Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра математики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**РОЗРОБКА ГРИ НА UNITY/C# З ІМПЛЕМЕНТАЦІЄЮ ЕЛЕМЕНТІВ ШТУЧНОГО ІНТЕЛЕКТУ**»

Виконав: студент 4-го року навчання,

Освітньої програми «Прикладна математика», 113

Стефанюк Євген Ігорович

Керівник Корнійчук Ю.В.____

Асистент

Рецензент _____
(прізвище та ініціали)

Кваліфікаційна робота захищена з оцінкою _____

Секретар ЕК _____

«____» _____ 20____ р.

Київ – 20___

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри математики,
проф., д.ф.-м.н.
_____Р. К. Чорней
(підпис)
«___»_____2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на кваліфікаційну роботу

студентові Стефанюку Євгену Ігоровичу факультету інформатики 4 курсу

ТЕМА Розробка гри на Unity/C# з імплементацією елементів штучного інтелекту

Зміст ТЧ до курсової роботи:

Індивідуальне завдання
Зміст
Анотація
Вступ
1 Алгоритми машинного навчання для гри-платформера
2 Еволюційні алгоритми
3 Алгоритми навчання з підкріпленням
4 Розробка гри
5 Результати
Висновки
Список літератури

Дата видачі „___” _____ 2022р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Розробка гри на Unity/C# з імплементацією елементів штучного інтелекту

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на кваліфікаційну роботу	Жовтень	
2.	Обговорення деталей теми та розробка плану виконання	Жовтень-Листопад	
3.	Огляд технічної літератури за темою роботи та опрацювання матеріалів	Жовтень-Січень	
4.	Написання теоретичної частини роботи	Лютий-Квітень	
5.	Розробка практичної частини та завершення теоретичної частини роботи	Березень-Квітень	
6.	Попередній захист кваліфікаційної роботи	Травень	
7.	Захист кваліфікаційної роботи	Травень-Червень	

Студент: Стефанюк Євген Ігорович

Керівник: Корнійчук Юлія Вікторівна

“__” _____ 2023

ЗМІСТ

Анотація

Метою цієї кваліфікаційної роботи є розробка гри на Unity та дослідження ефективності алгоритму NEAT для проходження цієї гри. У роботі розглядались різноманітні алгоритми машинного навчання, які використовуються в ігрових середовищах. Були отримані результати навчання агентів, використовуючи алгоритм NEAT, та було проведено порівняння ефективності з іншими методами машинного навчання.

Вступ

Актуальність

Складно переоцінити важливість штучного інтелекту та машинного навчання в нашому житті. Автопілот допомагає посадити літак, медики за допомогою штучного інтелекту можуть ставити точніші діагнози пацієнтам, а рекомендації для покупок в Інтернеті за допомогою нього стають все точнішими. У багатьох сферах життя, від спорту до систем безпеки, він покращує якість людського життя, допомагає вирішувати різноманітні проблеми та збільшувати продуктивність.

Штучний інтелект - це галузь науки, яка почала свій розвиток з досліджень моделі нейронної мережі, запропонованої W. Pitts та W. McCulloch у 1943 [1]. Ця праця покращила розуміння людського мозку, а отже, штучного інтелекту і машинного навчання. У цій галузі працювали не лише видатні математики та інформатики, а й нейропсихологи та нейробіологи. Темпи розвитку штучного інтелекту зростають експоненційно, і все більше робіт, чимраз складніших, стають автоматизованими. Дедалі більше людей через це висловлюють занепокоєння і звертають увагу на небезпеки, пов'язані з штучним інтелектом, від перевершення розумового рівня людини до захоплення влади і навмисного чи ненавмисного винищення людства, однак поки що це є спекуляціями і далеко не всі поділяють таку думку.

Також машинне навчання швидко розвивається і в ігровій сфері [3]. Сам термін "машинне навчання" вперше був введений Артуром Семюелем у 1959 році у праці, де він досліджував процедури машинного навчання для гри в шашки [2]. Найбільш відомими застосуваннями машинного навчання є стратегічні ігри, такі як шахи чи го, в яких агенти глибинного навчання змагаються з гравцем, однак штучний інтелект присутній у дуже багатьох жанрах ігор, від ботів у шутерах до процедурно згенерованих рівнів ігор action-adventure.

У своїй роботі я досліджую різноманітні алгоритми пошуку інформації та навчання агентів проходити рівні. Один з цих алгоритмів, а саме алгоритм NEAT, було імплементовано у гру-платформер і досліджено його переваги та недоліки.

Ці дослідження допомагають розуміти, які алгоритми та підходи до навчання агентів є найбільш ефективними в контексті ігрових середовищ.

Ігрова індустрія з кожним роком розвивається все швидше [4], так само як і машинне навчання, тому тема кваліфікаційної роботи є дуже актуальною. Активно досліджуються і розвиваються нові алгоритми в іграх для різноманітних задач: від виявлення нових підходів для розв'язання проблем та оптимізації цих алгоритмів до звичайного збільшення задоволення гравця та вдосконалення геймплею.

Мета, завдання дослідження

Метою кваліфікаційної роботи є розробка гри на Unity з імплементацією алгоритму машинного навчання для проходження агентами гри та дослідження переваг та недоліків інших алгоритмів.

Це передбачає розгляд таких задач:

- Створення гри на Unity.
- Використання алгоритму NEAT для проходження різних рівнів.
- Дослідження того, як еволюція структури нейронної мережі впливає на прогрес агента в грі.
- Дослідження того, як агенти вчаться вирішувати нові задачі на різних рівнях.
- Дослідження того, які фактори можуть впливати на швидкість навчання та досягнення оптимальних результатів.
- Дослідження різних генетичних алгоритмів та їхнє порівняння.
- Дослідження різних алгоритмів з підкріпленням та їхнє порівняння.

РОЗДІЛ 1: Алгоритми машинного навчання для гри-платформера

Для проходження гри-платформера можна застосовувати різні алгоритми машинного навчання. У своїй роботі я досліджую два напрями таких алгоритмів - еволюційні алгоритми та навчання з підкріпленням.

1.1 Еволюційні алгоритми

Еволюційні алгоритми - це, як випливає з назви, алгоритми, які моделюють процес природної еволюції і вирішують проблеми за допомогою процесів, які імітують поведінку живих організмів [5]. Такі алгоритми використовуються в багатьох галузях, від комбінаторної оптимізації до прокладання маршрутів і планування. Ці алгоритми відбирають найкращих агентів з популяції, які еволюційно змінюються через мутації та схрещування, для знаходження кращих стратегій. Ці алгоритми дуже добре оптимізують задачі, однак вони не завжди знаходять оптимальні рішення, тому їх зазвичай використовують з іншими методами для досягнення кращої продуктивності.

Основними перевагами еволюційних алгоритмів є:

- Глобальна оптимізація: Еволюційні алгоритми спроможні знаходити оптимальні рішення для простору можливих рішень, що чудово підходить для задач на оптимізацію, які не можна розв'язати звичайними методами.
- Гнучкість: Ці алгоритми можуть бути застосовані до широкого спектру задач, включно зі змінними середовищами чи середовищами з локальними оптимумами.
- Різноманітність: Ці алгоритми можуть уникати потрапляння в локальні оптимуми за допомогою різноманітної популяції рішень.
- Автономність: Еволюційним алгоритмам не потрібно навчальних даних для пошуку оптимальних рішень.

Однак еволюційні алгоритми також мають кілька недоліків, наприклад:

- Застрягання в локальних оптимумах: Спроможність знаходити оптимальні рішення не дає гарантії, що алгоритм не застрягне в локальних оптимумах чи знайде оптимальне рішення.

- Час виконання: Для складних проблем з великими просторами пошуку знаходження оптимального рішення може бути витратним по часу.
- Розмірність: Також зі збільшенням розмірності простору параметрів проблеми, ефективність еволюційних алгоритмів може зменшуватись.
- Підбір параметрів: Для знаходження оптимального рішення необхідно підібрати правильні параметри, що може зайняти багато часу і ресурсів.

1.2 Навчання з підкріпленням

Навчання з підкріпленням - галузь машинного навчання, яка зосереджується на прийнятті рішень. Ці методи машинного навчання винагороджують певні дії, які нам потрібні від агентів, і карають за ті, які непотрібні [6]. Основна відмінністю навчання з підкріпленням від навчання з учителем полягає в тому, що агент повинен вчитися на своїх помилках. У навчанні з учителем модель отримує навчальні дані з правильними відповідями і навчається передбачати ці відповіді. З іншого боку, в навчанні з підкріпленням відсутні правильні відповіді, тому агент повинен самостійно визначати, які дії здійснювати для виконання завдання. Замість використання навчального набору даних, агент повинен навчитися приймати рішення в певному середовищі, які максимізують нагороду, зі свого досвіду.

Основними перевагами навчання з підкріпленням є:

- Узагальнення: Навчання з підкріпленням навчає агента використовувати свої навички для нових задач і швидко адаптуватися до нових умов.
- Гнучкість: Як і еволюційні алгоритми, навчання з підкріпленням можна застосовувати до широкого спектру задач.
- Автономність: Навчанню з підкріпленням не потрібно початкові умови, оскільки він на основі власного досвіду вчиться виробляти оптимальні стратегії.

