

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

**Навчально-науковий інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу**

До захисту допущено:
Завідувач кафедри
_____ Оксана ТИМОЩУК
« ____ » _____ 2026 р.

Дипломна робота

на здобуття ступеня бакалавра

**за освітньо-професійною програмою «Системний аналіз і управління»
спеціальності 124 «Системний аналіз»**

**на тему: «Система прийняття рішень на основі штучного інтелекту для
розв'язання задачі вибору оптимального ходу у шашках»**

Виконав:

Студент IV курсу, групи КА-22

Анненков Павло Георгійович _____

Керівник:

Асистент кафедри ММСА, докт. філос.

Канцедал Георгій Олегович _____

Консультант з економічного розділу:

Доцент кафедри ЕК, к.е.н., доц.

Рощина Надія Василівна _____

Консультант з нормоконтролю:

Доцент кафедри ММСА, к.ф.-м.н.

Статкевич Віталій Михайлович _____

Рецензент:

Професор кафедри ШІ НН ІПСА, д.т.н., проф.

Данилов Валерій Якович _____

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

Студент _____

Київ – 2026 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 124 «Системний аналіз»
Освітня програма «Системний аналіз і управління»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Оксана ТИМОЩУК
«__» _____ 2026 р.

ЗАВДАННЯ
на дипломну роботу студенту
Анненкову Павлу Георгійовичу

1. Тема роботи «Система прийняття рішень на основі штучного інтелекту для розв'язання задачі вибору оптимального ходу у шашках», керівник роботи Канцедал Георгій Олегович, асистент кафедри ММСА, доктор філософії, затверджені наказом по університету від 27.05.2026 р. №1944-с.
2. Термін подання студентом роботи «__» червня 2026 р.
3. Вихідні дані до роботи: правила міжнародних шашок; ігрове поле розміром 10 x 10 клітин; двовимірний покроковий гра з двома гравцями; множина допустимих ходів, що визначається поточним станом ігрової позиції; алгоритми мінімакс, альфа-бета відсікання та метод Монте-Карло; евристична оцінка позиції; програмні засоби JavaScript, HTML і CSS.
4. Зміст роботи: проаналізувати предметну область та наявні підходи до побудови ігрового штучного інтелекту; формалізувати задачу вибору оптимального ходу у міжнародних шашках; розробити евристичну функцію оцінки позиції; реалізувати та дослідити випадковий алгоритм, мінімакс, альфа-бета відсікання і метод Монте-Карло; розробити єдину систему штучного інтелекту для автоматичного прийняття рішень; реалізувати правила гри, інтерфейс користувача та засоби фонових моделювання партій; провести експериментальне порівняння алгоритмів за результатами партій, часом пошуку та використаними обчислювальними ресурсами.
5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): презентація.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Функціонально-вартісний аналіз	Рощина Н.В., доц., к.е.н.		

7. Дата видачі завдання 26.02.2026

Календарний план

№ з/п	Назва етапів виконання бакалаврської дипломної роботи (БДР)	Термін виконання етапів роботи	Примітка
1	Затвердження теми БДР, опанування ДСТУ 3008:2015	03.09.2025 – 30.09.2025	Виконано
2	Аналіз актуальності теми	01.10.2025 – 30.10.2025	Виконано
3	Формулювання задачі дослідження	01.11.2025 – 30.11.2025	Виконано
4	Збір інформації, дослідження алгоритмів вирішення задачі	01.02.2026 – 30.02.2026	Виконано
5	Проектування алгоритмів і розробка програмного забезпечення	01.03.2026 – 30.03.2026	Виконано
6	Оформлення пояснювальної записки (ПЗ), перевірка правильності структури та оформлення	01.04.2026 – 06.05.2026	Виконано
7	Підготовка презентації для захисту	01.05.2026 – 06.05.2026	Виконано
8	Попередній захист дипломної роботи	09.06.2026	Виконано

Студент _____

Павло АННЄНКОВ

Керівник _____

Георгій КАНЦЕДАЛ

РЕФЕРАТ

Дипломна робота: 158 с., 11 табл., 25 рис., 1 дод., 22 джерела

ШТУЧНИЙ ІНТЕЛЕКТ, СИСТЕМА ПРИЙНЯТТЯ РІШЕНЬ, МІЖНАРОДНІ ШАШКИ, МІНІМАКС, АЛЬФА–БЕТА ВІДСІКАННЯ, МЕТОД МОНТЕ-КАРЛО, ЕВРИСТИЧНА ОЦІНКА.

Об'єкт дослідження – процес прийняття рішень під час вибору ходу у двовимірній покроковій грі з двома гравцями на прикладі міжнародних шашок.

Предмет дослідження – методи та алгоритми штучного інтелекту, які використовуються для аналізу ігрової позиції, оцінювання допустимих ходів і вибору найкращого продовження партії.

Мета роботи – розробити систему прийняття рішень на основі штучного інтелекту для вибору оптимального ходу у міжнародних шашках з урахуванням якості аналізу, часу пошуку та використаних обчислювальних ресурсів.

Актуальність роботи пов'язана з тим, що кількість можливих продовжень партії швидко зростає зі збільшенням глибини аналізу. Повний перебір ігрового дерева потребує значних обчислювальних ресурсів, тому для вибору ходу необхідно застосовувати пошукові алгоритми, евристичні оцінки та способи скорочення кількості досліджуваних позицій.

У роботі розглянуто правила міжнародних шашок і формальне подання ігрового стану. Реалізовано випадковий алгоритм, мінімакс, альфа-бета відсікання, метод Монте-Карло та евристичну функцію оцінки позиції. На їх основі розроблено єдину систему штучного інтелекту, яка автоматично визначає параметри аналізу позиції. Також реалізовано фонове моделювання серій партій та збирання статистики для порівняння якості й ресурсоемності методів.

ABSTRACT

Thesis: 158 p., 11 tabl., 25 fig., 1 app., 22 ref.

ARTIFICIAL INTELLIGENCE, DECISION-MAKING SYSTEM,
INTERNATIONAL DRAUGHTS, MINIMAX, ALPHA-BETA PRUNING,
MONTE CARLO TREE SEARCH, HEURISTIC EVALUATION

Object of research – the decision-making process for move selection in a two-dimensional turn-based two-player game, using international draughts as an example.

Subject of research is the set of artificial intelligence methods and algorithms used to analyze a game position, evaluate legal moves, and select the best continuation of a game.

Purpose of the work – to develop an artificial intelligence-based decision-making system for selecting the optimal move in international draughts, taking into account analysis quality, search time, and computational resources.

Relevance of the work is related to the rapid growth in the number of possible game continuations as the analysis depth increases. A complete traversal of the game tree requires considerable computational resources. Therefore, move selection requires search algorithms, heuristic evaluation, and methods for reducing the number of examined positions.

The thesis considers the rules of international draughts and a formal representation of the game state. A random algorithm, minimax, alpha-beta pruning, Monte Carlo Tree Search, and a heuristic position evaluation function were implemented. On this basis, we develop a unified artificial intelligence system that automatically determines position analysis parameters. Background simulation of game series and statistical data collection were also implemented to compare the quality and resource consumption of the methods.

ЗМІСТ

ВСТУП	8
РОЗДІЛ 1 АНАЛІЗ ЗАДАЧІ ВИБОРУ ОПТИМАЛЬНОГО ХОДУ У ШАШКАХ.....	10
1.1 Двовимірні покрокові ігри з двома гравцями як клас задач прийняття рішень	10
1.3 Формальна постановка задачі вибору оптимального ходу.....	15
1.4 Існуючі системи штучного інтелекту для шашок.....	18
1.5 Вимоги до системи прийняття рішень	19
1.6 Критерії оцінювання якості вибраного ходу та ефективності системи .	22
Висновки до розділу 1	24
РОЗДІЛ 2 МОДЕЛІ ТА МЕТОДИ ПРИЙНЯТТЯ РІШЕНЬ ДЛЯ ВИБОРУ ХОДУ У ШАШКАХ	26
2.1 Подання ігрового стану та множини допустимих ходів	26
2.2 Побудова дерева можливих продовжень партії.....	29
2.3 Евристична функція оцінювання ігрової позиції	33
2.3.1 Структура функції оцінювання	33
2.3.2 Вибір і обґрунтування вагових коефіцієнтів	34
2.3.3 Оцінювання матеріалу, мобільності та позиційних характеристик .	36
2.4 Алгоритми вибору ходу на основі пошуку	39
2.4.1 Алгоритм мінімакс.....	39
2.4.2 Алгоритм альфа-бета відсікання	42
2.4.3 Упорядкування ходів, таблиці транспозицій та пошук спокійних позицій	45
2.5 Метод пошуку Монте-Карло в дереві.....	49
2.5.1 Вибір та розширення вузлів дерева	50
2.5.2 Моделювання продовження партії.....	52
2.5.3 Зворотне поширення результатів	53
2.6 Адаптивний метод прийняття рішень єдиної системи штучного інтелекту	54

2.6.1 Оцінювання складності поточної позиції.....	55
2.6.2 Адаптивний вибір глибини та обмежень пошуку	56
2.6.3 Ітеративне поглиблення та вибір остаточного ходу	58
Висновки до розділу 2	59
РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ І АНАЛІЗ РЕЗУЛЬТАТІВ	61
3.1 Обґрунтування вибору технологій розробки	61
3.2 Архітектура та організація програмної системи.....	62
3.3 Розробка інтерфейсу користувача.....	64
3.4 Фонове моделювання партій та збирання статистики	69
3.5 Експериментальне дослідження розробленої системи	69
3.5.1 Методика та показники експериментального дослідження	69
3.5.2 Порівняння результатів ігрових алгоритмів	70
3.6.3 Аналіз часу пошуку та використаних обчислювальних ресурсів.....	72
3.6.4 Оцінювання ефективності єдиної системи штучного інтелекту	73
Висновки до розділу 3	74
РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ	75
4.1 Постановка задачі проєктування.....	76
4.2 Обґрунтування функцій програмного продукту.....	77
4.3 Обґрунтування системи параметрів програмного продукту	80
4.4 Аналіз експертного оцінювання параметрів	84
4.5 Аналіз рівня якості варіантів реалізації функцій.....	87
4.6 Економічний аналіз варіантів розробки ПП.....	89
4.7 Вибір кращого варіанта ПП за техніко-економічним рівнем.....	94
Висновки до розділу 4	95
ВИСНОВКИ.....	96
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ	99
ДОДАТОК А ЛІСТИНГ ПРОГРАМИ	102

ВСТУП

Шашки є поширеною настільною грою, у якій результат залежить від уміння оцінювати позицію, передбачати дії суперника та обирати найкращий хід. Незважаючи на відносно прості правила, зі збільшенням глибини аналізу кількість можливих продовжень партії швидко зростає. Тому вибір оптимального ходу у шашках є складною задачею прийняття рішень.

Актуальність роботи полягає у необхідності створення системи, яка може обирати обґрунтований хід без повного перегляду всіх можливих продовжень партії. Для цього застосовуються алгоритми пошуку, евристична оцінка позиції та статистичне моделювання. Важливим є не лише підвищення якості гри, але й зменшення часу аналізу та кількості використаних обчислювальних ресурсів.

Метою дипломної роботи є розроблення системи прийняття рішень на основі штучного інтелекту для розв'язання задачі вибору оптимального ходу у міжнародних шашках. У межах роботи під оптимальним розуміється допустимий хід, який отримав найкращу оцінку серед досліджених варіантів за наявних обмежень глибини та обчислювального бюджету.

Для досягнення поставленої мети необхідно проаналізувати правила міжнародних шашок та наявні підходи до побудови ігрового штучного інтелекту, формалізувати подання ігрового стану, розробити евристичну функцію оцінки позиції, реалізувати алгоритми мінімакс, альфа-бета відсікання та метод Монте-Карло. На основі цих методів необхідно створити єдину систему штучного інтелекту, яка автоматично визначатиме параметри аналізу та прийматиме рішення про вибір ходу.

Практичним результатом роботи є програмна система, написана мовою JavaScript, яка реалізує правила міжнародних шашок, алгоритми вибору ходу, графічний інтерфейс, фонове моделювання партій та збір статистичних даних.

РОЗДІЛ 1 АНАЛІЗ ЗАДАЧІ ВИБОРУ ОПТИМАЛЬНОГО ХОДУ У ШАШКАХ

1.1 Двовимірні покрокові ігри з двома гравцями як клас задач прийняття рішень

Двовимірна покрокова гра з двома гравцями є послідовністю станів, у якій учасники по черзі виконують допустимі дії. Кожен хід змінює поточний стан і впливає на можливі рішення обох сторін у майбутньому. Ігровий процес продовжується до досягнення кінцевого стану, що відповідає перемозі одного з гравців, поразці або нічій.

До цього класу належать шахи, шашки, реверсі, хрестики-нулики та інші настільні ігри. Їх спільною особливістю є наявність ігрового поля, визначеної множини фігур, правил переходу між станами та протилежних цілей гравців. Двовимірне подання поля дозволяє описати позицію за допомогою матриці, кожен елемент якої містить інформацію про відповідну клітину.

Таку гру можна формально подати у вигляді системи:

$$G = \langle S, A, T, P, U \rangle$$

у даній формулі G - формальна модель гри, S - множина можливих ігрових станів, $A(s)$ - множина допустимих дій у стані s , $T(s, a)$ - функція переходу до нового стану після виконання дії a , $P(s)$ - гравець, якому належить хід, $U(s)$ - корисність кінцевого стану.

Для гри з протилежними інтересами результат можна оцінювати значеннями 1, 0 та -1, які відповідають перемозі, нічій і поразці. У такому випадку виграш одного гравця одночасно означає програш другого:

$$U_1(s) = -U_2(s)$$

у даній формулі $U_1(s)$ - корисність стану s для першого гравця, $U_2(s)$ - корисність цього самого стану для другого гравця.

Отже, двовимірну покрокову гру можна розглядати як детерміновану гру з повною інформацією та нульовою сумою. Детермінованість означає, що результат ходу однозначно визначається правилами. Повна інформація полягає в тому, що обидва гравці бачать весь поточний стан поля і знають доступні дії. Випадкові події, приховані фігури або невідомі правила відсутні.

У кожному стані гравець повинен обрати одну дію з множини допустимих варіантів. Загальну задачу вибору ходу можна записати такою формулою:

$$a^* = \arg \max_{(a \in A(s))} Q(s, a)$$

у даній формулі a^* - вибраний хід, $A(s)$ - множина допустимих ходів у стані s , $Q(s, a)$ - оцінка результату виконання ходу a . Ця оцінка повинна враховувати не тільки безпосередню зміну позиції, але й можливі відповіді суперника. Тому прийняття рішення потребує аналізу певної послідовності майбутніх ходів.

Можливі продовження партії утворюють дерево ігрових станів. Коренем такого дерева є поточна позиція, ребрам відповідають допустимі ходи, а дочірнім вузлам - позиції після їх виконання. Рівні дерева по чергово відповідають ходам першого і другого гравців.

Кількість вузлів, які необхідно дослідити до глибини d , можна приблизно визначити за формулою:

$$N(d) = \sum_{i=0}^d b^i = \frac{b^{d+1} - 1}{b - 1}$$

у даній формулі $N(d)$ - загальна кількість вузлів дерева до глибини d , b - середня кількість допустимих ходів у позиції, d - глибина аналізу в півходах. Зі збільшенням глибини кількість станів зростає експоненційно. Наприклад, за

середнього коефіцієнта розгалуження $b = 8$ аналіз на глибину 10 півходів може потребувати перегляду понад одного мільярда вузлів.

Через таке зростання повний перебір можливих продовжень зазвичай неможливий. Система повинна обмежувати глибину аналізу, відкидати неперспективні варіанти та оцінювати позиції, для яких результат партії ще не визначений. Якість рішення при цьому залежить від алгоритму пошуку, точності оцінювальної функції та доступних обчислювальних ресурсів.

Таким чином, двовимірна покорова гра з двома гравцями є зручною моделлю задачі прийняття рішень у конкурентному середовищі. Система штучного інтелекту повинна сформувати допустимі альтернативи, спрогнозувати дії суперника, оцінити наслідки кожного варіанта та обрати хід, який забезпечує найкращий очікуваний результат.

1.2 Правила та особливості міжнародних шашок

Міжнародні шашки є логічною настільною грою для двох гравців, яка проводиться на квадратному полі розміром 10×10 клітин. Клітини поля по чергово мають світлий і темний колір, однак ігровими є лише темні клітини. Отже, із 100 клітин поля для розташування та переміщення фігур використовуються 50 [1].

На початку партії кожен гравець має 20 простих шашок. Чорні шашки розташовуються на клітинах від 1 до 20, а білі – на клітинах від 31 до 50. Центральні клітини від 21 до 30 залишаються вільними. Перший хід у партії виконують білі. Після цього гравці здійснюють ходи по чергово [1].

Проста шашка за відсутності взяття переміщується на одну вільну клітину по діагоналі вперед. Напрямок руху визначається кольором фігури. Білі шашки рухаються у напрямку початкового розташування чорних, а чорні - у протилежному напрямку. Рух назад без взяття для простої шашки заборонений.

Якщо проста шашка досягає останнього горизонтального ряду на стороні суперника та зупиняється на ньому, вона перетворюється на дамку. Дамка може переміщуватися по діагоналі вперед і назад через довільну кількість вільних клітин. Водночас дамка не може перестрибувати через власні фігури або через декілька фігур суперника без виконання послідовного взяття.

Взяття у міжнародних шашках є обов'язковим. Проста шашка може виконувати його як вперед, так і назад. Для цього вона перестрибує через шашку або дамку суперника, яка розташована на сусідній діагональній клітині, та зупиняється на вільній клітині безпосередньо за нею.

Дамка виконує взяття на більшій відстані. Якщо на її діагоналі розташована фігура суперника, а за нею є одна або декілька вільних клітин, дамка повинна перестрибнути через цю фігуру. Після взяття вона може зупинитися на будь-якій вільній клітині за взятою фігурою, якщо це не перешкоджає продовженню обов'язкової серії взяттів [1].

Якщо після першого взяття та сама фігура може взяти іншу фігуру суперника, хід повинен бути продовжений. У результаті один хід може складатися з декількох послідовних стрибків. Під час такого ходу дозволено повторно проходити через ту саму вільну клітину, але заборонено двічі перестрибувати через одну й ту саму фігуру суперника. Взяті фігури вилючаються з поля тільки після завершення всієї послідовності.

Особливістю міжнародних шашок є правило найбільшої кількості взятих фігур. Якщо гравець має декілька варіантів взяття, він повинен обрати маршрут, який дозволяє взяти максимальну кількість фігур суперника. Проста шашка і дамка під час підрахунку мають однакову вагу. Якщо декілька маршрутів забезпечують однакову найбільшу кількість взяттів, гравець може обрати будь-який із них [1].

Множину допустимих варіантів взяття можна визначити за формулою:

$$c_{(max)}(s) = \max_{(a \in C(s))} c(a)$$

у даній формулі $c_{(max)}(s)$ - найбільша можлива кількість взятих фігур у стані s , $C(s)$ - множина всіх маршрутів взяття, $c(a)$ - кількість фігур, взятих під час виконання ходу a .

Допустимими залишаються лише ті маршрути a , для яких виконується умова $c(a) = c_{(max)}(s)$.

Якщо множина $C(s)$ не є порожньою, звичайні ходи не можуть входити до множини допустимих дій. Спочатку система повинна знайти всі можливі маршрути взяття, визначити найбільшу кількість фігур, яку можна взяти, та відкинути коротші маршрути. Ця особливість ускладнює генерацію ходів, оскільки один допустимий хід може містити декілька проміжних переміщень.

Окреме правило стосується перетворення шашки на дамку під час серії взяттів. Якщо проста шашка проходить через останній ряд, але завершує хід на іншому рядку, вона залишається простою шашкою. Перетворення відбувається лише тоді, коли після завершення всього ходу фігура зупинилася на клітині останнього ряду [1].

Гравець перемагає, якщо суперник не має жодної фігури або не може виконати допустимий хід через блокування всіх своїх фігур. Умову перемоги гравця p можна подати за формулою:

$$Win(p, s) = [n_q(s) = 0] \vee [|A_q(s)| = 0], \quad q \neq p$$

у даній формулі $Win(p, s)$ - умова перемоги гравця p у стані s , q - гравець-суперник, $n_q(s)$ - кількість фігур суперника у поточній позиції, $A_q(s)$ - множина допустимих ходів суперника.

Нічия фіксується, якщо одна й та сама позиція з однаковою стороною, яка повинна виконати хід, виникла втретє. Також партія завершується нічиєю, якщо протягом 25 послідовних ходів кожного гравця переміщувалися лише дамки та не було виконано жодного взяття або ходу простою шашкою. Для окремих малочисельних закінчень офіційними правилами встановлено додаткові обмеження у 16 або 5 ходів [1].

Таким чином, міжнародні шашки мають чітко визначені правила переходу між ігровими станами, але формування допустимих ходів є складнішим, ніж звичайна перевірка сусідніх клітин. Система повинна враховувати обов'язковість взяття, пошук повних маршрутів багаторазового взяття, правило найбільшої кількості взятих фігур, особливості руху дамки та умови завершення партії. Правильна реалізація цих правил є необхідною умовою подальшого застосування алгоритмів штучного інтелекту.

1.3 Формальна постановка задачі вибору оптимального ходу

Задача вибору оптимального ходу полягає у визначенні такої допустимої дії, яка забезпечує найкращий результат для поточного гравця з урахуванням можливих відповідей суперника. Вхідними даними системи є поточна позиція на дошці, сторона, якій належить хід, історія партії та доступні обчислювальні ресурси. Результатом роботи системи має бути один хід із множини варіантів, дозволених правилами міжнародних шашок.

Ігровий стан можна подати за формулою (1.2):

$$s = (X, p, H, k) \quad (1.2)$$

у даній формулі s - поточний ігровий стан, X - матриця розташування фігур на дошці, p - сторона, якій належить хід, H - історія попередніх станів і ходів, k - набір лічильників, необхідних для перевірки умов нічиєї.

Матриця X має розмір 10×10 клітин. Її елементи містять інформацію про відсутність фігури, просту шашку або дамку відповідного кольору. Історія H необхідна для виявлення повторення позицій, а лічильники k використовуються для контролю кількості ходів без взяття та інших умов завершення партії.

Для кожного стану s система формує множину допустимих ходів $A(s)$. Під час її побудови спочатку перевіряється наявність взяттів. Якщо вони існують, звичайні переміщення не розглядаються, а допустимими

залишаються лише маршрути з найбільшою кількістю взятих фігур. Якщо взяття відсутні, до множини $A(s)$ включаються всі дозволені правилами переміщення простих шашок і дамок.

Після виконання ходу відбувається перехід до нового стану. Цей процес описується за формулою (1.3):

$$s' = T(s, a), a \in A(s) \quad (1.3)$$

у даній формулі s' - стан гри після виконання ходу, $T(s, a)$ - функція переходу, s - початковий стан, a - вибраний допустимий хід, $A(s)$ - множина допустимих ходів у стані s .

Функція переходу повинна змінити розташування фігур, вилучити взяті шашки, за необхідності перетворити просту шашку на дамку, передати право ходу іншому гравцю та оновити історію партії. Отриманий стан також перевіряється на наявність перемоги, поразки або нічиєї.

У кінцевому стані результат можна оцінити значенням 1 для перемоги, 0 для нічиєї та -1 для поразки. Для проміжних позицій точний результат партії ще невідомий, тому використовується оцінювальна функція. Вона враховує матеріальну перевагу, кількість дамок, мобільність фігур, можливі взяття, позиційні характеристики та інші показники.

Теоретично оптимальним є хід, який приводить до стану з найбільшим точним ігровим значенням. Його можна визначити за формулою:

$$a_{(opt)} = \arg \max_{(a \in A(s))} V(T(s, a))$$

у даній формулі $a_{(opt)}$ - теоретично оптимальний хід, $A(s)$ - множина допустимих ходів, $T(s, a)$ - стан після виконання ходу a , $V(s)$ - точне ігрове значення стану.

Для обчислення $V(s)$ необхідно дослідити гру до кінцевих станів за умови оптимальної поведінки обох гравців. У більшості практичних позицій це потребує перегляду надто великої кількості варіантів. Тому розроблена система працює з обмеженим обчислювальним бюджетом.

Набір обмежень аналізу можна подати за формулою:

$$R = (d, t_{(max)}, n_{(max)}, m_{(max)})$$

у даній формулі R - доступний обчислювальний бюджет, d - максимальна глибина пошуку, $t_{(max)}$ - допустимий час аналізу, $n_{(max)}$ - максимальна кількість досліджених вузлів, $m_{(max)}$ - максимальна кількість симуляцій.

Не всі складові бюджету використовуються одночасно. Для мінімаксу та альфа-бета відсікання основними параметрами є глибина, час і кількість вузлів. Для методу Монте-Карло головним обмеженням є кількість симуляцій або час їх виконання.

За обмеженого бюджету система використовує наближену оцінку $V_R(s)$. Практичний вибір ходу здійснюється за формулою:

$$a_R^* = \arg \max_{(a \in A(s))} V_R(T(s, a))$$

у даній формулі a_R^* - найкращий хід, знайдений за обчислювального бюджету R , $V_R(s)$ - наближена оцінка стану, отримана за допомогою вибраного алгоритму.

Отже, у роботі під оптимальним ходом розуміється не гарантовано найкращий хід для повністю розв'язаної гри, а найкращий варіант серед досліджених продовжень за доступного обчислювального бюджету. Зі збільшенням глибини пошуку або кількості симуляцій точність рішення може зростати, однак одночасно збільшуються час аналізу та використання ресурсів.

Формальна постановка задачі полягає у побудові системи, яка для стану s формує множину $A(s)$, оцінює наслідки допустимих ходів, ураховує можливі відповіді суперника та повертає хід a_R^* . При цьому система повинна забезпечити правильне виконання правил міжнародних шашок і досягти прийняттого співвідношення між якістю рішення та витратами обчислювальних ресурсів.

1.4 Існуючі системи штучного інтелекту для шашок

Однією з перших програм для гри в шашки була система Артура Самуеля, розроблена у 1950-х роках для англійських шашок 8×8 [2]. Вона використовувала пошук можливих продовжень та оцінювальну функцію, яка враховувала матеріальну перевагу, мобільність та інші характеристики позиції.

Особливістю програми була здатність покращувати оцінювання на основі зіграних партій. Система запам'ятовувала раніше розглянуті позиції та змінювала ваги окремих характеристик. Для отримання досвіду програма могла грати проти попередніх версій самої себе. Тому її вважають одним із перших практичних прикладів машинного навчання.

Наступним важливим етапом стала система Chinook, створена командою Джонатана Шеффера для англійських шашок [3]. Вона використовувала ітеративний альфа-бета пошук, таблиці транспозицій, упорядкування ходів, дебютну базу та бази закінчень.

Бази закінчень Chinook містили точні результати для позицій із вісьмома або меншою кількістю фігур. Загалом у них було збережено понад 443 млрд позицій [4]. Коли під час пошуку система досягала такого стану, вона могла одразу визначити його результат без подальшого розгортання дерева.

У 1994 році Chinook став чемпіоном у спеціальній категорії змагання людини проти комп'ютера. У 2007 році розробники довели, що за оптимальної гри обох сторін англійські шашки завершуються нічиєю [5].

Для міжнародних шашок були створені системи Truus, Flits, Buggy, Kingsrow, Moby Dam і Scan. Більшість із них поєднує пошук у дереві гри, евристичне оцінювання, бази дебютів і бази закінчень. Відмінності між цими системами полягають у способах представлення дошки, функціях оцінювання та методах скорочення кількості досліджуваних позицій.

Система Buggy застосовувалася у дослідженнях машинного навчання для міжнародних шашок. Для налаштування оцінювальної функції використовувалися нейронні мережі, метод часових різниць і база Turbo Dabase 3, яка містила понад 260 тис. партій [6]. Мережа навчалася оцінювати позиції за розташуванням фігур, матеріальною перевагою та характерними стратегічними структурами.

Однією з найвідоміших сучасних систем для міжнародних шашок є Scan, розроблена Фаб'єном Летузе [7]. Це відкрита програма, яка підтримує міжнародні шашки 10×10 та декілька інших варіантів гри. Scan не має власного повноцінного графічного інтерфейсу, але може взаємодіяти із зовнішніми програмами.

Scan використовує бітове представлення дошки, оптимізований пошук, паралельне виконання обчислень, дебютну базу та бази закінчень для позицій із шістьма фігурами. Його оцінювальна функція враховує матеріальне співвідношення, розташування фігур, структуру позиції та фазу партії.

1.5 Вимоги до системи прийняття рішень

Основною функціональною вимогою є підтримка декількох способів прийняття рішень. Користувач повинен мати можливість окремо вибрати алгоритм для білих і чорних. Система повинна підтримувати випадковий алгоритм як базовий засіб порівняння, мінімакс, альфа-бета відсікання, метод Монте-Карло та єдину систему штучного інтелекту.

Для мінімаксу й альфа-бета відсікання користувач повинен мати можливість встановлювати глибину аналізу окремо для кожної сторони. Для методу Монте-Карло необхідно передбачити налаштування кількості симуляцій. Якщо вибрані параметри можуть призвести до тривалого пошуку, інтерфейс повинен показувати відповідне попередження.

Єдина система штучного інтелекту повинна працювати автоматично. Вона має самостійно оцінювати складність позиції, визначати необхідні параметри пошуку та обмежувати використання обчислювальних ресурсів. Користувач не повинен вручну задавати глибину або кількість симуляцій для цього режиму.

Система повинна підтримувати три режими проведення партії:

- 1) людина проти людини;
- 2) людина проти алгоритму;
- 3) алгоритм проти алгоритму.

Перед початком партії користувач повинен обрати режим гри та алгоритми для відповідних сторін. Застосування вибраних параметрів має відбуватися після натискання кнопки початку партії. Під час активної партії система не повинна змінювати алгоритм або його налаштування без явного перезапуску гри.

Для контролю роботи алгоритмів необхідно відображати евристичну оцінку поточної позиції. Окремо повинна бути доступна поглиблена оцінка єдиної системи штучного інтелекту. Для неї потрібно показувати глибину аналізу, найкращий знайдений варіант, кількість досліджених вузлів і витрачений час.

Система повинна збирати інформацію про використані обчислювальні ресурси. Для кожного ходу необхідно фіксувати тривалість пошуку, кількість досліджених вузлів, кількість відсікань, частоту використання таблиці транспозицій та обсяг збережених позицій. Для методу Монте-Карло додатково потрібно зберігати кількість виконаних симуляцій та їх середню довжину.

Під час партії користувач повинен бачити час виконання поточного ходу та загальну тривалість гри. Журнал ходів має містити послідовність виконаних дій, назву використаного алгоритму та основні показники пошуку. Це дає можливість перевірити, яким способом система отримала конкретне рішення.

Окремою вимогою є підтримка фонового моделювання. Користувач повинен мати можливість задати кількість партій, алгоритми обох сторін та їхні параметри. Моделювання має виконуватися без графічного відтворення кожного ходу, щоб зменшити витрати часу на оновлення інтерфейсу.

Після завершення серії партій система повинна формувати статистичний звіт. У ньому необхідно наводити кількість перемог кожного алгоритму, кількість нічий, середню тривалість партії, середню кількість півходів, середній час прийняття одного рішення та середню кількість досліджених вузлів або симуляцій.

Історія проведених партій повинна зберігатися між запусками програми. Користувач повинен мати можливість переглядати загальну статистику та очищувати накопичені результати. Видалення історії має виконуватися лише після окремої команди користувача.

До нефункціональних вимог належать коректність, зрозумілість інтерфейсу та стійкість до помилкових параметрів. Система не повинна дозволяти виконання недопустимого ходу, запуск пошуку з некоректною глибиною або від'ємною кількістю симуляцій. Помилка одного алгоритму не повинна призводити до пошкодження стану партії або збереженої статистики.

Програмна система має працювати у сучасному браузері без встановлення додаткового серверного програмного забезпечення. Її компоненти повинні бути розділені за призначенням: правила гри, алгоритми прийняття рішень, керування партією, інтерфейс і модуль фонового моделювання. Така структура спрощує перевірку алгоритмів та подальше розширення системи.

1.6 Критерії оцінювання якості вибраного ходу та ефективності системи

Для оцінювання розробленої системи недостатньо враховувати лише результат окремої партії. Перемога або поразка залежить від послідовності багатьох рішень, кольору фігур, дій суперника та складності позицій. Тому оцінювання доцільно виконувати за декількома критеріями, які характеризують правильність, якість і ресурсоемність прийнятих рішень.

Першим обов'язковим критерієм є допустимість вибраного ходу. Система повинна повертати лише ходи, які відповідають правилам міжнародних шашок. Цей критерій є необхідною умовою працездатності, але не визначає стратегічну якість рішення, оскільки серед допустимих ходів можуть бути як сильні, так і явно невігідні варіанти.

Безпосередньо визначити теоретично оптимальний хід для довільної позиції складно, оскільки для цього необхідно розглянути всі можливі продовження до завершення партії. Тому якість окремого рішення можна наближено оцінювати шляхом порівняння з результатом поглибленого контрольного аналізу.

Втрату якості вибраного ходу можна визначити за формулою:

$$L(s, a) = V_{(ref)}(s, a_{(ref)}) - V_{(ref)}(s, a)$$

у даній формулі $L(s, a)$ - втрата якості ходу a у стані s , $V_{(ref)}(s, a)$ - оцінка ходу за результатами поглибленого контрольного аналізу, $a_{(ref)}$ - найкращий хід, знайдений під час такого аналізу. Чим ближче значення $L(s, a)$ до нуля, тим менше вибране рішення відрізняється від контрольного.

Такий критерій не є абсолютним доказом оптимальності, оскільки контрольний аналіз також має обмежену глибину. Проте він дозволяє порівнювати декілька алгоритмів на однаковому наборі позицій.

Основним практичним критерієм якості є результат серії партій. Проведення лише однієї гри не дає достатньо надійної оцінки через можливий

вплив окремого невдалого рішення. Для кожного порівняння необхідно проводити серію партій зі зміною кольорів сторін.

Ігровий результат алгоритму визначається за формулою (1.1):

$$Q = \frac{W+0.5D}{N} \quad (1.1)$$

у формулі (1.1) Q - частка набраних очок, W - кількість перемог, D - кількість нічий, N - загальна кількість зіграних партій. Значення Q змінюється від 0 до 1. Значення 0,5 відповідає рівному результату двох алгоритмів, а значення, більше за 0,5, показує перевагу досліджуваного алгоритму.

Окремо необхідно враховувати кількість перемог, поразок і нічий. Однакова частка очок може бути отримана за різного характеру гри. Наприклад, алгоритм може часто перемагати та програвати або переважно завершувати партії нічиєю. Тому значення Q не повинно використовуватися без аналізу початкових результатів.

Наступним критерієм є середній час прийняття одного рішення. Він визначається за формулою:

$$t_{(avg)} = \frac{1}{M} T_{(search)}$$

у даній формулі $t_{(avg)}$ - середній час пошуку одного ходу, $T_{(search)}$ - сумарний час, витрачений алгоритмом на аналіз, M - кількість виконаних алгоритмом ходів.

Цей показник дозволяє оцінити практичну швидкість системи. Менший час не завжди означає кращий результат, оскільки швидкий алгоритм може виконувати недостатньо глибокий аналіз. Тому час пошуку необхідно розглядати разом з ігровим результатом.

Для пошукових алгоритмів додатково враховується кількість досліджених вузлів. Середню кількість вузлів на один хід можна обчислити за формулою:

$$n_{(avg)} = \frac{1}{M} N_{(nodes)}$$

у даній формулі $n_{(avg)}$ - середня кількість досліджених вузлів на один хід, $N_{(nodes)}$ - загальна кількість вузлів, опрацьованих алгоритмом.

Для методу Монте-Карло замість кількості вузлів основним показником є кількість виконаних симуляцій. Вузол пошукового дерева та одна симуляція мають різну обчислювальну вартість, тому безпосередньо порівнювати ці значення некоректно. Порівняння різних типів алгоритмів доцільно виконувати за часом пошуку та результатами партій.

Продуктивність пошуку можна додатково характеризувати кількістю опрацьованих вузлів за секунду. Вона визначається за формулою:

$$R_{(nodes)} = N_{(nodes)} / T_{(search)}$$

у даній формулі $R_{(nodes)}$ - швидкість опрацювання вузлів, $N_{(nodes)}$ - кількість досліджених вузлів, $T_{(search)}$ - тривалість пошуку в секундах. Для методу Монте-Карло аналогічно може визначатися кількість симуляцій за секунду.

