підстави для її виконання. Дослідження гри «Міжнародні шашки» та ефективне моделювання найкращих ходів розширить уже наявні способи вирішення даної задачі та допоможе гравцям у створенні більш цікавої та складної для них гри, сприятиме їхній здатності будувати кращі стратегії гри. Популярні способи моделювання виграшних ходів у шашках побудовані з використанням методів машинного навчання, а у даній роботі буде досліджено методи вирішення цієї задачі з використанням теорії ігор.

Мета й завдання роботи. Метою даної роботи є дослідити алгоритми теорії ігор та вибрати найбільш відповідний для пошуку виграшних ходів у грі «Міжнародні шашки». З використанням вибраного алгоритму потрібно реалізувати застосунок, що моделюватиме наступні найкращі ходи у даній грі.

Далі відбудеться тестування та аналіз застосунку, на цьому етапі буде оцінено чи вирішує реалізований застосунок поставлену початково задачу. Після аналізу буде запропоновано можливі способи удосконалення застосунку.

Об'єкт, методи й засоби розроблення. Об’єктом розроблення застосунку є процес моделювання виграшних ходів у грі «Міжнародні шашки». Буде застосовано математичне моделювання для побудови математичної моделі гри. Основна задача застосунку – моделювання виграшних ходів, буде реалізована з використанням алгоритмів теорії ігор. Застосунок буде розроблено з використанням мови програмування Python та середовища програмування PyCharm.

Можливі сфери застосування. Розроблений застосунок може використовуватися гравцями «Міжнародних шашок» для розвитку їхніх навичок гри та створення більш складних ігрових ситуацій. Також застосунок може бути використано у галузі прикладної математики для розширення наявних способів моделювання виграшних ходів у шашках.

РОЗДІЛ 1 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка задачі

Задача полягає у пошуку наступних ходів у грі «Шашки», які будуть максимізувати виграш гравця та мінімізувати виграш опонента.

Буде задаватись початкове розташування фігур на дошці і кількість ходів, які потрібно спрогнозувати. Потрібно розробити застосунок, який за заданими користувачем вхідними даними буде знаходити найкращі наступні ходи. Потрібно дослідити алгоритми теорії ігор, які можуть бути застосовані для моделювання виграшних ходів та розробити застосунок з використанням вибраного алгоритму.

1.1.1 Правила гри «Шашки»

У даній задачі під грою «Шашки» мається на увазі гра «Міжнародні шашки». Міжнародні шашки, також відомі як шашки-100, є однією з найпопулярніших варіацій шашок у світі. Розглянемо правила гри «Міжнародні шашки»:

1) Дошка: Гра проводиться на дошці 10x10 з темними і світлими клітинами. Початкове заповнення полів полягає у встановленні на них 20 чорних шашок для одного гравця та 20 білих для іншого.
2) Розташування: Гравці розташовують свої шашки на темних клітинках зліва від себе. Таким чином, ліва нижня клітинка повинна бути темною.
3) Рух шашок: Шашки рухаються діагонально на одну клітину вперед. Вони можуть рухатися тільки на темні клітини дошки. Шашка може рухатися вперед і назад, коли вона стає дамкою.
4) Биття шашок: Шашка може бити шашку противника, якщо наступна клітинка містить шашку противника, і наступна клітинка за шашкою противника порожня. Гравець може виконати послідовне биття кількох шашок за один хід.
5) Дамка: Якщо шашка доходить до крайнього ряду протилежного кінця дошки, вона стає дамкою. Дамка може рухатися по діагоналі на будь-

яку вільну клітину в межах дошки. Вона також може бити шашки противника вперед і назад.

6) Завершення гри: Гра закінчується, коли один із гравців втрачає всі свої шашки або не може виконати жодного ходу. В такому випадку інший гравець вважається переможцем.
7) Нічия в грі "Міжнародні шашки" виникає, якщо однакова позиція повторюється три рази без зміни, або якщо обидва гравці погоджуються з результатом нічиєї.

1.2 Алгоритми теорії ігор, які можуть бути використані для розв'язку задачі

В теорії ігор існує кілька алгоритмів, які можуть бути використані для прогнозування ходів у шашках. Розглянемо ці алгоритми:

Мінімаксний алгоритм: Цей алгоритм використовується для пошуку оптимального ходу у антагоністичних іграх, таких як шашки. Він розглядає всі можливі ходи гравців і оцінює їх за допомогою певної функції оцінки. Мінімаксний алгоритм враховує ходи обох гравців і обирає хід, що максимізує виграш одного гравця та мінімізує виграш іншого.

Альфа-бета відсічення: Це покращений варіант мінімаксного алгоритму, який дозволяє зменшити кількість гілок, які потрібно перебрати. Він використовує оцінки граничних значень (альфа та бета) для відсічення непотрібних гілок у дереві пошуку. Альфа представляє найкращий виграш для максимізуючого гравця, а бета - найкращий виграш для мінімізуючого гравця.

Monte Carlo Tree Search: Цей алгоритм використовує методи Монте-Карло для прогнозування ходів. Він поєднує випадковий пошук з деревом пошуку, що зберігає результати попередніх симуляцій. Даний алгоритм використовує оцінки виграшу для керування пошуком і вибором ходу. Він широко застосовується в складних іграх, де повний пошук не є практичним через великий обсяг можливих ходів.

Також для розв’язку поставленого завдання можна використати генетичний алгоритм. Основна ідея полягає в тому, що генетичний алгоритм створює популяцію рішень, які представлені у вигляді генотипів (наприклад, послідовності ходів у грі). Ці рішення оцінюються за допомогою функції оцінки, яка визначає як добре вони пристосовані до вирішення поставленої задачі. Далі застосовується схрещування та мутація для створення нової популяції можливих ходів. Після кількох ітерацій виконання алгоритму, тобто створення нових поколінь, отримуємо популяцію, що буде складатися з кращих ходів.

Окрім використання алгоритмів з теорії ігор, для розв’язку поставленої задачі можна використовувати алгоритми машинного навчання, зокрема алгоритми навчання з підкріпленням. Ці алгоритми можуть навчитися самостійно приймати рішення на основі досвіду гри з іншими гравцями або самими собою.

Далі розглянемо детальніше деякі з даних алгоритмів.

1.2.1 Мінімаксний алгоритм

Спочатку розглянемо теорему мінімаксу. Теорема мінімаксу стверджує: Для будь-якої скінченної антагоністичної гри можна знайти значення S та змішані стратегії для кожного із гравців, такі, що для кожної стратегії другого гравця, перший гравець може гарантовано отримати виграш S, і для кожної стратегії першого гравця, другий гравець може гарантовано отримати виграш (-S) [2].

Основна ідея алгоритму мінімакс полягає у врахуванні можливих ходів гравців та їхніх наслідків для визначення оптимального ходу. Розглянемо кроки застосування алгоритму мінімакс для прогнозування ходів у шашках:

1) Представлення стану гри: Спочатку потрібно представити поточний стан шашкової дошки. Це може бути зроблено у вигляді матриці або списку з позначенням положення фігур на дошці.

2) Генерація можливих ходів: Наступним кроком є генерація всіх можливих ходів, які гравець може зробити з поточного стану гри. Для кожної фігури визначаємо всі можливі ходи, які можуть бути здійснені з поточного стану гри.
3) Оцінка стану гри: Для кожного можливого ходу потрібно оцінити, наскільки вигідний цей хід для гравця. Можна використовувати різні оцінювальні функції, які враховують такі фактори, як кількість захоплених фігур, позиції фігур на дошці, можливість отримання кращого стану гри в наступних ходах тощо.
4) Рекурсивний аналіз: алгоритм мінімакс використовує рекурсивний підхід для прогнозування кращого ходу. Він виконується шляхом почергового розгляду ходів кожного гравця і розрахунку їхніх оцінок. Рекурсивний процес продовжується до досягнення кінця гри або досягнення певної глибини аналізу.
5) Вибір оптимального ходу: коли рекурсивний аналіз повертається назад до початкового стану гри, потрібно вибрати оптимальний хід для поточного гравця. Якщо гравець, для якого проводився аналіз, є максимізуючим гравцем, то обирається хід з найвищою оцінкою. Це означає, що гравець буде вибирати найкращі ходи, які максимізують його виграш. Якщо гравець, для якого проводився аналіз, є мінімізуючим гравцем, то обирається хід з найнижчою оцінкою. Оцінка обраного оптимального ходу повертається вгору по рекурсивному ланцюжку, щоб визначити найкращий хід для початкового стану гри [3].