Основними недоліками навчання з підкріпленням є:

- Ресурсоємність: Навчання з підкріпленням потребує багато ресурсів та часу для навчання, особливо для складних алгоритмів та великих просторів дій та станів.
- Проблема часового розподілу кредитів (credit assignment problem): Оскільки у навчанні з підкріпленням агент намагається максимізувати нагороду в довгостроковій перспективі, буває важко визначити, які саме дії агента призвели до досягнення позитивного результату.
- Застрягання в локальних оптимумах: Як і з еволюційними алгоритмами, агент може застрягати в локальних оптимумах і бути неспроможним навчитися стратегіям, за допомогою яких можна досягнути оптимального рішення.
- Підбір параметрів: Вибір оптимальних параметрів потребує багато часу і експериментів, а погані параметри можуть давати низьку продуктивність.

РОЗДІЛ 2: Еволюційні алгоритми

2.1 Класичний генетичний алгоритм

Генетичний алгоритм - це еволюційний алгоритм пошуку для вирішення оптимізаційних проблем і проблем пошуку в машинному навчанні, схожий на природній відбір [7]. Ідея алгоритму доволі проста і ґрунтується на принципі "виживає найсильніший". Основні етапи генетичного алгоритму такі:

- Створення початкової популяції.
- Тестування та оцінювання популяції за допомогою функції допасованості (fitness function).
- Відбір найкращих осіб для створення нового покоління.
- Мутації та/або схрещування осіб. Після цього знову переходимо до тестування популяції, допоки не буде досягнено поставленої задачі.

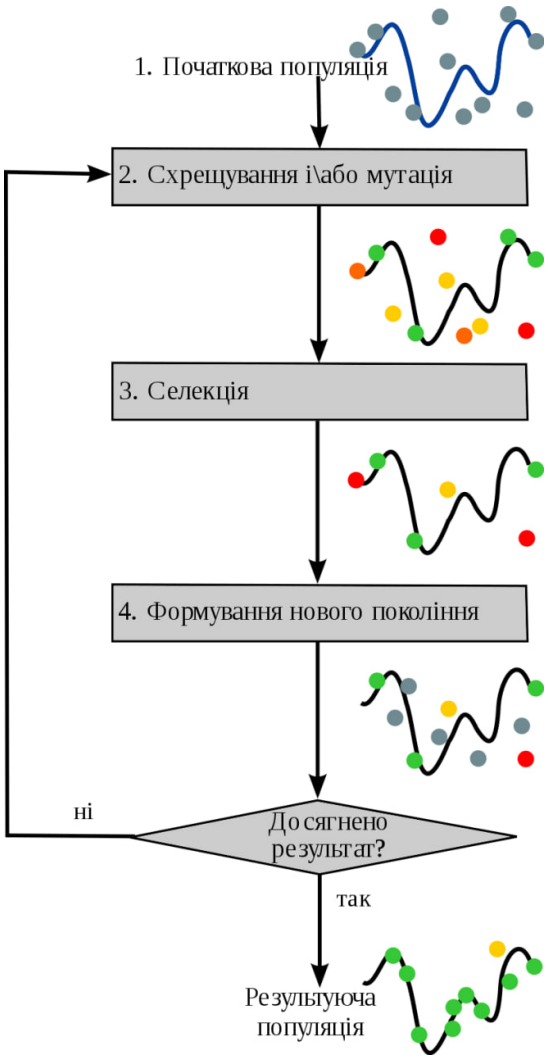

Рис. 1 — Схема роботи генетичного алгоритму [21]

2.1.1 Створення початкової популяції

Першим кроком алгоритму є створення початкової популяції. Генетичний алгоритм не залежить від вхідних даних, тому популяція створюється випадковим чином. Це дозволяє перевірити весь діапазон простору пошуку, а особи з поганими діями швидко відсіюються.

2.1.2 Тестування та оцінювання популяції

Другий крок - тестування та оцінювання популяції. Функція допасованості визначає, наскільки "допасованою" є особа до задачі, себто наскільки вона конкурентноспроможна порівняно з іншими. Ця функція кожній хромосомі присвоює якесь значення, за допомогою якого проводиться відбір. Сама функція допасованості залежить від поставлених цілей. Наприклад, для платформера вона присвоює особі тим більше значення, чим ближче ця особа до фінішу.

2.1.3 Відбір найкращих осіб для створення нового покоління

Наступним кроком є відбір найкращих осіб за допомогою значень, які їм присвоїла функція допасованості. Є декілька способів провести відбір:

- Пропорційний відбір (рулетка): За допомогою значення, яке функція допасованості присвоює кожній хромосомі, визначається ймовірність відбору цих хромосом. Якщо f_i - функція допасованості певної хромосоми i, а N - кількість особин популяції, то ймовірність відбору задається такою формулою [9]:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

  Наприклад, ймовірність хромосоми з оцінкою 2 пройти відбір вдвічі більша, ніж у хромосоми з оцінкою 1. Це відрізняється від відбору з усіченням, який просто вибирає якусь пропорцію найкращих особин, і дозволяє не застрягати в локальних оптимумах — існує ймовірність, що у слабших хромосом можуть бути певні особливості, які дозволяють досягнути глобального оптимуму в довгостроковій перспективі.

- Відбір за рангом: Це модифікована форма пропорційного відбору [7]. Замість точної оцінки функції допасованості цей відбір використовує ранги - особинам з більшим рангом надають більшу перевагу, ніж тим, ранг яких менший. Метод рангового вибору зменшує шанси передчасної збіжності розв'язку до локальних мінімумів [9]. Також використовується тиск відбору, який є ступенем переваги кращих особин - чим він більший, тим більше кращим особинам надається перевага [11]. Він може приймати значення від 1 (тиску нема) до 2 (високий тиск). Ймовірність P для певного рангу R_i з тиском відбору sp розраховується за формулою [22]:

$$P(R_i) = \frac{1}{n}(sp - (2sp - 2)\frac{i-1}{n-1}, 1 \leq i \leq n, 1 \leq sp \leq 2,$$

$$P(R_i) \geq 0, \sum_{i=1}^{n} P(R_i) = 1$$

  Перевага відбору за рангом, окрім тиску відбору, який можна відрегулювати, полягає в тому, що в слабших хромосом є шанс покращуватися і відтворюватися [10].

- Турнірний відбір: Суть цього методу полягає в тому, щоб провести декілька так званих "турнірів" між випадковим чином вибраними хромосомами з популяції. Тиск відбору можна легко налаштувати залежно від розміру популяції, яка бере участь у турнірі [11]. Переможцями турніру стають особини, які мають найбільшу оцінку функції допасованості, а ймовірність їхнього відбору далі для схрещування розраховується за формулою:

$$P_i = p * (1-p)^{i-1}$$

Звісно, це не всі способи відбору [8]. Різні типи методів відбору в генетичному алгоритмі використовуються для забезпечення різноманітності та пошуку оптимальних рішень в просторі пошуку.

2.1.4 Схрещування

Одним з найголовніших етапів є створення нащадків, наступного покоління з відібраних "батьків". Це відбувається за допомогою схрещування — процесу, що

комбінує генетичну інформацію двох "батьків" для створення популяції нащадків. Рекомбінуючи частини хороших рішень, генетичний алгоритм має більшу ймовірність створити краще рішення [13].

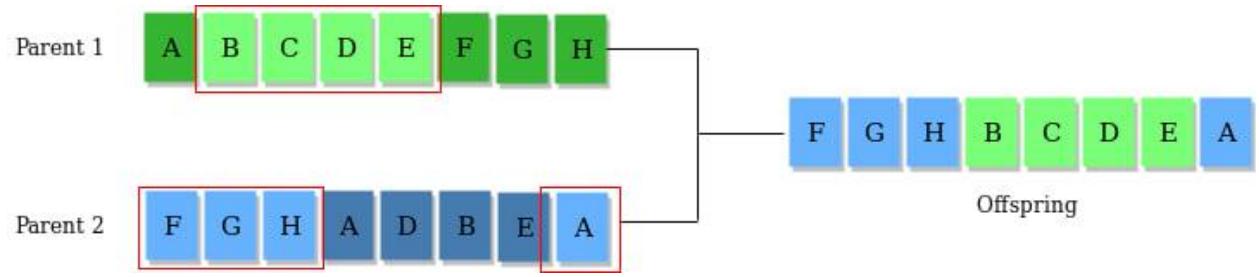

Рис. 2 — Приклад оператора схрещування для двох масивів бітів [23]

Одним з найпопулярніших операторів схрещування є k-точкове схрещування, для якого випадковим чином вибирається k точок схрещування і між хромосомами відбувається обмін ділянками, які обмежені цими точками схрещування, що створює двох нащадків, які несуть в собі інформацію від кожного "батька". Також використовується уніфіковане схрещування, у якому кожен біт вибирається з однаковою ймовірністю від кожного батька [14]. Іншим способом створити нащадків може бути клонування відібраних батьків, які після цього мутують для спроби створити кращі рішення для проблеми.