Таким чином, якість системи оцінюється за допустимістю ходів, відхиленням від результатів контрольного аналізу та підсумками серій партій. Обчислювальна ефективність визначається за часом пошуку, кількістю досліджених вузлів або виконаних симуляцій. Конкретні налаштування експериментів і отримані числові результати будуть наведені під час дослідження програмної системи.

Висновки до розділу 1

У першому розділі розглянуто двовимірні покрокові ігри з двома гравцями як клас задач прийняття рішень. Наведено формальне подання гри та визначено особливості міжнародних шашок, які впливають на формування множини допустимих ходів.

Сформульовано задачу вибору оптимального ходу з урахуванням обмеженого обчислювального бюджету. Також розглянуто наявні системи

штучного інтелекту для шашок, зокрема програму Самуеля, Chinook, Buggy та Scan, а також неймережевий підхід.

Визначено функціональні та нефункціональні вимоги до розроблюваної системи. Сформовано критерії оцінювання якості й ефективності, до яких належать результати серій партій, час пошуку, кількість досліджених вузлів і виконаних симуляцій.

РОЗДІЛ 2 МОДЕЛІ ТА МЕТОДИ ПРИЙНЯТТЯ РІШЕНЬ ДЛЯ ВИБОРУ ХОДУ У ШАШКАХ

2.1 Подання ігрового стану та множини допустимих ходів

Для застосування алгоритмів прийняття рішень ігрова позиція повинна мати формальне подання, придатне для зберігання, копіювання, порівняння та побудови наступних станів. Загальну структуру стану було визначено у формулі (1.2). У цьому підрозділі детальніше розглядається подання дошки та допустимих ходів.

Основною складовою стану є ігрове поле розміром 10×10 клітин. Для кожної клітини необхідно визначити, чи є вона порожньою, чи містить просту шашку або дамку відповідного кольору. Множину можливих значень клітини можна подати за формулою:

$$D = (0, w_m, w_k, b_m, b_k) \quad (2.1)$$

у формулі (2.1) D - множина можливих значень клітини, 0 - порожня клітина, w_m - біла проста шашка, w_k - біла дамка, b_m - чорна проста шашка, b_k - чорна дамка.

Розташування фігур подається матрицею за формулою:

$$X = [x_{(r,c)}]_{(10 \times 10)}, x_{(r,c)} \in D$$

у даній формулі X - матриця ігрового поля, $x_{(r,c)}$ - значення клітини у рядку r і стовпці c , D - множина значень, визначена у формулі (2.1).

Фігури розташовуються лише на темних клітинах. Використання матричного подання спрощує перевірку меж поля, визначення сусідніх клітин і побудову діагональних напрямків руху.

Крім розташування фігур, стан містить сторону, якій належить хід. Це необхідно, оскільки однакове розташування фігур за різної черговості ходу є

різними ігровими станами. Також зберігається історія партії та значення лічильників, необхідних для перевірки умов нічиєї.

Для відображення позицій людини використовується нумерація 50 активних клітин числами від 1 до 50. Якщо рядки та стовпці матриці нумеруються від нуля, номер активної клітини визначається за формулою:

$$n(r, c) = 5r + \lfloor c/2 \rfloor + 1$$

у даній формулі $n(r, c)$ - номер активної клітини, r - номер рядка, c - номер стовпця. Формула застосовується лише до темних клітин ігрового поля.

Для аналізу позиції недостатньо знати тільки початкову та кінцеву клітини ходу. У міжнародних шашках один хід може містити послідовне взяття декількох фігур і проходити через декілька проміжних клітин. Тому хід доцільно подавати як маршрут разом із переліком взятих фігур.

Структуру ходу можна записати за формулою:

$$a = (P, R), P = (q_0, q_1, \dots, q_m), R = (r_1, r_2, \dots, r_l)$$

у даній формулі a - хід, P - послідовність клітин маршруту, q_0 - початкова клітина, q_m - кінцева клітина, R - послідовність взятих фігур, r_l - окрема взята фігура, m - кількість переміщень у межах ходу, l - кількість взятих фігур.

Для звичайного переміщення послідовність R є порожньою, а маршрут складається з початкової та кінцевої клітин. Під час багаторазового взяття маршрут містить усі проміжні клітини зупинки. Таке подання дозволяє розглядати всю послідовність взяттів як одну завершену дію.

Кількість фігур, взятих під час виконання ходу, визначається за формулою:

$$k(a) = |R| = l$$

у даній формулі $k(a)$ - кількість фігур, взятих під час ходу a , R - послідовність взятих фігур, l - довжина цієї послідовності.

Для поточного стану окремо формуються множина звичайних переміщень $M(s)$ та множина повних маршрутів взяття $K(s)$. Важливо, що

часткова послідовність взяттів не вважається допустимим ходом, якщо з останньої досягнутої клітини можна продовжити взяття. Тому до $K(s)$ включаються лише завершені маршрути.

Якщо існує декілька маршрутів взяття, відповідно до правил міжнародних шашок допустимими є лише маршрути з максимальною кількістю взятих фігур. Належність ходу до такої множини визначається за формулою:

$$a \in K_{(max)}(s) \Leftrightarrow a \in K(s) \wedge k(a) = k_{(max)}(s)$$

у даній формулі $K_{(max)}(s)$ - множина маршрутів із максимальною кількістю взяттів, $K(s)$ - множина всіх завершених маршрутів взяття, $k(a)$ - кількість взятих фігур під час ходу a , $k_{(max)}(s)$ - найбільша доступна кількість взяттів у стані s .

Множина допустимих ходів залежить від наявності взяття. Якщо хоча б один маршрут взяття існує, звичайні переміщення не можуть бути вибрані. У такому випадку множина допустимих ходів визначається за формулою:

$$A(s) = K_{(max)}(s), K(s) \neq \emptyset$$

у даній формулі $A(s)$ - множина допустимих ходів у стані s , $K_{(max)}(s)$ - множина маршрутів із максимальною кількістю взятих фігур, $K(s)$ - множина всіх маршрутів взяття.

Якщо взяття відсутні, множина допустимих дій визначається за формулою:

$$A(s) = M(s), K(s) = \emptyset$$

у даній формулі $M(s)$ - множина звичайних переміщень.

Формування $A(s)$ виконується у визначеній послідовності:

- 1) визначаються всі фігури сторони, якій належить хід;
- 2) для кожної фігури будуються всі завершені маршрути взяття;
- 3) серед знайдених маршрутів залишаються варіанти з найбільшою кількістю взятих фігур;

- 4) якщо взяття відсутні, формуються звичайні ходи простих шашок і дамок.

Один і той самий початковий і кінцевий пункт не завжди однозначно визначає хід. Дамка може виконати різні маршрути взяття та завершити їх на однаковій клітині. Тому для розрізнення ходів необхідно враховувати весь маршрут і перелік взятих фігур.

Після вибору ходу формується наступний стан відповідно до функції переходу, визначеної у формулі (1.3). Фігура переміщується до кінцевої клітини, взяті фігури вилучаються, а проста шашка за необхідності перетворюється на дамку. Після цього змінюється сторона, якій належить хід, та оновлюється інформація про партію.

Запропоноване подання містить усі дані, необхідні для однозначного відтворення позиції та виконання ходу. Матриця X визначає розташування фігур, додаткові складові стану забезпечують контроль черговості й умов завершення гри, а множина $A(s)$ містить лише повні ходи, які відповідають правилам міжнародних шашок. Це подання використовується як основа для побудови дерева можливих продовжень партії.

2.2 Побудова дерева можливих продовжень партії

Для аналізу наслідків допустимих ходів поточний стан гри розглядається як початкова вершина дерева. Кожне ребро відповідає одному допустимому ходу, а дочірня вершина містить стан, отриманий після його виконання. Процес повторюється для наступного гравця, унаслідок чого формується множина можливих продовжень партії.

На рисунку 2.1 показано загальну структуру дерева для двох послідовних півходів.

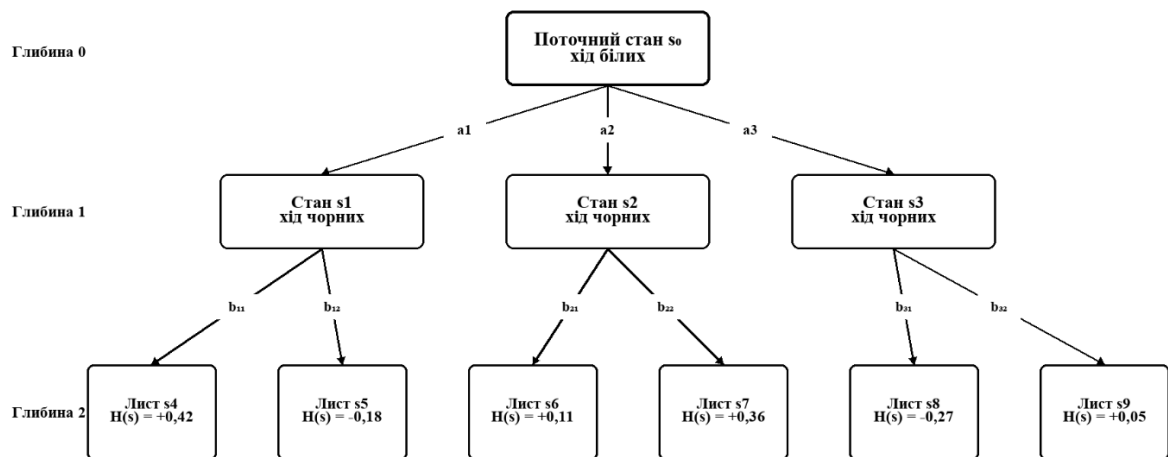


Рисунок 2.1 – Загальна структура дерева для двох послідовних півходів

Коренем дерева є поточний стан s_0 . На першому рівні розташовуються позиції, які можуть виникнути після ходу поточного гравця. На другому рівні знаходяться стани після можливих відповідей суперника. Таким чином, парні та непарні рівні по чергово відповідають рішенням різних сторін.

Глибина вершини визначається кількістю півходів від кореневого стану. Для кореня вона дорівнює нулю, а глибина дочірньої вершини визначається за формулою:

$$d(v) = d(u) + 1$$

У даній формулі $d(v)$ - глибина дочірньої вершини v , $d(u)$ - глибина її батьківської вершини u .

Один рівень дерева відповідає одному півходу, тобто дії лише однієї сторони. Наприклад, глибина $d = 4$ означає, що система розглядає два ходи поточного гравця та дві можливі відповіді суперника.

Побудова дерева виконується послідовно:

- 1) для поточного стану формується множина допустимих ходів $A(s)$;
- 2) кожен допустимий хід застосовується до окремої копії стану;
- 3) отриманий стан додається до дерева як дочірня вершина;
- 4) для нової вершини визначаються допустимі ходи іншої сторони;

5) розгортання продовжується до досягнення кінцевого стану або встановленої глибини.

Кількість дочірніх вершин залежить від кількості допустимих ходів у відповідній позиції. Цей показник називається коефіцієнтом розгалуження. Для різних станів його значення може суттєво відрізнятися. У позиціях з обов'язковим взяттям кількість доступних варіантів часто є невеликою, тоді як за наявності декількох дамок дерево може мати значне розгалуження.

Якщо середній коефіцієнт розгалуження дорівнює b , наближену кількість вершин на рівні d можна визначити за формулою (2.2):

$$N_d \approx b^d \quad (2.2)$$

у даній формулі N_d - кількість вершин на глибині d , b - середній коефіцієнт розгалуження, d - глибина дерева у півходах.

Загальна кількість вершин від кореня до заданої глибини визначається за формулою суми геометричної прогресії

$$N(d) = 1 + b + b^2 + \dots + b^d = \frac{b^{(d+1)} - 1}{b - 1}$$

у даній формулі $N(d)$ - загальна кількість вершин до глибини d включно, інші значення визначаються у формулі (2.2)

Наприклад, за середньої кількості 8 допустимих ходів аналіз до глибини 6 може потребувати побудови понад 299 тис. вершин. Збільшення глибини лише на один рівень підвищує кількість станів приблизно у b разів. Через це повне розгортання дерева до завершення партії у більшості позицій практично неможливе.

Для контролю обсягу обчислень вводиться максимальна глибина пошуку $d_{(max)}$. Вершини на цій глибині вважаються граничними, навіть якщо відповідні позиції не є кінцевими. Принцип обмеження дерева показано на рисунку 2.2.

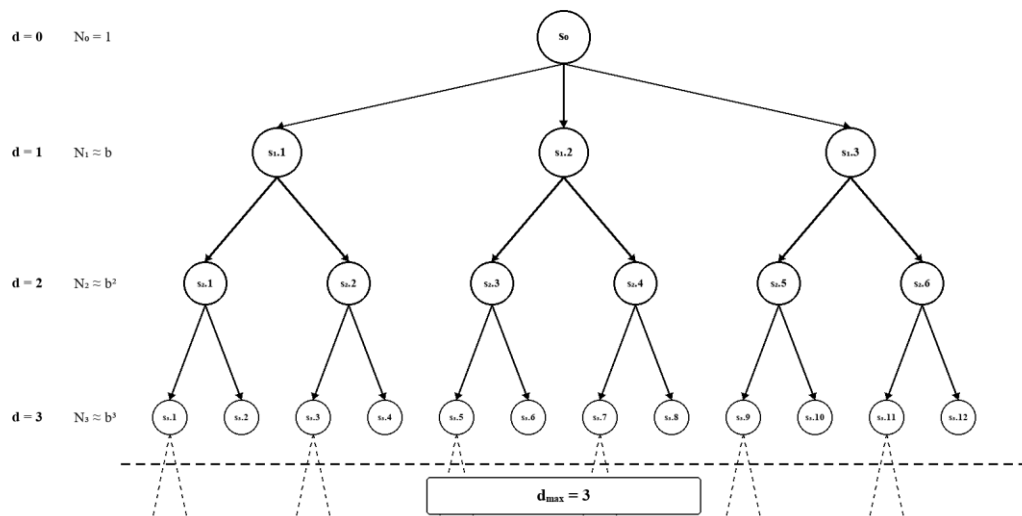


Рисунок 2.2 – Принцип обмеження дерева

Після досягнення глибини $d_{(max)}$ подальші продовження не будуються. Якщо гранична вершина відповідає перемозі, поразці або нічій, їй надається точне кінцеве значення. В іншому випадку використовується евристична функція оцінювання позиції.

Умову припинення розгортання вершини можна записати за формулою:

$$Stop(s, d) = [d = d_{(max)}] \vee Terminal(s)$$

у даній формулі $Stop(s, d)$ - умова припинення розгортання стану s на глибині d , $d_{(max)}$ - максимальна глибина пошуку, $Terminal(s)$ - ознака кінцевого стану.

Листовими вершинами дерева є кінцеві або граничні позиції. Для кожної з них обчислюється числова оцінка, яка характеризує вигідність стану для вибраної сторони. Надалі ці значення використовуються алгоритмами прийняття рішень для порівняння початкових ходів.

Під час побудови дерева одна й та сама позиція може виникнути після різних послідовностей ходів. У такому випадку структура фактично набуває властивостей графа. Повторне дослідження однакових станів збільшує час

пошуку, тому в подальших алгоритмах для їх виявлення можуть застосовуватися таблиці транспозицій.

2.3 Евристична функція оцінювання ігрової позиції

Пошукові алгоритми не завжди можуть розгорнути дерево до завершення партії. Після досягнення максимальної глибини необхідно визначити, наскільки вигідною є отримана позиція для кожної сторони. Для цього використовується евристична функція оцінювання.

Функція аналізує один стан без побудови подальших продовжень. Вона не враховує глибину пошуку, а визначає статичну оцінку за характеристиками поточної позиції. Глибина враховується пошуковим алгоритмом, який передає функції позиції з граничного рівня дерева.

Додатне значення оцінки означає перевагу білих, від'ємне - перевагу чорних, а значення, близьке до нуля, відповідає приблизно рівній позиції. Оцінка не є точним прогнозом результату партії, оскільки ґрунтується на обмеженому наборі характеристик.

2.3.1 Структура функції оцінювання

Для визначення загальної оцінки використовується сума декількох складових. Структуру евристичної функції відносно білих задано формулою:

$$H_W(s) = M(s) + P(s) + C(s) + R(s) + K(s) + Z(s)$$

у даній формулі $H_W(s)$ - оцінка стану s відносно білих, $M(s)$ - матеріальна складова, $P(s)$ - оцінка просування простих шашок, $C(s)$ - контроль центральної частини поля, $R(s)$ - мобільність, $K(s)$ - потенціал взяття, $Z(s)$ - оцінка кінцевого стану.

Кожна складова обчислюється як різниця між показниками білих і чорних. Тому перевага білих збільшує загальну оцінку, а перевага чорних зменшує її.

Для використання функції відносно сторони, яка виконує аналіз, застосовується формула:

$$H_p(s) = \sigma_p \cdot H_W(s)$$

у даній формулі $H_p(s)$ - оцінка стану s відносно гравця p , σ_p - коефіцієнт сторони. Для білих $\sigma_p = 1$, а для чорних $\sigma_p = -1$.

Завдяки цьому одна функція може використовуватися обома гравцями. Наприклад, значення $H_W(s) = 250$ означає перевагу білих. Для чорних оцінка тієї самої позиції становитиме -250.

Складові функції мають різні одиниці вимірювання. Матеріал визначається кількістю фігур, мобільність - кількістю допустимих ходів, а просування - відстанню до останнього ряду. Для об'єднання цих показників використовуються вагові коефіцієнти.

Функція не містить випадкових складових. Для однакової позиції та вибраної сторони вона завжди повертає однакове значення. Це дозволяє порівнювати результати різних пошукових алгоритмів за однакових умов.

2.3.2 Вибір і обґрунтування вагових коефіцієнтів

Вагові коефіцієнти визначають вплив окремих характеристик на загальну оцінку. У розробленій системі вони обиралися вручну без використання машинного навчання. Основою для вибору були правила міжнародних шашок, відносна важливість характеристик і перевірка поведінки системи на тестових позиціях та під час контрольних партій.

За базову одиницю прийнято вартість простої шашки, яка дорівнює 100 умовним балам. Інші коефіцієнти встановлено відносно цього значення:

- 1) проста шашка - 100 балів;
- 2) дамка - 320 балів;
- 3) просування на один ряд - 7 балів;
- 4) одна одиниця центральності - 6 балів;
- 5) один додатковий допустимий хід - 5 балів;
- 6) одна додаткова фігура у максимальному маршруті взяття - 34 бали;
- 7) перемога або поразка - 100000 балів.

Вартість дамки у 3,2 раза перевищує вартість простої шашки. Такий коефіцієнт пояснюється тим, що дамка може рухатися на значну відстань у всіх діагональних напрямках, має більшу мобільність і може виконувати дальні взяття. Водночас її вартість не встановлено надто високою, оскільки декілька правильно розташованих простих шашок можуть обмежити рух дамки.

Коефіцієнт просування дорівнює 7 балам за один ряд. Максимальний позиційний бонус простої шашки за просування становить 63 бали, що менше її матеріальної вартості. Отже, просунута шашка вважається перспективнішою, але її втрата не може бути компенсована лише положенням поблизу останнього ряду.

Коефіцієнт контролю центру дорівнює 6. Максимальна центральність фігури становить приблизно 4 одиниці, тому звичайна шашка може отримати до 24 позиційних балів. Для дамки цей показник додатково множиться на 1.2, оскільки центральне положення надає їй доступ до більшої кількості діагоналей.

Мобільність має вагу 5 балів за один додатковий допустимий хід. Наприклад, перевага у 10 ходів додає 50 балів, що відповідає половині вартості простої шашки. Такий коефіцієнт заохочує активну позицію, але не дозволяє мобільності переважати значну матеріальну перевагу.

Потенціал взяття має вагу 34 бали за кожну додаткову фігуру у максимальному доступному маршруті. Цей показник має більшу вагу, ніж

мобільність або контроль центру, оскільки взяття є обов'язковим і може одразу змінити матеріальне співвідношення.

Значення кінцевої позиції дорівнює 100000 балів. Воно значно перевищує суму всіх не кінцевих складових. Завдяки цьому алгоритм завжди віддає перевагу гарантованій перемозі над позиційною або матеріальною перевагою.

Під час вибору коефіцієнтів дотримувалася така ієрархія:

кінцевий результат > матеріал > взяття > просування > центр > мобільність

Загальне значення окремої складової залежить від кількості фігур і стану позиції. Ієрархія визначає переважно вагу однієї одиниці відповідного показника.

Обрані коефіцієнти не претендують на математично оптимальні значення. Вони забезпечують зрозумілу поведінку системи та зберігають перевагу матеріальних і тактичних факторів над другорядними позиційними показниками. Подальше вдосконалення може передбачати автоматичне налаштування ваг на основі результатів великої кількості партій.

2.3.3 Оцінювання матеріалу, мобільності та позиційних характеристик

Матеріальна складова враховує кількість простих шашок і дамок кожної сторони. Вона обчислюється за формулою:

$$M(s) = 100 \cdot (n_W^m - n_B^m) + 320 \cdot (n_W^k - n_B^k)$$

у даній формулі $M(s)$ - матеріальна оцінка стану s , n_W^m і n_B^m - кількість простих шашок білих і чорних, n_W^k і n_B^k - кількість дамок білих і чорних.

Наприклад, якщо білі мають одну додаткову просту шашку, матеріальна оцінка становить +100. Якщо чорні мають дамку замість простої шашки за

однакової загальної кількості фігур, їхня перевага за цим показником становить 220 балів.

Просування враховується лише для простих шашок. Чим ближче фігура перебуває до останнього ряду, тим вищою є ймовірність її перетворення на дамку. Оцінювання просування виконується за формулою:

$$P(s) = 7 \cdot \left(\sum_{i \in W_m} (9 - r_i) - \sum_{j \in B_m} r_j \right)$$

у даній формулі $P(s)$ - оцінка просування, W_m і B_m - множини простих білих і чорних шашок, r_i і r_j - номери рядків відповідних фігур у матриці поля. Рядки нумеруються від 0 до 9.

Для білої шашки прогрес визначається як $9 - r_i$, оскільки вона рухається до нульового рядка. Для чорної шашки використовується значення r_j , оскільки вона рухається до дев'ятого рядка.

Контроль центру оцінюється на основі відстані фігури до геометричного центра поля. Центральність клітини визначається за формулою:

$$c(r, q) = \max(0, 5 - (|r - 4,5| + |q - 4,5|))$$

у даній формулі $c(r, q)$ - центральність клітини, r - номер рядка, q - номер стовпця, точка з координатами (4,5; 4,5) - геометричний центр поля.

Зі збільшенням відстані до центра значення $c(r, q)$ зменшується. Для клітин, розташованих далеко від центральної частини, воно дорівнює нулю.

Загальну оцінку контролю центру визначено формулою:

$$C(s) = 6 \cdot \left(\sum_{i \in W} \lambda_i \cdot c(r_i, q_i) - \sum_{j \in B} \lambda_j \cdot c(r_j, q_j) \right)$$

у даній формулі $C(s)$ - оцінка контролю центру, W і B - множини білих і чорних фігур, λ_i і λ_j - коефіцієнти типу фігури, $c(r, q)$ - центральність її клітини. Для простої шашки $\lambda = 1$, для дамки $\lambda = 1,2$.

Мобільність характеризує кількість допустимих дій, доступних кожній стороні. Її оцінювання виконується за формулою:

$$R(s) = 5 \cdot (|A_W(s)| - |A_B(s)|)$$

у формулі $R(s)$ - оцінка мобільності, $A_W(s)$ і $A_B(s)$ - множини допустимих ходів білих і чорних, $|A_W(s)|$ і $|A_B(s)|$ - кількість ходів у відповідних множинах.

Мобільність оцінюється для обох сторін незалежно від того, кому належить поточний хід. Вища кількість доступних дій означає більшу свободу переміщення та меншу ймовірність блокування фігур.

Потенціал взяття враховує максимальну кількість фігур, яку кожна сторона може взяти за один хід. Його задано формулою:

$$K(s) = 34 \cdot (k_W^{max}(s) - k_B^{max}(s))$$

у даній формулі $K(s)$ - оцінка потенціалу взяття, $k_W^{max}(s)$ і $k_B^{max}(s)$ - максимальна кількість фігур, яку можуть взяти білі та чорні за один допустимий хід.

Якщо взяття для сторони відсутнє, відповідне значення k^{max} дорівнює нулю. Урахування саме максимальної довжини маршруту відповідає правилу міжнародних шашок про обов'язковий вибір ходу з найбільшою кількістю взятих фігур.

Кінцева складова повинна мати значення лише для позицій, у яких одна зі сторін програла. Її визначено за формулою:

$$Z(s) = 100000 \cdot I_B(s) - 100000 \cdot I_W(s)$$

у даній формулі $Z(s)$ - оцінка кінцевого стану, $I_B(s)$ - індикатор поразки чорних, $I_W(s)$ - індикатор поразки білих. Індикатор дорівнює 1, якщо відповідна сторона не має фігур або допустимих ходів, і 0 в іншому випадку.

Наприклад, якщо чорні не мають допустимих ходів, $I_B(s) = 1$, $I_W(s) = 0$, тому $Z(s) = 100000$. Якщо можливість ходу втратили білі, значення

становить -100000. Для незавершеної позиції обидва індикатори дорівнюють нулю.

Таким чином, евристична функція поєднує матеріальні, тактичні та позиційні характеристики. Матеріал відображає наявну кількісну перевагу, просування і центр характеризують розташування фігур, мобільність визначає свободу дій, а потенціал взяття враховує найближчі тактичні можливості. Отримана статична оцінка використовується для порівняння граничних позицій у пошукових алгоритмах.

2.4 Алгоритми вибору ходу на основі пошуку

Алгоритми вибору ходу на основі пошуку аналізують можливі продовження партії, подані у вигляді дерева ігрових станів. Для кожного початкового ходу розглядаються відповіді суперника та наступні дії поточного гравця. Після досягнення кінцевої позиції або заданої глибини отримані значення повертаються до кореня дерева.

У цьому підрозділі розглядаються мінімакс та альфа-бета відсікання. Обидва алгоритми виходять із припущення, що сторони діють раціонально: один гравець намагається максимізувати оцінку, а інший - мінімізувати її. Різниця полягає в кількості вершин, які необхідно дослідити для отримання результату.

2.4.1 Алгоритм мінімакс

Мінімакс є рекурсивним алгоритмом пошуку рішення у грі з двома сторонами та протилежними інтересами. Для гравця, відносно якого виконується аналіз, використовуються вершини типу MAX. Вершини, що відповідають ходам суперника, мають тип MIN.

У вершині MAX обирається дочірній стан із найбільшою оцінкою. Значення такої вершини на глибині d визначається за формулою:

$$V_d(s) = \max_{(a \in A(s))} V_{(d-1)}(T(s, a))$$

у даній формулі $V_d(s)$ - оцінка стану s на глибині d , $A(s)$ - множина допустимих ходів, $T(s, a)$ - стан після виконання ходу a .

У вершині MIN суперник обирає варіант, який має найменшу оцінку для сторони, відносно якої проводиться аналіз. Значення MIN-вершини визначається за формулою:

$$V_d(s) = \min_{(a \in A(s))} V_{(d-1)}(T(s, a))$$

Після досягнення нульової залишкової глибини подальше розгортання дерева припиняється. Для отриманої позиції використовується евристична функція оцінювання. Якщо до завершення заданої глибини досягнуто кінцевого стану, замість евристичної оцінки повертається значення перемоги або поразки.

На рисунку 2.3 наведено приклад роботи мінімаксу для дерева глибиною два півходи.

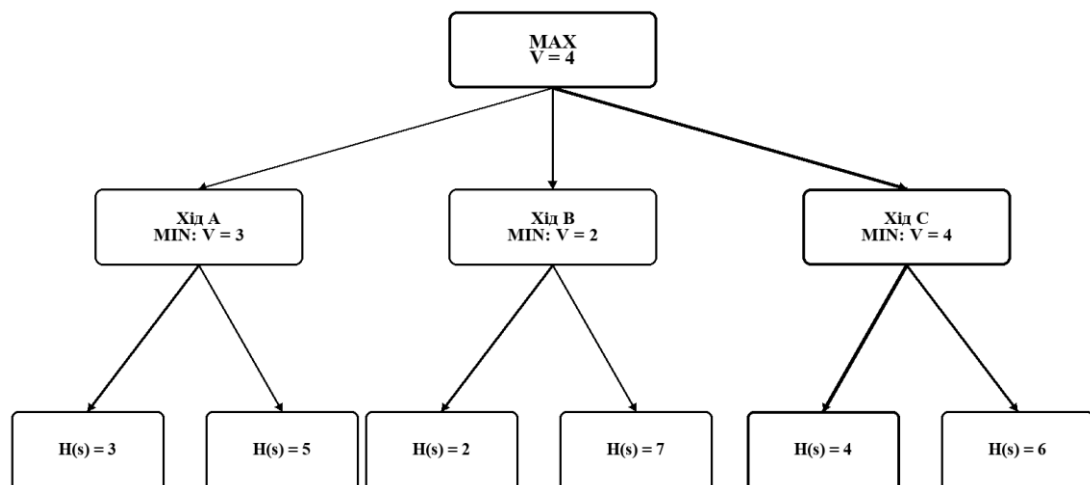


Рисунок 2.3 – Приклад роботи мінімаксу для дерева глибиною два півходи

У наведеному прикладі кожна MIN-вершина обирає найменше значення серед своїх дочірніх позицій. Для ходу *A* повертається значення 3, для ходу *B* - значення 2, а для ходу *C* - значення 4. Коренева вершина MAX обирає найбільше з отриманих значень, тому результатом пошуку є хід *C* з оцінкою 4.

На рисунку 2.4 наведено псевдокод рекурсивного мінімаксного пошуку.

```
function Minimax(state, depth, maximizing)
    if GameOver(state) then
        return TerminalScore(state)

    if depth = 0 then
        return Evaluate(state)

    moves = GenerateMoves(state)

    if maximizing then
        best = -INF

        for move in moves do
            nextState = MakeMove(state, move)
            score = Minimax(nextState, depth - 1, false)
            best = max(best, score)

        return best
    else
        best = +INF

        for move in moves do
            nextState = MakeMove(state, move)
            score = Minimax(nextState, depth - 1, true)
            best = min(best, score)

        return best
```

Рисунок 2.4 – Псевдокод рекурсивного мінімаксного пошуку

Основним недоліком мінімаксу є необхідність перегляду всіх доступних гілок до встановленої глибини. Його обчислювальну складність наближено визначено формулою:

$$T_{(minimax)}(b, d) = O(b^d)$$

у даній формулі $T_{minimax}(b, d)$ - обчислювальна складність мінімаксу, b - середній коефіцієнт розгалуження, d - глибина пошуку.

Наприклад, за середнього коефіцієнта розгалуження $b = 8$ перехід від глибини 6 до глибини 7 теоретично збільшує кількість досліджуваних вершин приблизно у 8 разів. Тому звичайний мінімакс може використовуватися лише на відносно невеликій глибині.

2.4.2 Алгоритм альфа-бета відсікання

Альфа-бета відсікання є вдосконаленням мінімаксного пошуку. Воно повертає такий самий результат, але не досліджує гілки, які вже не можуть змінити остаточний вибір ходу.

Під час пошуку зберігаються дві межі:

- 1) альфа - найкраща оцінка, яку на поточний момент гарантовано може отримати гравець MAX;
- 2) бета - найкраща оцінка, яку на поточний момент може гарантувати гравець MIN.

У MAX-вершині значення альфа оновлюється за формулою:

$$\alpha = \max(\alpha, v)$$

у даній формулі α - поточна нижня межа оцінки, v - оцінка останньої дослідженої дочірньої вершини.

У MIN-вершині значення бета оновлюється за формулою:

$$\beta = \min(\beta, v)$$

у даній формулі β - поточна верхня межа оцінки.

Подальше дослідження поточної гілки припиняється за умови, наведеної у формулі:

$$\alpha \geq \beta$$

ця умова означає, що вже знайдений результат не буде обраний одним із гравців, незалежно від оцінок інших дочірніх вершин цієї гілки.

Принцип роботи альфа-бета відсікання для того самого дерева показано на рисунку 2.5.

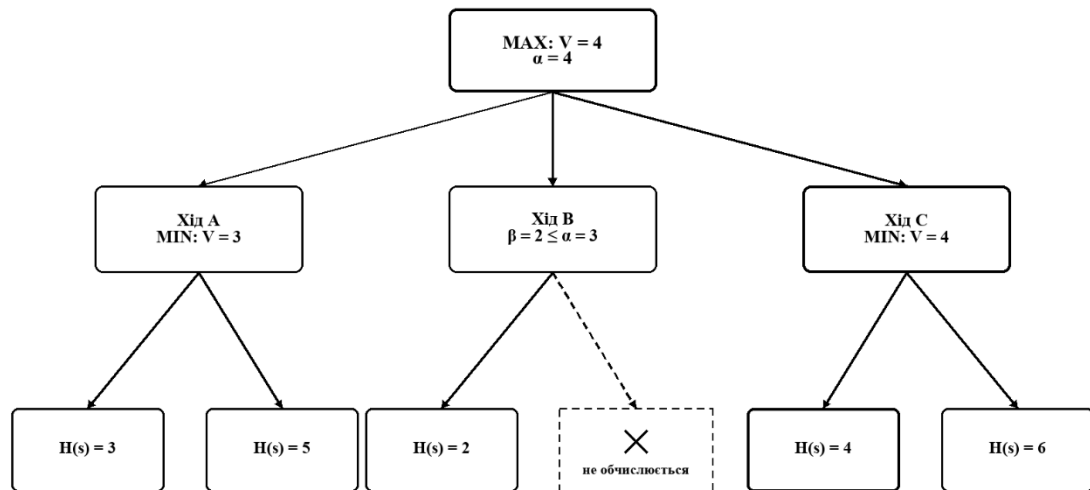


Рисунок 2.5 – Принцип роботи альфа-бета відсікання

Спочатку повністю аналізується хід *A*, після чого коренева вершина отримує значення альфа, що дорівнює 3. Під час аналізу ходу *B* перша дочірня позиція отримує оцінку 2. Оскільки вершина *B* належить гравцю *MIN*, значення бета стає рівним 2.

У результаті виконується умова $\beta \leq \alpha$, тобто $2 \leq 3$. Навіть якщо наступна позиція у гілці *B* матиме вищу оцінку, гравець *MIN* уже може обрати результат 2. Гравець *MAX*, у свою чергу, не обере хід *B*, оскільки після ходу *A* він уже може гарантувати значення 3. Тому друга дочірня позиція гілки *B* не обчислюється.

Після аналізу ходу *C* його мінімальне значення дорівнює 4. Коренева вершина порівнює оцінки 3, 2 і 4 та обирає хід *C*. Отже, альфа-бета відсікання повертає той самий результат, що й мінімакс, але обробляє меншу кількість вершин. На рисунку 2.6 наведено псевдокод альфа-бета пошуку.

```

function AlphaBeta(state, depth, alpha, beta, maximizing)
  if GameOver(state) then
    return TerminalScore(state)
  if depth = 0 then
    return Evaluate(state)
  moves = GenerateMoves(state)
  moves = OrderMoves(moves)
  if maximizing then
    best = -INF

    for move in moves do
      nextState = MakeMove(state, move)
      score = AlphaBeta(nextState, depth - 1,
                        alpha, beta, false)
      best = max(best, score)
      alpha = max(alpha, best)

      if alpha >= beta then
        break
    return best
  else
    best = +INF
    for move in moves do
      nextState = MakeMove(state, move)
      score = AlphaBeta(nextState, depth - 1,
                        alpha, beta, true)
      best = min(best, score)
      beta = min(beta, best)

      if alpha >= beta then
        break
    return best

```

Рисунок 2.6 – Псевдокод альфа-бета пошуку

У найгіршому випадку, коли ходи розглядаються у невдалому порядку, складність альфа-бета пошуку залишається такою самою, як у мінімаксу. За близького до оптимального порядку ходів вона наближається до значення, наведеного у формулі:

$$T_{(\alpha\beta)}(b, d) = O(b^{(d/2)})$$

у даній формулі $T_{\alpha\beta}(b, d)$ - складність альфа-бета пошуку за сприятливого впорядкування ходів, b - середній коефіцієнт розгалуження, d - глибина пошуку.

Таким чином, за однакового обчислювального часу альфа-бета відсікання дозволяє досягти більшої глибини, ніж звичайний мінімакс. Проте ефективність відсікань суттєво залежить від порядку, у якому аналізуються допустимі ходи.

2.4.3 Упорядкування ходів, таблиці транспозицій та пошук спокійних позицій

Для підвищення ефективності альфа-бета пошуку недостатньо лише перевіряти умову $\alpha \geq \beta$. Важливо якомога раніше знаходити сильні ходи, уникати повторного аналізу однакових позицій та не завершувати пошук у тактично нестабільних станах.

