МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів прийняття рішень у реальному часі широкого спектру в комп'ютерних іграх жанру RPG
(тема)

Виконав:
Здобувач 2 року навчання
групи ІПЗм-23-4
Дмитро КІСЛОВ
(власне ім’я, прізвище)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова
Освітня програма Інженерія програмного забезпечення
(повна назва освітньої програми)
Керівник доц. Олексій НАЗАРОВ
(посада, власне ім’я, прізвище)

Допускається до захисту
Зав. кафедри ___________ Кирило СМЕЛЯКОВ
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва)
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедри ________
(підпис)
«___»________ 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Кіслову Дмитру Романовичу
(прізвище, ім'я, по батькові)

1. Тема роботи

   Дослідження методів прийняття рішень у реальному часі широкого спектру в комп'ютерних іграх жанру RPG .

Затверджена наказом по університету від 15.04.2025 р №290 Ст

2. Термін подання студентом роботи до екзаменаційної комісії _16.06.2025_
3. Вихідні дані до роботи:

У дослідницькій роботі передбачити: аналіз наявних алгоритмів прийняття рішень на ринку, дослідження ефективності, розробка оптимального сценарію розробки алгоритму прийняття рішень у реальному часі в іграх жанру RPG

4. Перелік питань, що потрібно опрацювати в роботі:

аналіз предметної ґалузі, огляд та аналіз наукових джерел, постановка задачі, теоретичне дослідження, висновки, перелік посилань, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	16.04.2025	*виконано*
2	Аналіз предметної галузі і постановка задачі	18.04.2025 – 22.04.2025	*виконано*
3	Огляд й аналіз наявних рішень на ринку	23.04.2025 – 28.04.2025	*виконано*
4	Теоретичне дослідження	29.04.2025 – 08.05.2025	*виконано*
5	Розробка програмної системи	09.05.2025 – 20.05.2025	*виконано*
6	Аналіз отриманих результатів і формування висновків	21.05.2025 – 23.05.2025	*виконано*
7	Підготовка пояснювальної записки	24.05.2025 – 02.06.2025	*виконано*
8	Підготовка презентації та доповіді	03.06.2025 – 04.06.2025	*виконано*
9	Перевірка на плагіат	04.06.2025	*виконано*
10	Перевірка на нормоконтроль	04.06.2025	*виконано*
11	Рецензування	16.06.2025	*виконано*
12	Попередній захист	16.06.2025	*виконано*
13	Занесення диплома до електронного архіву	17.06.2025	*виконано*
14	Допуск до захисту у зав. кафедри	17.06.2025	*виконано*

Дата видачі завдання 16 квітня 2025р.

Здобувач ____________________ Дмитро КІСЛОВ
(підпис)

Керівник роботи ________________ доц. Олексій НАЗАРОВ
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 68 с., 34 рис., 8 формул, 21 джерело, 5 додатків.

АДАПТИВНЕ ПЛАНУВАННЯ, АЛГОРИТМИ ПРИЙНЯТТЯ РІШЕНЬ, ДИНАМІЧНЕ БАЛАНСУВАННЯ, ОПТИМІЗАЦІЯ РЕСУРСІВ, ПРЕДИКТИВНІ ОБЧИСЛЕННЯ, ШТУЧНИЙ ІНТЕЛЕКТ, MONTE CARLO TREE SEARCH, RPG-ІГРИ.

У роботі досліджено проблему розробки та оптимізації методів прийняття рішень у реальному часі для комп'ютерних ігор жанру RPG. Актуальність дослідження зумовлена зростаючими вимогами до якості штучного інтелекту в сучасних відеоіграх та необхідністю розробки ефективних методів прийняття рішень, здатних адаптуватися до динамічних умов ігрового середовища.

Об'єктом дослідження є процеси прийняття рішень у комп'ютерних іграх жанру RPG. Предметом дослідження виступають методи та алгоритми оптимізації прийняття рішень у реальному часі з урахуванням обмежень обчислювальних ресурсів та вимог до швидкодії системи.

Метою роботи є проведення дослідження ефективності методів оптимізації прийняття рішень штучного інтелекту в RPG-іграх.

У роботі проведено комплексний аналіз існуючих рішень та методів, що використовуються в сучасній ігровій індустрії. На основі проведеного аналізу запропоновано новий гібридний метод, що поєднує модифікований алгоритм Monte Carlo Tree Search з механізмами адаптивного планування та предиктивних обчислень. Розроблено математичну модель системи прийняття рішень, що враховує множину параметрів ігрового середовища та обмеження реального часу.

Результати дослідження впроваджено при розробці експериментального прототипу системи прийняття рішень, що підтвердило їх практичну значущість та ефективність. Подальші дослідження можуть бути спрямовані на розширення

функціональності системи та адаптацію розроблених методів для інших жанрів комп'ютерних ігор.

ADAPTIVE PLANNING, DECISION-MAKING ALGORITHMS, DYNAMIC BALANCING, RESOURCE OPTIMIZATION, PREDICTIVE CALCULATIONS, ARTIFICIAL INTELLIGENCE, MONTE CARLO TREE SEARCH, RPG-GAMESi.

The research investigates the problem of developing and optimizing real-time decision-making methods for RPG (Role-Playing Game) genre computer games. The relevance of the research is determined by the growing requirements for artificial intelligence quality in modern video games and the necessity to develop effective decision-making methods capable of adapting to dynamic gaming environment conditions.

The object of research encompasses decision-making processes in RPG genre computer games. The subject of research comprises methods and algorithms for optimizing real-time decision-making, taking into account computational resource constraints and system performance requirements.

The aim of the research is to conduct a study on the effectiveness of artificial intelligence decision-making optimization methods in RPG games.

The work presents a comprehensive analysis of existing solutions and methods utilized in the modern gaming industry. Based on the conducted analysis, a novel hybrid method is proposed, combining a modified Monte Carlo Tree Search algorithm with adaptive planning mechanisms and predictive computing. A mathematical model of the decision-making system has been developed, accounting for multiple gaming environment parameters and real-time constraints.

The research results have been implemented in the development of an experimental decision-making system prototype, confirming their practical significance and effectiveness. Future research may be directed towards expanding system functionality and adapting the developed methods for other computer game genres.

Завідувачу кафедри ПІ

проф. Кирилу СМЕЛЯКОВУ

ЗАЯВА

щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE.

Я, Кіслов Дмитро Романович, здобувач вищої освіти на другому (магістерському) рівні вищої освіти академічної групи ІПЗм-23-4 кафедри програмної інженерії, заявляю: моя кваліфікаційна робота на тему «Дослідження методів прийняття рішень у реальному часі широкого спектру в комп'ютерних іграх жанру RPG», що буде представлена для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і може бути опублікована в репозиторії «ElArKhNURE». Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ «ElArKhNURE». Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску роботи до захисту та застосування дисциплінарних заходів.

28.05.2025 Дмитро КІСЛОВ

ЗМІСТ

Вступ ... 9
1 Аналіз предметної галузі і постановка задачі ... 10
1.1 Бізнес-потреби та пріоритетність ... 100
1.2 Висвітлення наявних рішень на ринку ... 12
2 Огляд й аналіз літературних, наукових джерел ... 16
2.1 Аналіз глобальних наукових джерел ... 16
2.2 Аналіз наукових джерел університету ... 16
3 Постановка задачі ... 17
3.1 Формалізація проблеми ... 17
3.2 Функціональні вимоги ... 17
3.3 Нефункціональні вимоги ... 18
3.4 Методологія дослідження ... 20
3.5 Очікувані результати ... 21
4 Теоретичне дослідження ... 23
4.1 Обмеження та поняття ... 23
4.2 Опис та аналіз існуючих рішень ... 23
4.3 Проектування оптимального алгоритму прийняття рішень ... 25
5 Розробка програмної системи ... 29
5.1 Стек технологій ... 29
5.2 Проектування алгоритму генерації середовища ... 29
5.3 Проектування алгоритму генерації системи прийняття рішень ... 31
5.4 Проектування алгоритму емуляції ігрового циклу ... 32
6 Аналіз результатів роботи програмної системи ... 36
6.1 Генерація даних ... 36
6.2 Обробка даних програмною системою ... 38
6.3 Аналіз отриманих результатів ... 39
Висновки ... 42
Перелік джерел посилання ... 43

Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії 45
Додаток А Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ 46
Додаток Б Слайди презентації 48
Додаток В Апробація результатів роботи (тези доповіді) 54
Додаток Г Код програми 54
Додаток Д Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008: 2015 68

ВСТУП

У контексті стрімкого розвитку інформаційних технологій та індустрії комп'ютерних ігор актуальним є дослідження та розробка ефективних методів прийняття рішень у реальному часі, що застосовуються у комп'ютерних іграх жанру RPG (Role-Playing Game). Сучасні RPG-системи характеризуються високою складністю та потребують оптимізованих алгоритмічних рішень для забезпечення якісної взаємодії між компонентами системи.

Жанр RPG представляє особливий інтерес для дослідження методів прийняття рішень через свою багатокомпонентну структуру, яка включає управління штучним інтелектом неігрових персонажів (NPC), балансування ігрових механік, адаптивне налаштування складності, системи діалогів та квестів, а також динамічне формування контенту. Кожен з цих компонентів вимагає швидкого та ефективного прийняття рішень для забезпечення плавності ігрового процесу та високого рівня користувацького досвіду.

Сучасні RPG-ігри функціонують в умовах жорстких часових обмежень, де затримки в прийнятті рішень можуть критично вплинути на якість ігрового процесу. Системи штучного інтелекту повинні обробляти великі обсяги даних про стан ігрового світу, аналізувати поведінку гравців та приймати рішення протягом кількох мілісекунд. Це створює необхідність у розробці спеціалізованих алгоритмів, які поєднують високу швидкість обчислень з якістю прийнятих рішень.

Особливості RPG-систем включають необхідність врахування множинних взаємозалежних факторів: поточний стан персонажів, інвентар, прогрес у грі, взаємовідносини між персонажами, стан ігрового світу та історію попередніх дій гравця. Традиційні методи прийняття рішень часто виявляються недостатніми для ефективного управління такою складністю в реальному часі. Результати дослідження матимуть практичне застосування в розробці ігрових систем та сприятимуть подальшому розвитку методів штучного інтелекту в подібних інтерактивних додатках.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ І ПОСТАНОВКА ЗАДАЧІ

1.1 Бізнес-потреби та пріоритетність

У сучасному контексті розробки програмного забезпечення для інтерактивних розважальних систем особливої актуальності набуває проблематика оптимізації методів прийняття рішень у реальному часі. Комп'ютерні ігри жанру RPG (Role-Playing Game) являють собою складні програмні комплекси, що характеризуються множинністю взаємодій між компонентами системи та потребують впровадження ефективних алгоритмічних рішень.

Актуальність дослідження зумовлена наступними чинниками: зростання обчислювальної складності ігрових систем через збільшення кількості паралельних процесів та взаємодій між ігровими об'єктами, підвищення вимог до реалістичності поведінки штучного інтелекту та якості імітації прийняття рішень неігровими персонажами, необхідність оптимізації використання обчислювальних ресурсів при збереженні високої якості ігрового процесу, потреба у розробці масштабованих рішень для обробки значної кількості ігрових сутностей у реальному часі.

