НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО”
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ
КАФЕДРА ШТУЧНОГО ІНТЕЛЕКТУ

До захисту допущено:
Завідувач кафедри
_____Олена ЧУМАЧЕНКО
«___»_____20__р

Дипломна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системи і методи штучного інтелекту» спеціальності 122 «Комп’ютерні науки»
на тему: «Штучний інтелект в шахах»

Виконав:
студент 4 курсу, групи КІ-02
Задорожний Олег Андрійович

_____

Керівник:
професор, д.т.н., Зайченко Олена Юріївна

_____

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

_____

Консультант з нормоконтролю:
фахівець першої категорії Гончарук Максим Миколайович

_____

Рецензент:
професор, д.т.н., Мухін Вадим Євгенович

_____

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

Київ 2024

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ
КАФЕДРА ШТУЧНОГО ІНТЕЛЕКТУ

Рівень вищої освіти — перший(бакалаврський)
Спеціальність — 122 «Комп’ютерні науки»
Освітньо-професійна програма «Системи та методи штучного ігнтелекту»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____О.І. Чумаченко
«___»_____20__р.

ЗАВДАННЯ
на дипломну роботу студенту
Задорожному Олегу Андрійовичу

1. Тема роботи «Штучний інтелект в шахах», керівник роботи Зайченко Олена Юріївна, професор кафедри ММСА, затверджені наказом по університету від «30» травня 2023 р. №2065-с
2. Термін подання студентом роботи 15.06.2024
3. Вихідні дані до роботи: шахова гра
4. Зміст роботи: Аналіз предметної області дослідження та даних, вибір доцільних методів та моделей для роботи, моделювання прогнозу та класифікації обраними методами, оцінка отриманих результатів та вибір найкращої моделі.
5. Перелік ілюстративного матеріалу: використані метрики, опис набору даних, принцип роботи програмного продукту, результати прогнозування, схеми роботи алгоритмів.
6.Консультанти розділів роботи

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

7. Дата видачі завдання: 21 лютого 2024 року.
Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Огляд літератури за темою	12.03.2024	
2.	Підготовка першого розділу	16.04.2024	
3.	Підготовка другого розділу	01.05.2024	
4.	Розробка програмного продукту	16.05.2023	
5.	Підготовка третього розділу	22.05.2024	

6.	Підготовка економічної частини	28.05.2024	
7.	Оформлення розділів відповідно до нормоконтролю	30.05.2024	
8.	Підготовка презентації доповіді	02.06.2024	
9.	Оформлення дипломної роботи	03.06.2024	

Студент

Олег Задорожний

Керівник

Олена Зайченко

РЕФЕРАТ

Дипломна робота — 106 с., 11 табл., 39 рис., 1 додаток, 16 джерел.

ШАХИ. СТРАТЕГІЇ. МЕТОДИ ШТУЧНОГО ІНТЕЛЕКТУ

Тема: Штучний інтелект в шахах

У роботі розглянуто та проаналізовано шахи та методи штучного інтелекту в них

Об’єкт дослідження: процес знаходженя оптимального ходу

Мета роботи: проаналізувати та розробити програмне забезпечення для гри

Створено програмний продукт для знаходження оптимальних ходів

ABSTRACT

Thesis: 106 p., 11 tabl., 39 fig., 1 appendix, 16 sources.

CHESS. STRATEGIES. TOOLS OF AI

Topic: AI in chess

The paper considers and analyses chess and ai tools

The object of the study: the finding of optimal move

Research subject: minimax

The purpose of the work: to analyse and to develop AI for chess

A software product has been created and tested

ВСТУП

Шахи, як древнє й вічне мистецтво, завжди привертали увагу не лише гравців, а й дослідників, математиків та інженерів. З появою комп'ютерів та штучного інтелекту (ШІ) ця гра отримала новий рівень еволюції. Історично, розвиток шахів був суцільним випробуванням інтелекту людини, проте з впровадженням штучного інтелекту ця гра стала лабораторією для вивчення та розвитку алгоритмів, що намагаються мати не меншу стратегічну глибину та творчість, ніж людський розум.

У цьому дипломному проекті розглядається роль та вплив штучного інтелекту в шаховій грі. Досліджується, які методи та алгоритми використовуються для створення шахових програм, які стратегічні аспекти вони враховують, та як вони змагаються проти гравців на різних рівнях майстерності. Також розглядається етап еволюції від перших шахових програм до сучасних систем, що використовують нейронні мережі та глибоке навчання.

Цей дипломний проект має на меті розкрити значення штучного інтелекту в контексті шахів, як важливого інструменту для розвитку і поглиблення нашого розуміння не лише гри самої по собі, а й розвитку алгоритмів і навичок, що можуть знайти застосування у різних сферах життя.

РОЗДІЛ 1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Фігури

У грі в шахи кожна сторона має шість різних фігур, кожна з яких має власні унікальні можливості руху і значення. Ось короткий опис кожної фігури:

1. **Король (King)**:

   - Король є найважливішою фігурою в грі.

   - Він може рухатися на одне поле у будь-якому напрямку: вертикально, горизонтально або по діагоналі.

   - Ціль гри полягає в захисті свого короля від атаки та в одночасному наступі на короля супротивника.

2. **Ферзь (Queen)**:

   - Ферзь є найпотужнішою фігурою на дошці.

   - Він може рухатися будь-якою кількістю клітинок по вертикалі, горизонталі або по діагоналі.

   - Ферзь може виконувати функції як слона, так і тура.

3. **Тур (Rook)**:

   - Тур може рухатися вздовж вертикалей або горизонталей на будь-яку відстань.

   - Він не може перескакувати інші фігури.

4. **Слон (Bishop)**:

   - Слон рухається по діагоналі на будь-яку відстань.

   - На початку гри кожен гравець має два слона: один рухається по світлим клітинках, інший - по темним.

5. **Кінь (Knight)**:

   - Кінь рухається по 'L'-подібній траєкторії: два кліки в одному напрямку, а потім один клік перпендикулярно.

   - Він може перескакувати навіть найближчі фігури.

6. **Пішак (Pawn)**:

   - Пішак рухається вперед на одну клітинку за раз, але з першого руху може рухатися на дві клітинки.

   - Він атакує по діагоналі, але рухається прямо.

   - Пішак може бути просунутий до іншої фігури при досягненні останнього ряду протилежного краю дошки, це називається промоцією.

Ці фігури утворюють складну мережу можливих ходів і тактик, що визначають гру в шахи.

1.2 Дошка

У грі в шахи дошка є полем битви для гравців. Ось деякі основні характеристики дошки:

1. **Розмір**: Стандартна шахова дошка має розмір 8x8, що складається з 64 квадратних клітинок.

2. **Кольори клітинок**: Кожна клітинка на дошці має свій колір, який може бути світлим або темним. Клітинки на дошці чередуються між цими двома кольорами.

3. **Координати**: Кожна клітинка на дошці має свої координати, які вказуються у вигляді пари букви та цифри. Наприклад, клітинка на верхньому лівому куті дошки має координати "a1", а на нижньому правому - "h8".

4. **Сектори**: Дошка поділяється на сектори: вертикалі (колонки), горизонталі (рядки) та квадрати (клітинки). Це дозволяє гравцям легко вказувати місце розташування фігур під час гри.

5. **Стартове розташування фігур**: Фігури кожного гравця розташовані на початкових позиціях на дошці перед початком гри. Пішаки розташовані на другому та сьомому рядках, а інші фігури - на першому та восьмому рядах.

6. **Кінцеві ряди**: Кінцеві ряди дошки (перший та восьмий ряди) є цільовими рядами для пішаків. Якщо пішак досягає кінця протилежного боку дошки, він може бути просунутий до іншої фігури, що називається промоцією.

7. **Центральні квадрати**: Квадрати центральної частини дошки (квадрати e4, d4, e5 та d5) часто вважаються ключовими для контролю над центром та встановлення позиційної переваги.

Дошка є основним інструментом для гри в шахи і надає гравцям платформу для стратегічного мислення та тактичних розрахунків.

1.3 Хід

Хід - це кожен рух, який гравець робить у свою чергу під час гри в шахи. Ось декілька ключових аспектів ходу:

1. **Обрання фігури**: Гравець обирає одну зі своїх фігур, яку він хоче перемістити.

2. **Цільове поле**: Гравець вказує клітинку, на яку він хоче перемістити вибрану фігуру. Ця клітинка може бути порожньою або містити фігуру супротивника, яку гравець планує захопити.

3. **Переміщення фігури**: Вибрана фігура переміщується на обрану клітинку. Рух фігури повинен відповідати її правилам руху, які визначені правилами гри.

4. **Захоплення фігури**: Якщо фігура переміщується на клітинку, де вже знаходиться фігура супротивника, остання захоплюється і вилучається з гри.

5. **Спеціальні ходи**: Деякі ходи мають спеціальні правила, такі як рокада, промоція пішака або відпускний хід.

6. **Шах та мат**: Гравець може виставити свого супротивника під шах, коли його король знаходиться під атакою. Якщо король не може уникнути цієї атаки, це вважається матом і закінчує гру.

Кожен гравець робить хід після ходу супротивника, і така послідовність продовжується до завершення гри. У кожного гравця є обмежена кількість часу для здійснення свого ходу, що додає напруги та стратегічних викликів до гри.

1.4 Мат

"Мат" - це ситуація в грі в шахи, коли король одного з гравців перебуває під атакою і немає можливості втекти або захиститися. В інших словах, король виставлений під угрозу "шаху", але гравець не може зробити жодного ходу, який дозволив би йому уникнути цього шаху.

Існує кілька типів "матів", включаючи:

1. **Мат в один хід**: Коли король перебуває під атакою однієї фігури і не може втекти на будь-яку клітинку, тому що вони всі також під загрозою.

2. **Мат за допомогою двох фігур**: Це ситуація, коли дві фігури працюють разом, щоб виставити короля під атаку. Наприклад, одна фігура може заблокувати рухи короля, а інша фігура атакує короля.

3. **Мат з використанням фігур та пішаків**: У деяких випадках, пішаки також можуть грати важливу роль у маті. Наприклад, пішак може просунутися і стати фігурою, яка атакує короля, або блокувати його можливі рухи.

Виникнення "мату" означає завершення гри, і гравець, чий король був поставлений під мат, вважається переможеним. У цей момент гра закінчується, і гравці можуть розпочати нову гру або домовитися про результати поточної.

1.5 Пат

"Пат" - це ситуація в грі в шахи, коли гравець, що має чергу ходу, не може зробити будь-який хід, який не поставив би його короля під угрозу "шаху", але при цьому сам король не перебуває під шахом. Іншими словами, гравець не може зробити жодного законного ходу, але його король не перебуває під прямою загрозою "мата".

Пат вважається нічийним результатом гри. В цей момент гра також завершується, і гравці можуть домовитися про нічийний результат або домовитися про інші умови, такі як повторення позиції або трьохкратний повтор позиції (що також може призвести до пату). Пат виникає тоді, коли обидва гравці не можуть досягти перемоги, і ніхто не може змусити іншого визнати свою поразку.

Пат може виникнути з різних причин, включаючи:

1. Недостатньо фігур на дошці для досягнення мату.

2. Повторення позиції на дошці.

3. Переміщення фігур без зміни позиції (трьохкратний повтор позиції).

4. Блокування фігур на такий рівень, що жодна сторона не може досягти перемоги.