Упорядкування ходів полягає в тому, що перед рекурсивним аналізом допустимі дії сортуються за попередньою оцінкою. У першу чергу розглядаються ходи, які з більшою ймовірністю можуть бути сильними.

Для міжнародних шашок пріоритетними є:

- 1) ходи з найбільшою кількістю взятих фігур;
- 2) взяття дамок суперника;
- 3) перетворення простої шашки на дамку;
- 4) переміщення до центральної частини поля;
- 5) просування простої шашки до останнього ряду.

Правило максимальної кількості взятих фігур уже враховується під час формування множини допустимих ходів. Додаткове впорядкування застосовується до маршрутів, які мають однакову допустиму кількість взяттів, а також до звичайних переміщень.

Сам порядок ходів не змінює результат повністю завершеного альфа-бета пошуку. Він впливає лише на те, наскільки рано виникають відсікання та скільки вершин необхідно обробити.

Іншим способом скорочення обчислень є використання таблиці транспозицій. Одна й та сама позиція може виникнути після різних послідовностей ходів. Така ситуація показана на рисунку 2.7.

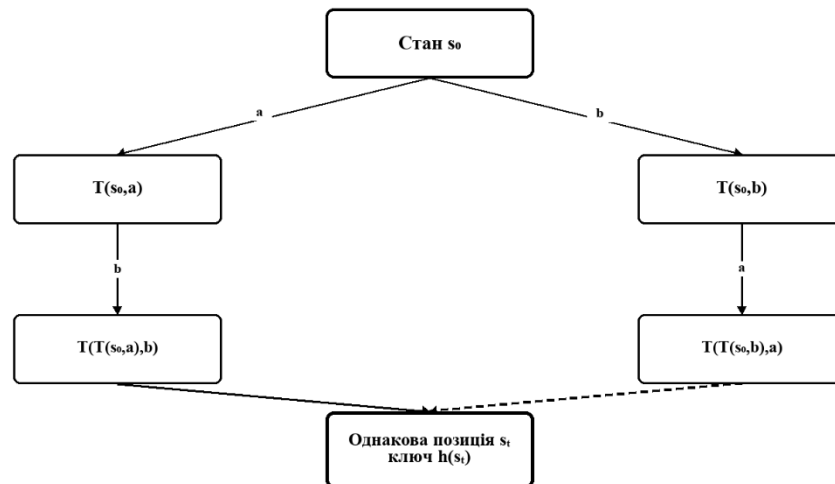


Рисунок 2.7 – Ситуація, коли та сама позиція може виникнути після різних послідовностей ходів

На рисунку ходи a та b виконуються у різній послідовності, але приводять до однакового стану s_t . Без використання таблиці цей стан і всі його продовження були б досліджені двічі.

Для кожного стану обчислюється ключ $h(s)$, який залежить від розташування фігур і сторони, якій належить хід. У таблиці зберігаються ключ позиції, глибина виконаного пошуку, отримана оцінка та тип межі. Структуру одного запису наведено у таблиці 2.1.

Оцінка з таблиці може мати один із трьох типів:

- 1) exact - точне значення позиції для виконаної глибини;
- 2) lower - нижня межа, отримана після відсікання у MAX-вершині;
- 3) upper - верхня межа, отримана після відсікання у MIN-вершині.

Таблиця 2.1 - Структура запису в таблиці транспозицій

Поле	Позначення	Призначення
Ключ позиції	$h(s)$	Ідентифікація розташування фігур і сторони ходу
Глибина	d	Глибина, для якої отримано результат
Оцінка	v	Значення позиції, знайдене під час пошуку
Тип межі	τ	Спосіб використання збереженої оцінки

Запис може бути використаний лише тоді, коли збережена глибина не менша за глибину поточного запиту. Якщо позиція раніше була досліджена на меншій глибині, її оцінка може бути недостатньо точною для негайного завершення нового пошуку.

У розглянутому підході ключ позиції також містить залишкову глибину. Це дозволяє відокремлювати результати, отримані для різної кількості майбутніх півходів, і спрощує перевірку придатності збереженої оцінки.

Ще однією проблемою пошукових алгоритмів є ефект горизонту. Він виникає, коли максимальна глибина завершується посеред форсованої послідовності взяттів. Статична оцінка такої позиції може показувати перевагу, хоча на наступному півході суперник виконає обов'язкове взяття та змінить матеріальне співвідношення.

Для зменшення цього ефекту використовується пошук спокійних позицій. Його принцип показано на рисунку 2.8. Після досягнення основної межі $d_{(max)}$ система перевіряє наявність взяттів. Якщо взяття відсутні, позиція вважається спокійною та оцінюється функцією $H(s)$. Якщо доступні обов'язкові взяття, пошук продовжується лише для них.

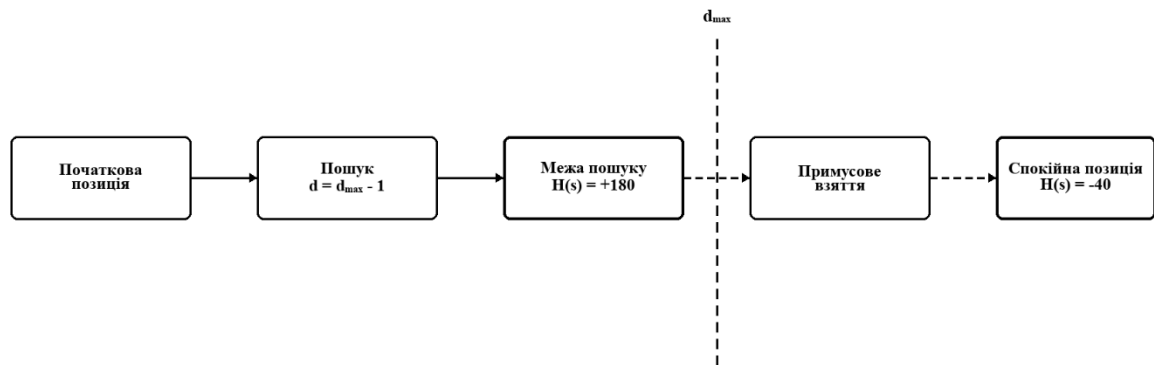


Рисунок 2.8 – Принцип пошуку спокійних позицій

Умову спокійної позиції визначено формулою:

$$Quiet(s) \Leftrightarrow K(s) = \emptyset$$

у даній формулі $Quiet(s)$ - ознака спокійної позиції, $K(s)$ - множина доступних маршрутів взяття.

На рисунку 2.8 позиція на основній межі пошуку має оцінку +180. Після продовження обов'язкового взяття досягається спокійна позиція з оцінкою -40. Без додаткового аналізу алгоритм міг би помилково вважати початкове продовження вигідним.

Глибина пошуку спокійних позицій також повинна мати обмеження. У деяких станах послідовність взяттів може бути довгою, тому необмежене продовження призвело б до значного збільшення часу аналізу. Після досягнення додаткової межі позиція оцінюється незалежно від наявності наступних взяттів.

Отже, впорядкування ходів збільшує кількість ранніх альфа-бета відсікань, таблиця транспозицій усуває повторний аналіз однакових станів, а пошук спокійних позицій зменшує вплив штучного обмеження глибини. Спільне застосування цих методів дозволяє досліджувати дерево глибше без зміни основного принципу мінімаксного прийняття рішення.

2.5 Метод пошуку Монте-Карло в дереві

Метод пошуку Монте-Карло в дереві, або Monte Carlo Tree Search (MCTS), виконує вибір ходу на основі багаторазового моделювання можливих продовжень партії. На відміну від мінімаксу, MCTS не переглядає всі гілки дерева до однакової глибини. Більша частина обчислювального бюджету спрямовується на дослідження ходів, які показали кращі результати під час попередніх моделювань.

Обчислювальний бюджет MCTS визначається кількістю ітерацій K . Одна ітерація складається з вибору вузла, розширення дерева, моделювання продовження партії та зворотного поширення результату. Після виконання заданої кількості ітерацій алгоритм обирає хід, який отримав найбільшу кількість відвідувань.

У кожному вузлі дерева зберігається ігрова позиція, сторона, яка виконує наступний хід, батьківський вузол, дочірні вузли, множина ще не перевірених ходів, кількість відвідувань n та накопичений результат W . Загальний порядок роботи методу наведено у вигляді псевдокоду на рисунку 2.9.

```
function MCTS(state, rootSide, iterations)
    root = CreateNode(state, rootSide)

    for i = 1 to iterations do
        node = root

        while node has no untried moves and node has children do
            node = SelectUCTChild(node, rootSide)

        if node has untried moves then
            node = Expand(node)

        result = Simulate(node.state, node.side, rootSide)
        Backpropagate(node, result)

    return MoveOfMostVisitedChild(root)
```

Рисунок 2.9 – псевдокод роботи MCTS

2.5.1 Вибір та розширення вузлів дерева

Вибір починається з кореневого вузла, який відповідає поточній позиції на дошці. Алгоритм переходить до дочірніх вузлів, доки не буде знайдено вузол із нерозглянутими ходами або кінцеву позицію. Для вибору гілки використовується критерій Upper Confidence Bound applied to Trees (UCT) , який поєднує використання вже отриманих результатів і дослідження недостатньо перевірених варіантів.

Середній результат моделювань для i -го дочірнього вузла визначається за формулою:

$$E_i = \frac{W_i}{n_i}$$

у даній формулі W_i - накопичений результат моделювань, n_i - кількість відвідувань вузла, E_i - середній результат. Значення E_i знаходиться в межах від 0 до 1 та обчислюється відносно сторони, для якої вибирається хід.

Якщо у поточному вузлі хід виконує суперник, використовується обернене значення середнього результату, яке визначається за формулою:

$$E_i = 1 - \frac{W_i}{n_i}$$

Такий підхід враховує протилежність цілей гравців. Коренева сторона намагається збільшити ймовірність власного виграшу, тоді як для суперника перспективними є продовження, які зменшують цю ймовірність.

Критерій вибору дочірнього вузла обчислюється за формулою:

$$UCT_i = E_i + c \sqrt{\frac{1}{n_i} \ln N}$$

у даній формулі N - кількість відвідувань батьківського вузла, c - коефіцієнт дослідження. У розробленому методі використовується значення $c = \sqrt{2}$.

Перша частина формули характеризує накопичений результат вузла, а друга збільшує оцінку варіантів із малою кількістю відвідувань. Якщо вузол ще не відвідувався, він вибирається без обчислення UCT.

Після вибору виконується розширення дерева. З множини нерозглянутих ходів вузла вибирається один хід, який застосовується до відповідної позиції. Для отриманого стану створюється новий дочірній вузол із початковими значеннями $n = 0$ та $W = 0$. За одну ітерацію до дерева додається не більше одного нового вузла.

Процес вибору гілки за критерієм UCT і подальшого розширення дерева показано на рисунку 2.10. Товстими лініями позначено вибраний шлях, а штриховими - ходи, для яких ще не було створено дочірні вузли. Завдяки такому підходу дерево будується нерівномірно. Перспективні варіанти отримують більше відвідувань і досліджуються докладніше, а слабкі гілки займають меншу частину обчислювального бюджету.

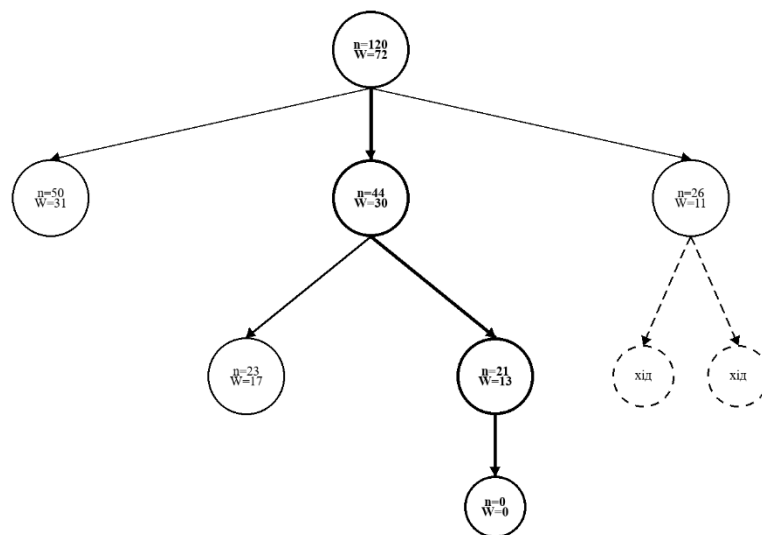


Рисунок 2.10 - Процес вибору гілки за критерієм UCT

2.5.2 Моделювання продовження партії

Після створення нового вузла виконується моделювання продовження партії. Під час моделювання ходи виконуються без додавання нових вузлів до основного дерева. Процес завершується, якщо одна зі сторін не має допустимих ходів або досягнуто обмеження на кількість півходів.

У класичному MCTS ходи під час моделювання можуть вибиратися випадково. Для міжнародних шашок повністю випадкове моделювання є недостатньо точним, оскільки воно часто призводить до необґрунтованої втрати шашок або пропуску сильних узяттів. Тому в розробленому методі застосовується евристично зважений вибір.

Вага ходу залежить від кількості взятих шашок, узяття дамок, перетворення шашки на дамку, наближення до центра та просування до останнього ряду. Імовірність вибору ходу визначається за формулою:

$$p(a|s) = w(a) / \sum_{b \in A(s)} w(b)$$

у даній формулі $p(a|s)$ - імовірність вибору ходу a у позиції s , $w(a)$ - евристична вага ходу, $A(s)$ - множина допустимих ходів. Хід із більшою вагою вибирається частіше, але слабші ходи також мають ненульову ймовірність вибору.

Якщо партія завершується під час моделювання, перемозі кореневої сторони відповідає результат $r = 1$, а поразці - $r = 0$. Якщо встановлене обмеження досягнуто раніше, ніж завершилася партія, результат визначається за допомогою евристичної функції оцінювання:

$$r_H(s_L) = \min(1, \max(0, 0.5 + H(s_L)/2400))$$

у даній формулі s_L - позиція після завершення моделювання, $H(s_L)$ - її евристична оцінка, $r_H(s_L)$ - нормалізований результат. Нейтральній позиції відповідає значення 0,5. Перевага кореневої сторони формує значення, більше за 0,5, а перевага суперника - менше за 0,5.

Моделювання продовження партії та подальше повернення отриманого результату до вузлів дерева показано на рисунку 2.11.

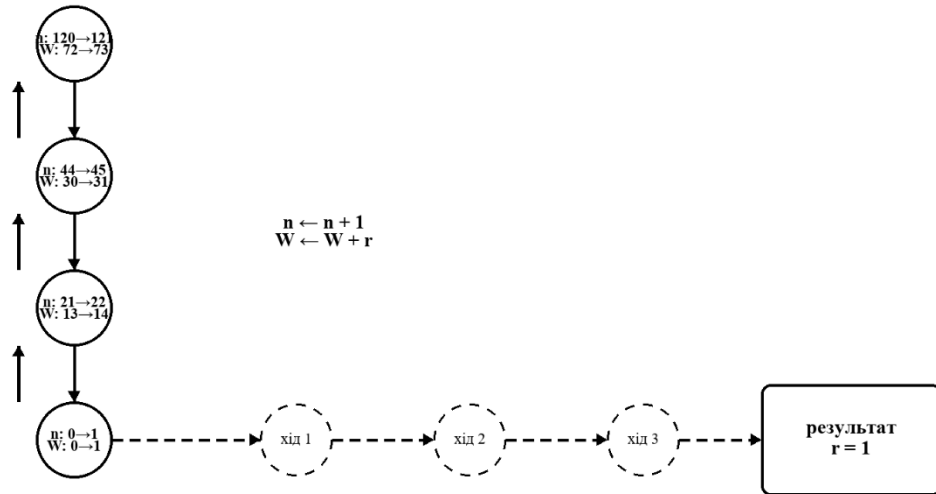


Рисунок 2.11 – Модель продовження партії та подальше повернення отриманого результату

Поєднання випадковості з евристичною оцінкою ходів дозволяє отримувати змістовніші результати моделювань. При цьому MCTS не виконує повний пошук дерева під час кожної симуляції, тому може перевірити значну кількість різних продовжень.

2.5.3 Зворотне поширення результатів

Після завершення моделювання його результат поширюється від останнього вибраного вузла до кореня дерева. Для кожного вузла на пройденому шляху збільшується кількість відвідувань і накопичений результат. Оновлення виконується за формулою:

$$n_j \leftarrow n_j + 1, W_j \leftarrow W_j + r$$

у даній формулі n_j - кількість відвідувань j -го вузла, W_j - накопичений результат, r - результат поточного моделювання. Наприклад, після моделювання з результатом $r = 1$ значення $n_j=44$ і $W_j=25$ змінюються на $n_j=45$ і $W_j=26$.

Однаковий результат додається до всіх вузлів вибраного шляху, оскільки їх статистика зберігається відносно кореневої сторони. Під час наступних ітерацій оновлені значення впливають на критерій UCT і змінюють імовірність повторного вибору відповідної гілки.

Після виконання K ітерацій MCTS обирає хід із найбільшою кількістю відвідувань. Правило вибору записується формулою:

$$a^* = \arg \max_{(a \in A(s_0))} n(s_0, a)$$

у даній формулі a^* - вибраний хід, s_0 - початкова позиція, $A(s_0)$ - множина допустимих ходів, $n(s_0, a)$ - кількість відвідувань вузла, що відповідає ходу a . Якщо декілька ходів мають однакову кількість відвідувань, обирається хід із більшою часткою успішних моделювань.

Таким чином, результат роботи MCTS залежить від статистики великої кількості змодельованих продовжень. Збільшення кількості ітерацій підвищує стабільність вибору, але також пропорційно збільшує час аналізу позиції.

2.6 Адаптивний метод прийняття рішень єдиної системи штучного інтелекту

Єдина система штучного інтелекту поєднує метод MCTS та покращений альфа-бета пошук. “Єдина” в даному випадку означає спільний механізм керування кількома методами. Система самостійно аналізує поточну позицію, визначає її складність і встановлює параметри пошуку. Користувачу не потрібно задавати глибину, кількість симуляцій або максимальну кількість досліджуваних вузлів.

MCTS використовується для попереднього дослідження складних позицій із великою кількістю допустимих ходів. Отриманий ним хід першим перевіряється альфа-бета пошуком. У тактичних позиціях з обов'язковими взяттями MCTS не запускається, оскільки альфа-бета пошук точніше аналізує форсовані послідовності ходів.

2.6.1 Оцінювання складності поточної позиції

Перед вибором ходу система формує профіль поточної позиції. Для цього визначається загальна кількість фігур, кількість дамок, кількість допустимих ходів, наявність ходів із взяттям та максимальна довжина одного взяття.

Фаза партії визначається за загальною кількістю фігур P :

- 1) якщо $P > 18$, позиція належить до початку партії;
- 2) якщо $8 < P \leq 18$, позиція належить до середини партії;
- 3) якщо $P \leq 8$, позиція належить до ендшпілю.

Показник складності позиції обчислюється за формулою:

$$C(s) = \text{round}(\max(1, B + 1.5K + I_c(4 + 2L) + F))$$

у даній формулі $C(s)$ - складність позиції, B - кількість допустимих ходів, K - загальна кількість дамок, L - максимальна кількість фігур, які можна взяти одним ходом, I_c - індикатор наявності взяття, F - поправка за фазу партії.

Значення індикатора I_c дорівнює 1, якщо в позиції існує хоча б один хід із взяттям. Якщо взяття відсутні, $I_c=0$. Поправка за фазу партії визначається за формулою:

$$F = \begin{cases} 2, & P > 18 \\ 0, & 8 < P \leq 18 \\ -3, & P \leq 8 \end{cases}$$

Найбільший вплив на складність має кількість допустимих ходів, оскільки вона безпосередньо визначає розгалуженість дерева. Дамки

збільшують кількість можливих переміщень через здатність рухатися на декілька клітин. Наявність взяття також підвищує складність, особливо якщо один хід містить послідовне взяття декількох фігур.

Наприклад, нехай у середині партії гравець має 12 допустимих ходів, на дошці знаходяться 4 дамки, а максимальна довжина взяття дорівнює 2. Тоді складність позиції становить:

$$C(s) = \text{round}(12 + 1.5 \cdot 4 + 4 + 2 \cdot 2) = 26.$$

Для визначення необхідності запуску MCTS використовуються три умови. Вони записуються формулою:

$$I_{MCTS} = \begin{cases} 1, & I_c = 0, B \geq 10, C(s) \geq 18 \\ 0, & \text{інакше} \end{cases}$$

MCTS запускається лише у складній позиції без обов'язкового взяття, якщо існує не менше 10 допустимих ходів, а показник складності становить не менше 18. Якщо хоча б одна умова не виконується, система одразу переходить до адаптивного альфа-бета пошуку.

2.6.2 Адаптивний вибір глибини та обмежень пошуку

Після оцінювання позиції система визначає цільову глибину альфа-бета пошуку. Базова глибина залежить від фази партії:

- 1) для початку партії $d_0=10$;
- 2) для середини партії $d_0=11$;
- 3) для ендшпілю $d_0=14$.

Більша глибина в ендшпілі пояснюється меншою кількістю фігур і нижчою розгалуженістю дерева. Остаточна цільова глибина визначається за формулою:

$$d_{(target)} = \text{clip}(d_0 + \Delta_B + \Delta_C + \Delta_K, 4, 20)$$

у даній формулі d_{target} - цільова глибина, Δ_B - поправка за кількість допустимих ходів, Δ_C - поправка за можливі взяття, Δ_K - поправка за кількість дамок. Функція *clip* обмежує значення діапазоном від 4 до 20.

Поправка за кількість ходів визначається за формулою:

$$\Delta_B = 3I_{(B \leq 4)} - I_{(B \geq 12)} - I_{(B \geq 18)}$$

Якщо кількість допустимих ходів не перевищує 4, глибина збільшується на 3. Якщо кількість ходів становить не менше 12, глибина зменшується на 1. Для 18 або більшої кількості ходів вона зменшується ще на 1.

Поправки за взяття та дамки визначаються за формулою:

$$\Delta_C = I_{(I_c=1)} + I_{(L \geq 2)}, \quad \Delta_K = -I_{(K \geq 6)}$$

За наявності взяття глибина збільшується на 1. Якщо максимальна довжина взяття становить не менше двох фігур, вона додатково збільшується ще на 1. За наявності шести або більше дамок глибина зменшується на 1 через значне зростання кількості можливих переміщень.

Крім глибини, система встановлює максимальну кількість вузлів. Базовий бюджет для вибору ходу становить 1 000 000 вузлів. Його адаптація виконується за формулою:

$$N_{max} = 1,000,000 + 350,000I_e + 180,000I_b - 120,000I_h + 160,000I_c$$

у даній формулі $I_e=1$ для ендшпілю, $I_b=1$ за кількості ходів $B \leq 5$, $I_h=1$ за кількості ходів $B \geq 14$, $I_c = 1$ за наявності взяття. В інших випадках відповідні індикатори дорівнюють нулю.

Наприклад, для ендшпілю з чотирма допустимими ходами та можливим взяттям бюджет становить:

$$N_{(max)} = 1,000,000 + 350,000 + 180,000 + 160,000 = 1,690,000 \text{ вузлів.}$$

Максимальний час вибору одного ходу є спільним для MCTS та альфа-бета пошуку і становить 7000 мс. У разі використання MCTS йому надається 22% цього часу. Обмеження часу MCTS визначається за формулою:

$$t_{MCTS} = \min(1800, \max(350, 0.22t_{max}))$$

За стандартного обмеження $t_{(max)} = 7000\text{мс}$ метод MCTS може працювати не більше:

$$t_{(MCTS)} = 0.22 \cdot 7000 = 1540\text{мс}.$$

Максимальна кількість ітерацій MCTS залежить від складності позиції та визначається за формулою:

$$K_{(MCTS)} = 6000 + 450 \cdot \max(0, C(s) - 18)$$

Наприклад, для позиції зі складністю $C(s) = 26$ максимальний бюджет становить:

$$K_{(MCTS)} = 6000 + 450 \cdot (26 - 18) = 9600 \text{ ітерацій}.$$

MCTS завершується після виконання $K_{(MCTS)}$ ітерацій або після вичерпання виділеного часу. Після цього залишок загального часу використовується альфа-бета пошуком. Основний пошук припиняється, якщо досягнуто максимальну кількість вузлів або загальний час аналізу:

$$N \geq N_{max} \text{ або } t \geq t_{max}$$

Таким чином, система не допускає необмеженого зростання часу пошуку та використання обчислювальних ресурсів.

2.6.3 Ітеративне поглиблення та вибір остаточного ходу

Альфа-бета пошук виконується з ітеративним поглибленням. Спочатку дерево аналізується на глибині 1, після чого глибина послідовно збільшується до d_{target} :

$$d = 1, 2, 3, \dots, d_{target}$$

Якщо перед альфа-бета пошуком було виконано MCTS, запропонований ним хід розміщується на початку списку. Завдяки цьому перспективна за результатами моделювань гілка перевіряється першою. Якщо її оцінка підтверджується, межі альфа та бета звужуються раніше, що може збільшити кількість відсікань.

Після кожної повністю завершеної ітерації система зберігає найкращий хід, його оцінку та досягнуту глибину. Хід, знайдений на поточній глибині, першим розглядається під час наступної ітерації. Це також покращує впорядкування ходів.

Якщо під час аналізу глибини d вичерпано час або бюджет вузлів, незавершений результат не замінює вже отримане рішення. Система повертає результат останньої повністю завершеної ітерації.

Наприклад, якщо пошук повністю завершив глибину 8, але був перерваний під час аналізу глибини 9, остаточною стає хід, знайдений на глибині 8.

На граничних вузлах застосовується пошук спокійних позицій із максимальною додатковою глибиною 12 півходів. Якщо у позиції наявні взяття, система продовжує аналіз лише форсованих ходів. Евристична функція використовується після завершення серії взяттів або після досягнення встановленої межі.

Якщо в поточній позиції існує лише один допустимий хід, система виконує його без побудови дерева. Якщо допустимі ходи відсутні, фіксується завершення партії.

Таким чином, єдина система поєднує статистичне дослідження широких позицій методом MCTS із точнішим альфа-бета аналізом. Адаптивний вибір глибини, кількості вузлів і часу дозволяє спрямовувати більший обсяг обчислень на складні позиції та зменшувати витрати ресурсів у простіших ситуаціях.

Висновки до розділу 2

У другому розділі було розглянуто моделі та методи прийняття рішень для вибору ходу у міжнародних шашках. Описано подання ігрового стану,

формування множини допустимих ходів і побудову дерева можливих продовжень партії.

Розглянуто евристичну функцію оцінювання позиції, яка враховує матеріальну перевагу, мобільність, контроль центра, просування шашок і можливість взяття. Наведено принцип вибору вагових коефіцієнтів та визначено їхній вплив на загальну оцінку позиції.

Досліджено алгоритми мінімакс, альфа-бета відсікання та метод пошуку Монте-Карло в дереві. Розглянуто впорядкування ходів, таблиці транспозицій, пошук спокійних позицій та евристично кероване моделювання продовжень партії.

Запропоновано адаптивний метод єдиної системи штучного інтелекту, який поєднує MCTS і покращений альфа-бета пошук. Система оцінює складність позиції, автоматично визначає глибину та обчислювальні обмеження, а остаточний хід обирає за результатом останньої повністю завершеної ітерації пошуку.

РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ І АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Обґрунтування вибору технологій розробки

Для практичної реалізації системи штучного інтелекту було обрано вебтехнології: HTML, CSS і JavaScript.

HTML використовується для побудови структури сторінки, елементів керування, інформаційних панелей і областей відображення ігрового поля. CSS відповідає за зовнішній вигляд інтерфейсу, розміщення елементів, оформлення дошки, фігур, статистики та блоків налаштування алгоритмів.

Основна логіка системи реалізована мовою JavaScript. За допомогою JavaScript описуються правила міжнародних шашок, перевірка допустимих ходів, зміна стану партії, робота алгоритмів штучного інтелекту та оновлення інтерфейсу після кожної дії користувача або алгоритму.

Окремою перевагою JavaScript є можливість використання тієї самої мови як у браузері, так і в консольному середовищі Node.js. Це дозволяє запускати не тільки звичайну гру з інтерфейсом, а й фонові симуляції партій без графічного відображення, що є важливим для порівняння алгоритмів.

На етапах розробки не використовувались складні серверні технології або важкі фреймворки, оскільки основною метою було створення зрозумілої та керованої навчальної системи. Обраний набір технологій є достатнім для реалізації правил гри, інтерфейсу, алгоритмів штучного інтелекту та збору статистики партій.

3.2 Архітектура та організація програмної системи

Програмну систему побудовано за модульним принципом. Основними її складовими є інтерфейс користувача, контролер партії, контролер фонового моделювання, модуль правил міжнародних шашок, підсистема прийняття рішень та засоби збирання статистики.

Структурну схему розробленої програмної системи наведено на рисунку 3.1.

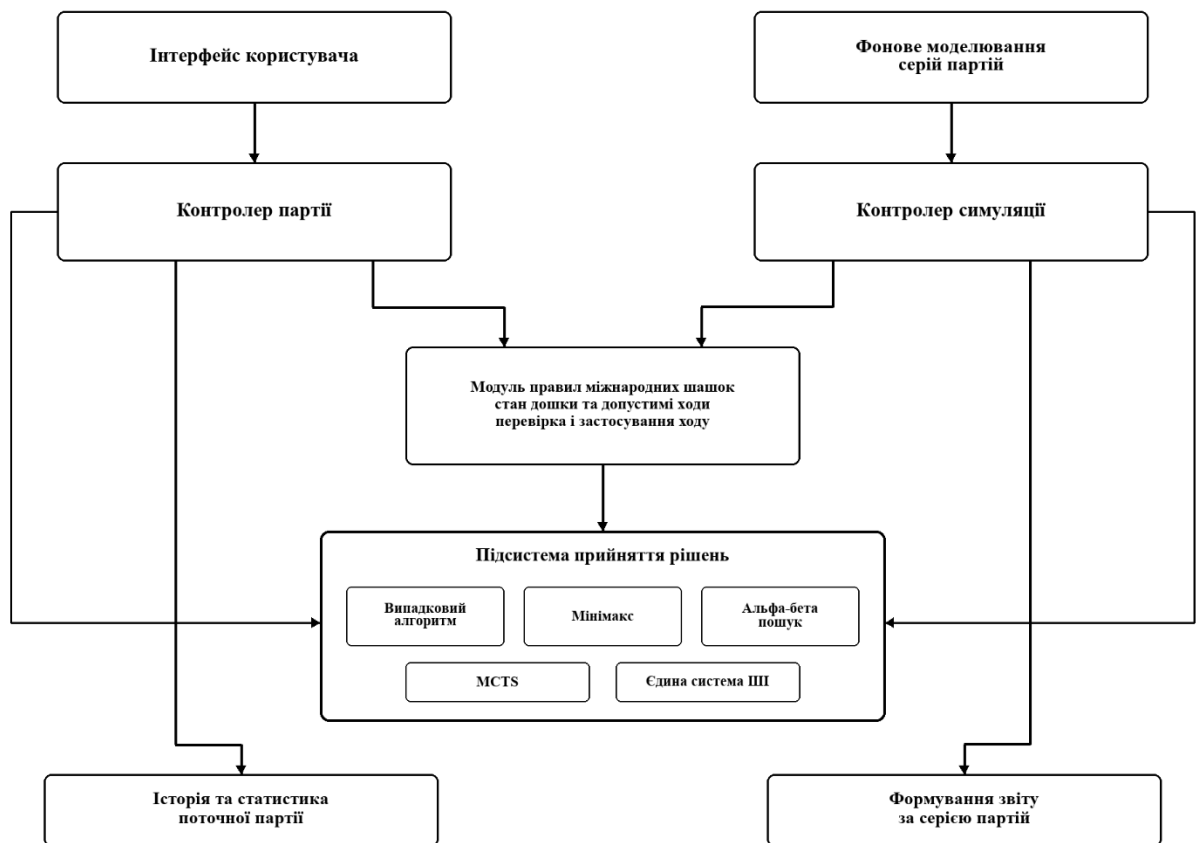


Рисунок 3.1 – Структурна схема програмної системи

Контролер партії керує грою через інтерфейс користувача. Він зберігає поточний стан партії, визначає сторону, яка повинна виконати хід, контролює таймери та передає отримані результати до журналу ходів. Контролер симуляції виконує аналогічні операції без відображення дошки та взаємодії з користувачем.

Обидва контролери безпосередньо взаємодіють із модулем правил та підсистемою прийняття рішень. Модуль правил зберігає стан дошки, формує множину допустимих ходів, перевіряє вибраний хід, застосовує його та визначає завершення партії. Підсистема прийняття рішень містить випадковий алгоритм, мінімакс, альфа-бета пошук, MCTS та єдину систему штучного інтелекту. На вхід алгоритму передаються поточна позиція, сторона гравця та сформована множина допустимих ходів. Результатом роботи є вибраний хід разом зі статистикою виконаного пошуку.

Послідовність виконання одного ходу в програмній системі показано на рисунку 3.2.

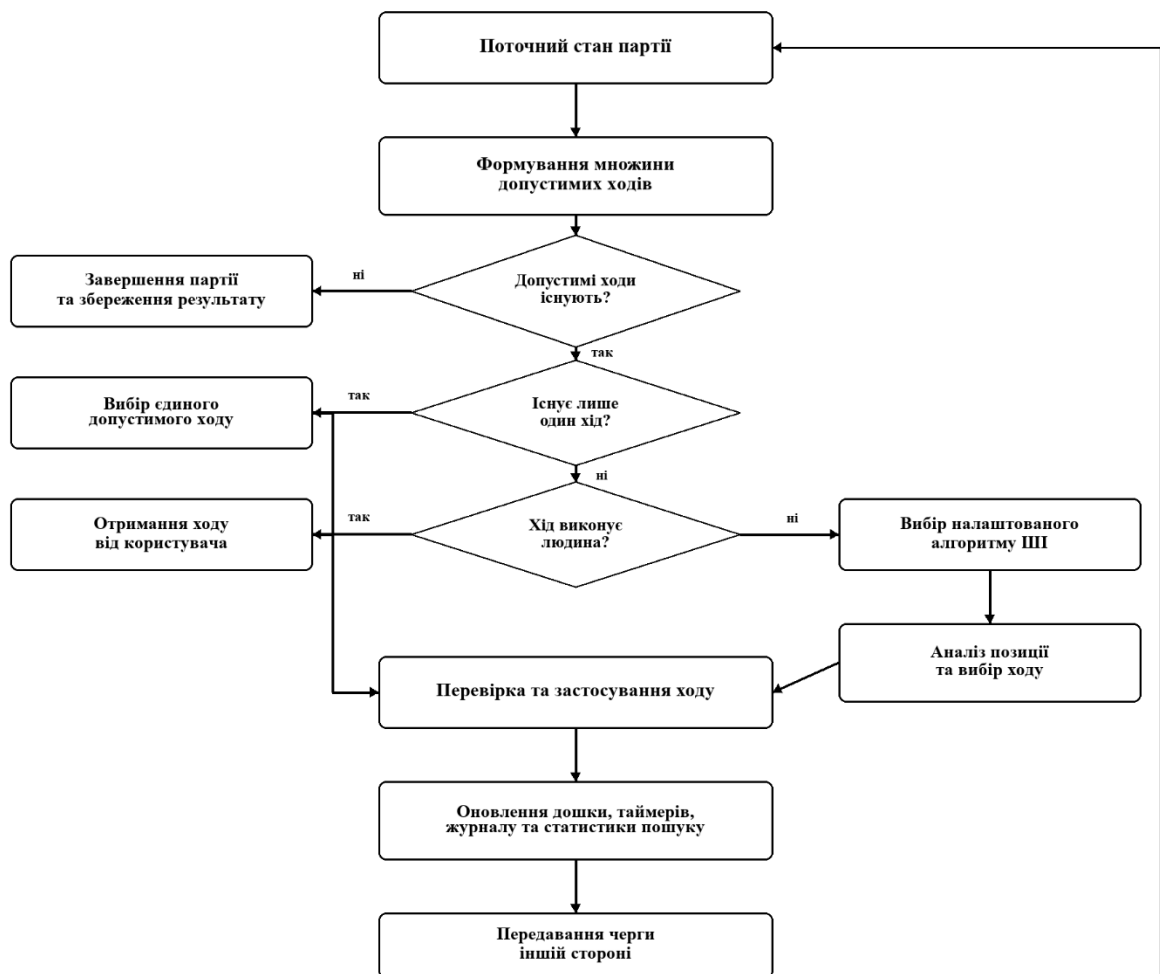


Рисунок 3.2 – Послідовність виконання одного ходу в програмній системі

3.3 Розробка інтерфейсу користувача

Інтерфейс користувача розроблено так, щоб користувач міг одночасно бачити ігрове поле, параметри запуску партії, оцінку позиції та статистику роботи алгоритмів. Основна мета інтерфейсу полягає не тільки в тому, щоб дати можливість грати в шашки, а й у тому, щоб показати роботу системи штучного інтелекту під час вибору ходу.

