МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»

Кафедра інженерії програмного забезпечення

КУРСОВИЙ ПРОЕКТ (РОБОТА)

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

з дисципліни: «Мови інтелектуального аналізу даних»

на тему:

«Аналіз ігрового штучного інтелекту та підходів до його розробки Застосунок-гра, що демонструє застосування ігрового штучного інтелекту»

студента І курсу групи ІПЗм-21-2
спеціальності 121 «Інженерія програмного забезпечення»
Ліщинського Олександра Анатолійовича
(прізвище, ім'я та по-батькові)

Керівник ст. викл. каф. КН Марчук Г. В.

Дата захисту: "____"_____20__р.
Національна шкала_____
Кількість балів:_____
Оцінка: ECTS _____

Члени комісії _____ Морозов А.В.
(підпис) *(прізвище та ініціали)*

_____ Марчук Г.В.
(підпис) *(прізвище та ініціали)*

_____ Кузьменко О.В.
(підпис) *(прізвище та ініціали)*

Житомир – 2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
Факультет інформаційно-комп'ютерних технологій
Кафедра інженерії програмного забезпечення
Освітній рівень: магістр
Спеціальність 121 «Інженерія програмного забезпечення»

«ЗАТВЕРДЖУЮ»
Зав. кафедри _____
_____
"__"_____20___р.

ЗАВДАННЯ
НА КУРСОВИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТУ
Ліщинському Олександру Анатолійовичу

1. Тема роботи: Аналіз ігрового штучного інтелекту та підходів до його розробки Застосунок-гра, що демонструє застосування ігрового штучного інтелекту _____,
керівник роботи: _____
2. Строк подання студентом: "___"_____20__р.
3. Вхідні дані до роботи: _____

4. Зміст розрахунково-пояснювальної записки(перелік питань, які підлягають розробці):
- _____
- _____
- _____

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

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посади консульта-нта	Підпис, дата	
		завдання видав	завдання прийняв
1	Марчук Г. В., ст. викладач каф. КН		
2	Марчук Г. В., ст. викладач каф. КН		

7. Дата видачі завдання "____"_____20___р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів курсового проекту (роботи)	Строк виконання етапів роботи	Примітки
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Метою даної курсової роботи є вивчення ігрового штучного інтелекту та підходів до його побудови.

В роботі описаний ігровий штучний інтелект як поняття та вимоги до ігрового штучного інтелекту. Проаналізовані підходи до його побудови, їхні особливості та визначені випадки, коли ці підходи найкраще застосовувати. Також наданий опис реалізації гри, що демонструє застосування штучного інтелекту.

Зміст

ВСТУП

Актуальність теми. Індустрія ігор – одна з найбільших в сфері розваг сьогодні. Розробникам необхідно потурбуватися про те, щоб користувачі отримали якнайкращий досвід від використання продукту та продовжували використовувати його. Для цього необхідно додати грі реалістичності, створити враження присутності «живих супротивників», створити відповідний рівень складності для гравця, створюючи випробування та забезпечуючи можливість виграшу для нього. Ігровий штучний інтелект допомагає досягти всіх цих цілей,а також збільшує рівень абстракції, що дозволяє розробникам лише визначати правила гри, а вже сам алгоритм вирішує, яке рішення прийняти.

Мета дослідження – вивчити поняття ігрового штучного інтелекту та проаналізувати підходи до його побудови.

Для досягнення мети необхідно виконаки наступні етапи:

- Вивчення поняття ігрового штучного інтелекту
- Вивчення підходів до його побудови
- Аналіз та визначення особливостей кожного з підходів
- Реалізація гри, що складається з кількох модулів, кожен з яких демонструє певний підхід до побудови ігрового штучного інтелекту. Основним завданням цього етапу є приклад реалізації цих підходів, а також демонстрація їхньої роботи.

Об’єкт дослідження – ігровий штучний інтелект.

Предмет дослідження – підходи до побудови ігрового штучного інтелекту.

Розділ 1: ознайомлення з індустрією відеоігор та їх структурою.

Роділ 2: пояснення поняття штучного інтелекту та аналіз його особливостей при застосуванні у іграх.

