СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Факультет ___інформаційних технологій та електроніки_____

Кафедра _____інформаційних технологій та програмування_____

Пояснювальна записка

до магістерської дипломної роботи

_____магістр_____
(освітньо-кваліфікаційний рівень)

на тему: ___Дослідження методів створення адаптивного штучного _____ інтелекту для покрокових ігор _____

Виконав: студент 2 курсу, групи ІСТ-23дм_ 126 «Інформаційні системи та технології___
(шифр і назва спеціальності)
_____Сниченков Д. А._____
(прізвище та ініціали)
Керівник _____Дьомін М. К._____
(прізвище та ініціали)
Рецензент_____Меняйленко О. С._____
(прізвище та ініціали)

Київ – 2024 року

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Факультет ___інформаційних технологій та електроніки_____
Кафедра _____інформаційних технологій та програмування_____
Освітньо-кваліфікаційний рівень __магістр_____
Спеціальність 126 «Інформаційні системи та технології»_____
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТП
_____ д.т.н., доц. Захожай О.І.
(підпис)
« ___ » _____2024 р.

ЗАВДАННЯ

на магістерську дипломну роботу студенту
_____Сниченков Даніїл Андрійович_____
(прізвище, ім'я, по батькові)

1.Тема роботи: Дослідження методів створення адаптивного штучного інтелекту для покрокових ігор,_____
керівник роботи к.т.н., доцент, Дьомін Максим Костянтинович,
(вчене звання, науковий ступінь, прізвище, ім'я, по батькові)
затверджені наказом університету від «_6_»_12_ 2024 року №361/15.15-С_
2. Строк подання студентом роботи: 12 грудня 2024 р._____
3. Вихідні дані до роботи: Матеріали науково-дослідної практики, науково-методична література; дані інтернет-мережі ._____
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

4.1 Вступ

4.2 Аналітичний огляд питання створення адаптивного штучного інтелекту для комп'ютерних ігор (огляд публічних джерел інформації)

4.3 Основна частина: висвітлення методів, які будуть використовуватися для реалізації проєкту (розгляд алгоритмів, їх адаптація до гри Quoridor).

4.4 Практична частина: огляд технологій, які використовуються під час реалізації проєкту (мінімакс, альфа-бета відсікання, A*, функція оцінки)

4.5 Висновки

4.6 Перелік використаних джерел

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

_____

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Розділ 1. Аналіз існуючих методів створення адаптивного штучного інтелекту	Дьомін М. К., к.т.н., доцент		
Розділ 2. Проектування та реалізація ШІ для гри Quoridor	Дьомін М. К., к.т.н., доцент		
Розділ 3. Тестування та аналіз ефективності ШІ	Дьомін М. К., к.т.н., доцент		

7. Дата видачі завдання __5 листопада 2024р._____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1.	Одержання завдання на виконання роботи	20.10.2024	
2.	Укладання і погодження з керівником плану і етапів виконання роботи	24.10.2024	
3.	Узагальнення даних літературних джерел	28.10.2024	
4.	Аналіз шляхів виконання завдання. Вибір і погодження з керівником оптимального шляху виконання завдання	01.11.2024	
5.	Аналіз технічних засобів та існуючих систем	07.11.2024	
6.	Реалізація практичної частини завдання	24.11.2024	
7.	Укладання, оформлення та погодження пояснювальної записки з керівником	11.12.2024	
8.	Надання пояснювальної записки на кафедру	12.12.2024	
9.	Підготовка доповіді та презентації	12.12.2024	

Студент _____ Сниченков Д. А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Дьомін М. К.
(підпис) (прізвище та ініціали)

ЗМІСТ

ВСТУП

Розвиток штучного інтелекту в сучасній індустрії ігор є ключовим фактором, що впливає на якість ігрового процесу та рівень взаємодії між гравцем і комп'ютерним супротивником. Особливе місце серед жанрів займають покрокові ігри, де ШІ повинен не лише ефективно виконувати роль супротивника, але й демонструвати стратегічну гнучкість, адаптуючись до стилю гри користувача. У цих умовах традиційні методи реалізації ШІ, які базуються на жорстко заданих алгоритмах, стають недостатньо ефективними, оскільки їхня поведінка є передбачуваною та обмеженою.

Проблема створення адаптивного штучного інтелекту, який здатний змінювати свою поведінку залежно від дій гравця, є актуальною для забезпечення унікального ігрового досвіду. Такий підхід дозволяє ШІ реагувати на різні стратегії, виявляти слабкі сторони опонента та оптимізувати власну гру в реальному часі. Гра Quoridor, завдяки своїй простій, але глибокій механіці, є ідеальним полем для дослідження та впровадження таких рішень. Ця гра вимагає від супротивників стратегічного мислення, аналізу ситуації та передбачення ходів опонента, що робить задачу створення адаптивного ШІ особливо цікавою та складною.

Метою роботи є розробка штучного інтелекту для гри Quoridor, який демонструватиме адаптивну поведінку та зможе ефективно протистояти гравцю. У процесі дослідження будуть розглянуті існуючі методи створення ШІ, обґрунтовано вибір алгоритму, що підходить для адаптації, а також розроблено і протестовано готову систему ШІ.

Практична значущість дослідження полягає в тому, що розроблений адаптивний ШІ може бути застосований не лише у грі Quoridor, а й адаптований для інших покрокових ігор, що мають схожі механіки.

Впровадження таких систем дозволяє підвищити рівень реалістичності та інтересу до гри, що є важливим аспектом розвитку сучасної ігрової індустрії.

РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ СТВОРЕННЯ АДАПТИВНОГО ШТУЧНОГО ІНТЕЛЕКТУ

1.1. Огляд класичних методів створення штучного інтелекту

Розробка штучного інтелекту (ШІ) для покрокових ігор базується на використанні перевірених часом алгоритмів, які аналізують стан гри, оцінюють можливі дії та обирають оптимальні стратегії. Серед найпоширеніших методів виділяють алгоритм мінімакс з альфа-бета відсіканням та різноманітні алгоритми пошуку такі як пошук в глибину і ширину, та алгоритм A*. Ці підходи забезпечують ефективну гру комп'ютера та слугують фундаментом для розробки складніших систем.

Алгоритм мінімакс широко використовується у стратегічних іграх, зокрема у шахах, шашках та Quoridor. Його суть полягає у знаходженні оптимального ходу для ШІ через аналіз усіх можливих сценаріїв розвитку гри. Алгоритм передбачає, що супротивник завжди діє раціонально та прагне мінімізувати виграш ШІ [1].

Мінімакс працює наступним чином:

1. Побудова дерева станів гри: кожен вузол дерева відповідає можливому стану гри після чергового ходу.
2. Оцінка кінцевих вузлів: застосовується функція оцінки, яка визначає "вигідність" конкретного стану для кожного з гравців.
3. Проходження дерева: алгоритм аналізує вузли знизу вгору, визначаючи найкращий хід для гравця на рівнях "максимізації" та супротивника — на рівнях "мінімізації".

Візуалізація алгоритму представлена на рисунку 1.1.

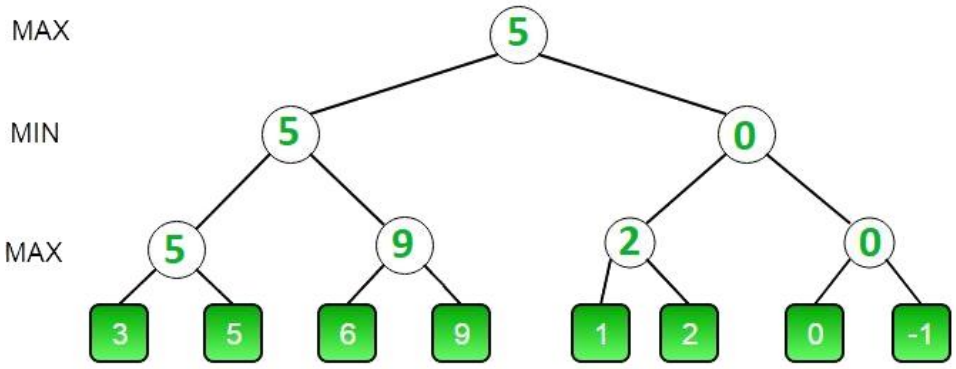

Рисунок 1.1 – Візуалізація алгоритму мінімакс

Попри ефективність у простих іграх, мінімакс має обмеження: його складність зростає експоненційно зі збільшенням глибини аналізу. У складних іграх, таких як Quoridor, з великою кількістю можливих ходів, цей алгоритм потребує оптимізацій.

Щоб скоротити обчислювальні витрати, застосовується альфа-бета відсікання — оптимізація алгоритму мінімакс. Цей метод дозволяє уникнути обробки гілок дерева, які не впливають на кінцевий результат.

