Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)
Навчально-науковий інститут комп'ютерних наук та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))
Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до дипломної роботи
перший (бакалаврський)
(рівень вищої освіти)

на тему: **«Розроблення алгоритму для гри в шашки засобами C#»**

Виконав: студент **4** курсу, групи **КН - 41**
Спеціальності 122–"Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)
Мищишин Д. В.
(прізвище та ініціали)
Керівник: **Капран І. Д., Думанський О. І**
(прізвище та ініціали)
Рецензент: _____
(прізвище та ініціали)

Львів – 2025 р.

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Навчально-науковий інститут *комп'ютерних наук та інформаційних технологій*
Кафедра *комп'ютерних наук*
Рівень вищої освіти *перший (бакалаврський)*
Спеціальність *122 – "Комп'ютерні науки"*
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
_____ Борецька І. Б.
"*10*" *червня* 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТУ

Мищишину Денису Володимировичу
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *«Розроблення алгоритму для гри в шашки засобами C#»* .

керівник проекту (роботи) *Капран Ігор Дмитрович, старший викладач кафедри комп'ютерних наук, Думанський Остап Іванович, доцент кафедри комп'ютерних наук, кандидат фізико-математичних наук* .
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від "*15*" *листопада* *2024*р.№ *С-882*

2. Термін подання студентом проекту (роботи) *10 червня 2024 року*

3. Вихідні дані до проекту (роботи) *Огляд та аналіз шляхів вирішення поставлених завдань, характеристика процесу гри в шашки з використанням поширених алгоритмів, аналіз технологій для програмної реалізації, організаційна структура для розробленого алгоритму гри в шашки на основі мови програмування C#. Аналіз основних інформаційних та програмних засобів необхідних для реалізації поставленого технічного завдання*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) *Вступ., Розділ 1. Стан проблемної області., Розділ 2. Інформаційне та математичне забезпечення., Розділ 3. Програмне та технічне забезпечення., Висновки., Список використаних джерел., Додатки.*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
слайди для доповіді (підготовка матеріалу загальним обсягом 10-15 слайдів)

6. Дата видачі завдання *18 листопада 2024 року*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1.	Системний аналіз стану проблемної області. Огляд літературних джерел згідно досліджуваної теми. Збір потрібних матеріалів. Формування функціональних вимог та постановка задачі проєкту.	*21. 02. 2025 р.* *28. 02. 2025 р.*	*Виконано*
2.	Огляд сучасного стану проблемної області. Оформлення першого розділу пояснювальної записки.	*01. 03. 2025 р.* *12. 03. 2025 р.*	*Виконано*
3.	Написання другого розділу. Аналіз математичного забезпечення.	*16. 03. 2025 р.* *29. 03. 2025 р.*	*Виконано*
4.	Оформлення третього розділу пояснювальної записки. Програмна реалізація	*02. 04. 2025 р.* *12. 04. 2025 р.*	*Виконано*
5.	Оформлення висновків пояснювальної записки. Формування апаратного забезпечення.	*13. 05. 2025 р.* *19. 05. 2025 р.*	*Виконано*
6.	Оформлення пояснювальної записки та здача на рецензування.	*23.05.2025 р.* *10.06.2025 р.*	*Виконано*

Студент _____ (підпис) *Мищишин Д. В.* (прізвище та ініціали)

Керівник проекту (роботи) _____ (підпис) *Капран І. Д.* (прізвище та ініціали)

Керівник проекту (роботи) _____ (підпис) *Думанський О. І.* (прізвище та ініціали)

ТЕХНІЧНЕ ЗАВДАННЯ

Необхідно дослідити та розробити власний алгоритм для гри в шашки на основі набору правил для прийняття рішень в іграх з нульовою сумою, який ґрунтується на тому, щоб для всіх можливих ходів передбачити відповіді суперника, і для кожної відповіді — вибрати свою відповідь, на яку знову передбачити можливі відповіді суперника.

Завдання включає такі кроки:

1. Провести огляд літературних джерел щодо алгоритмів для гри в шашки;
2. Охарактеризувати процес розроблення програми для гри в шашки з використанням власного алгоритму;
3. Провести аналіз технологій, які знадобляться для програмної реалізації;
4. Створити графічний інтерфейс для розробленого алгоритму;
5. Провести тестування розробленого алгоритму для гри в шашки з метою перевірки працездатності.

АНОТАЦІЯ

Бакалаврська дипломна робота (проект): пояснювальна записка: 65 стор., 15 рис., 2 додатки, 16 джерело.

В даній дипломній роботі було проведено огляд та аналіз алгоритмів для гри в шашки, їхні особливості, переваги та недоліки. Визначені та проаналізовані основні інформаційні та програмні засоби необхідні для реалізації поставленого технічного завдання. Досліджені параметри моделі, які впливають на процес гри в шашки.

Результатом проведеної роботи є розроблений власний унікальний алгоритм, який заснований на наборі правил для прийняття рішень в іграх з нульовою сумою. Алгоритм представлений у форматі інтерактивної форми на якій можна змінювати параметри та характеристики гри. Програмне забезпечення реалізовано за допомогою таких технологій: Unity, C#.

Ключові слова: Алгоритм, C#, Unity, шашки, машинне навчання, штучний інтелект.

ABSTRACT

Bachelor's thesis (project): explanatory note: 65 pages, 15 figures, 2 appendices, 16 sources.

This thesis reviewed and analyzed algorithms for playing checkers, their features, advantages and disadvantages. The main information and software tools necessary for the implementation of the technical task were identified and analyzed. The parameters of the model that affect the process of playing checkers were studied.

The result of the work is the development of our own unique algorithm, which is based on a set of rules for making decisions in zero-sum games. The algorithm is presented in the format of an interactive form on which you can change the parameters and characteristics of the game. The software was implemented using the following technologies: Unity, C#.

Keywords: Algorithm, C#, Unity, checkers, machine learning, artificial intelligence.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

Алгоритм - набір інструкцій, які описують порядок дій виконавця, щоб досягти результату розв'язання задачі за скінченну кількість дій.

C# - об'єктно-орієнтована мова програмування з безпечною системою типізації для платформи .NET.

Unity - багатоплатформовий інструмент для розроблення відеоігор і застосунків, і рушій, на якому вони працюють.

Штучний інтелект (ШІ) - розділ комп'ютерної лінгвістики й інформатики, який швидко розвивається і зосереджений на розробці інтелектуальних машин, здатних виконувати завдання, які зазвичай потребують людського інтелекту.

ВСТУП

На просторах Інтернету можна легко знайти сотні, а в його англомовному сегменті — тисячі, статей про розроблення алгоритмів для гри в шахи. Однак гра в шашки не викликає такого інтересу. В Інтернеті мені не вдалося знайти майже жодної цікавої статті про послідовну розробку алгоритму для гри в шашки, тому виникла потреба створити даний алгоритм власноруч.

Об'єктом дослідження є унікальний алгоритм для гри в шашки.

Предметом дослідження – методи та алгоритми, які використовуються під час розроблення логічних ігор на основі штучного інтелекту.

Метою роботи є розробити власний алгоритм для гри в шашки на основі набору правил для прийняття рішень в іграх з нульовою сумою.

Актуальність роботи полягає в тому, що гра в шашки залишається актуальною та популярною логічною грою для багатьох людей. Їхній розвиток та впровадження у формі програмних застосунків дає можливість збагатити і покращити індивідуальний досвід користувачів. Крім того даний проєкт є цікавим тим, що для розроблення власного алгоритму гри в шашки був використаний набір правил для прийняття рішень в іграх з нульовою сумою, який ґрунтується на тому, щоб для всіх можливих ходів передбачити відповіді суперника, і для кожної відповіді — вибрати свою відповідь, на яку знову передбачити можливі відповіді суперника.

Практична значимість роботи полягає в тому, що розроблений алгоритм для гри в шашки дає можливість користувачеві визначати оптимальні ходи гравця в режимі реального часу.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1. Логічна настільна гра шашки

Коли людина грає в комп'ютерні ігри, пояснюючи це саморозвитком, її поблажливо називають геймером. Коли в ігри грає сам комп'ютер з тією ж метою, це називають машинним навчанням і штучним інтелектом. Логічні настільні ігри — справді один із найкращих способів удосконалення систем, що самонавчаються (рис. 1. 1).

Рисунок 1. 1 – Самонавчання системи на основі гри шашки

Основоположником технології машинного навчання на основі ігрових механік, був американський вчений та спеціаліст у галузі інформатики Артур Самуель. Перейшовши працювати у корпорацію IBM він зайнявся створенням програм для комп'ютера IBM 701, і саме на цьому комп'ютері він написав свою знамениту програму для гри в шашки, що самонавчається, після публічної демонстрації якої акції компанії IBM відразу злетіли на 15 пунктів. Роботу над цим проектом Артур Самуель розпочав ще 1949 року, коли працював в Університеті Іллінойсу. Він

вважав, що у майбутньому комп'ютерні програми повинні не тільки суворо виконувати закладені у них інструкції, а й самонавчатись, коригуючи свою поведінку залежно від змінних обставин з урахуванням набору базових алгоритмів. А найефективніший спосіб натренувати таку програму – це ігрові механіки, вважав Самуель [1].

В якості гри, на основі якої він вирішив розробити програму, що навчається, Артур Самуель взяв шашки. По-перше, йому самому дуже подобалася ця гра, а по-друге, її правила дуже прості, а кількість можливих стратегій — навпаки, дуже велика. Більшість розроблених на той час ігрових алгоритмів послідовно перебирали всі можливі варіанти наступного ходу, а потім вибирали серед них оптимальний. Самуель не міг будувати свій проект на базі такої моделі поведінки, оскільки IBM 701 використовував пам'ять на електронно-променевих трубках Вільямса ємністю 512 слів по 36 біт, і такого невеликого обсягу було для цього недостатньо. Завдяки старанням Самуеля пам'ять машини була в результаті збільшена до 2048 слів, середній час відмови зріс до півгодини, а в систему команд IBM 701 було додано низку розроблених ним директив для нечислових обчислень (рис. 1. 2).

Рисунок 1. 2 – Розроблення самонавчальної програми на основі гри шашки

Тому Артур Самуель застосував інший підхід: для розрахунку ходів у грі він використав механізм пошуку рішень, відомий як «альфа-бета-відсікання». Щоб зрозуміти, що це таке, спочатку потрібно розглянути алгоритм мінімаксу та концепцію ігрових дерев. Алгоритм мінімаксу зазвичай використовується в іграх з нульовою сумою для двох гравців, де виграш одного означає програш іншого. Ігрові дерева описують всі можливі ходи та їх результати у грі, причому кожен вузол є певним станом гри та пов'язаним з ним значенням [2].

Створена Артуром Самуелем програма для гри в шашки використовувала оціночні функції, кожна з яких мала власну «вагу». «Вага» функції змінювалася в процесі гри, якщо один із гравців прораховував партію на більшу кількість ходів уперед і грав сильніше. Більше того, в програму було закладено довідник із описом партій, зіграних професійними спортсменами — за основу Самуель взяв популярну книгу Джеймса Ліса «Посібник по шашкам» (рис. 1. 3). Довідник використовувався у випадку, коли на дошці складалася схожа ситуація: якщо в довіднику вже був відповідний запис, «вага» функції змінювався таким чином, що комп'ютер повторював хід, зроблений раніше в аналогічній диспозиції людиною.

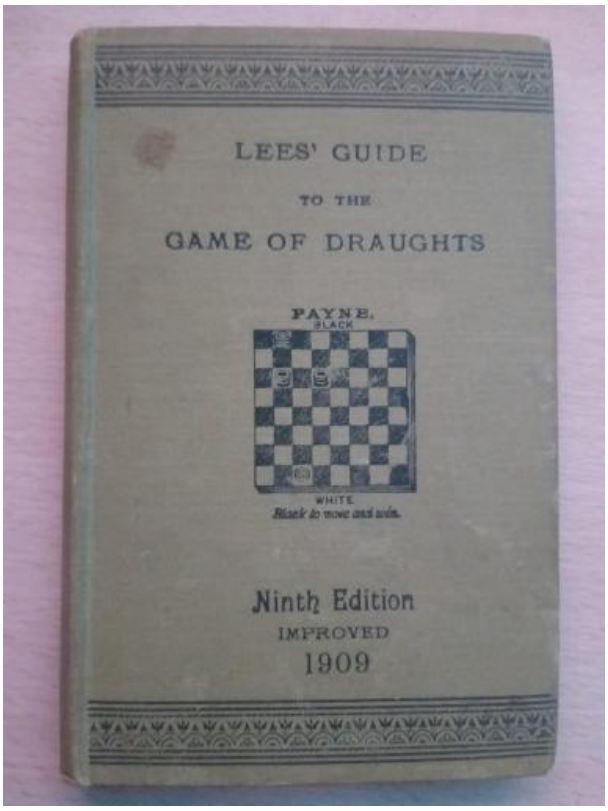

Рисунок 1. 3 - Книга Джеймса Ліса «Посібник по шашкам»

Машина «запам'ятовувала» всі зіграні партії (цей алгоритм Самуель назвав «rote learning»), враховувала невдалі ходи, та застосовувала отримані знання при підрахунку наступного ходу, збільшуючи цим глибину пошуку оптимального рішення. Таким чином, програма вийшла самонавчальною: вона ітеративно покращувала свою продуктивність, граючи сама проти себе та навчаючись на результатах. Отримані в ході гри дані накопичувалися, а потім використовувалися для визначення «вагової» функції наступних ходів та адаптації ігрової стратегії на основі результатів, що спостерігаються.

Програма Самуеля заклала основу для використання алгоритмів машинного навчання не тільки в ігровій індустрії, але також в інших галузях і відкрила дорогу майбутнім досягненням у галузі штучного інтелекту. Розроблені Артуром Самуелем концепції та методи, такі як самонавчання шляхом гри «сам із собою», ітеративне покращення пошуку рішень та впровадження функції оцінки, вплинули на розробку більш складних алгоритмів, включаючи навчання з підкріпленням. Подібні принципи використовуються сьогодні при створенні технологій машинного навчання в таких напрямках як розпізнавання образів, обробка природної мови, робототехніка та автономні системи.

Власне, обмежені апаратні можливості так чи інакше гальмували розвиток машинного навчання та інтелектуальних комп'ютерних систем, доки не стали широко доступними хмарні технології. Саме їхнє поширення і стало свого роду каталізатором, який подарував розвитку штучного інтелекту друге дихання.

Сучасні системи, що працюють на основі алгоритмів машинного навчання, мають на увазі обробку великих масивів даних та запуск складних моделей, що, у свою чергу, потребує значних обчислювальних потужностей. Хмарні платформи пропонують еластичні ресурси, які можна легко збільшувати або зменшувати в залежності від поточних потреб, а це дає можливість дослідникам та розробникам обійтися без попередніх інвестицій в апаратну інфраструктуру. Тим більше, при використанні такої схеми оплачуються тільки ресурси, що фактично споживаються, що, знову ж таки, означає економію.