Для різних еволюційних алгоритмів існують різні структури даних, щодо яких може бути використаний оператор схрещування, а також різні оператори схрещування. Застосування різних структур даних та операторів схрещування дозволяє забезпечити різноманітність популяції та дослідження простору рішень, що може покращити результати еволюційного процесу та знайти оптимальні рішення.

2.1.5 Мутація

Мутація - одна з ключових складових генетичного алгоритму. Цей процес проводить певні випадкові зміни в генотипі нащадків. Мутація допомагає врізноманітнювати популяцію, запобігаючи передчасній конвергенції особин, та уникати потрапляння в локальні оптимуми, тобто застрягати на рішеннях, що у підсумку не зможуть вирішити поставлену задачу. Мутації іноді призводять до того, що нащадки стають гіршими за батьків, однак вони також дозволяють експериментувати з новими рішеннями, які в довгостроковій перспективі

можуть досягнути глобального оптимуму і бути корисними в процесі еволюції для досягнення кращих рішень. Зазвичай мутації відбуваються з невеликою ймовірністю для уникнення значних змін в поведінці агента, які можуть вповільнити процес еволюції.

2.2 Алгоритм NEAT

Алгоритм NEAT - це генетичний алгоритм машинного навчання, призначений для еволюції штучних нейронних мереж. Основна відмінність від звичайного генетичного алгоритму полягає в тому, що алгоритм працює з нейронними мережами. Для його роботи непотрібно задавати розмір нейронної мережі - цей алгоритм розпочинає зі створення початкового набору простих нейронних мереж з невеликою кількістю нейронів і зв'язків. Після цього під дією мутацій і схрещування нейронні мережі еволюціонують, а зв'язки змінюються для того, щоб мережа могла виконувати складніші задачі. У NEAT можуть з'являтися нові структури, а наявні з'єднання — змінюватися або видалятися. Цей алгоритм намагається вирішити декілька проблем, які присутні в інших генетичний алгоритмах та TWEANN (штучні нейронні мережі, що еволюціонують топології та ваги) зокрема, про що пишуть його розробники: "We identify three major challenges for TWEANNs and present solutions to each of them: (1) Is there a genetic representation that allows disparate topologies to crossover in a meaningful way? Our solution is to use historical markings to line up genes with the same origin. (2) How can topological innovation that needs a few generations to optimize be protected so that it does not disappear from the population prematurely? Our solution is to separate each innovation into a different species. (3) How can topologies be minimized throughout evolution without the need for a specially contrived fitness function that measures complexity? Our solution is to start from a minimal structure and grow only when necessary." [12]

2.2.1 Представлення геному

У алгоритмі NEAT геном (індивідуальна нейронна мережа) представлений таким чином:

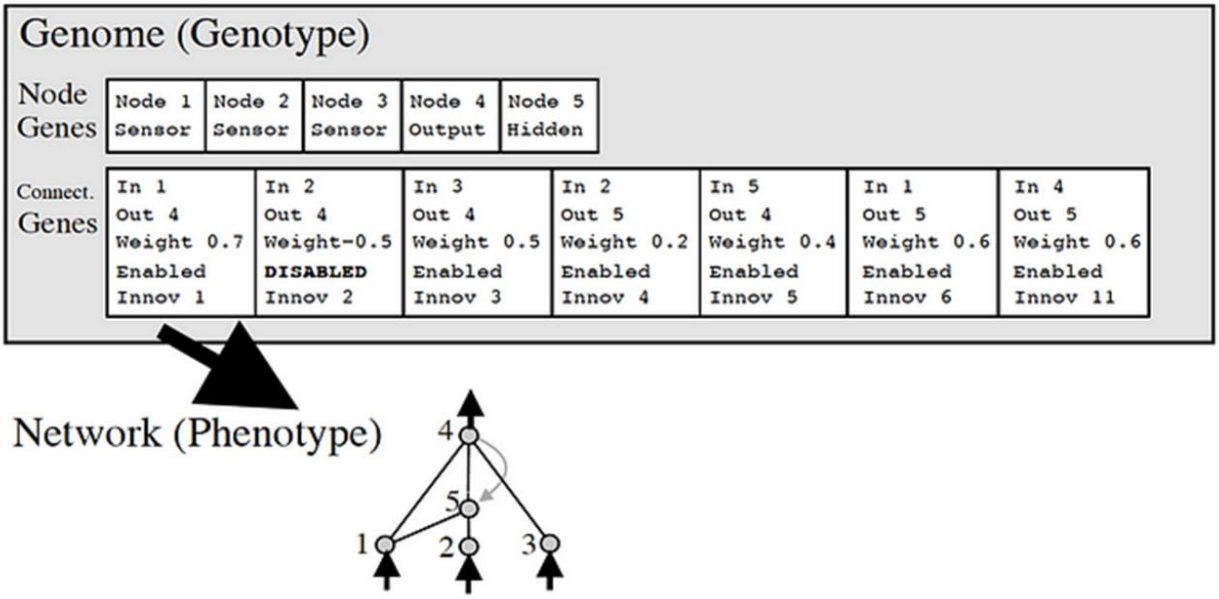

Рис. 3 — Структура геному алгоритму NEAT [24]

У цьому алгоритмі кожний агент представлений через два списки генів, гени вузлів та гени з’єднань. Під час еволюції приховані вузли можуть додаватися чи видалятися, а вхідні (inputs) та вихідні (outputs) вузли не еволюціонують. У генах з'єднання міститься інформація про початок та кінець з’єдання, його вагу (тобто силу зв'язку між нейронами), чи активне це з’єднання та номер іновації, який є ідентифікатором для відстеження еволюції мережевих структур між поколіннями [12]. Також у NEAT для кодування мережі у фенотип використовується пряме кодування, що означає, що кожне з'єднання та нейрон мережі мають явне представлення.

2.2.2 Типи мутацій

В алгоритмі NEAT використовуються різні типи мутацій для зміни генетичної інформації в геномах нейромереж. Серед них виділяють:

- Мутація з'єднання: ця мутація змінює структуру нейромережі, додаючи нове з'єднання між будь-якими двома генами вузлів. Пара генів вузлів вибирається випадковим чином, і між ними створюється з'єднання з випадковою вагою та наступним номером іновації.
- Мутація вузлів: ця мутація створює новий прихований вузол у нейромережі. Для цього випадковим чином вибирається з'єднання між двома вузлами і додається новий вузол. Після його додавання початкове

з'єднання деактивується і з'являються два нові з'єднання. З'єднанню між першим вузлом у списку і новим вузлом присвоюється вага зі значенням 1, а другому - вага початкового з'єднання. Зі зміною структури мережі також переоцінюється її функція допасованості.

- Мутація ваги: ця мутація змінює вагу випадково вибраного з'єднання. Вага може бути помножена на випадкове число від 0 до 2, або взагалі вибрана випадковим чином, що може змінити її знак.
- Мутація активації: ця мутація полягає в зміні активації з'єднання. Для цього береться випадковим чином певний процент з'єднань і змінюється активація з'єднання — якщо з'єднання активне, то воно деактивується, якщо неактивне — то активується.

Мутації в алгоритмі NEAT дозволяють експериментувати з різними змінами структури та параметрів нейромережі, що допомагає нейромережам розширювати структуру, знаходити оптимальні рішення для конкретних завдань, адаптуватись до зміни умов та покращувати продуктивність нейромереж у процесі еволюції.

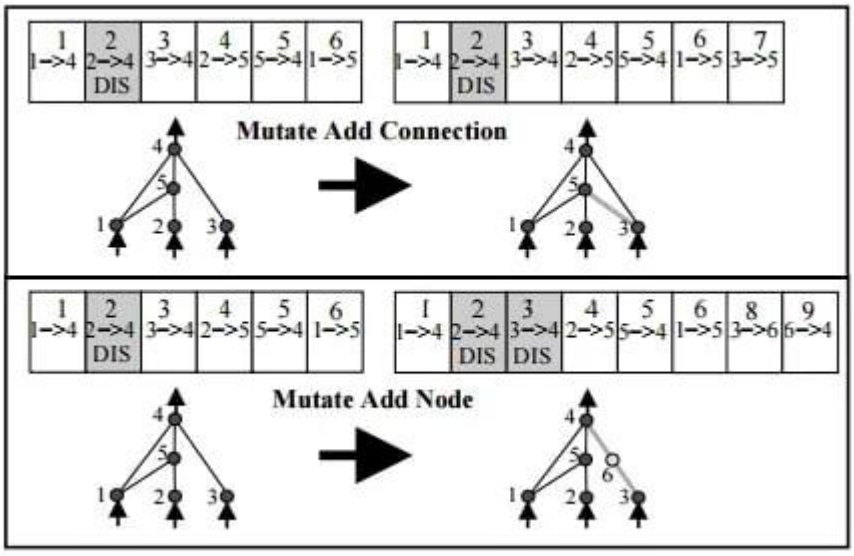

Рис.4 — Типи мутацій в алгоритмі NEAT, які змінюють структуру нейромережі [24]