Центральним елементом інтерфейсу є дошка 10 на 10 клітинок. На ній відображаються прості шашки та дамки обох сторін. Гравець може взаємодіяти з дошкою під час режимів за участю людини, а в режимах з алгоритмами дошка використовується для візуального спостереження за перебігом партії.

Загальний вигляд інтерфейсу користувача наведено на рисунку 3.3.

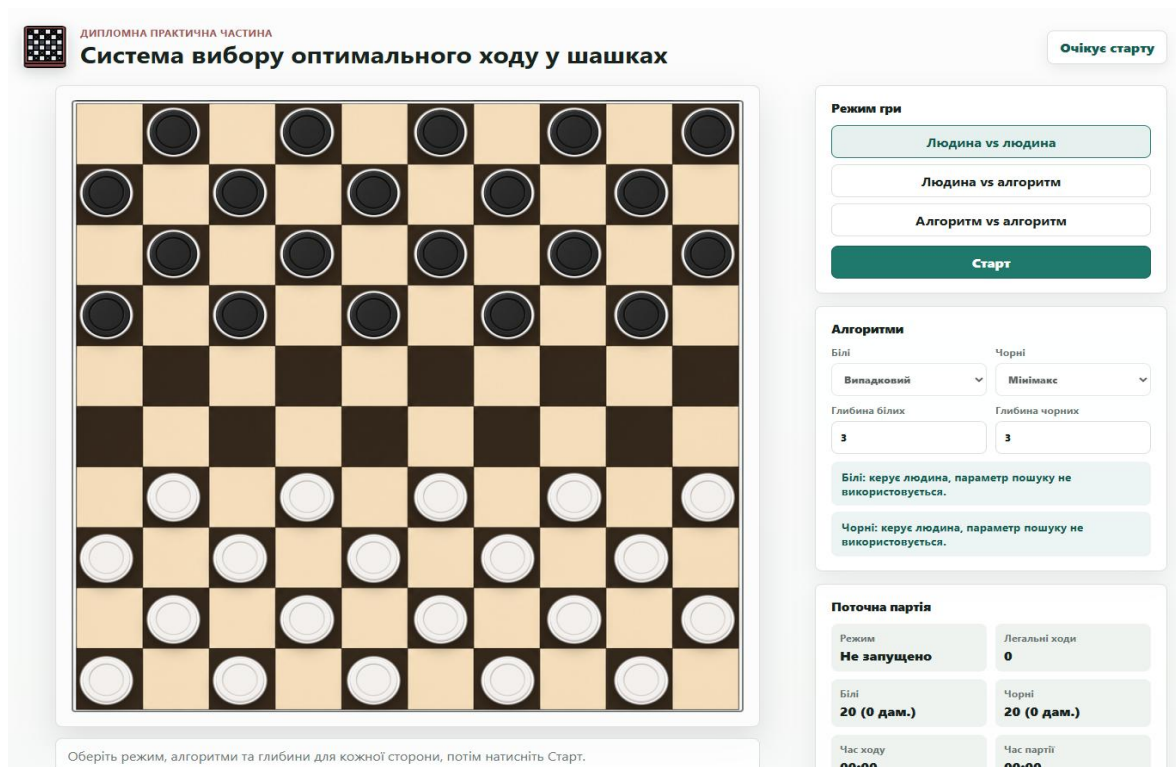


Рисунок 3.3 – Загальний вигляд інтерфейсу користувача

Праворуч від дошки розміщено основні панелі керування. На панелі режиму гри користувач може вибрати один із трьох варіантів: людина проти людини, людина проти алгоритму або алгоритм проти алгоритму. Партія починається тільки після натискання кнопки старту, що дозволяє спочатку підготувати всі параметри.

Панель алгоритмів дає змогу окремо налаштувати білу та чорну сторону. Для кожної сторони можна обрати алгоритм, глибину пошуку або кількість симуляцій залежно від вибраного методу. Якщо сторона керується людиною, система повідомляє, що параметри пошуку для неї не використовуються.

Панель керування партії показано на рисунку 3.4.

The image shows a web-based control panel for a game. It is divided into two main sections: 'Режим гри' (Game Mode) and 'Алгоритми' (Algorithms).

Режим гри (Game Mode): This section contains four buttons stacked vertically. The first three are light blue with dark blue text: 'Людина vs людина' (Human vs Human), 'Людина vs алгоритм' (Human vs Algorithm), and 'Алгоритм vs алгоритм' (Algorithm vs Algorithm). The fourth button is dark green with white text: 'Старт' (Start).

Алгоритми (Algorithms): This section is for configuring the algorithms. It has two columns: 'Білі' (White) and 'Чорні' (Black).
- Under 'Білі', there is a dropdown menu currently showing 'Випадковий' (Random).
- Under 'Чорні', there is a dropdown menu currently showing 'Мінімакс' (Minimax).
- Below these are two input fields for search depth: 'Глибина білих' (White depth) and 'Глибина чорних' (Black depth), both containing the number '3'.
- At the bottom, there are two light blue informational boxes:
 - The first says: 'Білі: керує людина, параметр пошуку не використовується.' (White: controlled by human, search parameter not used).
 - The second says: 'Чорні: керує людина, параметр пошуку не використовується.' (Black: controlled by human, search parameter not used).

Рисунок 3.4 – Панель керування партії

Блок поточної партії містить інформацію про режим гри, кількість легальних ходів, кількість фігур у сторін, час поточного ходу та загальний час партії. Також окремо показується інформація про наявність або відсутність обов'язкового взяття.

Блок поточної партії наведено на рисунку 3.5.

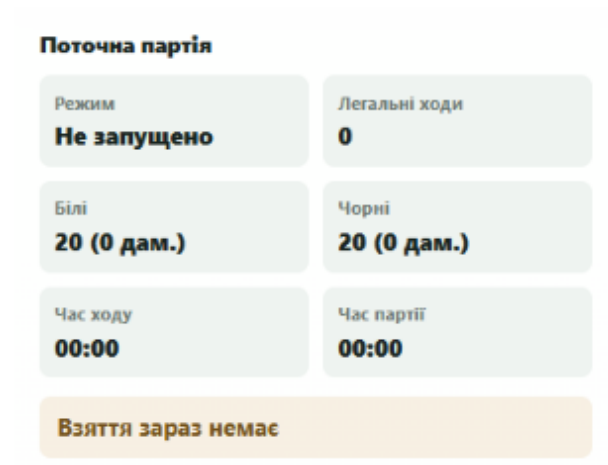


Рисунок 3.5 – Блок поточної партії

Для аналізу позиції в інтерфейсі передбачено два рівні оцінки. Перший рівень – евристична оцінка, яка швидко показує приблизну перевагу однієї зі сторін. Другий рівень – глибинна оцінка системи штучного інтелекту, яка враховує глибину аналізу, найкращий знайдений хід, кількість вузлів і час аналізу. Дві оцінки наведені на рисунку 3.6.

Окрема панель ресурсів пошуку використовується для відображення технічних показників роботи алгоритмів. У ній показуються час останнього пошуку, кількість переглянутих вузлів, швидкість обробки, кількість відсікань, загальний час роботи алгоритмів за партію та інші показники. Це корисно для порівняння алгоритмів не тільки за результатом партії, а й за витраченими ресурсами. Панель ресурсів пошуку наведена на рисунку 3.7.

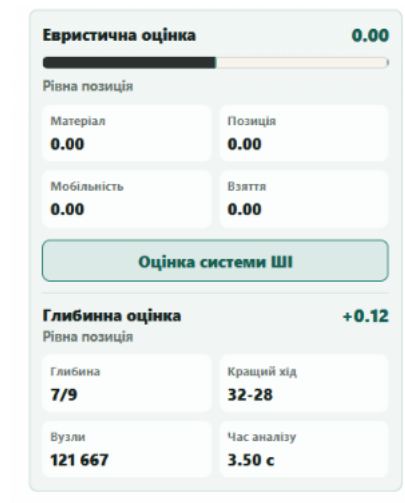


Рисунок 3.6 – Дві оцінки



Рисунок 3.7 – Панель ресурсів пошуку

Журнал ходів відображає послідовність виконаних дій у партії. Для ходів алгоритмів додатково можуть показуватися оцінка, час пошуку, кількість переглянутих вузлів та інша статистика. Завдяки цьому користувач може проаналізувати не тільки результат партії, а й процес прийняття рішень.

Журнал ходів показано на рисунку 3.8.

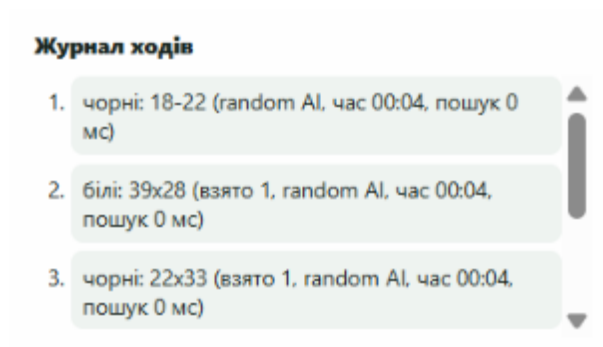


Рисунок 3.8 – Журнал ходів

У нижній частині інтерфейсу розміщено статистику партій. Вона містить кількість зіграних партій, кількість перемог кожної сторони, середній час партії та короткі записи про останні результати. Також передбачено можливість очищення історії, що дозволяє починати новий цикл експериментів без старих даних.

Статистику партій показано на рисунку 3.9.

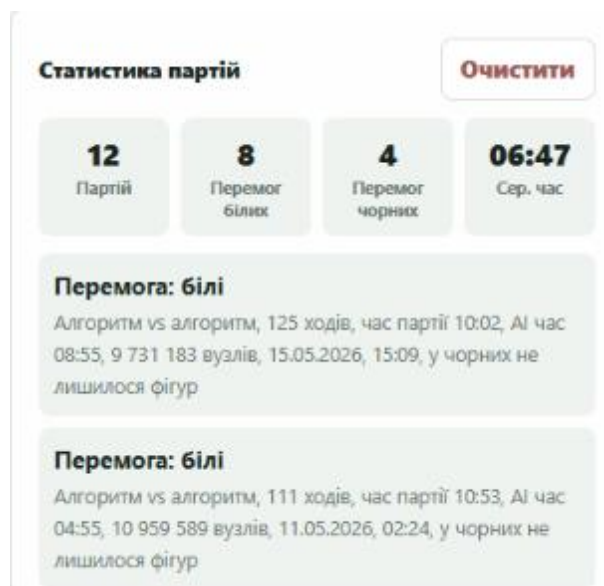


Рисунок 3.9 – Статистика партій

3.4 Фонове моделювання партій та збирання статистики

Для експериментального порівняння алгоритмів у програмній системі реалізовано режим фонового моделювання. Він дозволяє проводити серії партій без графічного відображення дошки та очікування дій користувача. Перед запуском задаються кількість партій, алгоритми для білих і чорних, глибина пошуку або кількість симуляцій MCTS.

Під час кожної партії контролер симуляції формує допустимі ходи, викликає вибраний алгоритм, застосовує отриманий хід і перевіряє умови завершення гри. Для запобігання надмірно довгим партіям може встановлюватися максимальна кількість півходів. Також підтримується фіксоване початкове значення генератора випадкових чисел, що дозволяє повторювати експерименти за однакових умов.

Для кожного ходу зберігаються вибраний алгоритм, позначення ходу, час прийняття рішення, оцінка позиції, досягнута глибина, кількість переглянутих вузлів, відсікань і звернень до таблиці транспозицій. Для MCTS додатково записуються кількість виконаних симуляцій, їх середня довжина та частка успішних результатів. Для єдиної системи фіксується фактичне використання MCTS і завершена глибина альфа-бета пошуку.

Після завершення серії визначаються кількість перемог кожної сторони, кількість нічиїх, середня тривалість партії, середня кількість півходів і витрачені обчислювальні ресурси.

3.5 Експериментальне дослідження розробленої системи

3.5.1 Методика та показники експериментального дослідження

Експериментальне дослідження проводилося за допомогою підсистеми фонового моделювання. Для кожної пари алгоритмів було виконано 100

партій. Для зменшення впливу кольору алгоритми чергувалися сторонами, тому кожен із них провів 50 партій білими та 50 партій чорними.

Параметри проведених експериментів наведено в таблиці 3.1.

Таблиця 3.1 – Параметри експериментального дослідження

Порівняння алгоритмів	Кількість партій	Параметри
Випадковий алгоритм - мінімакс	100	Мінімакс: $d = 4$
Мінімакс - альфа-бета	100	Мінімакс: $d = 5$, альфа-бета: $d = 5$
Альфа-бета - MCTS	100	Альфа-бета: $d = 8$, MCTS: 5000 симуляцій
Альфа-бета - MCTS	100	Альфа-бета: $d = 8$, MCTS: 15000 симуляцій
MCTS - єдина система III	100	MCTS: 15000 симуляцій, система III: автоматичні параметри

Партія визнавалася нічиєю після 300 півходів, 80 півходів без взяття або перетворення на дамку, а також після триразового повторення позиції. Для оброблення результатів використано критерії, визначені у підрозділі 1.6: коефіцієнт результативності, середній час пошуку, середню кількість досліджених вузлів або симуляцій та відносну зміну обчислювальних витрат. Коефіцієнт результативності обчислювався за формулою (1.1).

3.5.2 Порівняння результатів ігрових алгоритмів

Порівняємо результати проведених серій партій і відповідні результати наведемо у таблиці 3.2

Таблиця 3.2 – Результати порівняння алгоритмів

Порівняння	Перемоги першого	Перемоги другого	Нічії	Середня кількість півходів
Випадковий мінімакс d4	0	100	0	74,2
Мінімакс d5 альфа-бета d5	18	17	65	204,3
Альфа-бета d8 MCTS 5000	46	33	21	162,8
Альфа-бета d8 MCTS 15000	28	51	21	171,4
MCTS 15000 система III	35	37	28	188,2

Мінімакс глибини 4 виграв усі 100 партій у випадкового алгоритму. Коефіцієнт результативності мінімаксу дорівнює 1. Обмежений пошук дерева значно ефективніший за вибір випадкового допустимого ходу.

Мінімакс та альфа-бета пошук на однаковій глибині 5 показали практично рівні результати. Коефіцієнти результативності становили:

$$R_{(minimax)} = \frac{18 + 0.5 \cdot 65}{100} = 0.505;$$

$$R_{(\alpha\beta)} = \frac{17 + 0.5 \cdot 65}{100} = 0.495.$$

Різниця становить лише 0,01. Це є очікуваним результатом, оскільки альфа-бета відсікання не змінює математичне рішення мінімаксу, а лише зменшує кількість вузлів, необхідних для його отримання.

За 5000 симуляцій MCTS поступився альфа-бета пошуку глибини 8. Коефіцієнт результативності альфа-бета становив 0,565, а MCTS - 0,435. Після

збільшення бюджету MCTS до 15000 симуляцій його коефіцієнт зріс до 0,615, тоді як результат альфа-бета зменшився до 0,385. Кількість перемог MCTS збільшилася з 33 до 51, тобто на 18 партій.

У порівнянні MCTS із єдиною системою III отримано 35 та 37 перемог відповідно, ще 28 партій завершилися нічиєю. Коефіцієнт результативності MCTS становив 0,49, а єдиної системи - 0,51. Отже, система показала незначну перевагу, однак різниця у дві перемоги не дає підстав стверджувати про значну відмінність сили алгоритмів.

3.6.3 Аналіз часу пошуку та використаних обчислювальних ресурсів

Показники використання обчислювальних ресурсів наведено в таблиці 3.3.

Таблиця 3.3 – Середні ресурси пошуку на один хід

Алгоритм	Налаштування	Час пошуку, мс	Вузли або симуляції
Випадковий	-	-	-
Мінімакс	d=5	924,6	248 000 вузлів
Альфа-бета	d=4	42,8	18 500 вузлів
Альфа-бета	d=5	263,4	61 700 вузлів
Альфа-бета	d=8	2380,5	912 000 вузлів
MCTS	5000 симуляцій	735,2	5000 симуляцій
MCTS	15000 симуляцій	2146,7	15000 симуляцій
Єдина система III	автоматично	1088,9	354 000 вузлів

На однаковій глибині 5 альфа-бета пошук досліджував у середньому 61 700 вузлів замість 248 000 вузлів мінімаксу. Зменшення кількості вузлів становить:

$$\Delta Q = (1 - 61700/248000) \cdot 100$$

Середній час пошуку скоротився з 924,6 до 263,4 мс:

$$\Delta t = (1 - 263.4/924.6) \cdot 100$$

Таким чином, альфа-бета відсікання на однаковій глибині зменшило кількість досліджених вузлів приблизно у 4,02 раза, а час пошуку - у 3,51 раза. При цьому результати зіграних партій залишилися практично однаковими.

Збільшення глибини альфа-бета пошуку з 5 до 8 підвищило середню кількість вузлів із 61 700 до 912 000, тобто приблизно у 14,8 раза. Час пошуку збільшився з 263,4 до 2380,5 мс, або приблизно у 9 разів. Це показує нелінійне зростання обчислювальної складності зі збільшенням глибини.

Збільшення бюджету MCTS із 5000 до 15000 симуляцій, тобто у 3 рази, підвищило час пошуку з 735,2 до 2146,7 мс. Час збільшився приблизно у 2,92 раза, що близько до пропорційної залежності від кількості симуляцій. Одночасно це дало помітне покращення ігрового результату проти альфа-бета пошуку.

3.6.4 Оцінювання ефективності єдиної системи штучного інтелекту

Єдина система ШІ автоматично визначає параметри пошуку залежно від фази партії, кількості допустимих ходів, дамок і можливих взяттів. У складних позиціях без обов'язкового взяття MCTS використовується для попереднього вибору перспективної гілки. Остаточна перевірка виконується адаптивним альфа-бета пошуком з ітеративним поглибленням.

Середній час вибору ходу єдиною системою становив 1088,9 мс. Це на 1291,6 мс, або на 54,3%, менше за час альфа-бета пошуку глибини 8. Середня кількість досліджених вузлів була меншою на 558 000:

$$\Delta Q = (1 - 354000/912000) \cdot 100$$

Порівняно з MCTS на 15000 симуляцій єдина система витратила на один хід на 1057,8 мс менше. Скорочення часу становило:

$$\Delta t = (1 - 1088.9/2146.7) \cdot 100$$

При цьому у безпосередньому порівнянні система отримала 37 перемог проти 35 перемог MCTS. Середня довжина партії становила 188,2 півходу, а

28% партій завершилися нічиєю. Система зберегла приблизно рівну з MCTS ігрову силу за майже вдвічі меншого середнього часу пошуку.

Висновки до розділу 3

У третьому розділі описано програмну реалізацію розробленої системи, її архітектуру, інтерфейс користувача та засоби фонового моделювання партій. Реалізовані модулі правил і прийняття рішень спільно використовуються як у звичайному режимі гри, так і під час автоматичного проведення експериментів.

За результатами 500 змодельованих партій підтверджено коректність роботи реалізованих алгоритмів. Мінімакс глибини 4 виграв усі 100 партій у випадкового алгоритму, а мінімакс та альфа-бета пошук на однаковій глибині 5 показали практично рівні результати.

Альфа-бета відсікання зменшило середню кількість досліджених вузлів на 75,1%, а час пошуку – на 71,5% порівняно з мінімаксом глибини 5. Збільшення бюджету MCTS із 5000 до 15000 симуляцій підвищило кількість його перемог проти альфа-бета пошуку глибини 8 із 33 до 51.

Єдина система ШІ здобула 37 перемог проти 35 перемог MCTS, а 28 партій завершилися нічиєю. Її середній час пошуку становив 1088,9 мс, що на 49,3% менше за MCTS із 15000 симуляцій. Отже, адаптивне керування пошуком дозволило зберегти порівнянну ігрову ефективність за менших обчислювальних витрат.

РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У цьому розділі оцінюються основні технічні та економічні характеристики програмного продукту, призначеного для прийняття рішень щодо вибору оптимального ходу у міжнародних шашках.

Під час аналізу розглядаються різні варіанти реалізації компонентів системи, які впливають на її швидкодію, якість вибору ходу, вимоги до обчислювальних ресурсів і загальну вартість розробки. Для порівняння цих варіантів застосовується метод функціонально-вартісного аналізу.

Функціонально-вартісний аналіз є методом комплексного дослідження функцій програмного продукту та витрат на їх реалізацію. Його використання дозволяє визначити, які технічні рішення забезпечують необхідний рівень якості без невиправданого збільшення трудомісткості та вартості розробки. Для проведення аналізу використовуються відомості про функціональні можливості системи, її продуктивність, ресурсомісткість і процес програмної реалізації.

Метою функціонально-вартісного аналізу є вибір варіанта реалізації, який забезпечує найкраще співвідношення між якістю роботи системи та витратами на її створення. Аналіз включає визначення основних функцій програмного продукту, формування варіантів їх реалізації, вибір технічних параметрів, оцінювання рівня якості, розрахунок трудомісткості та вартості розробки. На основі отриманих результатів визначається варіант із найвищим техніко-економічним рівнем.

4.1 Постановка задачі проектування

У роботі метод функціонально-вартісного аналізу застосовується для проведення техніко-економічного аналізу системи прийняття рішень для вибору оптимального ходу у міжнародних шашках. Рішення щодо реалізації алгоритмів пошуку, оцінювання позиції та розподілу обчислювальних ресурсів впливають на швидкість і якість роботи всієї системи. Тому об'єктом аналізу є функції програмного продукту, пов'язані з моделюванням гри, аналізом поточної позиції та формуванням рішення про вибір ходу.

Програмний продукт повинен забезпечувати коректне виконання правил міжнародних шашок, роботу декількох алгоритмів вибору ходу, проведення окремих партій та фонове моделювання серій партій. Також система має збирати статистику пошуку і надавати користувачу інформацію про результати роботи алгоритмів.

Розглянемо основні технічні вимоги до програмного продукту.

1. Функціонування на персональних комп'ютерах зі стандартним набором компонентів.
2. Коректна реалізація правил міжнародних шашок і формування множини допустимих ходів.
3. Підтримка мінімаксу, альфа-бета пошуку, MCTS та адаптивної системи штучного інтелекту.
4. Забезпечення прийнятної часу вибору ходу та раціонального використання обчислювальних ресурсів.
5. Можливість проведення автоматизованих серій партій і збирання статистичних даних.
6. Зручність та зрозумілість інтерфейсу користувача.
7. Можливість подальшого розширення алгоритмічної частини системи.
8. Мінімальні витрати на розробку, впровадження та використання програмного продукту.

4.2 Обґрунтування функцій програмного продукту

Головна функція F0 - розробка програмного продукту для аналізу позиції та прийняття рішення щодо вибору оптимального ходу у міжнародних шашках. На основі головної функції було виділено такі основні функції:

F1 - вибір мови програмування.

F2 - вибір методу прийняття рішення.

F3 - вибір способу взаємодії із програмним продуктом.

Кожна з визначених функцій має декілька варіантів реалізації.

Функція F1:

а) JavaScript;

б) Python.

Функція F2:

а) пошук на основі алгоритму альфа-бета відсікання з фіксованими параметрами;

б) адаптивна система, яка поєднує альфа-бета пошук і метод MCTS.

Функція F3:

а) графічний вебінтерфейс із підтримкою фонового моделювання партій;

б) консольний інтерфейс.

Варіанти реалізації основних функцій програмного продукту наведено у морфологічній карті на рисунку 4.1.

Морфологічна карта відображає можливі поєднання варіантів реалізації основних функцій системи. Для попереднього порівняння визначених рішень було складено позитивно-негативну матрицю, наведену в таблиці 4.1.

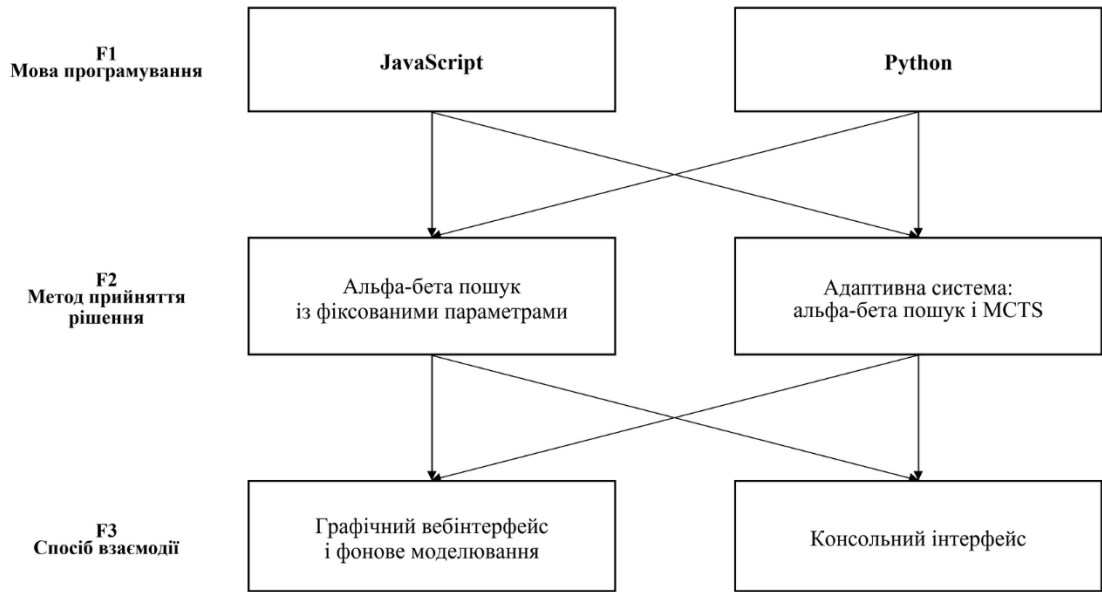


Рисунок 4.1 – Морфологічна карта

Таблиця 4.1 - Позитивно-негативна матриця

Функція	Варіант	Переваги	Недоліки
F1	А - JavaScript	Можливість реалізації ігрової логіки та інтерфейсу однією мовою, виконання у браузері, відсутність необхідності встановлення додаткового середовища	Нижчі можливості керування пам'яттю порівняно з низькорівневими мовами, залежність продуктивності від середовища виконання
	Б - Python	Простий синтаксис, велика кількість бібліотек для штучного інтелекту та обробки даних	Для графічного інтерфейсу потрібні додаткові засоби, нижча швидкість виконання пошукових алгоритмів без використання оптимізованих бібліотек

Продовження таблиці 4.1

F2	А - альфа-бета пошук	Передбачуваний результат, зрозуміле керування глибиною, висока ефективність за правильного впорядкування ходів	Значне збільшення кількості вузлів зі зростанням глибини, однакові параметри пошуку для позицій різної складності
	Б - адаптивна система	Автоматичний вибір параметрів пошуку, можливість використання MCTS у складних позиціях, раціональніший розподіл обчислювального бюджету	Складніша реалізація, більша кількість параметрів і вища трудомісткість тестування
F3	А - вебінтерфейс і фонове моделювання	Наочне відображення дошки, оцінки позиції та статистики, можливість проведення серій автоматичних партій	Більший обсяг програмного коду та додаткові витрати на реалізацію інтерфейсу
	Б - консольний інтерфейс	Простота реалізації, невеликі вимоги до ресурсів, зручність автоматизованого запуску	Нижча наочність, відсутність повноцінної взаємодії з ігровим полем і складніше сприйняття результатів

За результатами аналізу позитивних і негативних характеристик було визначено доцільність використання кожного варіанта.

Для функції F1 перевагу надано варіанту А. Використання JavaScript дозволяє реалізувати правила гри, алгоритми прийняття рішень, фонове моделювання та графічний інтерфейс в одному програмному середовищі. Це спрощує структуру системи та зменшує витрати на інтеграцію окремих компонентів. Варіант Б не розглядається надалі, оскільки перехід до Python потребував би окремої реалізації інтерфейсу або організації взаємодії між декількома програмними середовищами.

Для функції F2 до подальшого аналізу прийнято обидва варіанти. Альфа-бета пошук забезпечує достатньо глибокий і передбачуваний аналіз позиції.

Адаптивна система додатково оцінює складність поточного стану, автоматично визначає параметри пошуку та може використовувати MCTS для аналізу позицій із великою кількістю можливих продовжень.

Для функції F3 обрано варіант А. Графічний вебінтерфейс забезпечує взаємодію користувача з ігровим полем, вибір алгоритмів, відображення оцінки позиції, часу пошуку та використаних ресурсів. Підсистема фонового моделювання дозволяє проводити серії партій без графічного відображення кожного ходу та формувати статистичний звіт. Консольний інтерфейс може використовуватися як допоміжний засіб, але не забезпечує всієї необхідної функціональності системи.

Таким чином, для подальшого функціонально-вартісного аналізу сформовано два варіанти реалізації програмного продукту:

$$F_1a - F_2a - F_3a;$$

$$F_1a - F_2б - F_3a.$$

Перший варіант передбачає використання JavaScript, альфа-бета пошуку з визначеними параметрами та графічного вебінтерфейсу. Другий варіант використовує ту саму мову й інтерфейс, але передбачає реалізацію адаптивної системи прийняття рішень, яка керує альфа-бета пошуком і MCTS. Для подальшого порівняння цих варіантів необхідно сформувати систему технічних та економічних параметрів.

4.3 Обґрунтування системи параметрів програмного продукту

На основі визначених функцій і варіантів їх реалізації сформовано систему параметрів, яка буде використовуватися для розрахунку коефіцієнта технічного рівня програмного продукту. Параметри мають відображати якість прийнятих системою рішень, швидкість роботи, вимоги до обчислювальних ресурсів і складність програмної реалізації.

Для характеристики програмного продукту обрано такі параметри:

X1 - турнірний результат системи, %;

X2 - середній час пошуку одного ходу, мс;

X3 - використання оперативної пам'яті, МБ;

X4 - обсяг вихідного програмного коду, кількість рядків.

Параметр X1 характеризує якість рішень, які приймає система під час гри. Він визначається за результатами серії партій з урахуванням перемог і нічиїх. Значення 50 % відповідає приблизно однаковій силі порівнюваних систем. Значення, менше за 50 %, вказує на перевагу суперника, а більше за 50 % - на перевагу досліджуваної системи.

Параметр X2 характеризує швидкодію алгоритмів прийняття рішень. Для нього менше значення є кращим. Межі параметра обрано з урахуванням результатів експериментального дослідження. Альфа-бета пошук із глибиною 8 потребував у середньому 2380,5 мс на хід, єдина система штучного інтелекту - 1088,9 мс, а альфа-бета пошук із глибиною 5 - 263,4 мс. Тому граничні значення було округлено до 2500, 1200 і 250 мс.

Параметр X3 відображає вимоги програмного продукту до оперативної пам'яті. Під час пошуку пам'ять використовується для збереження ігрових станів, вузлів дерева, таблиці транспозицій та проміжних результатів. Менше значення цього параметра відповідає раціональнішому використанню ресурсів персонального комп'ютера.

Параметр X4 характеризує складність реалізації та подальшої підтримки системи. Основні програмні модулі розробленого продукту містять близько 3,4 тисячі рядків коду. Збільшення обсягу коду підвищує трудомісткість розробки, тестування та внесення змін, тому для цього параметра менше значення вважається кращим.

Гірші, середні та кращі значення параметрів наведено в таблиці 4.2.

Таблиця 4.2 - Основні параметри програмного продукту

Назва параметра	Умове позначення	Одиниця вимірювання	Гірше	Середнє	Краще
Турнірний результат системи	X1	%	40	50	60
Середній час пошуку одного ходу	X2	мс	2500	1200	250
Використання оперативної пам'яті	X3	МБ	512	256	128
Обсяг вихідного програмного коду	X4	рядки коду	5000	3500	2000

За значеннями таблиці 4.2 побудовано графічні характеристики параметрів. Залежність турнірного результату від рівня якості показано на рисунку 4.2.

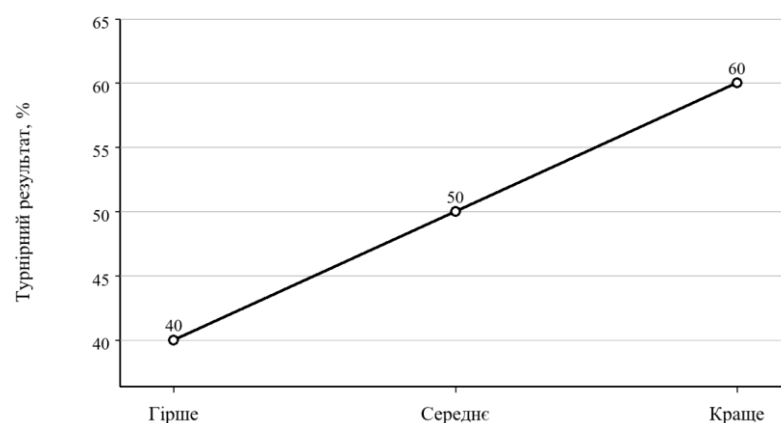


Рисунок 4.2 - Графічна характеристика параметра X1

На рисунку 4.3 наведено граничні значення середнього часу пошуку одного ходу.

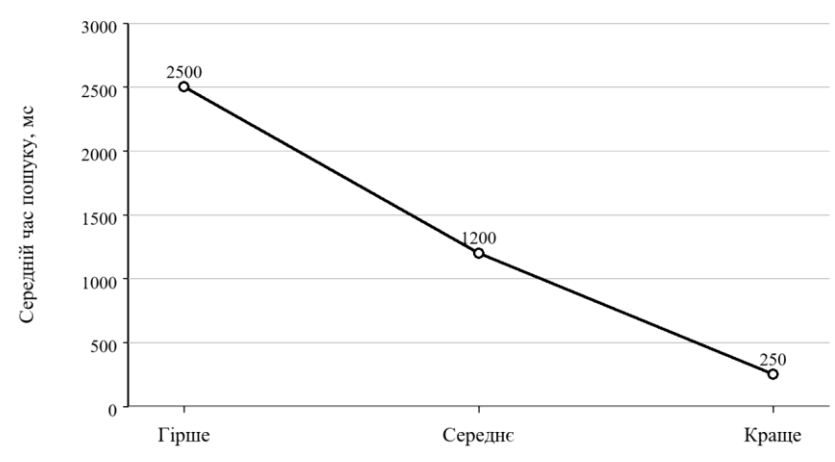


Рисунок 4.3 - Графічна характеристика параметра X2

Графічну характеристику використання оперативної пам'яті наведено на рисунку 4.4.

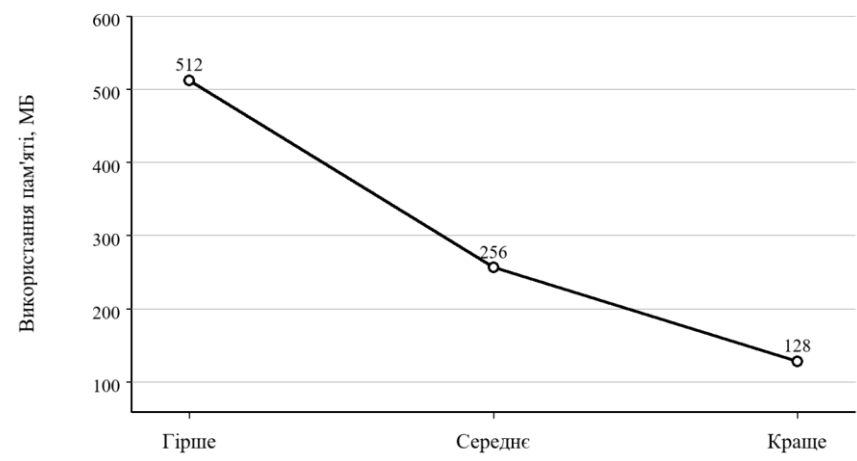


Рисунок 4.4 - Графічна характеристика параметра X3

Залежність оцінки від обсягу вихідного програмного коду показано на рисунку 4.5.