Використовуючи хмарні платформи, розробники можуть розгорнути свої моделі у вигляді сервісів або API, що робить їх доступними для сторонніх додатків та помітно полегшує інтеграцію можливостей ІІ у різні системи та сервіси, включаючи мобільні програми, веб платформи та пристрої IoT Крім того, у хмарі набагато простіше реалізувати централізоване та безпечне сховище величезних обсягів даних, необхідних для роботи системи, що самонавчається.

Експоненційне зростання кількості побудованих на базі штучного інтелекту проєктів у другому десятилітті XXI століття багато в чому зумовлене помітним зниженням вхідного бар'єру у цій сфері. Завдяки доступності обчислювальних потужностей, попередньо налаштованих середовищ та заздалегідь навчених моделей на сучасних хмарних платформах, дослідникам та розробникам набагато простіше зосередитися на впровадженні інновацій, а не на налаштуванні та управлінні інфраструктурою.

1.2. Обчислювальні алгоритми та програми для гри в шашки

Скільки часу існують комп'ютери, стільки і розвиваються алгоритми, здатні змагатися з людиною в логічних іграх. Для кожної з ігор шлях розвитку алгоритму був своїм. Але здебільшого все починалося з набору евристик, які було порівняно легко зрозуміти. Намагаючись ускладнити рішення, застосовували метод Монте-Карло для пошуку по дереву та різні гібридні підходи – це дозволило покращити якість ботів і навіть грати нарівні з людиною. А зростання доступних апаратних потужностей у деяких іграх остаточно передав перемогу машині. Наприклад, шашки було вирішено повністю у 2007 році. На тлі інших логічних ігор шашки прості в прорахунку, кількість комбінацій у них — близько 10-20 ступеня.

Розроблення програм для гри в шашки розпочалося із середини 50-х років минулого століття. Найбільш відома серед них – Chinook, створена Джонатаном Шеффером [3]. Chinook була представлена на комп'ютерній олімпіаді у 1990 році. А у 1992 році програма зіграла у міжнародному чемпіонаті з шашок проти багаторазового чемпіона світу Маріона Тінслі та виграла у нього два матчі. На той момент Chinook передбачала гру на 14-17 кроків уперед і могла повністю прорахувати всі можливі позиції, якщо на дошці залишалося вісім шашок. З розвитком техніки глибина прорахунку збільшувалася, як і база даних закінчень ігор. На початку 2000-х років програма знала всі можливі закінчення матчу, якщо на дошці залишалося 10 шашок. А 2007 року з'явилася публікація, в якій повідомлялося, що шашки вирішено повністю. На це завдання у Шеффера пішло 19 років [4].

Потім обчислювальні алгоритми продовжили підкорювати інші ігри з великою кількістю комбінацій. Рішення на основі старих підходів швидко уперлися в «комбінаторний вибух» та обчислювальну складність. Тому їм на зміну прийшли архітектури, що базуються на нейронних мережах. Але вони теж виявилися недосконалими. У тому ж таки повністю детерміноване рішення — це NP-повне завдання, яке не може бути вирішене чисельно в рамках існуючих архітектур комп'ютерних систем та алгоритмів.

Людина підходить до гри принципово інакше, ніж обчислювальний алгоритм. Вона використовує інтуїцію, припущення про противника та його минулі ігри, невербальні комунікації з ним. У результаті не перебираючи всі варіанти, а працюючи з образами, вона знаходить виграшну стратегію.

«Цифровий кентавр» — поєднання людини та алгоритмів — структура, яка бере краще від обох «світів». Комп'ютер прораховує найближчі можливі кроки, але людина приймає рішення яким саме шляхом піти. Фактично «цифровий кентавр» — це демонстрація підтримки прийняття рішень, яка може стати в нагоді поза логічними іграми: в енергетиці, в медицині, в управлінні транспортом, в юриспруденції, в адмініструванні складних систем: міст і держав тощо.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Алгоритм мінімакс

Мінімакс – популярний алгоритм для прийняття рішень в іграх з нульовою сумою (один виграв – інший програв). Цей алгоритм часто використовується для ігор з нульовою сумою, інакше кажучи: коли вигода одного гравця призводить до такої ж невигоди іншого. До таких ігор можна віднести: хрестики-нолики, шахи, шашки, та багато інших класичних ігор.

Мінімакс це алгоритм прийняття рішень, який ґрунтується на тому, що один учасник буде обирати найкращий хід для себе, тобто мінімізувати шанс перемоги іншого учасника. Мінімакс можна представити як рекурсивний метод, який здійснює ходи від імені обох гравців, поки не досягне певної глибини (кількості ходів 'вперед'), після чого обчислює наскільки хороша поточна позиція (вага позиції). Таке відбувається для всіх можливих ходів на обраній глибині. Алгоритм обчислює вагу тільки на кінцях гілок ходів (пелюстках). Далі виходячи з того, що гравець намагатиметься виграти, тобто звести вагу ходу штучного інтелекту до мінімального значення, можна буде знайти вагу у всій гілці ходів. Тому кожен виклик методу мінімаксу можна розділити на максимізацію (ШІ) та мінімізацію (гравець), це буде простіше уявити у вигляді дерева вибору послідовності ходів. Відповідно якщо вказана черга ходу для гравця (мінімізує), то вузол отримає мінімальну вагу пелюсток і навпаки, у разі ШІ (максимізуючого) він отримає максимальну вагу. Наприклад, якщо вага пелюсток буде 3 і 5, а поточний вузол під час ходу ШІ (максимізуючого), то вузол отримає вагу 5, піднімаючись вище по гілці ходів будуть порівнюватися вже два нижчестоящих вузла вагою 5 і 9, так як хід гравця (мінімізує), то вузол отримає мінімальну вагу - 5. За підсумком результат виконання алгоритму можна уявити як звичайне дерево, яке відображає оцінки для всіх можливих гілок ходів і дозволяє вибрати оптимальний хід (рис. 2. 1).

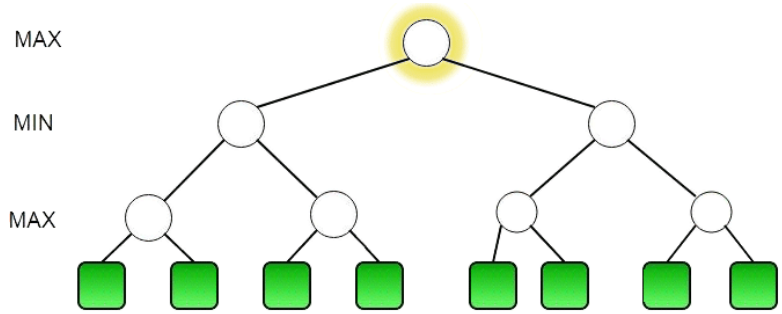

Рисунок 2. 1 - Робота алгоритму мінімакс

Алгоритм мінімакса сам по собі досить повільний і ресурсозатратний. Адже можливих комбінацій ходів, здавалося б, на такій маленькій дошці, мільйони і це при тому, що всього можливих станів фігур на дошці лише 4368. Розрахувати максимально можливу кількість комбінацій ходів можна за формулою:

$$(4*2*4)^5=33554432 \quad (2.\ 1)$$

Проблеми використання Мінімаксу:

- *Швидкість.* Навіть з урахуванням використання Альфа-бета відсікання швидкість роботи алгоритму не тішить. Крім Альфа-Бета відсікання можна використовувати інші алгоритми оптимізації. Можна зберігати позиції отримані в ході виконання ітеративного поглиблення і використовувати їх для виклику методу RunMinMax, з більш високим значенням aiLevel, замість нових прорахунків з нуля. Також можна використовувати результати попереднього ходу, якщо вдалося передбачити, який хід зробив гравець.
- *Складність налагодження*. Мінімакс є рекурсивним алгоритмом і лише це сильно збільшує складність налагодження. Так само якщо десь у реалізації алгоритму помилка, дізнатися про це до її фактичного виявлення в коді, практично неможливо: алгоритм може продовжити видавати ходи, які навіть можуть бути логічно хорошими, а яким чином перевірити чи правильно виконаний хід є найкращим - не вийде.
- *Наростаюча помилка.* Логіка Мінімаксу в тому, що ШІ робить найкращий хід для себе і передбачається, що гравець буде робити найгірший хід для ШІ, що в корені неправильно. Зрозуміла справа людини помилятися, а разом з тим падає точність алгоритму зі збільшенням глибини.

2.2. Алгоритм оптимізації Альфа-Бета відсікання

Альфа-Бета відсікання - алгоритм який дозволяє без впливу на якість обчислень істотно скоротити кількість прорахунків. Для порівняння можна подивитися скільки разів відбувається обчислення ваги ходу (виклик методу GetMoveWeight) без використання Альфа-Бета відсікання і з ним. Без відсікання метод викликається до 340000 разів, використовуючи Альфа-Бета відсікання до 6500 разів (рис. 2. 2).

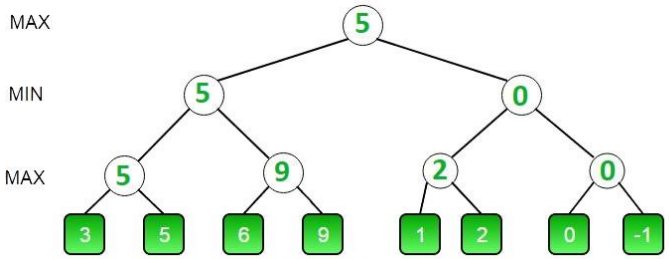

Рисунок 2. 2 - Альфа-Бета відсікання

Виходячи з того, що кожна глибина ділиться на максимізацію (ШІ) та мінімізацію (гравець), можна використовувати цей принцип, щоб скоротити кількість обчислень. Наприклад візьмемо вузол з вагою 5 на глибині мінімізації. Потрібно обчислити його вагу, для цього необхідно обчислити вагу нижчих вузлів, алгоритм обчислює вагу пелюсток першого вузла та встановлює вагу вузла 5, тепер можна обчислити вагу пелюстки іншого нижчестоящого вузла, який складе 6. Так як вузол, якому належить ця пелюстка, знаходиться на глибині максимізації, ми точно знаємо що буде обрана пелюстка з найбільшою вагою, тому можна відразу припустити, що вага цього вузла буде 6 або більше. Так як вищий вузол на глибині мінімізації, то в будь-якому випадку буде обраний вузол з найменшою вагою, а так як вага нижчестоящого вузла вже як мінімум дорівнює 6, то розраховувати останню його пелюстку не має сенсу, так як інший нижчестоящий вузол має вагу 5 тому буде обраний він. Такий процес повторюється у всіх вузлах і дозволяє відкинути гілки, що не вплинуть на результат. Для цього виділяються два поняття: Альфа – максимальна вага вузла, який максимізуючий гравець (ШІ) може досягти; і Бета - мінімальна вага вузла, який може досягти мінімізуючий гравець (гравець). Значення Альфи оновлюється в момент перебігу максимізуючого (ШІ), бети - мінімізує

(гравець). Спочатку Альфа є значення мінімально можливої ваги ходу, зазвичай негативну нескінченність, Бета у свою чергу навпаки, представляє позитивну нескінченність. У випадку, який представлений на рис. 2. 3, у момент коли вже обчислена пелюстка з вагою 6, для вищого вузла Альфа = 6, так як значення змінної було оновлено після обчислення ваги пелюстки, Бета = 5, так як це значення було передано з вищого вузла, який знаходиться на рівні мінімізації і встановив це значення в момент коли було обчислено вагу першого його нижчестоящого вузла з вагою 5. Тепер умова 6>=5 стає вірною і наступні пелюстки даного вузла знайдено нічого очікувати.

Тут також варто виділити деякі моменти щодо використання Альфа-Бета відсікання: ефективність алгоритму багаторазово збільшується зі зростанням максимальної рекурсії; для ефективної роботи алгоритму найкращі ходи всіх фігур мають прораховуватися першими, у разі моєї реалізації Мінімаксу достатньо встановити найкращі ходи на перші позиції в масиві з ходами, у випадку з фігурою це будуть ходи до іншого краю дошки, тобто ті, де y = -1.

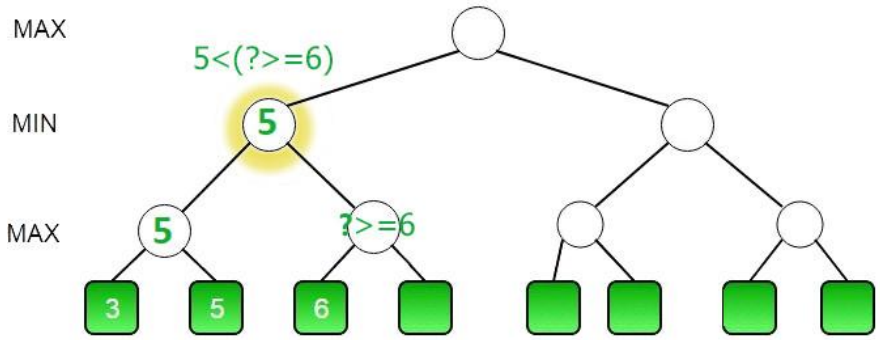

Рисунок 2. 3 – Реалізація алгоритму Альфа-бета відсікання

У грі в шашки використовуються різні математичні формули та логічні правила, які забезпечують коректну роботу алгоритму: перевірку допустимих ходів, захоплення (взяття) фігур, перетворення на дамку, визначення завершення гри тощо. Нижче наведено основні математичні аспекти, які використовуються у побудові алгоритму гри в шашки:

Модель ігрового поля

Ігрове поле — матриця розміром 8×8 (у класичних шашках):

$$B=\{b_{ij} \mid i, j \in \{1,2,\ldots,8\}, (i+j) \bmod 2 = 1\} \quad (2.2)$$

Оскільки ходи можливі лише по темних клітинках, використовується лише 32 клітини, тобто:

$$\text{PlayableCells}=\{(i, j) \in B \mid (i+j) \bmod 2 = 1\} \quad (2.3)$$

Математична формалізація фігури. Фігура описується як множина:

$$f=(x, y, t, p) \quad (2.4)$$

де:

- (x, y)-координати фігури на дошці,
- $t \in \{man, king\}$-тип фігури (пішак або дамка),
- $p \in \{white, black\}$-колір фігури.

Можливі ходи пішака. Пішак ходить на одну діагональну клітинку вперед (для білих — вгору, для чорних — вниз). Для білого:

$$(x', y') \in \{(x-1, y-1), (x+1, y-1)\} \quad (2.5)$$

Для чорного:

$$(x', y') \in \{(x-1, y+1), (x+1, y+1)\} \quad (2.6)$$

Умови взяття (удару). Фігура може зробити хід через ворожу фігуру, якщо за нею є вільна клітинка. Якщо f1 = (x, y), f2 = (x′, y′)-ворожа фігура, то можливе взяття, якщо

$$|x-x'| = 1 \wedge |y-y'| = 1 \wedge \text{Cell}(x+\Delta x, y+\Delta y) = \emptyset \quad (2.7)$$

де
$$\Delta x = x'-x, \Delta y = y'-y \quad (2.8)$$

Ходи дамки. Дамка може рухатися на довільну кількість клітин по діагоналі:

$$x' = x+k, y' = y+k \text{ або } x' = x+k, y' = y-k, k \in Z, |k|>0 \quad (2.9)$$

при умові, що всі клітини на шляху — порожні.