Розділ 3: детальні характеристики та особливості реалізації різних підходів до побудови ігрового штучного інтелекту а також визначення задач, для яких кожен

підхід є рекомендованим.

Розділ 4: опис власних прикладів реалізації штучного інтелекту. В даному розділі застосовуються підходи, що були проаналізовані в попередньому розділі, аргументується вибір конкретного алгоритму, а також демонструєтьсярезультат виконання гри.

1. РОЗРОБКА ІГОР

1.1. Індустрія відео ігор

Відео ігри – це одна з найпопулярніших індустрій в сфері розваг. Щодня все більше людей отримує доступ до цифрових технологій, а це означає, що вони швидше за все приєднаються до общини так званих геймерів – людей, що грають відео ігри. Ще у 2014 році кількість геймерів становила приблизно 1,8 мрлд людей. У 2020 це значення збільшилося майже в півтора раза – 2,6 млрд людей. Статистична компанія Statista у своєму дослідження оцінила приріст нанаступний рік у 125 мільйонів нових гравців.

Не дивно, що ігрова індустрія з такою кількістю користувачів є надзвичайно прибутковою. За 2019 рік прибуток індустрії склав понад $120 млрд.

Така статистика звучить доволі привабливо, здається, що в цій сфері легко бути успішним. Але варто взяти до уваги, що на ринку велика конкуренція, тому для того, щоб виділитися, необхідно зробити так, щоб користувач насолоджувався грою і в майбутньому порекомендував її своїм друзям. Для цього необхідно зробити гру якісною і цікавою на кожному рівні розробки.

1.2. Структура відео ігор

Рис. 1 Структура відео ігор

При створенні гри можна виділити 3 основні складові компоненти в її структурі: сценарій, дизайн та логіка.

Сценарій включає в себе головну ідею гри, її ціль, завдання, та для більшості видів ігор, наприклад, пригодницьких, стратегій, певну передісторію та сюжет. Дизайн значною мірою відповідає за естетичне сприйняття гри користувачами. Він поєднує в собі багато важливих складових продукту. Зручність і гармонійність відтінків інтерфейсу користувача, а також аудіосупровід, пошук або створення моделей для агентів та сцен/локацій – все це робота дизайнерів проекту [1].

Проте кістяком проекту можна вважати його логіку – програмну частину. Вона відповідає за поведінку агентів, генерацію об’єктів, взаємодію користувача зі світом гри. Є багато способів реалізувати цю логіку. Кожен спосіб пропонує певний рівень гнучності та підлаштування гри під користувача.

Досить часто розробники роблять алгоритм поведінки сталим. В ньому чітко визначена послідовність дій і на неї ніяким чином не впливає користувач. Це типова практика для аркад та платформерів. Наприклад, в Brick Breaker цеглини генеруються в послідовності та рухаються зі швидкістю, визначеними рівнем гри.

Проте деякі ігри, особливо, якщо в них присутні ворожі агенти, активно реагують на дії користувача. Ці агенти вміють підлаштовуватися під рівень вмінь гравця, створювати відчуття присутності інших гравців та виводять гру на новий рівень персоналізації [1].

Незалежно від рівня взаємодії гри з користувачем, сьогодні важко знайти гру, яка б не містила алгоритмів штучного інтелекту.

2. ІГРОВИЙ ШТУЧНИЙ ІНТЕЛЕКТ

2.1. Поняття штучного інтелекту

Штучний інтелект – це розділ комп'ютерних наук, що зосереджений на створенні пристроїв, що могли б виконувати завдання, які зазвичай потребують людського втручання [1].

Алан Тюринг у своїй статті запитав «Чи можуть машини думати?»[2]. І можна вважати, що розвиток штучного інтелекту прагне до того, щоб дати позитивну відповідь на це запитання. Він розробив тест, щоб перевіряти інтелект машин. Його суть полягає в тому, що в одній кімнаті знаходиться комп'ютер і людина, а в іншій допитувач. Допитувач повинен ставити запитання до тих, хто знаходиться в іншій кімнаті і за відповідями визначити хто є хто. Ціль комп'ютера – переконати допитувача, що він людина.

В основі штучного інтелекту може лежати машинне або глибинне навчання,а деякі алгоритми просто слідують набору визначених правил.