2.2.3 Проблема схрещування

Перша проблема для TWEANN виникає під час операції схрещування. Якщо геноми двох нейромереж схрещувати випадковим чином, то може бути створена нейромережа, яка втратить інформацію і не буде функціональною. Також різні нейромережі можуть бути різних розмірів з різними з'єднаннями, через що потрібно якось вирішити проблему вирівнювання двох нейромереж, які з першого погляду не є сумісними одна з одною. У випадку NEAT, ця проблема вирішується за допомогою номерів інновації, які є історичними маркерами. Це означає, що за допомогою них можна дізнатися "історію" певної нейромережі і наскільки вона сумісна з іншою нейромережею для схрещування. Для успішного схрещування алгоритм повинен вміти визначити, які гени з'єднання сумісні одне з одним, а які — ні. Якщо у генів однаковий номер інновації, то це значить, що вони походять від спільного предка, а отже, сумісні, навіть якщо їхні ваги різняться. Для цього вводиться глобальний номер інновації, змінна, яка збільшується на одиницю щоразу, коли через мутацію з'являється новий ген. Оскільки нащадки зберігають номер інновації для кожного гена, отриманого від батьків, то це дозволяє відстежувати історію еволюційних змін та передавати найкращі рішення нащадкам. За допомогою цього алгоритм NEAT проводить схрещування без втрати інформації та ймовірності створення нефункціональної нейромережі. [15]

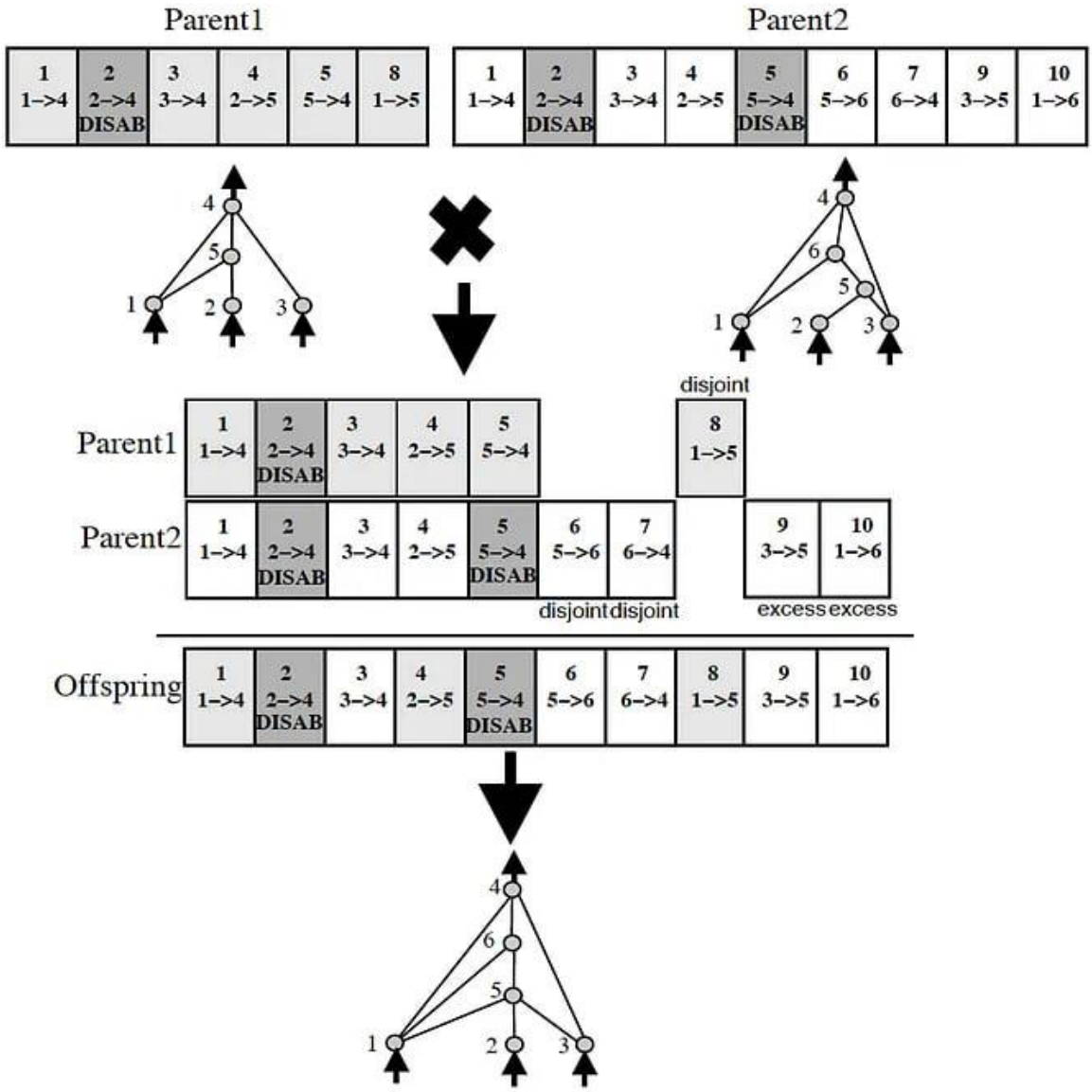

Рис. 5 — Схема роботи схрещування в алгоритмі NEAT [24]. Номери інновації дозволяють легко проводити схрещування без потреби попереднього аналізу батьків

Також існують гени, які не є сумісними. Їх називають надлишковими генами (якщо номер інновації більший за найбільший номер інновації гена іншої нейромережі) або неперетинний (якщо він менший). Кількість цих генів прямо впливає на сумісність двох нейромереж — чим їх більше, тим менше нейромережі сумісні. Під час схрещування несумісні гени успадковуються від батька, який є допасованішим до заданої задачі (якщо обидва батьки однаково допасовані, тоді вони успадковуються випадковим чином).

2.2.4 Групування популяції

Другою проблемою для TWEANN є передчасне зникнення особин популяції, які в довгостроковій перспективі можуть бути корисними для досягнення глобального оптимуму, однак яким потрібно доволі багато генерацій для покращення рішень. Для уникнення цього алгоритм NEAT розділяє агентів на групи залежно від схожості структури їхніх нейромереж. Це дозволяє забезпечити різноманіття популяції та поступовий розвиток структур для досягнення кращих рішень. Щоб розділити геноми на групи, необхідно визначити відстань їхньої сумісності. Для вимірювання цієї відстані використовується кількість несумісних генів, а також середня різниця їхньої ваги:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W}$$

де E - кількість надлишкових генів, D - кількість непересічних генів, $\overline{W}$ - середня різниця ваги відповідних генів, коефіцієнтами c_1, c_2 і c_3 можна регулювати важливість трьох факторів відповідно до поставленої задачі, а N відповідає за кількість генів у більшому геномі для нормалізації розміру [12]. Після цього ми вибираємо певний поріг схожості δ_t, і, порівнюючи з ним відстань між геномами, групуємо їх на види. Також у NEAT використовується явний поділ допасованості [16], тобто геноми певної групи ділять між собою допасованість цієї групи до задачі, щоб не було переваги одного геному над іншими. Якщо загальна розділена допасованість групи більша за середню в популяції, то кількість особин в популяції зростає, якщо менше - зменшується. Це розраховується за формулою [12]:

$$N_j' = \frac{\sum_{i=1}^{N_i} f_{ij}}{\overline{f}}$$

де N_j — кількість особин в групі до зміни, N_j' — кількість особин після зміни, f_{ij} — допасованість особини i в групі j, а $\overline{f}$ — середня допасованість популяції.

Третьою проблемою є те, що TWEANN зазвичай починають з популяцією випадковим чином створених топологій [17] [18] або ж розмір і форму

нейромереж необхідно задавати вручну [19]. Це врізноманітнює популяцію, однак оптимальні розмір та структура нейромережі для рішення поставленої задачі зазвичай невідомі, тому такий підхід може призвести до створення великої кількості неефективних, непрацездатних чи недостатньо пристосованих до задачі нейромереж або дуже складні нейромережі, оптимізація яких займає багато часу та ресурсів. NEAT цю проблему вирішує за допомогою створення малої нейромережі, які може виконувати базові задачі, після цього відбувається контрольована еволюція структур нейромереж для знаходження оптимального рішення задачі, без витрат на оптимізацію неефективних структур, що значно скорочує час навчання і забезпечує швидкий пошук оптимального рішення.

2.2.5 Висновки

Підсумовуючи, можна сказати, що алгоритм NEAT є хорошою альтернативою для навчання з підкріпленням, є гнучкішим алгоритмом та стійкішим до шумів порівняно з іншими TWEANN, а використання змінної структури нейромережі робить її ефективнішою для різноманітних задач. Однак цей алгоритм має і свої обмеження, наприклад, повільну конвергенцію, однак найголовнішим недоліком є те, що NEAT не створює дуже великі нейронні мережі, що може бути недостатнім для вирішення складніших завдань. У таких випадках рекомендується використовувати інші алгоритми, наприклад, навчання з підкріпленням, які здатні створювати складніші та потужніші нейронні мережі.

РОЗДІЛ 3: Алгоритми навчання з підкріпленням

3.1 Навчання з підкріпленням

Reinforcement Learning або навчання з підкріпленням - це підгалузь машинного навчання, яка займається розв'язанням задач, де агент повинен навчитися взаємодіяти з навколишнім середовищем, виконуючи дії для максимізації нагороди або виграшу.