Проста оцінювальна функція для гри в шашки:

$$\text{Eval}(S) = w1 \cdot (P_{white}-P_{black})+w2 \cdot (K_{white}-K_{black}) \quad (2.10)$$

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Правила гри в шашки

Короткі правила гри в шашки наведені нижче:

1. Шашки ходять тільки по клітинам чорного кольору по діагоналі.
2. Проста шашка ходить тільки вперед на одне поле, а б'є вперед і назад, перестрибуючи одне поле (у checkers б'є тільки вперед).
3. Дамка ходить і б'є вперед і назад на будь-яку кількість полів (у checkers - ходить вперед і назад тільки на 1 поле; б'є, перестрибуючи тільки 1 поле).
4. Бити обов'язково! За наявності кількох варіантів бою можна вибрати будь-який.
5. Програє той, хто не може зробити хід.

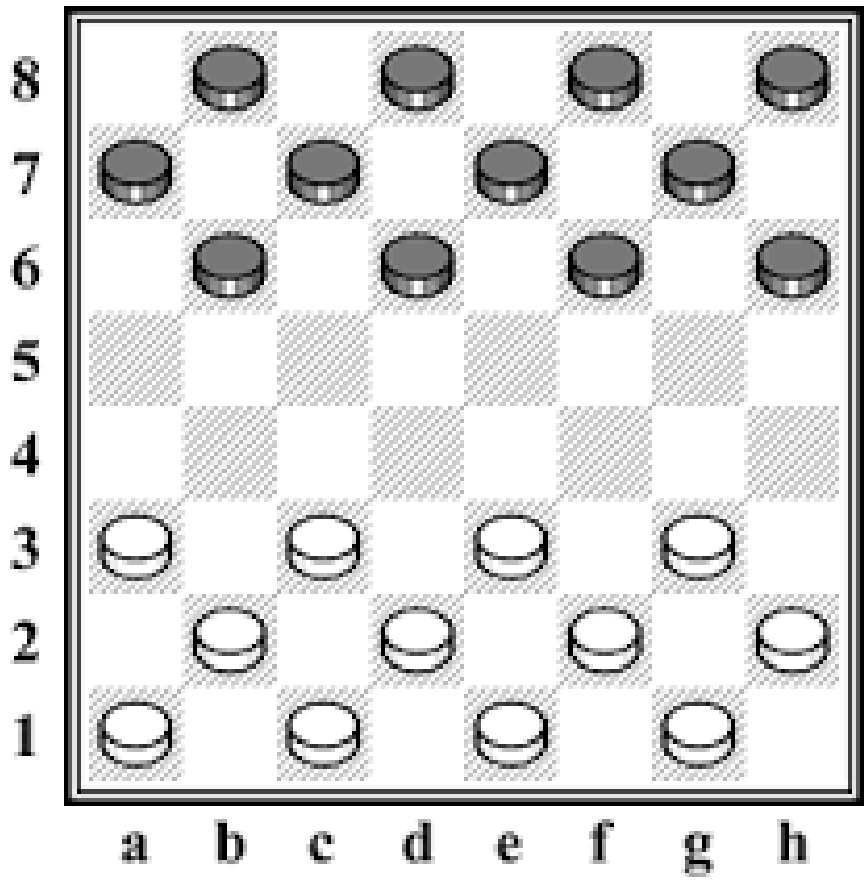

Рисунок 3.1 – Логічна гра шашки

3.2. Реалізація алгоритму

Зрозуміло, що класи, які відповідають за шашки на екрані й у пам'яті алгоритму, різні. Ми не будемо зупинятися на MonoBehaviour Unity-скриптах, а докладніше розберемо саме реалізацію алгоритму. Спочатку опишемо, як ми зберігаємо дошку для гри в шашки у пам'яті. Клас шашки досить простий: містить, заголовок, тип шашки та її позицію на полі, а також кілька допоміжних змінних:

```
public enum PieceType
{
    EMPTY, WHITE_MAN, WHITE_KING, BLACK_MAN, BLACK_KING
}

public class Piece
{
    public PieceType Type { get; private set; }
    public Vector2Int Pos { get; private set; }
    public bool IsWhite { get; private set; }
    public bool IsKing { get; private set; }


    public Piece(PieceType type, Vector2Int pos)
    {
        Type = type;
        Pos = pos;
        IsWhite = type == PieceType.WHITE_MAN || type == PieceType.WHITE_KING;
        IsKing = type == PieceType.WHITE_KING || type == PieceType.BLACK_KING;
    }

    public void ChangePos(Vector2Int newPos)
    {
        Pos = newPos;
    }
```

```
    public void BecomeKing()
    {
        Type = IsWhite ? PieceType.WHITE_KING : PieceType.BLACK_KING;
        IsKing = true;
    }
    public void BecomeMan()
    {
        Type = IsWhite ? PieceType.WHITE_MAN : PieceType.BLACK_MAN;
        IsKing = false;
    }
}
```

Дошка представляє собою набір шашок та їх ходів. Це не повний код дошки, а лише частина:

```
public class Board
    {
        private Piece[,] _board = new Piece[8, 8]; // фігури
        public List<Piece> Pieces { get; private set; } = new List<Piece>(); // ті ж фігури, тільки у вигляді списку
        private List<Move> _currentMoves; // список можливих ходів
        private int[] _countCheckers = new int[5]; // лічильник шашок визначених груп (всіх, білих звичайних, білих дамок, чорних звичайних, чорних дамок)
        private List<MemorisedMove> LastMoves = new List<MemorisedMove>();

      ...

        // Конструктор для встановлення дошки за рядком
        // searchAllMoves - чи потрібно шукати можливі ходи
        public Board (string arr, bool whitesMove = true, bool searchAllMoves = true)
        {
            int index = 0;
            // Прохід по всім клітинкам
            for (int y = 0; y < 8; y++)
            {
                for (int x = (y+1) % 2; x < 8; x += 2)
                {
                    if (arr[index] != '0')
                    {
                      // Індекс фігури
                        int num = int.Parse(arr[index].ToString());
                      // Отримаєм фігуру
                        Piece piece = new Piece((PieceType) num, new Vector2Int(x, y));

                      // Встановлюємо і заносимо у список
                        _board[y, x] = piece;
                        Pieces.Add(piece);
                        _countCheckers[num]++;
                    }

                    index++;
                }
```

```
            }
            WhitesMove = whitesMove;
            _rowKingsMoves = 0;
            _jumpIndex = 0;
            _countCheckers[0] = _countCheckers[1] + _countCheckers[2] + _countCheckers[3] +
_countCheckers[4];

            // Якщо потрібно, шукаємо можливі ходи
            if (searchAllMoves)
                FindAllMoves();
        }

      ...
    }
```

- Конструктор Board() тут будує дошку по рядку цифр, де кожна цифра позначає конкретну шашку (див. перелік PieceType у класі Piece).
- Також є конструктор, що створює глибоку копію дошки.

Розіб'ємо весь клас на частини, щоб не було забагато коду і можна було легко пояснити.

Наступні функції використовують для пошуку можливих ходів.

```
public void FindAllMoves ()
{
    List<Move> takingMoves = new List<Move>(); // взяття
    List<Move> simpleMoves = new List<Move>(); // прості ходи

    foreach (Piece piece in Pieces)
    {
        if (piece.IsWhite == WhitesMove)
        {
          // Для кожної фігури спочатку шукаємо всі взяття
            takingMoves.AddRange(GetAllTakingMovesOfPiece(piece));
          // Якщо взяття немає, шукаєм прості ходи
            if (takingMoves.Count == 0)
                simpleMoves.AddRange(GetAllSimpleMovesOfPiece(piece));
        }
    }

    // Якщо є взяття, відкидаємо прості ходи; інакше є тільки прості ходи
    if (takingMoves.Count > 0)
    {
        // Взяття сортуємо по спаданню кількості побитих шашок, щоб спочатку йшли найкращі
// це допоможе нам більш ефективно шукати сильні ходи, оцінюючи потенційно кращі першими
        takingMoves.Sort((Move a, Move b) => -a.Taken.Count.CompareTo(b.Taken.Count));

        AllMoves = _currentMoves = takingMoves;
    }
    else
        AllMoves = _currentMoves = simpleMoves;
}
```

```
// Рекурсивна функція пошуку взятої фігури
// У масиві exc зберігаються поля, шашки на які ми вже побили, так як в простих шашках,
// згідно турецькому правилу, шашки знімаються з дошки вже після ходу


private List<Move> GetAllTakingMovesOfPiece (Piece piece, List<Vector2Int> exc = null)
{
    if (exc == null)
        exc = new List<Vector2Int>();
    List<Move> moves = new List<Move>(); // все взяття
    List<Move> movesWithFollowingTake = new List<Move>(); // взяття, після яких можна побити
ще

    if (piece.IsKing)
    {
      // Перебираємо всі 4 направлення руху
        for (int x = 1; x > -2; x -= 2)
        {
            for (int y = 1; y > -2; y -= 2)
            {
                bool opp_found = false;
                Vector2Int pos_opp = Vector2Int.zero;

              // Куди дамка встане після прижку
                Vector2Int target = piece.Pos + new Vector2Int(x, y);
                while (InField(target)) // Функція InField перевіряє, що координати (x, y)
належать полю
                {
                    if (IsEmpty(target)) // Функція IsEmpty перевіряє, що поле не зайнято
                    {
                        if (opp_found) // Якщо, пригнув на клітинку target ми перепригнемо
шашку суперника, то це взяття
                            AddMove(piece.Pos, target, pos_opp); // Побічно рекурсивно
продовжуємо пошук подальших стрибків із взяттям
                    }
                    else if (_board[target.y, target.x].IsWhite == piece.IsWhite) // Якщо
вперлися у свою шашку, то все                         break;
                    else
                    {
                        if (!opp_found) // Якщо вперлися в шашку суперника, запам'ятовуємо
це
                        {
                            opp_found = true;
                            pos_opp = target;
                        }
                        else // Якщо уткнулися у 2-у шашку суперника, то далі стрибнути не
вдасться                             break;
                    }
                    target += new Vector2Int(x, y);
                }
            }
        }
    }
    else
    {
      // Тут перебираємо всі 4 варіанти взяття звичайної шашки (для стислості показано лише
одне)
```

```
 // target - поле куди приземлимся, middle - поле, яке перепригнемо. В даному випадку
пригаємо на збільшення обох координат (вниз вправо)
        Vector2Int target = new Vector2Int(piece.Pos.x + 2, piece.Pos.y + 2);
        Vector2Int middle = new Vector2Int(piece.Pos.x + 1, piece.Pos.y + 1);
        if (InField(target) && IsEmpty(target) && !IsEmpty(middle) && _board[middle.y,
middle.x].IsWhite != piece.IsWhite)
            AddMove(piece.Pos, target, middle);
        ...
        ...
        ...
    }
    if (movesWithFollowingTake.Count > 0)
        return movesWithFollowingTake;
    return moves;

    bool AddMove (Vector2Int fr, Vector2Int to, Vector2Int taken)
    {
      // Турецький удар
        if (exc.Contains(taken))
            return false;

      // Моделюємо дошку, на яку цей хід зроблено
        Board nextBoard = new Board(this, deepCopyMoves:false);
        Piece thisPiece = nextBoard.MovePiece(fr, to);
        List<Vector2Int> newExc = new List<Vector2Int>(exc);
        newExc.Add(taken);

      // Перевіряємо, чи не перетворилася наша шашка на дамку цим ходом
        bool isThisMoveKinging = !piece.IsKing && IsKinging(to, piece.IsWhite);
        List<Move> nextTakes = nextBoard.GetAllTakingMovesOfPiece(thisPiece, newExc);

        if (nextTakes.Count == 0)
        {
            moves.Add(new Move(new List<Vector2Int>() { fr, to }, new List<Vector2Int>() {
taken }, isThisMoveKinging));
            return false;
        }
        else
        {
            foreach (Move nextTake in nextTakes)
            {
                List<Vector2Int> pos = nextTake.Pos;
                pos.Insert(0, fr);
                List<Vector2Int> takes = nextTake.Taken;
                takes.Add(taken);
                moves.Add(new Move(pos, takes, isThisMoveKinging || nextTake.IsKinging));
                movesWithFollowingTake.Add(new Move(pos, takes, isThisMoveKinging ||
nextTake.IsKinging));
            }
            return true;
        }
    }
}
// Ця функція шукає всі звичайні ходи шашки. Вона дуже проста і не представляє особливого
інтересу
private List<Move> GetAllSimpleMovesOfPiece (Piece piece)
{
    ...
}
```

Тут варто звернути увагу на те, що всі звичайні ходи вважаються рівноцінними, а взяття — ні: найсильніші взяття це ті, що б'ють більше за шашки суперника.

У функції GetAllTakingMoves, яка шукає всі ходи-взяття, важливу роль відіграє так зване турецьке правило, згідно з яким побиті шашки знімаються з дошки після ходу та можуть заважати продовжити взяття. Наприклад, у позиції нижче, якщо білі візьмуть дамкою e1:a5:d8:f6:d4, вони не зможуть взяти ще й шашку c5, оскільки, хоча шашка b6 на той час вже буде побита, вона все ще стоятиме на дошці, заважаючи дамці білих (рис. 3. 2).

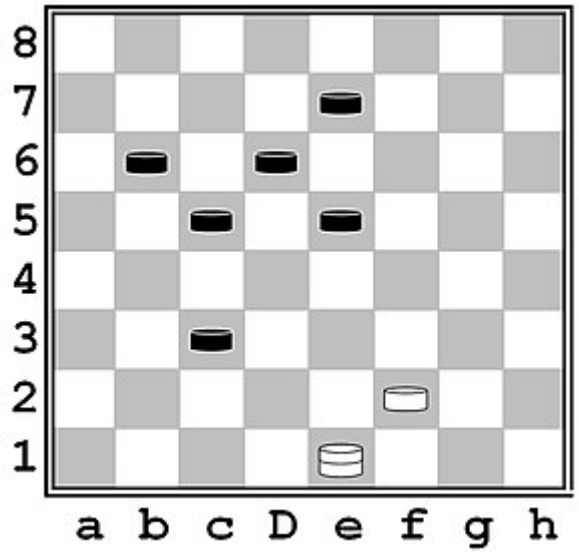

Рисунок 3. 2 - Приклад турецького удару

У функції AddMove() інтерес також представляє окрема обробка ситуації, коли шашка своїм ходом перетворюється на дамку - у такому разі можна продовжити взяття за її правилами. Функція MakeMove здійснює хід на дошці:

```
public void MakeMove(Move move, bool memoriseMove = false)
{
    // Хапамаємо хід, якщо треба
    if (memoriseMove)
        LastMoves.Add(new MemorisedMove(move.Fr, move.To, null, move.IsKinging,
_rowKingsMoves));

    // Рухаємо фігуру (оновлює масив _board та позицію в екземплярі самої фігури,
    // можливо, також перетворює екземпляр фігури на дамку)
    MovePiece(move.Fr, move.To);

    // Видаляємо з масивів побиті шашки
    foreach (Vector2Int taken in move.Taken)




    {
        Piece takenPiece = GetPiece(taken);
        _countCheckers[(int)takenPiece.Type]--;
        _countCheckers[0]--;

        Pieces.Remove(takenPiece);
        _board[taken.y, taken.x] = null;

        if (memoriseMove)
            LastMoves[LastMoves.Count - 1].AddTakenPiece(takenPiece);
    }
}
```

Зрозуміло, це лише основні функції. Повний код, що моніторить дошку, займає майже 500 рядків. Це досить багато, але не думаю, що можна якось поділити відповідальність: все стосується безпосередньо властивостей нинішнього стану гри.