Штучний інтелект можна поділити на 2 категорії:

А) Вузьконаправлений штучний інтелект. Він використовується в застосунках і пристроях, які націлені на вирішення проблем в вузькому спектрі. Прикладами вузьконаправленого інтелекту є пошукові системи, персональні помічники на голосовому управлінні, автопілот.

Б) Загальний штучний інтелект. Його завданням є повна симуляція людського мислення і поведінки. Це поняття не є новим, але складність цієї задачі не змінилася і ідея створити робота, який буде зможе підлаштовуватися під будь-яке середовище і мати повний набір пізнавальних здібностей, все ще залишається складним завданням. Попри складнощі, створення такого робота, який би міг демонструвати інтелект на рівні людини, - це ключова ціль для більшості дослідників і розробників штучного інтелекту[1]

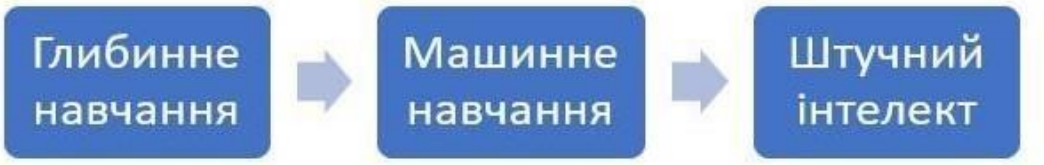


Рисунок 1. 1 Зв'язок між глибинним навчанням, машинним навчанням та штучним інтелектом

Штучний інтелект це набір алгоритмів, що симулюють людське мислення або поведінку.

Машинне навчання – це один з підходів до побудови штучного інтелекту, який полягає в тому, щоб дати комп'ютеру великий об'єм інформації та алгоритми для її опрацювання. Використовуються 2 види датасетів: мічені та немічені. Тренування відбувається доти, доки результати, що продукує сам комп'ютер, не стануть достатньо точними.

Особливість глибинного навчання полягає в тому, що вибірка даних опрацьовується нейромережами, що що за архітектурою нагадують біологічні. Вони містять кілька прихованих рівнів, на яких обробляється надана вибірка, утворюються нові зв’язки, та формуються результати, які в подальшому підвищують точність аналізу даних[1].

2.2. Особливості ігрового штучного інтелекту

Штучний інтелект, який застосовується в іграх, значно відрізняється від того, який розробляють для використання в реальному житті, наприклад, для системи автопілоту чи дослідницьких лабораторій.

До ігрового штучного інтелекту є цілий ряд вимог, невиконання яких може призвести до повного незадоволення зі сторони користувача.

а) Ігровий штучний інтелект повинен економно використовувати ресурси. Гра відбувається в реальному часі, тому рішення повинно прийматися за долю

секунди, щоб не виникало затримок. До того ж, більшість використовує мобільні пристрої та персональні комп'ютери. На них одночасно відбуваються багато процесів, тому монополізація процесора нашим штучним інтелектом є недопустима.

б) Головна ціль гри – розважити користувача. Для цього треба змушувати працювати його логіку та реакцію, одночасно не дозволяючи йому думати, що гра надто легка/важка, що призведе до втрати інтересу. Штучний інтелект в цьому випадку не повинен вибирати найоптимальніший варіант, а швидше такий, при якому у користувача буде можливість виграти. Так як наш агент розташований в середовищі, створеному повністю нами, то йому може бути відомо про розташування всіх об'єктів і їхні характеристики, проте тоді в гравця не буде жодних шансів, а це суперечить меті гри.

в) Поведінка агентів має бути реалістична. Якщо користувач грає в гру, де він знаходиться на території, що охороняють агенти, то в нього повинне скластися враження, що проти нього грають інші люди. Тобто агенти наділені тими ж можливостями, що й гравець (наприклад, бачити лише те, що розташоване в полі зору і не далі певної відстані, проходити лише через двері і т д).

г) Рішення, що приймаються штучним інтелектом повинні базуватися на даних, які визначені під час розробки. Так як гра потрапляє до користувачів тільки тоді, коли вона повністю готова до використання, то можливості попереднього тренування алгоритмів немає[7].