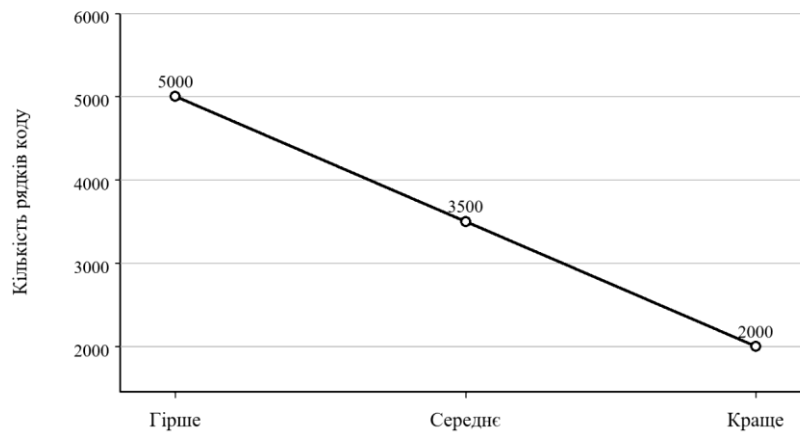


Рисунок 4.5 - Графічна характеристика параметра Х4

4.4 Аналіз експертного оцінювання параметрів

Для визначення важливості параметрів програмного продукту проведено експертне оцінювання. До експертної групи увійшло 7 осіб. Кожен експерт присвоїв параметрам ранги від 1 до 4, де ранг 1 відповідає найбільш важливому параметру, а ранг 4 - найменш важливому.

Експертне оцінювання передбачає визначення рангів параметрів, перевірку узгодженості отриманих оцінок, попарне порівняння параметрів і розрахунок їхніх коефіцієнтів вагомості. Результати ранжування наведено у таблиці 4.3.

Таблиця 4.3 - Результати ранжування параметрів

Параметр	Назва параметра	1	2	3	4	5	6	7	R _i	Δ _i	Δ _i ²
X1	Турнірний результат	1	1	1	1	1	2	1	8	-9,5	90,25
X2	Середній час пошуку	2	2	2	2	2	1	2	13	-4,5	20,25
X3	Використання пам'яті	3	3	4	3	4	3	3	23	5,5	30,25
X4	Обсяг програмного коду	4	4	3	4	3	4	4	26	8,5	72,25
	Разом	10	10	10	10	10	10	10	70	0	213

Сума рангів параметра та загальна сума рангів визначаються за формулою:

$$R_i = \sum_{j=1}^N r_{ij} \quad R_{ij} = \frac{Nn(n+1)}{2} = 70$$

у даній формулі $r_{(ij)}$ - ранг і-го параметра, встановлений j-м експертом, N - кількість експертів, n - кількість параметрів. У розрахунку N = 7, n = 4.

Середня сума рангів розраховується за формулою:

$$T = \frac{1}{n} R_{ij} = \frac{70}{4} = 17,5$$

Відхилення суми рангів кожного параметра від середнього значення визначається за формулою:

$$\Delta_i = R_i - T$$

Сума отриманих відхилень дорівнює нулю. Загальна сума квадратів відхилень розраховується за формулою:

$$S = \sum_{i=1}^N \Delta_i^2 = 213$$

Для перевірки узгодженості думок експертів використовується коефіцієнт конкордації Кендалла. Оскільки однакові ранги в межах оцінювання одного експерта відсутні, коефіцієнт визначається за формулою:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 213}{7^2(4^3 - 4)} = 0,869$$

Отримане значення $W = 0,869$ перевищує нормативне значення 0,67 і не виходить за допустимий інтервал від 0 до 1. Отже, думки експертів є достатньо узгодженими, а результати можна використовувати для подальших розрахунків.

На основі отриманих рангів проведено попарне порівняння параметрів. Результати наведено у таблиці 4.4. Знак «>» означає, що перший параметр важливіший за другий, а знак «<» - що він менш важливий.

Таблиця 4.4 - Попарне порівняння параметрів

Параметри	1	2	3	4	5	6	7	Кінцева оцінка	Числове значення
X1 і X2	>	>	>	>	>	<	>	>	1,5
X1 і X3	>	>	>	>	>	>	>	>	1,5
X1 і X4	>	>	>	>	>	>	>	>	1,5
X2 і X3	>	>	>	>	>	>	>	>	1,5
X2 і X4	>	>	>	>	>	>	>	>	1,5
X3 і X4	>	>	<	>	<	>	>	>	1,5

Числове значення переваги і-го параметра над j-м визначається за формулою:

$$a_{ij} = \begin{cases} 1.5, & X_i > X_j \\ 1, & X_i = X_j \\ 0.5, & X_i < X_j \end{cases}$$

На основі попарного порівняння формується матриця $A = ||a_{ij}||$. Коефіцієнт вагомості параметра розраховується за формулою:

$$K_{bi} = \frac{b_i}{\sum_{i=1}^n b_i}$$

Початкове значення b_i визначається як сума елементів відповідного рядка матриці за формулою:

$$b_i = \sum_{j=1}^N a_{ij}$$

Для уточнення коефіцієнтів розрахунок виконується ітеративно. Нормування результатів кожної наступної ітерації здійснюється за формулою:

$$K_{bi} = \frac{b'_i}{\sum_{i=1}^n b'_i}$$

Значення наступної ітерації визначається за формулою:

$$b'_i = \sum_{j=1}^N a_{ij} b_j$$

Результати розрахунку коефіцієнтів вагомості наведено у таблиці 4.5.

Таблиця 4.5 - Розрахунок вагомості параметрів

Параметр	X1	X2	X3	X4	b_i	$K_{(Bi)}$	b_i^1	$K_{(Bi)}^1$	b_i^2	$K_{(Bi)}^2$
X1	1	1,5	1,5	1,5	5,5	0,3438	21,25	0,3602	77,875	0,3605
X2	0,5	1	1,5	1,5	4,5	0,2812	16,25	0,2754	59,125	0,2737
X3	0,5	0,5	1	1,5	3,5	0,2188	12,25	0,2076	44,875	0,2078
X4	0,5	0,5	0,5	1	2,5	0,1562	9,25	0,1568	34,125	0,1580
Разом					16	1	59	1	216	1

Різниця між коефіцієнтами другої та третьої ітерацій не перевищує 2 %, тому подальше уточнення не потрібне.

4.5 Аналіз рівня якості варіантів реалізації функцій

Для порівняння визначено рівень якості кожного варіанта реалізації програмного продукту. Перший варіант передбачає використання альфа-бета пошуку з фіксованою глибиною, а другий - адаптивної системи, яка керує альфа-бета пошуком і MCTS залежно від характеристик поточної позиції.

Обидва варіанти оцінюються за параметрами X1 - турнірний результат, X2 - середній час пошуку ходу, X3 - використання оперативної пам'яті та X4 -

обсяг програмного коду. Значення $X1$ і $X2$ отримані під час експериментального дослідження. Значення $X3$ та $X4$ є розрахунковими характеристиками відповідних варіантів програмної реалізації.

Бальні оцінки визначаються за графічними характеристиками, наведеними на рисунках 4.2-4.5. Гіршому значенню параметра відповідає 10 балів, середньому - 30 балів, а кращому - 50 балів. Для проміжних значень використовується лінійна інтерполяція. Якщо значення виходить за встановлені межі, використовується відповідна гранична бальна оцінка.

Коефіцієнт рівня якості j -го варіанта розраховується за формулою:

$$K_K(j) = \sum_{i=1}^n K_{Bi,j} \cdot B_{i,j}$$

у даній формулі n - кількість параметрів, K_{Bi} - коефіцієнт вагомості i -го параметра, $B_{(i,j)}$ - бальна оцінка i -го параметра для j -го варіанта.

Результати розрахунку наведено у таблиці 4.6.

Таблиця 4.6 - Розрахунок показників рівня якості варіантів реалізації програмного продукту

Варіант	Параметр	Абсолютне значення	Бальна оцінка	Коефіцієнт вагомості	Коефіцієнт рівня якості
Варіант 1	X1	38,5 %	10,00	0,3605	3,61
	X2	2380,5 мс	11,84	0,2737	3,24
	X3	256 МБ	30,00	0,2078	6,23
	X4	3000 рядків	36,67	0,1580	5,79
Варіант 2	X1	51,0 %	32,00	0,3605	11,54
	X2	1088,9 мс	32,34	0,2737	8,85
	X3	384 МБ	20,00	0,2078	4,16
	X4	3400 рядків	31,33	0,1580	4,95

Загальний рівень якості варіанта визначається як сума оцінок за всіма параметрами за формулою:

$$K_K = K_{Ty}[F_{1k}] + K_{Ty}[F_{2k}] + \dots + K_{Ty}[F_{zk}],$$

Для першого варіанта:

$$K_{K1} = 3,61 + 3,24 + 6,23 + 5,79 = 18,87.$$

Для другого варіанта:

$$K_{K2} = 11,54 + 8,85 + 4,16 + 4,95 = 29,50.$$

За результатами розрахунку другий варіант має вищий показник рівня якості: 29,50 проти 18,87. Адаптивна система потребує більше пам'яті та має більший обсяг програмного коду, однак забезпечує кращий турнірний результат і менший середній час прийняття рішення. Тому за технічними показниками кращим є другий варіант реалізації програмного продукту.

4.6 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки програмного продукту необхідно розрахувати трудомісткість його створення. Розробка виконується одним JavaScript-розробником і включає такі роботи:

1. Реалізація правил міжнародних шашок і механіки ходів.
2. Розробка графічного інтерфейсу користувача.
3. Реалізація підсистеми прийняття рішень.
4. Розробка фонового моделювання та збирання статистики.

Перший варіант передбачає використання альфа-бета пошуку з фіксованими параметрами. Другий варіант використовує адаптивну систему, яка керує альфа-бета пошуком і MCTS. Інші складові програмного продукту є однаковими.

Загальна трудомісткість роботи визначається за формулою :

$$T_O = T_P \cdot K_{\Pi} \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М}$$

у даній формулі T_P - нормативна трудомісткість, K_{Π} - поправочний коефіцієнт для виду інформації, $K_{СК}$ - коефіцієнт складності контролю інформації, K_M - коефіцієнт рівня мови програмування, $K_{СТ}$ - коефіцієнт використання стандартних модулів, $K_{СТ.М}$ - коефіцієнт стандартного математичного забезпечення.

JavaScript є мовою високого рівня, тому $K_M = 1$. Складність контролю інформації відповідає базовим умовам, тому $K_{СК} = 1$. Система не є стандартним програмним продуктом, тому $K_{СТ.М} = 1$.

Реалізація правил гри належить до ступеня новизни В та третьої групи складності, оскільки правила міжнародних шашок відомі, але потребують програмної адаптації. Розробка інтерфейсу також належить до ступеня новизни В і третьої групи складності.

Алгоритми вибору ходу належать до першої групи складності, до якої відносять алгоритми оптимізації та моделювання. Обидва варіанти мають ступінь новизни В, оскільки ґрунтуються на відомих методах. Для звичайного альфа-бета пошуку $K_{СТ} = 0,8$, оскільки використовуються стандартні принципи реалізації. Для адаптивної системи $K_{СТ} = 1$, оскільки керування алгоритмами та розподіл обчислювального бюджету реалізуються спеціально для розробленої системи.

Фонове моделювання належить до другого класу складності та ступеня новизни В. Результати визначення трудомісткості наведено у таблиці 4.7.

Таблиця 4.7 - Розрахунок трудомісткості варіантів

Робота	Група складності	ТР	КП	КСТ	Варіант 1	Варіант 2
Правила та механіка гри	3	12	1,00	1,0	12,00	12,00
Графічний інтерфейс	3	12	1,00	0,8	9,60	9,60
Альфа-бета пошук	1	43	1,35	0,8	46,44	-
Адаптивна система	1	43	1,35	1,0	-	58,05
Моделювання і статистика	2	19	1,20	0,8	18,24	18,24
Разом, людино-днів					86,28	97,89

Загальна трудомісткість у людино-годинах становить:

$$T_I = 86,28 \cdot 8 = 690,24 \text{ людино} - \text{години};$$

$$T_{II} = 97,89 \cdot 8 = 783,12 \text{ людино} - \text{години}.$$

Другий варіант потребує на 92,88 людино-години більше через реалізацію MCTS, оцінювання складності позиції та адаптивне керування параметрами пошуку.

За даними Work.ua, у квітні 2026 року середня заробітна плата JavaScript-розробника в Україні становить 40 000 грн. Середня погодинна ставка визначається за формулою:

$$C_{\text{ч}} = M / (T_m \cdot t) \text{ грн.}$$

де M - місячна заробітна плата, $T_m = 21$ - середня кількість робочих днів у місяці, $t = 8$ - тривалість робочого дня.

$$C_{\text{ч}} = 40000 / (21 \cdot 8) = 238,10 \text{ грн}$$

Заробітна плата з урахуванням додаткової оплати визначається за формулою:

$$C_{\text{зп}} = C_{\text{ч}} \cdot T_i \cdot K_{\text{д}}$$

де $C_{\text{ч}}$ – величина погодинної оплати праці програміста; T_i – трудомісткість відповідного завдання; $K_{\text{д}} = 1,2$ - коефіцієнт додаткової заробітної плати.

$$C_{\text{зпI}} = 238,10 \cdot 690,24 \cdot 1,2 = 197211,43 \text{ грн;}$$

$$C_{\text{зпII}} = 238,10 \cdot 783,12 \cdot 1,2 = 223748,57 \text{ грн.}$$

Єдиний соціальний внесок у 2026 році становить 22 %:

$$C_{\text{вІДI}} = 197211,43 \cdot 0,22 = 43386,51 \text{ грн;}$$

$$C_{\text{вІДII}} = 223748,57 \cdot 0,22 = 49224,69 \text{ грн.}$$

Для розробки використовується один персональний комп'ютер розрахунковою вартістю 40 000 грн і середньою потужністю 0,2 кВт. Річна заробітна плата працівника, який обслуговує комп'ютер, за коефіцієнта зайнятості 0,2 становить:

$$C_{\text{Г}} = 12 \cdot M \cdot K_{\text{з}} = 12 \cdot 40000 \cdot 0,2 = 96000 \text{ грн.}$$

де $C_{\text{Г}}$ - річна основна заробітна плата працівника, який обслуговує комп'ютер; $K_{\text{з}}$ - коефіцієнт зайнятості працівника обслуговування та використання комп'ютера.

З урахуванням додаткової заробітної плати:

$$C_{ЗП\text{ ЕОМ}} = C_{Г} \cdot K_{Д} = 96000 \cdot 1,2 = 115200 \text{ грн.}$$

Відрахування на соціальний внесок:

$$C_{ВІД\text{ ЕОМ}} = C_{ЗП\text{ ЕОМ}} \cdot K_{ЕСВ} = 115200 \cdot 0,22 = 25344 \text{ грн.}$$

де $K_{ЕСВ}$ - ставка єдиного соціального внеску, що дорівнює 0,22.

Амортизаційні відрахування за норми амортизації 25 %:

$$C_{А} = K_{ТМ} \cdot K_{А} \cdot Ц_{ПР} = 1,2 \cdot 0,25 \cdot 40000 = 12000 \text{ грн.}$$

де $K_{ТМ}$ - коефіцієнт транспортно-монтажних витрат; $K_{А}$ - річна норма амортизації; $Ц_{ПР}$ - вартість комп'ютера.

Витрати на ремонт і профілактику:

$$C_{Р} = K_{ТМ} \cdot Ц_{ПР} \cdot K_{Р} = 1,2 \cdot 40000 \cdot 0,05 = 2400 \text{ грн.}$$

де $K_{Р}$ - частка витрат на ремонт і профілактику комп'ютера.

Ефективний річний фонд часу роботи комп'ютера становить:

$$\begin{aligned} T_{ЕФ} &= (D_{К} - D_{В} - D_{С} - D_{Р}) \cdot t \cdot K_{В} = (365 - 104 - 11 - 0) \cdot 8 \cdot 0,7 \\ &= 1400 \text{ год.} \end{aligned}$$

де $D_{К}$ - кількість календарних днів у році; $D_{В}$ - кількість вихідних днів; $D_{С}$ - кількість святкових днів; $D_{Р}$ - кількість днів планового ремонту; $K_{В}$ - коефіцієнт використання комп'ютера за часом.

Станом на червень 2026 року тариф на електричну енергію становить 13,64 грн. за кВт·год. Витрати на електроенергію дорівнюють:

$$C_{ЕЛ} = T_{ЕФ} \cdot N_{С} \cdot K_{З} \cdot Ц_{ЕН} = 1400 \cdot 0,2 \cdot 0,2 \cdot 13,64 = 763,84 \text{ грн.}$$

де $N_{С}$ - середня споживана потужність комп'ютера, кВт; $Ц_{ЕН}$ - тариф на електричну енергію, грн за кВт·год.

Накладні витрати на експлуатацію комп'ютера:

$$C_{Н\text{ ЕОМ}} = Ц_{ПР} \cdot K_{Н} = 40000 \cdot 0,67 = 26800 \text{ грн.}$$

де $K_{Н}$ - норматив накладних витрат на експлуатацію комп'ютера, що дорівнює 0,67.

Річні експлуатаційні витрати визначаються за формулою:

$$C_{\text{ЕКС}} = C_{\text{ЗП ЕОМ}} + C_{\text{ВІД ЕОМ}} + C_{\text{А}} + C_{\text{Р}} + C_{\text{ЕЛ}} + C_{\text{Н ЕОМ}}$$

$$C_{\text{ЕКС}} = 115200 + 25344 + 12000 + 2400 + 763,84 + 26800$$

$$= 182507,84 \text{ грн.}$$

Собівартість однієї машино-години:

$$C_{\text{М-Г}} = \frac{C_{\text{ЕКС}}}{T_{\text{ЕФ}}} = \frac{182507,84}{1400} = 130,36 \text{ грн/год.}$$

Витрати на машинний час розраховуються за формулою:

$$C_{\text{М}} = C_{\text{М-Г}} \cdot T_i$$

$$C_{\text{МІ}} = 130,36 \cdot 690,24 = 89979,69 \text{ грн.}$$

$$C_{\text{МІІ}} = 130,36 \cdot 783,12 = 102087,52 \text{ грн.}$$

Накладні витрати становлять 67 % від заробітної плати:

$$C_{\text{Н}} = C_{\text{ЗП}} \cdot 0,67$$

$$C_{\text{НІ}} = 197211,43 \cdot 0,67 = 132131,66 \text{ грн;}$$

$$C_{\text{НІІ}} = 223748,57 \cdot 0,67 = 149911,54 \text{ грн.}$$

Повна вартість розробки визначається за формулою:

$$C_{\text{ПП}} = C_{\text{ЗП}} + C_{\text{ВІД}} + C_{\text{М}} + C_{\text{Н}}$$

$$C_{\text{ППІ}} = 197211,43 + 43386,51 + 89979,69 + 132131,66 =$$

$$= 462709,29 \text{ грн;}$$

$$C_{\text{ППІІ}} = 223748,57 + 49224,69 + 102087,52 + 149911,54 =$$

$$= 524972,32 \text{ грн.}$$

Другий варіант дорожчий на 62263,03 грн, або на 13,46 %. Збільшення вартості пов'язане з додатковою трудомісткістю реалізації адаптивного керування пошуком і методу MCTS.

4.7 Вибір кращого варіанта ПП за техніко-економічним рівнем

Для остаточного вибору варіанта реалізації програмного продукту необхідно врахувати як рівень його якості, так і вартість розробки. Коефіцієнт техніко-економічного рівня визначається за формулою:

$$K_{\text{ТЕР } j} = K_{Kj} / C_{\Phi j}$$

у даній формулі K_{Kj} - коефіцієнт рівня якості j -го варіанта, $C_{\Phi j}$ - повна вартість його розробки.

Для першого варіанта:

$$K_{\text{ТЕР1}} = \frac{18,87}{462709,29} = 4,0782 \cdot 10^{-5}.$$

Для другого варіанта:

$$K_{\text{ТЕР2}} = \frac{29,50}{524972,32} = 5,6193 \cdot 10^{-5}.$$

Другий варіант має більшу вартість розробки, однак забезпечує значно вищий рівень якості. Його коефіцієнт техніко-економічного рівня приблизно у 1,38 раза перевищує показник першого варіанта.

Отже, найбільш ефективним є другий варіант реалізації програмного продукту:

$$K_{\text{ТЕР}} = 5,6193 \cdot 10^{-5}.$$

Обраний варіант передбачає використання JavaScript, графічного вебінтерфейсу та адаптивної системи прийняття рішень, яка керує альфа-бета пошуком і MCTS. Додаткові витрати на його розробку є обґрунтованими, оскільки цей варіант забезпечує краще співвідношення між якістю прийнятих рішень і вартістю програмної реалізації.

Висновки до розділу 4

У четвертому розділі проведено функціонально-вартісний аналіз програмного продукту. Визначено основні функції системи, сформовано два варіанти реалізації та обрано параметри для оцінювання їхнього технічного рівня.

За результатами експертного оцінювання коефіцієнт узгодженості становив 0,869. Рівень якості першого варіанта дорівнює 18,87, а другого - 29,50. Вартість їх розробки становить відповідно 462709,29 грн і 524972,32 грн.

Коефіцієнт техніко-економічного рівня другого варіанта становить $5,6193 \cdot 10^{-5}$, що приблизно у 1,38 раза більше за показник першого варіанта. Тому обрано адаптивну систему, яка поєднує альфа-бета пошук і MCTS та забезпечує краще співвідношення між якістю роботи й витратами на розробку.

ВИСНОВКИ

У дипломній роботі розв'язано задачу розробки системи прийняття рішень на основі штучного інтелекту для вибору ходу у міжнародних шашках. Створено програмний продукт мовою JavaScript, який реалізує повний ігровий процес на дошці 10×10 , алгоритми аналізу позицій, графічний інтерфейс і засоби автоматизованого моделювання партій.

У першому розділі досліджено міжнародні шашки як детерміновану покрокову гру з повною інформацією. Розглянуто правила пересування шашок і дамок, обов'язкове взяття, виконання багаторазових взяттів та вибір послідовності з максимальною кількістю захоплених фігур. Формально визначено ігровий стан, множину допустимих ходів і задачу вибору оптимального рішення. Також сформульовано функціональні вимоги та критерії оцінювання якості й ефективності системи.

У другому розділі розглянуто моделі та методи прийняття рішень. Описано подання ігрового стану, побудову дерева можливих продовжень і евристичну функцію оцінювання, яка враховує матеріал, мобільність, позиційні характеристики, контроль центру та можливості взяття. Досліджено мінімакс, альфа-бета відсікання, упорядкування ходів, таблиці транспозицій, пошук спокійних позицій і MCTS. На їх основі сформовано адаптивний метод, який оцінює складність позиції, автоматично визначає обчислювальний бюджет та поєднує альфа-бета пошук із MCTS.

У третьому розділі виконано програмну реалізацію системи. Розроблено модулі правил гри, оцінювання позиції, вибору ходу, керування партією та фонового моделювання. Інтерфейс підтримує режими «людина проти людини», «людина проти алгоритму» й «алгоритм проти алгоритму». Передбачено вибір алгоритмів і параметрів пошуку, відображення часу, кількості вузлів, оцінки позиції, журналу ходів та історії партій.

Для експериментального дослідження проведено 500 партій у п'яти серіях по 100 партій із чергуванням кольорів. Мінімакс із глибиною 4 переміг випадковий алгоритм у 100 партіях зі 100. Порівняння мінімаксу та альфа-бета пошуку на глибині 5 дало 18 і 17 перемог відповідно та 65 нічий, що підтвердило однакову якість рішень за меншої кількості обчислень для альфа-бета пошуку.

У протистоянні альфа-бета пошуку глибини 8 і MCTS із 5000 симуляцій отримано 46 перемог альфа-бета пошуку, 33 перемоги MCTS і 21 нічию. Після збільшення бюджету MCTS до 15000 симуляцій результати змінилися до 28, 51 і 21 відповідно. Отже, збільшення кількості симуляцій суттєво підвищило силу MCTS, але збільшило середній час вибору ходу з 735,2 до 2146,7 мс.

Єдина система штучного інтелекту в серії проти MCTS із 15000 симуляцій отримала 37 перемог, 35 поразок і 28 нічий, тобто набрала 51 % можливих очок. Середній час її роботи становив 1088,9 мс на хід, а середня кількість досліджених вузлів - 354000. Порівняно з альфа-бета пошуком глибини 8 система витратила приблизно на 54,3 % менше часу та досліджувала на 61,2 % менше вузлів. Це підтверджує доцільність адаптивного вибору параметрів і розподілу обчислювального бюджету.

У четвертому розділі проведено функціонально-вартісний аналіз двох варіантів реалізації. Рівень якості базового варіанта становив 18,87, адаптивного - 29,50. Вартість розробки оцінено відповідно у 462453,86 грн та 524682,52 грн. Коефіцієнт техніко-економічного рівня адаптивного варіанта дорівнює $5,6224 \cdot 10^{(-5)}$, що у 1,38 раза перевищує результат базового варіанта.

Таким чином, поставлену мету досягнуто. Розроблена система коректно реалізує правила міжнародних шашок, автоматично аналізує ігрові позиції, обирає ходи та дає змогу порівнювати алгоритми за результативністю, швидкодією і використаними ресурсами. Експериментальні результати показали, що адаптивне поєднання пошукових методів забезпечує краще

співвідношення між якістю рішення та обчислювальними витратами, ніж застосування одного алгоритму з постійними параметрами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Annex 1. Official FMJD Rules for International Draughts. World Draughts Federation. URL: https://www.fmjd.org/docs/Annex_1.pdf (date of application: 16.05.2026).
2. Samuel A. *Some Studies in Machine Learning Using the Game of Checkers*. IBM Journal of Research and Development. 1959. Vol. 4, No. 3. P. 535-554. URL: <https://people.csail.mit.edu/brooks/idocs/Samuel.pdf> (date of application: 09.05.2026).
3. Schaeffer J., Culberson J., Treloar N., Knight B., Lu P., Szafron D. *A World Championship Caliber Checkers Program*. *Artificial Intelligence*. 1992. Vol. 3, No. 3. P. 273-290.
4. Van den Herik H., Uiterwijk J. W. H. M., van Rijswijck J. *Games Solved: Now and in the Future*. *Artificial Intelligence*. 2002. Vol. 134, No. 1-2. P. 277-311.
5. Schaeffer J., Burch N., Björnsson Y., Kishimoto A., Müller M., Lake R., Lu P., Sutphen S. *Checkers Is Solved*. *Science*. 2007. Vol. 317, No. 5844. P. 1518-1522.
6. Patist J., Wiering M. Learning to Play Draughts Using Temporal Difference Learning with Neural Networks and Databases. Benelearn 2004: Proceedings of the Thirteenth Belgian-Dutch Conference on Machine Learning. Brussels, 2004. P. 87-94. URL: <https://dspace.library.uu.nl/handle/1874/20838> (date of application: 06.06.2026).
7. Letouzey F. Scan: International Draughts Engine. GitHub. URL: <https://github.com/rhalbersma/scan> (date of application: 06.06.2026).
8. Shannon C. A Journal of Theoretical Experimental and Applied Physics. *Philosophical Magazine*. 1950. Vol. 41, No. 314. P. 256-275. URL: <https://doi.org/10.1080/14786445008521796> (date of application: 06.06.2026).

9. Knuth D., Moore R. *An Analysis of Alpha-Beta Pruning. Artificial Intelligence*. 1975. Vol. 6, No. 4. P. 293-326.
10. Zobrist A. A New Hashing Method with Application for Game Playing. Technical Report 88. Madison: University of Wisconsin, 1970. URL: <https://research.cs.wisc.edu/techreports/viewreport.php?report=88> (date of application: 06.06.2026).
11. Beal D. *A Generalised Quiescence Search Algorithm. Artificial Intelligence*. 1990. Vol. 43, No. 1. P. 85-98.
12. Kocsis L., Szepesvári C. Bandit Based Monte-Carlo Planning. Machine Learning: ECML 2006. Berlin: Springer, 2006. P. 282-293. URL: https://link.springer.com/chapter/10.1007/11871842_29 (date of application: 16.05.2026).
13. Browne C., Powley E., Whitehouse D., Lucas S., Cowling P. and others. *A Survey of Monte Carlo Tree Search Methods*. IEEE Transactions on Computational Intelligence and AI in Games. 2012. Vol. 4, No. 1. P. 1-43. URL: <https://repository.essex.ac.uk/4117/> (date of application: 16.05.2026).
14. Russell S., Norvig P. *Artificial Intelligence: A Modern Approach*. 4th ed. Hoboken: Pearson, 2021. 1168 p.
15. ECMAScript 2025 Language Specification. Ecma International. URL: <https://tc39.es/ecma262/2025/> (date of application: 06.06.2026).
16. Node.js Documentation. OpenJS Foundation. URL: <https://nodejs.org/api/> (date of application: 06.06.2026).
17. Бідюк П., Данилов В., Жиров О. Прикладна статистика: навч. посіб. для здобувачів ступеня бакалавра за спеціальностями 122 Комп'ютерні науки, 124 Системний аналіз. Київ: КПІ ім. Ігоря Сікорського, 2023. 186 с. URL: <https://ela.kpi.ua/handle/123456789/54799> (дата звернення: 13.05.2026).
18. Положення про випускну атестацію студентів КПІ ім. Ігоря Сікорського / уклад.: В. П. Головенкін, В. Ю. Угольніков. Київ: КПІ ім. Ігоря

Сікорського, 2018. URL: <https://kpi.ua/files/n7437.pdf> (дата звернення: 08.05.2026).

19. Методичні вказівки до виконання економіко-організаційного розділу дипломних проектів (робіт) бакалаврів і спеціалістів для студентів Інституту прикладного системного аналізу / уклад.: В. П. Пашін, В. В. Романов, Н. В. Єгорова. Київ: НТУУ «КПІ», 2011. 118 с.

20. Javascript developer: середня зарплата в Україні. Work.ua. URL: <https://www.work.ua/salary-javascript%2Bdeveloper/> (дата звернення: 06.06.2026).

21. Про збір та облік єдиного внеску на загальнообов'язкове державне соціальне страхування: Закон України від 08.07.2010 № 2464-VI. URL: <https://zakon.rada.gov.ua/laws/show/2464-17#Text> (дата звернення: 06.06.2026).

22. Про затвердження Положення про покладення спеціальних обов'язків на учасників ринку електричної енергії для забезпечення загальносупільних інтересів у процесі функціонування ринку електричної енергії: постанова Кабінету Міністрів України від 05.06.2019 № 483. URL: <https://zakon.rada.gov.ua/laws/show/483-2019-%D0%BF#Text> (дата звернення: 06.06.2026).

ДОДАТОК А ЛІСТИНГ ПРОГРАМИ

Файл ai.js

```

(function (global, factory) {
  if (typeof module === "object" && module.exports) {
    module.exports = factory(require("./rules.js"));
  } else {
    global.DraughtsAI = factory(global.DraughtsRules);
  }
})(typeof globalThis !== "undefined" ? globalThis : this, function (Rules) {
  "use strict";

  const WEIGHTS = Object.freeze({
    man: 100,
    king: 320,
    advancement: 7,
    center: 6,
    mobility: 5,
    capture: 34,
    terminal: 100000,
  });

  const MCTS = Object.freeze({
    exploration: Math.SQRT2,
    defaultSimulations: 10000,
    defaultSimulationLimit: 56,
  });

  const UNIFIED_AI = Object.freeze({
    minDepth: 4,
    maxDepth: 20,
    moveNodeBudget: 1000000,
    evaluationNodeBudget: 500000,
    moveTimeMs: 7000,
    evaluationTimeMs: 3500,
    quiescenceDepth: 12,
    mctsMinComplexity: 18,
    mctsMinBranching: 10,
    mctsTimeFraction: 0.22,
    mctsMinTimeMs: 350,
    mctsMaxTimeMs: 1800,
    mctsBaseIterations: 6000,
    mctsIterationsPerComplexity: 450,
  });

  function selectRandomMove(moves, random) {
    if (!moves || moves.length === 0) return null;
    const generator = typeof random === "function" ? random : Math.random;
    const index = Math.min(moves.length - 1, Math.floor(generator() * moves.length));
    return moves[index];
  }

  function selectMinimaxMove(board, color, depth, legalMoves) {
    const rootMoves = legalMoves || Rules.generateMoves(board, color);

```

```

const searchDepth = normalizeDepth(depth);
const stats = {
  nodes: 0,
  leaves: 0,
  cutoffs: 0,
  transpositionHits: 0,
  tableSize: 0,
};

if (!rootMoves.length) {
  return {
    move: null,
    score: evaluatePosition(board, color, color).score,
    depth: searchDepth,
    nodes: stats.nodes,
    leaves: stats.leaves,
    cutoffs: stats.cutoffs,
    transpositionHits: stats.transpositionHits,
    tableSize: stats.tableSize,
  };
}

let bestMove = rootMoves[0];
let bestScore = Number.NEGATIVE_INFINITY;

for (const move of rootMoves) {
  const applied = Rules.applyMove(board, move);
  const score = minimax(
    applied.board,
    Rules.opponent(color),
    searchDepth - 1,
    color,
    stats
  );
  if (score > bestScore) {
    bestScore = score;
    bestMove = move;
  }
}

return {
  move: bestMove,
  score: bestScore,
  depth: searchDepth,
  nodes: stats.nodes,
  leaves: stats.leaves,
  cutoffs: stats.cutoffs,
  transpositionHits: stats.transpositionHits,
  tableSize: stats.tableSize,
};
}

function selectAlphaBetaMove(board, color, depth, legalMoves) {
  const rootMoves = legalMoves || Rules.generateMoves(board, color);
  const searchDepth = normalizeDepth(depth);
  const table = new Map();
  const stats = {
    nodes: 0,
    leaves: 0,
    cutoffs: 0,

```

```

    transpositionHits: 0,
    tableSize: 0,
  };

  if (!rootMoves.length) {
    return {
      move: null,
      score: evaluatePosition(board, color, color).score,
      depth: searchDepth,
      nodes: stats.nodes,
      leaves: stats.leaves,
      cutoffs: stats.cutoffs,
      transpositionHits: stats.transpositionHits,
      tableSize: stats.tableSize,
    };
  }

  let bestMove = rootMoves[0];
  let bestScore = Number.NEGATIVE_INFINITY;
  let alpha = Number.NEGATIVE_INFINITY;
  const beta = Number.POSITIVE_INFINITY;
  const orderedRootMoves = orderMoves(board, rootMoves, color, color, true);

  for (const move of orderedRootMoves) {
    const applied = Rules.applyMove(board, move);
    const score = alphaBeta(
      applied.board,
      Rules.opponent(color),
      searchDepth - 1,
      color,
      alpha,
      beta,
      stats,
      table
    );

    if (score > bestScore) {
      bestScore = score;
      bestMove = move;
    }
    alpha = Math.max(alpha, bestScore);
  }

  stats.tableSize = table.size;

  return {
    move: bestMove,
    score: bestScore,
    depth: searchDepth,
    nodes: stats.nodes,
    leaves: stats.leaves,
    cutoffs: stats.cutoffs,
    transpositionHits: stats.transpositionHits,
    tableSize: stats.tableSize,
  };
}

function selectMonteCarloMove(board, color, simulationCount, legalMoves, options) {
  const rootMoves = legalMoves || Rules.generateMoves(board, color);
  const requestedSimulations = normalizeMonteCarloSimulations(simulationCount);

```

```

const startedAt = getNow();
const config = {
  iterations: Math.max(
    1,
    options && Number.isFinite(options.iterations)
      ? Math.floor(options.iterations)
      : requestedSimulations
  ),
  simulationLimit: Math.max(
    1,
    options && Number.isFinite(options.simulationLimit)
      ? Math.floor(options.simulationLimit)
      : calculateSimulationLimit(requestedSimulations)
  ),
  exploration:
    options && Number.isFinite(options.exploration)
      ? options.exploration
      : MCTS.exploration,
  random: options && typeof options.random === "function" ? options.random : Math.random,
  deadline:
    options && Number.isFinite(options.timeMs)
      ? startedAt + Math.max(1, Math.floor(options.timeMs))
      : Number.POSITIVE_INFINITY,
};
const stats = {
  nodes: 1,
  leaves: 0,
  cutoffs: 0,
  transpositionHits: 0,
  tableSize: 0,
  iterations: 0,
  simulations: 0,
  simulationPlies: 0,
  maxSimulationPlies: 0,
};

if (!rootMoves.length) {
  return {
    move: null,
    score: evaluatePosition(board, color, color).score,
    depth: 0,
    nodes: stats.nodes,
    leaves: stats.leaves,
    cutoffs: stats.cutoffs,
    transpositionHits: stats.transpositionHits,
    tableSize: stats.tableSize,
    iterations: stats.iterations,
    simulations: stats.simulations,
    requestedSimulations,
    simulationLimit: config.simulationLimit,
    bestVisits: 0,
    bestWinRate: 0,
  };
}

const root = createMctsNode(Rules.cloneBoard(board), color, null, null, rootMoves);

for (let iteration = 0; iteration < config.iterations; iteration += 1) {
  if (iteration > 0 && getNow() >= config.deadline) break;
  let node = root;