3.3. Розроблення переможного алгоритму

Вибраний алгоритм - мінімакс. Це набір правил для прийняття рішень для ігор з нульовою сумою (один виграв - інший програв). Він ґрунтується на тому, щоб для всіх можливих ходів передбачити відповіді суперника, і для кожної відповіді — вибрати свою відповідь, на яку знову передбачити можливі відповіді суперника. Тобто, ми перебираємо всі варіанти на певну глибину. Зрозуміло, ми не можемо перебрати всі ходи на всю гру, тому дійшовши до максимальної глибини, ми просто аналізуємо отриману позицію і розуміємо, чи варто нам йти в цю гілку гри чи ні. Звичайно, ми в першу чергу розраховуємо, що противник робитиме найсильніші (на нашу думку) ходи.

Якщо кожну позицію оцінити одним числом (наприклад, позитивна перевага у білих, негативна у чорних), то на кожній ітерації ми намагаємося максимізувати оцінку позиції, якщо хід наш, і мінімізувати, якщо хід суперника. Справді, якщо позиція покращується для суперника, то вона погіршується для нас. Звідси назва алгоритму – мінімакс (рис. 3. 3). Мені дуже хочеться назвати це не алгоритмом, а штучним інтелектом, хоча він не є таким. Це тому, що при розробленні я назвав клас ШІ і так і продовжував про нього думати, зрозумівши помилку лише пізніше.

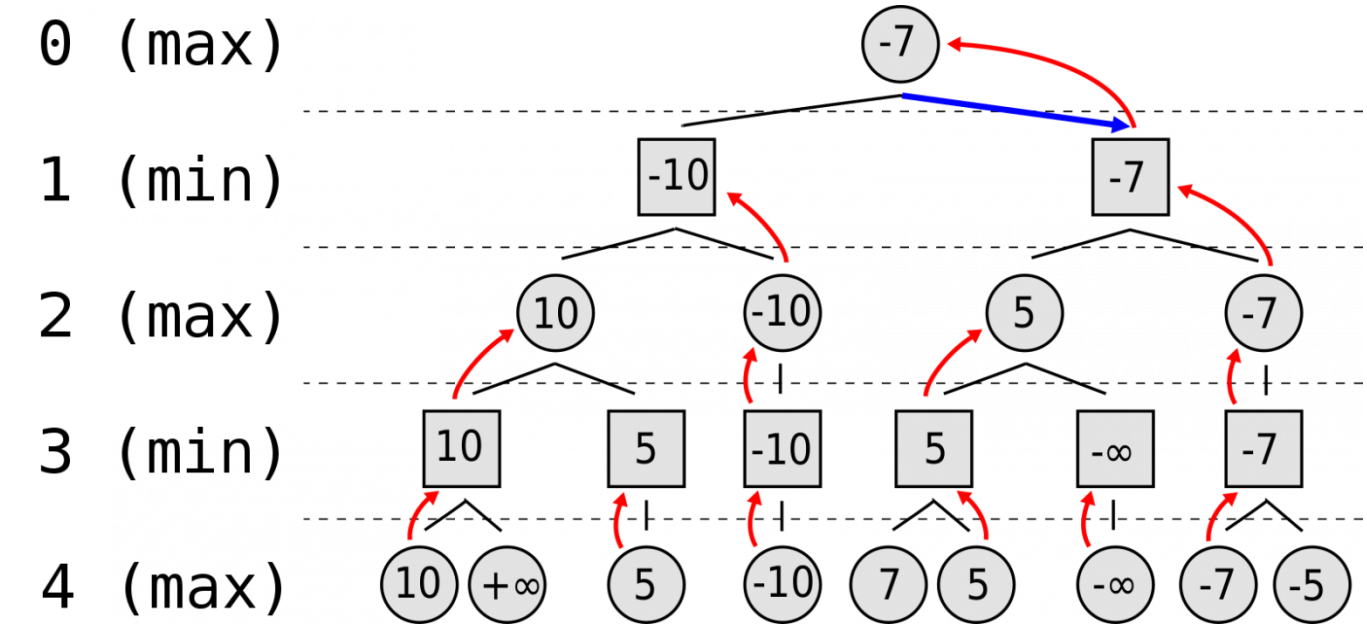

Рисунок 3. 3 – Приклад застосування алгоритму Мінімакс

Спочатку розберемося, як оцінювати конкретну позицію на дошці, а вже потім вчитимемося думати наперед, оцінюючи отримані позиції. Клас AI.sc вміє оцінювати позицію, підраховуючи якості шашок обох кольорів на дошці. Якість розраховується як добуток вартості шашки, спеціального бонусу клітини (наприклад, шашки в центрі дорожчі за шашки з лівого або правого краю дошки) та Y-бонусу (бонус по вертикалі: проста шашка тим дорожча, чим ближче вона до дамкових полів).

```
Якість шашки = value * _squareBonus * yBonus
```

Значення вартості та бонусів ми обрали наступні:

```
int _checkerValue = 100; // вартість простої шашки
int _kingValue = 250; // вартість короля
float[,] _squareBonus = new float[8, 4] // бонус клітинки
    {
        { 1.2f, 1.2f, 1.2f, 1.2f },
        { 1.15f, 1.2f, 1.2f, 1.15f },
        { 1.15f, 1.2f, 1.2f, 1.13f },
        { 1.0f, 1.2f, 1.15f, 1.0f },
        { 1.0f, 1.2f, 1.2f, 1.0f },
        { 1.0f, 1.0f, 1.0f, 1.0f },
        { 1.0f, 1.0f, 1.0f, 1.0f },
        { 1.0f, 1.0f, 1.0f, 1.0f },
    };

private float[] _yBonus = new float[8]; // Y-бонус
```

```
public float EvaluateMaterialAndPosition (Board board)
    {
        float eval = 0;
        // Розраховуємо якість кожної шашки
        foreach (Piece piece in board.Pieces)
        {
            Vector2Int coord = piece.Pos;
            switch (piece.Type)
            {
                case PieceType.WHITE_MAN:
                    eval += _checkerValue * _yBonus[coord.y] * _squareBonus[coord.y, coord.x
/ 2];
                    break;
                case PieceType.BLACK_MAN:
                    eval -= _checkerValue * _yBonus[7 - coord.y] * _squareBonus[7 - coord.y,
3 - coord.x / 2];
                    break;
                case PieceType.WHITE_KING:
                    eval += _kingValue;
                    break;
                case PieceType.BLACK_KING:
                    eval -= _kingValue;
                    break;
            }
        }
        return eval;
    }
```

Тепер, коли ми вміємо оцінювати позицію, рекурсивно перебиратимемо всі наші ходи, відповіді на них суперника, наші відповіді на відповіді суперника тощо. Оскільки ми не знаємо, наскільки складна нинішня позиція і скільки таких ітерацій потрібно, будемо збільшувати кількість ітерацій, тобто. спочатку проаналізуємо позицію на глибину 2 ходи, потім 4, 6 і т. д. Це називається пошук у глибину з ітеративним поглибленням. При цьому, щоб уникнути нескінченного пошуку, введемо максимальний час аналізу: після його закінчення ми негайно виходимо з усіх функцій і використовуємо результат останньої повністю завершеної ітерації.

```
// Функція активного пошуку ходу запускає пошук кращого ходу в позиції
public void ActiveSearch ()
{
    int depth = 0,  startDepth = 2;
    CurrentBestMove = Move.None;
  // Єдиний у позиції хід робиться без роздумів
    if (_board.AllMoves.Count == 1)
    {
        CurrentBestMove = _board.AllMoves[0];
        return;
    }
    // Робимо копію дошки, на якій будемо проводити аналіз
    // Це потрібно, тому що під час аналізу ми пересуватимемо фігури
    Board boardCopy = new Board(_board, deepCopyMoves: true);
    _searchStartTime = DateTime.Now;

    IterativeDeepeningMinimax(boardCopy, _timeLimit, startDepth, ActiveSearchDepth, ref
CurrentBestMove, ref depth, true);

    if (CurrentBestMove == Move.None)
        CurrentBestMove = boardCopy.AllMoves[new System.Random().Next(0,
boardCopy.AllMoves.Count)];
}

// Функція мінімакса з ітеративним поглибленням: запускає мінімакс з дедалі більшою та
більшою глибиною,
// при цьому стежачи за обмеженням у часі
public void IterativeDeepeningMinimax (Board board, float timeLimit, int minDepth, int
maxDepth, ref Move bestMove, ref int depth, bool isWhileActiveSearch)
    {
        for (depth = minDepth; depth <= maxDepth; depth++)
        {
            (float eval, Move tempBestMove) = Minimax(board, depth, board.WhitesMove,
timeLimit);
            // Якщо встигли повністю завершити ітерацію, зберігаємо її результат
            if ((DateTime.Now - _searchStartTime).TotalSeconds < timeLimit && tempBestMove
is not null && tempBestMove != Move.None)
            {
                bestMove = (Move) tempBestMove.Clone();
            }
          // Якщо не встигли та ітерація завершилася екстрено, вона неповна і її результат
нам не потрібен            else
            {
                depth -= 1;
                break;
            }

            // Ми перестаємо шукати, якщо на ітерації знайдемо форсований виграш
            if (eval >= Infinity && board.WhitesMove || eval <= -Infinity &&
!board.WhitesMove)
                break;
        }
    }
```

```
// Функція мінімаксу знаходить найкращий хід у позиції конкретного гравця
// Повертає сам хід, а також оцінку позиції, яка вийде, якщо цей хід зробити
// depth показує, на скільки ще ітерацій-рекурсій нам залишилося заглибитись (з кожним новим
рекурсивним викликом depth зменшується)
// maximizingPlayer показує, якого гравця ми шукаємо кращий хід, тобто позицію для якого
гравця ми намагаємося покращити
public (float, Move) Minimax (Board board, int depth, bool maximizingPlayer, float
timeLimit)
        {
          // Перевірка часу
            if ((DateTime.Now - _searchStartTime).TotalSeconds >= timeLimit)
                return (0, null);

          // Перевіряємо нинішній стан позиції (можливо вже гейм овер)
            GameState state = board.GetGameState();
            if (state != GameState.IN_PROCESS)
            {
                if (state == GameState.WHITE_WIN)
                    return (Infinity + depth, Move.None);
                if (state == GameState.BLACK_WIN)
                    return (-Infinity - depth, Move.None);

                else
                    return (0, Move.None);
            }

            // Якщо це остання ітерація, просто повертаємо оцінку позиції
              // Хід тут не важливий, тому що найкращим стане саме хід, що веде до позиції з
найкращою оцінкою
            if (depth == 0)
            {
                float eval = Evaluate(board);
                return (eval, Move.None);
            }

            // Якщо хід єдиний – див. коментарі під кодом
            if (board.AllMoves.Count == 1)
            {
                Move move = board.AllMoves[0];

                board.MakeMove(board.AllMoves[0], memoriseMove: true);
                board.OnMoveFinished(board.AllMoves[0]);
                float eval = Minimax(board, depth, alpha, beta, !maximizingPlayer,
timeLimit, isWhileActiveSearch).Item1;
                board.UnmakeLastMove();
                _transpositions.Add(new Transposition(PositionCache, eval, Infinity,
board.AllMoves[0]));
                return (eval, board.AllMoves[0]);
            }

            // Шукаємо найкращий хід (за білих)
            Move bestMove = Move.None;
            if (maximizingPlayer)
            {
                float maxEval = -Infinity;
              // Проходимося всіма ходами
                foreach (Move move in board.AllMoves)
                {
                  // Робимо його
                    board.MakeMove(move, memoriseMove: true);
                    board.OnMoveFinished(move);
```

```
                // І запускаємо мінімакс із отриманої позиції, але з боку ПРОТИВНИКА
                  (float eval, Move compMove) = Minimax(board, depth - 1, alpha, beta,
false, timeLimit, isWhileActiveSearch);

                // Скасовуємо зроблений хід
                  board.UnmakeLastMove();

                // Перевірка, що мінімакс з боку супротивника не завершився екстрено через
брак часу
                  if (compMove == null)
                      return (0, null);
                // Перевіряємо, чи є цей хід кращим за найкраще знайденого
                  if (eval > maxEval)
                  {
                      maxEval = eval;
                      bestMove = move;
                  }
              }
              return (maxEval, bestMove);
          }


        // Аналогічно за чорних
          else
          {
              float minEval = Infinity;
              foreach (Move move in board.AllMoves)
              {
                  board.MakeMove(move, memoriseMove: true);
                  board.OnMoveFinished(move);

                  (float eval, Move compMove) = Minimax(board, depth - 1, alpha, beta,
true, timeLimit, isWhileActiveSearch);
                  board.UnmakeLastMove();

                  if (compMove == null)
                      return (0, null);
                  if (eval < minEval)
                  {
                      minEval = eval;
                      bestMove = move;
                  }
              }
              return (minEval, bestMove);
          }
      }
```

У функції IterativeDeepeningMinimax наочно показано, як ми поступово заглиблюємося у пошуку. Якщо чергова ітерація завершилася успішно, ми запам'ятовуємо найкращий згідно з нею хід; якщо вона завершилася достроково та екстрено через закінчення часу пошуку, то вона неповна і її результат, некоректний, нам не потрібен.

Варто відмітити, якщо хід у позиції єдиний, то він гарантовано буде зроблено, а тому ми не витрачаємо на нього обчислювальні потужності, а просто робимо його. При цьому ми навіть не зменшуємо кількість ітерацій, що залишилися, щоб аналізувати глибше. Тому в результаті ми можемо навіть отримати гілку на кілька ітерацій, глибше запланованого.

Слід звернути увагу на те, що ми не копіюємо дошку, щоб проаналізувати кожен хід: ВЕСЬ пошук кращого ходу виконується на одній дошці (копії ігрової), ми вміємо здійснювати (make) і скасовувати ходи (unmake). Таким чином, важка функція глибокого копіювання дошки викликається лише один раз.

В цілому, такий код вже здатний грати в шашки і навіть не ігнорувати фігури в один-два ходи.