1.6 Рокада

"Рокада" - це спеціальний хід в грі в шахи, який може здійснювати король разом з одним із турів. Рокада дозволяє королю безпечно переміститися на дві клітинки в бік тура, який ставиться на клітинку, що знаходиться поруч з королем. Цей хід має кілька правил:

1. **Початкові умови**: Рокада може бути виконана тільки тоді, коли це ще не робився жоден з гравців у даній грі. Крім того, король та тур, які беруть участь в рокаді, не повинні бути переміщені раніше протягом гри.

2. **Вільні клітинки**: Усі клітинки між королем та туром, які беруть участь в рокаді, повинні бути порожніми. Це означає, що жодна фігура не повинна перебувати між ними, і також жодна з цих клітинок не може перебувати під атакою ворожої фігури.

3. **Заборона рокади під шахом**: Рокада неможлива, якщо король знаходиться під шахом, якщо буде переміщенося поле, через яке король проходить.

4. **Виконання рокади**: Щоб здійснити рокаду, король переміщується на дві клітинки в бік тура, який ставиться на клітинку поруч з королем. У той же час, цей тур переміщується на клітинку, яка знаходиться поруч з королем, на протилежний бік короля.

5. **Види рокади**: Є два типи рокади: коротка (з королем рухається на дві клітинки вправо або вліво) та довга (з королем рухається на дві клітинки вправо або вліво).

Рокада - це важливий стратегічний елемент в грі в шахи, який дозволяє гравцеві захистити свого короля та встановити безпеку у центрі дошки.

1.7 Промоція

"Промоція" - це правило в грі в шахи, за яким пішак, який досягає кінця дошки протилежного боку поля, може бути замінений на будь-яку іншу фігуру, крім короля. Це важливе правило дозволяє пішаку, який дійшов до кінця дошки, стати більш потужною фігурою і вплинути на хід гри.

Коли пішак досягає останнього ряду протилежного боку дошки (восьмого ряду для білого гравця або першого ряду для чорного), гравець має право замінити його на будь-яку фігуру свого кольору, за винятком короля:

1. **Ферзь**: Це найпопулярніший вибір для промоції пішака, оскільки ферзь є найпотужнішою фігурою, яка може рухатися будь-яким способом по вертикалі, горизонталі або діагоналі.

2. **Тур**: Іноді вибирають тура для промоції, особливо якщо у гравця вже є ферзь або виникає потреба в додаткових турах для атаки.

3. **Слон**: У випадках, коли гравець бажає підсилити свою діагональну атаку або використати слона в центрі дошки.

4. **Кінь**: Хоча кінь не є найбільш популярним вибором для промоції, іноді вибирають кінь для створення додаткової атаки або захисту.

Гравець може обрати будь-яку фігуру для промоції відповідно до потреб і стратегії гри. Промоція пішака може вирішити хід гри та вплинути на її подальший розвиток.

1.8 Відпускний хід

"Відпускний хід" - це спеціальний хід в грі в шахи, який дозволяє гравцю зробити два ходи замість одного у свою чергу. Цей хід може виконуватися тільки за певних умов і має деякі обмеження:

1. **Умови**: Відпускний хід може бути зроблений тільки тоді, коли пішак ще не був переміщений з його початкової позиції. Це означає, що пішак не повинен рухатися раніше в грі.

2. **Хід пішака**: Під час відпускного ходу, пішак рухається вперед на дві клітинки замість однієї. Це може бути зроблено тільки в перший хід пішака.

3. **Захоплення пішака**: Інші правила щодо захоплення пішака застосовуються так само, як і в інших випадках. Якщо пішак стає під атаку на чотирнадцятій клітинці, інший пішак може захопити його на цій клітинці, як це було б у звичайному русі пішака.

4. **Переваги відпускного ходу**: Відпускний хід дає можливість швидше розвивати свої фігури на дошці та отримувати перевагу в центрі. Також він може використовуватися для стратегічних маневрів або підготовки до атаки.

Відпускний хід - це важлива стратегічна можливість у грі в шахи, яка може вплинути на подальший розвиток партії та дати гравцю перевагу у позиції.

1.9 Позиційна гра та тактика

Позиційна гра та тактика є ключовими аспектами гри в шахи і вимагають від гравця глибокого стратегічного мислення та здатності читати дошку.

1. **Позиційна гра**:

   - **Центр контролю**: Гравці зазвичай борються за контроль над центральними квадратами дошки, оскільки це дозволяє краще контролювати гру та розвивати фігури.

   - **Розвиток фігур**: Важливо розташовувати фігури таким чином, щоб вони могли діяти ефективно і брати участь у грі. Наприклад, слони можуть бути розміщені на діагоналях, тури - на відкритих лініях тощо.

   - **Слабкі клітини**: Виявлення слабких клітин на дошці і зайняття їх контролю, наприклад, за допомогою пішаків або фігур, може стати ключовим елементом позиційної гри.

2. **Тактика**:

   - **Комбінації**: Включаючи фігури в комбінації, що приводять до переваги в матеріалі або підсилення позиційних переваг на дошці.

   - **Відволікаючі маневри**: Використання фігур для відволікання уваги супротивника від важливих клітинок або фігур на дошці.

   - **Простеження варіацій**: Здатність аналізувати різні варіанти розвитку гри і обирати найбільш ефективний для досягнення поставленої мети.

Позиційна гра та тактика взаємопов'язані і вимагають від гравця глибокого розуміння гри, а також здатності використовувати стратегічні та тактичні прийоми для досягнення перемоги. Комбінація цих двох аспектів дозволяє гравцеві створювати складні плани та реалізовувати їх на дошці шахів.

РОЗДІЛ 2 ТЕОРИТИЧНІ ОСНОВИ МЕТОДІВ ДОСЛІДЖЕННЯ

2.1 Постановка задачі

Постановка задачі прийняття рішень в грі полягає в тому, щоб розробити стратегію, яка дозволяє гравцеві приймати оптимальні рішення в умовах невизначеності і конкуренції. Основна мета полягає у максимізації вигоди або мінімізації втрат гравця.

Основні компоненти постановки задачі прийняття рішень в грі включають:

1. **Гравці**: Це учасники гри, кожен з яких має свої власні цілі та стратегії.

2. **Стратегії**: Це набір дій, які гравці можуть виконати в грі. Вони можуть бути детермінованими або випадковими, залежно від характеру гри.

3. **Вигоди (платежі)**: Це оцінки того, наскільки вигідними є різні варіанти гри для кожного гравця. Вони можуть бути виражені у виграшах, втратах або інших метриках.

4. **Стан гри**: Це поточний стан гри, який враховує всі рішення, прийняті гравцями до цього моменту.

5. **Правила гри**: Це набір правил, які визначають, як гра розвивається від початку до кінця, включаючи легальні ходи, умови виграшу або програшу.

6. **Ціль**: Це те, чого гравець намагається досягти в грі, наприклад, максимізація свого виграшу або мінімізація втрат.

Задача полягає в тому, щоб розробити стратегію для кожного гравця, яка б дозволила досягти їх цілей в умовах конкуренції з іншими гравцями. Це може включати в себе аналіз різних можливих ходів, врахування можливих відповідей від опонентів, оцінку вигод та ризиків кожного варіанту дій. У результаті гравець обирає найкращий хід для досягнення своєї цілі.

2.2 Мінімакс як метод вирішення

Мінімакс є одним із найпоширеніших методів прийняття рішень в стратегічних іграх, таких як шахи або покер. Основна ідея полягає в тому, щоб гравець максимізував свою мінімальну вигоду, припускаючи, що опонент намагається максимізувати свою мінімальну вигоду.

У методі мінімакс гравець розглядає всі можливі ходи, які він може зробити, а також всі можливі відповіді опонента на кожен з цих ходів. Він обчислює вигоду, яку він може отримати від кожного свого ходу, та обчислює мінімальну вигоду, яку може отримати опонент від відповіді на цей хід. Потім гравець обирає хід, який максимізує його мінімальну вигоду.

Це стратегічне прийняття рішень здатне до використання навіть у складних ігрових ситуаціях, оскільки воно дозволяє гравцю адаптуватися до можливих відповідей опонента. Однак метод мінімакс може бути обмеженим в реальних застосуваннях через велику обчислювальну складність, особливо в грах з великою кількістю можливих ходів.

У методі мінімакс гравець намагається мінімізувати свої втрати, припускаючи, що опонент намагається максимізувати свою максимальну вигоду. Тобто, гравець розглядає всі можливі ходи, які він може зробити, та всі можливі відповіді опонента на кожен з цих ходів. Він обчислює вигоду, яку він може отримати від кожного свого ходу, та обирає хід, який максимізує його мінімальну вигоду.

Метод мінімаксу є одним із фундаментальних підходів до прийняття рішень в галузі штучного інтелекту та теорії ігор. Він виник у 1940-х роках та був одним із перших формальних способів моделювання стратегій у стратегічних іграх.

Історія методу мінімаксу починається з робіт математиків та економістів, таких як Джон фон Нейман та Оскар Морґенштерн. Вони розвивали теорію ігор, де ставили за мету формалізувати стратегічні взаємодії між гравцями у вигляді математичних моделей. Одним із ключових понять, яке вони вперше ввели, був поняття "змішаних стратегій" — стратегій, де гравець вибирає свої ходи з певною ймовірністю.

Метод мінімаксу став результатом роботи Джона фон Неймана та Оскара Морґенштерна з формалізації стратегій у нуль-сумових іграх, де сума виграшів одного гравця дорівнює сумі втрат іншого. Вони показали, що в таких іграх можна знайти оптимальні стратегії для кожного гравця. Для цього вони запропонували метод мінімаксу — алгоритм, який дозволяє гравцю максимізувати його гарантований мінімум, припускаючи, що опонент грає оптимально.

Метод мінімаксу виявився дуже важливим у розвитку ігрової теорії, а також у штучному інтелекті. Він став основою для багатьох алгоритмів прийняття рішень у штучних системах,

таких як комп'ютерні програми для гри в шахи або покер, де стратегія гри базується на аналізі можливих ходів і їх наслідків.

Давайте розглянемо приклад застосування методу мінімаксу на прикладі простої стратегічної гри, наприклад, гри "камінь-ножиці-папір".

У грі "камінь-ножиці-папір" кожен гравець може обрати один з трьох можливих ходів: камінь, ножиці або папір. Задача гравця - обрати такий хід, який максимізує його вигоду, враховуючи можливі варіанти відповіді опонента.

Припустимо, що ми розглядаємо гру між двома гравцями - гравець A та гравець B. Для прикладу, нехай гравець A обрав хід, а гравець B відповів на цей хід.

Ось таблиця можливих виграшів для гравця A (рядки) та гравця B (стовпці):

```
      | Камінь  |  Ножиці  |  Папір
----------------------------------------
Камінь  |   0, 0    |   1, -1   |  -1, 1
----------------------------------------
Ножиці  |  -1, 1    |   0, 0    |   1, -1
----------------------------------------
Папір   |   1, -1   |  -1, 1    |   0, 0
```

У цій таблиці перший елемент в кожній комірці показує вигоду гравця A, а другий - вигоду гравця B. Наприклад, якщо обидва гравці обирають камінь, то вигода буде рівна 0 для обох гравців.

Метод мінімаксу дозволяє гравцю A максимізувати його гарантований мінімум. Отже, гравець A обирає хід так, щоб максимізувати його мінімальний виграш у кожному можливому випадку.