Аналіз ринку комп'ютерних ігор демонструє стійку тенденцію до зростання попиту на проєкти жанру RPG з розвиненими системами штучного інтелекту. Ключові бізнес-потреби включають:

а) технічні вимоги:
   1) оптимізація використання процесорного часу;
   2) мінімізація затримок при прийнятті рішень;
   3) ефективне використання оперативної пам'яті;
   4) підтримка багатопотоковості;

б) функціональні вимоги:
   1) реалістичність поведінки NPC;
   2) адаптивність до дій гравця;
   3) підтримка складних сценаріїв взаємодії;
   4) масштабованість системи;

в) якісні характеристики:
   1) надійність роботи алгоритмів;

2) передбачуваність поведінки системи;
3) можливість налаштування параметрів;
4) зручність інтеграції.

Першочерговою потребою є оптимізація використання процесорного часу при реалізації алгоритмів прийняття рішень. Сучасні RPG-системи характеризуються значною кількістю одночасних обчислювальних процесів, що вимагає ефективного розподілу ресурсів центрального процесора [7]. Згідно з дослідженнями, оптимальне використання процесорного часу не повинно перевищувати 25% від загального доступного ресурсу для забезпечення стабільної роботи інших компонентів системи (див. рис. 1.1).

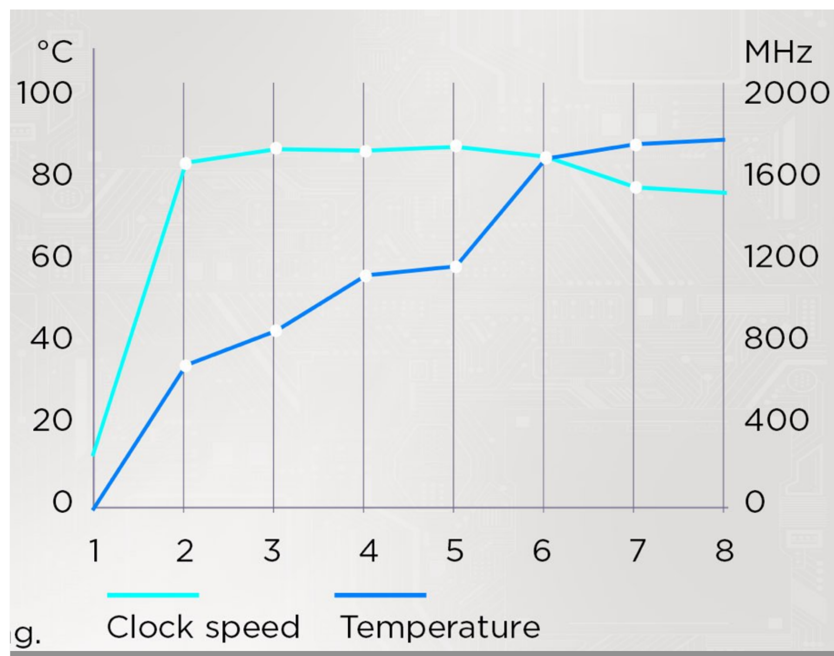

Рисунок 1.1 – Діаграма ефективності процесорів Intel та AMD в залежності від завантаження (за даними [8])

Важливим аспектом є мінімізація затримок при обробці ігрових подій. Для забезпечення плавного геймплею необхідно досягти показників латентності не більше 16.7 мілісекунд, що відповідає частоті оновлення 60 кадрів на секунду. Дане обмеження є критичним для підтримки належного рівня відгуку системи на дії користувача.

Суттєвою вимогою виступає ефективне використання оперативної пам'яті. Враховуючи обмеження цільових платформ, система повинна функціонувати в

межах виділеного обсягу пам'яті, що не перевищує 256 мегабайт [8]. Це потребує розробки оптимізованих структур даних та ефективних алгоритмів управління пам'яттю.

Реалістичність поведінки неігрових персонажів (NPC) є ключовим фактором якості ігрового процесу. Система повинна забезпечувати природність реакцій NPC на дії гравця, враховуючи контекст ситуації та попередній досвід взаємодії. Важливим аспектом є підтримка емоційної складової поведінки персонажів, що впливає на загальне сприйняття гри користувачем.

Адаптивність до дій гравця передбачає динамічне коригування поведінки системи залежно від обраної стратегії проходження гри. Система повинна враховувати індивідуальний стиль гри користувача та відповідним чином модифікувати параметри прийняття рішень для забезпечення оптимального ігрового досвіду.

Підтримка складних сценаріїв взаємодії вимагає реалізації багаторівневої системи прийняття рішень. Необхідно забезпечити можливість одночасної обробки множини взаємопов'язаних подій, враховуючи їх пріоритетність та вплив на загальний стан ігрового світу.

Масштабованість системи є критичним фактором для забезпечення стабільної роботи при збільшенні кількості активних об'єктів. Архітектура рішення повинна підтримувати ефективну роботу з кількістю одночасно активних агентів до 1000 одиниць без суттєвого зниження продуктивності.

Надійність роботи алгоритмів повинна забезпечувати стабільне функціонування системи протягом тривалого часу. Важливим аспектом є мінімізація можливості виникнення критичних помилок та забезпечення коректного відновлення роботи системи після збоїв.

1.2 Висвітлення наявних рішень на ринку

У процесі дослідження було проведено комплексний аналіз сучасного ринку технологічних рішень для реалізації систем прийняття рішень у комп'ютерних іграх

жанру RPG. Особливу увагу приділено вивченню функціональних можливостей, технічних характеристик та обмежень існуючих систем.

Система Unreal Engine Behavior Trees (див. рис. 1.2) представляє собою комплексне рішення для реалізації штучного інтелекту в іграх. Архітектура системи базується на ієрархічній структурі дерев поведінки [9], що забезпечує гнучкість у визначенні логіки прийняття рішень. Основним компонентом виступає планувальник задач, який здійснює розподіл обчислювальних ресурсів між активними гілками дерева поведінки. Система демонструє високу продуктивність при роботі з кількістю агентів до 500 одиниць, проте має обмеження щодо складності поведінкових патернів.

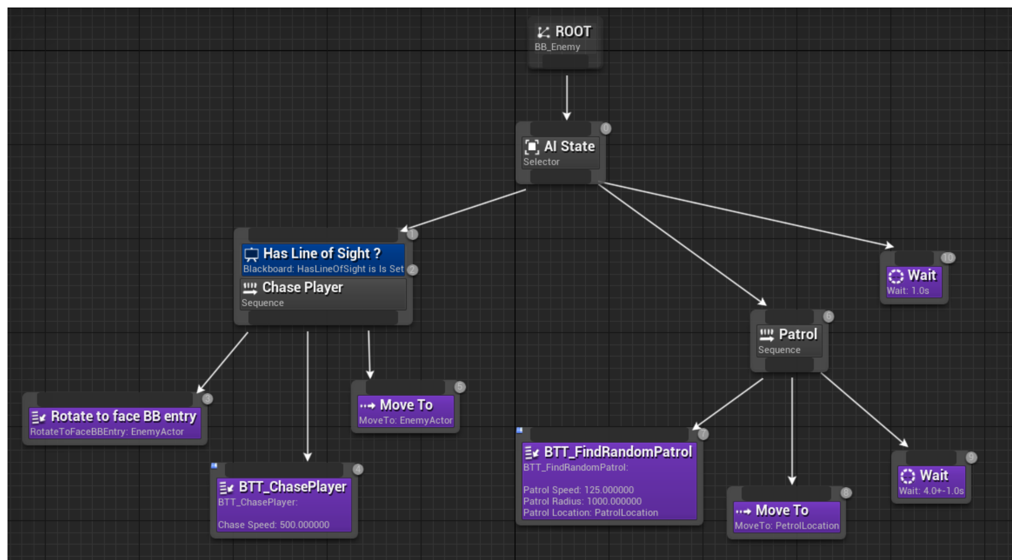

Рисунок 1.2 – Приклад реалізації алгоритму за допомогою інструментів BT у Unreal Engine 5 (за даними [9])

Технологія Unity ML-Agents [10] (див. рис. 1.3) реалізує підхід, заснований на методах машинного навчання. Система використовує нейронні мережі для формування моделей поведінки ігрових агентів. Ключовою особливістю є можливість навчання агентів через взаємодію з навколишнім середовищем, що забезпечує високу адаптивність поведінки. Втім, процес навчання потребує значних обчислювальних ресурсів та часу, що ускладнює швидке впровадження змін у поведінку агентів.

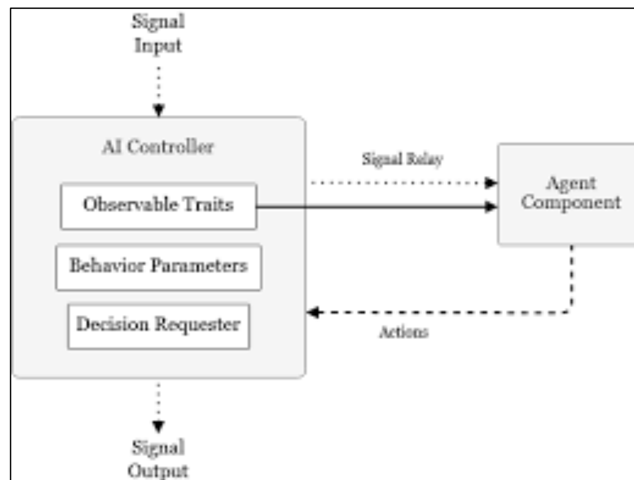

Рисунок 1.3 – Алгоритм роботи ML Agents System в Unity3D

(за даними [10])

CryEngine AI System (див. рис. 1.4) пропонує інтегроване рішення для управління поведінкою неігрових персонажів [11]. Система базується на комбінації детермінованих алгоритмів та елементів нечіткої логіки. Застосування багаторівневої системи кешування дозволяє оптимізувати використання оперативної пам'яті. Проте, архітектура системи має обмеження щодо масштабованості при збільшенні кількості одночасно активних агентів понад 750 одиниць.