3.2 Основні терміни

Основними термінами навчання з підкріпленням є:

- Агент – суб’єкт, який може взаємодіяти з навколишнім середовищем
- Середовище – віртуальне або реальне оточення, у якому агент може виконувати певні дії. Зазвичай ця система є стохастичною, тобто генерується випадковим чином.
- Дії – сукупність рухів, які агент може здійснювати у відповідь на стан середовища.
- Стан – різні ознаки та спостереження про середовище після виконаних агентом дій.
- Нагорода – відгук про виконані агентом дії. Нагорода може бути як позитивною, так і негативною .
- Стратегія – спосіб вибору дій з огляду на стан середовища.
- Цінність – нагорода, яка буде видана агенту в довгостроковій перспективі через виконання певних дій

Оскільки навчання з підкріпленням застосовується до задач з послідовними рішеннями, то ця підгалузь широко застосовується в ігровій сфері. Алгоритми пошуку з підкріпленням, такі як Q-навчання або методи на основі градієнту, можуть бути використані для навчання агента здійснювати дії в гри-платформері. Агент взаємодіє з грою, спостерігає поточний стан і отримує нагороду залежно від того, що він робить, тобто отримує підкріплення своїх дій. Він навчається з кожним кроком і покращує свої стратегії, щоб максимізувати нагороду та досягати кращих результатів. Неоптимальні рішення в таких

алгоритмах не караються явно, а правильних схем поведінки не надається, тому агент повинен вчитися на своєму досвіді методом проб і помилок. Агент отримує оцінку про виконані ним дії та стан середовища після них, і намагається максимізувати винагороду.

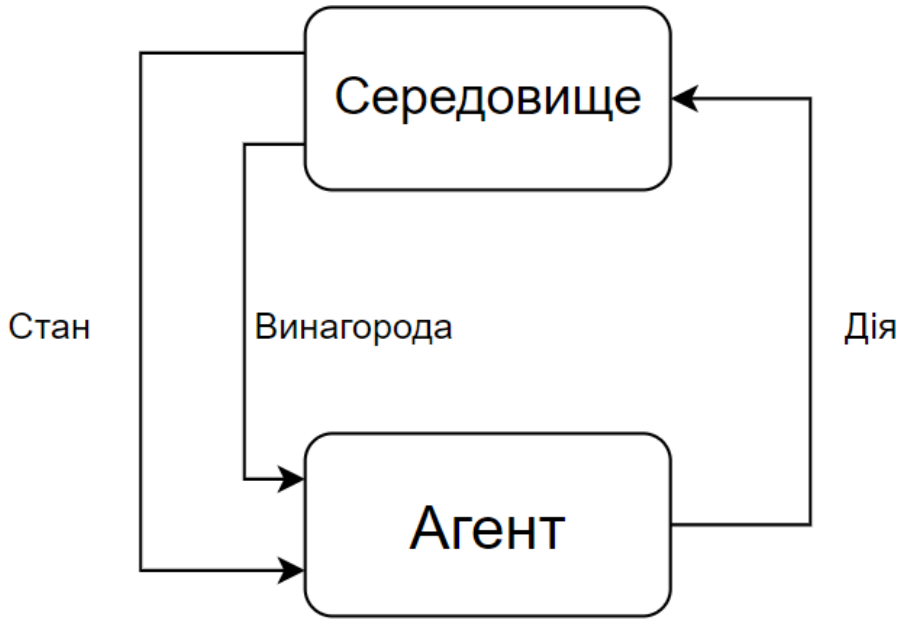

Рис. 6 – типова структура алгоритму навчання з підкріпленням. Агент взаємодіє з середовищем, виконуючи якусь дію, і після інтерпретації агенту повертаються стан середовища та винагорода

3.3 Основні підходи щодо пошуку стратегії

Є декілька підходів для пошуку оптимальної стратегії [20]:

- Підхід функції цінності: агент намагається знайти оптимальну стратегію для максимізації довгострокової нагороди, навіть якщо йому в короткостроковій перспективі доводиться робити дії, які дають меншу нагороду.
- Підхід на основі політики: агент намагається знайти оптимальну стратегію для отримання максимальної винагороди без використання функції цінності, тому кожною дією агент намагається максимізувати винагороду в короткостроковій перспективі.
- Модельний підхід: для дослідження агентом середовища створюється модель, представлення якої відрізняється для конкретного середовища.

Ці підходи можуть варіюватися в залежності від конкретного алгоритму навчання з підкріпленням та його параметрів. Вибір підходу до вибору стратегії залежить від характеристик задачі, вимог до розвідки середовища проти експлуатації найбільшої нагороди, а також від балансу між дослідженням та використанням вже відомих знань.

Навчання з підкріпленням використовується для розв'язання широкого спектру задач, де агент повинен вчитися взаємодіяти з динамічним середовищем і приймати оптимальні рішення для досягнення певної мети. Основна перевага навчання з підкріпленням над алгоритмом NEAT в тому, що нейромережі, які ці алгоритми створюють, є більшими. Тому навчання з підкріпленням добре підходить для проблем, де простір дій агента великий і складний, а середовище є мінливим.

РОЗДІЛ 4: Розробка гри

4.1 Розробка гри у Unity

Гра розроблялася у Unity. Було вирішено зробити гру-платформер з декількома рівнями. У цій грі агенти повинні вчитися проходити рівні за допомогою алгоритмів машинного навчання. Задача агента – дійти до фінішу рівня, оминувши всі пастки на рівні. Зазвичай для таких ігор використовують навчання з підкріпленням, однак я вирішив використовувати в своїй дипломній генетичний алгоритм, а саме алгоритм NEAT.

Так як мої знання Unity були обмеженими, було вирішив створити гру, користуючись посібником на Youtube [25]. Спочатку було створено блоки, з яких створюються ігрові зони. Для цього використовувались палітра плиток, а набір спрайтів для гри було взято з Unity Asset Store. Після цього було зроблено анімації для різних об’єктів, таких як гравець, колекційні предмети та різноманітні пастки.

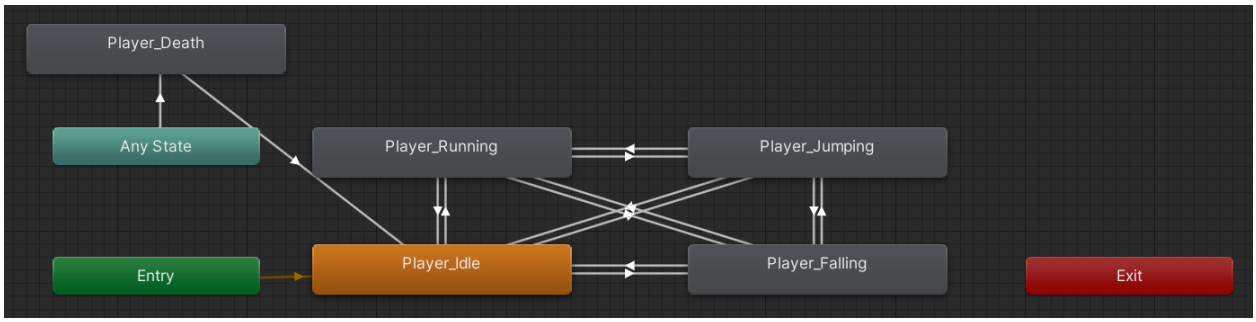

Рис. 7 – компонент Animator, який контролює анімації гравця та перехід між ними у грі. Переходи між анімаціями відбуваються через зміни параметра state

Після цього розроблялася логіка для гри. Для початку потрібно було зробити скрипт для контролювання гравцем. У цьому скрипті потрібно оброблювати вхідні дані, які даються гравцю, і переміщувати його чи виконувати стрибок, не забуваючи про зміну анімацій гравця. Також було вирішено, що гравець не може стрибати в повітрі, тому було прописано метод IsGrounded(), який перевіряє, чи торкається гравець землі. Змінна jumpableGround визначає шар, з якого наш гравець може стрибати.

```
1 reference
private bool IsGrounded()
{
    return Physics2D.BoxCast(coll.bounds.center, coll.bounds.size, 0f, Vector2.down, 0.2f, jumpableGround);
}
```

Також було написано скрипти про обробку смерті гравця та різноманітних звуків, перехід на наступний рівень, збирання колекційних предметів, переміщення пасток тощо.

Під час розробки гри траплялося багато багів, які потрібно було виправляти. Анімації гравця не змінювалися, після смерті гравця він не зникав, Unity не міг знайти прив’язаний до гравця звуковий ефект, гравець провалювався крізь текстури чи не міг стрибнути тощо. Деякі вади залишилися, наприклад, згаданий раніше метод IsGrounded() може повернути значення true, якщо гравець знаходиться дуже близько до стіни, через що можливий подвійний стрибок. Однак ця вада не сильно впливає на дослідження загалом, тому її було вирішено не усувати. Усі інші проблеми вимагали відлагодження, знаходження відповідних рішень та виправлення багів, щоб забезпечити правильну роботу гри.