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
maxEva= -infinity
 for each child of node do
  eva= minimax(child, depth-1, false)
  maxEva= max(maxEva,eva)        //gives Maximum of the values
return maxEva

else                          // for Minimizer player
 minEva= +infinity
 for each child of node do
  eva= minimax(child, depth-1, true)
  minEva= min(minEva, eva)        //gives minimum of the values
 return minEva
```

Рисунок 1.1 - Неформальний запис алгоритму мінімакс

1.2.2 Алгоритм альфа-бета відсічення

Алгоритм альфа-бета відсічення є оптимізованою версією алгоритму мінімакс, яка дозволяє зменшити кількість розглядуваних гілок у дереві гри, що пришвидшує процес прийняття рішень. Цей алгоритм використовує два значення, відомі як "альфа" і "бета", для відслідковування найкращих відомих максимальних та мінімальних значень оцінок. Розглянемо основні ідеї алгоритму альфа-бета відсічення.

Альфа (α) представляє найкращу відому максимальну оцінку для максимізуючого гравця, а бета (β) - найкращу відому мінімальну оцінку для мінімізуючого гравця. Спочатку α встановлюється на величину мінус нескінченність, а β - на величину плюс нескінченність.

Під час рекурсивного розгляду усіх можливих ходів, коли максимізуючий гравець розглядає хід, оцінка кожного стану порівнюється з альфою. Якщо оцінка перевищує α, то α оновлюється значенням оцінки.

Аналогічно коли мінімізуючий гравець розглядає хід, оцінка кожного стану гри порівнюється з бетою. Якщо оцінка менше, ніж β, то β оновлюється значенням оцінки.

Альфа-бета відсічення виконується шляхом відсічення гілок, що не потребують детального аналізу. Якщо при аналізі деякого вузла виявляється, що $\alpha \geq \beta$, то аналіз наступних дочірніх вузлів для даного вузла не виконується [4].

```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
  maxEva= -infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, False)
 maxEva= max(maxEva, eva)
 alpha= max(alpha, maxEva)
  if beta<=alpha
 break
 return maxEva
```

Рисунок 1.2 - Неформальний запис алгоритму альфа-бета відсічення

```
else                          // for Minimizer player
  minEva= +infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, true)
  minEva= min(minEva, eva)
  beta= min(beta, eva)
   if beta<=alpha
 break
return minEva
```

Рисунок 1.3 – продовження неформального запису алгоритму

1.2.3 Генетичний алгоритм

Основна ідея генетичного алгоритму для прогнозування ходів у шашках полягає в тому, щоб створити популяцію ходів шашок, представлених у вигляді генетичних кодів, і застосувати еволюційні оператори для відбору, схрещування та мутації, щоб знайти оптимальні стратегії.

Основні кроки генетичного алгоритму для моделювання виграшних ходів у шашках:

1) Кодування ходів: спочатку потрібно вибрати спосіб кодування можливих ходів фігур з певного стану дошки.
2) Створення початкової популяції: Випадковим чином генеруємо початкову популяцію шашкових ходів із початкового стану дошки.
3) Вибір функції оцінки: потрібно означити функцію, яка оцінює якість кожного ходу. Аналогічно до функції оцінки у алгоритмі мінімакс, дана функція може враховувати кількість фігур гравців, їхнє розташування, відстань до центру дошки та інші параметри.
4) Застосування еволюційних операторів: застосовуємо відбір, схрещування та мутацію до популяції шашкових ходів.
5) Ітеративне застосування еволюційних операторів для створення нової популяції ходів та їхня оцінка. Після виконання певної кількості ітерацій отримаємо популяцію, що буде містити найкращі ходи [5].

1.2.4 Навчання з підкріпленням

Алгоритми навчання з підкріпленням можна застосовувати для моделювання виграшних ходів у шашках шляхом тренування агента (штучної інтелектуальної системи) на основі досвіду гри. Основна ідея алгоритму полягає в тому, що агент взаємодіє з середовищем гри (у цьому випадку - шашкова дошка) і отримує певну винагороду за кожний зроблений хід. Ця винагорода представляє оцінку, за допомогою якої можна визначити, наскільки зроблений агентом хід є вигідним або ні.

Розглянемо основні етапи алгоритму навчання з підкріпленням:

1) Створення моделі агента: Початково модель агента може бути ініціалізована випадковими вагами або застосовано певну попередню модель. Ця модель буде використовуватись для прийняття рішень щодо ходів у грі.
2) Генерація тренувальних даних: Агент грає проти себе або проти інших агентів у багатьох шашкових партіях. Під час гри він збирає дані про стани гри та зроблені ходи.
3) Обчислення винагороди: Після кожного зробленого ходу агент отримує винагороду, яка оцінює, наскільки цей хід був вигідним або не вигідним для його перемоги у грі. Винагорода може бути визначена, наприклад, на основі різниці кількості шашок на дошці, позицій фігур, їхніх відстаней до центру дошки тощо.
4) Оновлення моделі агента: Агент використовує отримані дані про стани гри та винагороди для навчання моделі. Зазвичай використовується метод градієнтного спуску [6].

1.3 Висновки щодо розгляду алгоритмів

У даному розділі були розглянуті алгоритми теорії ігор, які можна використати для вирішення поставленої задачі. Детально розглянуто такі алгоритми, як мінімакс, алгоритм альфа-бета відсічення, генетичний алгоритм, навчання з підкріпленням.

Після розгляду даних алгоритмів можна стверджувати, що найкраще для моделювання переможних ходів у шашках можна використати мінімакс, альфа-бета відсічення та навчання з підкріпленням.

Генетичний алгоритм теж застосовний для вирішення даної задачі, проте потрібно чітко означити генетичні оператори - мутацію та схрещування, обрати правильний спосіб кодування можливих ходів.

Детальне порівняння розглянутих алгоритмів буде наведено у Розділі 3.

РОЗДІЛ 2 РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

2.1 Опис етапів вирішення поставленого завдання

Для вирішення поставленого завдання буде розроблено додаток з використанням алгоритму мінімакс. Для цього спочатку потрібно вибрати спосіб представлення станів дошки, ходів, реалізувати можливість зміни станів фігур на дошці відповідно до правил міжнародних шашок. Потім потрібно обрати оцінювальну функцію, яка буде визначати оцінку ходів. Основним етапом розробки буде реалізація алгоритму мінімакс, який знаходитиме найкращі ходи. Після цього необхідно вибрати спосіб представлення результатів виконання програми, зрозумілих користувачу.

Дошку доцільно буде представити у вигляді двовимірного списку, детально це описано у пункті 2.2 даного розділу.

Ходи у даній програмі буде представлено двох видів – звичайний хід та атака. У коді програми відповідно будуть використовуватися назви «move» та «attack».

Реалізацію ходів та атак відповідно до правил шашок та пошук усіх можливих ходів і атак детально описано у пунктах 2.4 та 2.5 даного розділу.

Оцінювальну функцію обрано як різницю кількості білих та чорних фігур на дошці: $F = W – B$

Алгоритм мінімакс буде реалізовано з використанням функції search з пакету jmespath, яка виконує пошук даних у структурах JSON за допомогою JMESPath-виразів.

Результат виконання програми складатиметься з трьох частин, які будуть повторюватися таку кількість разів, що відповідає кількості прогнозованих ходів.

Одна з частин відображатиме номер ходу або атаки та координати переміщення шашки, або координати знищених шашок супротивника. Друга частина виводу зображатиме стан фігур на дошці після виконання відповідного ходу.

2.2 Опис вхідних та вихідних даних

Вхідні дані

Одними із вхідних даних є двовимірний список, який відображає заповнену дошку для гри. Кожен елемент списку відповідає клітинці на дошці.

У даному випадку, дошка має розмір 10x10, тому список складається з 10 списків, кожен з яких містить 10 елементів. Кожен елемент може приймати одне з трьох значень: None, "w" або "b":

- Значення None відповідає порожній клітинці на дошці.
- Значення "w" вказує на клітинку, де знаходиться біла фігура.
- Значення "b" вказує на клітинку, де знаходиться чорна фігура.

Ось один із прикладів вхідних даних, де задано 4 білих та 5 чорних шашок:

```
filled_board = [
    [None, None, None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, "b", None, None, None],
    [None, "b", None, None, None, "b", None, None, None, None],
    [None, None, None, None, "w", None, None, None, None, None],
    [None, "b", None, None, None, None, None, None, None, "b"],
    ["w", None, None, None, None, None, None, None, "w", None],
    [None, None, None, None, None, None, None, "w", None, None],
    [None, None, None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None, None, None]]
```

Рисунок 2.1 – приклад задання вхідних даних

Далі користувач вводить кількість ходів, які потрібно спрогнозувати та колір фігур, які починають гру з даного стану.

Приклад вхідних даних:

Enter number of moves: 3

Chose color to play W or B: W

Користувач повинен ввести ціле число що відповідає кількості ходів, які прогнозуються та англійські літери `W` або `B`, які відповідають кольору фігури, що починає гру з даного стану. Регістр літер значення не має.

Вихідні дані

Вихідні дані представляють прогнозовані наступні ходи фігур.

Спочатку вказується ‘Move N’, що означає номер наступного ходу. Далі вказується вид ходу – атака (хід із побиттям шашки супротивника) або звичайний хід без побиття фігури:

- Moved from [x1, y1] to [x2, y2]
- Attacked from [x1, y1] to [[x2, y2], [x3, y3], … [xn, yn]]

При звичайному ході вказується позиція у вигляді двох координат, з якої відбувається хід та позиція у яку переміщується фігура. При атаці вказується позиція, з якої відбувається хід та список позицій шашок супротивника, які будуть знищені під час ходу.

Координати клітинок для білих та чорних фігур відрізняються, а саме: для чорних фігур координати додатні, точка відліку (0, 0) розташована в лівому верхньому куті, для білих фігур координати від’ємні, точка відліку (-1, -1) розташована в правому нижньому куті. Перша координата означає номер рядка, друга – номер фігури у даному рядку.

Наступна частина виводу програми показує стан дошки після виконання ходу.

Дошка позначається у вигляді символів ‘*’, ‘w’ та ‘b’.

‘*’ позначає клітинки дошки, у яких не містяться фігури.

‘w’ та ‘b’ позначають відповідно позицію білої та чорної фігури на дошці після виконання ходу.

За допомогою таких вихідних даних подається один із наступних ходів у грі. Залежно від введеної користувачем кількості наступних ходів, що потрібно спрогнозувати, ці частини вихідних даних повторюються відповідну кількість разів і відображають наступні ходи.

Приклад вихідних даних, що відображають наступний перший та другий хід:

Move 1:

Attacked from [-5, -10] to [[-6, -9], [-8, -9]]

Board view:

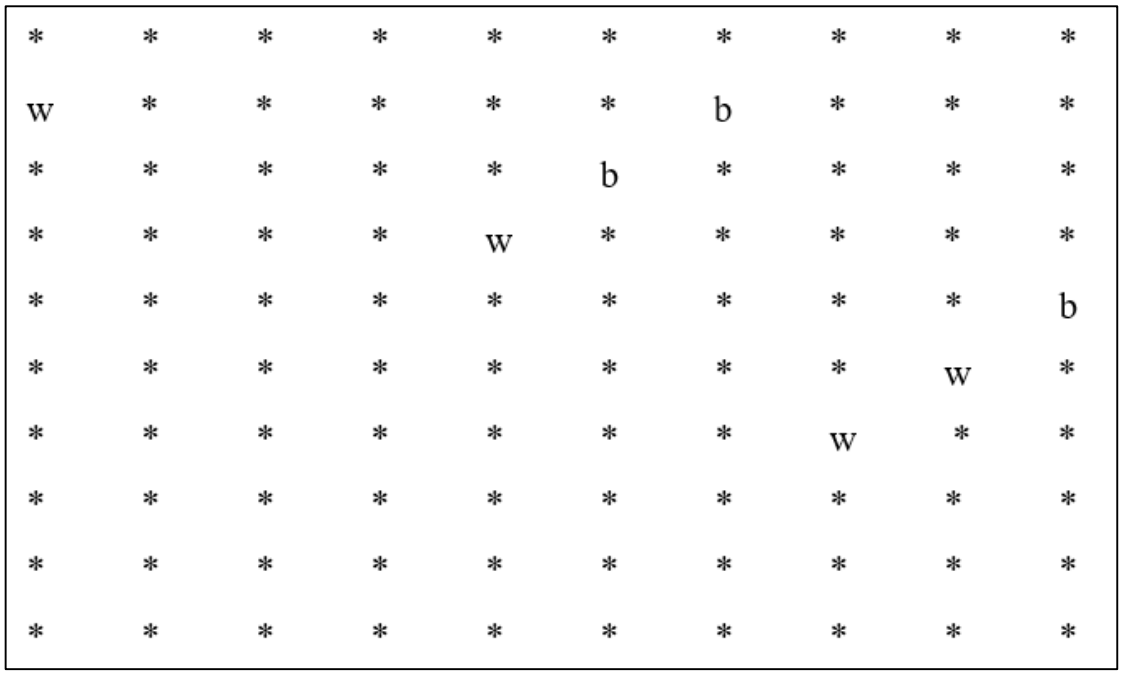

Рисунок 2.2 – стан дошки після першого ходу

Тут перший хід виконують білі фігури, що також можна зрозуміти із від’ємних координат позицій.

Move 2:

Attacked from [2, 5] to [[3, 4]]

Board view:

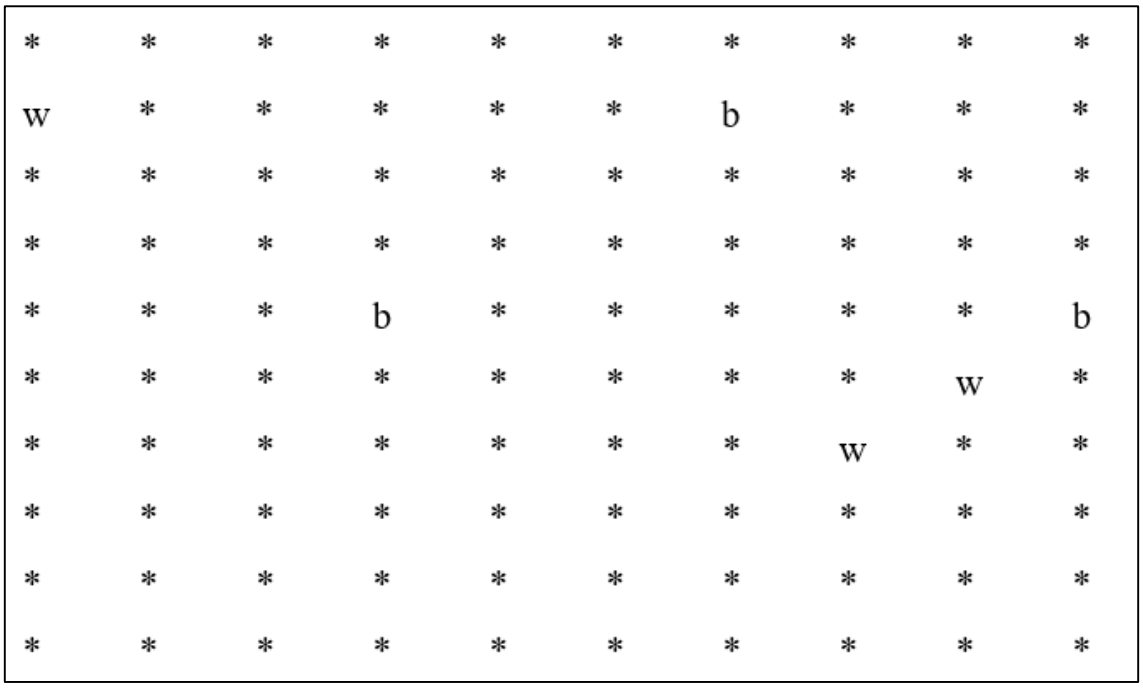

Рисунок 2.3 – стан дошки після другого ходу

2.3 Опис використаних вбудованих функцій

Найголовнішою використаною вбудованою функцією є jmespath.search(<jmespath expr>, <JSON document>).

JMESPath (JSON Matching Expression Path) - це мова запитів та бібліотека для пошуку та фільтрації даних в структурах JSON. Основна ідея JMESPath полягає у використанні рядкових виразів для опису шаблону, який визначає, які елементи потрібно вибрати зі структури JSON. За допомогою цього пакету можна виконувати різноманітні операції, такі як вибірка полів, фільтрація, сортування, перетворення та агрегація даних.

Метод jmespath.search(<jmespath expr>, <JSON document>) є основною функцією пакету jmespath в Python, яка виконує пошук даних у структурах JSON за допомогою JMESPath-виразів.

Функція jmespath.search() отримує два аргументи: JMESPath-вираз і дані, у яких потрібно виконати пошук. Вона обходить дані та застосовує JMESPath-вираз для вибірки підмножини даних, що відповідають виразу.

Функція jmespath.search() повертає результат пошуку, який може бути різного типу, залежно від JMESPath-виразу та структури даних. Наприклад, це може бути окреме значення, список, або словник [1].

2.4 Опис авторських функцій

Розглянемо функцію check_moveable_black.

Дана функція для усіх чорних шашок на дошці визначає усі можливі ходи та атаки.

Спочатку створюється порожній список points, який буде містити інформацію про можливі ходи та атаки. Далі відбувається проходження по усім клітинкам дошки і розгляд лише чорних шашок. Створюється словник point з ключами point_i, point_j, які містять координати чорної шашки, що розглядається. Ключі move_to та attack є ключами до списків позицій клітинок, на які відбувається хід або атака.

Викликаються дві допоміжні функції: check_black_move(board, i, j) та check_black_attack(board, i, j).

Вони повертають список можливих ходів та атак для шашки з координатами (i, j).

Вкінці відбувається перевірка, чи для даної шашки є хоча б один можливий хід або атака, якщо так, то додаємо словник point до списку points.