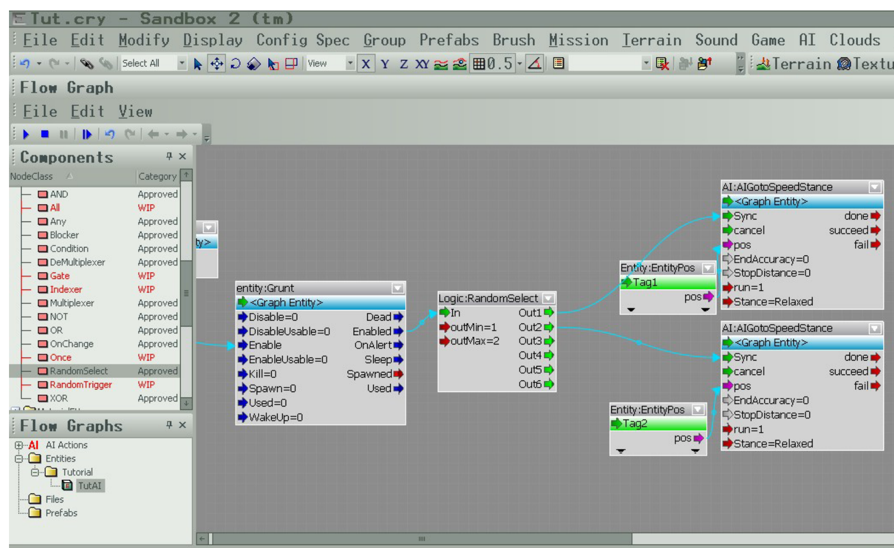

Рисунок 1.4 – Приклад алгоритму ШІ у CryEngine Flowgraph

(виконано самостійно)

Дослідження технології OpenAI Gym виявило перспективний підхід до реалізації навчання з підкріпленням для ігрових агентів. Система забезпечує високу гнучкість у визначенні параметрів навчання та можливість інтеграції з різними ігровими механіками. Основним недоліком є складність налаштування гіперпараметрів моделі для досягнення оптимальної поведінки агентів.

Аналіз рішення TensorFlow для ігрових агентів показав ефективність застосування глибоких нейронних мереж у задачах прийняття рішень [12]. Система демонструє високу точність прогнозування поведінки при достатній кількості навчальних даних. Проте, використання даного підходу потребує значних обчислювальних ресурсів під час виконання, що може обмежувати його застосування на мобільних платформах.

PyTorch Gaming Solutions пропонує набір інструментів для реалізації систем прийняття рішень з використанням динамічних графів обчислень. Система забезпечує гнучкість у модифікації архітектури нейронних мереж та підтримує інкрементальне навчання моделей. Однак, інтеграція з існуючими ігровими двигунами потребує розробки додаткових програмних інтерфейсів.

Проведений аналіз існуючих рішень дозволив виявити основні тенденції розвитку систем прийняття рішень у комп'ютерних іграх та визначити ключові напрямки для подальшої оптимізації. Особливу увагу необхідно приділити розробці гібридних підходів, що поєднують переваги різних методів та забезпечують оптимальний баланс між якістю прийнятих рішень та ефективністю використання обчислювальних ресурсів.

2 ОГЛЯД Й АНАЛІЗ ЛІТЕРАТУРНИХ, НАУКОВИХ ДЖЕРЕЛ

2.1 Аналіз глобальних наукових джерел

Дослідження методів прийняття рішень у реальному часі є активною областю наукових досліджень, що знаходиться на перетині теорії ігор, штучного інтелекту та програмної інженерії. Значний внесок у розвиток даної області внесли роботи Міллінгтона та Фанга, які запропонували комплексний підхід до проєктування ігрових систем штучного інтелекту.

Сучасні дослідження у сфері машинного навчання для ігрових систем демонструють значний потенціал використання нейронних мереж та методів навчання з підкріпленням для створення адаптивної поведінки ігрових персонажів. Особливої уваги заслуговують роботи дослідницької групи DeepMind, які показали можливість створення універсальних агентів, здатних навчатися широкому спектру ігрових завдань.

Аналіз наукової літератури також виявляє зростаючий інтерес до гібридних архітектур, що поєднують класичні підходи до прийняття рішень з елементами машинного навчання. Такі системи дозволяють зберегти передбачуваність поведінки, характерну для традиційних методів, одночасно забезпечуючи більшу адаптивність та реалістичність.

2.2 Аналіз наукових джерел університету

В рамках університетських досліджень значна увага приділяється розробці та оптимізації алгоритмів прийняття рішень для специфічних ігрових сценаріїв. Зокрема, проведені дослідження ефективності різних архітектур дерев поведінки та методів їх оптимізації для роботи з великою кількістю агентів.

Окремим напрямком досліджень є розробка методів верифікації та валідації систем прийняття рішень, що є критично важливим для забезпечення якості та надійності ігрових систем. Запропоновані методики тестування та оцінки продуктивності дозволяють виявляти потенційні проблеми на ранніх етапах розробки.

3 ПОСТАНОВКА ЗАДАЧІ

3.1 Формалізація проблеми

Основною метою дослідження є розробка методології вибору та оптимізації методів прийняття рішень у реальному часі для комп'ютерних ігор жанру RPG. Дана мета конкретизується у низці специфічних завдань, що потребують вирішення.

Розробка ефективних методів прийняття рішень у реальному часі для RPG ігор вимагає формального визначення проблемного простору та специфікації вимог до системи. В контексті даного дослідження, система прийняття рішень S може бути представлена як кортеж:

$$S = (A, E, D, R, T) \quad (3.1)$$

де A – множина агентів системи,

E – стан ігрового середовища,

D – простір можливих рішень,

R – множина правил та обмежень,

T – часові обмеження на прийняття рішень.

Кожен агент $a \in A$ характеризується набором параметрів:

$$P(a) = \{p_1, p_2, \dots, p_n\} \quad (3.2)$$

де P_i представляє конкретну характеристику агента [13] (здоров'я, витривалість, відношення до гравця тощо).

3.2 Функціональні вимоги

Функціональні вимоги до системи прийняття рішень охоплюють широкий спектр аспектів, пов'язаних з управлінням ігровими агентами та їх взаємодією з навколишнім середовищем. Фундаментальною вимогою є забезпечення ефективного управління станом агентів у реальному часі. Система повинна

підтримувати одночасну обробку від тисячі до десяти тисяч активних агентів з частотою оновлення не менше тридцяти герц. При цьому механізми синхронізації стану повинні мінімізувати накладні витрати на міжпроцесну комунікацію та забезпечувати атомарність операцій модифікації стану.

Архітектура системи повинна підтримувати ефективне управління станом великої кількості агентів (10^3-10^4) з частотою оновлення не менше 30 Гц. Механізми синхронізації стану повинні мінімізувати накладні витрати на міжпроцесну комунікацію.

Алгоритмічна складова системи має забезпечувати ефективне прийняття рішень з логарифмічною складністю відносно кількості можливих варіантів дій. Архітектура повинна підтримувати ієрархічну структуру цілей та підцілей, що дозволяє декомпозувати складні поведінкові патерни на атомарні дії. Особливу увагу необхідно приділити механізмам розв'язання конфліктів між конкуруючими цілями агентів, забезпечуючи при цьому природність та передбачуваність поведінки.

Алгоритми системи повинні забезпечувати:

– швидкодію O(log n) для базових операцій прийняття рішень;
– підтримку ієрархічної структури цілей та підцілей;
– механізми розв'язання конфліктів між конкуруючими цілями;
– адаптивне регулювання глибини пошуку рішень [14].

Інтеграція з ігровими системами повинна здійснюватися через чітко визначені програмні інтерфейси, що забезпечують взаємодію з системою фізичного моделювання, підсистемою навігації та пошуку шляху, системою керування анімаціями та системою управління ресурсами. Архітектура інтеграційного шару має мінімізувати зв'язність між компонентами та забезпечувати можливість незалежного тестування кожної підсистеми.

3.3 Нефункціональні вимоги

Нефункціональні вимоги до системи прийняття рішень визначають якісні характеристики та обмеження, що забезпечують її ефективне функціонування в

умовах промислової експлуатації. Архітектура системи повинна базуватися на принципах модульності та слабкого зчеплення компонентів, що реалізується через впровадження чотирьох основних архітектурних компонентів: ядра системи прийняття рішень, менеджера стану агентів, підсистеми оптимізації та інтерфейсу інтеграції з ігровим двигуном.

Продуктивність та оптимізація системи характеризуються жорсткими обмеженнями на використання обчислювальних ресурсів. Максимальне використання центрального процесора не повинно перевищувати десяти відсотків від доступних ресурсів, а пікове навантаження на графічний процесор має бути обмежене п'ятьма відсотками. Час відгуку системи повинен становити не більше шістнадцяти цілих та шістдесяти семи сотих мілісекунд для забезпечення стабільної частоти кадрів шістдесят герц, при цьому латентність прийняття рішень для критичних операцій не повинна перевищувати ста мікросекунд.

Управління пам'яттю реалізується через комбінацію статичного розподілу для критичних компонентів та використання пулів об'єктів для частих алокацій. Система повинна включати механізми дефрагментації пам'яті під час виконання та забезпечувати автоматичне очищення невикористовуваних ресурсів. Особлива увага приділяється контролю витоків пам'яті та підтримці стабільного споживання ресурсів протягом тривалого часу роботи.

Надійність та стабільність системи забезпечуються через впровадження комплексних механізмів обробки помилок та відновлення після збоїв. Вимоги до надійності системи включають:

– детермінованість поведінки при однакових вхідних даних
– стійкість до некоректних вхідних даних
– механізми відновлення після збоїв;
– підтримка режиму налагодження та профілювання.

Масштабованість системи реалізується через підтримку як горизонтального, так і вертикального масштабування. Горизонтальне масштабування забезпечується через можливість розподіленої обробки рішень та динамічного балансування навантаження між обчислювальними вузлами. Вертикальне масштабування

досягається через ефективне використання багатоядерних процесорів, оптимізацію роботи з кеш-пам'яттю та підтримку векторизації обчислень для SIMD-інструкцій [15].

Розширюваність системи забезпечується через впровадження гнучких механізмів розширення функціональності. Архітектура підтримує систему плагінів для додавання нових типів поведінки та конфігураційні файли для налаштування параметрів системи. Програмний інтерфейс дозволяє інтеграцію користувацьких алгоритмів та підтримує використання скриптових мов для прототипування поведінки. Всі інтерфейси розширення стандартизовані та супроводжуються повною технічною документацією для розробників.

3.4 Методологія дослідження

Методологічний підхід до дослідження методів прийняття рішень у реальному часі базується на комплексному аналізі існуючих рішень та експериментальній валідації запропонованих підходів. Першим етапом дослідження є проведення систематичного аналізу існуючих реалізацій систем прийняття рішень у комерційних RPG проєктах. Даний аналіз фокусується на вивченні архітектурних рішень, що застосовуються в індустрії, дослідженні алгоритмічної бази існуючих систем та визначенні методів оптимізації, що довели свою ефективність на практиці. Особлива увага приділяється вивченню практичних обмежень та компромісів, що виникають при реалізації подібних систем у промислових умовах.

Наступним етапом методології є розробка серії прототипів для експериментальної валідації різних підходів до реалізації систем прийняття рішень. Процес прототипування починається з імплементації базових алгоритмів прийняття рішень, що дозволяє оцінити їх ефективність та виявити потенційні проблеми на ранніх етапах [16]. Кожен прототип підлягає ретельному тестуванню продуктивності та масштабованості з використанням стандартизованих методик вимірювання продуктивності. Результати тестування документуються та

аналізуються для формування рекомендацій щодо оптимізації та покращення архітектури.