Наприклад, якщо гравець А обрав камінь, то гравець В має вибір між ножицями та папером. У цьому випадку гравець В максимізує його мінімальний виграш, обираючи папір (1, -1), тобто гравець А має виграш 1, а гравець В -1. Таким чином, камінь є оптимальним ходом для гравця А в цьому випадку.

Аналогічно, можна розглянути всі можливі ходи гравця А та вибрати той, що максимізує його мінімальний виграш у кожному випадку.

Це лише простий приклад, який ілюструє застосування методу мінімаксу в стратегічних іграх. У більш складних іграх, таких як шахи чи покер, метод мінімаксу використовується для прийняття рішень в умовах невизначеності та конкуренції.

метод мінімаксу є дуже потужним і ефективним інструментом для прийняття рішень в стратегічних іграх та інших ситуаціях, де важливо ураховувати дії опонента. Він дозволяє гравцям аналізувати різні можливі ходи та їх наслідки, щоб вибрати оптимальну стратегію, яка максимізує їх вигоду або мінімізує втрати.

Застосування методу мінімаксу може бути дуже широким і варіюватися від ігрового програмування до управління проектами та фінансовим плануванням. Він є важливим інструментом для багатьох областей, де потрібно приймати стратегічні рішення в умовах конкуренції та невизначеності.

2.3 Інші типи методів

Існують різні альтернативи методу мінімаксу, які використовуються для прийняття рішень у стратегічних іграх та інших ситуаціях, де необхідно враховувати дії опонента. Деякі з них включають:

1. **Алгоритми α-бета відсічення**: Це розширення методу мінімаксу, яке дозволяє зменшити кількість вузлів, які потрібно переглянути у дереві гри, шляхом відсічення непотрібних гілок.

2. **Метод Монте-Карло**: Цей метод використовує випадкову генерацію ходів та оцінку їх результатів для прийняття рішень. Це особливо корисно в іграх з великою кількістю можливих ходів, де метод мінімаксу може бути непрактичним через великий обсяг обчислень.

3. **Метод динамічного програмування**: Цей метод використовується для прийняття рішень в ситуаціях з послідовними діями, де кожне рішення впливає на майбутні можливості. Він дозволяє ефективно розв'язувати задачі з оптимальним керуванням в складних умовах.

4. **Алгоритми засновані на навчанні з підкріпленням**: Ці алгоритми використовуються для прийняття рішень в ситуаціях, де можливі наступні кроки не відомі наперед. Вони використовують методи навчання з підкріпленням для вивчення оптимальної стратегії в процесі гри.

Кожен з цих методів має свої переваги та недоліки і може бути використаний в різних ситуаціях в залежності від конкретних вимог та обмежень задачі.

Алгоритм α-бета відсічення є покращенням методу мінімаксу, призначеним для ефективного пошуку оптимальної стратегії в стратегічних іграх, таких як шахи або гомінь. Він дозволяє відсікати (пропускати) гілки в дереві розглядуваних ходів, які не впливають на оцінку кращого ходу, тим самим значно зменшуючи кількість обчислень, необхідних для пошуку оптимального ходу.

Основна ідея полягає в тому, щоб обмежити діапазон можливих значень (α та β), які можуть приймати оцінки вигоди гравців. Коли алгоритм відслідковує розглянуті ходи, він оновлює значення α та β відповідно до найкращих вигод, які вже були знайдені. Це дозволяє відсікати гілки дерева, коли виявляється, що вони не можуть призвести до кращого результату для гравця.

Алгоритм α-бета відсічення працює ефективно в тому випадку, коли дерево можливих ходів має великий розмір, але багато гілок можна відсікти, не обчислюючи всі можливі ходи. Це робить його важливим інструментом для розв'язання складних ігрових ситуацій, де метод мінімаксу стає недоцільним через велику кількість обчислень.

Метод Монте-Карло - це підхід до прийняття рішень, який використовує випадкову генерацію ходів та оцінку їх результатів для прийняття рішень в умовах невизначеності. Цей метод базується на ідеї використання випадкових експериментів для апроксимації найкращого рішення.

У контексті стратегічних ігор, метод Монте-Карло може бути використаний для оцінки якості можливих ходів, шляхом генерації випадкових ходів та оцінки їх результатів на основі випадкових симуляцій гри. Цей підхід особливо корисний у випадках, коли метод мінімаксу стає непрактичним через велику кількість можливих ходів.

Процес методу Монте-Карло може бути наступним:

1. **Генерація ходів**: Випадковим чином генеруються можливі ходи для гравця або гравців у грі.

2. **Симуляція гри**: Для кожного згенерованого ходу виконується симуляція гри, де решта ходів обирається випадковим чином або згідно певних правил.

3. **Оцінка результатів**: Оцінюється результат кожної симуляції, наприклад, шляхом підрахунку кількості виграшів, втрат або іншої метрики вигоди.

4. **Прийняття рішення**: На основі результатів симуляцій обирається хід з найбільшою очікуваною вигодою для гравця.

Метод Монте-Карло є ефективним інструментом для розв'язання складних ігрових ситуацій, де аналітичні методи можуть бути недостатньо ефективними. Він може бути успішно використаний у великій кількості галузей, включаючи ігри, штучний інтелект, фінанси та багато інших.

Метод динамічного програмування (Dynamic Programming) - це математичний метод прийняття рішень, який використовується для розв'язання проблем оптимізації. Він дозволяє знайти оптимальне рішення в задачах зі складною структурою, шляхом розбиття проблеми на менші підзадачі та ефективного використання результатів попередніх обчислень.

У контексті стратегічних ігор, метод динамічного програмування може бути використаний для прийняття рішень в ситуаціях з послідовними діями, де кожне рішення впливає на майбутні можливості. Основна ідея полягає в тому, щоб визначити оптимальну стратегію для кожного можливого стану гри, використовуючи результати попередніх обчислень.

Процес методу динамічного програмування може бути наступним:

1. **Формулювання проблеми**: Визначення початкових умов, кінцевої мети та можливих дій, які можна виконати.

2. **Розбиття на підзадачі**: Розбиваємо вихідну задачу на менші підзадачі, які можна розв'язати окремо.

3. **Вирішення підзадач**: Знаходимо оптимальні рішення для кожної підзадачі, шляхом рекурсивного обчислення та зберігання проміжних результатів.

4. **Побудова оптимального рішення**: Використовуючи результати розв'язання підзадач, побудовуємо оптимальне рішення для вихідної задачі.

Метод динамічного програмування є потужним інструментом для прийняття рішень в умовах невизначеності та послідовності дій. Він застосовується в багатьох галузях, включаючи ігри, економіку, оптимізацію процесів та багато інших.

Алгоритми засновані на навчанні з підкріпленням (Reinforcement Learning algorithms) - це клас методів машинного навчання, які дозволяють агентові навчатися приймати оптимальні рішення в результаті взаємодії з оточуючим середовищем. Ці алгоритми дозволяють агентові вчитися на основі винагороди (позитивної або негативної зворотного зв'язку), яку він отримує за виконання певних дій.

У контексті стратегічних ігор, алгоритми засновані на навчанні з підкріпленням можуть бути використані для створення програм, які навчаються грати в ігри, приймаючи рішення на основі результатів гри та отриманих винагород. Вони можуть бути особливо корисними в ситуаціях, де правила гри або взаємодія з опонентом можуть змінюватися з часом.

Основні етапи використання алгоритмів заснованих на навчанні з підкріпленням для прийняття рішень включають:

1. **Визначення станів і дій**: Визначення можливих станів гри та дій, які може виконувати агент.

2. **Вибір стратегії**: Агент вибирає стратегію (наприклад, випадкову або згідно певних правил) для взаємодії з оточуючим середовищем.

3. **Отримання винагороди**: Після вибору дії агент отримує винагороду (або штраф) залежно від результату вибраної дії.

4. **Оновлення стратегії**: На основі отриманої винагороди або штрафу агент оновлює свою стратегію, щоб зробити кращі рішення в майбутньому.

5. **Повторення процесу**: Процес вибору дій, отримання винагороди та оновлення стратегії повторюється декілька разів для навчання агента.

Алгоритми засновані на навчанні з підкріпленням є потужним інструментом для прийняття рішень в ситуаціях з невизначеністю та можуть бути успішно використані в різних галузях, включаючи ігри, робототехніку, управління та багато інших.

2.4 Метрики для мінімаксу

Метрики для оцінки алгоритмів, включаючи метод мінімаксу, можуть включати такі показники:

1. **Час виконання**: Час, необхідний для розрахунку оптимального ходу за допомогою методу мінімаксу. Це важливий показник, особливо у великих або складних іграх, де розрахунок може зайняти значну кількість часу.

2. **Пам'ять**: Кількість пам'яті, яка потрібна для збереження даних та проміжних результатів під час розрахунків методом мінімаксу. Великі обсяги пам'яті можуть бути обмежені в деяких обмежених середовищах.

3. **Глибина пошуку**: Кількість ходів вглиб, яку обчислює метод мінімаксу перед тим, як зробити рекомендацію щодо оптимального ходу. Більша глибина пошуку може призвести до більш точних результатів, але також вимагає більше обчислювальних ресурсів.

4. **Якість рішення**: Оцінка того, наскільки ефективно або краще рішення, знайдене методом мінімаксу, в порівнянні з іншими можливими стратегіями. Це може включати виграш або втрату в грі, оцінку якості виконання поставлених цілей тощо.

5. **Ефективність пошуку**: Співвідношення між якістю рішення та ресурсами, які витрачаються для його пошуку (час, пам'ять, обчислювальні ресурси). Важливо знайти баланс між точністю рішення та обмеженістю ресурсів.

Ці метрики можуть бути використані для порівняння різних реалізацій методу мінімаксу або для оцінки його ефективності в різних ігрових сценаріях.

РОЗДІЛ 3. ОГЛЯД ДАНИХ ТА ОПИС ПРОГРАМНОГО ПРОДУКТУ

3.1 Огляд вхідних наборів даних

Вхідні дані для алгоритму мінімаксу включають в себе:

1. **Поточний стан гри**: Це представлення поточного стану гри, включаючи розміщення фігур, очки, можливі ходи та будь-яку іншу інформацію, необхідну для прийняття рішення.

2. **Правила гри**: Правила гри визначають, які ходи можливі у кожному стані гри, які наслідки має кожен хід і як визначається переможець або результат гри.

3. **Глибина пошуку**: Це кількість ходів, які алгоритм мінімаксу розглядає вглиб перед тим, як прийняти рішення. Чим більша глибина пошуку, тим більше обчислювальних ресурсів потрібно для розрахунку.

4. **Оцінка станів гри**: Це функція, яка оцінює вигоду гравця у кожному стані гри. Вона може бути визначена аналітично або емпірично, в залежності від конкретного випадку.

5. **Алгоритм оптимізації**: Метод мінімаксу може використовувати різні алгоритми для оптимізації пошуку оптимального рішення, такі як α-бета відсічення або методи оптимізації пошуку.

Ці вхідні дані дозволяють алгоритму мінімаксу аналізувати гру, визначати можливі ходи, розраховувати наслідки кожного ходу та приймати рішення щодо оптимального ходу для гравця.

Поточний стан гри - це представлення всієї інформації про ситуацію в грі в певний момент часу. Це включає в себе розміщення фігур, стан дошки, кількість очок, можливі ходи для кожного гравця та будь-яку іншу важливу інформацію, що впливає на подальший розвиток гри.