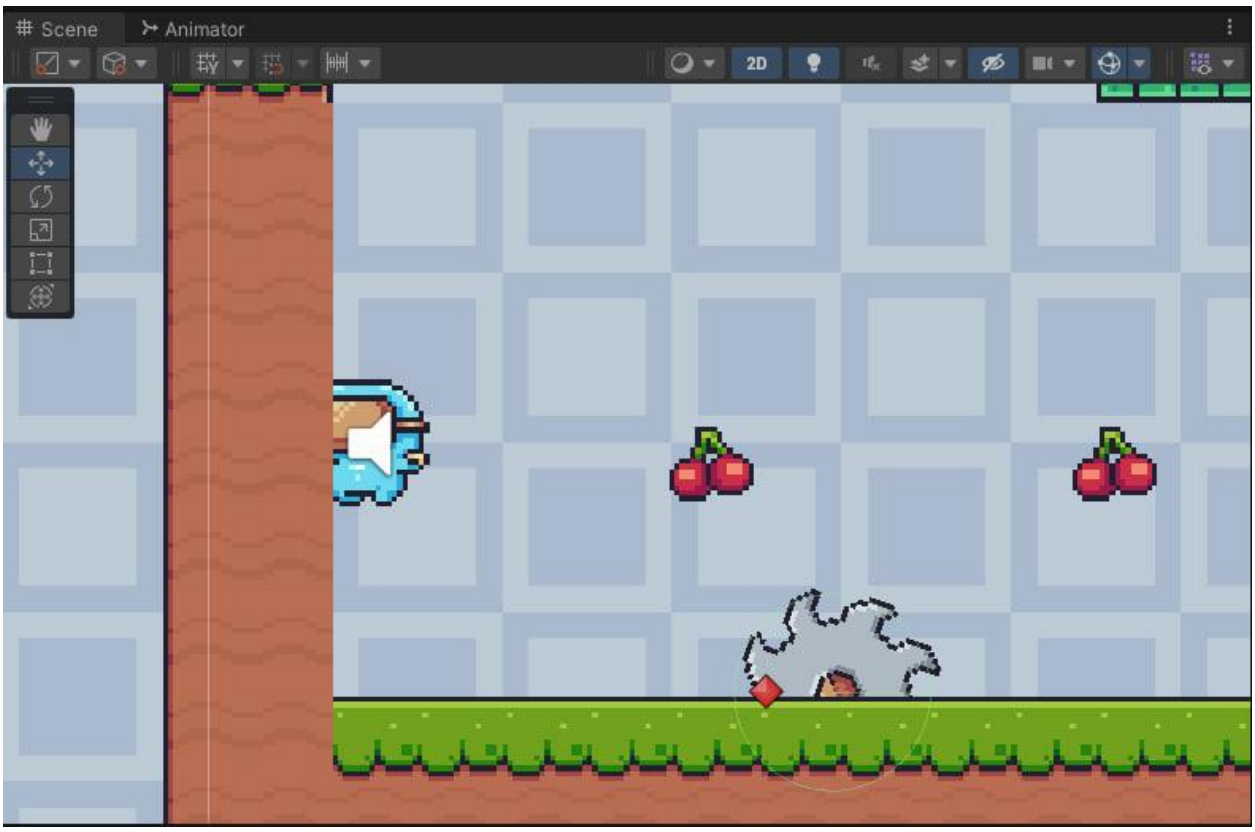

Рис. 8 — приклад багу. Агент "застрягає" в землі, якщо зажата кнопка переміщення. Цей баг виправляється додаванням до землі Platform Effector 2D, за допомогою якого можна усунути бокове тертя з гравцем

4.2 Імплементація алгоритму NEAT

Імплементація алгоритму NEAT була взята з Github [26]. Для коректної роботи потрібно було задати вхідні та вихідні дані, функцію допасованості. Задача агента — знайти оптимальну стратегію для того, щоб максимізувати оцінку функції допасованості.

По-перше, треба було задати вхідні вузли, в які йдуть дані про те, що наш гравець може побачити. Для гри-платформера було вибрано загалом дев’ять вхідних даних, а саме - x та y позиції агента, чи торкається агент землі, x та y позиції найближчої пили відносно агента, x та y позиції платформи, що рухається, відносно агента, а також два сенсори, які направлені прямо вниз і під кутом 45% донизу. Сенсори повертають відстань до об’єктів, які мають тег "Terrain" чи "MPlatform" (платформа, що рухається) і допомагають агентам зрозуміти, наскільки вони близько до краю, а отже, повинні стрибати. Для того, щоб знайти позицію найближчої пили, ми на початку виконання програми створюємо масив, який містить усі ігрові об’єкти, в яких є тег "Saw", а в самому методі під час виконання програми шукаємо пилу з найменшою відстанню до теперішньої позиції гравця:

```
2 references
public GameObject FindClosestSaw()
{
    GameObject[] gos;
    gos = saws;
    GameObject closest = null;
    float distance = Mathf.Infinity;
    Vector3 position = transform.position;
    foreach (GameObject go in gos)
    {
        Vector3 diff = go.transform.position - position;
        float curDistance = diff.sqrMagnitude;
        if (curDistance < distance)
        {
            closest = go;
            distance = curDistance;
        }
    }
    return closest;
}
```

Однак через те, що платформа рухається на за допомогою вбудованих функцій Rigidbody, а за допомогою скрипта, який просто переміщує її між двома точками, відносну швидкість цієї платформи щодо гравця не надається у вхідні дані.

Вихідних вузлів всього два - рух і стрибок. Гравець може стрибати, якщо на вихід вузол стрибка дає число, більше за 0, а сам гравець торкається землі. Рух теж простий – якщо на вихід приходить число більше за 0, гравець рухається вправо, якщо менше – вліво. Висота стрибка є фіксованою, а дальність переміщення гравця визначається як множення числа, яке приходить на вихід, на швидкість руху гравця, яка теж є фіксованою.

Функція допасованості визначає, наскільки добре агенти проходять рівень. Спочатку були спроби зробити так, щоб функція допасованості давала тим більше число, чим ближче агент до фінішу, однак через неможливість задати це лінійно, було вирішено відмовитися від цієї ідеї. Оцінка в такому разі за відстань зростала б експоненційно, через що агенти дуже довго вчилися б на початку, оскільки незначні зміни у функції допасованості несильно впливають на розвиток нейромережі. Оскільки фініш у трьох рівнях гри завжди є найдальшою

точкою від початкової позиції гравця, то в моїй програмі функція допасованості дає тим більшу оцінку, чим далі агент знаходиться від початкової позиції.

Оскільки гра початково не робилася для машинного навчання, а для проходження її людьми, виникали проблеми з уже зробленою реалізацією. Наприклад, через те, що в грі не задумувалось одночасне її проходження декількома гравцями, виникли баги під час імплементації алгоритму NEAT – агенти провалювались крізь текстури, тому що виштовхували одне одного і не могли рухатися.

Рис. 9 — агенти виштовхують одне одного, оскільки мають колайдер для обрахування зіткнень з іншими ігровими об’єктами. Згодом був знайдений спосіб вирішення цієї проблеми – у матриці визначення зіткнень з урахуванням шарів було відключено зіткнення гравців з іншими гравцями

Також через певні складнощі з імплементацією NEAT у грі було вирішено відмовитися від колекційних предметів через невеликий вплив на навчання агентів та переходу на наступний рівень, оскільки виникали проблеми з вхідними вузлами для різних рівнів. Також різноманітні звуки було вимкнено через непотрібність для тренування нейромережі.

РОЗДІЛ 5: Результати

5.1 Результати для першого рівня

Агенти вчилися проходити 3 рівні у грі-платформері. На першому рівні треба було перестрибнути провалля та пастку, щоб досягнути фінішу. Для другого треба було застрибнути на платформу, що рухається, і почекати, доки вона достатньо підніметься і досягнути фінішу. 3 рівень був певною комбінацією двох попередніх: агенти повинні оминути пастки та застрибнути на платформу, щоб досягнути фінішу.