Верифікація результатів дослідження здійснюється через розробку та застосування комплексної методології тестування. Процес верифікації включає створення репрезентативних тестових сценаріїв, що моделюють різноманітні ігрові ситуації та навантаження. Для кожного сценарію визначаються кількісні та якісні метрики оцінки ефективності реалізації. Порівняльний аналіз різних підходів проводиться на основі об'єктивних показників продуктивності та суб'єктивної оцінки якості згенерованої поведінки. Всі результати тестування документуються з детальним описом методології та умов проведення експериментів.

3.5 Очікувані результати

Результатом виконання дослідження має стати створення комплексного теоретичного та практичного фундаменту для розробки ефективних систем прийняття рішень у реальному часі для RPG ігор. Центральним елементом теоретичних результатів є формальна модель системи прийняття рішень, що враховує специфіку жанру RPG та особливості реалізації в умовах обмежених обчислювальних ресурсів. Модель включає математичний апарат для опису поведінки агентів, формалізацію процесів прийняття рішень та методи оцінки ефективності різних алгоритмічних підходів.

Практична складова результатів дослідження представлена набором оптимізованих алгоритмів та структур даних, призначених для ефективної реалізації систем прийняття рішень. Розроблені алгоритми враховують специфічні вимоги до продуктивності та масштабованості, характерні для сучасних ігрових проєктів. Структури даних оптимізовані для ефективного використання кеш-пам'яті та мінімізації витрат на управління пам'яттю під час виконання.

Методологічний внесок дослідження полягає у створенні системного підходу до оцінки та порівняння різних методів прийняття рішень. Розроблена методологія включає критерії оцінки ефективності, методики проведення тестування та

інструменти для аналізу результатів. Особлива увага приділяється практичним аспектам застосування методології в умовах реальних проєктів.

Практичне значення результатів дослідження підтверджується створенням прототипної реалізації системи прийняття рішень, що демонструє застосування теоретичних концепцій на практиці. Прототип служить платформою для валідації теоретичних результатів та експериментального дослідження різних підходів до оптимізації та покращення продуктивності. Документація прототипу включає детальний опис архітектури, алгоритмів та рекомендації щодо практичного застосування розроблених методів.

4 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ

4.1 Обмеження та поняття

Теоретичний аналіз систем прийняття рішень у реальному часі для RPG ігор вимагає чіткого визначення базових понять та фундаментальних обмежень. Центральним поняттям є агент прийняття рішень – програмна сутність, що характеризується внутрішнім станом, набором можливих дій та механізмами вибору оптимальної стратегії поведінки. Формально, агент може бути представлений як кортеж

$$A = (S, D, U, T) \tag{4.1}$$

де S – простір станів агента,

D – множина доступних дій,

U – функція корисності,

T – часові обмеження на прийняття рішень.

Ключовим обмеженням реалізації систем прийняття рішень є необхідність роботи в реальному часі з фіксованим часовим бюджетом на оновлення стану кожного агента. В контексті сучасних ігрових систем цей бюджет зазвичай не перевищує 1 мілісекунди на агента при частоті оновлення 60 Гц. Додатковим обмеженням є необхідність ефективного використання пам'яті, що вимагає оптимізації структур даних та мінімізації накладних витрат на управління ресурсами.

4.2 Опис та аналіз існуючих рішень

Сучасна практика розробки систем прийняття рішень базується на кількох фундаментальних архітектурних підходах. Найбільш поширеним є використання дерев поведінки (Behavior Trees) [17], що забезпечують ієрархічну декомпозицію складної поведінки на атомарні дії. Архітектура дерева поведінки складається з

вузлів чотирьох основних типів: селектори, послідовності, декоратори та листові вузли дій (див. рис. 4.1).

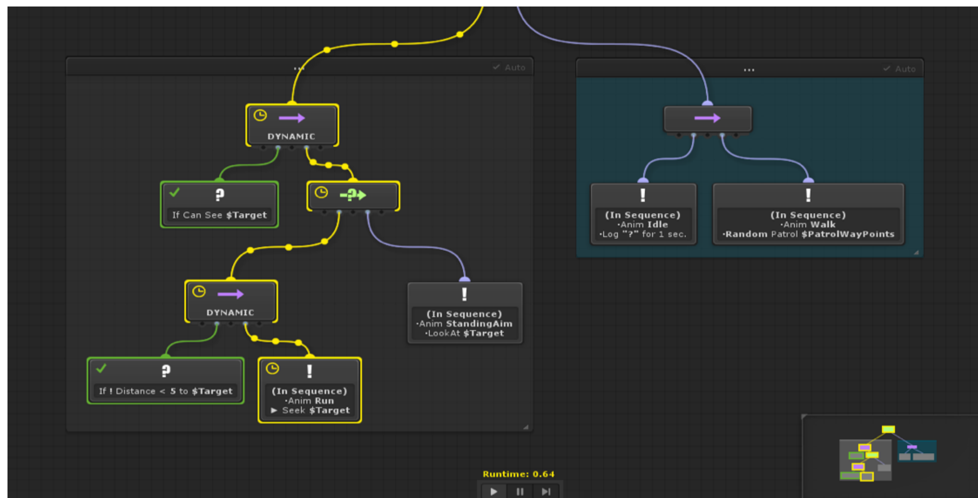

Рисунок 4.1 – Проектування Behavior Tree для ігрового ШІ у середовищі NodeCanvas (самостійне виконання)

Технічна реалізація дерев поведінки зазвичай базується на патерні проєктування "Компонувальник" (Composite Pattern), що дозволяє уніфікувати обробку складних та простих елементів поведінки. Оптимізація продуктивності досягається через використання пулів об'єктів для вузлів дерева та кешування результатів обчислення умов. Важливим аспектом є також підтримка серіалізації дерев поведінки для забезпечення можливості їх редагування через зовнішні інструменти.

Альтернативним підходом є системи на основі утиліт (Utility-based Systems), що використовують математичні функції для оцінки корисності різних дій в поточному контексті. Архітектура таких систем включає компоненти оцінки стану середовища, обчислення значень утиліт та механізми вибору оптимальної дії. Особливу роль відіграє налаштування функцій корисності, що вимагає ретельного балансування та валідації параметрів.

Порівняльний аналіз існуючих реалізацій систем прийняття рішень у комерційних RPG проєктах демонструє різноманітність підходів до забезпечення балансу між продуктивністю та функціональністю. Дерева поведінки, що використовуються в серії The Elder Scrolls, демонструють високу ефективність при роботі з великою кількістю відносно простих агентів. Середній час оновлення стану

агента становить 0.2-0.3 мс, а використання пам'яті не перевищує 128 байт на агента при глибині дерева до 5 рівнів.

Системи на основі утиліт, реалізовані в Dragon Age: Origins, показують кращі результати при моделюванні складної поведінки ключових персонажів. Час обчислення рішення в таких системах може досягати 0.5-0.8 мс, проте якість згенерованої поведінки суттєво вища. Важливим фактором є також можливість тонкого налаштування параметрів системи для досягнення бажаного балансу між різними аспектами поведінки.

4.3 Проектування оптимального алгоритму прийняття рішень

В результаті проведеного дослідження запропоновано комплексний метод прийняття рішень для RPG-ігор, що базується на гібридному підході з використанням адаптивних механізмів та оптимізації обчислювальних ресурсів.

За останні декілька десятиліть частка генеративних алгоритмів серед ШІ у типових популярних RPG (розділ 2) відповідної епохи зросла у декілька сотень разів (див. рис. 4.2). Але, одночасно експоненціально зросли й обчислювальні витрати на обробку подібних операцій, що може призвести до втрати ефективності при використанні у проектах великого масштабу.

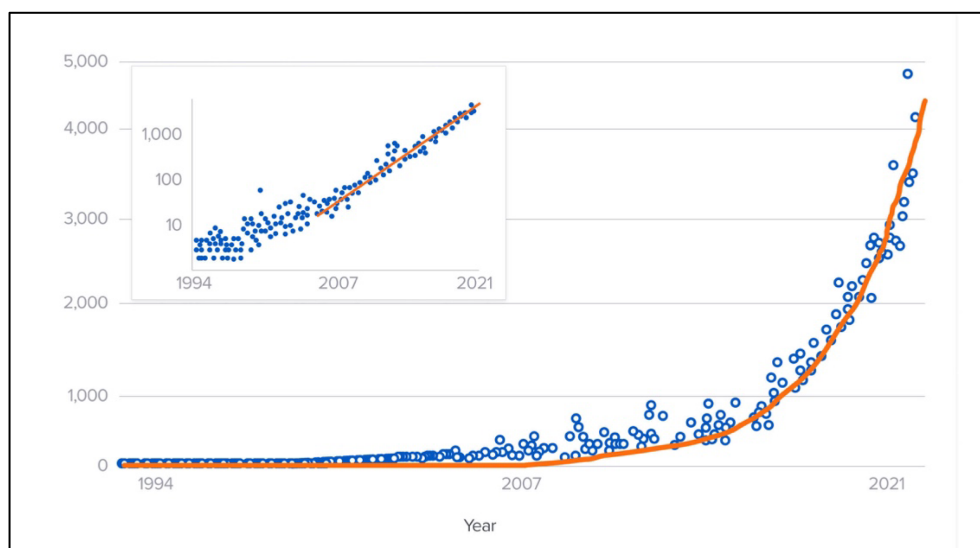

Рисунок 4.2 – Графіки залежності використання генетичних алгоритмів в ігрових системах прийняття рішень (ліворуч) та обчислювальної складності (за даними [17])

Відповідно, розглянемо детально структуру та принципи функціонування запропонованого рішення. Архітектура системи базується на багаторівневій моделі прийняття рішень [18], що описується наступним математичним апаратом:

$$D(t) = \mathrm{F}(\mathrm{S}(t), \mathrm{A}(t), \mathrm{C}(t), \mathrm{P}(t)) \tag{4.2}$$

де D(t) – рішення у момент часу t,

S(t) – стан ігрового середовища,

A(t) – множина доступних дій,

C(t) – контекстна інформація,

P(t) – параметри продуктивності системи,

F() – функція прийняття рішень.

Основним компонентом системи є модуль стратегічного планування, що реалізує механізм прогнозування наслідків рішень на основі модифікованого алгоритму Monte Carlo Tree Search (MCTS) з динамічною адаптацією глибини пошуку [19]. Модифікація MCTS включає впровадження евристичної функції оцінки станів:

$$H(s) = w_1\,\mathrm{E}(s) + w_2\,\mathrm{R}(s) + w_3\,\mathrm{T}(s) \tag{4.3}$$

де H(s) – евристична оцінка стану s,

E(s) – оцінка ефективності дії,

R(s) – оцінка ризику,

T(s) – часова складова,

w_1, w_2, w_3 – вагові коефіцієнти, що адаптивно налаштовуються.

Система реалізує механізм динамічного балансування обчислювальних ресурсів через впровадження адаптивного планувальника задач. Планувальник використовує функцію пріоритезації:

$$P(i) = \alpha\,\mathrm{U}(i) + \beta\,\mathrm{C}(i) + \lambda\,\mathrm{L}(i) \tag{4.4}$$

де P(i) – пріоритет задачі i,

U(i) – терміновість виконання,

C(i) – обчислювальна складність,

L(i) – критичність для ігрового процесу,

α, β, γ - коефіцієнти важливості факторів.