```
def check_moveable_black(board):
    points = []
    for i in range(10):
        for j in range(10):
            point = {"point_i": None, "point_j": None, "move_to": [], "attack": []}
            p = board[i][j]
            if p == "b":
                point["point_i"] = i
                point["point_j"] = j
                point["move_to"] = check_black_move(board, i, j)
                point["attack"] = check_black_attack(board, i, j)

                if point["move_to"] != [] or point["attack"] != []:
                    points.append(point)
    return points
```

Рисунок 2.4 – код функції check_moveable_black

Розглянемо функцію check_moveable_white.

Вона визначає можливі ходи та атаки для усіх білих шашок. Логіка даної функції аналогічна до check_moveable_black. Відмінність полягає у тому, що по дошці проходження відбувається з правого нижнього кута. Ця відмінність пов’язана з різними точками відліку для відслідковування позицій чорних та білих шашок. Код функції check_moveable_white наведено у Додатку А.

Розглянемо функцію check_black_move.

Функція check_black_move(board, i, j) перевіряє можливі ходи для чорної шашки з координатами (i, j) на дошці board. Ходи повертаються у вигляді списку move_to.

Використовуються конструкції try та except для перехоплення винятків IndexError, що виникають при спробі доступу до неіснуючих клітинок дошки.

Спочатку перевіряється можливість ходу вліво-вниз (i + 1, j - 1). Перевіряємо умову, чи клітинка вільна та умову, щоб не вийти за межі дошки зліва (j - 1 >= 0).

Якщо у розглянуту клітинку можна зробити хід, то словник з її координатами додається до списку move_to.

Аналогічно виконується перевірка можливості ходу у клітинку, що знаходиться справа внизу відносно даної.

```
def check_black_move(board, i, j):
    move_to = []
    try:
        if not board[i + 1][j - 1] and j - 1 >= 0:
            move_to.append({"point_i": i + 1, "point_j": j - 1})
    except IndexError:
        pass
    try:
        if not board[i + 1][j + 1]:
            move_to.append({"point_i": i + 1, "point_j": j + 1})
    except IndexError:
        pass
return move_to
```

Рисунок 2.5 – код функції check_black_move

Функція check_white_move працює аналогічно до функції check_black_move, відмінність полягає у тому, що ходи білих шашок здійснюються вліво-вверх та вправо-вверх.

```
def check_white_move(board, i, j):
    move_to = []
    try:
        if not board[i - 1][j + 1] and j - 1 <= 0:
            move_to.append({"point_i": i - 1, "point_j": j + 1})
    except IndexError:
        pass
    try:
        if not board[i - 1][j - 1]:
            move_to.append({"point_i": i - 1, "point_j": j - 1})
    except IndexError:
        pass
    return move_to
```

Рисунок 2.6 – код функції check_white_move

Розглянемо функцію check_black_attack.

Функція check_black_attack(board, i, j, passed_from=None) перевіряє усі можливі атаки для чорної шашки з координатами (i, j). Оскільки на відміну від ходів, атакувати шашка може в усі чотири діагональні до неї клітинки, то дана функція розглядає можливість атак у ці чотири напрямки.

Дана функція є рекурсивною, що пов’язано з тим, що після побиття шашки супротивника, можна виконувати атаки далі. Тому після перевірки виконання атаки у одну із можливих позицій, виконується рекурсивний пошук наступних можливих атак.

Функція check_black_attack повертає результат у вигляді списку attack, де елемент списку – словник, що містить координати позиції, з якої відбувається атака, координати позиції шашки супротивника, яка буде знищена, та координати позиції, куди переміститься шашка після атаки.

Функція check_white_attack (board, i, j, passed_from=None) визначає усі можливі атаки для білої шашки із заданими координатами. Її логіка роботи є аналогічною до функції check_black_attack(board, i, j, passed_from=None).

Розглянемо функцію make_a_move. Функція make_a_move(board, starts, moves) повертає список move_level, який представляє усі можливі стани дошки та описи ходів до заданої глибини, яка дорівнює початково введеній кількості прогнозованих ходів.

Спочатку знаходимо всі можливі ходи та атаки для усіх фігур певного кольору для наступного ходу за допомогою функції check_moveable_white або check_moveable_black.

Для кожної фігури розглядаємо можливі ходи та для кожного ходу створюємо опис ходу move_desc, який містить координати ходу, оцінку стану дошки, дошку після ходу та список наступних ходів, який створюємо порожнім. Якщо ще можна зробити ходи, то рекурсивно викликаємо дану функцію, передаючи стан дошки після виконаного ходу, у starts колір протилежної фігури, та кількість ходів зменшену на 1. Результат додаємо у список move_desc['next_moves'], який початково створено порожнім.

Далі для кожної фігури розглядаємо можливі атаки та аналогічно створюємо опис атак. Результат пошуку наступних атак та ходів додається у список move_level, який повертається функцією.

Код функцій check_black_attack, check_white_attack, make_a_move наведено у Додатку А.

Функція move(board, point_from, point_to) повертає стан дошки після виконання певного ходу, замінюючи стан клітинки із шашкою що виконує хід та стан клітинки яка порожня (у яку виконується хід).

```
        def move(board, point_from, point_to):
            board_after = [[board[i][j] for j in range(10)] for i in range(10)]
            board_after[point_from['point_i']][point_from['point_j']],
board_after[point_to['point_i']][point_to['point_j']] \
                = None, board_after[point_from['point_i']][point_from['point_j']]
            return board_after
```

Рисунок 2.7 – код функції move

Функція attack(board, points_to) повертає стан дошки після виконання певної атаки.

```
        def attack(board, points_to):
            board_after = [[board[i][j] for j in range(10)] for i in range(10)]
            for point_to in points_to:
                board_after[point_to['point_from_i']][point_to['point_from_j']], \
                board_after[point_to['point_i']][point_to['point_j']], \
                board_after[point_to['point_to_i']][point_to['point_to_j']], \
                    =                    None,                    None,
board_after[point_to['point_from_i']][point_to['point_from_j']]
            return board_after
```

Рисунок 2.8 – код функції attack

Функція get_score(board) рахує оцінку стану дошки. У даній реалізації оцінка дорівнює різниці кількості білих та чорних шашок.