3. ПІДХОДИ ДО ПОБУДОВИ ШТУЧНОГО ІНТЕЛЕКТУ

3.1. Practical Path Finding

3.1.1 Задача пошуку шляху

Одне з найважливіших завдань, які покладають на ігровий штучний інтелект – це пошук шляху. Ця проблема зустрічається в різних жанрах ігор: пазли, симулятори, стратегії і т. д. Переміщення агентів в грі, підказки в «П'ятнашках», пошук шляху в лабіринтах чи нарахування очок гравцю за певнудію – приклади таких проблем.

Пошук шляху – це задача, що вирішується в теорії графів. На ігрове поле накладають граф з вершинами на точках, де може розташовуватися об'єкт. Ребра графу з'єднують вершини-клітинки, між якими об'єкт може безпосередньо переміщуватися. Деколи ребрам надаються вагові коефіцієнти, які беруться до уваги при пошуку найкращого шляху. Ці коефіцієнти можуть залежати від відстані між вершинами, перешкод на шляху або будь-яких інших умов, які накладаються на гру[4].

Важливо вибрати стратегію розміщення вершин графу таким чином, щоб пошук шляху був ефективним.

Для ігор, в яких ігрове поле складається з квадратів чи інших фігур, вершини зазвичай розмішують в точках, що знаходяться по центру кожної клітинки. Проте цей метод ефективний для невеликої кількості клітинок – якщополе має розмір 50x50, то кількість вершин буде становити 2,500, а кількість ребер ще більше(залежно від правил переміщення). При таких великих значеннях пошук шляху стає ресурсо-затратним завданням[4].

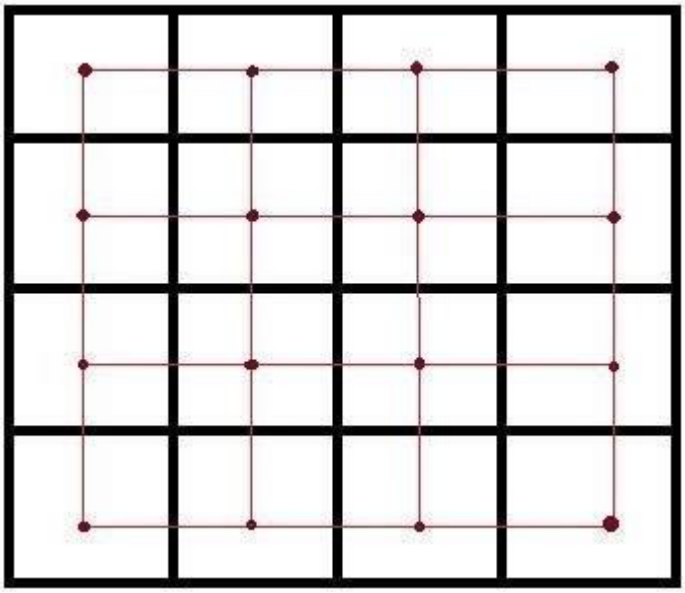


Рисунок 3. 1 Викляд графу на ігровому полі з вершинами кожній клітинці

На рисунку 3.1 білі прямокутники – це клітинки ігрового поля, бордові точки – точки всередині клітинок, червоні відрізки – ребра, що сполучають 2 сусідні вершини.

Наступний спосіб вибору точок – граф навігації за точками видимості. На локації вибираються найважливіші точки так, щоб кожна з них була в області видимості як мінімум ще одної точки.