Наприклад, у шахах поточний стан гри включає в себе розташування фігур на дошці, чий хід зараз, чи є можливість рокіровки, чи можна зробити "ен пассант" та інші правила. У грі Tic Tac Toe (крестики-нулики), поточний стан може бути представлений матрицею 3x3, де кожен елемент вказує, хто поставив свою фігуру в цю клітинку (X, O або пусто).

Залежно від конкретної гри, поточний стан може мати різне представлення, але завжди містить інформацію, яка необхідна для алгоритму мінімаксу для аналізу та визначення найкращого ходу для гравця.

Добре, розглянемо правила гри в шахи:

1. **Розташування фігур**: Гра відбувається на дошці 8x8, на якій розміщені шашки двох кольорів: білих і чорних. Кожен гравець має 16 фігур, включаючи пішаків, тури, коней, слонів, ферзя та короля.

2. **Ходи**: Гравці ходять по черзі. Кожна фігура має свої власні правила ходу, наприклад, пішаки можуть ходити вперед на одне поле, але за першим ходом вони можуть піти на два поля. Король може рухатися на одне поле в будь-якому напрямку, а ферзь - на будь-яку кількість полів у будь-якому напрямку.

3. **Ціль гри**: Метою гри є поставити супротивника в шахматному маті (коли король противника під атакою і йому немає можливості уникнути захоплення) або в шахматному паті (коли король противника не під атакою, але він не має можливості зробити хід).

4. **Виграш і поразка**: Якщо гравець досягає мети - ставить короля противника в мат або примушує його до пату - цей гравець перемагає. Якщо гравець не може зробити ходу (шахматний пат), гра закінчується у нічию.

На основі цих правил ми можемо розробити алгоритм мінімаксу для гри в шахи. Після того, як ми матимемо допустимі ходи для певного стану гри, ми можемо рекурсивно викликати алгоритм мінімаксу для кожного можливого наступного кроку, обчислюючи оцінку для кожного з них і обираючи оптимальний хід для гравця.

Для застосування алгоритму мінімаксу нам потрібно знати поточний стан гри. Давайте розглянемо приклад початкового стану гри в шахах:

```
8 | r  n  b  q  k  b  n  r
7 | p  p  p  p  p  p  p  p
6 | .  .  .  .  .  .  .  .
```

5 | . . . . . .

4 | . . . . . .

3 | . . . . . .

2 | P P P P P P P P

1 | R N B Q K B N R

  a b c d e f g h

```

У цьому прикладі:

- `r`, `n`, `b`, `q`, `k`, `p` - фігури білого гравця (король, ферзь, слон, конь, тура, пішак відповідно).

- `R`, `N`, `B`, `Q`, `K`, `P` - фігури чорного гравця.

- Крапки (`.`) позначають порожні клітини на дошці.

Цей стан гри показує початкове розташування фігур на шаховій дошці перед початком гри. Тепер ми можемо використати цей поточний стан гри для аналізу за допомогою алгоритму мінімаксу, щоб прийняти рішення щодо найкращого ходу для білого гравця.

Глибина пошуку - це параметр, який визначає, на скільки ходів вперед алгоритм мінімакс буде розглядати можливі варіанти. Вибір оптимальної глибини пошуку залежить від складності гри, обчислювальних ресурсів і часу, доступного для прийняття рішення.

Зазвичай глибина пошуку обмежується певним числом ходів, наприклад, 2, 3, 4 тощо. Чим більша глибина пошуку, тим більш точне рішення може бути прийняте алгоритмом мінімакс, але це також призводить до збільшення часу обчислень.

За замовчуванням глибина пошуку може бути встановлена на значення, яке може забезпечити прийнятний рівень продуктивності та точності. Однак у деяких випадках може бути варто налаштувати цей параметр вручну, особливо якщо у вас є обмежені обчислювальні ресурси або якщо ви знаєте, що глибина пошуку може суттєво вплинути на якість рішення.

Оцінка стану гри - це числова оцінка, яка визначає виграшність або програшність даного стану гри для конкретного гравця. В шахах ця оцінка може бути базована на різних факторах, таких як кількість та якість фігур, контрольовані позиції на дошці, можливість зробити матуючий хід тощо.
```

Наприклад, можна призначити числові значення кожній фігурі (пішаку - 1, коню - 3, слону - 3, турі - 5, ферзю - 9, королю - 100), а потім просто підсумувати ці значення для всіх фігур на дошці. Чим більше сума для гравця, тим кращий його стан гри.

Проте оцінка стану гри може бути і більш складною. Наприклад, вона може враховувати контрольовані позиції на дошці, потенційні загрози та можливості для атаки. Чим більше факторів враховується, тим точніша може бути оцінка стану гри.

Важливо також враховувати величезну кількість можливих станів гри, тому точні оцінки стану гри можуть бути складним завданням. Алгоритм мінімакс буде використовувати ці оцінки для вибору кращих ходів, які призведуть до максимізації виграшу гравця.

Якщо ви шукаєте алгоритм оптимізації для використання в контексті алгоритму мінімакс у грі в шахи або подібному середовищі, важливо знати, що алгоритм мінімакс сам по собі є алгоритмом оптимізації, оскільки він спрямований на знаходження оптимального ходу для гравця в кожній ситуації гри.

Однак, існують підходи для оптимізації алгоритму мінімакс, які допомагають покращити його продуктивність та ефективність, зокрема:

1. **Відсічення (Pruning)**: Алгоритми відсічення, такі як альфа-бета відсічення, дозволяють викинути гілки пошуку, які не впливають на результат. Це дозволяє значно зменшити кількість розгляданих станів гри та покращити продуктивність алгоритму.

2. **Кешування (Caching)**: Зберігання результатів обчислень для певних станів гри дозволяє уникнути повторного обчислення в майбутньому, що допомагає зменшити час виконання алгоритму.

3. **Паралельне обчислення (Parallel Computing)**: Розподілення обчислень між декількома обчислювальними ресурсами може значно прискорити алгоритм мінімаксу.

4. **Використання евристик (Heuristics)**: Деякі евристики можуть допомогти зменшити кількість станів гри, які потрібно розглядати, або прискорити процес визначення оцінки стану гри.

5. **Оптимізовані структури даних (Optimized Data Structures)**: Використання ефективних структур даних для збереження інформації про стан гри може допомогти зменшити час доступу та обробки.

Застосування цих підходів може значно покращити продуктивність алгоритму мінімакс та зробити його більш ефективним у розрахунку оптимальних ходів у грі.

3.2 Підготовка даних

Підготовка даних до використання в алгоритмі мінімакс може включати в себе наступні кроки:

1. **Представлення стану гри**: Дані про поточний стан гри (наприклад, шахова дошка) потрібно представити у вигляді, зрозумілому для комп'ютера. Це може бути матриця, список або будь-яка інша структура даних, яка дозволяє легко представляти розташування фігур, статуси клітинок і т.д.

2. **Можливі ходи**: Для кожного гравця потрібно знайти всі можливі ходи, які він може зробити у поточному стані гри. Це може бути реалізовано шляхом аналізу правил гри та поточного розташування фігур.

3. **Оцінка стану гри**: Реалізуйте алгоритм для оцінки стану гри. Це може включати в себе розрахунок числових значень для кожної фігури на дошці, а також оцінку інших факторів, таких як контроль позицій, загрози та можливості для атаки.

4. **Завершення гри**: Необхідно реалізувати умови, за яких гра вважається завершеною, такі як мат або пат. Це дозволяє алгоритму мінімакс зупинятися у відповідний момент.

5. **Обробка результатів**: Після обчислення оцінок для кожного можливого ходу, алгоритм мінімакс повинен обрати найкращий хід для поточного гравця.

Після цих кроків дані будуть готові для використання в алгоритмі мінімакс для прийняття рішень в грі.

Представлення стану гри в шахах може бути реалізовано за допомогою структури даних, яка відображає розташування фігур на дошці. Одним з можливих способів представлення є використання двовимірного масиву (матриці), де кожна клітина відповідає одному квадрату на шаховій дошці.

Наприклад, можемо використовувати мову програмування Python для створення такого представлення:

```python
# Представлення стану гри в шахах за допомогою матриці
class ChessBoard:
    def __init__(self):
        self.board = [
            ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'],
            ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['.', '.', '.', '.', '.', '.', '.', '.'],
            ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
            ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r']
        ]

    def print_board(self):
        for row in self.board:
            print(' '.join(row))

# Приклад використання
board = ChessBoard()
board.print_board()
```

У цьому коді ми створюємо клас `ChessBoard`, який містить матрицю `board`, що відображає розташування фігур на дошці. Символи `R`, `N`, `B`, `Q`, `K`, `P` представляють фігури білого

гравця, а `r`, `n`, `b`, `q`, `k`, `p` - фігури чорного гравця. Крапки (`.`) позначають порожні клітини на дошці.

Це лише один з можливих способів представлення стану гри в шахах, інші методи можуть використовувати інші структури даних або формати залежно від вимог вашого проекту.

Для реалізації функції, що знаходить всі можливі ходи для конкретного гравця в поточному стані гри в шахах, нам потрібно розглянути правила кожного типу фігури та перевірити їх можливі ходи.

Ось загальний підхід до знаходження можливих ходів для кожного типу фігури:

1. **Пішаки (Pawns)**: Пішаки можуть рухатися вперед на одну клітину, але на початку гри вони можуть рухатися на дві клітини. Вони можуть брати фігури по діагоналі, але не можуть перейти через фігури на своєму шляху.

2. **Тури (Rooks)**: Тури можуть рухатися по горизонталі та вертикалі на будь-яку кількість вільних клітин.

3. **Коні (Knights)**: Коні рухаються "Л"-подібно: два кліки в одному напрямку, а потім один клік в перпендикулярному напрямку або навпаки. Вони можуть перескакувати через фігури.

4. **Слони (Bishops)**: Слони можуть рухатися по діагоналі на будь-яку кількість вільних клітин.

5. **Ферзі (Queens)**: Ферзі можуть рухатися як тури і слони: по горизонталі, вертикалі та діагоналі на будь-яку кількість вільних клітин.

6. **Королі (Kings)**: Королі можуть рухатися на одну клітину в будь-якому напрямку.

Ураховуючи ці правила, можна написати функцію для кожного типу фігури, яка знаходить всі можливі ходи. Після цього можна об'єднати всі ці функції в одну, яка знаходить всі можливі ходи для всіх фігур на дошці.

Оцінка стану гри в алгоритмі мінімакс важлива для визначення виграшності або програшності даного стану гри для конкретного гравця. У шахах оцінка стану гри може бути здійснена шляхом

оцінки позицій фігур на дошці та інших факторів, таких як контрольовані позиції, потенційні загрози та можливості для атаки.

Ось кілька загальних підходів до оцінки стану гри в шахах:

1. **Матеріальна оцінка**: Сума значень фігур на дошці. Зазвичай, кожній фігурі надається певне числове значення, наприклад: пішак - 1, конь - 3, слон - 3, тура - 5, ферзь - 9, король - 100. Сума цих значень для кожного гравця може служити оцінкою його позиції.

2. **Контроль позицій**: Оцінка того, скільки клітинок контролює кожен гравець. Гравець, який контролює більше клітинок, має перевагу.

3. **Потенційні загрози та можливості для атаки**: Оцінка потенційних загроз та можливостей для атаки для кожного гравця. Наприклад, якщо гравець може зробити матуючий хід на наступному кроці, його оцінка стану гри буде дуже високою.

4. **Позиції королів**: Безпека королів гравців також може бути важливим фактором. Небезпечна позиція короля може призвести до швидкого закінчення гри.