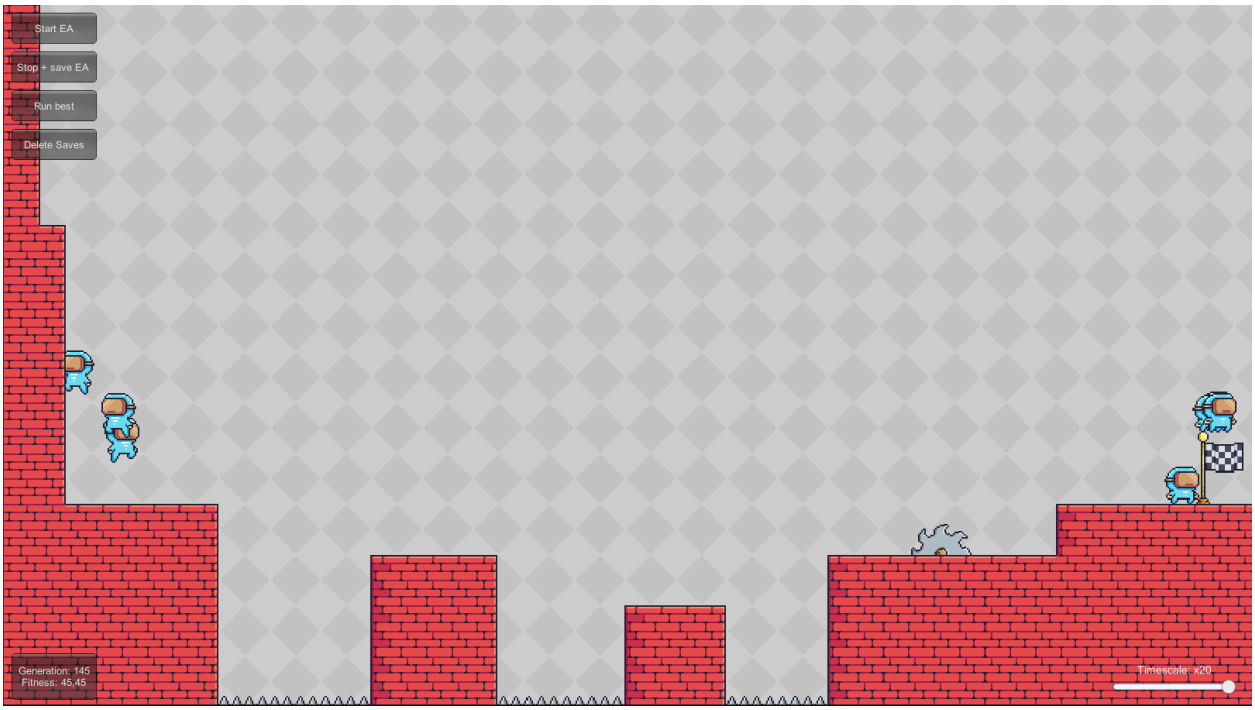

Рис. 10 — перший рівень гри-платформера. Як можна побачити, декілька агентів стрибають на місці, оскільки алгоритм NEAT групує особини і експериментує зі стратегіями

Їм приходить 7 вхідних даних – x та y позиції агента, чи торкається агент землі, x та y позиції найближчої пили відносно агента, а також два сенсори, які направлені прямо вниз і під кутом 45% донизу. На основі цих даних агенти вчаться проходити рівень. Спочатку вони виконують випадкові дії, і функція допасованості ці дії оцінює. Через 80 генерацій агенти вперше перестрибують на наступну платформу. На 83 генерації агенти досягають другої платформи, на 92 – третьої. На 133 один з агентів вперше досягає фінішу, і до 145 генерації більшість агентів навчилася проходити рівень.

5.2 Результати для другого рівня

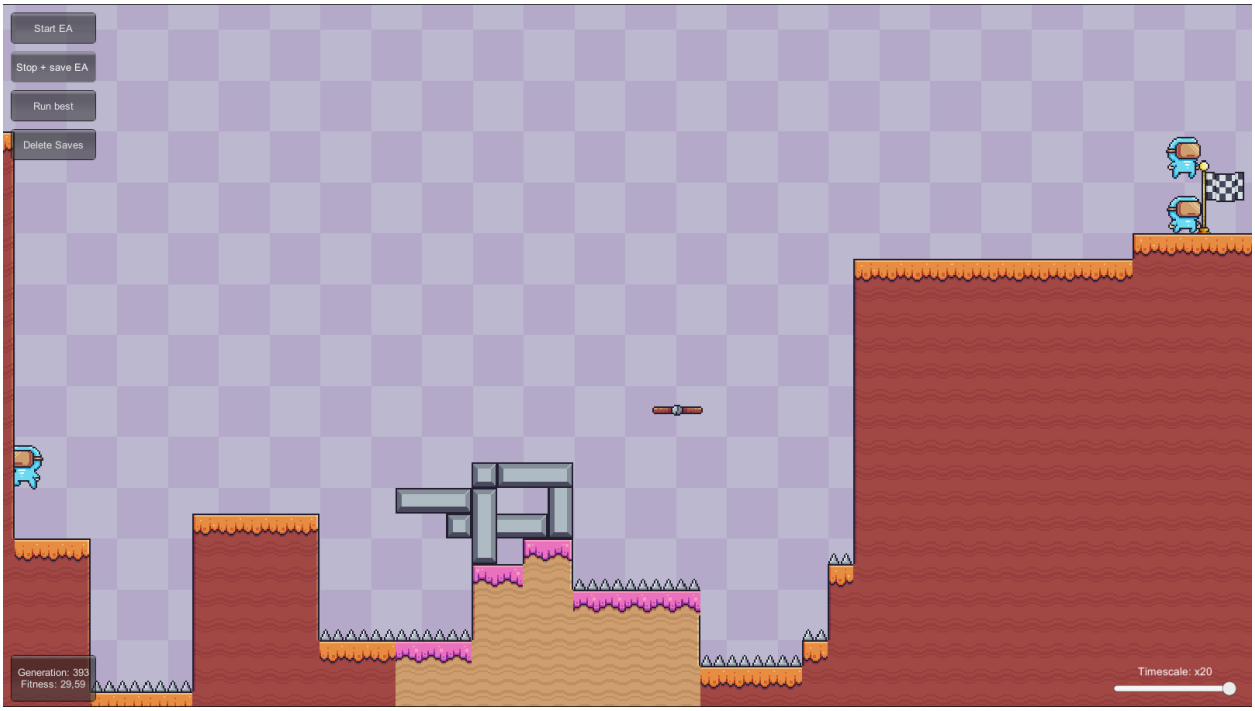

Рис. 11 — другий рівень гри-платформера. Через недостатньо добре задані вхідні вузли навчання займає велику кількість генерацій

На другому рівні агентові тепер треба навчитися застрибувати на платформу, яка рухається, щоб досягнути фінішу. Агенти через випадкові мутації вже на 5 генерації доходять до місця, звідки можна застрибувати на платформу. Однак через те, що їм не приходить швидкість руху платформи, навчання агентів займає доволі довго — тільки на 206 генерації одному з них вдається досягнути фінішу, тоді як решта випробовують стратегії, які в результаті не є ефективними, наприклад, стрибання на місці. До 393 генерації популяція розвинулась достатньо, щоби більшість з них доходили до фінішу.

5.3 Результати для третього рівня

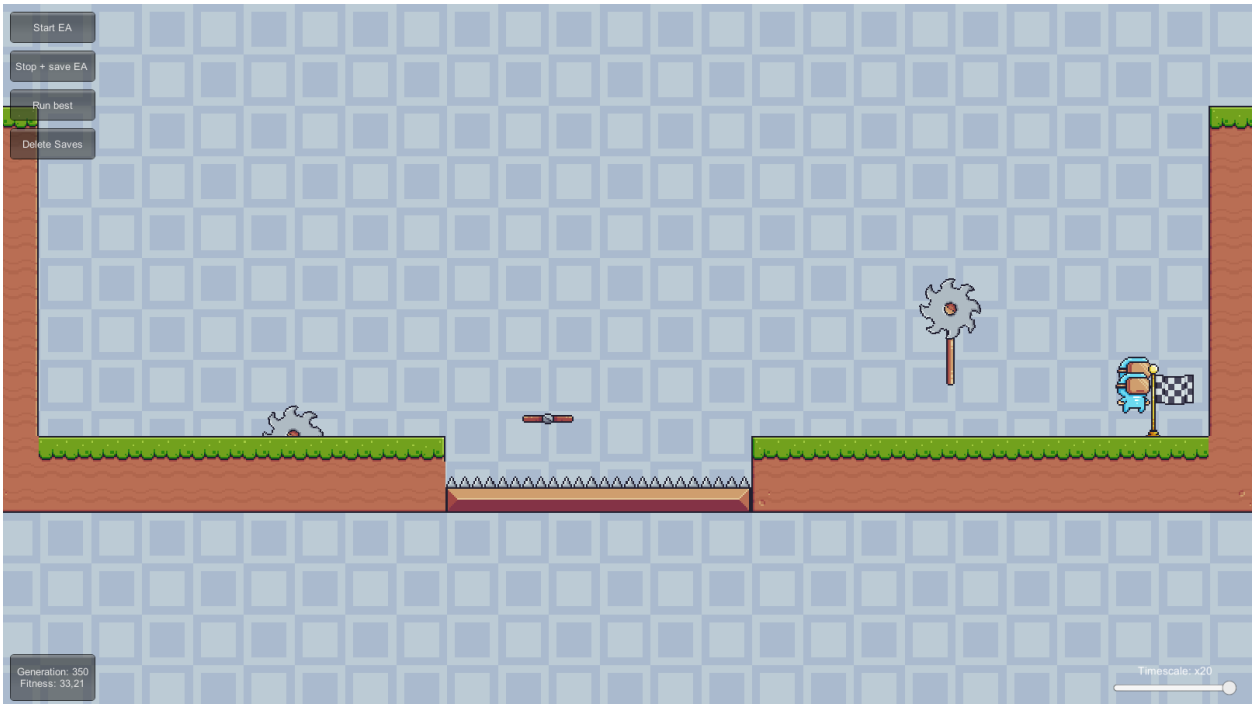

Рис. 12 — третій рівень гри-платформера, для проходження якого агентам треба і оминути пастки, і застрибнути на платформу

На 3 рівні присутні і пили, і платформа, тому на вхід агентам тепер дається 9 параметрів. Агенти до 16 генерації навчилися перестрибувати пастку, а застрибнути на платформу і досягнути фінішу одному з них вдалося на 37 генерації. Однак це була більшою мірою випадковість, тому що на наступній генерації ніхто з них не досягнув фінішу. До 350 генерації нейромережі розвинулися достатньо, щоби більшій частині популяції вдалося пройти рівень.