3.4. Оптимізація алгоритму

Першим кроком оптимізації даного алгоритму, який в рази покращив швидкість аналізу, став метод альфа-бета відсікання. Його суть полягає в тому, що, наприклад, передбачаючи хід противника, ми навіть не розглядатимемо відверто зайві ходи, які він не зробить. Тобто він, звичайно, може їх зробити, але якщо вони незрозумілі — то нам же краще.

Наприклад, на рисунку 3. 4, який зображений нижче ми (граючи за максимізуючу оцінку сторону (за білих)), прорахувавши перші 2 варіанти ходу, бачимо, що можемо отримати позицію з оцінкою 6. Тому, коли ми розраховуємо третій хід і бачимо, що одна з його подальших гілок призводить до позиції з оцінкою 5, то далі ми навіть не розраховуємо, оскільки, навіть якщо там і буде щось вище, суперник краще вибере 5, адже він мінімізує (чорні) сторону. А тому ми навіть не будемо далі аналізувати 3-ю гілку, адже краще просто піти по 2-й.

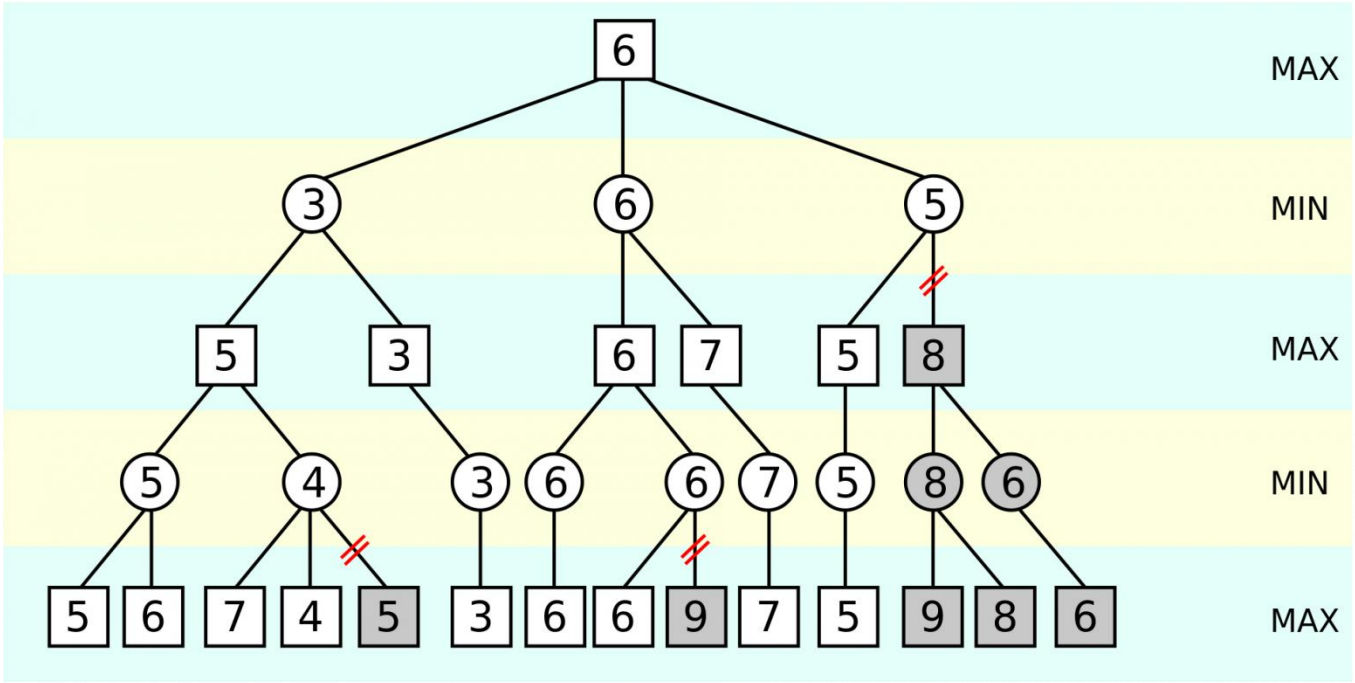

Рисунок 3. 4 - Метод альфа-бета відсікання

Таким чином, ми можемо повідсікати (prune) багато зайвих гілок, позбавившись від заздалегідь непотрібних обчислень. В інтернеті багато інформації про це покращення, тому я не буду далі його пояснювати, просто покажу, як я його реалізував.

У функції мінімакс я додав аргументи alpha і beta, які за першого виклику з IterativeDeepeningMinimax передаються як -Infinity і Infinity відповідно. Після 115-го рядка я додав перевірку на відсікання по альфі:

```
...
alpha = Mathf.Max(alpha, eval);
if (beta <= alpha)
  break;
...
```

А після 139-ї по беті:

```
...
beta = Mathf.Min(beta, eval);
if (beta <= alpha)
    break;
...
```

Під час прорахунку позицій часто виникають однакові позиції. Зрозуміло, немає сенсу розраховувати їх знову і знову в різних гілках мінімаксу, а тому ми можемо зберігати якось знайдений кращий хід позиції.

Варто розуміти, що якщо позиції X на глибині d знайдено кращий хід n, то цей же хід є кращим в тій же позиції і на глибинах менше d. А ось на глибинах більше d — не факт: можливо, нам здавалося, що він найкращий, але ми просто не дорахували і справді хід програшний. Якщо ж у позиції X хід n веде до форсованої перемоги, то глибину можна вважати нескінченною, оскільки ми точно знаємо, що хід виграшний.

Позиції вважаються однаковими, якщо не лише всі шашки стоять на однакових місцях, а й черговість ходу збігається. Транспозиції зберігатимуться у списку private List<Transposition> _transpositions = new List<Transposition>();

Де клас транспозиції виглядає так:

```
public class Transposition
{
    public string Pos { get; private set; }
    public float Eval { get; private set; }
    public int Depth { get; private set; }
    public Move BestMove { get; private set; }

    public Transposition(string pos, float eval, int depth, Move bestMove)
    {
        Pos = pos;
        Eval = eval;
        Depth = depth;
        BestMove = bestMove;

    }
    public bool IsSameTo (string otherPos)
        => Pos == otherPos;
}
```

На початку функції Minimax() (після, звичайно, перевірки на закінчення часу) перевірятимемо, чи не зустрічали ми раніше цю позицію на відповідній глибині:

```
string PositionCache = board.Board2Number(); // Переводить позицію в рядок

Transposition pos_trans = null;
pos_trans = _transpositions.FirstOrDefault(tr => tr.IsSameTo(PositionCache));
if (pos_trans != null)
{
    if (pos_trans.Depth >= depth)
    {
        return (pos_trans.Eval, pos_trans.BestMove);
    }
}
```

Позиції, до речі, порівнюються як рядки, так як кожну позицію можна однозначно подати як рядок із 32 символів - фігур на чорних полях (+1 символ для черговості ходу).

Якщо ж транспозиція не знайдена, запускаємо звичайний пошук. Коли ми завершуємо цикл, перед тим як винести рішення щодо позиції, додаватимемо її до списку транспозицій, як внесок у майбутнє:

```
// за білих
AddTransposition(new Transposition(PositionCache, maxEval, depth, bestMove));
return (maxEval, bestMove);


// за чорних
AddTransposition(new Transposition(PositionCache, minEval, depth, bestMove));
return (minEval, bestMove);
```

Функція AddTransposition просто додає транспозицію до списку, не забуваючи переконатися, що там вже немає такої позиції, але на меншій глибині. У такому разі вона стирається, бо навіщо нам менш глибокий аналіз, якщо вже маємо глибший.

Отже, дане покращення дозволяє отримати ще більш швидких характеристик з нашого алгоритму.

Особливу складність при розрахунках становлять дебюти (початок гри) та ендшпілі (її закінчення). Це так, тому що в дебюті треба вміти рахувати дуже далекоглядно, щоб зрозуміти наслідки нашого ходу, адже фігури лише розвиваються; в ендшпілі ж фігур менше, проте набагато більше ходів (за рахунок наявності далекобійних дамок) та аналіз навіть найпростіших позицій може бути довгим. До речі, з цієї причини саме в ендшпілях транспозиції стають дуже актуальними і корисними.

На жаль, мені не вдалося знайти якихось величезних дебютних та ендшпильних баз даних для шашок, хоча для таких шахів існують, причому можуть займати петабайти даних. Однак мені вдалося знайти стандартизовані записи шашкових партій і отримати з них дебюти та ендшпілі (кілька перших ходів з кожної партії та кілька останніх). Тому я підключаю до свого алгоритму два блоки:

```
private OpeningBook _openingBook;
private EndgameBook _endgameBook;
```

Класи OpeningBook і EndgameBook успадковуються від абстрактного TheoryBook, який вміє шукати цю позицію в довіднику та повертати найкращий у ній хід:

```
public abstract class TheoryBook
{
    protected abstract string TheoryPath { get; }
    private BookRecord _records;

    public TheoryBook()
    {
        Debug.Log(TheoryPath);
        using (StreamReader reader = new StreamReader(TheoryPath))
        {
            _records = JsonUtility.FromJson<BookRecord>(reader.ReadToEnd());
        }
        _records.BuildUpDictionary();
    }

    public bool TryGetBestMove(string pos, out string move)
    {
        if (_records.ContainsPosition(pos))
        {
            move = _records.GetMoveFor(pos, BookRecord.BookMoveSelector.Random);
            return true;
        }
        else
        {
            move = null;
            return false;
        }
    }
}

[System.Serializable]
public class BookRecord
{
    public List<string> positions;
    public List<string> moves;
    public int CountRecords => moves.Count;

    private Dictionary<string, List<string>> _pairs;
```

```
    public enum BookMoveSelector
    {
        Random, First, Last
    }

    public BookRecord()
    {
        positions = new List<string>();
        moves = new List<string>();
    }

    public void AddRecord(string pos, string move)
    {
        positions.Add(pos);
        moves.Add(move);
    }

    public void BuildUpDictionary()



    {
        _pairs = new Dictionary<string, List<string>>();
        for (int i = 0; i < CountRecords; i++)
        {
            if (_pairs.ContainsKey(positions[i]))
                _pairs[positions[i]].Add(moves[i]);
            else
                _pairs.Add(positions[i], new List<string>() { moves[i] });
        }
    }

    public bool ContainsPosition(string pos)
    {
        return _pairs.ContainsKey(pos);
    }
    public string GetMoveFor(string pos, BookMoveSelector selector)
    {
        switch (selector)
        {
            case BookMoveSelector.Random:
                return _pairs[pos][new System.Random().Next(0, _pairs[pos].Count)];
            case BookMoveSelector.First:
                return _pairs[pos][0];
            case BookMoveSelector.Last:
                return _pairs[pos][_pairs[pos].Count - 1];
            default:
                return _pairs[pos][0];
        }
    }
}
```

Тепер наша програма вміє робити перші 4-5 ходів моментально, доки у неї не закінчиться теорія. На жаль, вибірка партій була дуже маленькою (кілька тисяч), а й ендшпілі, і особливо дебюти в ній повторювалися, тому дуже сильного приросту якості гри це не дало. Знайти ж велику вибірку чи готові бази мені не вдалося.

3.5. Генерація ходів у грі шашки

У сфері розроблення ігор та штучного інтелекту створення компетентного ШІ для гри в шашки є цікавим завданням. Сьогодні ми розглянемо реалізацію ШІ мовою Golang, яка генерує всі можливі ходи для заданого стану дошки. Цей ефективний алгоритм генерації ходів є основою створення ШІ для гри в шашки, який може змагатися на високому рівні.

Представлений тут код є частиною пакету під назвою damka, який представляє шашкову дошку та її можливі ходи. Він використовує типи і константи, такі як Board, Pos, Dir, і WhiteMan, BlackMan, WhiteKing і BlackKing для представлення стану гри і фігур на дошці.

Основною функцією цієї реалізації є AllMovesWithFlag(), яка повертає всі можливі ходи для поточного гравця разом із бульовим прапором, що вказує на наявність ходу захоплення. Якщо прапор дорівнює true, буде повернено лише ходи захоплення, оскільки вони обов'язкові в шашках. Функція AllMoves() є простою згорткою, яка не містить прапора.

Алгоритм починається з визначення фігур поточного гравця (або білих або чорних), потім ітеративно переглядає всі 32 позиції на дошці. Для кожної позиції, якщо знайдено фігуру поточного гравця, програма генерує всі можливі ходи цієї фігури, враховуючи як звичайні ходи, так і ходи захоплення.

Функція manMoves() генерує регулярні ходи для простих шашок, беручи до уваги напрям, в якому вона може рухатися (вгору-вліво і вгору-вправо для білих, вниз-вліво і вниз-вправо для чорних). Для цього викликається manMovesDir() з відповідним напрямом.

Функція manEats() обробляє ходи захоплення для фігури людини, пробуючи всі чотири можливі напрями (вгору-вліво, вгору-вправо, вниз-вліво і вниз-вправо) і викликаючи manEatsDir() для кожного з них. Якщо ви знайшли правильний хід взяття, функція повертає true, вказуючи, що можуть бути доступні інші ходи взяття.

Функція kingMoves() генерує звичайні ходи для дамок. Оскільки дамки можуть рухатися в будь-якому напрямку, вона викликає функцію kingMovesDir() для кожного з чотирьох напрямків (вгору-вліво, вгору-вправо, вниз-вліво та вниз-вправо).

Функція kingEats() генерує ходи взяття для дамок, аналогічно пробуючи всі чотири можливі напрями і викликаючи kingEatsDir() для кожного.

Алгоритм використовує комбінацію бітових маніпуляцій та попередньо виділених фрагментів для ефективного використання пам'яті. Параметр eaten - це 32-бітове ціле число без знака, де кожен біт представляє позицію на дошці. Це дозволяє алгоритму відстежувати захоплені фігури без додаткових витрат пам'яті.

Попередньо виділені фрагменти moves та eatMoves допомагають мінімізувати виділення пам'яті при генерації ходів, прискорюючи процес. Місткість цих фрагментів встановлена на розумну оцінку максимальної кількості ходів, яка може бути зроблена в будь-якій грі.

Ця реалізація генерації ходів для ШІ у шашках мовою Golang демонструє можливості ефективних алгоритмів та структур даних. Досліджуючи різні підходи до створення всіх можливих ходів при заданому стані дошки, ми заклали основу для створення конкурентоспроможного ШІ для гри в шашки.

3.6. Подання дошки за допомогою двох uint64

Під час розроблення ігор, особливо настільних ігор, як шашки, представлення дошки є важливим компонентом продуктивності гри. Дошка - це, по суті, стан гри: на ній зберігається вся інформація про положення фігур, про те, чий зараз хід та інші важливі дані гри. Ефективність представлення дошки впливає на швидкість і ефективність генерації ходів, що у своє чергу впливає на ігровий процес.