Це лише деякі загальні підходи до оцінки стану гри в шахах. Фактичний вибір оцінки може залежати від специфічних вимог вашого додатка чи ігри.

Завершення гри в шахах може статися з різних причин, але найбільш поширеними є мат, пат або проголосування гравців про нічию. Давайте розглянемо кожен з цих випадків:

1. **Мат (Checkmate)**: Мат відбувається, коли король одного з гравців опиняється під ударом (в шаху) і немає можливості зробити такий хід, який би захистив його від атаки. У такому випадку гра закінчується перемогою протилежного гравця.

2. **Пат (Stalemate)**: Пат виникає, коли король одного з гравців не перебуває в шаху, але він також не має можливості зробити жодного ходу, який би не виставив його під удар. У такому випадку гра закінчується нічиєю.

3. **Нічия за згодою гравців (Draw by Agreement)**: Гравці можуть домовитися про нічию в будь-який момент під час гри. Це може статися, якщо обидва гравці вважають, що немає шляху до перемоги або якщо гра перетягується.

4. **Нічия за недостатньою кількістю фігур (Insufficient Material Draw)**: Це випадок, коли жоден з гравців не має достатньої кількості фігур для досягнення мату. Наприклад, якщо на дошці залишаються лише дві тури, тура та король протилежного кольору, гра закінчується нічиєю.

Визначення кінця гри може бути важливим кроком у вашому алгоритмі мінімаксу, оскільки це дозволить завершити рекурсивний пошук та визначити остаточну оцінку стану гри.

Обробка результатів в алгоритмі мінімакс полягає в тому, щоб визначити найкращий хід для поточного гравця, враховуючи оцінки стану гри, отримані з рекурсивного аналізу можливих ходів.

Основні кроки обробки результатів можуть включати такі дії:

1. **Вибір найкращого ходу**: Обчисліть оцінку для кожного можливого ходу за допомогою алгоритму мінімаксу. Потім оберіть хід з найвищою або найнижчою оцінкою в залежності від того, який гравець здійснює хід.

2. **Повернення результату**: Поверніть обраний хід як результат роботи алгоритму мінімаксу. Цей хід буде використаний для здійснення наступного кроку у грі.

3. **Оцінка стану гри**: Якщо поточний стан гри веде до завершення гри (наприклад, мат або пат), поверніть відповідну оцінку (наприклад, +безкінечність для перемоги, -безкінечність для програшу) для використання в оцінці стану гри у попередній рекурсивній виклик.

4. **Відсічення (Pruning)**: Застосуйте методи відсічення, такі як альфа-бета відсічення, для скорочення кількості обчислень та поліпшення продуктивності алгоритму.

5. **Кешування (Caching)**: Зберігайте результати обчислень для певних станів гри, щоб уникнути повторного обчислення тих самих станів у майбутньому.

Ці кроки допомагають забезпечити ефективну обробку результатів в алгоритмі мінімакс та вибір оптимального ходу для гравця.

3.3 Опис програми

Опис програми для алгоритму мінімакс у грі в шахи може включати наступні елементи:

1. **Класи та функції для представлення гри**: Створіть класи та функції для представлення шахової дошки, фігур, гравців та правил гри. Це може включати функції для руху фігур, перевірки легальності ходу та перевірки стану гри (мат, пат, нічия).

2. **Алгоритм мінімакс**: Реалізуйте алгоритм мінімаксу для прийняття рішень у грі. Це включає функції для обчислення оцінок стану гри, рекурсивного пошуку можливих ходів та обробки результатів.

3. **Можливі ходи та оцінка стану гри**: Напишіть функції для знаходження всіх можливих ходів для гравця та оцінки стану гри на основі розташування фігур на дошці.

4. **Взаємодія з користувачем**: Реалізуйте інтерфейс користувача для взаємодії з грою. Це може бути текстовий інтерфейс командного рядка або графічний інтерфейс, що відображає шахову дошку та дозволяє гравцям робити ходи.

5. **Додаткові функції та класи**: Реалізуйте будь-які додаткові функції або класи, які можуть бути потрібні для вашого проекту, такі як збереження гри, логування ходів, реалізація іншого типу алгоритму для покращення продуктивності або ефективності.

Програма може бути написана на будь-якій мові програмування залежно від вашого вибору та вимог вашого проекту. Важливо створити добре структурований код з чіткою логікою та коментарями для полегшення розуміння та підтримки програми.

Для представлення гри в шахах можна створити класи для шахової дошки, фігур та гравців. Нижче наведено загальні класи та функції, які можна використовувати для цього:

```python
class ChessBoard:
    def __init__(self):
```

```
        self.board = self.initialize_board()

    def initialize_board(self):
        # Створення та ініціалізація шахової дошки
        pass

    def move_piece(self, start_position, end_position):
        # Рух фігури на дошці
        pass

    def is_checkmate(self):
        # Перевірка на мат
        pass

    def is_stalemate(self):
        # Перевірка на пат
        pass

    def print_board(self):
        # Виведення шахової дошки
        pass

class Piece:
    def __init__(self, color):
        self.color = color
        self.position = None

    def move(self, new_position):
        # Рух фігури
        pass
```

```
    def get_possible_moves(self):
        # Отримання можливих ходів фігури
        pass

class Player:
    def __init__(self, color):
        self.color = color

    def make_move(self):
        # Зробити хід гравця
        pass
```

Це загальна структура класів, яка може бути розширена та налаштована відповідно до ваших потреб. Наприклад, клас `ChessBoard` може містити методи для перевірки правил гри, таких як рокіровка, просунення пішака на дві клітини з початкової позиції та інші. Клас `Piece` може мати підкласи для кожного типу фігури (пішак, тура, конь тощо), які реалізують специфічні правила для кожної фігури. Клас `Player` може також містити методи для взаємодії з гравцем, такі як отримання введених даних для зроблення ходу.

Це лише загальна концепція, і реалізація може відрізнятися в залежності від специфічних вимог вашого проекту та вибраної мови програмування.

Алгоритм мінімакс є основним алгоритмом в штучному інтелекті для прийняття рішень у розподілених стратегічних іграх, таких як шахи. Основна ідея полягає у тому, щоб обирати кращий хід, спираючись на прогнозовані наслідки кожного ходу.

Нижче наведено загальний опис алгоритму мінімакс:

1. **Рекурсивний пошук**: Починаючи з поточного стану гри, рекурсивно розглядаються всі можливі ходи для гравців у межах обмеженого глибини дерева пошуку.

2. **Оцінка стану гри**: Для кожного стану гри в кінці глибини пошуку або для тих станів, які ведуть до завершення гри, обчислюється оцінка, що відображає виграшність гравця у цьому стані.

3. **Максимізація та мінімізація**: Гравці чергуються у максимізації та мінімізації оцінок. Максимізуючий гравець (наприклад, гравець, який хоче максимізувати свої вигоди, зазвичай гравець, що робить поточний хід) обирає хід, який має найбільшу оцінку, тоді як мінімізуючий гравець (гравець, який хоче мінімізувати вигоди максимізуючого гравця) обирає хід, який має найменшу оцінку.

4. **Повернення результату**: Після завершення рекурсивного пошуку обирається найкращий хід для поточного гравця.

5. **Відсічення (Pruning)**: Для покращення продуктивності алгоритму можуть використовуватися методи відсічення, такі як альфа-бета відсічення, для скорочення кількості розглянутих станів гри.

6. **Глибина пошуку**: Чим більше глибина пошуку, тим точніша буде оцінка стану гри, але це може призвести до збільшення обчислювального часу. Тому необхідно збалансувати точність та продуктивність, обираючи оптимальну глибину пошуку.

Це загальний огляд алгоритму мінімакс. При його реалізації важливо враховувати конкретні правила гри та специфічні особливості вашого проекту для досягнення найкращої продуктивності та ефективності.

Для написання функцій для знаходження всіх можливих ходів та оцінки стану гри у шахах, нам спочатку потрібно мати клас для представлення шахової дошки та фігур. Ось загальна структура коду для цього:

```python
class ChessBoard:
    def __init__(self):
        self.board = self.initialize_board()

    def initialize_board(self):
```

```
        # Створення та ініціалізація шахової дошки
        pass

    def get_possible_moves(self, color):
        # Отримання всіх можливих ходів для гравця заданого кольору
        pass

    def evaluate_board(self, color):
        # Оцінка стану гри на основі розташування фігур на дошці
        pass

class Piece:
    def __init__(self, color):
        self.color = color
        self.position = None

    def get_possible_moves(self, board):
        # Отримання всіх можливих ходів для фігури на заданій дошці
        pass

    def is_valid_move(self, move, board):
        # Перевірка, чи є хід допустимим для фігури на заданій дошці
        pass
```

Тепер розглянемо реалізацію цих методів:

```python
class ChessBoard:
    # ... інші методи класу ...
```

```
    def get_possible_moves(self, color):
        possible_moves = []
        for row in range(8):
            for col in range(8):
                piece = self.board[row][col]
                if piece is not None and piece.color == color:
                    moves = piece.get_possible_moves(self)
                    for move in moves:
                        possible_moves.append(((row, col), move))
        return possible_moves

    def evaluate_board(self, color):
        # Простий приклад оцінки: сума значень фігур на дошці
        evaluation = 0
        for row in range(8):
            for col in range(8):
                piece = self.board[row][col]
                if piece is not None:
                    value = 1 if piece.color == color else -1
                    evaluation += value * piece.value  # Припущення: кожній фігурі присвоєне числове значення
        return evaluation

class Piece:
    # ... інші методи класу ...

    def get_possible_moves(self, board):
        possible_moves = []
        for move in self.get_legal_moves(board):
```

```
            if self.is_valid_move(move, board):
                possible_moves.append(move)
        return possible_moves

    def is_valid_move(self, move, board):
        # Перевірка, чи є хід допустимим для фігури на заданій дошці
        pass
```

Це загальний підхід до написання функцій для знаходження всіх можливих ходів для гравця та оцінки стану гри на основі розташування фігур на дошці у шахах. Вам може знадобитися налаштувати ці методи залежно від конкретних правил вашої гри та ваших потреб.

Реалізація інтерфейсу користувача може бути здійснена за допомогою текстового інтерфейсу командного рядка. Нижче наведено приклад простого текстового інтерфейсу для взаємодії з гравцями у грі в шахи:

```python
class ChessGame:
    def __init__(self):
        self.board = ChessBoard()
        self.current_player = 'white'

    def start(self):
        print("Welcome to Chess!")
        while True:
            self.board.print_board()
            print(f"It's {self.current_player}'s turn.")
            move = input("Enter your move (e.g., 'e2 e4'): ")
            if self.is_valid_move(move):
                self.board.move_piece(move)
                self.current_player = 'black' if self.current_player == 'white' else 'white'
```

```
            else:
                print("Invalid move. Please try again.")

    def is_valid_move(self, move):
        # Перевірка валідності ходу
        pass

    # Додаткові методи, які можуть знадобитися
    # def is_checkmate(self):
    # def is_stalemate(self):

# Початок гри
game = ChessGame()
game.start()
```

У цьому простому текстовому інтерфейсі гравцям пропонується вводити ходи за допомогою введення з клавіатури. Після кожного ходу виводиться поточний стан дошки, а гравці змінюються через кожен хід.

Цей приклад може бути розширений залежно від ваших потреб. Наприклад, ви можете додати функціональність для перевірки на мат або пат після кожного ходу, або ви можете реалізувати більш складні перевірки валідності ходу. Також можна розробити графічний інтерфейс за допомогою бібліотек, таких як Tkinter для Python або Unity для створення ігор.