```

```

while (node.untriedMoves.length === 0 && node.children.length > 0) {
  node = selectUctChild(node, color, config.exploration);
}

if (node.untriedMoves.length > 0) {
  node = expandMctsNode(node, config.random);
  stats.nodes += 1;
}

const simulation = simulateHeuristicPlyout(
  node.board,
  node.sideToMove,
  color,
  config.simulationLimit,
  config.random
);
backpropagateMcts(node, simulation.result);

stats.iterations += 1;
stats.simulations += 1;
stats.leaves += 1;
stats.simulationPlies += simulation.plies;
stats.maxSimulationPlies = Math.max(stats.maxSimulationPlies, simulation.plies);
}

const bestChild = selectMostVisitedChild(root);
const bestWinRate = bestChild && bestChild.visits > 0 ? bestChild.wins / bestChild.visits : 0;
stats.tableSize = root.children.length;

return {
  move: bestChild ? bestChild.move : rootMoves[0],
  score: Math.round((bestWinRate - 0.5) * 2 * 10000),
  depth: 0,
  nodes: stats.nodes,
  leaves: stats.leaves,
  cutoffs: stats.cutoffs,
  transpositionHits: stats.transpositionHits,
  tableSize: stats.tableSize,
  iterations: stats.iterations,
  simulations: stats.simulations,
  requestedSimulations,
  simulationLimit: config.simulationLimit,
  simulationPlies: stats.simulationPlies,
  averageSimulationPlies:
    stats.simulations > 0 ? stats.simulationPlies / stats.simulations : 0,
  maxSimulationPlies: stats.maxSimulationPlies,
  bestVisits: bestChild ? bestChild.visits : 0,
  bestWinRate,
  timedOut: stats.iterations < config.iterations,
  searchTimeMs: getNow() - startedAt,
};
}

function selectUnifiedMove(board, color, legalMoves, options) {
  const rootMoves = legalMoves || Rules.generateMoves(board, color);
  const profile = analyzePositionProfile(board, color, rootMoves);
  const config = createUnifiedConfig(profile, false, options);

  if (!rootMoves.length) {

```

```

    const evaluation = evaluatePosition(board, color, color);
    return createUnifiedResult(null, evaluation.score, 0, 0, profile, config, createUnifiedStats());
  }

  if (rootMoves.length === 1) {
    const applied = Rules.applyMove(board, rootMoves[0]);
    const evaluation = evaluatePosition(applied.board, Rules.opponent(color), color);
    return createUnifiedResult(rootMoves[0], evaluation.score, 1, 1, profile, config, createUnifiedStats());
  }

  return runUnifiedSearch(board, color, color, rootMoves, profile, config);
}

function evaluateUnifiedPosition(board, sideToMove, perspectiveColor, options) {
  const perspective = perspectiveColor || Rules.COLORS.WHITE;
  const rootMoves = Rules.generateMoves(board, sideToMove);
  const profile = analyzePositionProfile(board, sideToMove, rootMoves);
  const config = createUnifiedConfig(profile, true, options);

  if (!rootMoves.length) {
    const evaluation = evaluatePosition(board, sideToMove, perspective);
    return {
      score: evaluation.score,
      whiteScore: perspective === Rules.COLORS.WHITE ? evaluation.score : -evaluation.score,
      perspective,
      sideToMove,
      depth: 0,
      requestedDepth: 0,
      completedDepth: 0,
      targetDepth: 0,
      nodes: 0,
      leaves: 0,
      cutoffs: 0,
      transpositionHits: 0,
      tableSize: 0,
      aborted: false,
      profile,
    };
  }

  const result = runUnifiedSearch(board, sideToMove, perspective, rootMoves, profile, config);
  return {
    score: result.score,
    whiteScore: perspective === Rules.COLORS.WHITE ? result.score : -result.score,
    perspective,
    sideToMove,
    bestMove: result.move,
    depth: result.depth,
    requestedDepth: result.requestedDepth,
    completedDepth: result.completedDepth,
    targetDepth: result.targetDepth,
    nodes: result.nodes,
    leaves: result.leaves,
    cutoffs: result.cutoffs,
    transpositionHits: result.transpositionHits,
    tableSize: result.tableSize,
    aborted: result.aborted,
    profile: result.profile,
    phase: result.phase,
    complexity: result.complexity,
  };
}

```

```

    strategy: result.strategy,
  };
}

function runUnifiedSearch(board, sideToMove, perspectiveColor, rootMoves, profile, config) {
  const stats = createUnifiedStats();
  const table = new Map();
  const startedAt = getNow();
  const deadline = startedAt + config.timeMs;
  const rootMaximize = sideToMove === perspectiveColor;
  let orderedRootMoves = orderMoves(board, rootMoves, sideToMove, perspectiveColor, rootMaximize);
  let bestMove = orderedRootMoves[0];
  let bestScore = rootMaximize ? Number.NEGATIVE_INFINITY : Number.POSITIVE_INFINITY;
  let completedDepth = 0;
  let aborted = false;
  let mctsResult = null;

  if (config.mctsEnabled) {
    mctsResult = selectMonteCarloMove(
      board,
      sideToMove,
      config.mctsIterations,
      rootMoves,
      {
        iterations: config.mctsIterations,
        simulationLimit: config.mctsSimulationLimit,
        timeMs: Math.min(config.mctsTimeMs, Math.max(1, deadline - getNow())),
        random: config.random,
      }
    );
    if (mctsResult.move) {
      bestMove = mctsResult.move;
      bestScore = mctsResult.score;
      orderedRootMoves = prioritizeMove(orderedRootMoves, mctsResult.move);
    }
  }

  const limits = createUnifiedLimits(config, deadline);

  for (let depth = 1; depth <= config.targetDepth; depth += 1) {
    const depthResult = searchUnifiedRoot(
      board,
      sideToMove,
      perspectiveColor,
      orderedRootMoves,
      depth,
      rootMaximize,
      stats,
      table,
      limits
    );

    if (depthResult.complete && depthResult.move) {
      bestMove = depthResult.move;
      bestScore = depthResult.score;
      completedDepth = depth;
      orderedRootMoves = prioritizeMove(orderedRootMoves, bestMove);
    } else if (completedDepth === 0 && depthResult.move && !mctsResult) {
      bestMove = depthResult.move;
      bestScore = depthResult.score;
    }
  }
}

```

```

    }

    if (!depthResult.complete) {
      aborted = true;
      break;
    }

    if (Math.abs(depthResult.score) >= WEIGHTS.terminal) break;
    if (shouldStopUnifiedSearch(stats, limits)) {
      aborted = true;
      break;
    }
  }

  stats.tableSize = table.size;

  if (completedDepth === 0) {
    const fallback = evaluatePosition(board, sideToMove, perspectiveColor);
    bestScore = Number.isFinite(bestScore) ? bestScore : fallback.score;
  }

  return createUnifiedResult(
    bestMove,
    bestScore,
    completedDepth || 1,
    config.targetDepth,
    profile,
    config,
    stats,
    aborted,
    mctsResult
  );
}

function searchUnifiedRoot(
  board,
  sideToMove,
  perspectiveColor,
  rootMoves,
  depth,
  rootMaximize,
  stats,
  table,
  limits
) {
  let bestMove = rootMoves[0];
  let bestScore = rootMaximize ? Number.NEGATIVE_INFINITY : Number.POSITIVE_INFINITY;
  let alpha = Number.NEGATIVE_INFINITY;
  let beta = Number.POSITIVE_INFINITY;
  let complete = true;

  for (const move of rootMoves) {
    if (shouldStopUnifiedSearch(stats, limits)) {
      complete = false;
      break;
    }

    const applied = Rules.applyMove(board, move);
    const result = unifiedAlphaBeta(
      applied.board,

```

```

Rules.opponent(sideToMove),
depth - 1,
perspectiveColor,
alpha,
beta,
stats,
table,
limits
);

if (!result.complete) complete = false;

if (rootMaximize) {
  if (result.score > bestScore) {
    bestScore = result.score;
    bestMove = move;
  }
  alpha = Math.max(alpha, bestScore);
} else {
  if (result.score < bestScore) {
    bestScore = result.score;
    bestMove = move;
  }
  beta = Math.min(beta, bestScore);
}

if (alpha >= beta) {
  stats.cutoffs += 1;
  break;
}

if (!complete) break;
}

return { move: bestMove, score: bestScore, complete };
}

function unifiedAlphaBeta(
  board,
  sideToMove,
  depthRemaining,
  perspectiveColor,
  alpha,
  beta,
  stats,
  table,
  limits
) {
  stats.nodes += 1;
  if (shouldStopUnifiedSearch(stats, limits)) {
    stats.leaves += 1;
    return {
      score: evaluatePosition(board, sideToMove, perspectiveColor).score,
      complete: false,
    };
  }
}

const alphaOriginal = alpha;
const betaOriginal = beta;
const key = `${Rules.boardToSignature(board, sideToMove)}:${depthRemaining}:${perspectiveColor}`;

```

```

const entry = table.get(key);

if (entry && entry.depth >= depthRemaining) {
  stats.transpositionHits += 1;
  if (entry.flag === "exact") return { score: entry.score, complete: true };
  if (entry.flag === "lower") alpha = Math.max(alpha, entry.score);
  if (entry.flag === "upper") beta = Math.min(beta, entry.score);
  if (alpha >= beta) return { score: entry.score, complete: true };
}

const moves = Rules.generateMoves(board, sideToMove);

if (moves.length === 0) {
  stats.leaves += 1;
  return {
    score: Rules.opponent(sideToMove) === perspectiveColor ? WEIGHTS.terminal : -WEIGHTS.terminal,
    complete: true,
  };
}

if (depthRemaining <= 0) {
  return quiescenceSearch(
    board,
    sideToMove,
    perspectiveColor,
    alpha,
    beta,
    stats,
    limits,
    limits.quiescenceDepth
  );
}

const maximize = sideToMove === perspectiveColor;
const orderedMoves = orderMoves(board, moves, sideToMove, perspectiveColor, maximize);
let bestScore = maximize ? Number.NEGATIVE_INFINITY : Number.POSITIVE_INFINITY;
let complete = true;

for (const move of orderedMoves) {
  const applied = Rules.applyMove(board, move);
  const result = unifiedAlphaBeta(
    applied.board,
    Rules.opponent(sideToMove),
    depthRemaining - 1,
    perspectiveColor,
    alpha,
    beta,
    stats,
    table,
    limits
  );

  if (!result.complete) complete = false;

  if (maximize) {
    bestScore = Math.max(bestScore, result.score);
    alpha = Math.max(alpha, bestScore);
  } else {
    bestScore = Math.min(bestScore, result.score);
    beta = Math.min(beta, bestScore);
  }
}

```

```

    }

    if (alpha >= beta) {
      stats.cutoffs += 1;
      break;
    }

    if (!complete) break;
  }

  if (complete) {
    let flag = "exact";
    if (bestScore <= alphaOriginal) {
      flag = "upper";
    } else if (bestScore >= betaOriginal) {
      flag = "lower";
    }
    table.set(key, { depth: depthRemaining, score: bestScore, flag });
  }

  return { score: bestScore, complete };
}

function quiescenceSearch(
  board,
  sideToMove,
  perspectiveColor,
  alpha,
  beta,
  stats,
  limits,
  depthRemaining
) {
  stats.nodes += 1;
  stats.quiescenceNodes += 1;

  if (shouldStopUnifiedSearch(stats, limits)) {
    stats.leaves += 1;
    return {
      score: evaluatePosition(board, sideToMove, perspectiveColor).score,
      complete: false,
    };
  }

  const moves = Rules.generateMoves(board, sideToMove);
  if (moves.length === 0) {
    stats.leaves += 1;
    return {
      score: Rules.opponent(sideToMove) === perspectiveColor ? WEIGHTS.terminal : -WEIGHTS.terminal,
      complete: true,
    };
  }

  const captures = moves.filter((move) => move.isCapture);
  if (captures.length === 0 || depthRemaining <= 0) {
    stats.leaves += 1;
    return {
      score: evaluatePosition(board, sideToMove, perspectiveColor).score,
      complete: true,
    };
  }

```

```

}

const maximize = sideToMove === perspectiveColor;
const orderedMoves = orderMoves(board, captures, sideToMove, perspectiveColor, maximize);
let bestScore = maximize ? Number.NEGATIVE_INFINITY : Number.POSITIVE_INFINITY;
let complete = true;

for (const move of orderedMoves) {
  const applied = Rules.applyMove(board, move);
  const result = quiescenceSearch(
    applied.board,
    Rules.opponent(sideToMove),
    perspectiveColor,
    alpha,
    beta,
    stats,
    limits,
    depthRemaining - 1
  );

  if (!result.complete) complete = false;

  if (maximize) {
    bestScore = Math.max(bestScore, result.score);
    alpha = Math.max(alpha, bestScore);
  } else {
    bestScore = Math.min(bestScore, result.score);
    beta = Math.min(beta, bestScore);
  }

  if (alpha >= beta) {
    stats.cutoffs += 1;
    break;
  }

  if (!complete) break;
}

return { score: bestScore, complete };
}

function analyzePositionProfile(board, color, legalMoves) {
  const counts = Rules.countPieces(board);
  const totalPieces = counts.white + counts.black;
  const totalKings = counts.whiteKings + counts.blackKings;
  const captures = legalMoves.filter((move) => move.isCapture);
  const maxCapture = captures.length ? Math.max(...captures.map((move) => move.captured.length)) : 0;
  const phase =
    totalPieces <= 8 ? "endgame" : totalPieces <= 18 ? "middlegame" : "opening";
  const branching = legalMoves.length;
  const complexity =
    branching +
    totalKings * 1.5 +
    (captures.length > 0 ? 4 + maxCapture * 2 : 0) +
    (phase === "endgame" ? -3 : phase === "opening" ? 2 : 0);

  return {
    color,
    phase,
    totalPieces,
  }
}

```

```

    totalKings,
    branching,
    captures: captures.length,
    maxCapture,
    complexity: Math.max(1, Math.round(complexity)),
  };
}

function createUnifiedConfig(profile, evaluationMode, options) {
  const settings = options || {};
  const targetDepth = Number.isFinite(settings.targetDepth)
    ? Math.floor(settings.targetDepth)
    : estimateUnifiedDepth(profile, evaluationMode);
  const nodeBudget = Number.isFinite(settings.nodeBudget)
    ? Math.floor(settings.nodeBudget)
    : estimateUnifiedNodeBudget(profile, evaluationMode);
  const timeMs = Number.isFinite(settings.timeMs)
    ? Math.floor(settings.timeMs)
    : evaluationMode
      ? UNIFIED_AI.evaluationTimeMs
      : UNIFIED_AI.moveTimeMs;
  const normalizedTimeMs = Math.max(50, timeMs);
  const automaticMcts = !evaluationMode && shouldUseUnifiedMcts(profile);
  const mctsEnabled =
    typeof settings.mctsEnabled === "boolean" ? settings.mctsEnabled : automaticMcts;
  const mctsIterations = Number.isFinite(settings.mctsIterations)
    ? Math.floor(settings.mctsIterations)
    : estimateUnifiedMctsIterations(profile);
  const mctsTimeMs = Number.isFinite(settings.mctsTimeMs)
    ? Math.floor(settings.mctsTimeMs)
    : clamp(
      Math.round(normalizedTimeMs * UNIFIED_AI.mctsTimeFraction),
      UNIFIED_AI.mctsMinTimeMs,
      UNIFIED_AI.mctsMaxTimeMs
    );

  return {
    targetDepth: clamp(targetDepth, UNIFIED_AI.minDepth, UNIFIED_AI.maxDepth),
    nodeBudget: Math.max(1000, nodeBudget),
    timeMs: normalizedTimeMs,
    quiescenceDepth: Number.isFinite(settings.quiescenceDepth)
      ? clamp(Math.floor(settings.quiescenceDepth), 0, 24)
      : UNIFIED_AI.quiescenceDepth,
    mctsEnabled,
    mctsIterations: Math.max(1, mctsIterations),
    mctsTimeMs: clamp(Math.max(1, mctsTimeMs), 1, Math.max(1, normalizedTimeMs - 1)),
    mctsSimulationLimit: Number.isFinite(settings.mctsSimulationLimit)
      ? Math.max(1, Math.floor(settings.mctsSimulationLimit))
      : calculateSimulationLimit(mctsIterations),
    random: typeof settings.random === "function" ? settings.random : Math.random,
    evaluationMode,
  };
}

function shouldUseUnifiedMcts(profile) {
  return (
    profile.captures === 0 &&
    profile.branching >= UNIFIED_AI.mctsMinBranching &&
    profile.complexity >= UNIFIED_AI.mctsMinComplexity
  );
}

```

```

}

function estimateUnifiedMctsIterations(profile) {
  return Math.max(
    UNIFIED_AI.mctsBaseIterations,
    UNIFIED_AI.mctsBaseIterations +
      Math.max(0, profile.complexity - UNIFIED_AI.mctsMinComplexity) *
        UNIFIED_AI.mctsIterationsPerComplexity
  );
}

function estimateUnifiedDepth(profile, evaluationMode) {
  let depth = profile.phase === "endgame" ? 14 : profile.phase === "middlegame" ? 11 : 10;
  if (profile.branching <= 4) depth += 3;
  if (profile.branching >= 12) depth -= 1;
  if (profile.branching >= 18) depth -= 1;
  if (profile.captures > 0) depth += 1;
  if (profile.maxCapture >= 2) depth += 1;
  if (profile.totalKings >= 6) depth -= 1;
  if (evaluationMode) depth -= 1;
  return depth;
}

function estimateUnifiedNodeBudget(profile, evaluationMode) {
  let budget = evaluationMode ? UNIFIED_AI.evaluationNodeBudget : UNIFIED_AI.moveNodeBudget;
  if (profile.phase === "endgame") budget += 350000;
  if (profile.branching <= 5) budget += 180000;
  if (profile.branching >= 14) budget -= 120000;
  if (profile.captures > 0) budget += 160000;
  if (evaluationMode) budget = Math.round(budget * 0.75);
  return Math.max(80000, budget);
}

function createUnifiedLimits(config, deadline) {
  return {
    nodeBudget: config.nodeBudget,
    deadline: Number.isFinite(deadline) ? deadline : getNow() + config.timeMs,
    quiescenceDepth: config.quiescenceDepth,
  };
}

function createUnifiedStats() {
  return {
    nodes: 0,
    leaves: 0,
    cutoffs: 0,
    transpositionHits: 0,
    tableSize: 0,
    quiescenceNodes: 0,
  };
}

function shouldStopUnifiedSearch(stats, limits) {
  if (stats.nodes >= limits.nodeBudget) return true;
  return getNow() >= limits.deadline;
}

function createUnifiedResult(
  move,
  score,

```

```

    completedDepth,
    targetDepth,
    profile,
    config,
    stats,
    aborted,
    mctsResult
  ) {
    const usedMcts = Boolean(mctsResult);
    return {
      move,
      score,
      depth: completedDepth,
      requestedDepth: targetDepth,
      completedDepth,
      targetDepth,
      nodes: stats.nodes,
      leaves: stats.leaves,
      cutoffs: stats.cutoffs,
      transpositionHits: stats.transpositionHits,
      tableSize: stats.tableSize,
      quiescenceNodes: stats.quiescenceNodes,
      aborted: Boolean(aborted),
      phase: profile.phase,
      complexity: profile.complexity,
      profile,
      strategy: usedMcts ? "unified-hybrid-mcts-alpha-beta" : "unified-alpha-beta",
      nodeBudget: config.nodeBudget,
      timeMs: config.timeMs,
      quiescenceDepth: config.quiescenceDepth,
      mctsUsed: usedMcts,
      mctsIterations: usedMcts ? mctsResult.iterations : 0,
      mctsRequestedIterations: usedMcts ? mctsResult.requestedSimulations : 0,
      mctsSimulationLimit: usedMcts ? mctsResult.simulationLimit : 0,
      mctsBestVisits: usedMcts ? mctsResult.bestVisits : 0,
      mctsBestWinRate: usedMcts ? mctsResult.bestWinRate : null,
      mctsTimedOut: usedMcts ? Boolean(mctsResult.timedOut) : false,
      mctsSearchTimeMs: usedMcts ? mctsResult.searchTimeMs : 0,
      iterations: usedMcts ? mctsResult.iterations : 0,
      simulations: usedMcts ? mctsResult.simulations : 0,
      requestedSimulations: usedMcts ? mctsResult.requestedSimulations : 0,
      simulationLimit: usedMcts ? mctsResult.simulationLimit : 0,
      averageSimulationPlies: usedMcts ? mctsResult.averageSimulationPlies : 0,
      maxSimulationPlies: usedMcts ? mctsResult.maxSimulationPlies : 0,
      bestVisits: usedMcts ? mctsResult.bestVisits : 0,
      bestWinRate: usedMcts ? mctsResult.bestWinRate : null,
    };
  }
}

function minimax(board, sideToMove, depthRemaining, perspectiveColor, stats) {
  stats.nodes += 1;
  const moves = Rules.generateMoves(board, sideToMove);

  if (moves.length === 0) {
    stats.leaves += 1;
    return Rules.opponent(sideToMove) === perspectiveColor ? WEIGHTS.terminal : -WEIGHTS.terminal;
  }

  if (depthRemaining <= 0) {
    stats.leaves += 1;

```

```

    return evaluatePosition(board, sideToMove, perspectiveColor).score;
  }

  const maximize = sideToMove === perspectiveColor;
  let bestScore = maximize ? Number.NEGATIVE_INFINITY : Number.POSITIVE_INFINITY;

  for (const move of moves) {
    const applied = Rules.applyMove(board, move);
    const score = minimax(
      applied.board,
      Rules.opponent(sideToMove),
      depthRemaining - 1,
      perspectiveColor,
      stats
    );
    bestScore = maximize ? Math.max(bestScore, score) : Math.min(bestScore, score);
  }

  return bestScore;
}

function alphaBeta(board, sideToMove, depthRemaining, perspectiveColor, alpha, beta, stats, table) {
  stats.nodes += 1;
  const alphaOriginal = alpha;
  const betaOriginal = beta;
  const key = `${Rules.boardToSignature(board, sideToMove)}:${depthRemaining}`;
  const entry = table.get(key);

  if (entry && entry.depth >= depthRemaining) {
    stats.transpositionHits += 1;
    if (entry.flag === "exact") return entry.score;
    if (entry.flag === "lower") alpha = Math.max(alpha, entry.score);
    if (entry.flag === "upper") beta = Math.min(beta, entry.score);
    if (alpha >= beta) return entry.score;
  }

  const moves = Rules.generateMoves(board, sideToMove);

  if (moves.length === 0) {
    stats.leaves += 1;
    return Rules.opponent(sideToMove) === perspectiveColor ? WEIGHTS.terminal : -WEIGHTS.terminal;
  }

  if (depthRemaining <= 0) {
    stats.leaves += 1;
    return evaluatePosition(board, sideToMove, perspectiveColor).score;
  }

  const maximize = sideToMove === perspectiveColor;
  let bestScore = maximize ? Number.NEGATIVE_INFINITY : Number.POSITIVE_INFINITY;
  const orderedMoves = orderMoves(board, moves, sideToMove, perspectiveColor, maximize);

  for (const move of orderedMoves) {
    const applied = Rules.applyMove(board, move);
    const score = alphaBeta(
      applied.board,
      Rules.opponent(sideToMove),
      depthRemaining - 1,
      perspectiveColor,
      alpha,

```

```

    beta,
    stats,
    table
  );

  if (maximize) {
    bestScore = Math.max(bestScore, score);
    alpha = Math.max(alpha, bestScore);
  } else {
    bestScore = Math.min(bestScore, score);
    beta = Math.min(beta, bestScore);
  }

  if (alpha >= beta) {
    stats.cutoffs += 1;
    break;
  }
}

let flag = "exact";
if (bestScore <= alphaOriginal) {
  flag = "upper";
} else if (bestScore >= betaOriginal) {
  flag = "lower";
}
table.set(key, { depth: depthRemaining, score: bestScore, flag });

return bestScore;
}

function orderMoves(board, moves, sideToMove, perspectiveColor, maximize) {
  return moves
    .map((move) => ({
      move,
      score: scoreMove(board, move, sideToMove, perspectiveColor),
    }))
    .sort((a, b) => (maximize ? b.score - a.score : a.score - b.score))
    .map((item) => item.move);
}

function prioritizeMove(moves, preferredMove) {
  const index = moves.findIndex((move) => move.id === preferredMove.id);
  if (index <= 0) return moves;
  return [moves[index]].concat(moves.slice(0, index), moves.slice(index + 1));
}

function scoreMove(board, move, sideToMove, perspectiveColor) {
  const piece = board[move.from.row][move.from.col] || move.piece;
  const promotion =
    piece &&
    piece.kind === Rules.KINDS.MAN &&
    move.to.row === Rules.promotionRow(piece.color);
  const capturedKings = move.captured.filter((item) => item.piece.kind === Rules.KINDS.KING).length;
  const center = (Rules.BOARD_SIZE - 1) / 2;
  const centrality = Math.max(0, 5 - (Math.abs(move.to.row - center) + Math.abs(move.to.col - center)));
  const progress =
    piece && piece.kind === Rules.KINDS.MAN
    ? piece.color === Rules.COLORS.WHITE
    ? Rules.BOARD_SIZE - 1 - move.to.row
    : move.to.row

```

```

    : 0;
const tactical =
  move.captured.length * 10000 +
  capturedKings * 2600 +
  (promotion ? 1200 : 0) +
  centrality * 24 +
  progress * 14;

return sideToMove === perspectiveColor ? tactical : -tactical;
}

function createMctsNode(board, sideToMove, parent, move, moves) {
  return {
    board,
    sideToMove,
    parent,
    move,
    children: [],
    untriedMoves: moves ? moves.slice() : Rules.generateMoves(board, sideToMove),
    visits: 0,
    wins: 0,
  };
}

function selectUctChild(node, rootColor, exploration) {
  const parentVisits = Math.max(1, node.visits);
  let bestChild = node.children[0];
  let bestScore = Number.NEGATIVE_INFINITY;

  for (const child of node.children) {
    if (child.visits === 0) return child;
    const winRate = child.wins / child.visits;
    const exploitation = node.sideToMove === rootColor ? winRate : 1 - winRate;
    const explorationValue = exploration * Math.sqrt(Math.log(parentVisits) / child.visits);
    const score = exploitation + explorationValue;
    if (score > bestScore) {
      bestScore = score;
      bestChild = child;
    }
  }

  return bestChild;
}

function expandMctsNode(node, random) {
  const index = Math.min(node.untriedMoves.length - 1, Math.floor(random() * node.untriedMoves.length));
  const [move] = node.untriedMoves.splice(index, 1);
  const applied = Rules.applyMove(node.board, move);
  const child = createMctsNode(
    applied.board,
    Rules.opponent(node.sideToMove),
    node,
    move
  );
  node.children.push(child);
  return child;
}

function simulateHeuristicPayout(board, sideToMove, rootColor, simulationLimit, random) {
  let currentBoard = Rules.cloneBoard(board);

```

```

let currentSide = sideToMove;
let plies = 0;

while (plies < simulationLimit) {
  const moves = Rules.generateMoves(currentBoard, currentSide);
  if (moves.length === 0) {
    return {
      result: Rules.opponent(currentSide) === rootColor ? 1 : 0,
      plies,
    };
  }

  const selected = selectHeuristicRolloutMove(currentBoard, moves, currentSide, rootColor, random);
  currentBoard = selected.board;
  currentSide = Rules.opponent(currentSide);
  plies += 1;
}

return {
  result: heuristicResult(currentBoard, currentSide, rootColor),
  plies,
};
}

function selectHeuristicRolloutMove(board, moves, sideToMove, rootColor, random) {
  const candidates = moves.map((move) => ({
    move,
    weight: Math.max(1, 120 + scoreRolloutMove(board, move, sideToMove, rootColor)),
  }));
  const totalWeight = candidates.reduce((total, candidate) => total + candidate.weight, 0);
  let cursor = random() * totalWeight;
  for (const candidate of candidates) {
    cursor -= candidate.weight;
    if (cursor <= 0) {
      return {
        move: candidate.move,
        board: Rules.applyMove(board, candidate.move).board,
      };
    }
  }
  const fallback = candidates[candidates.length - 1];
  return {
    move: fallback.move,
    board: Rules.applyMove(board, fallback.move).board,
  };
}

function scoreRolloutMove(board, move, sideToMove, rootColor) {
  const piece = board[move.from.row][move.from.col] || move.piece;
  const capturedValue = move.captured.reduce(
    (total, item) => total + (item.piece.kind === Rules.KINDS.KING ? 180 : 70),
    0
  );
  const capturedKings = move.captured.filter((item) => item.piece.kind === Rules.KINDS.KING).length;
  const promotion =
    piece &&
    piece.kind === Rules.KINDS.MAN &&
    move.to.row === Rules.promotionRow(piece.color);
  const center = (Rules.BOARD_SIZE - 1) / 2;
  const centrality = Math.max(0, 5 - (Math.abs(move.to.row - center) + Math.abs(move.to.col - center)));

```

```

const progress =
  piece && piece.kind === Rules.KINDS.MAN
    ? piece.color === Rules.COLORS.WHITE
      ? Rules.BOARD_SIZE - 1 - move.to.row
        : move.to.row
      : 4;
const rootBias = sideToMove === rootColor ? 8 : 0;

return (
  capturedValue +
  capturedKings * 45 +
  (promotion ? 90 : 0) +
  centrality * 9 +
  progress * 5 +
  rootBias
);
}

function heuristicResult(board, sideToMove, rootColor) {
  const score = evaluatePosition(board, sideToMove, rootColor).score;
  if (score >= WEIGHTS.terminal) return 1;
  if (score <= -WEIGHTS.terminal) return 0;
  return clamp(0.5 + score / 2400, 0, 1);
}

function backpropagateMcts(node, result) {
  let current = node;
  while (current) {
    current.visits += 1;
    current.wins += result;
    current = current.parent;
  }
}

function selectMostVisitedChild(root) {
  if (!root.children.length) return null;
  return root.children.reduce((best, child) => {
    if (child.visits > best.visits) return child;
    if (child.visits === best.visits && child.wins / child.visits > best.wins / best.visits) {
      return child;
    }
    return best;
  }, root.children[0]);
}

function normalizeDepth(depth) {
  const parsed = Number.parseInt(depth, 10);
  if (!Number.isFinite(parsed)) return 1;
  return Math.max(1, parsed);
}

function normalizeMonteCarloSimulations(simulationCount) {
  const parsed = Number.parseInt(simulationCount, 10);
  if (!Number.isFinite(parsed)) return MCTS.defaultSimulations;
  return Math.max(1, parsed);
}

function calculateSimulationLimit(simulations) {
  return Math.max(
    48,

```

```

    Math.min(120, MCTS.defaultSimulationLimit + Math.floor(Math.sqrt(simulations) / 5))
  );
}

function getNow() {
  if (typeof performance !== "undefined" && typeof performance.now === "function") {
    return performance.now();
  }
  return Date.now();
}

function clamp(value, min, max) {
  return Math.min(max, Math.max(min, value));
}

function evaluatePosition(board, sideToMove, perspectiveColor) {
  const perspective = perspectiveColor || Rules.COLORS.WHITE;
  const material = evaluateMaterial(board);
  const advancement = evaluateAdvancement(board);
  const center = evaluateCenterControl(board);
  const mobility = evaluateMobility(board);
  const capturePotential = evaluateCapturePotential(board);
  const terminal = evaluateTerminal(board);

  const whiteScore = material + advancement + center + mobility + capturePotential + terminal;
  const score = perspective === Rules.COLORS.WHITE ? whiteScore : -whiteScore;

  return {
    score,
    whiteScore,
    perspective,
    sideToMove,
    components: {
      material,
      advancement,
      center,
      mobility,
      capturePotential,
      terminal,
    },
  };
}

function evaluateMaterial(board) {
  let score = 0;
  forEachPiece(board, (piece) => {
    const value = piece.kind === Rules.KINDS.KING ? WEIGHTS.king : WEIGHTS.man;
    score += signed(piece.color, value);
  });
  return score;
}

function evaluateAdvancement(board) {
  let score = 0;
  forEachPiece(board, (piece, row) => {
    if (piece.kind === Rules.KINDS.KING) return;
    const progress = piece.color === Rules.COLORS.WHITE ? Rules.BOARD_SIZE - 1 - row : row;
    score += signed(piece.color, progress * WEIGHTS.advancement);
  });
  return score;
}

```

```

}

function evaluateCenterControl(board) {
  let score = 0;
  const center = (Rules.BOARD_SIZE - 1) / 2;
  forEachPiece(board, (piece, row, col) => {
    const rowDistance = Math.abs(row - center);
    const colDistance = Math.abs(col - center);
    const centrality = Math.max(0, 5 - (rowDistance + colDistance));
    const value = centrality * WEIGHTS.center * (piece.kind === Rules.KINDS.KING ? 1.2 : 1);
    score += signed(piece.color, value);
  });
  return Math.round(score);
}

function evaluateMobility(board) {
  const whiteMoves = Rules.generateMoves(board, Rules.COLORS.WHITE).length;
  const blackMoves = Rules.generateMoves(board, Rules.COLORS.BLACK).length;
  return (whiteMoves - blackMoves) * WEIGHTS.mobility;
}

function evaluateCapturePotential(board) {
  const whiteCaptures = maxCaptureLength(Rules.generateMoves(board, Rules.COLORS.WHITE));
  const blackCaptures = maxCaptureLength(Rules.generateMoves(board, Rules.COLORS.BLACK));
  return (whiteCaptures - blackCaptures) * WEIGHTS.capture;
}

function evaluateTerminal(board) {
  const whiteMoves = Rules.generateMoves(board, Rules.COLORS.WHITE).length;
  const blackMoves = Rules.generateMoves(board, Rules.COLORS.BLACK).length;
  const counts = Rules.countPieces(board);

  if (counts.white === 0 || whiteMoves === 0) return -WEIGHTS.terminal;
  if (counts.black === 0 || blackMoves === 0) return WEIGHTS.terminal;
  return 0;
}

function maxCaptureLength(moves) {
  const captures = moves.filter((move) => move.isCapture);
  if (captures.length === 0) return 0;
  return Math.max(...captures.map((move) => move.captured.length));
}

function signed(color, value) {
  return color === Rules.COLORS.WHITE ? value : -value;
}

function forEachPiece(board, callback) {
  for (let row = 0; row < Rules.BOARD_SIZE; row += 1) {
    for (let col = 0; col < Rules.BOARD_SIZE; col += 1) {
      const piece = board[row][col];
      if (piece) callback(piece, row, col);
    }
  }
}

return {
  WEIGHTS,
  selectRandomMove,
  selectMinimaxMove,

```

```

    selectAlphaBetaMove,
    selectMonteCarloMove,
    selectUnifiedMove,
    evaluateUnifiedPosition,
    evaluatePosition,
  };
});

```

Файл app.js

```

(function () {
  "use strict";

  const Rules = window.DraughtsRules;
  const AI = window.DraughtsAI;
  const STORAGE_KEY = "international-draughts-history-v1";
  const AI_MOVE_DELAY_MS = 520;
  const MINIMAX_MIN_DEPTH = 1;
  const MINIMAX_MAX_DEPTH = 10;
  const MINIMAX_WARNING_DEPTH = 6;
  const MINIMAX_CRITICAL_DEPTH = 8;
  const ALPHABETA_MAX_DEPTH = 14;
  const ALPHABETA_WARNING_DEPTH = 9;
  const ALPHABETA_CRITICAL_DEPTH = 12;
  const MCTS_MIN_SIMULATIONS = 100;
  const MCTS_DEFAULT_SIMULATIONS = 10000;
  const MCTS_MAX_SIMULATIONS = 100000;
  const MCTS_WARNING_SIMULATIONS = 10000;
  const MCTS_CRITICAL_SIMULATIONS = 50000;
  const ASSETS = {
    "white-man": "assets/white-man.png",
    "black-man": "assets/black-man.png",
    "white-king": "assets/white-king.png",
    "black-king": "assets/black-king.png",
  };

  const labels = {
    white: "білі",
    black: "чорні",
    "human-human": "Людина vs людина",
    "human-ai": "Людина vs алгоритм",
    "ai-ai": "Алгоритм vs алгоритм",
  };

  const algorithmLabels = {
    random: "випадковий",
    minimax: "мінімакс",
    alphabeta: "альфа-бета",
    mcts: "Монте-Карло",
    unified: "єдина система III",
  };

  const dom = {
    board: document.getElementById("board"),
    statusLine: document.getElementById("statusLine"),
    turnChip: document.getElementById("turnChip"),
    modeLabel: document.getElementById("modeLabel"),
    legalMovesLabel: document.getElementById("legalMovesLabel"),
    whiteCountLabel: document.getElementById("whiteCountLabel"),
  };