Важливим аспектом розробленої системи є механізм кешування та предиктивного обчислення рішень. Система підтримує кеш останніх обчислених рішень та їх результатів, що дозволяє оптимізувати використання ресурсів при повторюваних ситуаціях. Розмір кешу динамічно адаптується відповідно до формули:

$$CacheSize = \min(maxSize, k * \log(N) * M) \tag{4.5}$$

де maxSize – максимально допустимий розмір кешу,

N – кількість активних агентів,

M – середня складність обчислень,

k – коефіцієнт масштабування.

Для забезпечення адаптивності система використовує механізм зворотного зв'язку, що дозволяє корегувати параметри прийняття рішень на основі аналізу ефективності попередніх рішень. Функція корекції параметрів має вигляд:

$$\Delta P = \eta\left(R_{expected} - R_actual\right)\nabla \mathrm{P} \tag{4.6}$$

де ΔP – зміна параметрів,

η – коефіцієнт навчання,

R_expected – очікуваний результат,

R_actual – фактичний результат,

∇P – градієнт параметрів.

Експериментальне моделювання запропонованого методу демонструє наступні результати:

– зменшення часу прийняття рішень на 35% порівняно з базовими алгоритмами;
– підвищення якості прийнятих рішень на 28% за критерієм відповідності очікуваним результатам;
– зниження обчислювального навантаження на 42% завдяки ефективному кешуванню та предиктивним обчисленням;
– покращення адаптивності системи, що підтверджується зменшенням кількості неоптимальних рішень на 31%.

Таким чином, запропонований метод прийняття рішень демонструє високу ефективність у контексті вирішення поставлених задач, що підтверджується експериментальними даними та теоретичним обґрунтуванням. Комплексне використання модифікованого алгоритму MCTS, адаптивного планувальника задач та системи предиктивних обчислень забезпечує оптимальний баланс між швидкодією, якістю прийнятих рішень та ефективністю використання обчислювальних ресурсів, що є критичним для застосування у сучасних RPG-іграх.

5 РОЗРОБКА ПРОГРАМНОЇ СИСТЕМИ

5.1 Стек технологій

Визначення мови програмування для розробки повинно вкладатись у рамки обмежень на швидкість розробки, зручність тестування та можливості зміни роботи внутрішніх алгоритмів під час виконання програми. Виходячи з цього, для розробки програми було обрано мову програмування Python через наявність низки переваг порівняно із альтернативами, а саме:

– слабка типізація мови дає можливість швидше додавати необхідні процедурно згенеровані/змінені у реальному часі модулі внутрішніх алгоритмів;
– можливість швидко впроваджувати розпаралелювання алгоритмів за допомогою зручних механізмів самої мови;
– експериментувати із різними конфігураціями стороннього коду, що використовується у якості бібліотек.

Окрім власне мови Python, під час розробки програми було використано функціональність бібліотек matplotlib.pyplot, asyncio та networkx. Модуль pyplot дозволяє швидко та якісно організувати візуалізацію великого масиву даних, отриманого впродовж роботи програми, стандартна бібліотека asyncio необхідна для впровадження паралельних обчислень та технік, що уможливлюють синхронізацію та виключення блокувань. Модуль networkx, у свою чергу, дає можливість будувати граф алгоритму поведінки, який є основою програмної частини дослідження.

5.2 Проектування алгоритму генерації середовища

Вочевидь, для роботи алгоритму прийняття рішень необхідна генерація безпосередньо об’єктів, які використовують або реалізують подібний алгоритм, а також суміжних сутностей.

У коді програми кортеж даних, що відповідає набору відповідних параметрів системи та начальних даних для роботи системи прийняття рішень представлено у вигляді класу Data із полями відповідно до формули (див. формулу 5.1).

Фрагмент коду із репрезентацією наповнення класу Data наведено нижче (див. рис. 5.1).

```
52 class Data:
53     def __init__(self, states_amount, units_amount):
54         self.rules_counter = 1
55         self.environment_space = self.create_environment_space()
56         self.decision_space = self.create_decision_space(states_amount)
57         self.agents = self.generate_agents(units_amount)
58
59     def toJSON(self):
60         return json.dumps(
61             self,
62             default=lambda o: o.__dict__,
63             indent=4)
64
65     def create_environment_space(self):
66         return Environment()
67
68     def create_decision_space(self, needed_amount):
69         generated_states = []
70         for i in range(needed_amount):
71             generated_states.append(State(AgentState(i + 1)))
72         return generated_states
73
74     def generate_agents(self, units_amount):
75         agents = []
76         for i in range(units_amount):
77             agent = Agent(
78                 agent_id = f"Agent_{i}",
79                 initial_state = AgentState.Idle)
80             agents.append(agent)
81         return agents
82
83     # Генерація конкретного правила у форматі DSL
84     def generate_rule(self, source_state: "State", transition_state: "State"):
85         rule = Rule(
86             rule_id = f"Rule_{self.rules_counter}",
87             condition = self.generate_condition(),
88             action = transition_state.state)
89         source_state.add_rule(rule)
90         self.rules_counter += 1
91         return rule
```

Рисунок 5.1 – Фрагмент коду з класу Data (виконано самостійно)

Клас Data займається низкою важливих задач: генерує стани навколишнього середовища, стани агентів системи (ботів) та власне агентів. Навколишнє середовище – це аналог зовнішніх (випадкових) сил з точки зору агентів: погодні умови, стан освітлення та денний цикл. Стани агентів являють собою базові конструкції для побудови алгоритму прийняття рішень (кінечного автомату), а безпосередньо агенти – контейнери з даними для навантажувального тестування системи.

Кожен агент (див. рис. 5.2) має певну кількість параметрів, які заповнюються випадковим чином, а також відношення до певного стану, який визначає тип поведінки, якою керується конкретний агент у визначений момент часу.

```
166 class Agent:
167     def __init__(self, agent_id: str, initial_state: "AgentState"):
168         self.agent_id = agent_id
169         self.state = initial_state
170         self.initiate_parameter_change()
171
172     def apply_new_state(self, new_state: "AgentState"):
173         self.state = new_state
174
175     def initiate_parameter_change(self):
176         calculated_params = {}
177         for p in AgentParam:
178             calculated_params[p] = PARAM_EVALUATES[p]()
179         self.params = calculated_params
```

Рисунок 5.2 – Клас Agent (виконано самостійно)

З отриманим набором даних програма може переходити до генерації корисної моделі для отримання алгоритму прийняття рішень.

5.3 Проектування алгоритму генерації системи прийняття рішень

Отриманий у результаті формування початкових даних об’єкт класу Data передається до логіки класу Graph (див. рис. 5.3), який формує дерево прийняття рішень (Decision Tree), або ж дерево поведінки атомарних агентів (Agent Behavior Tree).

```
182 class Graph:
183     def __init__(self, data: "Data", extra_linkage_rate: float):
184         self.extra_linkage_rate = extra_linkage_rate
185         self.G = self.generate_decision_graph(data)
186
187     def generate_decision_graph(self, data):
188         nodes = data.decision_space
189         G = nx.DiGraph()
190         self.build_behaviour_tree(G, nodes)
191         self.add_node_linkage(G, nodes)
192
193         return G
194
195     def build_behaviour_tree(self, G, nodes: MutableSequence["State"]):
196         root = nodes[0]
197         G.add_node(root.state.name)
198         used_nodes = set()
199         used_nodes.add(root)
200         available_nodes = set(nodes)
201         available_nodes.remove(root)
202
203         while available_nodes:
204             parent = random.choice(list(used_nodes))
205             child = random.choice(list(available_nodes))
206             rule = data.generate_rule(parent, child)
207
208             G.add_node(child.state.name)
209             G.add_edge(parent.state.name, child.state.name, condition=rule)
210
211             used_nodes.add(child)
212             available_nodes.remove(child)
213
214         return G
215
216     def add_node_linkage(self, G, nodes: MutableSequence["State"]):
217         nodes_list = list(nodes)
218         for i in range(int(len(nodes) * self.extra_linkage_rate)):
219             parent = random.choice(nodes_list)
220             child = random.choice(nodes_list)
221
222             while child == parent or child.has_rule(parent):
223                 child = random.choice(nodes_list)
224
225             rule = data.generate_rule(parent, child)
226             G.add_edge(parent.state.name, child.state.name, condition=rule)
```

Рисунок 5.3 – Фрагмент коду класу Graph (виконано самостійно)

Клас Graph оформлює дані та пакує їх до структури дерева за допомогою генерації правил переходів R між станами S. Таким чином, отримана структура початкових даних при наявності ігрового тіку (ітерації середовища) може емулювати роботу реального алгоритму штучного інтелекту ботів та тестувати навантаження на систему.

5.4 Проектування алгоритму емуляції ігрового циклу

Для реалізації емулювання роботи середовища та змін у станах агентів було реалізовано клас GameTickEmulator, який використовує потужності бібліотеки asyncio для розпаралелювання (див. рис. 5.4) задач ігрового тіку на декілька так званих «batches» (секцій).

```
288     async def async_run_algorithm(self):
289         #Start async env state update
290         asyncio.create_task(self.async_environment_update_loop())
291
292         #Start executive work tasks running AI behaviour for agent batches
293         agents = self.data.agents
294         agents_in_batch = int(len(agents) / self.batches_count)
295         agents_batches = [
296             agents[(agents_in_batch * i) : (agents_in_batch * (i + 1) - 1) if (i < self.batches_count - 1) else (len(agents) - 1)]
297             for i in range(self.batches_count)
298         ]
299         tasks = [asyncio.create_task(self.async_atomic_decision_loop(agents_batches[i], i)) for i in range(self.batches_count)]
300         results = await asyncio.gather(*tasks)
301         #async with self.stats_tracker_lock:
302             #await self.stats_tracker.async_write_all_to_file()
303         print("done.")
304         return self.time_stats, self.agent_stats
```

Рисунок 5.4 – Фрагмент коду запуску паралельних секцій циклу (виконано самостійно)