Додаткові функції та класи можуть значно поліпшити ваш проект гри в шахи. Ось декілька ідей, які можна реалізувати:

1. **Збереження гри**: Додайте можливість збереження поточного стану гри у файл. Це дозволить гравцям продовжувати гру в майбутньому з того ж місця, де вони зупинилися.

2. **Завантаження гри**: Реалізуйте можливість завантаження збережених ігор для продовження гри з попереднього стану.

3. **Логування ходів**: Додайте систему логування ходів, яка буде записувати всі ходи гравців у спеціальний файл або базу даних. Це може бути корисно для аналізу гри після її завершення.

4. **Інші алгоритми**: Реалізуйте інші алгоритми для прийняття рішень, такі як альфа-бета відсічення, які можуть покращити продуктивність або ефективність вашої програми.

5. **Інтелект штучного інтелекту для комп'ютерних гравців**: Додайте функціонал для створення комп'ютерних гравців зі штучним інтелектом, які можуть змагатися з реальними гравцями або один з одним.

6. **Статистика ігор**: Зберіть та відображайте статистику ігор, таку як кількість перемог, поразок, нічиїх тощо, для кожного гравця.

7. **Покращений інтерфейс користувача**: Розширте функціонал графічного інтерфейсу, додавши можливість налаштування вигляду та інші зручні функції.

З додатковими функціями та класами ваша програма може стати більш розширеною та корисною для користувачів.

3.4 Висновки

Висновки:

1. **Шахи як складна гра**: Гра в шахи є складною інтелектуальною грою, яка вимагає стратегічного мислення, планування та аналізу.

2. **Алгоритм мінімакс для прийняття рішень**: Алгоритм мінімакс є основним методом для прийняття рішень у грі, де гравці роблять ходи в змінному середовищі.

3. **Реалізація програми**: Реалізація програми для гри в шахи включає класи та функції для представлення гри, взаємодії з користувачем, алгоритму мінімакс та додаткових функцій.

4. **Покращення програми**: Додаткові функції, такі як збереження гри, логування ходів та реалізація інших алгоритмів, можуть значно покращити вашу програму та забезпечити більшу користь для користувачів.

Гра в шахи відображає важливі аспекти стратегічного мислення та розробки програмного забезпечення, і програма для гри в шахи може бути відмінною платформою для вивчення і вдосконалення цих навичок.

РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

В заданому розділі буде проведено оцінювання основних характеристик для майбутнього програмного продукту, що спеціалізується на дослідженні демографічного стану.

Дана реалізація буде сприяти проведенню усіх необхідних досліджень, що дасть змогу якісно дослідити питання не лише в Україні, проте у всьому світі.

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

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

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

4.1 Постановка задачі проектування

У роботі застосовується метод функціонально-вартісного аналізу для проведення техніко-економічного аналізу розробки системи розпізнавання рентгенівських зображень грудної клітини. Оскільки рішення стосовно проектування та реалізації компонентів, що розробляється, впливають на всю систему, кожна окрема підсистема має її задовольняти. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, призначеного для збору, обробки та проведення аналізу даних по компанії.

Технічні вимоги до програмного продукту є наступні:

- функціонування на персональних комп'ютерах із стандартним набором компонентів;
- зручність та зрозумілість для користувача;
- швидкість обробки даних та доступ до інформації в реальному часі;
- можливість зручного масштабування та обслуговування;
- мінімальні витрати на впровадження програмного продукту.

4.2 Обгрунтування функцій програмного продукту

Головна функція F_0 – розробка можливого програмного продукту, яка дозволяє аналізувати різні характеристики, що безпосередньо впливають на стійкість підприємства. Беручи за основу цю функцію, можна виділити наступні:

F_1 – вибір мови програмування.

F_2 – вибір способу реалізації алгоритмів.

–

F_3 – вибір середовища розробки.

Кожна з цих функцій має декілька варіантів реалізації:

Функція F_1:

а) Python.

б) C/C++.

Функція F_2.

а) Використання готових бібліотек.

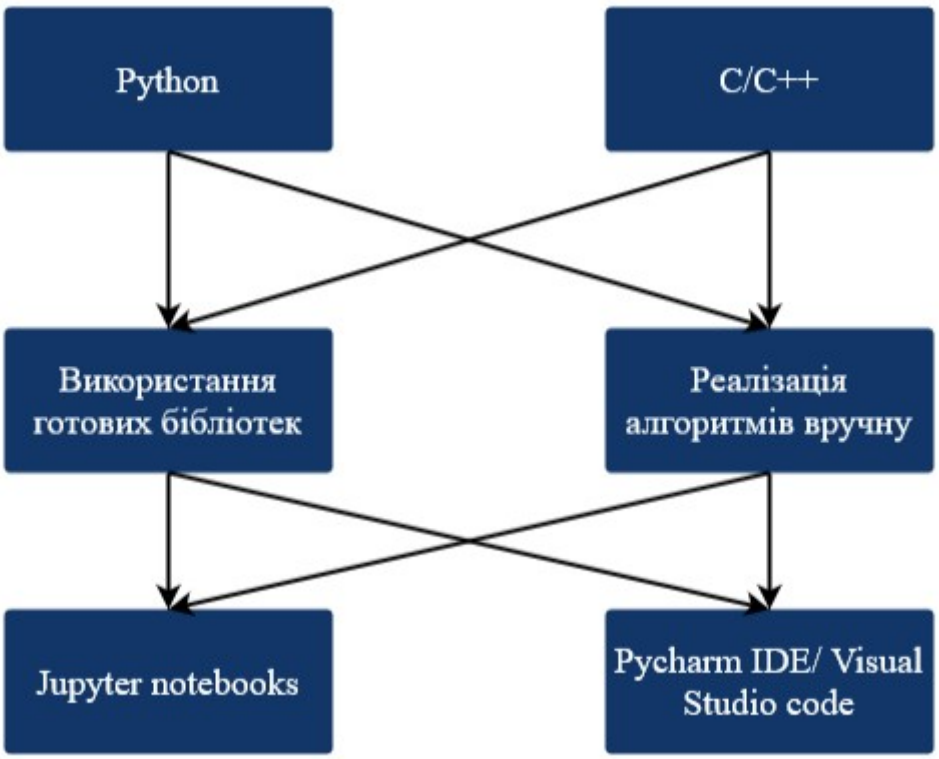
б) Реалізація алгоритмів вручну.

Функція F_3:

а) Середовище розробки – Jupyter Notebook.

б) Середовище розробки – Pycharm IDE/VS Code.

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



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

Таблиця 4.1 - Позитивно-негативна матриця

Функції	Варіанти реалізації	Переваги	Недоліки
F_1	*А*	Простота, доступність готових бібліотек	Швидкодія
	Б	Час виконання програми	Складність, потребує більше часу для написання програми
F_2	*А*	Доступність та легкість при написанні	Менша гнучкість
	Б	Реалізація саме того функціоналу, який потрібен для програми	Додатковий час на реалізацію, можливі помилки
F_3	*А*	Для кожної клітини можна обрати нову мову програмування	Відсутність дебагу коду
	Б	Багато інструментів, інтеграція з іншими сервісами	Підтримує лише одну мову програмування

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

Функція F_1:

Перевагу даємо загальнодоступності. Для спрощення роботи по написанню коду варіант Б має бути відкинутий.

Функція F_2:

Готові бібліотеки для мови програмування Python є зручними та оптимізованими, тому варіант Б має бути відкинутий.

Функція F_3:

Обидва варіанти можна використати у розробці.

Таким чином, будемо розглядати такий варіанти реалізації ПП:

F_1а – F_2а – F_3а

F_1а – F_2а – F_3б

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

4.3 Обгрунтування системи параметрів програмного продукту

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

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

- *X1* – швидкодія мови програмування;
- *X2* – об'єм пам'яті для обчислень та збереження даних;
- *X3* – час навчання даних;
- *X4* – потенційний об'єм програмного коду.

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

Таблиця 4.2 - Основні параметри програмного продукту

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Швидкодія мови програмування	X1	оп/мс	85	110	160
Об'єм пам'яті	X2	Мб	100	80	50
Час попередньої обробки даних	X3	мс	120	85	60
Потенційний об'єм програмного коду	X4	кількість рядків коду	1000	650	500

За даними таблиці 4.2 будуються графічні характеристики параметрів – рис. 4.2 – рис. 4.5.