Принцип роботи альфа-бета відсікання:

- Альфа — найкраще знайдене значення для гравця, що максимізує свій виграш.
- Бета — найкраще значення для гравця, що мінімізує виграш суперника.
- Якщо під час обчислення виявляється, що подальший аналіз вузла не змінить загального результату (значення вже перевищує альфа чи менше за бета), гілка "відсікається".

Використання альфа-бета відсікання дозволяє скоротити час виконання мінімаксу до 50-75%, залежно від структури дерева та глибини мінімаксу.

Цей метод особливо ефективний для ігор з великою кількістю можливих ходів, таких як Quoridor [2].

Візуалізація алгоритму представлена на рисунку 1.2.

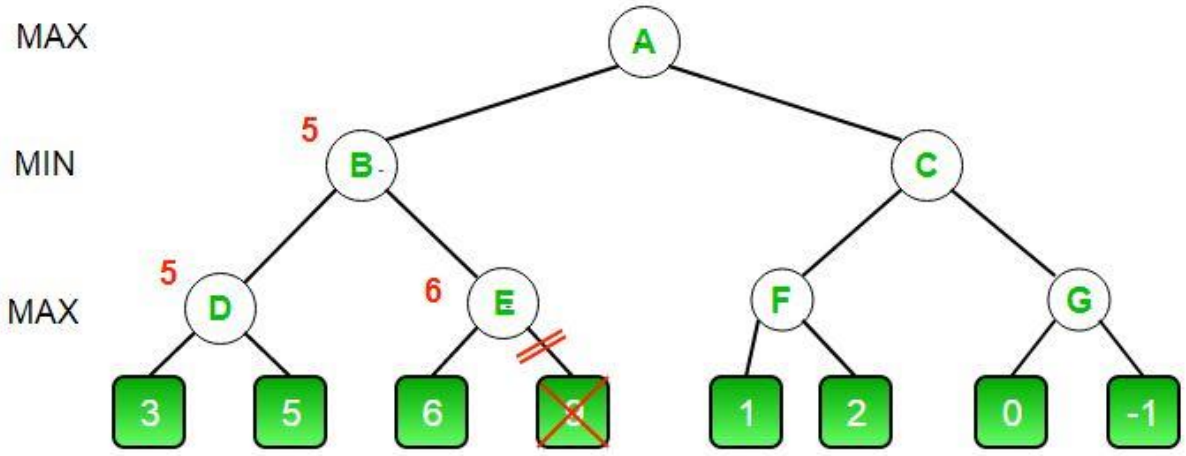

Рисунок 1.2 – Візуалізація алгоритму альфа-бета відсікання

Евристики відіграють важливу роль у скороченні простору пошуку. Для цього застосовуються функції оцінки, які визначають цінність конкретного стану гри. Ефективна функція оцінки дозволяє алгоритмам швидше знаходити оптимальні ходи, не обробляючи весь простір можливих станів [3].

Класичні методи, такі як мінімакс та альфа-бета відсікання, забезпечують базу для створення ШІ у покрокових іграх. Однак їхньою головною слабкістю є відсутність адаптивності та обмеженість у роботі з великими просторами пошуку. Для гри Quoridor ці методи можуть бути доповнені сучасними підходами, такими як навчання з підкріпленням чи еволюційні алгоритми, що дозволяє створити адаптивний ШІ.

1.2. Підходи до створення адаптивного штучного інтелекту

Адаптивність штучного інтелекту (ШІ) у покрокових іграх дозволяє йому реагувати на різні стилі гри супротивника, змінювати стратегію залежно від ситуації та ефективно балансувати між точністю й обчислювальною складністю. Один із практичних способів впровадження адаптивності в умовах обмежених ресурсів — це оптимізація параметрів класичних алгоритмів, таких як мінімакс, зокрема через динамічне регулювання глибини пошуку.

Евристичні функції оцінки відіграють ключову роль у створенні адаптивності ШІ. Вони визначають, наскільки привабливим є поточний стан гри для кожного гравця. Для гри Quoridor евристики можуть враховувати:

- Відстань до цільової лінії.
- Кількість доступних парканів.

Евристики дозволяють мінімізувати потребу в глибокому аналізі, зосереджуючись на найбільш перспективних діях.

Адаптація класичних алгоритмів, таких як мінімакс, шляхом регулювання вагів функції оцінки є практичним підходом для створення адаптивного ШІ у покрокових іграх. Для гри Quoridor це дозволяє реалізувати інтелект, який ефективно балансує між складністю рішень і швидкістю виконання, забезпечуючи конкурентний рівень гри навіть за обмежених обчислювальних ресурсів.

1.3. Приклади адаптивного штучного інтелекту в іграх

Адаптивний штучний інтелект (ШІ) стає важливою частиною сучасних відеоігор, забезпечуючи динамічний та інтуїтивно зрозумілий ігровий процес.

Такий ШІ здатен реагувати на зміни в стилі гри користувача, підлаштовуючи свої стратегії відповідно до дій гравця [4]. Різні підходи до адаптації використовуються у численних іграх для забезпечення різноманітних рівнів складності та інтерактивності. Ось кілька прикладів подібного ШІ:

1. Шахи

Шахи є однією з найбільш поширених ігор для демонстрації ШІ, ідеальним прикладом служить програма Deep Blue, яка перемогла Гаррі Каспарова, чемпіона світу з шахів [5]. У шахах застосовуються алгоритми, зокрема мінімакс та його оптимізація альфа-бета відсікання, для аналізу всіх можливих ходів та вибору найкращого варіанту.

Адаптивність ШІ в шахах може проявлятися у зміні глибини пошуку чи налаштувань функції оцінки в залежності від рівня суперника. У випадку з Deep Blue, система могла змінювати стратегію в залежності від дій супротивника. Наприклад, якщо гравець допускав прості помилки, ШІ могло збільшувати глибину пошуку для точного аналізу ситуації, а якщо суперник грав без помилок, алгоритм використовував менше ресурсів для збереження обчислювальної ефективності.

2. Go (AlphaGo)

Go є однією з найскладніших стратегічних ігор для ШІ через величезну кількість можливих комбінацій ходів. AlphaGo, створена компанією DeepMind, є прикладом використання сучасного адаптивного ШІ для гри в Go [6]. На відміну від традиційних шахових програм, AlphaGo використовує нейронні мережі для вивчення і оптимізації своїх стратегій.

ШІ у Go може адаптувати свою гру, вчиться з кожної партії, що дозволяє змінювати стратегії в залежності від поведінки суперника. AlphaGo використовує методи глибокого навчання та аналізує величезну кількість варіантів, роблячи прогнози щодо кроків суперника. Її адаптивність полягає в

здатності змінювати стратегію на ходу, реагуючи на поведінку гравця, і вчитися на базі попередніх ігор для досягнення оптимальних результатів.

3. F.E.A.R. (First Encounter Assault Recon)

F.E.A.R. є шутером від першої особи, в якому використовується адаптивний ШІ для створення інтелектуальних ворогів. Один із найбільш цікавих аспектів цієї гри — це адаптація поведінки ворогів в залежності від дій гравця. ШІ не просто рухається по заготовлених траєкторіях, а адаптується до стратегії гравця в реальному часі [7].

Наприклад, якщо гравець часто використовує укриття або шукає стратегічні позиції для стрільби, вороги починають шукати способи обійти ці укриття або використовувати оточення для атаки з іншого напрямку. Також якщо гравець займає оборонні позиції, вороги можуть почати спільно діяти, координуючи свої атаки, щоб перевершити гравця.

Це адаптивне поведінкове моделювання дозволяє створити відчуття, що вороги "розуміють" стратегію гравця і змінюють свої тактики, що робить гру більш захоплюючою та непередбачуваною.

4. Left 4 Dead

У цьому кооперативному шутері використовується система "AI Director", яка забезпечує адаптивний рівень складності в реальному часі. Система AI Director стежить за поведінкою гравців і змінює інтенсивність гри в залежності від того, як добре чи погано гравці справляються з хвилями зомбі. Якщо гравці чудово працюють разом і знищують ворогів ефективно, AI Director підвищує складність, додаючи більше ворогів або змінюючи типи супротивників.

З іншого боку, якщо команда починає зазнавати труднощів і часто вмирає, ШІ зменшує кількість ворогів, а також може змінити сценарії для полегшення гри [8]. Такий підхід створює інтерактивний і динамічний досвід, де рівень складності постійно підлаштовується під здібності гравців,

що забезпечує збереження інтересу і не дає гри стати надто легкою або надто важкою.

Ці приклади показують, як адаптивний ШІ в іграх допомагає створювати більш гнучкі та захоплюючі ігрові досвіди. ШІ не просто реагує на поточний стан гри, а й змінює стратегії, враховуючи стиль гри користувача, що робить кожну гру унікальною та непередбачуваною. Адаптивність ШІ значно покращує інтерактивність та інтерес до гри, адже вона може підлаштовуватися під потреби та здібності гравця, забезпечуючи виклик та задоволення від кожної ігрової сесії.