5.4 Висновки

	Оминуто першу перешкоду	Пройдено половину рівня	Вперше досягнено фініш	Більшість досягає фінішу
1 рівень	80 ген.	83 ген.	133 ген.	145 ген.
2 рівень	5 ген.	5 ген.	206 ген.	393 ген.
3 рівень	16 ген.	34 ген.	37 ген.	350 ген.

Велика кількість генерацій останніх двох рівнів може говорити про те, наскільки важливим для алгоритму NEAT є підбір правильних параметрів. Наприклад, агентам не приходить швидкість переміщення платформи, а тільки її

позиція, тому це може пояснити недостатньо високу продуктивність нейромережі. Разом з тим, NEAT не може створювати великих нейронних мереж, тому велика кількість вхідних даних призведе до того, що еволюція буде дуже повільною або ж нейромережа не зможе стати достатньо ефективною для вирішення задачі. В алгоритмах навчання з підкріпленням, наприклад Q-навчання, цих проблем немає, тому що на вхід можна задати значно більшу кількість параметрів, а отже, точний вибір оптимальних параметрів не так необхідний, однак у такому підході також присутні й недоліки, наприклад, необхідність у більшій кількості обчислювальних ресурсів та можливість перенавчання. Якщо брати загалом, вибір між алгоритмами залежить від конкретної задачі, обмежень та вимог щодо складності стратегій, швидкості навчання та обчислювальних ресурсів.

ВИСНОВКИ

Кваліфікаційна робота присвячена розробленню гри у рушії Unity та дослідженню ефективності генетичного алгоритму NEAT при використанні в грі-платформері для навчання агентів проходити рівні.

Було розроблено гру-платформер, в якій гравець повинен був оминати різноманітні пастки, щоб дійти до фінішу. Також було імплементовано алгоритм NEAT та оцінено його ефективність для ігрових середовищ такого типу.

Як можна побачити, NEAT є потужним інструментом для вирішення різноманітних задач. Цей алгоритм є серйозним покращенням, порівняно з іншими TWEANN, а структура, яка збільшується під час навчання агентів, дозволяє підвищити продуктивність і не витрачати час та ресурси на оптимізацію неефективних або непрацездатних мереж. NEAT є хорошою альтернативою навчанню з підкріпленням для простих задач, в яких не потрібно великого комплексу рішень, і зазвичай досягає нейромереж, які можуть ефективно знаходити рішення, швидше за інші схожі методи. Однак головним обмеженням NEAT залишається нездатність створювати великі нейронні мережі. У випадку комплексних задач, особливо в ігрових середовищах, де потрібні складні стратегії, NEAT не є оптимальним вибором.

У таких ситуаціях рекомендується використовувати навчання з підкріпленням, яке дозволяє використовувати більш гнучкі алгоритми та створювати складні та потужні нейромережі. Також можна розглядати поєднання NEAT з іншими методами, що дозволить поєднати переваги обох підходів і досягти кращої продуктивності в складних завданнях. Загалом, вибір між NEAT і навчанням з підкріпленням залежить від конкретної задачі і вимог щодо складності стратегій та швидкості знаходження рішень.

Список використаної літератури

1. McCulloch W., Pitts W. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*. 1943. Т. 5, № 4. С. 115-133.
   URL: https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf.
2. Samuel A. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*. 1959. Т. 3, № 3. С. 210–229. URL: http://people.csail.mit.edu/brooks/idocs/Samuel.pdf.
3. Hovsepyan T. Machine Learning in Gaming. *Plat.AI*. [Електронний ресурс] URL: https://plat.ai/blog/machine-learning-in-gaming/.
4. Gaming is booming and is expected to keep growing. *The World Economic Forum*. [Електронний ресурс]
   URL: https://www.weforum.org/agenda/2022/07/gaming-pandemic-lockdowns-pwc-growth/
5. What is evolutionary algorithm. *WhatIs*. [Електронний ресурс]
   URL: https://www.techtarget.com/whatis/definition/evolutionary-algorithm.
6. What is reinforcement learning. *GeeksforGeeks*. [Електронний ресурс]
   URL: https://www.geeksforgeeks.org/what-is-reinforcement-learning/.
7. Katoch S., Chauhan S., Kumar V. A review on genetic algorithm: past, present, and future. *Multimed Tools Appl*. 2021. Т. 80, № 5. С. 8091–8126.
   URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7599983/.
8. Bäck T. Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms. Нью-Йорк : Oxford University Press, 1996. 314 с.
   URL:
   https://books.google.com.ua/books?id=htJHI1UrL7IC&lpg=PR9&ots=fBt-ZRWBkS&dq=Evolutionary%20algorithms%20in%20theory%20and%20practice%20-%20evolution%20strategies%2C%20evolutionary%20programming%2C%20genetic%20algorithms.&lr&pg=PP1#v=onepage&q=Evolutionary%20a

lgorithms%20in%20theory%20and%20practice%20-%20evolution%20strategies,%20evolutionary%20programming,%20genetic%20algorithms.&f=false

9. Jebari K. Selection Methods for Genetic Algorithms. *International Journal of Emerging Sciences*. 2013. T. 3. C. 333–344.
URL: https://www.researchgate.net/publication/259461147_Selection_Methods_for_Genetic_Algorithms.
10. Whitley D. The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. 2000.
URL: https://www.researchgate.net/publication/2527551_The_GENITOR_Algorithm_and_Selection_Pressure_Why_Rank-Based_Allocation_of_Reproductive_Trials_is_Best..
11. Miller B., Goldberg D. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Syst.* 1995. T. 9.
URL: https://wpmedia.wolfram.com/uploads/sites/13/2018/02/09-3-2.pdf.
12. Stanley, Stanley K., Miikkulainen R. Efficient evolution of neural network topologies. *Proceedings of the 2002 Congress on Evolutionary Computation (CEC'02)*. 2002. T. 2. C. 1757–1762.
URL: https://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf.
13. Introduction to Genetic Algorithms.
URL: https://web.archive.org/web/20150811025830/http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html.
14. Syswerda G. Uniform crossover in genetic algorithms. *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA)*. 1989. C. 2–9.
URL: https://www.researchgate.net/publication/201976488_Uniform_Crossover_in_Genetic_Algorithms.
15. Stanley K. Efficient Evolution of Neural Networks Through Complexification. Austin, 2004. 180 c.
URL: https://nn.cs.utexas.edu/downloads/papers/stanley.phd04.pdf.
16. Goldberg D., Richardson J. Genetic Algorithms with Sharing for Multimodal Function Optimization. *Proceedings of the Second International Conference*

on Genetic Algorithms on Genetic Algorithms and Their Application. 1987. С. 41–49.
URL: https://books.google.com.ua/books?id=MYJ_AAAAQBAJ&lpg=PA41&ots=XxuJsq6EIw&lr&hl=uk&pg=PA41#v=onepage&q&f=false.

17. Angeline P., Saunders G., Pollack J. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*. 1994. Т. 5. С. 54–65.
URL: http://www.demo.cs.brandeis.edu/papers/ieeenn.pdf.
18. Yao X. Evolving artificial neural networks. *Proceedings of the IEEE*. 1999. Т. 87, No 9. С. 1423–1447.
URL: https://www.cs.bham.ac.uk/~xin/papers/published_iproc_sep99.pdf.
19. Hertz J., Krough A., Palmer R. Introduction To The Theory Of Neural Computation. *Physics Today*. 1991. Т. 44.
URL: https://nessie.ilab.sztaki.hu/~kornai/2020/AdvancedMachineLearning/Hertz_Krogh_Palmer__IntroToNeuralComp.pdf.
20. Reinforcement Learning Tutorial. Javatpoint. [Електронний ресурс]
URL: https://www.javatpoint.com/reinforcement-learning.
21. Генетичний алгоритм. Вікіпедія. [Електронний ресурс]
URL: https://uk.wikipedia.org/wiki/Генетичний_алгоритм.
22. Baker J. Adaptive Selection Methods for Genetic Algorithms. *International Conference on Genetic Algorithms*. 1985.
URL: https://doi.org/10.4324/9781315799674.
23. Genetic algorithms. *GeeksforGeeks*. [Електронний ресурс]
URL: https://www.geeksforgeeks.org/genetic-algorithms/.
24. NEAT: An Awesome Approach to NeuroEvolution. *Medium*. [Електронний ресурс] URL: https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f

25. Build a 2D Platformer Game in Unity. *YouTube*. [Електронний ресурс] URL: https://www.youtube.com/playlist?list=PLrnPJCHvNZuCVTz6lvhR81nnaf1a-b67U

26. UnitySharpNEAT. *Github*. [Електронний ресурс] URL: https://github.com/flo-wolf/UnitySharpNEAT