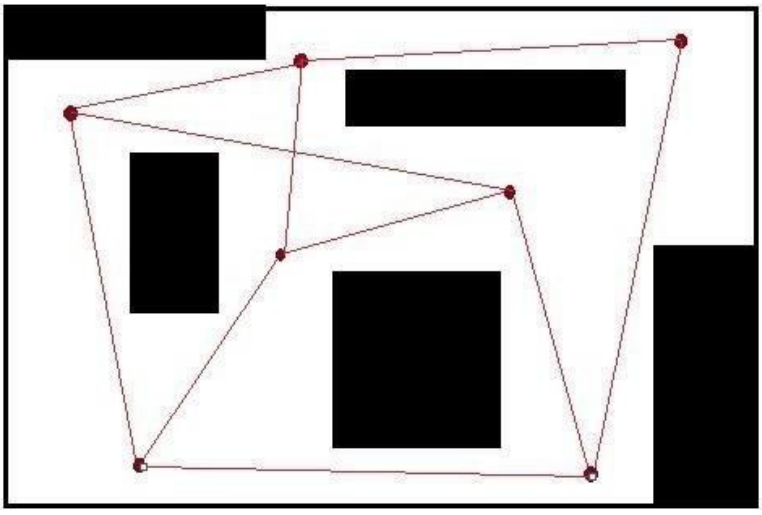


Рисунок 3. 2 Побудова графа навігації за точками видимості

На рисунку 3.2 Чорні прямокутники – перешкоди, що обмежуютьвидимість, бордові точки – точки видимості, червоні відрізки – шляхи між точками видимості. У цьому випадку, як і деколи це роблять розробники, точкибули вибрані вручну. Проте у випадках, коли локації великі і перешкод надто багато – вибір точок вручну займає багато часу, а також допускає неточності. Для цього використовують алгоритм розширеної геометрії – навколо кожного об'єкта-перешкоди створюють полігони, вершини яких відступають від вершихоб'єктів на відстань, що дорівнює радіусу описаного кола навколо агентів, що будуть цими шляхами рухатися. Тоді відбувається автоматичний процес додавання ребер до графу, в результаті чого отримуємо завершений граф [5].

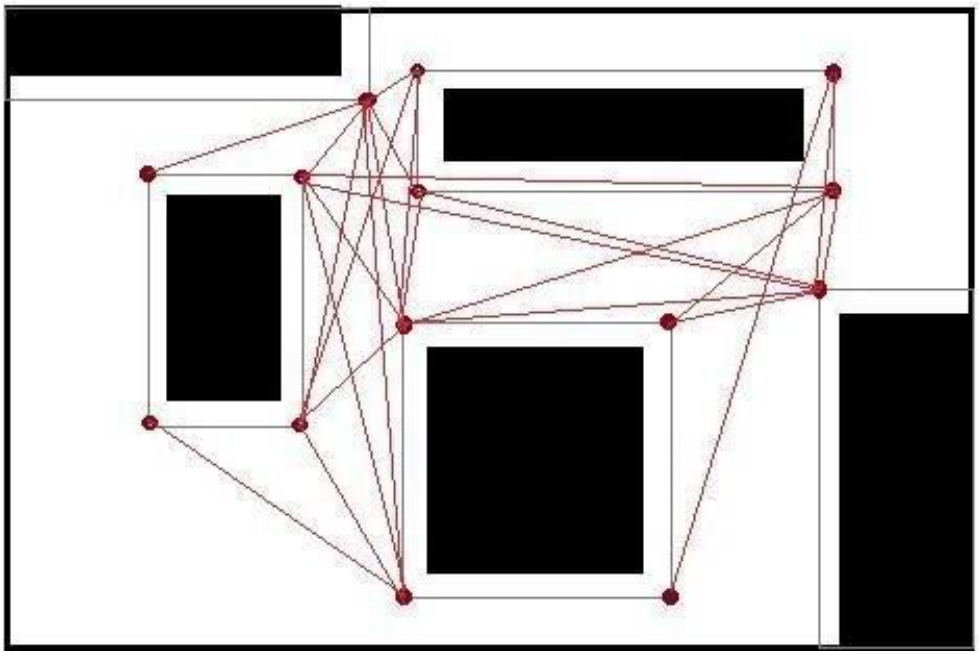


Рисунок 3. 3 Граф, створений за допомогою розширеної геометрії та точок видимості

На рисунку 3.3 чорні прямокутники – перешкоди, сірі контури – розширена геометрія (їх сторони теж є ребрами графу), бордові точки – вершини видимості, а червоні відрізки – ребра графу.

Ще один підхід, який тепер набуває популярності використовує Navmesh(navigation mesh – укр. сітка навігації). Він надає значно більше можливостей, так як не обмежує переміщення агентів лише визначеними точками.

Navmesh покриває всі частини локації, що доступні для переміщення. Таким чином доступні не лише певні точки, а повністю вся площина.