ВИСНОВКИ ДО РОЗДІЛУ 1

У першому розділі було проведено детальний аналіз існуючих методів створення адаптивного штучного інтелекту для покрокових ігор, таких як Quoridor. Огляд класичних алгоритмів штучного інтелекту, таких як мінімакс, альфа-бета обрізка, а також сучасних підходів до адаптації цих алгоритмів, показав, що класичні методи мають обмеження щодо адаптивності і здатності обробляти великі простори пошуку. Зокрема, вони вимагають значних обчислювальних ресурсів для досягнення оптимальних результатів, що не завжди є практичним у реальних ігрових умовах.

Модернізація класичних алгоритмів, зокрема адаптація мінімакс-алгоритму за допомогою динамічного налаштування функції оцінки, є ефективним рішенням для створення адаптивного штучного інтелекту. Це дозволяє зберегти баланс між складністю гри і ефективністю обчислень, підвищуючи рівень виклику для гравців.

Також розглянуто приклади застосування адаптивних ШІ в інших іграх, що продемонструвало різноманітність підходів до адаптації штучного інтелекту в залежності від стилю гри та рівня складності. Використання адаптивного мінімакса у грі Quoridor було проаналізовано з урахуванням конкретних особливостей цієї гри, таких як стратегічне розміщення бар'єрів та рух фігур. Це дає можливість створити більш реалістичні та непередбачувані сценарії, що підвищують цікавість і виклик для гравців.

Таким чином, у першому розділі було показано, що адаптивний підхід до мінімакс-алгоритму може значно поліпшити функціонування штучного інтелекту в покрокових іграх, надаючи гравцям більш цікавий і варіативний досвід. Визначено, що для подальших досліджень важливо розвивати методи адаптації та оптимізації алгоритмів для підвищення їх ефективності та здатності до взаємодії з гравцями з різними стилями гри.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ШІ ДЛЯ ГРИ QUORIDOR

2.1. Особливості гри Quoridor та вимоги до ШІ

Гра Quoridor є стратегічною покроковою грою для двох гравців, мета якої полягає в тому, щоб один з гравців першим досягнув своєї цільової лінії на протилежному боці поля. Гравці переміщують своїх фігур на квадратному полі 9×9 клітин та мають можливість будувати парканчики, що перешкоджають просуванню супротивника. Кожен гравець володіє обмеженою кількістю таких парканів, що накладає додаткові обмеження на стратегію гри. У грі є два основних компоненти: переміщення фігур та стратегічне використання парканів для блокування супротивника або створення власного шляху до мети [9].

Основні характеристики гри:

1. Поле гри – це квадратна сітка розміром 9×9, на якій гравці переміщують свої фігури та розташовують паркани. Простір обмежений, тому кожне рішення, яке приймає гравець, має велике значення.
2. Механіка руху фігур – гравці можуть переміщати свої фігури на одну клітину за хід у вертикальному чи горизонтальному напрямку. Переміщення не може відбуватися через парканчики, тому стратегічне їх розташування є ключовим елементом гри.
3. Парканчики – кожен гравець має певну кількість парканів (зазвичай 10), які можна розмістити на полі для блокування руху супротивника. Важливість цієї механіки полягає в тому, що паркан не лише блокує шлях, але й змушує гравця планувати заздалегідь, щоб уникнути пасток і знаходити оптимальні маршрути.

4. Кінець гри – гра завершується, коли один з гравців досягає своєї цільової лінії, тобто проходить весь шлях до протилежного боку поля.

Приклад партії у Quoridor представлений на рисунку 2.1.

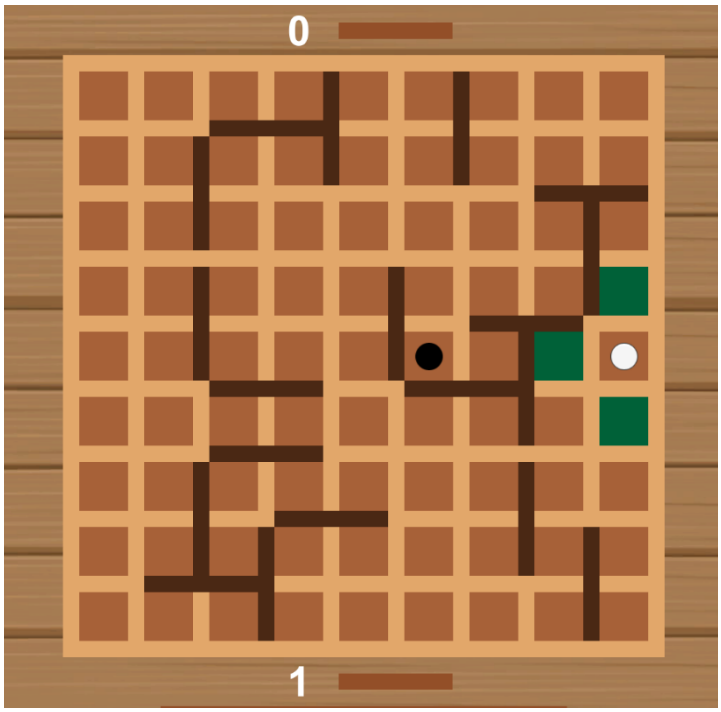

Рисунок 2.1 – Приклад партії у Quoridor

Перед розробкою алгоритму для штучного інтелекту, важливо оцінити складність гри Quoridor, оскільки вона впливає на вибір методів і підходів до реалізації. Два ключові параметри складності — це розмір простору станів та складність дерева гри.

Розмір простору станів визначається кількістю можливих позицій на ігровому полі, включаючи розташування фішок і парканів. Хоча врахування всіх можливих станів, включаючи недійсні, є складним, верхню межу можна оцінити:

Гравець має 81 можливу позицію для першої фішки, після чого залишається 80 для другої. Таким чином:

$$S_p = 81 * 80 = 6480$$

Один паркан займає дві клітинки, і його можна розташувати горизонтально чи вертикально. У кожному ряду чи колонці доступно 64

можливі позиції для одного паркану. Враховуючи всі можливі комбінації, включаючи взаємний вплив парканів, загальна кількість позицій для парканів оцінюється як:

$$S_f = \sum_{i=0}^{20} \prod_{j=0}^{i} (128 - 4i) \approx 6.1582 * 10^{38}$$

Щоб знайти загальний розмір простору станів SSS, потрібно перемножити кількість станів для фішок і парканів:

$$S = S_f * S_f \approx 6480 * 6.1582 * 10^{38} \approx 3.9905 * 10^{42}$$

Складність дерева гри визначається загальною кількістю можливих ігор. Це оцінюється за допомогою середнього фактору розгалуження (кількість можливих ходів у кожній позиції) та середньої довжини гри (кількість пліїв — кроків обох гравців).

1. Фактор розгалуження:

У середньому кожна позиція має 60.4 можливих ходів.

2. Середня довжина гри:

Середня гра триває близько 91.1 ходів.

3. Складність дерева:

Загальну кількість можливих ігор можна оцінити як:

$$G = (60.4)^{91.1} \approx 1.7884 * 10^{162}$$

Складність Quoridor можна порівняти з іншими відомими іграми, наприклад, шахами та Ґо, використовуючи логарифми цих параметрів. Порівняння ігор представлено в таблиці 2.1.

Таблиця 2.1 – Складність гри у порівнянні з іншими покроковими іграми

Гра	log(простір станів)	log(дерево гри)
Tic-tac-toe	3	5
Шахи	46	123
Go	172	350
Quoridor	42	162

Як видно, Quoridor має подібний розмір простору станів до шахів, але складність дерева гри перевищує шахи, що свідчить про вищу складність у моделюванні всіх можливих ігор.

Quoridor належить до ігор IV категорії складності, які характеризуються високою складністю як простору станів, так і дерева гри. Це робить Quoridor складним для повного аналізу або "розв'язання" за допомогою традиційних методів. Для ефективного створення штучного інтелекту необхідно застосовувати алгоритми, які поєднують оптимізацію та адаптацію [10].

2.2. Вибір алгоритму для реалізації ШІ

Реалізація штучного інтелекту (ШІ) для гри Quoridor потребує ретельного аналізу доступних алгоритмів і їхньої адаптації до специфіки гри. У Quoridor ШІ повинен ефективно вирішувати дві основні задачі: пошук найкоротшого шляху для досягнення мети та прийняття стратегічних рішень для забезпечення переваги над супротивником. У цьому розділі ми розглянемо кілька популярних алгоритмів, їх переваги та недоліки, а також обґрунтуємо вибір комбінації алгоритму A* для пошуку шляху і модифікованого мінімаксу із альфа-бета відсіканням та адаптивною функцією оцінки.