Одним з найпоширеніших методів представлення дошки є використання двовимірного масиву, де кожна клітина є квадратом на дошці. Хоча цей підхід інтуїтивно зрозумілий, він не є найефективнішим, особливо для ігор зі складними правилами або великими розмірами дошки. Для таких ігор нам потрібний ефективніший метод представлення дошки. На допомогу приходить "бітове пакування" - техніка, що використовує бітові операції для зберігання та маніпулювання інформацією про стан гри. У даному підрозділі ми розглянемо реалізацію гри в шашки, яка використовує бітову упаковку для представлення ігрового поля за допомогою двох uint64.

Фрагмент коду, який ми розглянемо, представляє шашкову дошку за допомогою двох 64-бітових беззнакових цілих чисел (uint64) U та V. Структура Board містить ці два цілих числа:

```
type Board struct {
    U, V uint64
}
```

Таке представлення значно скорочує обсяг пам'яті, необхідної для зберігання стану дошки, що дозволяє швидше генерувати та оцінювати хід.

Основна ідея цього подання - "упаковка бітів". Упаковка бітів - це техніка, коли кілька частин інформації зберігаються у одному двійковому числі шляхом виділення певної кількості бітів кожної частини інформації. У цій реалізації кожна фігура на дошці представлена всього 3 бітами, що дозволяє зберігати до 8 різних типів фігур:

```
const (
    Empty     Piece = 0b000
    WhiteMan  Piece = 0b001
    BlackMan  Piece = 0b010
    WhiteKing Piece = 0b011
    BlackKing Piece = 0b100
)
```

У нашому поданні шашкової дошки два цілих числа uint64, U і V, зберігають стан дошки та деяку додаткову інформацію:

1. U зберігає фігури у перших 16 позиціях на дошці та поточний хід (білі чи чорні) у старшому биті (MSB).
2. V зберігає постаті останніх 16 позиціях на дошці, і навіть кількість ходів, у яких беруть участь лише королі (у п'ять найменш значних бітах з 59-го по 63-й).

Даний код включає кілька методів для маніпулювання та взаємодії з поданням дошки, таких як:

1. Get(i Pos) - Отримує фігуру заданої позиції i.
2. Set(i Pos, c Piece) - встановлює фігуру в цій позиції i значення c.
3. OnlyKingMoves() - Повертає кількість ходів, у яких беруть участь лише королі.

4. inc() - Збільшує кількість ходів, у яких беруть участь лише королі.
5. IsDraw() - Визначає, чи є партія нічийною, перевіряючи кількість ходів лише королів.
6. Turn(kingMove bool) - Оновлює хід і збільшує ходи короля, якщо потрібно.
7. Transpose() - Перевертає дошку та інвертує кольори фігур.
8. GenerateMoveName(target Board) – генерує назву ходу для заданого стану дошки.

Це компактне та ефективне уявлення шашкової дошки дає безліч переваг:

- Зменшення використання пам'яті - використання лише двох цілих чисел uint64 значно зменшує обсяг пам'яті представлення дошки.
- Швидша генерація ходів - Ефективні операції маніпулювання бітами дозволяють швидше генерувати та оцінювати ходи, що має вирішальне значення для продуктивності ШІ.
- Спрощена робота з дошкою - Надані методи спрощують роботу з поданням дошки, знижуючи складність коду ШІ.

Подання дошки з бітовою упаковкою, представлене в цьому підрзділі, пропонує потужний спосіб оптимізації генерації ходів ШІ шашок. Використовуючи лише два цілих числа uint64 для зберігання стану дошки та застосовуючи ефективні операції маніпулювання бітами, ШІ може швидше оцінювати та генерувати ходи, що призводить до створення більш сильного та конкурентоспроможного гравця у шашки.

3.7. Реалізація розробленого алгортму в грі шашки

У попередніх підрозділах ми оглянули, як ефективно генерувати ходи та представляти шашкову дошку в Golang. Тепер ми заглибимося в серце нашої гри в шашки: ШІ, який ухвалює рішення. ШІ буде використовувати наш власний алгоритм, популярний метод прийняття рішень у настільних іграх.

Структура нашого алгоритму - це ядро ШІ, яке включає не тільки логіку гри, але й деяку статистику і кешування для оптимізації продуктивності. Ключовими полями у цій структурі є:

- MaxDepth: Межа глибини алгоритму Minimax. Він визначає, на скільки ходів уперед розраховуватиме ШІ.
- Net: Нейронна мережа для оцінки стану дошки.
- Cache: Кеш, в якому зберігаються оцінені стани, щоб уникнути зайвих обчислень.
- DB: База даних ендшпиля, яка містить попередньо обчислені оптимальні ходи для всіх можливих ендшпільних позицій.

Структура алгоритму також містить додаткові поля для внутрішнього використання та відстеження статистики:

- nodes: масив float64, який використовується для оцінки нейронної мережі.
- Evaluated: кількість позицій дошки, оцінених нейронною мережею.
- CutOffs: кількість гілок, обрізаних за допомогою альфа-бета відсікання.
- CacheHits: кількість звернень до кешу під час пошуку.
- DBHits: кількість звернень до бази даних ендшпіля під час пошуку.

Ці поля допомагають аналізувати продуктивність алгоритму та його оптимізації.

Функція Minimax представляє ядро алгоритму, приймаючи як вхідні дані стан дошки, глибину пошуку та значення альфа-бета. Вона перевіряє наявність кінцевих станів гри (виграш, програш, нічия) та базових випадків рекурсії (глибина досягає 0 або більше немає ходів захоплення). Потім він ітеративно перебирає всі можливі ходи, оновлюючи значення альфа або бета відповідно, і обрізає гілки, якщо альфа більша або дорівнює бета.

Функції BestMove та BestRandomMove - це те місце, де ШІ робить свій хід. Обидві функції генерують всі можливі ходи, перевіряють, чи гра закінчилася, а потім використовують функцію Minimax для пошуку кращого ходу.

Різниця між цими двома функціями полягає у виборі наступного ходу:

- BestMove вибирає хід з найбільшим (для білих) або найменшим (для чорних) рахунком як найкращий хід.

- В BestRandomMove використовується виважений випадковий вибір, де ваги розраховуються на основі оцінки функції Minimax. Це може бути використане для додавання деякої непередбачуваності гру ШІ.

Кешування є важливим оптимізацією продуктивності алгоритму Minimax. Зберігаючи рахунок раніше оцінених позицій дошки у полі Cache, ми можемо уникнути зайвих обчислень та значно прискорити пошук.

Для представлення оцінки використовується структура Score. Вона містить Rate, яка є оцінкою дошки, і Steps, що вказує на кількість ходів, необхідних для досягнення оціненої позиції.

Зі структурою Score пов'язаний цікавий метод Byte. Цей метод перетворює оцінку та кроки в один байт. Старший біт байта представляє оцінку (1 для перемоги, 0 для поразки, і обидва для нічиєї), інші біти представляють кроки. Це компактне уявлення зручне для зберігання та пошуку.

У шашках ендшпіль може бути дуже складним, з безліччю можливих наслідків. Тому ми використовуємо базу даних ендшпіля (DB), яка містить попередньо обчислені оптимальні ходи всіх можливих позицій ендшпіля. Знайшовши поточний стан дошки у цій базі даних, ми можемо швидко визначити найкращий хід без необхідності шукати його у дереві гри. Це значно прискорює процес ухвалення рішень.

Функція Minimax використовує DB, якщо вона не дорівнює nil і стан дошки знаходиться в базі даних. Кількість успішних пошуків відстежується в полі DBHits структури Minimax.

Метод ClearStats використовується для скидання всіх статистичних полів у структурі Minimax. Його можна використовувати перед початком нового пошуку, щоб переконатися, що статистика точно відбиває продуктивність алгоритму поточного стану гри.

Реалізація власного алгоритму, як показано в даному розділі, закладає міцний фундамент для інтелектуального ШІ у грі шашки. Використовуючи ефективну генерацію ходів, компактне представлення дошки, кешування та базу даних

ендшпилів, ми можемо оптимізувати роботу ШІ та гарантувати, що він грає в гру оптимально (рис. 3. 5).

Рисунок 3. 5 – Процес роботи алгоритму

Проєкт включає в себе розроблений алгоритм та ШІ, який слідує правилам гри в шашки:

1. ШІ з розробленим алгоритмом і альфа-бета обрізанням.
2. База даних "endgame" (опціонально).

3. Нейромережева оцінка позиції: ШІ damka використовує нейронну мережу для оцінки позицій на дошці, що дозволяє більш точно оцінювати стан гри.

Тестування розробленого алгоритму для гри в шашки з метою перевірки працездатності наведено на рис. 3. 6, рис. 3. 7 та рис. 3. 8, рис. 3. 9.