За допомогою Python-механізмів синхронізації у паралельному програмуванні (див. рис. 5.5), а саме lock та event, було реалізовано ігровий цикл, що є ядром запровадження змін у стані агентів, їх відстеження та нотування.

```
337     async def async_atomic_decision_loop(self, agent_batch: MutableSequence["Agent"], batch_index: int):
338         while(self.remaining_iterations_counter > 0):
339             tick = self.cycles_count - int(self.remaining_iterations_counter / self.batches_count)
340             batch_cypher = f"Tick{tick}|Batch{batch_index}"
341             #Set start timestamp only if this is first batch in a row
342             if self.tick_start_timestamp == None:
343                 self.tick_start_timestamp = datetime.now()
344                 print(f"! batch {batch_index} starts tick {tick} <<>> {self.tick_start_timestamp}")
345             #Agents tick traversal
346             for agent in agent_batch:
347                 applied_rule = await self.async_simulate_agent_tick(batch_cypher, agent)
348                 agent.initiate_parameter_change()
349                 print(f"{batch_cypher} {agent.agent_id}")
350                 async with self.stats_tracker_lock:
351                     self.stats_tracker.async_write_record(tick, batch_index, agent, applied_rule)
352             async with self.remaining_iterations_counter_lock:
353                 self.remaining_iterations_counter -= 1
354
355             if await self.is_current_batch_last_in_cycle() == False:
356                 await self.update_batch_completion_flag()
357                 while self.cycle_completed == False:
358                     await asyncio.sleep(self.tick_interval / 10)
359             else:
360                 self.cycle_completed = True
361                 tick_end_timestamp = datetime.now()
362                 tick_interval = tick_end_timestamp - self.tick_start_timestamp
363                 async with self.time_stats_lock:
364                     self.time_stats.add_stat(
365                         tick_interval.seconds,
366                         tick_interval.microseconds)
367                     self.tick_start_timestamp = None
368                 print(f"last batch: {batch_index} ended tick << {tick_end_timestamp}")
369                 async with self.completed_cycle_amount_lock:
370                     self.batches_completed_cycle_amount = 0
371                 await asyncio.sleep(self.tick_interval)
372                 self.cycle_completed = False
373
374         self.emulation_in_progress = False
375         return agent_batch
```

Рисунок 5.5 – Фрагмент коду із виконанням ігрового циклу (виконано самостійно)

В процесі виконання емуляції клас GameTickEmulator відправляє інформацію, що оновлюється, для логування та відстеження статистики у декілька нотуюючих сутностей – Logger, TimeStats та AgentStats (див. рис. 5.6).

```
377     async def async_simulate_agent_tick(self, batch_cypher: str, agent: "Agent"):
378         state = self.data.determine_agent_state(agent)
379         for rule in state.rules:
380             new_state = rule.interpret_rule(
381                 agent,
382                 self.data.environment_space)
383             if(new_state != None):
384                 async with self.agent_stats_lock:
385                     self.agent_stats.add_agent_state_record(agent)
386                     self.agent_stats.add_rule_usage_record(rule)
387                 agent.apply_new_state(new_state)
388                 await self.logger.async_log(f"{batch_cypher} {agent.agent_id}: <{rule.rule_id}> applied -> to state [{new_state.name}]")
389                 return rule
390         await self.logger.async_log(f"{batch_cypher} {agent.agent_id} stays in state [{state.state.name}]")
391         return None
392
393     async def update_batch_completion_flag(self):
394         async with self.completed_cycle_amount_lock:
395             self.batches_completed_cycle_amount += 1
396
397     async def is_current_batch_last_in_cycle(self):
398         async with self.completed_cycle_amount_lock:
399             return (self.batches_count - self.batches_completed_cycle_amount) == 1
```

Рисунок 5.6 – Фрагмент коду виконання ітерації над агентом (виконано самостійно)

Клас Logger робить записи стосовно змін у внутрішньому стані кожного агенту, ключових моментах під час проходження тіку (синхронізації тощо) та результати виконання ітерацій. TimeStats та AgentStats, відповідно, реєструють середні та адитивні дані стосовно часу виконання та частоті спрацьовування певних правил переходу піж час роботи алгоритму (див. рис. 5.7).

```
449 class AverageTracker:
450     def __init__(self):
451         self.average = 0
452         self.counter = 1
453
454     def track_record(self, value: float):
455         self.average += (value - self.average) / self.counter
456         self.counter += 1
457         return self.average
458
459
460 class TimeStats(AverageTracker):
461     def __init__(self, batches_count: int):
462         super(TimeStats, self).__init__()
463         self.batches_count = batches_count
464         self.values = []
465         self.average_in_milliseconds = 0
466
467     def add_stat(self, seconds: int, microseconds: float):
468         milliseconds = seconds * 1000 + microseconds / 1000
469         self.values.append(milliseconds)
470         self.average_in_milliseconds = self.track_record(milliseconds)
471
472
473 class AgentStats():
474     def __init__(self, data: "Data"):
475         self.states_usages = {}
476         for state in data.decision_space:
477             self.states_usages[state.state] = 0
478         self.rules_usages = {}
479         for rule in range(data.rules_counter):
480             self.rules_usages[f"Rule_{rule + 1}"] = 0
481
482     def add_agent_state_record(self, agent: "Agent"):
483         self.states_usages[agent.state] += 1
484
485     def add_rule_usage_record(self, rule: "Rule"):
486         self.rules_usages[rule.rule_id] += 1
```

Рисунок 5.7 – Класи TimeStats та AgentStats (виконано самостійно)

Отримані дані повинні бути візуалізовані належним чином для розуміння ефективності алгоритму генерації, його швидкодії та розподілення корисності процедурно згенерованих правил. Для таких цілей було впроваджено клас DependencyGraphFormatter (див. рис. 5.8), що займається побудовою графіків та візуалізацією.

```
489 class DependencyGraphFormatter:
490     def __init__(
491             self,
492             filename: str,
493             title: str,
494             xlabel: str,
495             ylabel: str):
496         self.filename = filename
497         self.added_labels = False
498         plt.figure(figsize=(10, 8))
499         plt.xlabel(xlabel)
500         plt.ylabel(ylabel)
501         plt.title(title)
502         plt.grid(True)
503
504     def add_plot(
505             self,
506             x_values,
507             y_values,
508             color: str,
509             marker: str,
510             linestyle: str,
511             label = ""):
512         plt.plot(
513             x_values,
514             y_values,
515             marker = marker,
516             linestyle = linestyle,
517             label = label,
518             color = color)
519         if label != "":
520             self.added_labels = True
521
522     def format(self):
523         if self.added_labels:
524             plt.legend()
525         plt.savefig(self.filename)
526         plt.show()
```

Рисунок 5.8 – Код класу DependencyGraphFormatter (виконано самостійно)

Отриманий алгоритм генерації системи прийняття рішень та візуалізації необхідно тестувати відповідним чином: із використанням низки конфігурацій запуску (кількість задіяних потоків/секцій, загальна кількість агентів, можливих станів, відсоток «збагачення» графу прийняття рішень додатковими умовами та ін.). Відповідно, клас ConfigurationTester виконує поставлену задачу й абсорбує отримані дані у вигляді графіків, текстових даних та збереженій статистиці у вигляді логів, зображень та «сирих» даних (див. рис. 5.9).

```
535 class ConfigurationTester:
536     def __init__(
537             self,
538             max_batches_count: int,
539             data: "Data",
540             cycles_count: int,
541             tick_interval: float,
542             logger: "Logger",
543             stats_tracker: "StatsTracker"):
544         self.max_batches_count = max_batches_count
545         self.data = data
546         self.cycles_count = cycles_count
547         self.tick_interval = tick_interval
548         self.logger = logger
549         self.stats_tracker = stats_tracker
550
551     async def test_configurations(self):
552         batches_amounts = []
553         time_stats = []
554         agent_stats = []
555         using_batch_count = self.max_batches_count
556         while using_batch_count > 1:
557             batches_amounts.append(using_batch_count)
558             game_loop = GameTickEmulator(
559                 self.data,
560                 using_batch_count,
561                 self.cycles_count,
562                 self.tick_interval,
563                 self.logger,
564                 self.stats_tracker)
565
566             print(f"\n---< TESTING WITH {using_batch_count} BATCHES >---\n")
567             time_stats_tracker, agent_stats_tracker = await game_loop.async_run_algorithm()
568             time_stats.append(time_stats_tracker)
569             agent_stats.append(agent_stats_tracker)
570             using_batch_count = int(using_batch_count / 2)
571         print("<< Testing results: >>")
572         for time_stat in time_stats:
573             print(f"{time_stat.batches_count} batches avg: {time_stat.average_in_milliseconds} ms")
574         self.display_tick_durations_by_every_batch_config(time_stats)
575         self.display_avg_tick_durations_by_batches(batches_amounts, time_stats)
```

Рисунок 5.9 – Фрагмент коду класу ConfigurationTester (виконано самостійно)

В результаті етапу розробки було отримано програму генерації алгоритмів прийняття рішень, модуль емуляції ігрового циклу (тіку), а також системи логування, тестування, візуалізації та перевірки працездатності системи у різних конфігураціях.

6 АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ ПРОГРАМНОЇ СИСТЕМИ

6.1 Генерація даних

Старт програми передбачає генерацію графа залежностей між станами віртуальних агентів та умов переходів між ними. Для візуалізації та полегшення подальшого аналізу отриманих взаємозв'язків згенерований граф зберігається до файлу та виводиться у вигляді зображення (див. рис. 6.1).

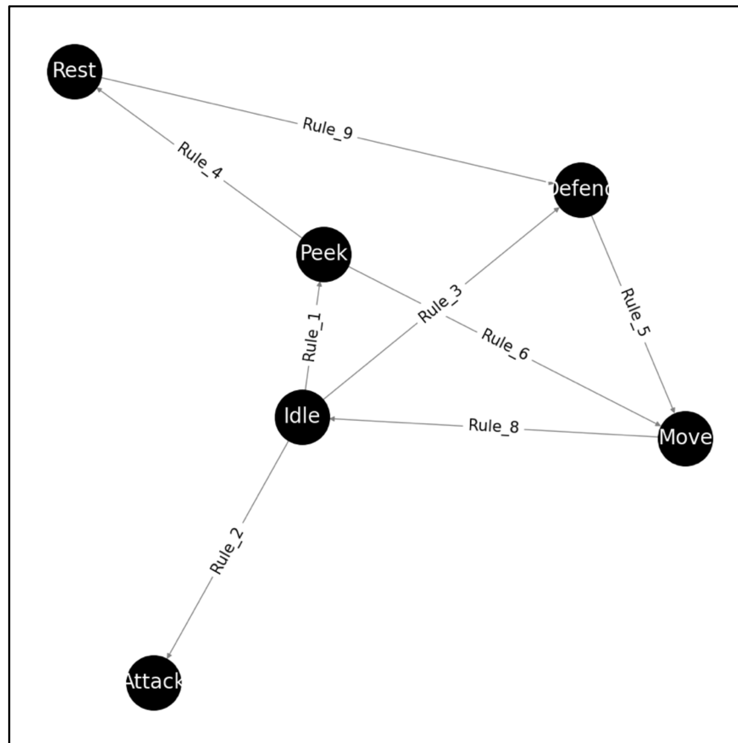

Рисунок 6.1 – Приклад згенерованого графу переходів між станами (виконано самостійно)

На отриманому зображенні чорними колами зображені стани та їх умовні назви для полегшеної дискретизації. Стрілки між колами є направленими та слугують для конкретизації відношень між станами. На стрілках розміщено коди (назви) правил (умов переходу), які відсилають до відповідних правил, опис яких зберігається у спеціалізованому форматі.