Quoridor має дві ключові особливості, які суттєво впливають на вибір алгоритмів:

1. Динамічність поля гри: Гравці можуть змінювати доступні маршрути, встановлюючи паркани, що потребує від ШІ швидкої переоцінки ситуації.
2. Стратегічність і довгострокове планування: Ефективний ШІ повинен враховувати не тільки прямий шлях до фінішної лінії, але й можливі дії супротивника, такі як блокування або створення пасток.

A* (A-star) є одним із найкращих алгоритмів для задач пошуку шляху завдяки його здатності враховувати поточну вартість шляху та передбачати майбутні витрати через евристичну функцію [11]. В контексті Quoridor він забезпечує точний і швидкий пошук найкоротшого шляху від поточного положення гравця до цілі.

1. Порівняння з BFS і DFS: Алгоритм пошуку в ширину (BFS) гарантує найкоротший шлях, але він досліджує всі вузли одного рівня, що призводить до високих витрат пам'яті [12]. Пошук у глибину (DFS), навпаки, економічний у використанні пам’яті, але може витрачати час на дослідження нерелевантних шляхів [13]. A* поєднує переваги цих підходів, використовуючи евристику для зосередження на перспективних маршрутах.
2. Порівняння з алгоритмом Дейкстри: Дейкстра також забезпечує оптимальний шлях, але він не використовує евристику, через що аналізує більше вузлів [14]. A* є швидшим, оскільки може передбачити напрямок до цілі.

У грі Quoridor A* легко адаптується до динамічних змін поля, що робить його ключовим компонентом ШІ для забезпечення адекватної реакції на зміни, викликані встановленням парканів.

Приклад роботи алгоритму A* представлено на рисунку 2.2.

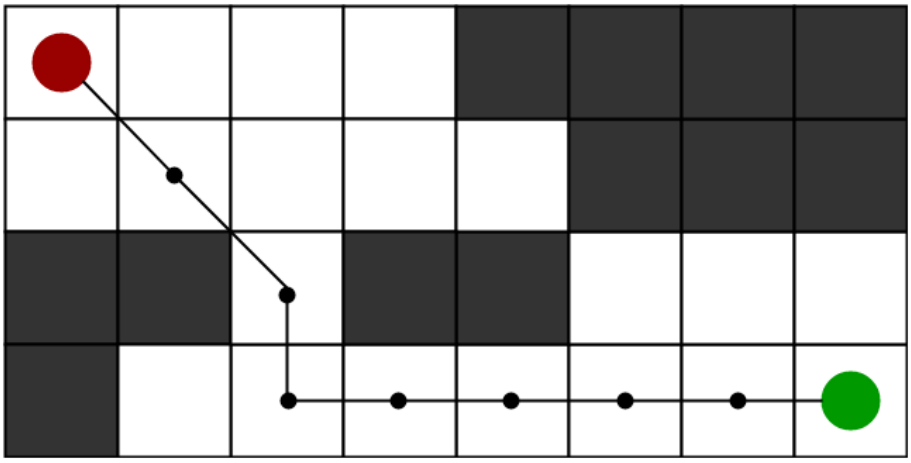

Рисунок 2.2 – Приклад роботи алгоритму A*

Алгоритм потребує адаптації до особливостей гри Quoridor, зокрема врахування неможливості діагонального переміщення фішок та специфічної мети гри, яка полягає не у досягненні конкретної точки, а у перетині фінального ряду.

Алгоритм мінімакс дозволяє аналізувати гру з точки зору обох гравців, забезпечуючи оптимальний вибір дій, які мінімізують виграш супротивника і максимізують виграш ШІ. У Quoridor це дозволяє ШІ не лише шукати шлях до цілі, але й ставити паркани так, щоб ускладнити досягнення мети супротивником.

Альфа-бета відсікання це оптимізація мінімаксу, яка знижує кількість обчислень, відсікаючи гілки дерева, які не можуть вплинути на кінцевий результат. Це дозволяє значно збільшити глибину пошуку без зростання витрат ресурсів.

Порівняння з іншими підходами:

1. Нейронні мережі: Нейромережі потребують великого датасету для навчання і мають високу обчислювальну складність [15]. Для Quoridor, де всі можливі стани гри можна прорахувати, мінімакс є ефективнішим.
2. Еволюційні алгоритми: Ефективні у задачах оптимізації, але повільні в умовах реального часу через велику кількість ітерацій [16].

3. Монтекарло (MCTS): Добре працює в умовах високої невизначеності, наприклад у го чи шахах [17]. Однак у Quoridor, де стан гри повністю відомий, мінімакс із відсіканням працює швидше.

Мінімакс із альфа-бета відсіканням дозволяє ШІ приймати ефективні стратегічні рішення навіть у складних умовах, таких як блокування супротивника чи визначення оптимального моменту для розміщення паркану.

Комбінація A* та мінімаксу забезпечує високоефективну роботу ШІ у Quoridor. A* відповідає за оперативний аналіз доступних маршрутів і пошук найкоротшого шляху, враховуючи динамічність поля. Мінімакс дозволяє визначати довгострокові стратегії, враховуючи як свої можливості, так і дії суперника.

Така архітектура ШІ дозволяє створити розумного і адаптивного опонента, який реагує на дії гравця, одночасно створюючи власні складнощі на полі. Цей підхід не лише покращує геймплей, але й підвищує цікавість ігрового процесу.

2.3. Реалізація ШІ

Реалізація штучного інтелекту (ШІ) для гри Quoridor базується на двох основних алгоритмах: алгоритмі A* для пошуку найкоротших шляхів і алгоритмі міні-макс із альфа-бета відсіканням для прийняття стратегічних рішень. Для оцінки стану гри використовується функція оцінки, яка враховує дві ключові метрики. Глибина міні-максу встановлена на рівні 2, що забезпечує баланс між продуктивністю та якістю аналізу гри.

Алгоритм A* працює з графом, де вузли представляють точки на ігровому полі, а ребра — можливі переміщення між ними. Основна ідея полягає у використанні комбінованої оцінки, яка враховує як відстань до поточного вузла, так і прогнозовану вартість до цільового.

Для кожного вузла n алгоритм розраховує значення:

$$f(n) = g(n) + h(n)$$

де:

- $g(n)$ — фактична вартість шляху від початкового вузла до поточного вузла nnn.
- $h(n)$ — евристична оцінка відстані від вузла nnn до цільового вузла.
- $f(n)$ — сумарна оцінка, яка визначає пріоритет обробки вузла.

Алгоритм дій

1. Ініціалізація:

   Початковий вузол додається до відкритого списку (Open Set).

   Вартість g для початкового вузла дорівнює 0, а його f — дорівнює h.

2. Вибір вузла:

   Вибирається вузол із найменшим значенням f із відкритого списку.

3. Оцінка вузла:

   Якщо вузол є цільовим, алгоритм завершується, і відновлюється шлях.

   Інакше вузол переноситься до закритого списку (Closed Set).

4. Обробка сусідів:

   Для кожного сусіднього вузла, який ще не в закритому списку:

   Розраховується нове значення g.

Якщо вузол не в відкритому списку або нове g менше попереднього, вузол додається або оновлюється у відкритому списку. Значення f оновлюється як $g + h$.

5. Повтор:

Процес триває, доки відкритий список не стане порожнім або не буде знайдений цільовий вузол.

Міні-макс із альфа-бета відсіканням використовується для моделювання ходів гравців і аналізу стратегічних рішень. Глибина алгоритму обмежується двома рівнями, що дає можливість оцінювати безпосередні наслідки ходів без значного навантаження на обчислювальні ресурси.

Етапи роботи алгоритму:

1. Генерація всіх можливих ходів для поточного гравця (переміщення фішки або встановлення паркану).
2. Оцінка кожного ходу за допомогою функції оцінки.
3. Застосування альфа-бета відсікання для виключення явно невигідних гілок дерева пошуку, що значно зменшує кількість обчислень.

Функція оцінки розроблена для оцінки переваги одного гравця над іншим. Вона базується на двох основних параметрах:

1. Різниця довжин найкоротших шляхів гравців: визначається наступним чином: $f_1 = S_{min} - S_{max}$, де S_{min} – довжина найкоротшого шляху для гравця, що мінімізує, S_{max} – довжина найкоротшого шляху для гравця, що максимізує.
2. Різниця кількості доступних парканів. Цей параметр визначається за наступною формулою: $f_2 = Q_{max} - Q_{min}$, де Q_{max} – кількість парканів для гравця, що максимізує, Q_{min} – кількість парканів для гравця, що мінімізує.

Формула функції оцінки: $E = w_1 * f_1 + w_2 * f_2$, де w_1 і w_2 — вагові коефіцієнти, які змінюються для адаптації ШІ до стилю гри суперника.