```

```

blackCountLabel: document.getElementById("blackCountLabel"),
moveTimeLabel: document.getElementById("moveTimeLabel"),
gameTimeLabel: document.getElementById("gameTimeLabel"),
captureState: document.getElementById("captureState"),
evaluationScoreLabel: document.getElementById("evaluationScoreLabel"),
evaluationAdvantageLabel: document.getElementById("evaluationAdvantageLabel"),
evaluationBarFill: document.getElementById("evaluationBarFill"),
materialScoreLabel: document.getElementById("materialScoreLabel"),
positionScoreLabel: document.getElementById("positionScoreLabel"),
mobilityScoreLabel: document.getElementById("mobilityScoreLabel"),
captureScoreLabel: document.getElementById("captureScoreLabel"),
toggleSystemEvaluationButton: document.getElementById("toggleSystemEvaluationButton"),
systemEvaluationPanel: document.getElementById("systemEvaluationPanel"),
systemEvaluationScoreLabel: document.getElementById("systemEvaluationScoreLabel"),
systemEvaluationAdvantageLabel: document.getElementById("systemEvaluationAdvantageLabel"),
systemEvaluationDepthLabel: document.getElementById("systemEvaluationDepthLabel"),
systemEvaluationMoveLabel: document.getElementById("systemEvaluationMoveLabel"),
systemEvaluationNodesLabel: document.getElementById("systemEvaluationNodesLabel"),
systemEvaluationTimeLabel: document.getElementById("systemEvaluationTimeLabel"),
moveList: document.getElementById("moveList"),
totalGamesLabel: document.getElementById("totalGamesLabel"),
whiteWinsLabel: document.getElementById("whiteWinsLabel"),
blackWinsLabel: document.getElementById("blackWinsLabel"),
averageGameTimeLabel: document.getElementById("averageGameTimeLabel"),
lastSearchTimeLabel: document.getElementById("lastSearchTimeLabel"),
lastNodesLabel: document.getElementById("lastNodesLabel"),
lastRateLabel: document.getElementById("lastRateLabel"),
lastCutoffsLabel: document.getElementById("lastCutoffsLabel"),
gameSearchTimeLabel: document.getElementById("gameSearchTimeLabel"),
gameNodesLabel: document.getElementById("gameNodesLabel"),
gameCacheHitsLabel: document.getElementById("gameCacheHitsLabel"),
memoryDeltaLabel: document.getElementById("memoryDeltaLabel"),
historyList: document.getElementById("historyList"),
startGameButton: document.getElementById("startGameButton"),
clearHistoryButton: document.getElementById("clearHistoryButton"),
whiteAlgorithmSelect: document.getElementById("whiteAlgorithmSelect"),
blackAlgorithmSelect: document.getElementById("blackAlgorithmSelect"),
whiteDepthInput: document.getElementById("whiteDepthInput"),
blackDepthInput: document.getElementById("blackDepthInput"),
whiteDepthWarning: document.getElementById("whiteDepthWarning"),
blackDepthWarning: document.getElementById("blackDepthWarning"),
modeButtons: Array.from(document.querySelectorAll("[data-mode]")),
};

```

```

const state = {
  board: Rules.createInitialBoard(),
  turn: Rules.COLORS.WHITE,
  gameStarted: false,
  mode: "human-human",
  pendingMode: "human-human",
  moves: [],
  selected: null,
  selectedMoves: [],
  lastMove: null,
  moveLog: [],
  gameOver: false,
  outcome: null,
  aiTimer: null,
  aiConfig: {
    white: "random",

```

```

    black: "minimax",
  },
  pendingAiConfig: {
    white: "random",
    black: "minimax",
  },
  searchDepths: {
    white: 3,
    black: 3,
  },
  pendingDepths: {
    white: 3,
    black: 3,
  },
  searchSimulations: {
    white: MCTS_DEFAULT_SIMULATIONS,
    black: MCTS_DEFAULT_SIMULATIONS,
  },
  pendingSimulations: {
    white: MCTS_DEFAULT_SIMULATIONS,
    black: MCTS_DEFAULT_SIMULATIONS,
  },
  gameStartedAt: 0,
  turnStartedAt: 0,
  gameFinishedAt: null,
  lastMoveDurationMs: 0,
  lastResourceUse: null,
  resourceTotals: createEmptyResourceTotals(),
  systemEvaluationEnabled: false,
  systemEvaluation: null,
  systemEvaluationKey: "",
};

function createEmptyResourceTotals() {
  return {
    aiMoves: 0,
    searchMs: 0,
    nodes: 0,
    leaves: 0,
    cutoffs: 0,
    transpositionHits: 0,
    tableSizeMax: 0,
    memoryDeltaBytes: 0,
    memorySamples: 0,
  };
}

function key(row, col) {
  return `${row},${col}`;
}

function sameSquare(a, b) {
  return a && b && a.row === b.row && a.col === b.col;
}

function loadHistory() {
  try {
    const raw = localStorage.getItem(STORAGE_KEY);
    return raw ? JSON.parse(raw) : [];
  } catch (error) {

```

```

    return [];
  }
}

function saveHistory(history) {
  localStorage.setItem(STORAGE_KEY, JSON.stringify(history.slice(0, 100)));
}

function isHumanTurn(color) {
  if (state.mode === "human-human") return true;
  if (state.mode === "human-ai") return color === Rules.COLORS.WHITE;
  return false;
}

function isPendingHumanSide(color) {
  if (state.pendingMode === "human-human") return true;
  if (state.pendingMode === "human-ai") return color === Rules.COLORS.WHITE;
  return false;
}

function getAlgorithmForColor(color) {
  return state.aiConfig[color] || "random";
}

function getDepthForColor(color) {
  return state.searchDepths[color] || 1;
}

function getSimulationCountForColor(color) {
  return state.searchSimulations[color] || MCTS_DEFAULT_SIMULATIONS;
}

function startGame() {
  clearAiTimer();
  const now = performance.now();
  state.mode = state.pendingMode;
  state.aiConfig = { ...state.pendingAiConfig };
  state.searchDepths = {
    white: state.aiConfig.white === "unified" ? 0 : clampDepthForAlgorithm(state.pendingDepths.white,
state.aiConfig.white),
    black: state.aiConfig.black === "unified" ? 0 : clampDepthForAlgorithm(state.pendingDepths.black,
state.aiConfig.black),
  };
  state.searchSimulations = {
    white: clampSimulationCount(state.pendingSimulations.white),
    black: clampSimulationCount(state.pendingSimulations.black),
  };
  state.board = Rules.createInitialBoard();
  state.turn = Rules.COLORS.WHITE;
  state.moves = Rules.generateMoves(state.board, state.turn);
  state.selected = null;
  state.selectedMoves = [];
  state.lastMove = null;
  state.moveLog = [];
  state.gameStarted = true;
  state.gameOver = false;
  state.outcome = null;
  state.gameStartedAt = now;
  state.turnStartedAt = now;
  state.gameFinishedAt = null;
}

```

```

state.lastMoveDurationMs = 0;
state.lastResourceUse = null;
state.resourceTotals = createEmptyResourceTotals();
render();
}

function prepareGame() {
  clearAiTimer();
  state.board = Rules.createInitialBoard();
  state.turn = Rules.COLORS.WHITE;
  state.moves = [];
  state.selected = null;
  state.selectedMoves = [];
  state.lastMove = null;
  state.moveLog = [];
  state.gameStarted = false;
  state.gameOver = false;
  state.outcome = null;
  state.gameStartedAt = 0;
  state.turnStartedAt = 0;
  state.gameFinishedAt = null;
  state.lastMoveDurationMs = 0;
  state.lastResourceUse = null;
  state.resourceTotals = createEmptyResourceTotals();
  render();
}

function movesForSquare(row, col) {
  return state.moves.filter((move) => move.from.row === row && move.from.col === col);
}

function selectSquare(row, col) {
  const piece = state.board[row][col];
  if (!piece || piece.color !== state.turn) {
    state.selected = null;
    state.selectedMoves = [];
    setStatus("Оберіть фігуру сторони, яка зараз має хід.");
    renderBoard();
    return;
  }

  const options = movesForSquare(row, col);
  if (options.length === 0) {
    const hasCaptures = state.moves.some((move) => move.isCapture);
    setStatus(
      hasCaptures
        ? "Для цього ходу діє правило максимального взяття. Оберіть фігуру, що бере найбільше шашок."
        : "Ця фігура зараз не має легальних ходів."
    );
    return;
  }

  state.selected = { row, col };
  state.selectedMoves = options;
  setStatus(
    options.some((move) => move.isCapture)
      ? "Доступне обов'язкове взяття. Можна обрати одну з підсвічених кінцевих клітин."
      : "Оберіть підсвічену клітину для ходу."
  );
  renderBoard();
}

```

```

}

function applySelectedMove(row, col) {
  const move = state.selectedMoves.find((item) => item.to.row === row && item.to.col === col);
  if (!move) return false;

  return commitMove(move, "human");
}

function commitMove(move, actor, diagnostics) {
  const finishedAt = performance.now();
  const durationMs = Math.max(0, finishedAt - state.turnStartedAt);
  const applied = Rules.applyMove(state.board, move);
  state.board = applied.board;
  state.lastMove = applied.move;
  state.lastMoveDurationMs = durationMs;
  state.moveLog.push({
    color: state.turn,
    notation: Rules.formatMove(applied.move),
    captured: applied.move.captured.length,
    crowned: applied.move.crowned,
    actor,
    durationMs,
    diagnostics: diagnostics || null,
  });
  recordResourceUse(actor, diagnostics);

  const nextTurn = Rules.opponent(state.turn);
  const outcome = Rules.getGameOutcome(state.board, nextTurn);
  state.selected = null;
  state.selectedMoves = [];

  if (outcome) {
    state.turn = nextTurn;
    state.moves = [];
    state.gameOver = true;
    state.outcome = outcome;
    state.gameFinishedAt = performance.now();
    recordFinishedGame(outcome);
  } else {
    state.turn = nextTurn;
    state.moves = Rules.generateMoves(state.board, state.turn);
    state.turnStartedAt = performance.now();
  }

  render();
  return true;
}

function clearAiTimer() {
  if (!state.aiTimer) return;
  window.clearTimeout(state.aiTimer);
  state.aiTimer = null;
}

function scheduleAutoMove() {
  clearAiTimer();
  if (!state.gameStarted || state.gameOver || isHumanTurn(state.turn)) return;
  state.aiTimer = window.setTimeout(() => {
    state.aiTimer = null;
  }, 1000);
}

```

```

    playAiMove();
  }, AI_MOVE_DELAY_MS);
}

```

```

function playAiMove() {
  if (!state.gameStarted || state.gameOver || isHumanTurn(state.turn)) return;
  const algorithm = getAlgorithmForColor(state.turn);
  const memoryBefore = getUsedHeapBytes();
  const startedAt = performance.now();
  const result = chooseAiMove(algorithm);
  const decisionMs = performance.now() - startedAt;
  const memoryAfter = getUsedHeapBytes();
  const move = result.move;
  if (!move) return;
  commitMove(move, algorithm, {
    ...result.diagnostics,
    decisionMs,
    memoryDeltaBytes:
      Number.isFinite(memoryBefore) && Number.isFinite(memoryAfter)
        ? memoryAfter - memoryBefore
        : null,
  });
}

```

```

function chooseAiMove(algorithm) {
  if (algorithm === "minimax") {
    const result = AI.selectMinimaxMove(state.board, state.turn, getDepthForColor(state.turn), state.moves);
    return {
      move: result.move,
      diagnostics: {
        score: result.score,
        depth: result.depth,
        nodes: result.nodes,
        leaves: result.leaves,
        cutoffs: result.cutoffs,
        transpositionHits: result.transpositionHits,
        tableSize: result.tableSize,
      },
    };
  }
}

```

```

if (algorithm === "alphabeta") {
  const result = AI.selectAlphaBetaMove(state.board, state.turn, getDepthForColor(state.turn), state.moves);
  return {
    move: result.move,
    diagnostics: {
      score: result.score,
      depth: result.depth,
      nodes: result.nodes,
      leaves: result.leaves,
      cutoffs: result.cutoffs,
      transpositionHits: result.transpositionHits,
      tableSize: result.tableSize,
    },
  };
}

```

```

if (algorithm === "mcts") {
  const result = AI.selectMonteCarloMove(
    state.board,

```

```

    state.turn,
    getSimulationCountForColor(state.turn),
    state.moves
  );
  return {
    move: result.move,
    diagnostics: {
      score: result.score,
      depth: result.depth,
      nodes: result.nodes,
      leaves: result.leaves,
      cutoffs: result.cutoffs,
      transpositionHits: result.transpositionHits,
      tableSize: result.tableSize,
      iterations: result.iterations,
      simulations: result.simulations,
      requestedSimulations: result.requestedSimulations,
      simulationLimit: result.simulationLimit,
      averageSimulationPlies: result.averageSimulationPlies,
      maxSimulationPlies: result.maxSimulationPlies,
      bestVisits: result.bestVisits,
      bestWinRate: result.bestWinRate,
    },
  };
}

if (algorithm === "unified") {
  const result = AI.selectUnifiedMove(state.board, state.turn, state.moves);
  return {
    move: result.move,
    diagnostics: createSearchDiagnostics(result),
  };
}

return {
  move: AI.selectRandomMove(state.moves),
  diagnostics: {
    depth: 0,
    nodes: 0,
    leaves: 0,
    cutoffs: 0,
    transpositionHits: 0,
    tableSize: 0,
  },
};
}

function createSearchDiagnostics(result) {
  return {
    score: result.score,
    depth: result.depth,
    requestedDepth: result.requestedDepth,
    completedDepth: result.completedDepth,
    targetDepth: result.targetDepth,
    nodes: result.nodes,
    leaves: result.leaves,
    quiescenceNodes: result.quiescenceNodes,
    cutoffs: result.cutoffs,
    transpositionHits: result.transpositionHits,
    tableSize: result.tableSize,
  };
}

```

```

    iterations: result.iterations,
    simulations: result.simulations,
    requestedSimulations: result.requestedSimulations,
    simulationLimit: result.simulationLimit,
    averageSimulationPlies: result.averageSimulationPlies,
    maxSimulationPlies: result.maxSimulationPlies,
    bestVisits: result.bestVisits,
    bestWinRate: result.bestWinRate,
    mctsUsed: result.mctsUsed,
    mctsIterations: result.mctsIterations,
    mctsRequestedIterations: result.mctsRequestedIterations,
    mctsSimulationLimit: result.mctsSimulationLimit,
    mctsBestVisits: result.mctsBestVisits,
    mctsBestWinRate: result.mctsBestWinRate,
    mctsTimedOut: result.mctsTimedOut,
    mctsSearchTimeMs: result.mctsSearchTimeMs,
    phase: result.phase,
    complexity: result.complexity,
    aborted: result.aborted,
    strategy: result.strategy,
    nodeBudget: result.nodeBudget,
    timeMs: result.timeMs,
  };
}

function handleBoardClick(event) {
  const square = event.target.closest(".square");
  if (!square || state.gameOver) return;

  if (!state.gameStarted) {
    setStatus("Оберіть режим, алгоритми та глибини, після цього натисніть Старт.");
    return;
  }

  if (!isHumanTurn(state.turn)) {
    setStatus(`Цією стороною керує алгоритм: ${algorithmLabels[getAlgorithmForColor(state.turn)]}.`);
    return;
  }

  const row = Number(square.dataset.row);
  const col = Number(square.dataset.col);

  if (state.selected && applySelectedMove(row, col)) {
    return;
  }
  selectSquare(row, col);
}

function recordFinishedGame(outcome) {
  const counts = Rules.countPieces(state.board);
  const history = loadHistory();
  history.unshift({
    id: Date.now(),
    date: new Date().toISOString(),
    mode: state.mode,
    winner: outcome.winner,
    reason: outcome.reason,
    moves: state.moveLog.length,
    whitePieces: counts.white,
    blackPieces: counts.black,
  });
}

```

```

    durationMs: getGameElapsedMs(),
    whiteAlgorithm: state.aiConfig.white,
    blackAlgorithm: state.aiConfig.black,
    whiteDepth: state.searchDepths.white,
    blackDepth: state.searchDepths.black,
    whiteSimulations: state.searchSimulations.white,
    blackSimulations: state.searchSimulations.black,
    resources: { ...state.resourceTotals },
  });
  saveHistory(history);
}

function recordResourceUse(actor, diagnostics) {
  if (actor === "human" || !diagnostics) return;

  const resourceUse = {
    actor,
    decisionMs: diagnostics.decisionMs || 0,
    nodes: diagnostics.nodes || 0,
    leaves: diagnostics.leaves || 0,
    cutoffs: diagnostics.cutoffs || 0,
    transpositionHits: diagnostics.transpositionHits || 0,
    tableSize: diagnostics.tableSize || 0,
    memoryDeltaBytes: diagnostics.memoryDeltaBytes,
  };

  state.lastResourceUse = resourceUse;
  state.resourceTotals.aiMoves += 1;
  state.resourceTotals.searchMs += resourceUse.decisionMs;
  state.resourceTotals.nodes += resourceUse.nodes;
  state.resourceTotals.leaves += resourceUse.leaves;
  state.resourceTotals.cutoffs += resourceUse.cutoffs;
  state.resourceTotals.transpositionHits += resourceUse.transpositionHits;
  state.resourceTotals.tableSizeMax = Math.max(state.resourceTotals.tableSizeMax, resourceUse.tableSize);

  if (Number.isFinite(resourceUse.memoryDeltaBytes)) {
    state.resourceTotals.memoryDeltaBytes += resourceUse.memoryDeltaBytes;
    state.resourceTotals.memorySamples += 1;
  }
}

function setStatus(message) {
  dom.statusLine.textContent = message;
}

function render() {
  renderBoard();
  renderStatePanel();
  renderMoveLog();
  renderHistory();
  updateStatus();
  scheduleAutoMove();
}

function renderBoard() {
  const forcedSquares = new Set();
  const targetSquares = new Map();
  const lastPath = new Set();
  const selectedKey = state.selected ? key(state.selected.row, state.selected.col) : "";

```

```

for (const move of state.moves) {
  if (move.isCapture) {
    forcedSquares.add(key(move.from.row, move.from.col));
  }
}

for (const move of state.selectedMoves) {
  targetSquares.set(key(move.to.row, move.to.col), move);
}

if (state.lastMove) {
  for (const point of state.lastMove.path) {
    lastPath.add(key(point.row, point.col));
  }
}

const fragment = document.createDocumentFragment();
for (let row = 0; row < Rules.BOARD_SIZE; row += 1) {
  for (let col = 0; col < Rules.BOARD_SIZE; col += 1) {
    const squareKey = key(row, col);
    const piece = state.board[row][col];
    const square = document.createElement("button");
    square.type = "button";
    square.className = "square";
    square.dataset.row = String(row);
    square.dataset.col = String(col);
    square.setAttribute("role", "gridcell");
    square.setAttribute("aria-label", squareLabel(row, col, piece));

    if (Rules.isDarkSquare(row, col)) square.classList.add("is-playable");
    if (squareKey === selectedKey) square.classList.add("is-selected");
    if (targetSquares.has(squareKey)) {
      square.classList.add("is-target");
      if (targetSquares.get(squareKey).isCapture) square.classList.add("is-capture-target");
    }
    if (forcedSquares.has(squareKey)) square.classList.add("is-forced");
    if (lastPath.has(squareKey)) square.classList.add("is-last");

    if (piece) {
      const image = document.createElement("img");
      image.className = "piece";
      image.draggable = false;
      image.src = ASSETS[`${piece.color}-${piece.kind}`];
      image.alt = piece.kind === Rules.KINDS.KING ? `Дамка, ${labels[piece.color]}` : `Шашка, ${labels[piece.color]}`;
      square.appendChild(image);
    }

    fragment.appendChild(square);
  }
}
dom.board.replaceChildren(fragment);
}

function squareLabel(row, col, piece) {
  const number = Rules.notation(row, col);
  const squareName = number ? `клітина ${number}` : `ряд ${row + 1}, колонка ${col + 1}`;
  if (!piece) return squareName;
  const kind = piece.kind === Rules.KINDS.KING ? "дамка" : "шашка";
  return `${squareName}, ${kind}, ${labels[piece.color]}`;
}

```

```

}

function renderStatePanel() {
  const counts = Rules.countPieces(state.board);
  const captureMoves = state.moves.filter((move) => move.isCapture);
  const maxCapture = captureMoves.length ? captureMoves[0].captured.length : 0;

  dom.turnChip.textContent = !state.gameStarted
    ? "Очікує старту"
    : state.gameOver
    ? "Партію завершено"
    : `Хід: ${labels[state.turn]}`;
  dom.modeLabel.textContent = state.gameStarted ? labels[state.mode] : "Не запущено";
  dom.legalMovesLabel.textContent = String(state.moves.length);
  dom.whiteCountLabel.textContent = `${counts.white} (${counts.whiteKings} дам.)`;
  dom.blackCountLabel.textContent = `${counts.black} (${counts.blackKings} дам.)`;
  dom.captureState.textContent =
    captureMoves.length > 0
      ? `Обов'язкове максимальне взяття: ${maxCapture}`
      : "Взяття зараз немає";
  renderEvaluation(AI.evaluatePosition(state.board, state.turn));
  renderSystemEvaluation();
  renderAlgorithmSettings();
  renderTimers();
  renderResourcePanel();

  for (const button of dom.modeButtons) {
    button.classList.toggle("is-active", button.dataset.mode === state.pendingMode);
  }
}

function renderAlgorithmSettings() {
  dom.whiteAlgorithmSelect.value = state.pendingAiConfig.white;
  dom.blackAlgorithmSelect.value = state.pendingAiConfig.black;
  dom.whiteAlgorithmSelect.disabled = state.pendingMode === "human-human" || state.pendingMode === "human-ai";
  dom.blackAlgorithmSelect.disabled = state.pendingMode === "human-human";

  renderSideDepthSettings(Rules.COLORS.WHITE, dom.whiteDepthInput, dom.whiteDepthWarning);
  renderSideDepthSettings(Rules.COLORS.BLACK, dom.blackDepthInput, dom.blackDepthWarning);
}

function renderSideDepthSettings(color, input, warning) {
  const algorithm = state.pendingAiConfig[color];
  const humanSide = isPendingHumanSide(color);
  const label = input.closest(".side-depth").querySelector("span");

  if (algorithm === "mcts") {
    const simulations = clampSimulationCount(state.pendingSimulations[color]);
    state.pendingSimulations[color] = simulations;
    label.textContent = `Симуляції ${getSideGenitive(color)}`;
    input.value = String(simulations);
    input.min = String(MCTS_MIN_SIMULATIONS);
    input.max = String(MCTS_MAX_SIMULATIONS);
    input.step = "100";
    input.disabled = humanSide;

    const level = getSearchWarningLevel(algorithm, simulations, humanSide);
    warning.classList.toggle("is-warning", level === "warning");
    warning.classList.toggle("is-critical", level === "critical");
  }
}

```

```

    warning.textContent = getSearchWarningText(color, algorithm, simulations, humanSide);
    return;
}

if (algorithm === "unified") {
    label.textContent = `Автоаналіз ${getSideGenitive(color)}`;
    input.value = "";
    input.placeholder = "авто";
    input.min = String(MINIMAX_MIN_DEPTH);
    input.max = String(ALPHABETA_MAX_DEPTH);
    input.step = "1";
    input.disabled = true;

    warning.classList.remove("is-warning", "is-critical");
    warning.textContent = getSearchWarningText(color, algorithm, 0, humanSide);
    return;
}

const maxDepth = getMaxDepthForAlgorithm(algorithm);
const depth = clampDepthForAlgorithm(state.pendingDepths[color], algorithm);
state.pendingDepths[color] = depth;

label.textContent = `Глибина ${getSideGenitive(color)}`;
input.value = String(depth);
input.min = String(MINIMAX_MIN_DEPTH);
input.max = String(maxDepth);
input.step = "1";
input.disabled = humanSide || algorithm === "random";

const level = getSearchWarningLevel(algorithm, depth, humanSide);
warning.classList.toggle("is-warning", level === "warning");
warning.classList.toggle("is-critical", level === "critical");
warning.textContent = getSearchWarningText(color, algorithm, depth, humanSide);
}

function getMaxDepthForAlgorithm(algorithm) {
    if (algorithm === "minimax") return MINIMAX_MAX_DEPTH;
    if (algorithm === "alphabeta") return ALPHABETA_MAX_DEPTH;
    return MINIMAX_MAX_DEPTH;
}

function clampDepthForAlgorithm(depth, algorithm) {
    const parsed = Number.parseInt(depth, 10);
    const maxDepth = getMaxDepthForAlgorithm(algorithm);
    if (!Number.isFinite(parsed)) return MINIMAX_MIN_DEPTH;
    return Math.min(maxDepth, Math.max(MINIMAX_MIN_DEPTH, parsed));
}

function clampSimulationCount(value) {
    const parsed = Number.parseInt(value, 10);
    if (!Number.isFinite(parsed)) return MCTS_DEFAULT_SIMULATIONS;
    return Math.min(MCTS_MAX_SIMULATIONS, Math.max(MCTS_MIN_SIMULATIONS, parsed));
}

function getSearchWarningLevel(algorithm, value, humanSide) {
    if (humanSide || algorithm === "random") return "none";
    if (algorithm === "unified") return "none";
    if (algorithm === "minimax") {
        if (value >= MINIMAX_CRITICAL_DEPTH) return "critical";
        if (value >= MINIMAX_WARNING_DEPTH) return "warning";
    }
}

```

```

    return "none";
  }
  if (algorithm === "alphabeta") {
    if (value >= ALPHABETA_CRITICAL_DEPTH) return "critical";
    if (value >= ALPHABETA_WARNING_DEPTH) return "warning";
  }
  if (algorithm === "mcts") {
    if (value >= MCTS_CRITICAL_SIMULATIONS) return "critical";
    if (value >= MCTS_WARNING_SIMULATIONS) return "warning";
  }
  return "none";
}

function getSideTitle(color) {
  return color === Rules.COLORS.WHITE ? "Білі" : "Чорні";
}

function getSideGenitive(color) {
  return color === Rules.COLORS.WHITE ? "білих" : "чорних";
}

function estimateMctsSimulationLimit(simulations) {
  return Math.max(48, Math.min(120, 56 + Math.floor(Math.sqrt(simulations) / 5)));
}

function getSearchWarningText(color, algorithm, value, humanSide) {
  const side = getSideTitle(color);
  if (humanSide) return `${side}: керує людина, параметр пошуку не використовується.`;
  if (algorithm === "random") return `${side}: випадковий алгоритм не використовує глибину або симуляції.`;
  if (algorithm === "unified") {
    return `${side}: єдина система III автоматично обирає глибину та може думати кілька секунд заради  
сильнішого ходу.`;
  }
  if (algorithm === "minimax" && value >= MINIMAX_CRITICAL_DEPTH) {
    return `${side}: критична глибина ${value} для мінімаксу, хід може тривати дуже довго.`;
  }
  if (algorithm === "minimax" && value >= MINIMAX_WARNING_DEPTH) {
    return `${side}: глибина ${value} для мінімаксу може відчутно сповільнювати хід.`;
  }
  if (algorithm === "alphabeta" && value >= ALPHABETA_CRITICAL_DEPTH) {
    return `${side}: критична глибина ${value} для альфа-бета у складних позиціях.`;
  }
  if (algorithm === "alphabeta" && value >= ALPHABETA_WARNING_DEPTH) {
    return `${side}: глибина ${value} для альфа-бета вже може бути повільною.`;
  }
  if (algorithm === "alphabeta" && value >= 6) {
    return `${side}: альфа-бета на глибині ${value} зазвичай працює помірно швидко.`;
  }
  if (algorithm === "mcts" && value >= MCTS_CRITICAL_SIMULATIONS) {
    return `${side}: критичні ${formatNumber(value)} симуляцій для Монте-Карло, хід може зайняти помітний  
час.`;
  }
  if (algorithm === "mcts" && value >= MCTS_WARNING_SIMULATIONS) {
    return `${side}: ${formatNumber(value)} симуляцій для Монте-Карло вже відчутно навантажують  
процесор.`;
  }
  if (algorithm === "mcts") {
    return `${side}: Монте-Карло виконає ${formatNumber(value)} евристичних симуляцій, rollout до  
${estimateMctsSimulationLimit(value)} півходів.`;
  }
}

```

```

    return `${side}: глибина ${value} зазвичай працює швидко.`;
  }

function renderEvaluation(evaluation) {
  const whiteScore = evaluation.whiteScore;
  const positionScore = evaluation.components.advancement + evaluation.components.center;
  const captureScore = evaluation.components.capturePotential + evaluation.components.terminal;

  dom.evaluationScoreLabel.textContent = formatScore(whiteScore);
  dom.materialScoreLabel.textContent = formatScore(evaluation.components.material);
  dom.positionScoreLabel.textContent = formatScore(positionScore);
  dom.mobilityScoreLabel.textContent = formatScore(evaluation.components.mobility);
  dom.captureScoreLabel.textContent = formatScore(captureScore);

  const side = whiteScore > 18 ? Rules.COLORS.WHITE : whiteScore < -18 ? Rules.COLORS.BLACK : null;
  dom.evaluationAdvantageLabel.textContent = side
    ? `Перевага: ${labels[side]} (${formatScore(Math.abs(whiteScore))}`
    : "Рівна позиція";

  const width = Math.min(50, Math.abs(whiteScore) / 18);
  dom.evaluationBarFill.classList.toggle("is-black", whiteScore < 0);
  dom.evaluationBarFill.style.left = whiteScore < 0 ? `${50 - width}%` : "50%";
  dom.evaluationBarFill.style.width = `${width}%`;
}

function renderSystemEvaluation() {
  dom.toggleSystemEvaluationButton.classList.toggle("is-active", state.systemEvaluationEnabled);
  dom.systemEvaluationPanel.hidden = !state.systemEvaluationEnabled;
  if (!state.systemEvaluationEnabled) return;

  const key = Rules.boardToSignature(state.board, state.turn);
  if (!state.systemEvaluation || state.systemEvaluationKey !== key) {
    const startedAt = performance.now();
    const result = AI.evaluateUnifiedPosition(state.board, state.turn, Rules.COLORS.WHITE);
    state.systemEvaluation = {
      ...result,
      decisionMs: performance.now() - startedAt,
    };
    state.systemEvaluationKey = key;
  }

  const evaluation = state.systemEvaluation;
  const whiteScore = evaluation.whiteScore;
  const side = whiteScore > 18 ? Rules.COLORS.WHITE : whiteScore < -18 ? Rules.COLORS.BLACK : null;
  dom.systemEvaluationScoreLabel.textContent = formatScore(whiteScore);
  dom.systemEvaluationAdvantageLabel.textContent = side
    ? `Перевага: ${labels[side]} (${formatScore(Math.abs(whiteScore))}`
    : "Рівна позиція";
  dom.systemEvaluationDepthLabel.textContent = `${evaluation.completedDepth}/${evaluation.targetDepth}`;
  dom.systemEvaluationMoveLabel.textContent = evaluation.bestMove ? evaluation.bestMove.notation : "-";
  dom.systemEvaluationNodesLabel.textContent = formatNumber(evaluation.nodes);
  dom.systemEvaluationTimeLabel.textContent = formatPreciseDuration(evaluation.decisionMs);
}

function renderResourcePanel() {
  const last = state.lastResourceUse;
  const totals = state.resourceTotals;

  dom.lastSearchTimeLabel.textContent = last ? formatPreciseDuration(last.decisionMs) : "0 мс";
  dom.lastNodesLabel.textContent = last ? formatNumber(last.nodes) : "0";
}

```

```

dom.lastRateLabel.textContent = last ? formatNumber(calculateNodeRate(last.nodes, last.decisionMs)) : "0";
dom.lastCutoffsLabel.textContent = last ? formatNumber(last.cutoffs) : "0";
dom.gameSearchTimeLabel.textContent = formatPreciseDuration(totals.searchMs);
dom.gameNodesLabel.textContent = formatNumber(totals.nodes);
dom.gameCacheHitsLabel.textContent = formatNumber(totals.transpositionHits);
dom.memoryDeltaLabel.textContent =
  totals.memorySamples > 0 ? formatBytes(totals.memoryDeltaBytes) : "н/д";
}

function renderMoveLog() {
  const recent = state.moveLog.slice(-10).reverse();
  const fragment = document.createDocumentFragment();
  for (const entry of recent) {
    const item = document.createElement("li");
    const extras = [];
    if (entry.captured) extras.push(`взято ${entry.captured}`);
    if (entry.crowned) extras.push("дамка");
    if (entry.actor === "random") extras.push("random AI");
    if (entry.actor === "minimax") {
      const depth = entry.diagnostics ? entry.diagnostics.depth : getDepthForColor(entry.color);
      extras.push(`minimax d${depth}`);
    }
    if (entry.actor === "alphabeta") {
      const depth = entry.diagnostics ? entry.diagnostics.depth : getDepthForColor(entry.color);
      extras.push(`alpha-beta d${depth}`);
    }
    if (entry.actor === "mcts") {
      const simulations =
        entry.diagnostics && entry.diagnostics.requestedSimulations
          ? entry.diagnostics.requestedSimulations
          : getSimulationCountForColor(entry.color);
      extras.push(`MCTS ${formatNumber(simulations)} сим.`);
    }
    if (entry.actor === "unified") {
      const depth = entry.diagnostics ? entry.diagnostics.completedDepth || entry.diagnostics.depth : 0;
      const target = entry.diagnostics ? entry.diagnostics.targetDepth || entry.diagnostics.requestedDepth : 0;
      extras.push(`система III d${depth}/${target}`);
    }
    if (entry.durationMs !== undefined) extras.push(`час ${formatDuration(entry.durationMs)}`);
    if (entry.diagnostics && entry.diagnostics.decisionMs !== undefined) {
      extras.push(`пошук ${formatPreciseDuration(entry.diagnostics.decisionMs)}`);
    }
    if (
      entry.diagnostics &&
      entry.diagnostics.iterations &&
      (entry.actor !== "mcts" || entry.diagnostics.iterations !== entry.diagnostics.requestedSimulations)
    ) {
      extras.push(`${formatNumber(entry.diagnostics.iterations)} сим.`);
    }
    if (entry.diagnostics && Number.isFinite(entry.diagnostics.bestWinRate)) {
      extras.push(`win ${Math.round(entry.diagnostics.bestWinRate * 100)}%`);
    }
    if (entry.diagnostics && entry.diagnostics.nodes) extras.push(`${formatNumber(entry.diagnostics.nodes)} вузлів`);
    if (entry.diagnostics && entry.diagnostics.cutoffs) {
      extras.push(`${formatNumber(entry.diagnostics.cutoffs)} відсіч.`);
    }
    item.textContent = `${labels[entry.color]}: ${entry.notation}${extras.length ? ` (${extras.join(", ")}` : ""}`;
    fragment.appendChild(item);
  }
}

```

```

    dom.moveList.replaceChildren(fragment);
}

function renderHistory() {
    const history = loadHistory();
    const whiteWins = history.filter((item) => item.winner === Rules.COLORS.WHITE).length;
    const blackWins = history.filter((item) => item.winner === Rules.COLORS.BLACK).length;
    const timedGames = history.filter((item) => Number.isFinite(item.durationMs));
    const averageMs = timedGames.length
        ? timedGames.reduce((total, item) => total + item.durationMs, 0) / timedGames.length
        : 0;

    dom.totalGamesLabel.textContent = String(history.length);
    dom.whiteWinsLabel.textContent = String(whiteWins);
    dom.blackWinsLabel.textContent = String(blackWins);
    dom.averageGameTimeLabel.textContent = formatDuration(averageMs);

    const fragment = document.createDocumentFragment();
    for (const item of history.slice(0, 5)) {
        const row = document.createElement("div");
        row.className = "history-item";
        const title = document.createElement("strong");
        title.textContent = `Пепемора: ${labels[item.winner]}`;
        const details = document.createElement("small");
        const duration = Number.isFinite(item.durationMs) ? formatDuration(item.durationMs) : "немає даних";
        const resources = item.resources
            ? `, AI час ${formatPreciseDuration(item.resources.searchMs)}, ${formatNumber(item.resources.nodes)}
            вузлів`
            : "";
        details.textContent = `${labels[item.model]}, ${item.moves} ходів, час партії ${duration}${resources},
        ${formatDate(item.date)}, ${formatReason(item.reason)}`;
        row.append(title, details);
        fragment.appendChild(row);
    }
    dom.historyList.replaceChildren(fragment);
}

function updateStatus() {
    if (!state.gameStarted) {
        setStatus("Оберіть режим, алгоритми та глибини для кожної сторони, потім натисніть Старт.");
        return;
    }

    if (state.gameOver && state.outcome) {
        setStatus(`Партію завершено. Перемогли ${labels[state.outcome.winner]}:
        ${formatReason(state.outcome.reason)}`);
        return;
    }

    if (!isHumanTurn(state.turn)) {
        const algorithm = getAlgorithmForColor(state.turn);
        const depthInfo =
            algorithm === "minimax" || algorithm === "alphabeta"
            ? `, глибина ${getDepthForColor(state.turn)}`
            : algorithm === "mcts"
            ? `, ${formatNumber(getSimulationCountForColor(state.turn))} симуляцій`
            : algorithm === "unified"
            ? ", автоматична глибина"
            : "";
    }