```
def get_score(board):
    white_count = 0
    black_count = 0
    for i in range(10):
        white_count += board[i].count("w")
        black_count += board[i].count("b")
    return white_count - black_count
```

Рисунок 2.9 – код функції get_score

Функція minimax (tree, query, subquery) є найголовнішою в даній програмі. Вона виконує алгоритм мінімакс для пошуку найкращих наступних ходів. Функція опрацьовує дерево можливих ходів і обчислює оптимальний хід для гравця на основі оцінки цих можливих ходів та атак.

У функцію minimax передаються такі параметри:

- tree: дерево ходів, на якому виконується алгоритм
- query: вираз для вибору відповідних вузлів дерева
- subquery: вираз для вибору наступних ходів у вибраних вузлах

Функція використовує бібліотеку jmespath для виконання пошуку в структурі даних дерева.

Алгоритм minimax працює в рекурсивному режимі. Він перевіряє, чи є вузли, що відповідають запиту subquery у вибраних вузлах дерева. При виклику функції запит subquery дорівнюватиме "next_moves[*]" , тобто будуть вибиратися вузли, з яких можливі наступні ходи або атаки. Якщо такі вузли знайдені, то вони додаються до графа дерева, і рекурсивно викликається функція minimax для кожного з цих вузлів.

Далі, алгоритм формує стратегію вибору ходів, враховуючи результати оцінки кожного ходу з урахуванням ворожих ходів. Для гравця білими фігурами, обирається хід з максимальною оцінкою (max_by), а для гравця чорними фігурами - хід з мінімальною оцінкою (min_by).

Код функції minimax (tree, query, subquery) наведено у Додатку А.

2.5 Інструкція користувачу

Відповідно до описаних вище вхідних даних, спочатку користувач повинен задати початковий стан дошки у списку filled_board. Після запуску програми необхідно спочатку ввести кількість ходів, які потрібно спрогнозувати та колір шашки, яка починає гру з даного стану.

Далі програма виведе наступні стани дошки та координати позицій яких та у які переміщуються шашки під час виконання ходів та атак. Більш детальна інформація щодо представлення вхідних та вихідних даних наведена у пункті 2.2 даного розділу та у пункті 3.1 розділу «ТЕСТИ І АНАЛІЗ».

РОЗДІЛ 3 ТЕСТИ І АНАЛІЗ

У даному розділі будуть розглянуті приклади результатів роботи програми та описані можливі варіанти удосконалення роботи.

3.1 Опис результатів роботи застосунку з визначеними вхідними параметрами

Розглянемо доволі просте розташування фігур, де хід розпочнуть чорні і прогнозуються наступні два ходи.

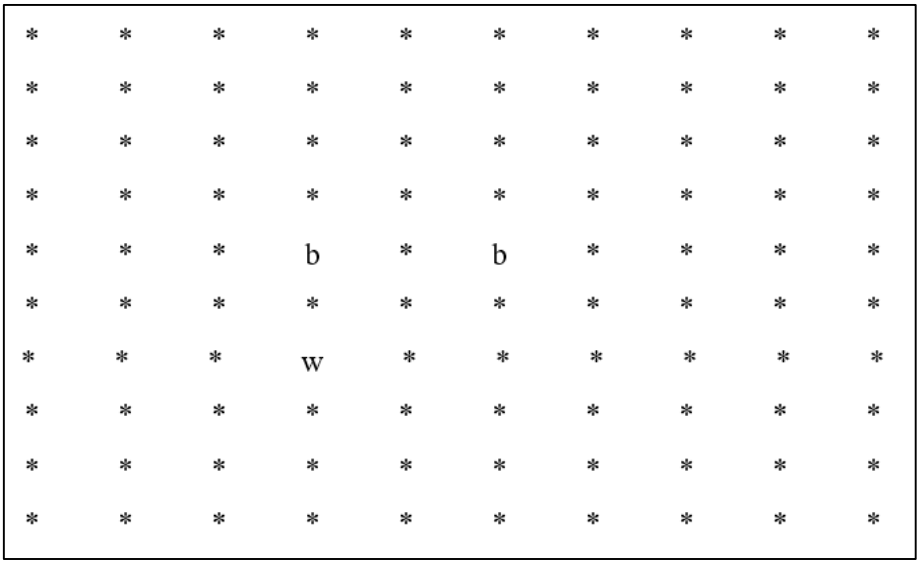

Рисунок 3.1 – початкове розташування фігур

В такому випадку вивід програми буде наступним:

```
Move 1:
Moved from [4, 3] to  [5, 4]
Board view:
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         b         *         *         *         *
    *         *         *         *         b         *         *         *         *         *
    *         *         *         w         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
```

Рисунок 3.2 – перший прогнозований хід

```
Move 2:
Moved from [-4, -7] to  [-5, -8]
Board view:
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         b         *         *         *         *
    *         *         w         *         b         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
```

Рисунок 3.3 – другий прогнозований хід

Щоб прочитати дані описи ходів, потрібно розуміти, що для опису ходів чорних шашок використовуємо додатні координати з точкою відліку (0, 0), яка знаходиться в лівому верхньому куті, для опису ходів білих шашок використовуємо від’ємні координати та точку відліку (-1, -1), яка знаходиться в правому нижньому куті.

Розглянемо розстановку фігур, ускладнивши попередній приклад. Починають білі фігури та прогнозуємо два ходи.

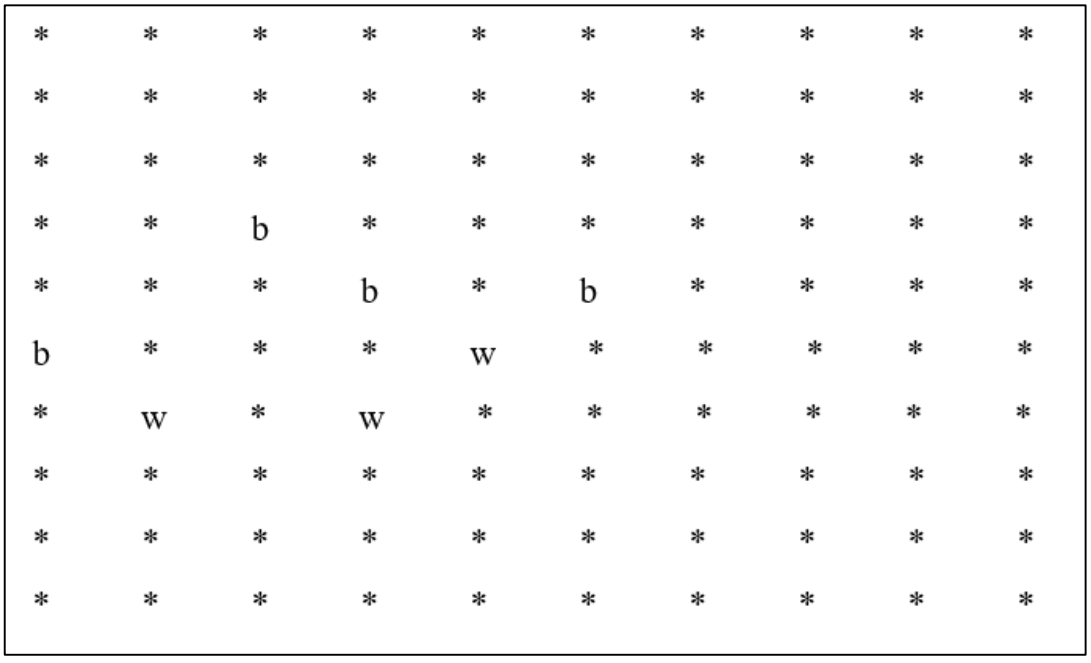

Рисунок 3.4 – початкове розташування фігур

У даному випадку вивід програми буде наступним:

```
Move 1:
Attacked from [-5, -6] to [[-6, -5]]
Board view:
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         b         *         *         *         w         *         *         *
    *         *         *         b         *         *         *         *         *         *
    b         *         *         *         *         *         *         *         *         *
    *         w         *         w         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
```

Рисунок 3.5 – перший прогнозований хід

```
Move 2:
Attacked from [5, 0] to [[6, 1], [6, 3]]
Board view:
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         b         *         *         *         w         *         *         *
    *         *         *         b         *         *         *         *         *         *
    *         *         *         *         b         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
```

Рисунок 3.6 – другий прогнозований хід

Можемо побачити, що спочатку біла шашка б’є одну чорну, потім у наступному ході чорна шашка виконує дві атаки. Це доволі раціональні ходи.

Розглянемо ще один приклад більшої складності. Гру починають чорні фігури і прогнозується чотири ходи.

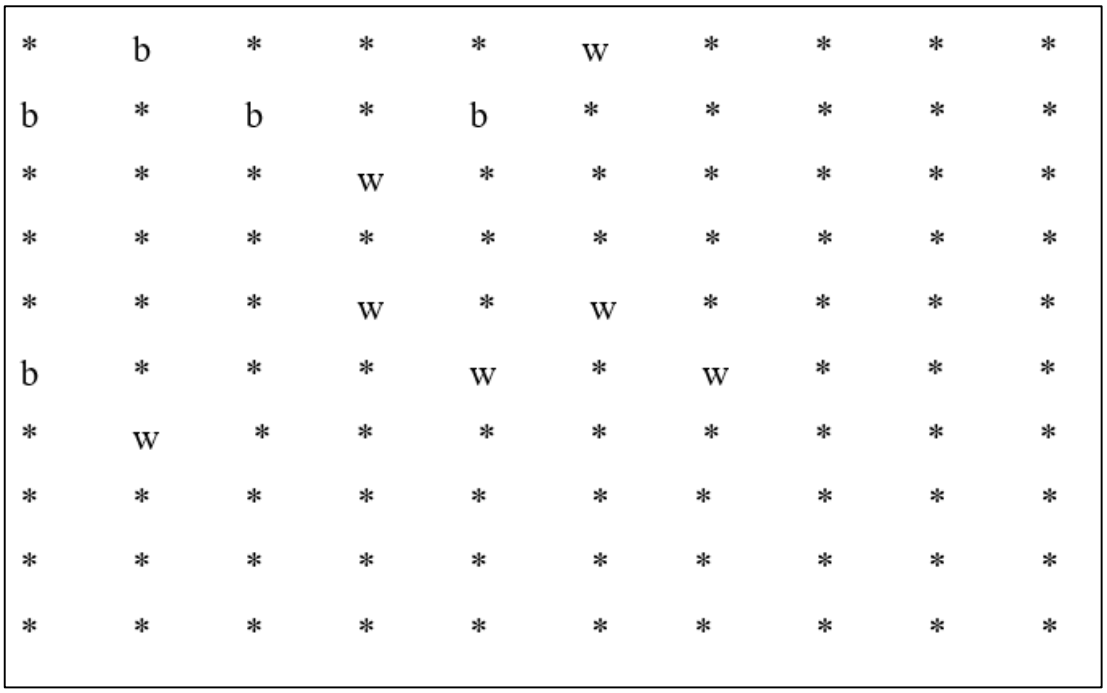

Рисунок 3.7 – початкове розташування фігур

Вивід програми буде таким:

```
Move 1:
Attacked from [1, 2] to [[2, 3], [4, 3], [6, 1]]
Board view:
    *         b         *         *         *         w         *         *         *         *
    b         *         *         *         b         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         w         *         *         *         *
    b         *         *         *         w         *         w         *         *         *
    *         *         *         *         *         *         *         *         *         *
    b         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
    *         *         *         *         *         *         *         *         *         *
```

Рисунок 3.8 – перший прогнозований хід

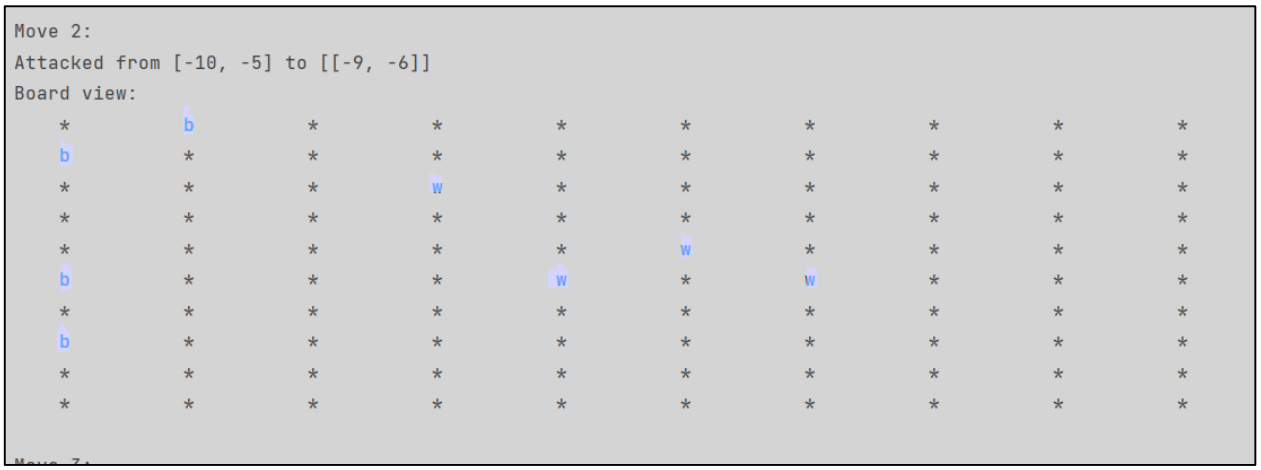

Рисунок 3.9 – другий прогнозований хід

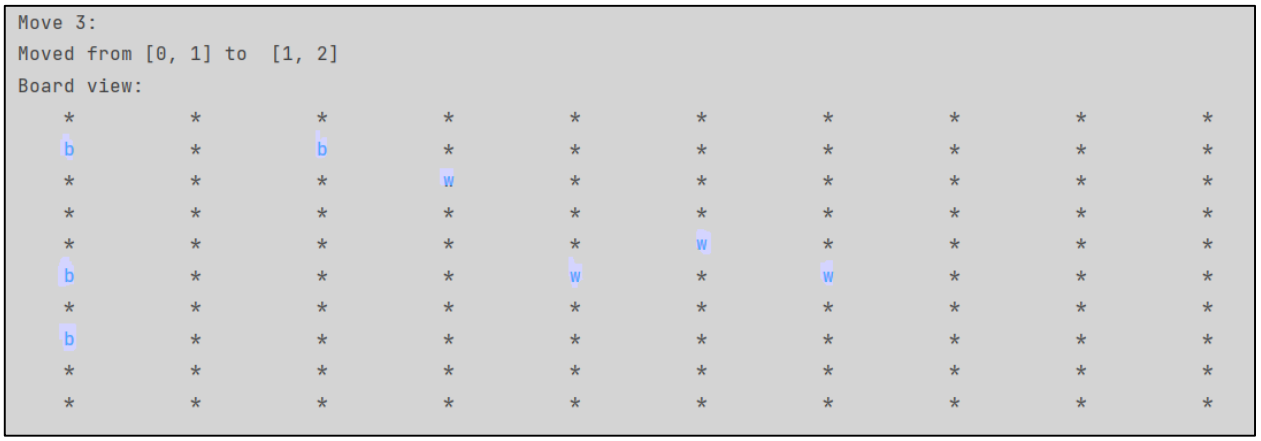

Рисунок 3.10 – третій прогнозований хід

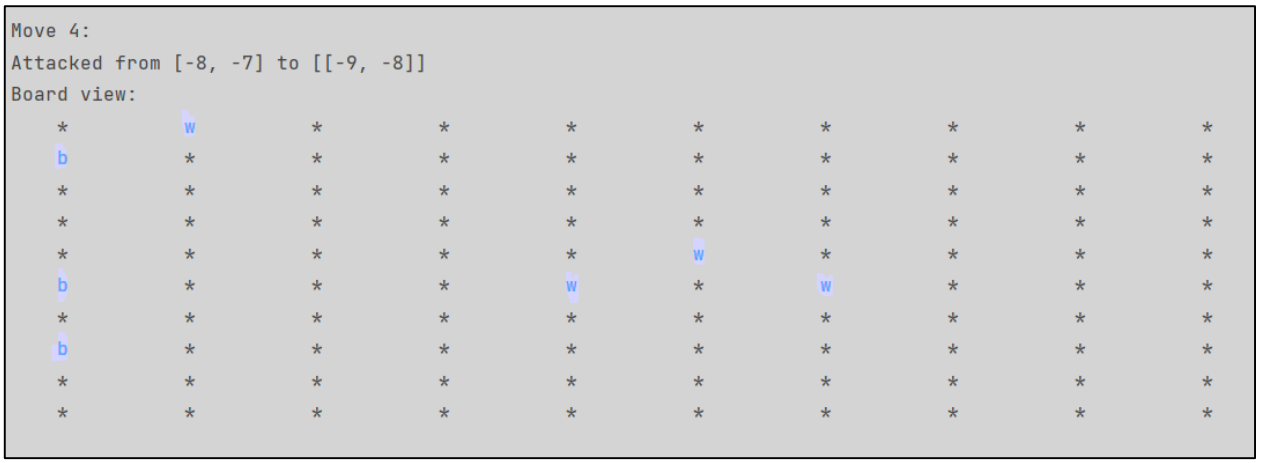

Рисунок 3.11 – четвертий прогнозований хід

Під час першого ходу чорні шашки вибили 3 білі, після чого білі вибили 1 чорну шашку, далі чорні зробили одне переміщення, і знову білі вибили 1

чорну шашку. Тут видно, що третій хід не зовсім раціональний враховуючи реальні стратегії гри, тільки якщо не вважати це як жертву фігури для відволікання сил противника.

3.2 Порівняння застосованого алгоритму з іншими алгоритмами для розв'язку поставленої задачі

Поставлену задачу також можна розв'язати за допомогою алгоритму альфа-бета відсічення, який було розглянуто у попередньому розділі. Даний алгоритм має мінімальні відмінності у реалізації в порівнянні з алгоритмом мінімакс, проте дозволяє вирішити поставлену задачу швидше завдяки меншій кількості перебору можливих ходів. Застосовуючи альфа-бета відсічення ми повністю припиняємо оцінювати певний хід, якщо перевірили, що цей хід гірше, ніж будь-який оцінений раніше. Відповідно не потрібно розглядати усі можливі подальші ходи після цього ходу.

Альфа-бета відсічення є чутливим до порядку ходів. У найгіршому випадку, кількість переглянутих можливих ходів буде такою ж, як і при застосуванні алгоритму мінімакс. Проте у найкращому випадку, цей алгоритм побудує значно менше дерево станів гри. У такому випадку кількість ходів, які потрібно проаналізувати приблизно буде дорівнювати кореню квадратному з кількості ходів, які розглядаються при використанні алгоритму мінімакс.

Окрім алгоритму мінімакс та альфа-бета відсічення, для моделювання виграшних ходів у шашках також можуть бути використані різні алгоритми, які базуються на методах машинного навчання, наприклад, навчання з підкріпленням (reinforcement learning). Ці алгоритми можуть навчитися самостійно приймати рішення на основі досвіду гри з іншими гравцями або із самими собою.

Порівняємо використання алгоритму мінімакс та навчання з підкріпленням для розв'язку поставленої задачі.

1) Підхід:

   Мінімакс: Алгоритм мінімакс базується на пошуку оптимального ходу у грі, розглядаючи всі можливі комбінації ходів до певної глибини дерева гри. Він працює на основі перебору всіх можливих ходів та обчислення їх оцінок.

   Навчання з підкріпленням: навчання з підкріпленням використовує методи машинного навчання для тренування агента на основі досвіду гри. Агент взаємодіє з середовищем гри, отримує винагороду за кожний зроблений хід і навчається приймати рішення на основі отриманої винагороди.

2) Використання інформації:

   Мінімакс: Алгоритм мінімакс потребує повної інформації про гру, тобто він має доступ до всіх правил та стану гри в кожен момент часу.

   Навчання з підкріпленням: навчання з підкріпленням може використовувати як повну, так і часткову інформацію про гру. Агент може навчатись на основі власного досвіду гри, без необхідності заздалегідь заданого правилами гри.

3) Обчислювальна складність:

   Мінімакс: Обчислювальна складність алгоритму мінімакс зростає експоненційно зі збільшенням глибини дерева гри. Це може стати проблемою для гри в шашки, де кількість можливих ходів є великою.

   Навчання з підкріпленням: Обчислювальна складність алгоритму навчання з підкріпленням залежить від кількості тренувальних ітерацій та складності моделі, яку необхідно навчати. У деяких випадках, особливо при використанні складних моделей, тренування може бути обчислювально витратним процесом. Однак, в порівнянні з алгоритмом мінімакс, навчання з підкріпленням може бути більш ефективним, оскільки не потребує повного перебору всіх можливих ходів.

4) Гнучкість:

Мінімакс: Алгоритм мінімакс є детермінованим і не може адаптуватися до зміни умов гри або стратегії противника без повного перерахування дерева гри.

Навчання з підкріпленням: навчання з підкріпленням дозволяє агенту адаптуватися до змін умов гри та стратегії противника без повного перерахунку. Агент може навчитися на основі нових даних та винагороди і покращувати свою стратегію з часом [3, 6].

Отже, навчання з підкріпленням теж найкраще підходить для моделювання наступних виграшних ходів у шашках. Проте оскільки при використанні навчання з підкріпленням ми можемо застосовувати часткову інформацію про гру, то навіть при великій кількості ітерацій навчання моделі, агент може робити помилки, які пов’язані з нежорстким заданням обмежень на етапі побудови моделі. Головною перевагою навчання з підкріпленням над алгоритмом мінімакс є менша обчислювальна складність алгоритму, що пов’язано з розглядом усіх можливих ходів у алгоритмі мінімакс, що не виконується при використанні навчання з підкріпленням.

У Розділі 1 було розглянуто генетичний алгоритм, який можна застосувати для розв’язку поставленої задачі. Перевагою даного алгоритму над алгоритмом мінімакс може бути обчислювальна складність, оскільки як і при використанні навчання з підкріпленням, розглядати усі можливі ходи не потрібно. Недоліком генетичного алгоритму є те, що потрібно чітко означити еволюційний оператор схрещування ходів, тобто операцію утворення індивіду, що представляє хід, з двох інших індивідів, що представляють кращі ходи у певному поколінні, що у даній задачі є не зовсім зрозумілим. Також може знадобитися велика кількість ітерацій для знаходження найкращих ходів, що збільшить обчислювальну складність, і у деяких випадках потрібно буде більше часу для отримання результату у порівнянні з алгоритмом мінімакс, незважаючи на те, що генетичному алгоритму не потрібно розглядати усі можливі ходи.

3.3 Опис можливих варіантів удосконалення роботи

У даній роботі поставлене завдання виконується, тобто програма прогнозує наступні найкращі ходи фігур відповідно до введених користувачем даних. Проте програму, що представлена в роботі можна удосконалити.

По-перше, одним із удосконалень може бути реалізація графічного інтерфейсу, щоб користувачеві було зручніше задавати початкове розташування фігур на дошці. Також таке представлення дошки буде корисним для кращого розуміння подальших прогнозованих програмою ходів. Можна вказати стрілками напрямок рухів фігур, або ж відтворити рух шашок з можливим переглядом попередніх станів дошки.

По-друге, код програми можна удосконалити, уникнувши дублювання коду в схожих за своєю логікою методах, таких як check_moveable_black та check_moveable_white, check_black_move і check_white_move, check_black_attack та check_white_attack.

Ще одним із удосконалень роботи може бути вибір оцінювальної функції, яка буде враховувати інші фактори, такі як позиції фігур на дошці, відстань від фігур до центру дошки.

ВИСНОВКИ

У даній кваліфікаційній роботі була поставлена задача реалізації застосунку для пошуку виграшних ходів у шашках.

Для вирішення поставленої задачі спочатку було розглянуто алгоритми теорії ігор, такі як мінімаксний алгоритм, алгоритм альфа-бета відсічення та генетичний алгоритм. Розглянуто алгоритм навчання з підкріпленням.

Після аналізу даних алгоритмів було вирішено використати у реалізації додатку алгоритм мінімакс.

До реалізації додатку було вибрано спосіб представлення станів дошки та оцінювальну функцію, які необхідні для використання алгоритму мінімакс.

Далі було реалізовано додаток на мові програмування python, який виконує поставлену початково задачу.

Розроблений додаток було протестовано та проаналізовано можливі способи покращення роботи додатку, такі як реалізація більш зручного для користувача графічного інтерфейсу, вибір кращої оцінювальної функції, яка враховуватиме інші характеристики гри. Також розглянуто покращення коду програми.

Оскільки «Міжнародні шашки» є популярною настільною грою з великою кількістю стратегій, то дослідження можливих способів моделювання найкращих ходів є доцільним для створення більш складних ігрових ситуацій для гравців, розвитку їхніх навичок гри. У галузі прикладної математики розроблений застосунок може використовуватися для дослідження ігор, подібних до шашок, розвитку наявних способів моделювання виграшних ходів у таких іграх.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Мова запитів JMESPath [Електронний ресурс] – Режим доступу до ресурсу: https://python.land/data-processing/working-with-json/jmespath.
2. ТЕОРІЯ ІГОР [Електронний ресурс] // КПІ ім. Ігоря Сікорського. – 2022. – Режим доступу до ресурсу: https://www.google.com/url?sa=t&source=web&rct=j&url=https://ela.kpi.ua/bitstream/123456789/49092/1/Teoriia_ihor.pdf&ved=2ahUKEwjEtuC1yIH_AhUM26QKHW8sAigQFnoECAwQAQ&usg=AOvVaw0Aleqm6U5DAbbce_lBc48P.
3. Mini-Max Algorithm in Artificial Intelligence [Електронний ресурс] – Режим доступу до ресурсу: https://www.javatpoint.com/mini-max-algorithm-in-ai.
4. Alpha-Beta Pruning [Електронний ресурс] – Режим доступу до ресурсу: https://www.javatpoint.com/ai-alpha-beta-pruning.
5. Генетичний алгоритм [Електронний ресурс] – Режим доступу до ресурсу: http://www.znannya.org/?view=ga_general.
6. Reinforcement learning [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/what-is-reinforcement-learning/.
7. GETTING STARTED AND STRATEGIC ON CHECKERS [Електронний ресурс] – Режим доступу до ресурсу: https://h-o-m-e.org/how-many-pieces-in-checkers/.

ДОДАТКИ

Додаток А