Адаптивність алгоритму забезпечується динамічною зміною вагових коефіцієнтів w_1 і w_2. ШІ аналізує дії суперника протягом гри, наприклад,

частоту використання парканів чи прагнення до агресивної гри, і змінює пріоритетність параметрів функції оцінки.

Якщо суперник часто блокує шляхи, вагу w_1 можна збільшити, щоб приділяти більше уваги різниці довжин шляхів. Якщо ж суперник економить паркани, вагу w_2 можна зменшити, знижуючи важливість цього параметра.

У кожному ході ШІ виконує наступне:

1. За допомогою A* визначає найкоротші шляхи для обох гравців.
2. Генерує всі можливі ходи та оцінює їх за функцією оцінки.
3. Використовує міні-макс із альфа-бета відсіканням, щоб обрати найкращий хід.

Ця реалізація забезпечує ефективність та адаптивність ШІ, дозволяючи йому не лише реагувати на дії суперника, але й змінювати стратегію під час гри.

ВИСНОВКИ ДО РОЗДІЛУ 2

У другому розділі дослідження було розглянуто процес розробки та реалізації адаптивного штучного інтелекту для гри Quoridor. Вибір алгоритмів для ШІ був визначений з урахуванням особливостей гри та вимог до ефективності. Для пошуку шляху був обраний алгоритм A*, який забезпечує оптимальний пошук за допомогою евристичних функцій, адаптуючись до зміни ситуації на полі. Цей алгоритм дозволяє ШІ швидко і точно визначати найкращий шлях до фінішу, враховуючи можливі перешкоди у вигляді парканів.

Для прийняття рішень був використаний адаптований алгоритм мінімакс з альфа-бета відсіканням. Така комбінація дозволяє ШІ здійснювати глибокий аналіз можливих ходів при збереженні високої ефективності за рахунок скорочення неперспективних гілок дерева пошуку. Адаптивна функція оцінки дає можливість варіювати обсяг обчислень в залежності від складності ситуації, що робить ШІ більш гнучким і здатним адаптуватися до змін у грі.

У процесі реалізації була створена модель гри, яка дозволяє ефективно інтегрувати алгоритми пошуку шляху та прийняття рішень, забезпечуючи оптимальні стратегії для ШІ. Пошук шляху та оцінка варіантів дій поєднуються в єдину систему, що дозволяє ШІ не тільки рухатися до мети, але й ефективно блокувати дії супротивника.

Процес тестування та налаштування ШІ показав, що обрані алгоритми дозволяють досягти високої ефективності та гнучкості в прийнятті рішень. Завдяки таким підходам ШІ здатний адаптуватися до різних стилів гри, створюючи цікавий та складний виклик для гравців.

Завдяки реалізації адаптивного ШІ, гра Quoridor отримує новий рівень складності, що може бути застосовано не лише для цієї гри, але й для інших

настільних стратегічних ігор. У майбутньому можна очікувати подальше вдосконалення та оптимізацію алгоритмів для ще більш ефективного управління процесами пошуку та прийняття рішень.

РОЗДІЛ 3. ТЕСТУВАННЯ ТА АНАЛІЗ ЕФЕКТИВНОСТІ ШІ

3.1. Методологія тестування ШІ

Методологія тестування штучного інтелекту (ШІ) для гри Quoridor орієнтована на збір аналітичних даних у реальному часі під час гри, зокрема через використання Unity Analytics. Цей інструмент дозволяє ефективно збирати та аналізувати метрики, які дають чітке уявлення про ефективність ШІ, рівень його адаптивності та вплив на досвід користувачів [18]. У межах цього дослідження основними метриками для оцінки є відсоток перемог/поразок ШІ та кількість раундів, які гравці грають перед тим, як припинити гру.

Відсоток перемог і поразок є однією з ключових метрик, що дозволяє оцінити, наскільки добре працює ШІ в порівнянні з реальними гравцями [19]. Ідея полягає в тому, щоб забезпечити баланс між складністю гри та її задоволенням для користувача. Якщо ШІ виграє занадто часто, це може свідчити про надмірну складність, що призводить до розчарування користувачів, оскільки вони не можуть перемогти. З іншого боку, якщо ШІ програє занадто часто, це може знизити інтерес до гри через її надмірну легкість.

Тому важливо, щоб ШІ мав баланс у перемогах і поразках, який залежить від вибору стратегії і адаптується до стилю гри кожного гравця. Порівняння цих показників дозволяє коригувати складність ШІ, підлаштовуючи його так, щоб рівень гри був цікавим і викликав бажання повернутися до наступних матчів.

Іншим важливим аспектом є кількість раундів, які гравець проводить до припинення гри. Цей показник допомагає визначити, чи достатньо інтересу і виклику створює ШІ, щоб гравці продовжували грати на протязі тривалого часу. Якщо гравець припиняє гру після кількох раундів, це може свідчити про нецікавий або надто складний геймплей. З іншого боку, якщо гравець продовжує гру протягом багатьох раундів, це може вказувати на оптимальний рівень складності, коли інтерес до гри зберігається, а сама гра викликає бажання продовжувати змагатися.

Для кожного раунду збирається інформація про кількість ходів, що були зроблені до того, як гравець завершив гру. Це дозволяє не лише оцінити, скільки часу гравець провів у грі, але й дає чітке уявлення про динаміку її інтересу та складності на різних етапах.

Дані для тестування збираються під час кожної гри користувача. Після завершення кожного матчу автоматично фіксуються результати гри (перемога чи поразка) та кількість раундів, що пройшли до того, як гравець вирішив припинити гру. Ці дані відправляються в Unity Analytics, де вони обробляються і зберігаються для подальшого аналізу. Зібрані метрики дозволяють оцінити загальний баланс складності гри та її привабливість для гравців.

Аналіз метрик, отриманих за допомогою Unity Analytics, дозволяє оцінити кілька важливих аспектів. По-перше, за допомогою відсотка перемог можна зрозуміти, чи є ШІ достатньо сильним для забезпечення адекватного виклику для гравця, але не настільки сильним, щоб він став непробивним і призвів до втрати інтересу. Окрім цього, важливим є розуміння того, скільки часу гравець проводить у грі і чи не припиняє гру надто швидко через занадто високу або низьку складність.

Використовуючи ці дані, можна провести коригування алгоритмів ШІ, щоб створити більш адаптивного суперника, який може змінювати стратегії в

залежності від дій гравця, адаптуючи рівень складності і забезпечуючи цікавий і захоплюючий ігровий процес.

Метод тестування з використанням Unity Analytics дозволяє не лише збирати точні дані, але й отримувати на основі цих даних інформацію для подальшого вдосконалення ШІ. Порівнюючи різні рівні складності, які можна налаштувати в ШІ, тестувальники можуть виявити найбільш оптимальні стратегії для кожного рівня і адаптувати систему таким чином, щоб гравці мали цікаву і конкурентну гру в різних умовах. Цей підхід допомагає забезпечити ефективність ШІ в реальних умовах, а також забезпечити високу задоволеність гравців від гри.

3.2. Аналіз результатів роботи ШІ

Після проведення серії тестів з використанням адаптивного ШІ в грі Quoridor було отримано позитивні результати, що підтверджують ефективність реалізованої системи. Основними метриками для оцінки роботи ШІ були відсоток перемог/поразок, а також кількість раундів, що гравці проходять перед тим, як припиняють грати. Виходячи з цих показників, можна зробити кілька важливих висновків про роботу ШІ. Результати роботи ШІ представлені в таблиці 3.1.

Таблиця 3.1 – Результати роботи ШІ

Параметр	Значення
Середній відсоток перемог/поразок	46.5%
Середня кількість раундів	4

1. Високий рівень адаптивності

Одним з основних критеріїв оцінки було здатність ШІ адаптувати свою стратегію до поведінки гравців, змінюючи складність на основі поточних результатів. Результати тестів показали, що ШІ ефективно адаптується до різних стилів гри, що дозволяє підтримувати належний рівень виклику для гравця. Зокрема, середній відсоток перемог ШІ варіювався в залежності від того, скільки раундів гравець продовжував гру, що підтверджує правильність налаштувань адаптивної функції оцінювання мінімакс-алгоритму. Гравці, які проявляли стратегії з глибшим плануванням, зустрічали складніші ситуації, а новачки – отримували спрощені варіанти, що підтримувало їх інтерес до гри.

2. Складність гри підтримується на оптимальному рівні

Рівень складності показав себе на високому рівні. Залежно від стилю гри, ШІ змінював свою стратегію так, щоб забезпечити як новачкам, так і досвідченим гравцям оптимальний виклик. Результати тестів показали, що кількість раундів, яку проходили гравці до того, як припиняли гру, залишалась на помірному рівні. Більшість гравців не відчували себе перевантаженими надмірно складними задачами, і водночас не було випадків, коли гра ставала надто легкою.