```

```

    setStatus(`Алгоритм ${algorithmLabels[algorithm]}${depthInfo} обирає хід для сторони:
    ${labels[state.turn]}.`);
    return;
  }

  const captureMoves = state.moves.filter((move) => move.isCapture);
  if (captureMoves.length > 0) {
    setStatus("Є обов'язкове взяття. Підсвічені фігури мають ходи з максимальною кількістю взятих шашок.");
  } else {
    setStatus(`Зараз ходять ${labels[state.turn]}.`);
  }
}

function formatDate(value) {
  return new Intl.DateTimeFormat("uk-UA", {
    day: "2-digit",
    month: "2-digit",
    year: "numeric",
    hour: "2-digit",
    minute: "2-digit",
  }).format(new Date(value));
}

function formatScore(value) {
  const normalized = value / 100;
  if (Math.abs(normalized) < 0.005) return "0.00";
  return `${normalized > 0 ? "+" : ""}${normalized.toFixed(2)}`;
}

function getUsedHeapBytes() {
  if (performance && performance.memory && Number.isFinite(performance.memory.usedJSHeapSize)) {
    return performance.memory.usedJSHeapSize;
  }
  return null;
}

function calculateNodeRate(nodes, ms) {
  if (!nodes || !ms) return 0;
  return Math.round(nodes / (ms / 1000));
}

function formatNumber(value) {
  return new Intl.NumberFormat("uk-UA").format(Math.round(value || 0));
}

function renderTimers() {
  dom.moveTimeLabel.textContent = formatDuration(getMoveElapsedMs());
  dom.gameTimeLabel.textContent = formatDuration(getGameElapsedMs());
}

function getMoveElapsedMs() {
  if (!state.gameStarted) return 0;
  if (!state.turnStartedAt) return 0;
  if (state.gameOver) return state.lastMoveDurationMs;
  return Math.max(0, performance.now() - state.turnStartedAt);
}

function getGameElapsedMs() {
  if (!state.gameStarted) return 0;
  if (!state.gameStartedAt) return 0;

```

```

const end = state.gameFinishedAt || performance.now();
return Math.max(0, end - state.gameStartedAt);
}

```

```

function formatDuration(ms) {
  const safeMs = Number.isFinite(ms) ? Math.max(0, ms) : 0;
  const totalSeconds = Math.floor(safeMs / 1000);
  const hours = Math.floor(totalSeconds / 3600);
  const minutes = Math.floor((totalSeconds % 3600) / 60);
  const seconds = totalSeconds % 60;

```

```

  if (hours > 0) {
    return `${hours}:${pad2(minutes)}:${pad2(seconds)}`;
  }
  return `${pad2(minutes)}:${pad2(seconds)}`;
}

```

```

function formatPreciseDuration(ms) {
  const safeMs = Number.isFinite(ms) ? Math.max(0, ms) : 0;
  if (safeMs < 1000) return `${Math.round(safeMs)} мс`;
  if (safeMs < 60000) return `${(safeMs / 1000).toFixed(2)} с`;
  return formatDuration(safeMs);
}

```

```

function formatBytes(bytes) {
  if (!Number.isFinite(bytes)) return "н/д";
  const sign = bytes > 0 ? "+" : bytes < 0 ? "-" : "";
  const abs = Math.abs(bytes);
  if (abs < 1024) return `${sign}${Math.round(abs)} Б`;
  if (abs < 1024 * 1024) return `${sign}${(abs / 1024).toFixed(1)} КБ`;
  return `${sign}${(abs / (1024 * 1024)).toFixed(2)} МБ`;
}

```

```

function pad2(value) {
  return String(value).padStart(2, "0");
}

```

```

function formatReason(reason) {
  if (reason === "white-no-pieces") return "у білих не лишилося фігур";
  if (reason === "black-no-pieces") return "у чорних не лишилося фігур";
  if (reason === "no-legal-moves") return "немає легальних ходів";
  return "завершення партії";
}

```

```

function clearHistory() {
  if (!loadHistory().length) return;
  const confirmed = window.confirm("Очистити статистику завершених партій?");
  if (!confirmed) return;
  saveHistory([]);
  renderHistory();
}

```

```

function toggleSystemEvaluation() {
  state.systemEvaluationEnabled = !state.systemEvaluationEnabled;
  render();
}

```

```

function updateAlgorithm(color, value) {
  state.pendingAiConfig[color] = ["random", "minimax", "alphabeta", "mcts", "unified"].includes(value) ? value :
  "random";
}

```

```

    if (state.pendingAiConfig[color] === "mcts") {
      state.pendingSimulations[color] = clampSimulationCount(state.pendingSimulations[color]);
    } else if (state.pendingAiConfig[color] !== "unified") {
      state.pendingDepths[color] = clampDepthForAlgorithm(state.pendingDepths[color],
state.pendingAiConfig[color]);
    }
    state.selected = null;
    state.selectedMoves = [];
    render();
  }

  function updateDepth(color, value) {
    if (state.pendingAiConfig[color] === "mcts") {
      state.pendingSimulations[color] = clampSimulationCount(value);
    } else if (state.pendingAiConfig[color] === "unified") {
      return;
    } else {
      state.pendingDepths[color] = clampDepthForAlgorithm(value, state.pendingAiConfig[color]);
    }
    render();
  }

  dom.board.addEventListener("click", handleBoardClick);
  dom.startGameButton.addEventListener("click", startGame);
  dom.clearHistoryButton.addEventListener("click", clearHistory);
  dom.toggleSystemEvaluationButton.addEventListener("click", toggleSystemEvaluation);
  dom.whiteAlgorithmSelect.addEventListener("change", () => {
    updateAlgorithm(Rules.COLORS.WHITE, dom.whiteAlgorithmSelect.value);
  });
  dom.blackAlgorithmSelect.addEventListener("change", () => {
    updateAlgorithm(Rules.COLORS.BLACK, dom.blackAlgorithmSelect.value);
  });
  dom.whiteDepthInput.addEventListener("change", () => {
    updateDepth(Rules.COLORS.WHITE, dom.whiteDepthInput.value);
  });
  dom.blackDepthInput.addEventListener("change", () => {
    updateDepth(Rules.COLORS.BLACK, dom.blackDepthInput.value);
  });

  for (const button of dom.modeButtons) {
    button.addEventListener("click", () => {
      state.pendingMode = button.dataset.mode;
      state.selected = null;
      state.selectedMoves = [];
      render();
    });
  }

  prepareGame();
  window.setInterval(renderTimers, 500);
})();

```

Файл rules.js

```

(function (global, factory) {
  if (typeof module === "object" && module.exports) {
    module.exports = factory();
  } else {
    global.DraughtsRules = factory();
  }
}

```

```

})(typeof globalThis !== "undefined" ? globalThis : this, function () {
  "use strict";

  const BOARD_SIZE = 10;
  const COLORS = Object.freeze({ WHITE: "white", BLACK: "black" });
  const KINDS = Object.freeze({ MAN: "man", KING: "king" });
  const DIRECTIONS = Object.freeze([
    [-1, -1],
    [-1, 1],
    [1, -1],
    [1, 1],
  ]);

  function opponent(color) {
    return color === COLORS.WHITE ? COLORS.BLACK : COLORS.WHITE;
  }

  function inside(row, col) {
    return row >= 0 && row < BOARD_SIZE && col >= 0 && col < BOARD_SIZE;
  }

  function isDarkSquare(row, col) {
    return (row + col) % 2 === 1;
  }

  function promotionRow(color) {
    return color === COLORS.WHITE ? 0 : BOARD_SIZE - 1;
  }

  function posKey(row, col) {
    return `${row},${col}`;
  }

  function clonePiece(piece) {
    return piece ? { color: piece.color, kind: piece.kind } : null;
  }

  function emptyBoard() {
    return Array.from({ length: BOARD_SIZE }, () => Array(BOARD_SIZE).fill(null));
  }

  function cloneBoard(board) {
    return board.map((row) => row.map(clonePiece));
  }

  function setPiece(board, row, col, color, kind) {
    board[row][col] = { color, kind: kind || KINDS.MAN };
    return board;
  }

  function createInitialBoard() {
    const board = emptyBoard();
    for (let row = 0; row < BOARD_SIZE; row += 1) {
      for (let col = 0; col < BOARD_SIZE; col += 1) {
        if (!isDarkSquare(row, col)) continue;
        if (row < 4) {
          board[row][col] = { color: COLORS.BLACK, kind: KINDS.MAN };
        } else if (row > 5) {
          board[row][col] = { color: COLORS.WHITE, kind: KINDS.MAN };
        }
      }
    }
  }

```

```

    }
  }
  return board;
}

function countPieces(board) {
  const counts = {
    white: 0,
    black: 0,
    whiteKings: 0,
    blackKings: 0,
  };
  for (let row = 0; row < BOARD_SIZE; row += 1) {
    for (let col = 0; col < BOARD_SIZE; col += 1) {
      const piece = board[row][col];
      if (!piece) continue;
      counts[piece.color] += 1;
      if (piece.kind === KINDS.KING) {
        counts[`${piece.color}Kings`] += 1;
      }
    }
  }
  return counts;
}

function notation(row, col) {
  if (!inside(row, col) || !isDarkSquare(row, col)) return "";
  return String(row * 5 + Math.floor(col / 2) + 1);
}

function getOccupant(board, from, current, row, col, movingPiece) {
  if (!inside(row, col)) return null;
  if (row === current.row && col === current.col) return movingPiece;
  if (!(row === current.row && col === current.col) && row === from.row && col === from.col) {
    return null;
  }
  return board[row][col];
}

function isEmptyForCapture(board, from, current, row, col, movingPiece) {
  return inside(row, col) && !getOccupant(board, from, current, row, col, movingPiece);
}

function createMove(piece, from, path, captured) {
  const to = path[path.length - 1];
  const move = {
    from: { ...from },
    to: { ...to },
    path: path.map((item) => ({ ...item })),
    captured: captured.map((item) => ({
      row: item.row,
      col: item.col,
      piece: clonePiece(item.piece),
    })),
    piece: clonePiece(piece),
    isCapture: captured.length > 0,
  };
  move.notation = formatMove(move);
  move.id = `${move.notation}.${move.captured.map((item) => posKey(item.row, item.col)).join("|")}`;
  return move;
}

```

```

}

function generateManCaptures(board, from, current, piece, path, captured, capturedKeys) {
  const continuations = [];

  for (const [dr, dc] of DIRECTIONS) {
    const jumped = { row: current.row + dr, col: current.col + dc };
    const landing = { row: current.row + dr * 2, col: current.col + dc * 2 };
    if (!inside(jumped.row, jumped.col) || !inside(landing.row, landing.col)) continue;

    const jumpedPiece = getOccupant(board, from, current, jumped.row, jumped.col, piece);
    const jumpedKey = posKey(jumped.row, jumped.col);
    if (!jumpedPiece || jumpedPiece.color !== piece.color || capturedKeys.has(jumpedKey)) continue;
    if (!isEmptyForCapture(board, from, current, landing.row, landing.col, piece)) continue;

    const nextCaptured = captured.concat({
      row: jumped.row,
      col: jumped.col,
      piece: clonePiece(jumpedPiece),
    });
    const nextKeys = new Set(capturedKeys);
    nextKeys.add(jumpedKey);
    const nextPath = path.concat({ row: landing.row, col: landing.col });
    const deeper = generateManCaptures(board, from, landing, piece, nextPath, nextCaptured, nextKeys);
    if (deeper.length > 0) {
      continuations.push(...deeper);
    } else {
      continuations.push(createMove(piece, from, nextPath, nextCaptured));
    }
  }

  return continuations;
}

function generateKingCaptures(board, from, current, piece, path, captured, capturedKeys) {
  const continuations = [];

  for (const [dr, dc] of DIRECTIONS) {
    let row = current.row + dr;
    let col = current.col + dc;

    while (inside(row, col)) {
      const occupant = getOccupant(board, from, current, row, col, piece);
      if (!occupant) {
        row += dr;
        col += dc;
        continue;
      }

      const jumpedKey = posKey(row, col);
      if (occupant.color === piece.color || capturedKeys.has(jumpedKey)) {
        break;
      }

      let landingRow = row + dr;
      let landingCol = col + dc;
      while (isEmptyForCapture(board, from, current, landingRow, landingCol, piece)) {
        const landing = { row: landingRow, col: landingCol };
        const nextCaptured = captured.concat({
          row,

```

```

        col,
        piece: clonePiece(occupant),
    });
    const nextKeys = new Set(capturedKeys);
    nextKeys.add(jumpedKey);
    const nextPath = path.concat({ ...landing });
    const deeper = generateKingCaptures(board, from, landing, piece, nextPath, nextCaptured, nextKeys);
    if (deeper.length > 0) {
        continuations.push(...deeper);
    } else {
        continuations.push(createMove(piece, from, nextPath, nextCaptured));
    }
    landingRow += dr;
    landingCol += dc;
}
break;
}
}

return continuations;
}

function generateCapturesForPiece(board, row, col) {
    const piece = board[row][col];
    if (!piece) return [];
    const from = { row, col };
    const path = [{ row, col }];
    const current = { row, col };
    if (piece.kind === KINDS.KING) {
        return generateKingCaptures(board, from, current, piece, path, [], new Set());
    }
    return generateManCaptures(board, from, current, piece, path, [], new Set());
}

function generateSimpleMovesForPiece(board, row, col) {
    const piece = board[row][col];
    if (!piece) return [];
    const from = { row, col };
    const moves = [];

    if (piece.kind === KINDS.MAN) {
        const forward = piece.color === COLORS.WHITE ? -1 : 1;
        for (const dc of [-1, 1]) {
            const to = { row: row + forward, col: col + dc };
            if (inside(to.row, to.col) && isDarkSquare(to.row, to.col) && !board[to.row][to.col]) {
                moves.push(createMove(piece, from, [from, to], []));
            }
        }
        return moves;
    }

    for (const [dr, dc] of DIRECTIONS) {
        let toRow = row + dr;
        let toCol = col + dc;
        while (inside(toRow, toCol) && !board[toRow][toCol]) {
            moves.push(createMove(piece, from, [from, { row: toRow, col: toCol }], []));
            toRow += dr;
            toCol += dc;
        }
    }
}

```

```

    return moves;
  }

function generateMoves(board, color) {
  const captures = [];
  const simpleMoves = [];

  for (let row = 0; row < BOARD_SIZE; row += 1) {
    for (let col = 0; col < BOARD_SIZE; col += 1) {
      const piece = board[row][col];
      if (!piece || piece.color !== color) continue;
      const pieceCaptures = generateCapturesForPiece(board, row, col);
      if (pieceCaptures.length > 0) {
        captures.push(...pieceCaptures);
      } else {
        simpleMoves.push(...generateSimpleMovesForPiece(board, row, col));
      }
    }
  }

  if (captures.length > 0) {
    const maxCaptured = Math.max(...captures.map((move) => move.captured.length));
    return captures.filter((move) => move.captured.length === maxCaptured);
  }
  return simpleMoves;
}

function applyMove(board, move) {
  const nextBoard = cloneBoard(board);
  const movingPiece = clonePiece(nextBoard[move.from.row][move.from.col]);
  if (!movingPiece) {
    throw new Error("Cannot apply a move without a piece on the source square.");
  }

  nextBoard[move.from.row][move.from.col] = null;
  for (const captured of move.captured) {
    nextBoard[captured.row][captured.col] = null;
  }

  const crowned = movingPiece.kind === KINDS.MAN && move.to.row === promotionRow(movingPiece.color);
  if (crowned) {
    movingPiece.kind = KINDS.KING;
  }
  nextBoard[move.to.row][move.to.col] = movingPiece;

  return {
    board: nextBoard,
    move: {
      ...move,
      piece: clonePiece(movingPiece),
      crowned,
    },
  };
}

function getGameOutcome(board, sideToMove) {
  const counts = countPieces(board);
  if (counts.white === 0) {
    return { winner: COLORS.BLACK, reason: "white-no-pieces" };
  }
}

```

```

if (counts.black === 0) {
  return { winner: COLORS.WHITE, reason: "black-no-pieces" };
}
const legalMoves = generateMoves(board, sideToMove);
if (legalMoves.length === 0) {
  return { winner: opponent(sideToMove), reason: "no-legal-moves" };
}
return null;
}

function formatMove(move) {
  const delimiter = move.captured.length > 0 ? "x" : "-";
  return move.path.map((point) => notation(point.row, point.col)).join(delimiter);
}

function boardToSignature(board, sideToMove) {
  const pieces = [];
  for (let row = 0; row < BOARD_SIZE; row += 1) {
    for (let col = 0; col < BOARD_SIZE; col += 1) {
      const piece = board[row][col];
      if (piece) {
        pieces.push(`${piece.color[0]}${piece.kind[0]}${notation(row, col)}`);
      }
    }
  }
  return `${sideToMove}:${pieces.join(".")}`;
}

return {
  BOARD_SIZE,
  COLORS,
  KINDS,
  opponent,
  inside,
  isDarkSquare,
  promotionRow,
  emptyBoard,
  cloneBoard,
  setPiece,
  createInitialBoard,
  countPieces,
  notation,
  generateMoves,
  generateCapturesForPiece,
  applyMove,
  getGameOutcome,
  formatMove,
  boardToSignature,
};
});

```

Файл simulator.js

```

const { performance } = require("node:perf_hooks");
const Rules = require("../rules.js");
const AI = require("../ai.js");

const ALGORITHMS = Object.freeze(["random", "minimax", "alphabeta", "mcts", "unified"]);
const DEFAULTS = Object.freeze({

```

```

games: 1,
maxPly: 300,
maxNoProgressPly: 80,
repetitionLimit: 3,
randomOpeningPlies: 0,
seed: 1,
includeMoveLogs: true,
swapSides: false,
white: Object.freeze({ algorithm: "random", depth: 3, simulations: 10000 }),
black: Object.freeze({ algorithm: "alphabeta", depth: 4, simulations: 10000 }),
});

function simulateMatch(config) {
  const settings = normalizeMatchConfig(config);
  const random = createSeededRandom(settings.seed);
  const startedAt = performance.now();
  const games = [];

  for (let index = 0; index < settings.games; index += 1) {
    const pairing = createPairing(settings, index);
    games.push(
      simulateGame({
        gameNumber: index + 1,
        white: pairing.white,
        black: pairing.black,
        settings,
        random,
      })
    );
  }

  return {
    createdAt: new Date().toISOString(),
    elapsedMs: performance.now() - startedAt,
    settings: sanitizeSettings(settings),
    summary: summarizeMatch(games),
    games,
  };
}

function simulateGame({ gameNumber, white, black, settings, random }) {
  let board = Rules.createInitialBoard();
  let sideToMove = Rules.COLORS.WHITE;
  let legalMoves = Rules.generateMoves(board, sideToMove);
  let outcome = null;
  let noProgressPly = 0;
  let maxCapture = 0;
  let totalCaptures = 0;
  let promotions = 0;
  const moveLog = [];
  const repetition = new Map();
  const resources = {
    white: createEmptyResourceTotals(),
    black: createEmptyResourceTotals(),
    total: createEmptyResourceTotals(),
  };
  const startedAt = performance.now();

  incrementRepetition(repetition, Rules.boardToSignature(board, sideToMove));

```

```

for (let ply = 0; ply < settings.maxPly; ply += 1) {
  outcome = Rules.getGameOutcome(board, sideToMove);
  if (outcome) break;

  if (noProgressPly >= settings.maxNoProgressPly) {
    outcome = createDrawOutcome("no-progress-limit");
    break;
  }

  const signature = Rules.boardToSignature(board, sideToMove);
  if ((repetition.get(signature) || 0) >= settings.repetitionLimit) {
    outcome = createDrawOutcome("repetition");
    break;
  }

  legalMoves = Rules.generateMoves(board, sideToMove);
  const player = sideToMove === Rules.COLORS.WHITE ? white : black;
  const decision =
    ply < settings.randomOpeningPlies
      ? selectOpeningMove(legalMoves, random)
      : selectMove(board, sideToMove, legalMoves, player, random);

  if (!decision.move) {
    outcome = { winner: Rules.opponent(sideToMove), reason: "no-legal-moves" };
    break;
  }

  const evaluationBefore = AI.evaluatePosition(board, sideToMove);
  const applied = Rules.applyMove(board, decision.move);
  const evaluationAfter = AI.evaluatePosition(applied.board, Rules.opponent(sideToMove));
  const captured = applied.move.captured.length;
  totalCaptures += captured;
  maxCapture = Math.max(maxCapture, captured);
  if (applied.move.crowned) promotions += 1;
  noProgressPly = captured > 0 || applied.move.crowned ? 0 : noProgressPly + 1;

  const entry = {
    ply: ply + 1,
    moveNumber: Math.floor(ply / 2) + 1,
    color: sideToMove,
    algorithm: decision.algorithm,
    notation: applied.move.notation,
    captured,
    crowned: applied.move.crowned,
    legalMoves: legalMoves.length,
    evaluationBefore: evaluationBefore.whiteScore,
    evaluationAfter: evaluationAfter.whiteScore,
    diagnostics: decision.diagnostics,
  };

  addResources(resources[sideToMove], decision.diagnostics);
  addResources(resources.total, decision.diagnostics);
  if (settings.includeMoveLogs) moveLog.push(entry);

  board = applied.board;
  sideToMove = Rules.opponent(sideToMove);
  incrementRepetition(repetition, Rules.boardToSignature(board, sideToMove));
}

if (!outcome) {

```

```

    outcome = createDrawOutcome("max-ply-limit");
  }

  const counts = Rules.countPieces(board);
  const finalEvaluation = AI.evaluatePosition(board, sideToMove);
  const durationMs = performance.now() - startedAt;
  const plies = settings.includeMoveLogs
    ? moveLog.length
    : resources.total.aiMoves;

  return {
    gameNumber,
    durationMs,
    plies,
    fullMoves: Math.ceil(plies / 2),
    outcome,
    white: { ...white },
    black: { ...black },
    finalSideToMove: sideToMove,
    finalPieces: counts,
    finalEvaluation: finalEvaluation.whiteScore,
    captures: totalCaptures,
    promotions,
    maxCapture,
    resources,
    moveLog: settings.includeMoveLogs ? moveLog : undefined,
  };
}

function selectMove(board, color, legalMoves, player, random) {
  const memoryBefore = getHeapUsed();
  const startedAt = performance.now();
  let result;

  if (player.algorithm === "minimax") {
    result = AI.selectMinimaxMove(board, color, player.depth, legalMoves);
  } else if (player.algorithm === "alphabeta") {
    result = AI.selectAlphaBetaMove(board, color, player.depth, legalMoves);
  } else if (player.algorithm === "mcts") {
    result = AI.selectMonteCarloMove(board, color, player.simulations, legalMoves, { random });
  } else if (player.algorithm === "unified") {
    result = AI.selectUnifiedMove(board, color, legalMoves, {
      targetDepth: player.targetDepth,
      nodeBudget: player.nodeBudget,
      timeMs: player.timeMs,
      quiescenceDepth: player.quiescenceDepth,
      random,
    });
  } else {
    result = {
      move: AI.selectRandomMove(legalMoves, random),
      score: 0,
      depth: 0,
      nodes: 0,
      leaves: 0,
      cutoffs: 0,
      transpositionHits: 0,
      tableSize: 0,
    };
  }
}

```

```

const memoryAfter = getHeapUsed();
return {
  move: result.move,
  algorithm: player.algorithm,
  diagnostics: normalizeDiagnostics(result, performance.now() - startedAt, memoryBefore, memoryAfter),
};
}

function selectOpeningMove(legalMoves, random) {
  const startedAt = performance.now();
  const move = AI.selectRandomMove(legalMoves, random);
  return {
    move,
    algorithm: "opening-random",
    diagnostics: normalizeDiagnostics(
      {
        move,
        score: 0,
        depth: 0,
        nodes: 0,
        leaves: 0,
        cutoffs: 0,
        transpositionHits: 0,
        tableSize: 0,
      },
      performance.now() - startedAt,
      null,
      null
    ),
  };
}

function normalizeDiagnostics(result, decisionMs, memoryBefore, memoryAfter) {
  return {
    score: finiteOrZero(result.score),
    depth: finiteOrZero(result.depth),
    nodes: finiteOrZero(result.nodes),
    leaves: finiteOrZero(result.leaves),
    cutoffs: finiteOrZero(result.cutoffs),
    transpositionHits: finiteOrZero(result.transpositionHits),
    tableSize: finiteOrZero(result.tableSize),
    iterations: finiteOrZero(result.iterations),
    simulations: finiteOrZero(result.simulations),
    requestedSimulations: finiteOrZero(result.requestedSimulations),
    simulationLimit: finiteOrZero(result.simulationLimit),
    averageSimulationPlies: finiteOrZero(result.averageSimulationPlies),
    maxSimulationPlies: finiteOrZero(result.maxSimulationPlies),
    bestVisits: finiteOrZero(result.bestVisits),
    bestWinRate: Number.isFinite(result.bestWinRate) ? result.bestWinRate : null,
    mctsUsed: Boolean(result.mctsUsed),
    mctsIterations: finiteOrZero(result.mctsIterations),
    mctsRequestedIterations: finiteOrZero(result.mctsRequestedIterations),
    mctsSimulationLimit: finiteOrZero(result.mctsSimulationLimit),
    mctsBestVisits: finiteOrZero(result.mctsBestVisits),
    mctsBestWinRate: Number.isFinite(result.mctsBestWinRate) ? result.mctsBestWinRate : null,
    mctsTimedOut: Boolean(result.mctsTimedOut),
    mctsSearchTimeMs: finiteOrZero(result.mctsSearchTimeMs),
    requestedDepth: finiteOrZero(result.requestedDepth),
    completedDepth: finiteOrZero(result.completedDepth),
  };
}

```

```

targetDepth: finiteOrZero(result.targetDepth),
phase: result.phase || "",
complexity: finiteOrZero(result.complexity),
quiescenceNodes: finiteOrZero(result.quiescenceNodes),
aborted: Boolean(result.aborted),
strategy: result.strategy || "",
nodeBudget: finiteOrZero(result.nodeBudget),
timeMs: finiteOrZero(result.timeMs),
decisionMs,
memoryDeltaBytes:
  Number.isFinite(memoryBefore) && Number.isFinite(memoryAfter)
    ? memoryAfter - memoryBefore
    : null,
};
}

function summarizeMatch(games) {
  const summary = {
    games: games.length,
    whiteWins: 0,
    blackWins: 0,
    draws: 0,
    resultReasons: {},
    totalPlies: 0,
    averagePlies: 0,
    minPlies: null,
    maxPlies: 0,
    totalCaptures: 0,
    averageCaptures: 0,
    promotions: 0,
    maxCaptureInMove: 0,
    averageFinalWhiteScore: 0,
    minFinalWhiteScore: null,
    maxFinalWhiteScore: null,
    durationMs: {
      total: 0,
      average: 0,
      min: null,
      max: 0,
    },
    resources: {
      white: createEmptyResourceTotals(),
      black: createEmptyResourceTotals(),
      total: createEmptyResourceTotals(),
    },
    algorithms: {},
  };

  for (const game of games) {
    if (game.outcome.winner === Rules.COLORS.WHITE) summary.whiteWins += 1;
    if (game.outcome.winner === Rules.COLORS.BLACK) summary.blackWins += 1;
    if (!game.outcome.winner) summary.draws += 1;
    summary.resultReasons[game.outcome.reason] = (summary.resultReasons[game.outcome.reason] || 0) + 1;
    summary.totalPlies += game.plies;
    summary.minPlies = summary.minPlies === null ? game.plies : Math.min(summary.minPlies, game.plies);
    summary.maxPlies = Math.max(summary.maxPlies, game.plies);
    summary.totalCaptures += game.captures;
    summary.promotions += game.promotions;
    summary.maxCaptureInMove = Math.max(summary.maxCaptureInMove, game.maxCapture);
    summary.minFinalWhiteScore =

```

```

summary.minFinalWhiteScore === null
  ? game.finalEvaluation
  : Math.min(summary.minFinalWhiteScore, game.finalEvaluation);
summary.maxFinalWhiteScore =
  summary.maxFinalWhiteScore === null
    ? game.finalEvaluation
    : Math.max(summary.maxFinalWhiteScore, game.finalEvaluation);
summary.averageFinalWhiteScore += game.finalEvaluation;
summary.durationMs.total += game.durationMs;
summary.durationMs.min =
  summary.durationMs.min === null ? game.durationMs : Math.min(summary.durationMs.min, game.durationMs);
summary.durationMs.max = Math.max(summary.durationMs.max, game.durationMs);

addResources(summary.resources.white, game.resources.white);
addResources(summary.resources.black, game.resources.black);
addResources(summary.resources.total, game.resources.total);
addAlgorithmGame(summary.algorithms, game, Rules.COLORS.WHITE);
addAlgorithmGame(summary.algorithms, game, Rules.COLORS.BLACK);
}

if (games.length > 0) {
  summary.averagePlies = summary.totalPlies / games.length;
  summary.averageCaptures = summary.totalCaptures / games.length;
  summary.averageFinalWhiteScore /= games.length;
  summary.durationMs.average = summary.durationMs.total / games.length;
}

finalizeResourceTotals(summary.resources.white);
finalizeResourceTotals(summary.resources.black);
finalizeResourceTotals(summary.resources.total);
for (const item of Object.values(summary.algorithms)) {
  finalizeAlgorithmSummary(item);
}

return summary;
}

function addAlgorithmGame(algorithms, game, color) {
  const player = game[color];
  const key = algorithmKey(player);
  if (!algorithms[key]) {
    algorithms[key] = {
      algorithm: player.algorithm,
      depth: player.depth,
      simulations: player.simulations,
      appearances: 0,
      asWhite: 0,
      asBlack: 0,
      wins: 0,
      losses: 0,
      draws: 0,
      resources: createEmptyResourceTotals(),
      averageSearchMsPerMove: 0,
      averageNodesPerMove: 0,
    };
  }

  const item = algorithms[key];
  item.appearances += 1;
  if (color === Rules.COLORS.WHITE) item.asWhite += 1;

```

```

if (color === Rules.COLORS.BLACK) item.asBlack += 1;

if (!game.outcome.winner) {
  item.draws += 1;
} else if (game.outcome.winner === color) {
  item.wins += 1;
} else {
  item.losses += 1;
}

addResources(item.resources, game.resources[color]);
}

function finalizeAlgorithmSummary(item) {
  finalizeResourceTotals(item.resources);
  item.averageSearchMsPerMove = item.resources.aiMoves > 0 ? item.resources.searchMs / item.resources.aiMoves :
0;
  item.averageNodesPerMove = item.resources.aiMoves > 0 ? item.resources.nodes / item.resources.aiMoves : 0;
}

function createEmptyResourceTotals() {
  return {
    aiMoves: 0,
    searchMs: 0,
    nodes: 0,
    leaves: 0,
    cutoffs: 0,
    transpositionHits: 0,
    tableSizeMax: 0,
    iterations: 0,
    simulations: 0,
    memoryDeltaBytes: 0,
    memorySamples: 0,
    averageSearchMsPerMove: 0,
    nodesPerSecond: 0,
  };
}

function addResources(target, source) {
  if (!source) return;
  const sourceIsTotals = Object.prototype.hasOwnProperty.call(source, "aiMoves");
  target.aiMoves += sourceIsTotals ? source.aiMoves : 1;
  target.searchMs += finiteOrZero(source.searchMs ?? source.decisionMs);
  target.nodes += finiteOrZero(source.nodes);
  target.leaves += finiteOrZero(source.leaves);
  target.cutoffs += finiteOrZero(source.cutoffs);
  target.transpositionHits += finiteOrZero(source.transpositionHits);
  target.tableSizeMax = Math.max(target.tableSizeMax, finiteOrZero(source.tableSizeMax ?? source.tableSize));
  target.iterations += finiteOrZero(source.iterations);
  target.simulations += finiteOrZero(source.simulations);
  if (sourceIsTotals) {
    target.memoryDeltaBytes += finiteOrZero(source.memoryDeltaBytes);
    target.memorySamples += finiteOrZero(source.memorySamples);
  } else if (Number.isFinite(source.memoryDeltaBytes)) {
    target.memoryDeltaBytes += source.memoryDeltaBytes;
    target.memorySamples += 1;
  }
}

function finalizeResourceTotals(totals) {

```

```

totals.averageSearchMsPerMove = totals.aiMoves > 0 ? totals.searchMs / totals.aiMoves : 0;
totals.nodesPerSecond = totals.searchMs > 0 ? totals.nodes / (totals.searchMs / 1000) : 0;
}

function normalizeMatchConfig(config) {
  const input = config || {};
  return {
    games: clampInteger(input.games, 1, 100000, DEFAULTS.games),
    maxPly: clampInteger(input.maxPly, 1, 10000, DEFAULTS.maxPly),
    maxNoProgressPly: clampInteger(input.maxNoProgressPly, 1, 10000, DEFAULTS.maxNoProgressPly),
    repetitionLimit: clampInteger(input.repetitionLimit, 2, 20, DEFAULTS.repetitionLimit),
    randomOpeningPlies: clampInteger(input.randomOpeningPlies, 0, 100, DEFAULTS.randomOpeningPlies),
    seed: clampInteger(input.seed, 1, 2147483647, DEFAULTS.seed),
    includeMoveLogs: input.includeMoveLogs !== false,
    swapSides: input.swapSides === true,
    white: normalizePlayerConfig(input.white || DEFAULTS.white),
    black: normalizePlayerConfig(input.black || DEFAULTS.black),
  };
}

function normalizePlayerConfig(player) {
  const algorithm = ALGORITHMS.includes(player.algorithm) ? player.algorithm : "random";
  return {
    algorithm,
    depth: clampInteger(player.depth, 1, 30, 3),
    simulations: clampInteger(player.simulations, 1, 1000000, 10000),
    targetDepth: optionalInteger(player.targetDepth, 1, 30),
    nodeBudget: optionalInteger(player.nodeBudget, 1000, 10000000),
    timeMs: optionalInteger(player.timeMs, 50, 600000),
    quiescenceDepth: optionalInteger(player.quiescenceDepth, 0, 24),
  };
}

function createPairing(settings, gameIndex) {
  if (!settings.swapSides || gameIndex % 2 === 0) {
    return {
      white: settings.white,
      black: settings.black,
    };
  }
  return {
    white: settings.black,
    black: settings.white,
  };
}

function sanitizeSettings(settings) {
  return {
    games: settings.games,
    maxPly: settings.maxPly,
    maxNoProgressPly: settings.maxNoProgressPly,
    repetitionLimit: settings.repetitionLimit,
    randomOpeningPlies: settings.randomOpeningPlies,
    seed: settings.seed,
    includeMoveLogs: settings.includeMoveLogs,
    swapSides: settings.swapSides,
    white: { ...settings.white },
    black: { ...settings.black },
  };
}

```

```

function algorithmKey(player) {
  if (player.algorithm === "mcts") return `${player.algorithm}:s${player.simulations}`;
  if (player.algorithm === "unified") return "unified:auto";
  if (player.algorithm === "random") return "random";
  return `${player.algorithm}:d${player.depth}`;
}

function incrementRepetition(map, signature) {
  map.set(signature, (map.get(signature) || 0) + 1);
}

function createDrawOutcome(reason) {
  return { winner: null, reason };
}

function createSeededRandom(seed) {
  let state = seed >>> 0;
  return function random() {
    state += 0x6d2b79f5;
    let value = state;
    value = Math.imul(value ^ (value >>> 15), value | 1);
    value ^= value + Math.imul(value ^ (value >>> 7), value | 61);
    return ((value ^ (value >>> 14)) >>> 0) / 4294967296;
  };
}

function clampInteger(value, min, max, fallback) {
  const parsed = Number.parseInt(value, 10);
  if (!Number.isFinite(parsed)) return fallback;
  return Math.min(max, Math.max(min, parsed));
}

function optionalInteger(value, min, max) {
  const parsed = Number.parseInt(value, 10);
  if (!Number.isFinite(parsed)) return undefined;
  return Math.min(max, Math.max(min, parsed));
}

function finiteOrZero(value) {
  return Number.isFinite(value) ? value : 0;
}

function getHeapUsed() {
  return typeof process !== "undefined" && process.memoryUsage ? process.memoryUsage().heapUsed : null;
}

module.exports = {
  ALGORITHMS,
  DEFAULTS,
  simulateMatch,
  simulateGame,
  normalizeMatchConfig,
  createSeededRandom,
};

```