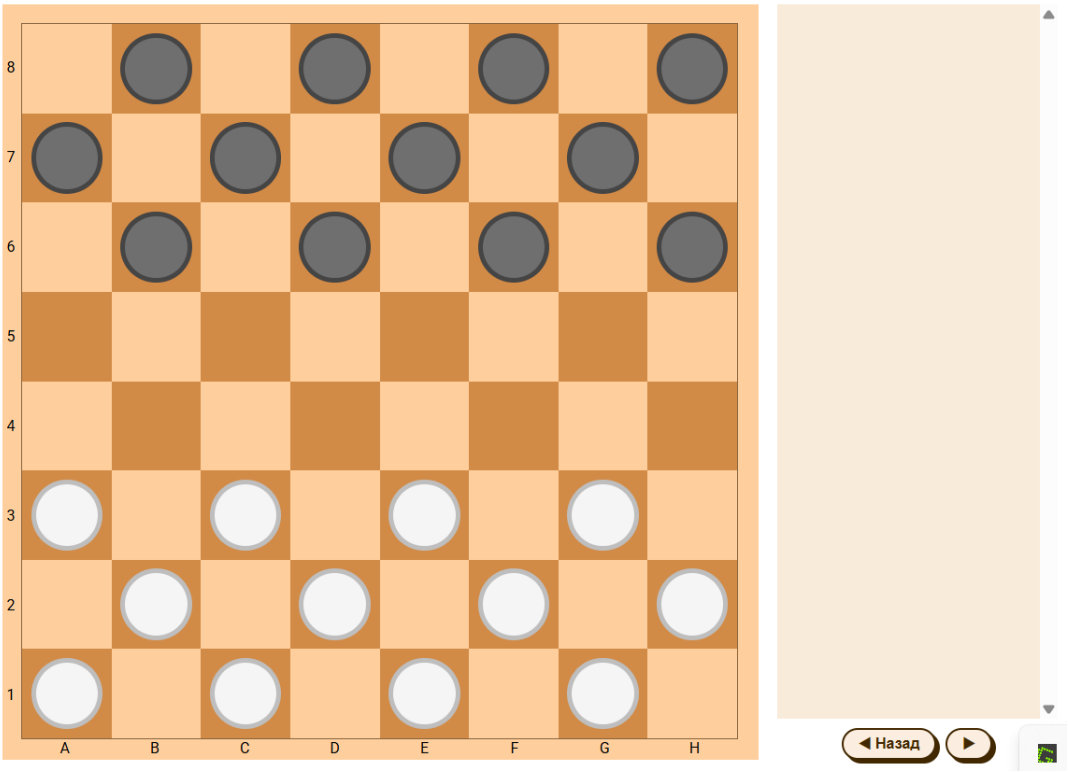

Рисунок 3. 6 – Графічний інтерфейс початку гри в шашки

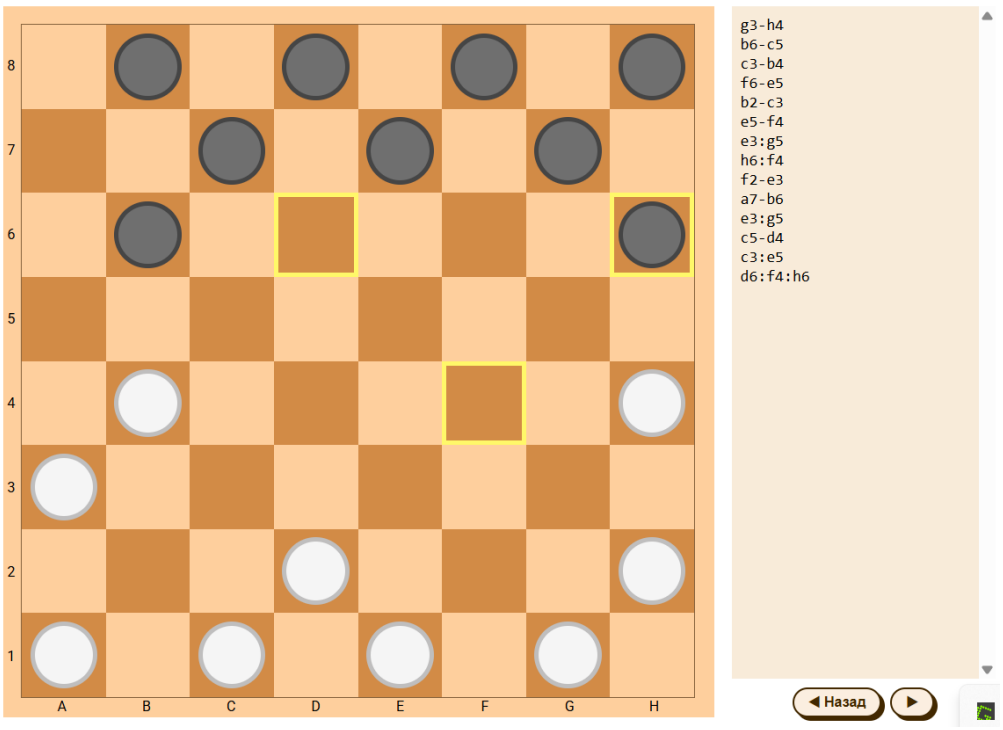

Рисунок 3. 7 – Процес гри в шашки

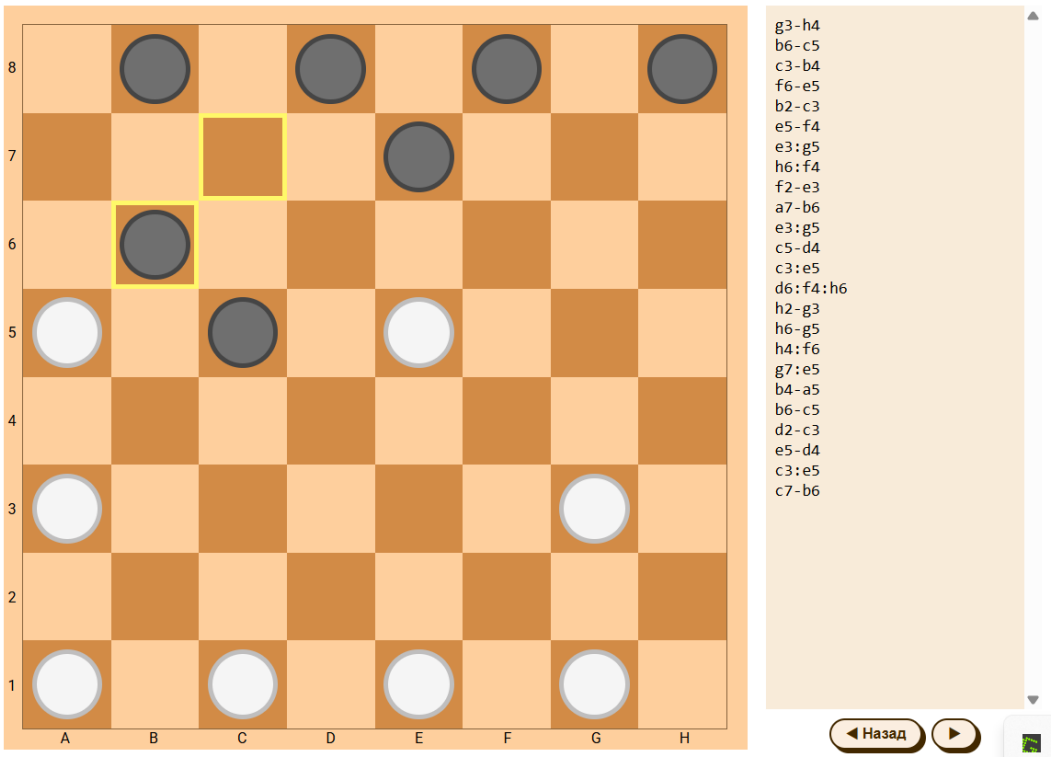

Рисунок 3. 8 – Рекомендація гравцеві побиття шашки

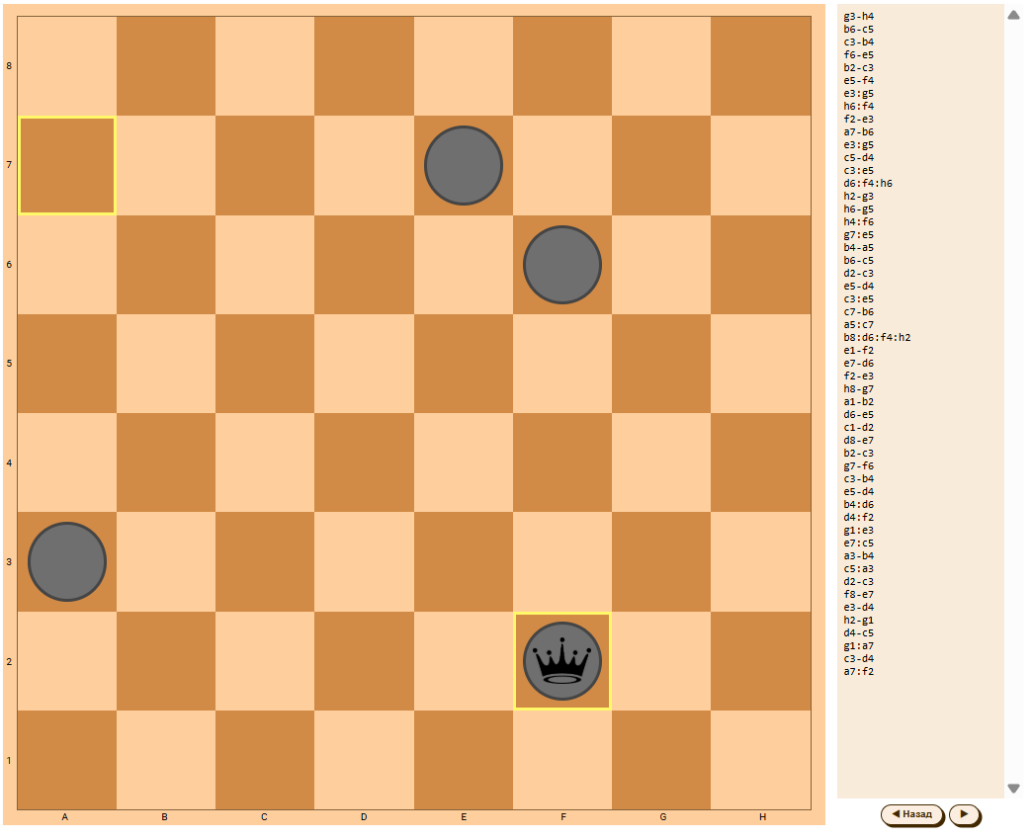

Рисунок 3. 9 – Завершення гри (перемога чорних шашок)

3.8. Вимоги до програмного та апаратного забезпечення

Для гри в **шашки** на комп'ютері Вам знадобиться відповідне програмне та апаратне забезпечення. Ось загальні вимоги:

Апаратні вимоги:

- **Операційна система**: Windows 7, 8, 10, 11 або macOS.
- **Процесор**: Intel Core i3 або аналогічний AMD.
- **Оперативна пам'ять**: 2 ГБ або більше.
- **Графічна карта**: Вбудована або дискретна, підтримка OpenGL або DirectX.
- **Місце на диску**: 100 МБ або більше.
- **Інтернет-з'єднання**: Необов'язкове, якщо грати офлайн.

Програмні вимоги:

- Мова програмування: C#;

- Ігровий движок: Unity

- Мережеві можливості: WebSockets або REST API.

ВИСНОВКИ

В результаті написання даної бакалаврської роботи було отримано працюючу програму та алгоритм для гри шашки. Розробляючи алгоритм та його оптимізацію, ми ґрунтувалися на статтях про розроблення програм для шахів. Саме тому мені довелося самому будувати функцію оцінки позиції і вона може бути неточною. Наприклад, можливо, відношення вартості звичайної шашки до дамки має бути не 100:250, а 100:150 або 100:500. Можливо, стояти в центрі, а не на краю шашкам вигідніше не в 1.25 рази, а в 1.1 чи 1.5.

Зрозуміло, це все можна налаштувати, якщо реалізувати "турнір" між комп'ютерами та поступово змінювати ці числа, проте щоб програма могла адекватно грати, їй потрібно 10-15 секунд на КОЖНИЙ ХІД (що дає глибину аналізу 9-10 ходів уперед). Так як у шашкової партії в середньому ходів 30, одна така партія може зайняти 5-8 хвилин, а щоб побудувати нормальний процес мутаційної еволюції потрібно організувати, мабуть, сотні чи тисячі партій.

Я також не заперечую, що певний поріг швидкості може задавати Unity, оскільки, швидше за все, на чистому C++ алгоритм працюватиме швидше, ніж на Unity+C.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. AlphaGo [Електронний ресурс] – Режим доступу до ресурсу: https://deepmind.com/research/case-studies/alphago-the-story-so-far.
2. Gaming tree concept [Електронний ресурс] – Режим доступу до ресурсу: https://philippmuens.com/minimax-and-mcts.
3. Tree search alghorithm [Електронний ресурс] – Режим доступу до ресурсу: https://towardsdatascience.com/4-types-of-tree-traversal-algorithms-d56328450846.
4. informal search [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/search-algorithms-in-ai/.
5. Minimax [Електронний ресурс] – Режим доступу до ресурсу: https://www.javatpoint.com/mini-max-algorithm-in-ai.
6. MCTS [Електронний ресурс] – Режим доступу до ресурсу: https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa.
7. Rule-based system [Електронний ресурс] – Режим доступу до ресурсу: https://stackoverflow.com/questions/5248063/what-is-rule-based-algorithm.
8. Unsure logic [Електронний ресурс] – Режим доступу до ресурсу: https://learntocodewith.me/posts/algorithmic-thinking/.
9. GOAT [Електронний ресурс] – Режим доступу до ресурсу: https://gamedev.stackexchange.com/questions/136832/what-are-the-basics-ofimplementing-a-goal-oriented-ai.
10. C# documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com /dotnet/csharp/.
11. Schaeffer, Jonathan. One Jump Ahead: Challenging Human Supremacy in Checkers // Springer, 1997 – 496 p.
12. Checkers app on Google Play: https://play.google.com/store/apps/details?id=com.dimcoms.checkers&h.
13. Butenko M. English Checkers Web App made with ClojureScript // GitHub, Sep 14, 2016, https://github.com/FI-Mihej/English-Checkers-Web-App.

14. Marissa Eppes, Game Theory — The Minimax Algorithm Explained // Towards Data Science, Aug 7, 2019, https://towardsdatascience.com/how-a-chess-playing-computer-thinksabout-its-next-move-8f028bd0e7b1
15. Don Ross, Game Theory // The Stanford Encyclopedia of Philosophy (Fall 2021 Edition), Edward N. Zalta (ed.), https://plato.stanford.edu/archives/fall2021/entries/game-theory/
16. Ekbia H., Artificial Dreams: The Quest for Non-Biological Intelligence. // Cambridge: Cambridge University Press, 2008. doi:10.1017/CBO9780511802126.

ДОДАТКИ

ДОДАТОК А

Лістинг програмного коду застосування алгоритму

```
package main

import (
        _ "embed"
        "fmt"
        "os"
        "time"

        . "checkers/src"
)

func main() {
        b := Board{}.
                Set(Parse("a7"), X).
                Set(Parse("c5"), O).
                Set(Parse("e5"), X).
                Set(Parse("h2"), X)
        fmt.Printf("Starting position:\n%s\n", b)
        minimax(b)
}

func minimax(b Board) {
        var db EndgameDB
        var err error
        db, err = os.ReadFile("endgame.db")
        if err != nil {
                panic(err)
        }

        m := NewMinimax(Zero, 8, db)

        start := time.Now()
        best, rate, steps := m.BestMove(b, true)
        d := time.Since(start)

        fmt.Println(best)
        fmt.Printf("  best move: %v\n", b.GenerateMoveName(best))
        fmt.Printf("       rate: %v\n", rate)
        fmt.Printf("      steps: %v\n", steps)
        fmt.Printf("   duration: %.2f seconds\n", d.Seconds())
        fmt.Printf("  evaluated: %v\n", m.Evaluated)
        fmt.Printf("   cut offs: %v\n", m.CutOffs)
        fmt.Printf(" cache size: %v\n", len(m.Cache))
        fmt.Printf(" cache hits: %v\n", m.CacheHits)
        fmt.Printf("    db hits: %v\n", m.DBHits)
}
package main

import (
        "fmt"

        . "checkers/src"
)

func main() {
```

```
        b := Board{}.
                Set(4, WhiteKing).
                Set(13, WhiteMan).
                Set(17, BlackKing).
                Turn(false)
        fmt.Printf("Starting position:\n%s\n", b)
        moves := b.AllMoves()
        for i, m := range moves {
                fmt.Printf("\n№%v %v\n%s\n", i+1, b.GenerateMoveName(m), m)
        }
}
package main

import (
        . "checkers/src"
)

func main() {
        b := NewBoard()
        player1 := NewMinimax(Zero, 6, nil)
        player2 := NewMinimax(Zero, 6, nil)
        Play(b, player1, player2, true)
}
```