3. Підвищення інтересу до гри

Аналізуючи середній час гри, можна зазначити, що адаптивний ШІ позитивно вплинув на загальний інтерес користувачів. Гравці, що зустрічали ШІ різних рівнів складності, мали більше шансів на тривалішу взаємодію з грою, оскільки система забезпечувала досить інтелектуальний виклик без зниження складності до рівня, що обмежує розвиток інтересу. Стратегії ШІ, що змінювались в залежності від дій користувача, значно покращили геймплей і дозволили зберегти захоплення навіть для новачків.

4. Продуктивність пошуку шляху

Алгоритм A* для пошуку шляху виявився ефективним у контексті реального часу, забезпечуючи необхідний баланс між швидкістю та точністю пошуку. Пошук шляху відбувався без значних затримок, що забезпечувало безперервний ігровий процес. Деякі гравці зазначили, що ШІ демонструє інтуїтивно зрозумілі та швидкі рішення при плануванні своїх ходів, що дозволяє зберігати швидкість і динаміку гри.

5. Загальний аналіз

Загалом, результати тестування показали, що адаптивний ШІ відповідає вимогам, що ставляться до нього в контексті гри Quoridor. Показники перемог/поразок та кількості раундів свідчать про те, що система ефективно адаптується до рівня гравця, підтримує оптимальну складність гри і не призводить до фрустрації чи нудьги у процесі гри. ШІ демонструє хорошу продуктивність, а також здатність підтримувати інтерес гравців до довгих сесій гри.

Ці результати підтверджують правильність вибору алгоритмів для реалізації ШІ та показують, що він успішно виконує завдання, поставлені в рамках дослідження. Враховуючи ці позитивні результати, можна продовжувати вдосконалення системи та додавання нових можливостей для ще більшої адаптації ШІ до індивідуальних стратегій користувачів.

3.3. Аналіз недоліків та перспектив вдосконалення

Попри позитивні результати роботи адаптивного ШІ, у процесі тестування було виявлено кілька недоліків, які можуть вплинути на подальшу ефективність системи. Однак, кожен з них є точкою для покращення та вдосконалення ШІ, що дозволить досягти ще більш високих результатів в майбутньому.

1. Обмежена глибина міні-максу та її вплив на якість рішень

Одним із ключових недоліків поточного ШІ є обмежена глибина алгоритму міні-максу, яка встановлена на рівні двох пліїв. Це обмеження було введене через обчислювальні обмеження, пов’язані з великою складністю гри Quoridor. Хоча така глибина дозволяє забезпечити достатньо швидке прийняття рішень, вона водночас робить ШІ менш ефективним у стратегічному плануванні, особливо в складних ситуаціях, які потребують врахування довгострокових наслідків.

У деяких випадках ШІ може обирати ходи, які є короткозорими або непродуманими. Наприклад, замість того щоб блокувати шлях супротивника стратегічно розташованим парканом, ШІ може вибрати менш ефективний хід, що несе користь лише в короткостроковій перспективі. Це особливо помітно в іграх з досвідченими гравцями, які здатні прорахувати ходи на значно більшу глибину.

Перспективи вдосконалення:

- Оптимізація алгоритмів.
- Використання специфічних для гри Quoridor оптимізацій (відсікання далеких від гравців парканчиків, як заздалегідь поганих ходів, тощо) [20].
- Покращення функції оцінки. Удосконалення функції оцінки, включаючи нові параметри, що краще відображають стратегічну важливість позицій, дозволить ШІ приймати більш збалансовані рішення навіть при обмеженій глибині дерева.

2. Продуктивність в умовах великих ігрових сесій

Хоча ШІ демонструє хорошу продуктивність при стандартних ігрових сесіях, у випадку більш тривалих або повторюваних ігор, продуктивність

алгоритмів пошуку шляху та прийняття рішень може знижуватися. Це стає помітним, коли система стикається з великою кількістю можливих ходів та варіантів для пошуку шляху.

Перспективи вдосконалення: Можна застосувати методи кешування чи зберігання часткових результатів, щоб прискорити пошук шляхів та зменшити обчислювальне навантаження [21]. Наприклад, використання трансферу навченої моделі ШІ, яка була попередньо налаштована на подібні стратегії, дозволить зменшити час, необхідний для обробки кожного ходу.

Попри успішну реалізацію адаптивного ШІ для гри Quoridor, існує ряд напрямків для покращення, які можуть зробити систему ще більш ефективною та інтерактивною. Вдосконалення алгоритмів пошуку, адаптивності до різних гравців, а також реалізація багатокористувацького режиму сприятиме значному покращенню якості гри та взаємодії з користувачем. Розвиток цих напрямків відкриває нові можливості для покращення ШІ та підвищення рівня складності гри, що зробить досвід гравців ще більш захоплюючим та інтуїтивно зрозумілим.

ВИСНОВКИ ДО РОЗДІЛУ 3

У третьому розділі було проведено тестування адаптивного штучного інтелекту (ШІ) для гри Quoridor, а також проаналізовано його ефективність і недоліки. Збір метрик за допомогою Unity Analytics дозволив отримати важливі дані, зокрема про відсоток перемог та поразок, а також про кількість раундів, які грають учасники перед тим, як припиняють гру. Отримані результати показали, що ШІ ефективно адаптується до різних рівнів складності, забезпечуючи гравців належним рівнем виклику.

Проведений аналіз результатів підтвердив, що адаптивний ШІ відповідає вимогам до створення інтелектуальних супротивників у грі, здатних змінювати свою стратегію в залежності від дій гравця. Однак було також виявлено кілька недоліків, таких як обмежена глибина пошуку на високих рівнях складності, що може призводити до затримок у прийнятті рішень, а також необхідність подальшої адаптації ШІ до різних стилів гри.

Аналіз недоліків і перспектив вдосконалення ШІ показав, що впровадження додаткових методів оптимізації, таких як кешування результатів пошуку та використання методів машинного навчання, може значно покращити швидкість роботи та адаптивність системи. Окрім того, додаткові методи прогнозування поведінки гравців і реалізація багатокористувацького режиму відкривають нові можливості для покращення ігрового досвіду.

Загалом, результати тестування показали, що розроблений ШІ успішно справляється з поставленими завданнями, але має потенціал для подальшого вдосконалення. Це дозволить забезпечити ще вищу якість гри та інтерактивність, підвищуючи рівень задоволення користувачів від гри.

ВИСНОВКИ

Дослідження методів створення адаптивного штучного інтелекту (ШІ) для покрокових ігор показало важливість інтеграції класичних підходів з сучасними технологіями для досягнення високої ефективності та адаптивності ШІ. У процесі роботи над проектом було проведено глибокий аналіз існуючих методів, таких як алгоритм мінімакс, альфа-бета відсікання, а також нестандартних підходів, зокрема адаптивної функції оцінювання.

Вибір мінімаксу з альфа-бета відсіканням виявився ефективним для гри Quoridor. Цей алгоритм дозволяє ШІ знаходити оптимальні ходи в межах обмежених обчислювальних ресурсів, забезпечуючи баланс між складністю і швидкістю прийняття рішень. Додатково, використання алгоритму A* для пошуку шляху значно покращило здатність ШІ швидко знаходити найкоротші маршрути до мети, що є критичним для досягнення успіху в грі.

Реалізація ШІ для гри Quoridor показала задовільні результати: комп'ютерний суперник адекватно реагує на стратегії гравця, демонструючи різні рівні складності в залежності від адаптивних механізмів. Зібрані метрики, такі як відсоток перемог і кількість раундів, підтверджують, що ШІ досягає високої ефективності, знижуючи обчислювальні витрати без втрати якості гри.

Проте, незважаючи на успіхи, існують певні недоліки, які потребують подальшого вдосконалення. Наприклад, можна покращити адаптивність ШІ шляхом впровадження більш складних евристичних функцій та використання методів машинного навчання для створення ще більш гнучких стратегій. Крім того, можна покращити швидкість роботи алгоритму за допомогою специфічних для гри Quoridor оптимізацій.

У підсумку, проведене дослідження демонструє, що поєднання класичних алгоритмів із сучасними методами адаптації забезпечує високу ефективність ігрового ШІ. Це відкриває перспективи для використання цих методів не лише в грі Quoridor, але й у розробці інтелектуальних систем для інших покрокових ігор.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Board game artificial intelligence: the minimax algorithm [Електронний ресурс]. – Режим доступу: https://www.instructables.com/Othello-Artificial-Intelligence/.

2. Artificial intelligence | alpha-beta pruning - javatpoint [Електронний ресурс]. – Режим доступу: https://www.javatpoint.com/ai-alpha-beta-pruning.

3. Introduction to evaluation function of minimax algorithm in game theory - geeksforgeeks [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/introduction-to-evaluation-function-of-minimax-algorithm-in-game-theory/.