```
def check_moveable_white(board):
    points = []
    for i in range(-1, -11, -1):
        for j in range(-1, -11, -1):
            point = {"point_i": None, "point_j": None, "move_to": [], "attack": []}
            p = board[i][j]
            if p == "w":
                point["point_i"] = i
                point["point_j"] = j
                point["move_to"] = check_white_move(board, i, j)
                point["attack"] = check_white_attack(board, i, j)
                  if point["move_to"] != [] or point["attack"] != []:
                   points.append(point)
    return points
```

Рисунок А.1 – код функції check_moveable_white

```
            def check_black_attack(board, i, j, passed_from=None):
                attack = []
                try:
                    if board[i + 1][j - 1] == "w" and not board[i + 2][j - 2] and j - 2 >= 0 and passed_from != [i + 1, j - 1]:
                        attack.append({"point_from_i": i, "point_from_j": j, "point_i": i + 1, "point_j": j - 1, "point_to_i": i + 2,
"point_to_j": j - 2})
                        try:
                            add_atack = check_black_attack(board, i + 2, j - 2, passed_from=[i + 1, j - 1])
                            for attacks in add_atack:
                                attack.append(attacks)
                        except IndexError:
                            pass
                except IndexError:
                    pass
                try:
                    if board[i + 1][j + 1] == "w" and not board[i + 2][j + 2] and passed_from != [i + 1, j + 1]:
                        attack.append({"point_from_i": i, "point_from_j": j, "point_i": i + 1, "point_j": j + 1, "point_to_i": i + 2,
"point_to_j": j + 2})
                        try:
                            add_atack = check_black_attack(board, i + 2, j + 2, passed_from=[i + 1, j + 1])
                            for attacks in add_atack:
                                attack.append(attacks)
                        except IndexError:
                            pass
                except IndexError:
                    pass
                try:
                    if board[i - 1][j - 1] == "w" and not board[i - 2][j - 2] and j - 2 >= 0 and passed_from != [i - 1, j - 1]:
```

Рисунок А.2 – код функції check_black_attack