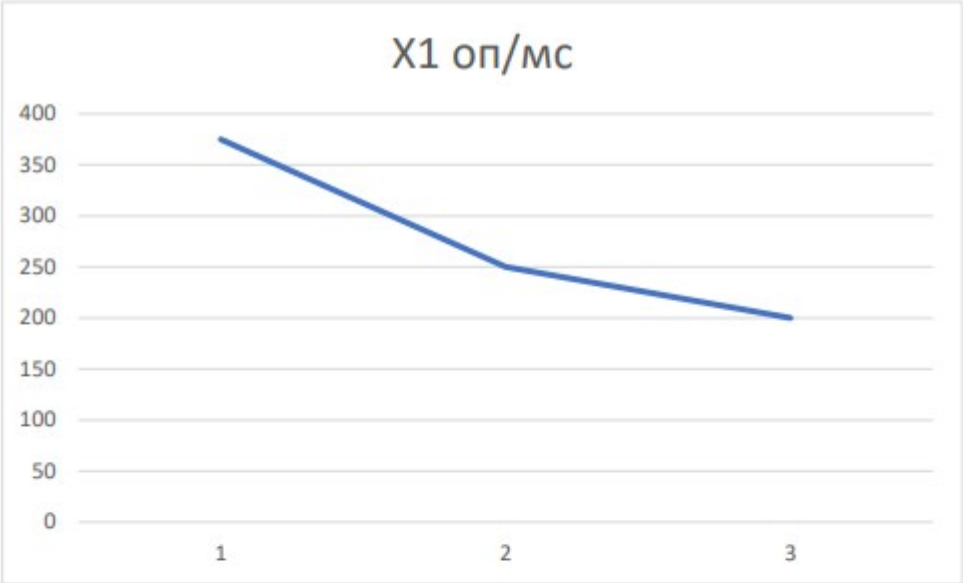


Рисунок 4.2 – X1, швидкодія мови програмування

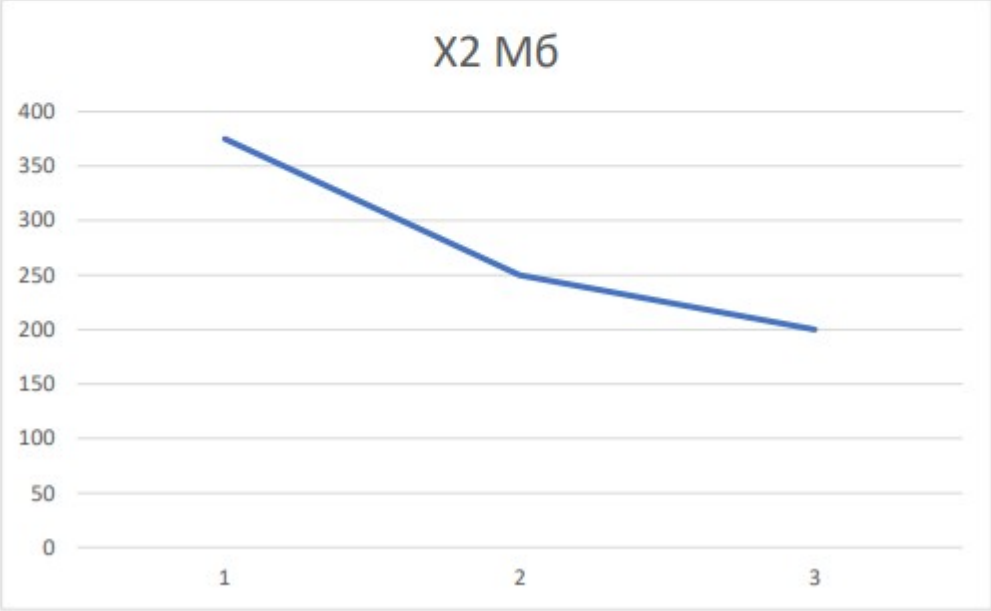


Рисунок 4.3 – X2, об’єм пам’яті

–

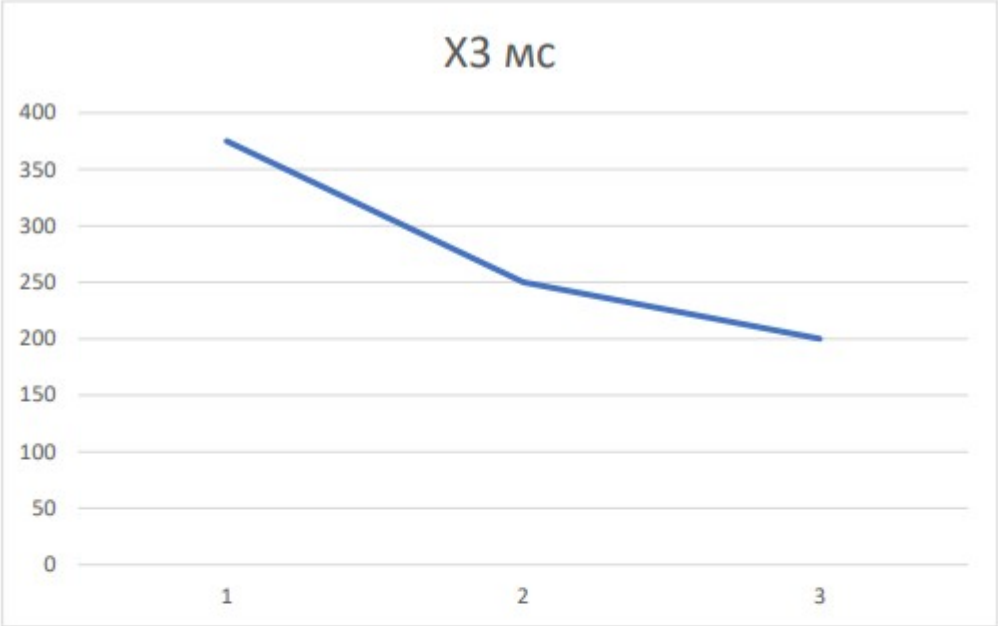


Рисунок 4.4 – X3, час попередньої обробки даних

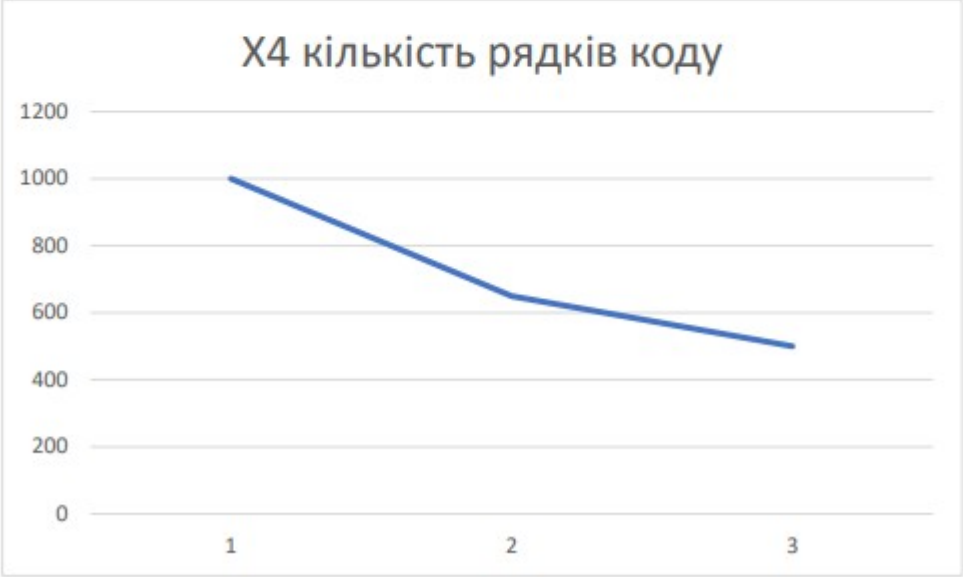


Рисунок 4.5 – X4, потенційний об’єм програмного коду

4.4 Аналіз експертного оцінювання параметрів

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

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

- визначення рівня значимості параметра шляхом присвоєння різних рангів;
- перевірку придатності експертних оцінок для подальшого використання;
- визначення оцінки попарного пріоритету параметрів;
- обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 4.3.

Таблиця 4.3 - Результати ранжування параметрів

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта						Сума рангів R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6			
X1	Швидкодія мови програмування	Оп/мс	2	2	1	1	1	2	9	-6	36
X2	Об’єм пам’яті	Mb	3	3	5	5	4	4	24	9	81

Продовження таблиці 4.3

X3	Час попередньої обробки даних	мс	3	1	1	2	2	2	11	-4	16
X4	Потенційний об'єм програмного коду	Кількість рядків коду	2	4	3	2	3	2	16	1	1
	Разом		10	10	10	10	10	10	60	0	134

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

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

$$R_i = \sum_{i=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 60, \qquad (4.1)$$

де N – число експертів

$$\Delta_i = R_i - T. \quad (4.3)$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 134. \quad (4.4)$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 134}{6^2(4^3 - 4)} = 0{,}744 > W_k = 0{,}67. \quad (4.5)$$

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

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

Таблиця 4.4 - Попарне порівняння параметрів.

Параметри	Експерти						Кінцева оцінка	Числове значення
	1	2	3	4	5	6		
X1 і X2	<	<	<	<	<	<	<	0,5

Продовження таблиці 4.4

X1 і X3	<	>	=	<	<	=	<	0,5
X1 і X4	=	<	<	<	<	=	<	0,5
X2 і X3	=	>	>	>	>	>	>	1,5
X2 і X4	>	<	>	>	>	>	>	1,5
X3 і X4	>	<	<	=	<	=	<	0,5

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

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

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

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

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

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

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%). На

другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

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

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

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

Таблиця 4.5 - Розрахунок вагомості параметрів

Параметри x_i	Параметри x_j				Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	X4	b_i	$K_{\text{ві}}$	b_i^1	$K_{\text{ві}}^1$	b_i^2	$K_{\text{ві}}^2$
X1	1	0,5	0,5	0,5	2,5	0,16	9,25	0,16	34,125	0,16
X2	1,5	1	1,5	1,5	5,5	0,34	21,25	0,35	77,875	0,36
X3	1,5	0,5	1	0,5	3,5	0,22	12,25	0,21	41,875	0,2
X4	1,5	0,5	1,5	1	4,5	0,28	16,25	0,28	59,125	0,28
Всього:					16	1	59	1	213	1

4.5 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів) *X1* (швидкість роботи мови програми) , *X2* (Об'єм пам'яті) відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра *X4* (потенційний об'єм програмного коду) обрано не найгіршим.

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

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

де n – кількість параметрів;

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

B_i – оцінка i–го параметра в балах.

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

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	A	X1	110	5	0,16	0,8

–

Продовження таблиці 4.6

F2	А	X2	80	3	0,36	1,08
F3	А	X4	800	7	0,2	1,4
	Б	X4	600	4	0,2	0,8

За даними з таблиці 4.6 за формулою:

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

визначаємо рівень якості кожного з варіантів:

$$K_{K1} = 0{,}8 + 1{,}08 + 1{,}4 = 3{,}28;$$

$$K_{K2} = 0{,}8 + 1{,}08 + 0{,}8 = 2{,}68.$$

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

4.6 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

–

1. Розробка проекту програмного продукту;

2. Розробка програмної оболонки;

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

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

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

Загальна трудомісткість обчислюється як:

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_М \cdot K_{СТ} \cdot K_{СТ.М}, \quad (4.13)$$

де $T_Р$ – трудомісткість розробки ПП;

$К_П$ – поправочний коефіцієнт;

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

$К_М$ – коефіцієнт рівня мови програмування;

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

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

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: $T_Р$ =90 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для

першого завдання: $К_П$ = 1.8. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: $К_{СК}$ = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $К_{СТ}$ = 0.9. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$Т_1 = 90 \cdot 1.8 \cdot 0.9 = 145,8 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто $Т_Р$ =29 людино-днів, $К_П$ =0.9, $К_{СК}$ = 1, $К_{СТ}$ =0.8:

$T_э = 29 \cdot 0.9 \cdot 0.8 = 20.88$ людино-днів

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

$$T_I = (145{,}8 + 20.88 + 4.8) \cdot 8 = 1371{,}84 \text{ людино-годин.}$$

$$T_{II} = (145{,}8 + 20.88 + 7.2) \cdot 8 = 1391{,}04 \text{ людино-годин.}$$

Найбільш високу трудомісткість має варіант II.

В розробці беруть участь два програмісти з окладом 22300 грн., один аналітик в області даних з окладом 20000. Визначимо середню зарплату за годину за формулою:

$$C_ч = \frac{M}{T_m \cdot t} \text{ грн.,} \qquad (4.14)$$

де M – місячний оклад працівників;

T_m – кількість робочих днів тиждень;

t – кількість робочих годин в день.

$$C_ч = \frac{22300 + 22300 + 20000}{3 \cdot 21 \cdot 8} = 128{,}17 \text{ грн.} \qquad (4.15)$$

Тоді, розрахуємо заробітну плату за формулою:

$$С_{ЗП} = С_{ч} \cdot Т_{i} \cdot К_{Д}, \qquad (4.16)$$

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

$Т_{i}$ – трудомісткість відповідного завдання;

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

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 128,17 \cdot 1371,84 \cdot 1.2 = 210994,48$ грн.

II. $С_{ЗП} = 128,17 \cdot 1391,04 \cdot 1.2 = 213947,52$ грн.

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_Г \cdot (1 + К_З) = 53520 \cdot (1 + 0.2) = 64224 \text{ грн.}$$

Відрахування на соціальний внесок:

$$С_{ВІД} = С_{ЗП} \cdot 0.22 = 64224 \cdot 0,22 = 14129,28 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 50000 грн.