4. How can you design adaptive AI for video games? [Електронний ресурс]. – Режим доступу: https://www.linkedin.com/advice/0/how-can-you-design-adaptive-ai-video-games-skills-game-development.

5. Deep Blue (chess computer) - Wikipedia [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer).

6. 2asaIt. ШІ-системі AlphaGo 9 років. Як програма, що перемогла чемпіона світу з ГО, нині впливає на розвиток ШІ в різних сферах [Електронний ресурс]. – Режим доступу: https://dev.ua/ru/news/alphago-1728890522.

7. History of AI in games - video game - F.E.A.R. [Електронний ресурс]. – Режим доступу: https://modl.ai/fear/.

8. Booth M. The AI systems of left 4 dead [Електронний ресурс]. – Режим доступу: https://steamcdn-a.akamaihd.net/apps/valve/2009/ai_systems_of_l4d_mike_booth.pdf.

9. Robinson R. Quoridor rulebook [Електронний ресурс]. – Режим доступу: https://cdn.1j1ju.com/medias/fe/36/08-quoridor-rulebook.pdf.

10. A quoridor-playing agent [Електронний ресурс]. – Режим доступу: https://project.dke.maastrichtuniversity.nl/games/files/bsc/Mertens_BSc-paper.pdf.

11. A* search algorithm [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/a-search-algorithm/.

12. Breadth first search or BFS for a graph [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.

13. S R. A. What is DFS (depth-first search): types, complexity & more | simplilearn [Електронний ресурс]. – Режим доступу: https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm#:~:text=Depth-First%20Search%20or%20DFS,nearby%20nodes%20into%20a%20stack..

14. Dijkstra's algorithm - Wikipedia [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Dijkstra's_algorithm.

15. Schaul T., Schmidhuber J. A scalable neural network architecture for board games [Електронний ресурс]. – Режим доступу: https://people.idsia.ch/~juergen/cig2008go.pdf.

16. Öberg V. Evolutionary ai in board games [Електронний ресурс]. – Режим доступу: https://www.diva-portal.org/smash/get/diva2:823737/FULLTEXT01.pdf.

17. Monte Carlo tree search - Wikipedia [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search.

18. Learn game development w/ unity | courses & tutorials in game design, VR, AR, & real-time 3D | unity learn [Електронний ресурс]. – Режим доступу: https://learn.unity.com/tutorial/get-started-with-unity-analytics.

19. Solo board game preferences - survey results – pine island games [Електронний ресурс]. – Режим доступу: https://www.pineislandgames.com/blog/solo-player-preferences.

20. Wächter L. Improving Minimax performance [Електронний ресурс]. – Режим доступу: https://levelup.gitconnected.com/improving-minimax-performance-fc82bc337dfd.

21. FRANKLIN L. Optimization Areas of the Minimax Algorithm [Електронний ресурс]. – Режим доступу: https://kth.diva-portal.org/smash/get/diva2:1778372/FULLTEXT01.pdf.

22. Fogel D. B. Evolutionary computation: Toward a new philosophy of machine intelligence. 3rd ed. Hoboken, N.J: Wiley-Interscience, 2007. – 274 с.

23. Russell S. J., Norvig P. Artificial Intelligence: A Modern Approach, Pearson EText, Global Edition. Pearson Education, Limited, 2021.

24. Yannakakis G. N., Togelius J. Artificial Intelligence and Games [Електронний ресурс]. – Режим доступу: https://doi.org/10.1007/978-3-319-63519-4.

ДОДАТКИ

Додаток А

Лістинг коду А.1 Псевдокод алгоритму прийняття рішень мінімакс

```
function alpha_beta_minmax(state, depth, alpha, beta, maximizing_player):
    if state is terminal or depth is 0:
        return utility value of state

    if maximizing_player:
        set max_eval to negative infinity
        for each action available in the current state:
            evaluate the result of the action using:
                eval = alpha_beta_minmax(next state from action, depth - 1,
alpha, beta, minimizing player)
            update max_eval to the maximum of max_eval and eval
            update alpha to the maximum of alpha and eval
            if beta is less than or equal to alpha:
                break the loop  # Beta cut-off
        return max_eval
    else:
        set min_eval to positive infinity
        for each action available in the current state:
            evaluate the result of the action using:
                eval = alpha_beta_minmax(next state from action, depth - 1,
alpha, beta, maximizing player)
            update min_eval to the minimum of min_eval and eval
            update beta to the minimum of beta and eval
            if beta is less than or equal to alpha:
                break the loop  # Alpha cut-off
        return min_eval
```