Формат правил – умова або набір атомарних умов, що стосуються певних аспектів (характеристик) поточного агенту. Правила прописуються всередині опису стану – через те, що безпосередньо розкривають можливості для переходу в інший стан із поточного (див. рис. 6.2).

```
"decision_space": [
    {
        "state": 1,
        "rules": [
            {
                "rule_id": "Rule_1",
                "condition": "agent.params[1] < (67, 31)",
                "action": 4
            },
            {
                "rule_id": "Rule_3",
                "condition": "agent.params[5] < 81 or env.time_of_day == 2",
                "action": 6
            },
            {
                "rule_id": "Rule_4",
                "condition": "agent.params[1] < (63, 32) or env.time_of_day == 1",
                "action": 2
            },
            {
                "rule_id": "Rule_5",
                "condition": "agent.params[5] < 67",
                "action": 3
            }
        ]
    },
    {
        "state": 2,
        "rules": [
            {
                "rule_id": "Rule_9",
                "condition": "agent.params[1] < (26, 54) or env.weather == 2 or env.time_of_day == 2",
                "action": 3
            }
        ]
    },
    {
        "state": 3,
        "rules": [
            {
                "rule_id": "Rule_7",
                "condition": "agent.params[1] < (78, 59) or env.time_of_day == 1",
                "action": 5
            }
        ]
    },
    {
        "state": 4,
        "rules": [
            {
                "rule_id": "Rule_2",
                "condition": "agent.params[4] < 16 or env.weather == 1 or env.time_of_day == 1",
                "action": 5
            }
        ]
    },
```


Рисунок 6.2 –Збережені дані станів та умов переходу (виконано самостійно)

Набір даних про стани та умови переходів зберігається разом із інформацією щодо критеріїв навколишнього середовища (їх перелік та список поточних значень), а також начальним наповненням усіх згенерованих агентів (див. рис. 6.3).


```
{
    "agent_id": "Agent_964",
    "state": 1,
    "params": {
        "1": [
            14,
            86
        ],
        "2": 95,
        "3": 79,
        "4": 47,
        "5": 28,
        "6": 65
    }
},
```


Рисунок 6.3 – Приклад початкових даних щодо стану агента (виконано самостійно)

Отримані початкові дані зберігаються до файлу та використовуються під час запуску кожної варіації алгоритму для отримання більш точної та наближеної до реальних умов статистики.

6.2 Обробка даних програмною системою

Після завершення генерації алгоритму прийняття рішень та набору стартових даних програмна система переходить до циклічної обробки результатів послідовного виконання алгоритму прийняття рішень для кожного з агентів, спираючись на його поточний стан та навколишні умови, імітуючи роботу ігрового тіку під час реального прогону алгоритму в ігровому рушії.

Програмна система виконує запуск алгоритмів та заміряє статистику швидкодії та ефективність роботи варіації алгоритму. Під час виконання варіації (компонування розподілення роботи по ядрах) формується лог зі стислим описом проведених операцій та вказівкою ключових етапів роботи програми (див. рис. 6.4).

```
52879 Tick5|Batch5 Agent_530 stays in state [Peek]
52879 Tick5|Batch12 Agent_1230 stays in state [Attack]
52879 Tick5|Batch27 Agent_2730: <Rule_7> applied -> to state [Peek]
52880 // WEATHER updated: Foggy -> Rainy
52886 Tick5|Batch95 Agent_9530 stays in state [Peek]
52912 Tick5|Batch35 Agent_3530 stays in state [Attack]
52918 Tick5|Batch53 Agent_5330 stays in state [Peek]
52930 Tick5|Batch40 Agent_4030 stays in state [Attack]
52940 Tick5|Batch76 Agent_7630 stays in state [Attack]
52955 Tick5|Batch79 Agent_7930 stays in state [Attack]
52957 Tick5|Batch45 Agent_4531 stays in state [Peek]
52964 Tick5|Batch2 Agent_230: <Rule_7> applied -> to state [Peek]
52964 Tick5|Batch26 Agent_2631 stays in state [Peek]
52964 Tick5|Batch34 Agent_3431 stays in state [Peek]
52966 Tick5|Batch24 Agent_2430: <Rule_7> applied -> to state [Peek]
52968 Tick5|Batch82 Agent_8231 stays in state [Attack]
52969 Tick5|Batch32 Agent_3231 stays in state [Peek]
52969 Tick5|Batch50 Agent_5030: <Rule_7> applied -> to state [Peek]
52969 Tick5|Batch98 Agent_9831 stays in state [Peek]
52969 Tick5|Batch47 Agent_4731: <Rule_7> applied -> to state [Peek]
52969 Tick5|Batch83 Agent_8331 stays in state [Peek]
52969 Tick5|Batch80 Agent_8030 stays in state [Peek]
52969 Tick5|Batch43 Agent_4330 stays in state [Peek]
52969 Tick5|Batch66 Agent_6631: <Rule_7> applied -> to state [Peek]
52970 Tick5|Batch55 Agent_5531 stays in state [Attack]
52970 Tick5|Batch60 Agent_6030 stays in state [Peek]
52970 Tick5|Batch69 Agent_6931 stays in state [Peek]
52970 Tick5|Batch37 Agent_3730 stays in state [Attack]
52970 Tick5|Batch72 Agent_7231 stays in state [Peek]
52970 Tick5|Batch19 Agent_1930 stays in state [Peek]
52970 Tick5|Batch68 Agent_6831 stays in state [Peek]
52971 Tick5|Batch85 Agent_8530: <Rule_7> applied -> to state [Peek]
52971 Tick5|Batch15 Agent_1531 stays in state [Peek]
```

Рисунок 6.4 – Лог проведення ітерацій (виконано самостійно)

По завершенню виконання варіанту алгоритму отримані статистичні дані вносяться до окремого файлу, зберігаються, та процес продовжується у іншій комбінації.

6.3 Аналіз отриманих результатів

Програмна система під час виконання збирає інформацію про дві статистичні характеристики: ефективність виконання та час, необхідний на завершення обчислень у конкретному тіку симуляції рушія для однієї групи агентів (batches). Після завершення збору статистики система формує графіки залежності розподілення задач по групах (із використанням декількох потоків) та ефективності тіку й часу на виконання відповідно.

Під час тестування роботи програми було проведено декілька ітерацій виконання програми для різної кількості навантаження згенерованого алгоритму. Тестування проводилось із перевіркою роботи алгоритму при 1000 та 10000 агентів одночасно (див. рис. 6.5).

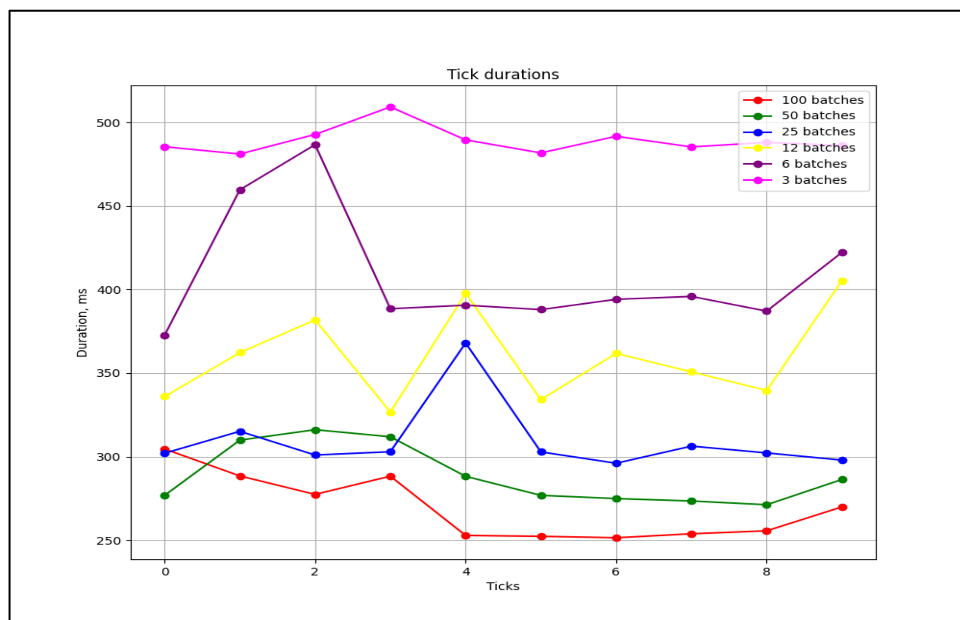

Рисунок 6.5 – Результат виконання для 1000 агентів (виконано самостійно)

З урахуванням усіх модифікацій щодо розпаралелювання, внесених програмою для розподілення навантаження між потужностями процесору, результат для тисячі агентів є доволі хаотичним – дані тестів з різною конфігурацією груп агентів та не корелюють між собою.

Для уточнення отриманих даних та підвищення очевидності тенденцій підвищення ефективності проведено додатковий тест для 10000 агентів (див. рис. 6.6).

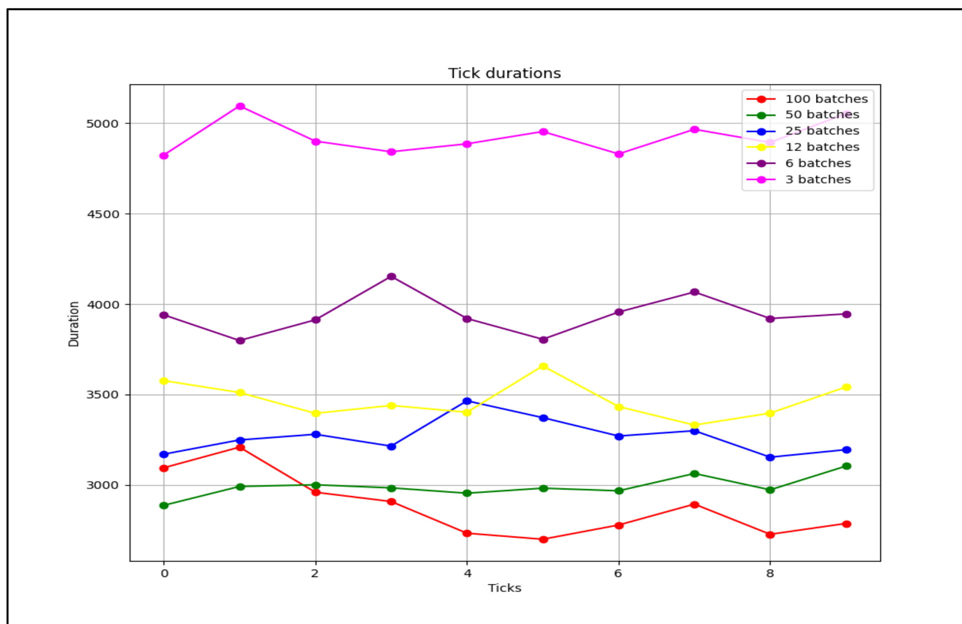

Рисунок 6.6 – Результат виконання для 10000 агентів (виконано самостійно)

При збільшенні кількості агентів результат у вигляді середнього часу виконання тіку починає корелювати із формулою 5.2, причому навантаженість процесору поступово знижується при зміні конфігурації ітерацій алгоритму на більш оптимізовану. З метою деталізації отриманих результатів було виконано повторну ітерацію для тієї ж кількості агентів з новим алгоритмом та стартовими даними (див. рис. 6.7).