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.4 \cdot 0.25 \cdot 50000 = 17500 \text{ грн.},$$

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

$К_А$– річна норма амортизації;

$Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.4 \cdot 50000 \cdot 0.08 = 5600 \text{ грн.},$$

$$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 22300 \cdot 0,2 = 53520 \text{ грн.}$$

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 12 – 16) \cdot 8 \cdot 0.35 =$$

$$= 627,2 \text{ години},$$

де $Д_К$ – календарна кількість днів у році;

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

$Д_Р$ – кількість днів планових ремонтів устаткування;

t –кількість робочих годин в день;

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

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 627,2 \cdot 0,2 \cdot 0,3 \cdot 4,87 = 183,27 \text{ грн.},$$

де $N_С$ – середньо-споживча потужність приладу;

$К_З$– коефіцієнтом зайнятості приладу;

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

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} \cdot 0.67 = 50000 \cdot 0,67 = 33500 \quad \text{грн.}$$

Тоді, річні експлуатаційні витрати будуть:

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

$$С_{ЕКС} = 64224 + 14129,28 + 17500 + 5600 + 183,27 + 33500 = 135136,55 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{М\text{-}Г} = С_{ЕКС} / Т_{ЕФ} = 135136,55 / 627,2 = 215,46 \quad \text{грн/год.}$$

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

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

I. $C_М = 215{,}46 \cdot 1371{,}84 = 295576{,}65$ грн.

II. $C_М = 215{,}46 \cdot 1391{,}04 = 299713{,}48$ грн.

Накладні витрати складають 67% від заробітної плати:

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

I. $C_Н = 295576{,}65 \cdot 0{,}67 = 198036{,}36$ грн.

II. $C_Н = 299713{,}48 \cdot 0{,}67 = 200808{,}03$ грн.

Отже, вартість розробки ПП за варіантами становить:

$$C_{пп} = C_{зп} + C_{від} + C_{м} + C_{н}, \quad (4.20)$$

I. $C_{пп}$ = 210994,48 + 46418,79 + 295576,65 + 198036,36 = 751026,28 грн.

II. $C_{пп}$ = 213947,52 + 47068,45 + 299713,48 + 200808,03 = 761537,48 грн.

4.7 Вибір кращого варіанту ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

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

$$К_{ТЕР1} = 3,28 / 751026,28 = 4,367 \cdot 10^{-6},$$

$$К_{ТЕР2} = 2,68 / 761537,48 = 3,519 \cdot 10^{-6}.$$

Як бачимо, найбільш ефективним є перший варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1} = 4,367 \cdot 10^{-6}$.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є перший варіант реалізації програмного продукту. У нього виявився найкращий показник техніко-економічного рівня якості.

$К_{ТЕР} = 4,367 \cdot 10^{-6}$.

Цей варіант реалізації програмного продукту має такі параметри:

- Вибір мови програмування - Python;
- Використання готових бібліотек;
- Середовище розробки – Jupyter Notebook.

Даний варіант виконання програмного комплексу дає користувачу зручний інтерфейс, швидку реалізацію програми та доступний функціонал для роботи.

4.8 Висновки до четвертого розділу

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

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

На основі аналізу вибрано варіант реалізації програмного продукту.

_

ВИСНОВКИ

У цій роботі була досліджена актуальність обраної теми та використання систем і методів штучного інтелекту для гри в шахи. Була опрацьована науково-технічна література з даної теми, розглянута інформація про сучасний алгоритм гри в шахи. Було розглянуто основні типи методів, виявлені переваги та недоліки кожного з них. У процесі виконання роботи було запропоновано та реалізовано декілька методів, проведений аналіз метрик для кожної з них. Завдяки останнім було створено програмний продукт, який дає непогані результати. Після аналізу був обраний метод, який дає найкращі результати за всіма показниками.

СПИСОК ЛІТЕРАТУРИ

Шахи:

1. **"The Complete Idiot's Guide to Chess" by Patrick Wolff**: Ця книга є чудовим посібником для новачків у світі шахів, яка розкриває основи гри та стратегії.

2. **"Bobby Fischer Teaches Chess" by Bobby Fischer**: Ця книга пропонує унікальний підхід до вивчення шахів від одного з найбільш відомих шахових геніїв усіх часів.

3. **"My System" by Aron Nimzowitsch**: Класична книга, яка представляє теорію шахів, що базується на концепції центру, контролю простору та інших стратегічних принципах.

Мінімакс та Штучний інтелект:

1. **"Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig**: Це стандартний підручник з штучного інтелекту, який включає розділ про пошук у гральних іграх та алгоритм мінімакс.

2. **"Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville**: Ця книга досліджує сучасні методи глибокого навчання, які використовуються у штучному інтелекті, включаючи методи, що застосовуються до гральних ігор.

3. **"Games, Diversions & Perl Culture: Best of the Perl Journal" by Jon Orwant, Jarkko Hietaniemi, and John Macdonald**: Ця книга містить розділ про програмування гральних ігор та реалізацію алгоритму мінімакс за допомогою Perl.

Додаток А

```
var board,
    game = new Chess();

/*The "AI" part starts here */

var minimaxRoot =function(depth, game, isMaximisingPlayer) {

    var newGameMoves = game.ugly_moves();
    var bestMove = -9999;
    var bestMoveFound;

    for(var i = 0; i < newGameMoves.length; i++) {
        var newGameMove = newGameMoves[i]
        game.ugly_move(newGameMove);
        var value = minimax(depth - 1, game, -10000, 10000, !isMaximisingPlayer);
        game.undo();
        if(value >= bestMove) {
            bestMove = value;
            bestMoveFound = newGameMove;
        }
    }
    return bestMoveFound;
};

var minimax = function (depth, game, alpha, beta, isMaximisingPlayer) {
    positionCount++;
    if (depth === 0) {
        return -evaluateBoard(game.board());
```

```
    }

    var newGameMoves = game.ugly_moves();

    if (isMaximisingPlayer) {
        var bestMove = -9999;
        for (var i = 0; i < newGameMoves.length; i++) {
            game.ugly_move(newGameMoves[i]);
            bestMove = Math.max(bestMove, minimax(depth - 1, game, alpha, beta, !
isMaximisingPlayer));
            game.undo();
            alpha = Math.max(alpha, bestMove);
            if (beta <= alpha) {
                return bestMove;
            }
        }
        return bestMove;
    } else {
        var bestMove = 9999;
        for (var i = 0; i < newGameMoves.length; i++) {
            game.ugly_move(newGameMoves[i]);
            bestMove = Math.min(bestMove, minimax(depth - 1, game, alpha, beta, !isMaximisingPlayer));
            game.undo();
            beta = Math.min(beta, bestMove);
            if (beta <= alpha) {
                return bestMove;
            }
        }
        return bestMove;
    }
```

```
};

var evaluateBoard = function (board) {
    var totalEvaluation = 0;
    for (var i = 0; i < 8; i++) {
        for (var j = 0; j < 8; j++) {
            totalEvaluation = totalEvaluation + getPieceValue(board[i][j], i ,j);
        }
    }
    return totalEvaluation;
};

var reverseArray = function(array) {
    return array.slice().reverse();
};

var pawnEvalWhite =
    [
        [0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
        [5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
        [1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
        [0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
        [0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
        [0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
        [0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5],
        [0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
    ];

var pawnEvalBlack = reverseArray(pawnEvalWhite);
```

```
var knightEval =
    [
        [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
        [-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
        [-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
        [-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
        [-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
        [-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
        [-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
        [-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
    ];

var bishopEvalWhite = [
    [ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
    [ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
    [ -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
    [ -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
    [ -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
    [ -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
    [ -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
    [ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
];

var bishopEvalBlack = reverseArray(bishopEvalWhite);

var rookEvalWhite = [
    [  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
    [  0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
    [ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
    [ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
```

```
    [ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
    [ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
    [ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
    [  0.0,   0.0, 0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
];

var rookEvalBlack = reverseArray(rookEvalWhite);

var evalQueen = [
    [ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
    [ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
    [ -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
    [ -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
    [  0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
    [ -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
    [ -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
    [ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
];

var kingEvalWhite = [

    [ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
    [ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
    [ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
    [ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
    [ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
    [ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
    [  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ],
    [  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 ]
];
```

```
var kingEvalBlack = reverseArray(kingEvalWhite);



var getPieceValue = function (piece, x, y) {
    if (piece === null) {
        return 0;
    }
    var getAbsoluteValue = function (piece, isWhite, x ,y) {
        if (piece.type === 'p') {
            return 10 + ( isWhite ? pawnEvalWhite[y][x] : pawnEvalBlack[y][x] );
        } else if (piece.type === 'r') {
            return 50 + ( isWhite ? rookEvalWhite[y][x] : rookEvalBlack[y][x] );
        } else if (piece.type === 'n') {
            return 30 + knightEval[y][x];
        } else if (piece.type === 'b') {
            return 30 + ( isWhite ? bishopEvalWhite[y][x] : bishopEvalBlack[y][x] );
        } else if (piece.type === 'q') {
            return 90 + evalQueen[y][x];
        } else if (piece.type === 'k') {
            return 900 + ( isWhite ? kingEvalWhite[y][x] : kingEvalBlack[y][x] );
        }
        throw "Unknown piece type: " + piece.type;
    };

    var absoluteValue = getAbsoluteValue(piece, piece.color === 'w', x ,y);
    return piece.color === 'w' ? absoluteValue : -absoluteValue;
};
```

```
/* board visualization and games state handling */

var onDragStart = function (source, piece, position, orientation) {
    if (game.in_checkmate() === true || game.in_draw() === true ||
        piece.search(/^b/) !== -1) {
        return false;
    }
};

var makeBestMove = function () {
    var bestMove = getBestMove(game);
    game.ugly_move(bestMove);
    board.position(game.fen());
    renderMoveHistory(game.history());
    if (game.game_over()) {
        alert('Game over');
    }
};



var positionCount;
var getBestMove = function (game) {
    if (game.game_over()) {
        alert('Game over');
    }

    positionCount = 0;
    var depth = parseInt($('#search-depth').find(':selected').text());
```

```
    var d = new Date().getTime();
    var bestMove = minimaxRoot(depth, game, true);
    var d2 = new Date().getTime();
    var moveTime = (d2 - d);
    var positionsPerS = ( positionCount * 1000 / moveTime);

    $('#position-count').text(positionCount);
    $('#time').text(moveTime/1000 + 's');
    $('#positions-per-s').text(positionsPerS);
    return bestMove;
};

var renderMoveHistory = function (moves) {
    var historyElement = $('#move-history').empty();
    historyElement.empty();
    for (var i = 0; i < moves.length; i = i + 2) {
        historyElement.append('<span>' + moves[i] + ' ' + ( moves[i + 1] ? moves[i + 1] : ' ') +
'</span><br>')
    }
    historyElement.scrollTop(historyElement[0].scrollHeight);

};

var onDrop = function (source, target) {

    var move = game.move({
        from: source,
        to: target,
        promotion: 'q'
```

```
    });

    removeGreySquares();
    if (move === null) {
        return 'snapback';
    }

    renderMoveHistory(game.history());
    window.setTimeout(makeBestMove, 250);
};

var onSnapEnd = function () {
    board.position(game.fen());
};

var onMouseoverSquare = function(square, piece) {
    var moves = game.moves({
        square: square,
        verbose: true
    });

    if (moves.length === 0) return;

    greySquare(square);

    for (var i = 0; i < moves.length; i++) {
        greySquare(moves[i].to);
    }
};
```

```
var onMouseoutSquare = function(square, piece) {
    removeGreySquares();
};

var removeGreySquares = function() {
    $('#board .square-55d63').css('background', '');
};

var greySquare = function(square) {
    var squareEl = $('#board .square-' + square);

    var background = '#a9a9a9';
    if (squareEl.hasClass('black-3c85d') === true) {
        background = '#696969';
    }

    squareEl.css('background', background);
};

var cfg = {
    draggable: true,
    position: 'start',
    onDragStart: onDragStart,
    onDrop: onDrop,
    onMouseoutSquare: onMouseoutSquare,
    onMouseoverSquare: onMouseoverSquare,
    onSnapEnd: onSnapEnd
};
board = ChessBoard('board', cfg);
```