Додаток Б

Лістинг коду Б.1 Код алгоритму пошуку доступних ходів

```
public struct Move
{
    public MoveType MoveType { get; set; }
    public Vector2Int Position { get; set; }
    public WallType WallType { get; set; }
}



public enum MoveType
{
    Move,
    Wall,
}



public struct Node
{
    public Vector2Int Position { get; set; }
    public Vector2Int Previous { get; set; }
    public int Cost { get; set; }

    public Node(Vector2Int position)
    {
        Position = position;
        Previous = new Vector2Int(-1, -1);
        Cost = int.MaxValue;
    }

    public bool HasPosition()
    {
        return Position != new Vector2Int(-1, -1);
    }

    public bool HasPrevious()
    {
        return Previous != new Vector2Int(-1, -1);
    }
}



public struct Cell
{
    public GameObject CellObject { get; set; }
    public Vector2Int Coordinates { get; set; }
    public PawnColor PawnColor { get; set; }
}
```

```
public enum PawnColor
{
    None = 0,
    White = 1,
    Black = 2,
}



public struct Wall
{
    public GameObject WallObject { get; set; }
    public Vector2Int Coordinates { get; set; }
    public WallType WallType { get; set; }
}



public enum WallType
{
    None = 0,
    Horizontal = 1,
    Vertical = 2,
}



public static class MovesProvider
{
    public static Move[] GetMoves(Cell[,] cells, Wall[,] walls,
Dictionary<PawnColor, int> playerWalls,
        Vector2Int position, Vector2Int opponentPosition, int goalRow, int
opponentGoalRow, PawnColor pawnColor,
        PawnColor opponentPawnColor)
    {
        var result = new List<Move>();

        var pawnMoves = PawnMovesProvider.GetPossiblePawnMoves(cells, walls,
position, opponentPawnColor);
        for (var i = 0; i < pawnMoves.Length; i++)
        {
            result.Add(pawnMoves[i]);
        }

        if (playerWalls[pawnColor] <= 0)
        {
            return result.ToArray();
        }

        var wallMoves = WallMovesProvider.GetPossibleWallMoves(cells, walls,
position, opponentPosition, goalRow,
            opponentGoalRow, pawnColor, opponentPawnColor);
        for (var i = 0; i < wallMoves.Length; i++)
        {
            result.Add(wallMoves[i]);
        }
```

```
        return result.ToArray();
    }
}



public static class PawnMovesProvider
{
    public static Move[] GetPossiblePawnMoves(Cell[,] cells, Wall[,] walls,
Vector2Int origin,
        PawnColor opponentPawnColor)
    {
        var possiblePawnMovePositions = GetPossiblePawnMovePositions(cells,
walls, origin, opponentPawnColor);

        var result = new Move[possiblePawnMovePositions.Length];
        for (var i = 0; i < possiblePawnMovePositions.Length; i++)
        {
            result[i] = new Move
            {
                MoveType = MoveType.Move,
                Position = possiblePawnMovePositions[i],
            };
        }
        return result;
    }

    public static Vector2Int[] GetPossiblePawnMovePositions(Cell[,] cells,
Wall[,] walls, Vector2Int origin,
        PawnColor opponentPawnColor)
    {
        var moves = new List<Vector2Int>();

        var rawMoves = RawPawnMovesProvider.Get(origin);
        for (var i = 0; i < rawMoves.Length; i++)
        {
            if (!CoordinatesValidator.Validate(cells, rawMoves[i]))
            {
                continue;
            }

            if (WallMovesProvider.WallBetweenCells(walls, origin,
rawMoves[i]))
            {
                continue;
            }

            if (cells[rawMoves[i].x, rawMoves[i].y].PawnColor !=
opponentPawnColor)
            {
                moves.Add(rawMoves[i]);
                continue;
            }

            var rawJumps = RawPawnMovesProvider.Get(rawMoves[i]);
```

```
            for (var j = 0; j < rawJumps.Length; j++)
            {
                if (!CoordinatesValidator.Validate(cells, rawJumps[j]))
                {
                    continue;
                }

                if (WallMovesProvider.WallBetweenCells(walls, rawMoves[i],
rawJumps[j]))
                {
                    continue;
                }

                if (cells[rawJumps[j].x, rawJumps[j].y].PawnColor !=
PawnColor.None)
                {
                    continue;
                }

                moves.Add(rawJumps[j]);
            }
        }

        return moves.ToArray();
    }
}

public static class WallMovesProvider
{
    public static Move[] GetPossibleWallMoves(Cell[,] cells, Wall[,] walls,
Vector2Int position,
        Vector2Int opponentPosition, int goalRow, int opponentGoalRow,
PawnColor pawnColor,
        PawnColor opponentPawnColor)
    {
        var possibleWalls = new List<Move>();

        for (var i = 0; i < walls.GetLength(0); i++)
        {
            for (var j = 0; j < walls.GetLength(1); j++)
            {
                if (walls[i, j].WallType != WallType.None)
                {
                    continue;
                }

                if (!WallOfTypeExists(walls, new Vector2Int(i - 1, j),
WallType.Horizontal)
                    && !WallOfTypeExists(walls, new Vector2Int(i + 1, j),
WallType.Horizontal))
                {
                    walls[i, j].WallType = WallType.Horizontal;
                    if (PathFinder.PathExists(cells, walls, position,
goalRow, opponentPawnColor)
                        && PathFinder.PathExists(cells, walls,
opponentPosition, opponentGoalRow, pawnColor))
```

```
                    {
                        possibleWalls.Add(new Move
                        {
                            MoveType = MoveType.Wall,
                            Position = new Vector2Int(i, j),
                            WallType = WallType.Horizontal,
                        });
                    }
                }

                if (!WallOfTypeExists(walls, new Vector2Int(i, j - 1),
WallType.Vertical)
                    && !WallOfTypeExists(walls, new Vector2Int(i, j + 1),
WallType.Vertical))
                {
                    walls[i, j].WallType = WallType.Vertical;
                    if (PathFinder.PathExists(cells, walls, position,
goalRow, opponentPawnColor)
                        && PathFinder.PathExists(cells, walls,
opponentPosition, opponentGoalRow, pawnColor))
                    {
                        possibleWalls.Add(new Move
                        {
                            MoveType = MoveType.Wall,
                            Position = new Vector2Int(i, j),
                            WallType = WallType.Vertical,
                        });
                    }
                }

                walls[i, j].WallType = WallType.None;
            }
        }

        return possibleWalls.ToArray();
    }

    private static bool WallOfTypeExists(Wall[,] walls, Vector2Int position,
WallType wallType)
    {
        if (!CoordinatesValidator.Validate(walls, position))
        {
            return false;
        }

        return walls[position.x, position.y].WallType == wallType;
    }

    public static bool WallBetweenCells(Wall[,] walls, Vector2Int origin,
Vector2Int goal)
    {
        if (origin.x < goal.x)
        {
            var wallCoordinates = new Vector2Int[]
            {
                new(origin.x, origin.y),
```

```
                new(origin.x, origin.y - 1),
            };

            if (WallsExist(walls, wallCoordinates, WallType.Vertical))
            {
                return true;
            }
        }
        else if (origin.x > goal.x)
        {
            var wallCoordinates = new Vector2Int[]
            {
                new(origin.x - 1, origin.y),
                new(origin.x - 1, origin.y - 1),
            };

            if (WallsExist(walls, wallCoordinates, WallType.Vertical))
            {
                return true;
            }
        }
        else if (origin.y < goal.y)
        {
            var wallCoordinates = new Vector2Int[]
            {
                new(origin.x, origin.y),
                new(origin.x - 1, origin.y),
            };

            if (WallsExist(walls, wallCoordinates, WallType.Horizontal))
            {
                return true;
            }
        }
        else if (origin.y > goal.y)
        {
            var wallCoordinates = new Vector2Int[]
            {
                new(origin.x, origin.y - 1),
                new(origin.x - 1, origin.y - 1),
            };

            if (WallsExist(walls, wallCoordinates, WallType.Horizontal))
            {
                return true;
            }
        }

        return false;
    }

    private static bool WallsExist(Wall[,] walls, Vector2Int[]
wallCoordinates, WallType wallType)
    {
        for (var i = 0; i < wallCoordinates.Length; i++)
        {
```

```
            if (!CoordinatesValidator.Validate(walls, wallCoordinates[i]))
            {
                continue;
            }

            if (walls[wallCoordinates[i].x, wallCoordinates[i].y].WallType ==
wallType)
            {
                return true;
            }
        }

        return false;
    }
}



public static class CoordinatesValidator
{
    public static bool Validate<T>(T[,] array, Vector2Int coordinates)
    {
        return coordinates.x >= 0
               && coordinates.x < array.GetLength(0)
               && coordinates.y >= 0
               && coordinates.y < array.GetLength(1);
    }
}



public static class RawPawnMovesProvider
{
    private static readonly Vector2Int[] RawOffsets =
    {
        new(1, 0),
        new(-1, 0),
        new(0, 1),
        new(0, -1),
    };

    public static Vector2Int[] Get(Vector2Int origin)
    {
        var result = new Vector2Int[RawOffsets.Length];

        for (var i = 0; i < RawOffsets.Length; i++)
        {
            result[i] = origin + RawOffsets[i];
        }

        return result;
    }
}
```

```
public static class PathFinder
{
    public static Vector2Int[] GetShortestPath(Cell[,] cells, Wall[,] walls,
Vector2Int from, int row,
        PawnColor opponentPawnColor)
    {
        var reachable = new Dictionary<Vector2Int, Node>()
        {
            { from, new Node(from) }
        };
        var explored = new Dictionary<Vector2Int, Node>();

        while (reachable.Count > 0)
        {
            var node = ChooseNode(reachable, row);

            if (!node.HasPosition())
            {
                break;
            }

            if (node.Position.y == row)
            {
                var path = new List<Vector2Int>();
                while (node.HasPosition() && node.HasPrevious())
                {
                    path.Add(node.Position);
                    node = explored[node.Previous];
                }

                path.Reverse();
                return path.ToArray();
            }

            reachable.Remove(node.Position);
            explored.Add(node.Position, node);

            var neighbours =
PawnMovesProvider.GetPossiblePawnMovePositions(cells, walls, node.Position,
opponentPawnColor);
            for (var i = 0; i < neighbours.Length; i++)
            {
                if (reachable.ContainsKey(neighbours[i]) ||
explored.ContainsKey(neighbours[i]))
                {
                    continue;
                }

                var neighbourNode = new Node(neighbours[i]);

                if (node.Cost + 1 < neighbourNode.Cost)
                {
```

```
                    neighbourNode.Previous = node.Position;
                    neighbourNode.Cost = node.Cost + 1;
                }

                reachable.Add(neighbours[i], neighbourNode);
            }
        }

        return new Vector2Int[] { };
    }

    public static bool PathExists(Cell[,] cells, Wall[,] walls, Vector2Int
from, int goalRow,
        PawnColor opponentPawnColor)
    {
        return GetShortestPath(cells, walls, from, goalRow,
opponentPawnColor).Length > 0;
    }

    public static int GetShortestPathLength(Cell[,] cells, Wall[,] walls,
Vector2Int from, int goalRow,
        PawnColor opponentPawnColor)
    {
        return GetShortestPath(cells, walls, from, goalRow,
opponentPawnColor).Length;
    }

    private static Node ChooseNode(Dictionary<Vector2Int, Node> reachable,
int goalRow)
    {
        var minCost = int.MaxValue;
        Node bestNode = new Node(new Vector2Int(-1, -1));

        foreach (var (position, node) in reachable)
        {
            var costStartToNode = node.Cost;
            var costNodeToGoal = Math.Abs(goalRow - position.y);
            var totalCost = costStartToNode + costNodeToGoal;

            if (minCost > totalCost)
            {
                minCost = totalCost;
                bestNode = node;
            }
        }

        return bestNode;
    }
}
```

Лістинг коду В.1 Псевдокод алгоритму пошуку найкоротшого шляху A*

```
function find_path(start_node, end_node):
    initialize reachable as a list containing start_node
    initialize explored as an empty list

    while reachable is not empty:
        node = choose_node(reachable)

        if node is the end_node:
            return build_path(end_node)

        remove node from reachable
        add node to explored

        new_reachable = adjacent nodes of node that are not in explored
        for each adjacent in new_reachable:
            if adjacent is not in reachable:
                add adjacent to reachable

            if node's cost + 1 is less than adjacent's cost:
                set adjacent's previous node to node
                update adjacent's cost to node's cost + 1

    return no path found

function build_path(target_node):
    initialize path as an empty list
    while target_node is not null:
        add target_node to path
        move to target_node's previous node
    return path in reverse order

function choose_node(reachable):
    set min_cost to infinity
    set best_node to null

    for each node in reachable:
        calculate cost from start to node as node.cost
        estimate cost from node to goal as estimate_distance(node, end_node)
        calculate total_cost as sum of the above two costs

        if total_cost is less than min_cost:
            update min_cost to total_cost
            update best_node to node

    return best_node
```