```
            attack.append({"point_from_i": i, "point_from_j": j, "point_i": i - 1, "point_j": j - 1, "point_to_i": i - 2, "point_to_j": j
    - 2})
                try:
                    add_atack = check_black_attack(board, i - 2, j - 2, passed_from=[i - 1, j - 1])
                    for attacks in add_atack:
                        attack.append(attacks)
                except IndexError:
                    pass
            except IndexError:
                pass
            try:
                if board[i - 1][j + 1] == "w" and not board[i - 2][j + 2] and passed_from != [i - 1, j + 1]:
                    attack.append({"point_from_i": i, "point_from_j": j, "point_i": i - 1, "point_j": j + 1, "point_to_i": i - 2,
"point_to_j": j + 2})
                try:
                    add_atack = check_black_attack(board, i - 2, j + 2, passed_from=[i - 1, j + 1])
                    for attacks in add_atack:
                        attack.append(attacks)
                except IndexError:
                    pass
            except IndexError:
                pass
            return attack
```

Рисунок А.3 – продовження коду функції check_black_attack

```
        def check_white_attack(board, i, j, passed_from=None):
            attack = []
            try:
                if board[i + 1][j - 1] == "b" and not board[i + 2][j - 2] and passed_from != [i + 1, j - 1]:
                    attack.append({"point_from_i": i, "point_from_j": j, "point_i": i + 1, "point_j": j - 1, "point_to_i": i + 2,
"point_to_j": j - 2})
                    try:
                        add_atack = check_white_attack(board, i + 2, j - 2, passed_from=[i + 1, j - 1])
                        for attacks in add_atack:
                            attack.append(attacks)
                    except IndexError:
                        pass
            except IndexError:
                pass
            try:
                if board[i + 1][j + 1] == "b" and not board[i + 2][j + 2] and j + 2 < 0 and passed_from != [i + 1, j + 1]:
                    attack.append({"point_from_i": i, "point_from_j": j, "point_i": i + 1, "point_j": j + 1, "point_to_i": i + 2,
"point_to_j": j + 2})
                    try:
                        add_atack = check_white_attack(board, i + 2, j + 2, passed_from=[i + 1, j + 1])
                        for attacks in add_atack:
                            attack.append(attacks)
                    except IndexError:
                        pass
```

Рисунок А.4 – код функції check_white_attack

```
            except IndexError:
                pass
            try:
                if board[i - 1][j - 1] == "b" and not board[i - 2][j - 2] and passed_from != [i - 1, j - 1]:
                    attack.append({"point_from_i": i, "point_from_j": j, "point_i": i - 1, "point_j": j - 1, "point_to_i": i - 2,
"point_to_j": j - 2})
                    try:
                        add_atack = check_white_attack(board, i - 2, j - 2, passed_from=[i - 1, j - 1])
                        for attacks in add_atack:
                            attack.append(attacks)
                    except IndexError:
                        pass
            except IndexError:
                pass
            try:
                if board[i - 1][j + 1] == "b" and not board[i - 2][j + 2] and j + 2 < 0 and passed_from != [i - 1, j + 1]:
                    attack.append({"point_from_i": i, "point_from_j": j, "point_i": i - 1, "point_j": j + 1, "point_to_i": i - 2,
"point_to_j": j + 2})
                    try:
                        add_atack = check_white_attack(board, i - 2, j + 2, passed_from=[i - 1, j + 1])
                        for attacks in add_atack:
                            attack.append(attacks)
                    except IndexError:
                        pass
            except IndexError:
                pass
            return attack
```

Рисунок А.5 – продовження коду функції check_white_attack

```
def make_a_move(board, starts, moves):
    if starts == "w":
        moves_can_be_done = check_moveable_white(board)
        move_level = []
        for point in moves_can_be_done:
            move_desc = None
            attack_desc = None
            if point['move_to']:
                for point_to in point['move_to']:
                    board_after_move = move(board, point, point_to)
                    move_desc = {'desc': f'Moved from [{point["point_i"]}, {point["point_j"]}] to '
                                         f'[{point_to["point_i"]}, {point_to["point_j"]}]',
                                 'score': get_score(board_after_move), 'attacked': 'w',
                                 'board': board_after_move,
                                 'next_moves': []}
                    if moves - 1 != -1:
                        move_desc['next_moves'].append(make_a_move(board_after_move, "b", moves - 1))
            if point['attack']:
                board_after_move = attack(board, point['attack'])
                points = [[p["point_i"], p["point_j"]] for p in point['attack']]
                attack_desc = {'desc': f'Attacked from [{point["point_i"]}, {point["point_j"]}] to {points}',
```

Рисунок А.6 – перша частина коду функції make_a_move

```
 'score': get_score(board_after_move), 'attacked': 'w',
                    'board': board_after_move,
                    'next_moves': []}
          if moves - 1 != -1:
            attack_desc['next_moves'].append(make_a_move(board_after_move, "b", moves - 1))
       if move_desc:
          move_level.append(move_desc)
       if attack_desc:
          move_level.append(attack_desc)
if starts == "b":
   moves_can_be_done = check_moveable_black(board)
   move_level = []
   for point in moves_can_be_done:
      move_desc = None
      attack_desc = None
      if point['move_to']:
         for point_to in point['move_to']:
            board_after_move = move(board, point, point_to)
            move_desc = {'desc': f'Moved from [{point["point_i"]}, {point["point_j"]}] to '
                                 f'[{point_to["point_i"]}, {point_to["point_j"]}]',
                    'score': get_score(board_after_move), 'attacked': 'b',
                    'board': board_after_move,
                    'next_moves': []}
            if moves - 1 != -1:
               move_desc['next_moves'].append(make_a_move(board_after_move, "w", moves - 1))
```

Рисунок А.7 – друга частина коду функції make_a_move

```
if point['attack']:
         board_after_move = attack(board, point['attack'])
         points = [[p["point_i"], p["point_j"]] for p in point['attack']]
         attack_desc = {'desc': f'Attacked from [{point["point_i"]}, {point["point_j"]}] to {points}',
                    'score': get_score(board_after_move), 'attacked': 'b',
                    'board': board_after_move,
                    'next_moves': []}
         if moves - 1 != -1:
           attack_desc['next_moves'].append(make_a_move(board_after_move, "w", moves - 1))
      if move_desc:
         move_level.append(move_desc)
      if attack_desc:
         move_level.append(attack_desc)
return move_level
```

Рисунок А.8 – третя частина коду функції make_a_move

```
def minimax(tree, query, subquery):
    path = jmespath.search(query, tree)
    while type(path[0]) != dict:
        path = path[0]
    subpath = jmespath.search(subquery, path[0])
    if subpath:
        for que in path:
            que['next_moves'] = minimax(que, subquery, subquery)
        strategy = {"moves": path}
        if path[0]['attacked'] == 'w':
            result = jmespath.search("max_by(moves, &score)", strategy)
else:
            result = jmespath.search("min_by(moves, &score)", strategy)
        return result
    else:
        branch = {"moves": path}
        if path[0]['attacked'] == 'w':
            result = jmespath.search("max_by(moves, &score)", branch)
        else:
            result = jmespath.search("min_by(moves, &score)", branch)
        return result
```

Рисунок А.9 – код функції minimax

```
moves_count = int(input('Enter number of moves: '))
    color = input('Chose color to play W or B: ')[0].lower()
    decision_tree = {'moves': make_a_move(filled_board, color, moves_count)}

    jmespath_query = "moves[*]"
    result = minimax(decision_tree, jmespath_query, "next_moves[*]")
    for i in range(moves_count):
        jpath = 'next_moves.' * i + 'desc'
        jpath_board = 'next_moves.' * i + 'board'
        description = jmespath.search(jpath, result)
        board = jmespath.search(jpath_board, result)
        print(f'\nMove {i + 1}:\n{description}\nBoard view:')
        for row in board:
            for column in row:
                st = column if column else '*'
                print('%5s' % (str(st)), end='     ')
            print()
        draw_board(board)
```

Рисунок А.10 – код підготовки даних та знаходження результатів