ДОДАТОК Б

Лістинг програмного коду гри шашки

```
"use strict";

(() => {
	const enosys = () => {
		const err = new Error("not implemented");
		err.code = "ENOSYS";
		return err;
	};

	if (!globalThis.fs) {
		let outputBuf = "";
		globalThis.fs = {
			constants: { O_WRONLY: -1, O_RDWR: -1, O_CREAT: -1, O_TRUNC: -
1, O_APPEND: -1, O_EXCL: -1 }, // unused
			writeSync(fd, buf) {
				outputBuf += decoder.decode(buf);
				const nl = outputBuf.lastIndexOf("\n");
				if (nl != -1) {
					console.log(outputBuf.substring(0, nl));
					outputBuf = outputBuf.substring(nl + 1);
				}
				return buf.length;
			},
			write(fd, buf, offset, length, position, callback) {
				if (offset !== 0 || length !== buf.length || position
!== null) {
					callback(enosys());
					return;
				}
				const n = this.writeSync(fd, buf);
				callback(null, n);
			},
			chmod(path, mode, callback) { callback(enosys()); },
			chown(path, uid, gid, callback) { callback(enosys()); },
			close(fd, callback) { callback(enosys()); },
			fchmod(fd, mode, callback) { callback(enosys()); },
			fchown(fd, uid, gid, callback) { callback(enosys()); },
			fstat(fd, callback) { callback(enosys()); },
			fsync(fd, callback) { callback(null); },
			ftruncate(fd, length, callback) { callback(enosys()); },
			lchown(path, uid, gid, callback) { callback(enosys()); },
			link(path, link, callback) { callback(enosys()); },
			lstat(path, callback) { callback(enosys()); },
			mkdir(path, perm, callback) { callback(enosys()); },
			open(path, flags, mode, callback) { callback(enosys()); },
			read(fd, buffer, offset, length, position, callback) {
callback(enosys()); },
			readdir(path, callback) { callback(enosys()); },
			readlink(path, callback) { callback(enosys()); },
			rename(from, to, callback) { callback(enosys()); },
			rmdir(path, callback) { callback(enosys()); },
			stat(path, callback) { callback(enosys()); },
			symlink(path, link, callback) { callback(enosys()); },
			truncate(path, length, callback) { callback(enosys()); },
			unlink(path, callback) { callback(enosys()); },
			utimes(path, atime, mtime, callback) { callback(enosys()); },
		};
	}

	if (!globalThis.process) {
```

```
		globalThis.process = {
			getuid() { return -1; },
			getgid() { return -1; },
			geteuid() { return -1; },
			getegid() { return -1; },
			getgroups() { throw enosys(); },
			pid: -1,
			ppid: -1,
			umask() { throw enosys(); },
			cwd() { throw enosys(); },
			chdir() { throw enosys(); },
		}
	}

	if (!globalThis.crypto) {
		throw new Error("globalThis.crypto is not available, polyfill
required (crypto.getRandomValues only)");
	}

	if (!globalThis.performance) {
		throw new Error("globalThis.performance is not available, polyfill
required (performance.now only)");
	}

	if (!globalThis.TextEncoder) {
		throw new Error("globalThis.TextEncoder is not available, polyfill
required");
	}

	if (!globalThis.TextDecoder) {
		throw new Error("globalThis.TextDecoder is not available, polyfill
required");
	}

	const encoder = new TextEncoder("utf-8");
	const decoder = new TextDecoder("utf-8");

	globalThis.Go = class {
		constructor() {
			this.argv = ["js"];
			this.env = {};
			this.exit = (code) => {
				if (code !== 0) {
					console.warn("exit code:", code);
				}
			};
			this._exitPromise = new Promise((resolve) => {
				this._resolveExitPromise = resolve;
			});
			this._pendingEvent = null;
			this._scheduledTimeouts = new Map();
			this._nextCallbackTimeoutID = 1;

			const setInt64 = (addr, v) => {
				this.mem.setUint32(addr + 0, v, true);
				this.mem.setUint32(addr + 4, Math.floor(v /
4294967296), true);
			}

			const getInt64 = (addr) => {
				const low = this.mem.getUint32(addr + 0, true);
				const high = this.mem.getInt32(addr + 4, true);
				return low + high * 4294967296;
```

```
                        }

                        const loadValue = (addr) => {
                                const f = this.mem.getFloat64(addr, true);
                                if (f === 0) {
                                        return undefined;
                                }
                                if (!isNaN(f)) {
                                        return f;
                                }

                                const id = this.mem.getUint32(addr, true);
                                return this._values[id];
                        }

                        const storeValue = (addr, v) => {
                                const nanHead = 0x7FF80000;

                                if (typeof v === "number" && v !== 0) {
                                        if (isNaN(v)) {
                                                this.mem.setUint32(addr + 4, nanHead,
true);
                                                this.mem.setUint32(addr, 0, true);
                                                return;
                                        }
                                        this.mem.setFloat64(addr, v, true);
                                        return;
                                }

                                if (v === undefined) {
                                        this.mem.setFloat64(addr, 0, true);
                                        return;
                                }

                                let id = this._ids.get(v);
                                if (id === undefined) {
                                        id = this._idPool.pop();
                                        if (id === undefined) {
                                                id = this._values.length;
                                        }
                                        this._values[id] = v;
                                        this._goRefCounts[id] = 0;
                                        this._ids.set(v, id);
                                }
                                this._goRefCounts[id]++;
                                let typeFlag = 0;
                                switch (typeof v) {
                                        case "object":
                                                if (v !== null) {
                                                        typeFlag = 1;
                                                }
                                                break;
                                        case "string":
                                                typeFlag = 2;
                                                break;
                                        case "symbol":
                                                typeFlag = 3;
                                                break;
                                        case "function":
                                                typeFlag = 4;
                                                break;
                                }
```

```
                        this.mem.setUint32(addr + 4, nanHead | typeFlag,
true);
                        this.mem.setUint32(addr, id, true);
                }

                const loadSlice = (addr) => {
                        const array = getInt64(addr + 0);
                        const len = getInt64(addr + 8);
                        return new Uint8Array(this._inst.exports.mem.buffer,
array, len);
                }

                const loadSliceOfValues = (addr) => {
                        const array = getInt64(addr + 0);
                        const len = getInt64(addr + 8);
                        const a = new Array(len);
                        for (let i = 0; i < len; i++) {
                                a[i] = loadValue(array + i * 8);
                        }
                        return a;
                }

                const loadString = (addr) => {
                        const saddr = getInt64(addr + 0);
                        const len = getInt64(addr + 8);
                        return decoder.decode(new
DataView(this._inst.exports.mem.buffer, saddr, len));
                }

                const timeOrigin = Date.now() - performance.now();
                this.importObject = {
                        go: {
                                // Go's SP does not change as long as no Go
code is running. Some operations (e.g. calls, getters and setters)
                                // may synchronously trigger a Go event
handler. This makes Go code get executed in the middle of the imported
                                // function. A goroutine can switch to a new
stack if the current stack is too small (see morestack function).
                                // This changes the SP, thus we have to update
the SP used by the imported function.

                                // func wasmExit(code int32)
                                "runtime.wasmExit": (sp) => {
                                        sp >>>= 0;
                                        const code = this.mem.getInt32(sp + 8,
true);
                                        this.exited = true;
                                        delete this._inst;
                                        delete this._values;
                                        delete this._goRefCounts;
                                        delete this._ids;
                                        delete this._idPool;
                                        this.exit(code);
                                },

                                // func wasmWrite(fd uintptr, p
unsafe.Pointer, n int32)
                                "runtime.wasmWrite": (sp) => {
                                        sp >>>= 0;
                                        const fd = getInt64(sp + 8);
                                        const p = getInt64(sp + 16);
                                        const n = this.mem.getInt32(sp + 24,
true);
```

```
                                        fs.writeSync(fd, new
Uint8Array(this._inst.exports.mem.buffer, p, n));
                                },

                                // func resetMemoryDataView()
                                "runtime.resetMemoryDataView": (sp) => {
                                        sp >>>= 0;
                                        this.mem = new
DataView(this._inst.exports.mem.buffer);
                                },

                                // func nanotime1() int64
                                "runtime.nanotime1": (sp) => {
                                        sp >>>= 0;
                                        setInt64(sp + 8, (timeOrigin +
performance.now()) * 1000000);
                                },

                                // func walltime() (sec int64, nsec int32)
                                "runtime.walltime": (sp) => {
                                        sp >>>= 0;
                                        const msec = (new Date).getTime();
                                        setInt64(sp + 8, msec / 1000);
                                        this.mem.setInt32(sp + 16, (msec %
1000) * 1000000, true);
                                },

                                // func scheduleTimeoutEvent(delay int64)
int32
                                "runtime.scheduleTimeoutEvent": (sp) => {
                                        sp >>>= 0;
                                        const id = this._nextCallbackTimeoutID;
                                        this._nextCallbackTimeoutID++;
                                        this._scheduledTimeouts.set(id,
setTimeout(
                                                () => {
                                                        this._resume();
                                                        while
(this._scheduledTimeouts.has(id)) {
                                                                // for some
reason Go failed to register the timeout event, log and try again
                                                                // (temporary
workaround for https://github.com/golang/go/issues/28975)

        console.warn("scheduleTimeoutEvent: missed timeout event");
                                                                this._resume();
                                                        }
                                                },
                                                getInt64(sp + 8) + 1, //
setTimeout has been seen to fire up to 1 millisecond early
                                        ));
                                        this.mem.setInt32(sp + 16, id, true);
                                },

                                // func clearTimeoutEvent(id int32)
                                "runtime.clearTimeoutEvent": (sp) => {
                                        sp >>>= 0;
                                        const id = this.mem.getInt32(sp + 8,
true);

        clearTimeout(this._scheduledTimeouts.get(id));
                                        this._scheduledTimeouts.delete(id);
                                },
```

```
                                        // func getRandomData(r []byte)
                                        "runtime.getRandomData": (sp) => {
                                                sp >>>= 0;
                                                crypto.getRandomValues(loadSlice(sp +
8));
                                        },

                                        // func finalizeRef(v ref)
                                        "syscall/js.finalizeRef": (sp) => {
                                                sp >>>= 0;
                                                const id = this.mem.getUint32(sp + 8,
true);
                                                this._goRefCounts[id]--;
                                                if (this._goRefCounts[id] === 0) {
                                                        const v = this._values[id];
                                                        this._values[id] = null;
                                                        this._ids.delete(v);
                                                        this._idPool.push(id);
                                                }
                                        },

                                        // func stringVal(value string) ref
                                        "syscall/js.stringVal": (sp) => {
                                                sp >>>= 0;
                                                storeValue(sp + 24, loadString(sp +
8));
                                        },

                                        // func valueGet(v ref, p string) ref
                                        "syscall/js.valueGet": (sp) => {
                                                sp >>>= 0;
                                                const result = Reflect.get(loadValue(sp
+ 8), loadString(sp + 16));
                                                sp = this._inst.exports.getsp() >>> 0;
// see comment above
                                                storeValue(sp + 32, result);
                                        },

                                        // func valueSet(v ref, p string, x ref)
                                        "syscall/js.valueSet": (sp) => {
                                                sp >>>= 0;
                                                Reflect.set(loadValue(sp + 8),
loadString(sp + 16), loadValue(sp + 32));
                                        },

                                        // func valueDelete(v ref, p string)
                                        "syscall/js.valueDelete": (sp) => {
                                                sp >>>= 0;
                                                Reflect.deleteProperty(loadValue(sp +
8), loadString(sp + 16));
                                        },

                                        // func valueIndex(v ref, i int) ref
                                        "syscall/js.valueIndex": (sp) => {
                                                sp >>>= 0;
                                                storeValue(sp + 24,
Reflect.get(loadValue(sp + 8), getInt64(sp + 16)));
                                        },

                                        // valueSetIndex(v ref, i int, x ref)
                                        "syscall/js.valueSetIndex": (sp) => {
                                                sp >>>= 0;
```

```
                        Reflect.set(loadValue(sp + 8),
getInt64(sp + 16), loadValue(sp + 24));
                },

                // func valueCall(v ref, m string, args []ref)
(ref, bool)
                "syscall/js.valueCall": (sp) => {
                        sp >>>= 0;
                        try {
                                const v = loadValue(sp + 8);
                                const m = Reflect.get(v,
loadString(sp + 16));
                                const args =
loadSliceOfValues(sp + 32);
                                const result = Reflect.apply(m,
v, args);
                                sp = this._inst.exports.getsp()
>>> 0; // see comment above
                                storeValue(sp + 56, result);
                                this.mem.setUint8(sp + 64, 1);
                        } catch (err) {
                                sp = this._inst.exports.getsp()
>>> 0; // see comment above
                                storeValue(sp + 56, err);
                                this.mem.setUint8(sp + 64, 0);
                        }
                },

                // func valueInvoke(v ref, args []ref) (ref,
bool)
                "syscall/js.valueInvoke": (sp) => {
                        sp >>>= 0;
                        try {
                                const v = loadValue(sp + 8);
                                const args =
loadSliceOfValues(sp + 16);
                                const result = Reflect.apply(v,
undefined, args);
                                sp = this._inst.exports.getsp()
>>> 0; // see comment above
                                storeValue(sp + 40, result);
                                this.mem.setUint8(sp + 48, 1);
                        } catch (err) {
                                sp = this._inst.exports.getsp()
>>> 0; // see comment above
                                storeValue(sp + 40, err);
                                this.mem.setUint8(sp + 48, 0);
                        }
                },

                // func valueNew(v ref, args []ref) (ref,
bool)
                "syscall/js.valueNew": (sp) => {
                        sp >>>= 0;
                        try {
                                const v = loadValue(sp + 8);
                                const args =
loadSliceOfValues(sp + 16);
                                const result =
Reflect.construct(v, args);
                                sp = this._inst.exports.getsp()
>>> 0; // see comment above
                                storeValue(sp + 40, result);
```

```
                                        this.mem.setUint8(sp + 48, 1);
                                } catch (err) {
                                        sp = this._inst.exports.getsp()
>>> 0; // see comment above
                                        storeValue(sp + 40, err);
                                        this.mem.setUint8(sp + 48, 0);
                                }
                        },

                        // func valueLength(v ref) int
                        "syscall/js.valueLength": (sp) => {
                                sp >>>= 0;
                                setInt64(sp + 16, parseInt(loadValue(sp
+ 8).length));
                        },

                        // valuePrepareString(v ref) (ref, int)
                        "syscall/js.valuePrepareString": (sp) => {
                                sp >>>= 0;
                                const str =
encoder.encode(String(loadValue(sp + 8)));
                                storeValue(sp + 16, str);
                                setInt64(sp + 24, str.length);
                        },

                        // valueLoadString(v ref, b []byte)
                        "syscall/js.valueLoadString": (sp) => {
                                sp >>>= 0;
                                const str = loadValue(sp + 8);
                                loadSlice(sp + 16).set(str);
                        },

                        // func valueInstanceOf(v ref, t ref) bool
                        "syscall/js.valueInstanceOf": (sp) => {
                                sp >>>= 0;
                                this.mem.setUint8(sp + 24,
(loadValue(sp + 8) instanceof loadValue(sp + 16)) ? 1 : 0);
                        },

                        // func copyBytesToGo(dst []byte, src ref)
(int, bool)
                        "syscall/js.copyBytesToGo": (sp) => {
                                sp >>>= 0;
                                const dst = loadSlice(sp + 8);
                                const src = loadValue(sp + 32);
                                if (!(src instanceof Uint8Array || src
instanceof Uint8ClampedArray)) {
                                        this.mem.setUint8(sp + 48, 0);
                                        return;
                                }
                                const toCopy = src.subarray(0,
dst.length);
                                dst.set(toCopy);
                                setInt64(sp + 40, toCopy.length);
                                this.mem.setUint8(sp + 48, 1);
                        },

                        // func copyBytesToJS(dst ref, src []byte)
(int, bool)
                        "syscall/js.copyBytesToJS": (sp) => {
                                sp >>>= 0;
                                const dst = loadValue(sp + 8);
                                const src = loadSlice(sp + 16);
```

```
                                        if (!(dst instanceof Uint8Array || dst
instanceof Uint8ClampedArray)) {
                                                this.mem.setUint8(sp + 48, 0);
                                                return;
                                        }
                                        const toCopy = src.subarray(0,
dst.length);
                                        dst.set(toCopy);
                                        setInt64(sp + 40, toCopy.length);
                                        this.mem.setUint8(sp + 48, 1);
                                },

                                "debug": (value) => {
                                        console.log(value);
                                },
                        }
                };
        }

        async run(instance) {
                if (!(instance instanceof WebAssembly.Instance)) {
                        throw new Error("Go.run: WebAssembly.Instance
expected");
                }
                this._inst = instance;
                this.mem = new DataView(this._inst.exports.mem.buffer);
                this._values = [ // JS values that Go currently has references
to, indexed by reference id
                        NaN,
                        0,
                        null,
                        true,
                        false,
                        globalThis,
                        this,
                ];
                this._goRefCounts = new
Array(this._values.length).fill(Infinity); // number of references that Go has to a
JS value, indexed by reference id
                this._ids = new Map([ // mapping from JS values to reference
ids
                        [0, 1],
                        [null, 2],
                        [true, 3],
                        [false, 4],
                        [globalThis, 5],
                        [this, 6],
                ]);
                this._idPool = [];   // unused ids that have been garbage
collected
                this.exited = false; // whether the Go program has exited

                // Pass command line arguments and environment variables to
WebAssembly by writing them to the linear memory.
                let offset = 4096;

                const strPtr = (str) => {
                        const ptr = offset;
                        const bytes = encoder.encode(str + "\0");
                        new Uint8Array(this.mem.buffer, offset,
bytes.length).set(bytes);
                        offset += bytes.length;
                        if (offset % 8 !== 0) {
```

```
                                offset += 8 - (offset % 8);
                        }
                        return ptr;
                };

                const argc = this.argv.length;

                const argvPtrs = [];
                this.argv.forEach((arg) => {
                        argvPtrs.push(strPtr(arg));
                });
                argvPtrs.push(0);

                const keys = Object.keys(this.env).sort();
                keys.forEach((key) => {
                        argvPtrs.push(strPtr(`${key}=${this.env[key]}`));
                });
                argvPtrs.push(0);

                const argv = offset;
                argvPtrs.forEach((ptr) => {
                        this.mem.setUint32(offset, ptr, true);
                        this.mem.setUint32(offset + 4, 0, true);
                        offset += 8;
                });

                // The linker guarantees global data starts from at least
wasmMinDataAddr.
                // Keep in sync with
cmd/link/internal/ld/data.go:wasmMinDataAddr.
                const wasmMinDataAddr = 4096 + 8192;
                if (offset >= wasmMinDataAddr) {
                        throw new Error("total length of command line and
environment variables exceeds limit");
                }

                this._inst.exports.run(argc, argv);
                if (this.exited) {
                        this._resolveExitPromise();
                }
                await this._exitPromise;
        }

        _resume() {
                if (this.exited) {
                        throw new Error("Go program has already exited");
                }
                this._inst.exports.resume();
                if (this.exited) {
                        this._resolveExitPromise();
                }
        }

        _makeFuncWrapper(id) {
                const go = this;
                return function () {
                        const event = { id: id, this: this, args: arguments };
                        go._pendingEvent = event;
                        go._resume();
                        return event.result;
                };
        }
}
```

```
})();
```