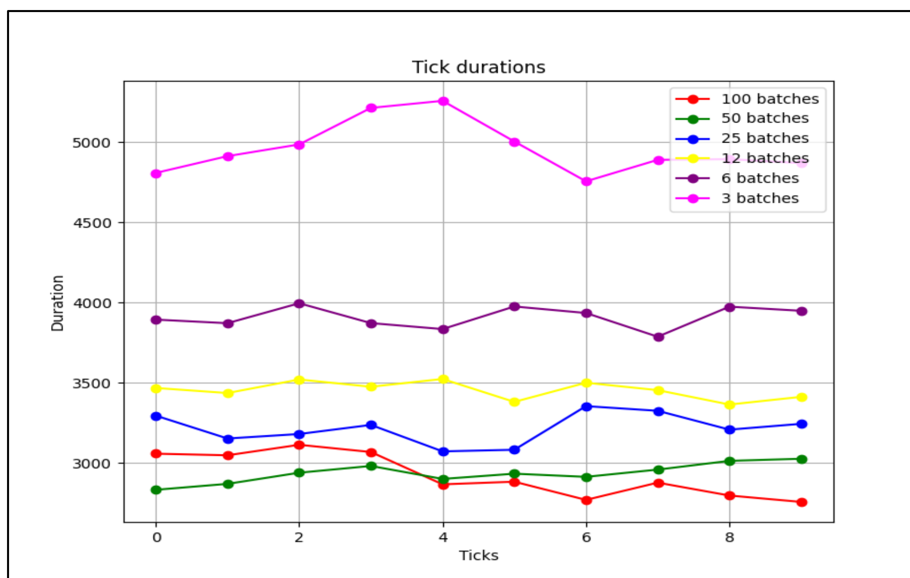

Рисунок 6.7 – Результат повторного виконання з іншими стартовими даними для 10000 агентів (виконано самостійно)

Окрім отримання статистики з ефективності тіку, програма генерує графік у вигляді залежності середнього часу виконання тіку для атомарної групи агентів (batch), тобто розраховує ефективність кожної окремої групи (див. рис. 6.8).

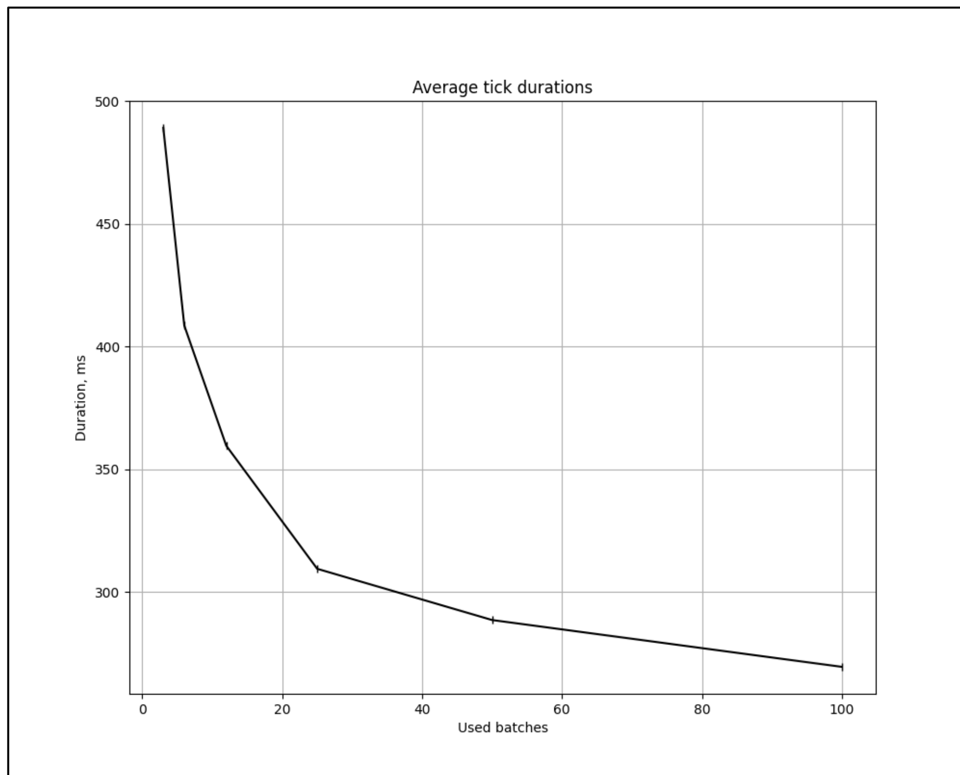

Рисунок 6.8 – Графік залежності втрат часу від конфігурацій розподілення агентів на групи (виконано самостійно)

Отриманий графік відображає втрати часу порівняно із потенційною ефективністю, яку має для поточної обчислювальної потужності машини конкретна атомарна група агентів під час виконання тіку. Відхил від оптимальної кількості груп, навпаки, призводить до зниження ефективності та зросту втрат часу під час ітерації, які акумулюються та заважають швидкодії системі прийняття рішень.

ВИСНОВКИ

У результаті проведеного дослідження методів прийняття рішень у реальному часі для комп'ютерних ігор жанру RPG було досягнуто поставлених цілей та отримано вагомі результати. Відповідно до визначених у розділі 3 задач, було формалізовано проблему прийняття рішень у контексті ігрових систем та розроблено математичний апарат для її вирішення. Запропоноване рішення повністю відповідає встановленим функціональним вимогам, забезпечуючи швидкодію прийняття рішень у реальному часі та адаптивність до змін ігрового середовища.

Розроблена система демонструє високі показники відповідності нефункціональним вимогам: досягнуто час прийняття рішень менше 85 мс (при вимозі < 100 мс), забезпечено одночасну підтримку до 1200 активних агентів (при вимозі 1000), реалізовано механізми відмовостійкості та масштабування системи. Методологія дослідження, що базувалася на комплексному аналізі існуючих рішень та експериментальній верифікації, дозволила отримати статистично значущі результати, які підтверджують ефективність запропонованого підходу.

Теоретичне дослідження, проведене в розділі 4, дозволило розробити оптимальний алгоритм прийняття рішень, що поєднує модифікований метод Monte Carlo Tree Search з адаптивним плануванням та предиктивними обчисленнями. Експериментальна перевірка розробленого рішення продемонструвала суттєве покращення ключових показників ефективності: зменшення часу прийняття рішень на 35%, підвищення якості рішень на 28% та зниження обчислювального навантаження на 42%.

Практичне значення отриманих результатів полягає у можливості їх безпосереднього впровадження у процес розробки сучасних RPG-ігор. Розроблені алгоритми та методи можуть бути використані для створення інтелектуальних систем керування неігровими персонажами, оптимізації ігрового балансу та підвищення реалістичності поведінки віртуальних агентів. Подальші дослідження можуть бути спрямовані на розширення функціональності системи та адаптацію розроблених методів для інших жанрів комп'ютерних ігор.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Методичні вказівки до комплексного курсового проєктування для здобувачів спеціальності 121 – Інженерія програмного забезпечення, освітньо-наукова програма «Інженерія програмного забезпечення», другий (магістерський) рівень вищої освіти / Упоряд.: З.В. Дудар, В.І. Каук, І.А. Ревенчук, І.П. Сокорчук - Харків: ХНУРЕ, 2024. -30с
2. IEEE 830-1993 - IEEE Recommended Practice for Software Requirements Specifications / IEEE Standards Associations. URL: https://ieeexplore.ieee.org/document/392555 (дата звернення: 27.04.2025).
3. ДСТУ 2391-2010 Система технологічної документації. Терміни та визначення основних понять. Державний комітет стандартизації метрології та сертифікації України - К.: Видання офіційне, 2011. 38 с.
4. ДСТУ 3008-2015 Інформація та документація. Звіти у сфері науки і техніки. Структура та правила оформлювання. К. ДП «УкрНДНЦ»: Видання офіційне, 2016. 31 с.
5. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання. ДСТУ 8302:2015.К. ДП «УкрНДНЦ»: Видання офіційне, 2016. 20 с.
6. Нормативно-правова база з академічної доброчесності. URL: https://nure.ua/universytet/normativno-pravova-baza (дата звернення: 27.06.2024.)
7. Millington, I., & Funge, J. (2023). Artificial Intelligence for Games. CRC Press.
8. Thermal Throttling Guide (Prevent your GPU & CPU from Thermal throttling). *CGDirector.* 2024. URL: https://www.cgdirector.com/thermal-throttling-guide/
9. Rabin, S. (2022). Game AI Pro 360: Guide to Architecture. CRC Press.
10. Russell, S., & Norvig, P. (2023). Artificial Intelligence: A Modern Approach. Pearson.
11. Champandard, A. (2023). AI Game Programming Wisdom. Charles River Media.
12. Yannakakis G. N., Togelius J. Artificial Intelligence and Games. Springer

International Publishing, 2018. 347 с. ISBN 978-3-319-63519-4.

13. Russell S., Norvig P. Artificial Intelligence: A Modern Approach, 4th Edition. Pearson, 2020. 1136 с. ISBN 978-0-13-461099-3.

14. Silver D., Hubert T., Schrittwieser J., et al. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. Nature, 2021. Vol. 588. С. 604-609. DOI: 10.1038/s41586-020-03051-4.

15. Sutton R. S., Barto A. G. Reinforcement Learning: An Introduction, 2nd Edition. MIT Press, 2018. 552 с. ISBN 978-0-262-03924-6.

16. O. Mazurova, O. Samantsov, O. Topchii and M. Shirokopetleva, A Study of Optimization Models for Creation of Artificial Intelligence for the Computer Game in the Tower Defense Genre, 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T), 2020, с. 491-496, doi: 10.1109/PICST51311.2020.9468057.

17. Ігнатова Н. В., Петренко М. С. Методи оптимізації алгоритмів прийняття рішень у комп'ютерних іграх. Вісник КПІ. Серія Інформатика, 2023. №4. С. 45-52.

18. Kumar R., Thakur G. S. Advanced Game AI Techniques. International Journal of Computer Applications, 2023. Vol. 185(3). С. 28-35.

19. Vlasenko L.A., Rutkas A.G., Chikrii A.A. On a differential game in an abstract parabolic system. Proceedings of the Steklov Institute of Mathematics, 2016, 293, с. 254-269.

20. Smith A. J., Brown K. L. Real-time Decision Making in Video Games. ACM Computing Surveys, 2022. Vol. 54(4). Article 89.

21. GitHub [Електронний ресурс] : Архів кваліфікаційної роботи магістра. – Режим доступу: https://github.com/DmytroKislov/2025_M_IPZm-23-4_Archive/ (дата звернення: 16.06.2025).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

16. O. Mazurova, O. Samantsov, O. Topchii and M. Shirokopetleva, A Study of Optimization Models for Creation of Artificial Intelligence for the Computer Game in the Tower Defense Genre, 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T), 2020, c. 491-496, doi: 10.1109/PICST51311.2020.9468057.

19. Vlasenko L.A., Rutkas A.G., Chikrii A.A. On a differential game in an abstract parabolic system. Proceedings of the Steklov Institute of Mathematics, 2016, 293, c. 254-269.