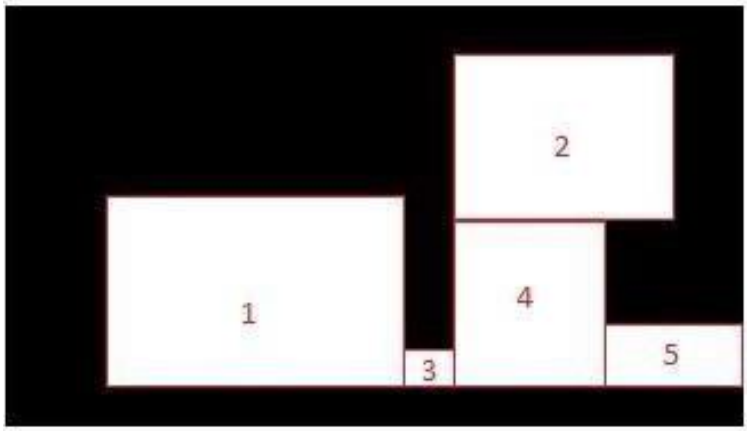


Рисунок 3. 4 Граф навігації з полігонами в якості нод

На рисунку 3.4 чорним зображені перешкоди, полігони з білим фоном – це navmesh, по яких можуть переміщатися сутності, червоним контуром відділені окремі вершини. [4]

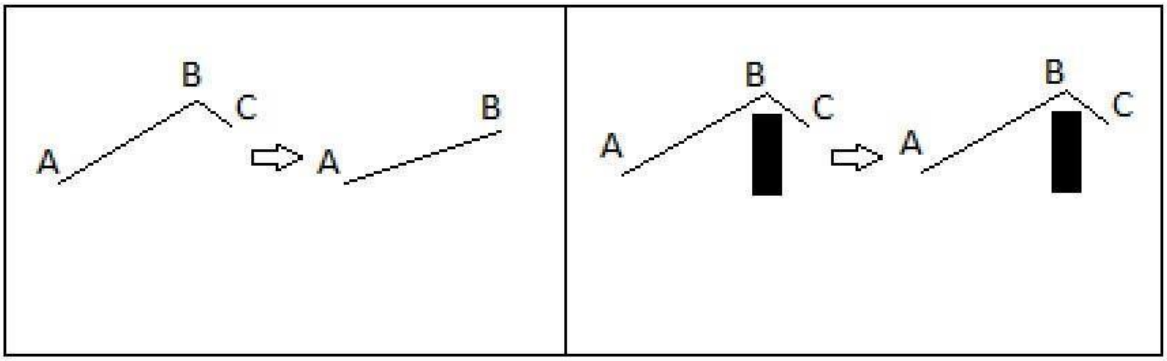


Рисунок 3. 5 Згладження шляху - важливий крок при переміщенні

Важливо пам'ятати, що часто шлях між двома точками буде мати вигляд ламаної. І як було описано у розділі 2.2 агенти повинні бути реалістичними, а тому ходити потрібно якомога плавніше. Тому, якщо серед обраних 3 послідовних

точок при сполученні крайніх двох утворений відрізок не перетинає перешкоду, то середню точку викидають. Такий спосіб

продемонстровано на рисунку 3.5 [4].

Після того, як граф готовий, по ньому можна здійснювати пошук за одним з існуючих алгоритмів.

3.1.2 Breadth-First Search

Цей алгоритм пошуку називають пошуком в ширину. Структура даних, в якій зберігаємо вершини, які потрібно перевірити – черга. Спочатку беремо початкову вершину і розглядаємо всі суміжні з нею вершини, тоді отримуємо всі вершини, суміжні до отриманих в попередньому кроці і так продовжується доки ми не дійдемо до вершини, яка є нашою ціллю. Це класична версія алгоритму, є також двосторонній пошук в ширину, коли одночасно шукати починають з точок старту і фінішу. В цьому випадку алгоритм завершується, коли вершина, знайдена з одного боку співпадає з вершиною, що вже була відмічена як відвідана з іншого боку [6].

Цей алгоритм дозволяти перевірити наявність шляху між двома вершинамиабо здійснити обхід всіх вершин. Для того, щоб знайти найкоротший шлях, алгоритм необхідно дещо удосконалити.

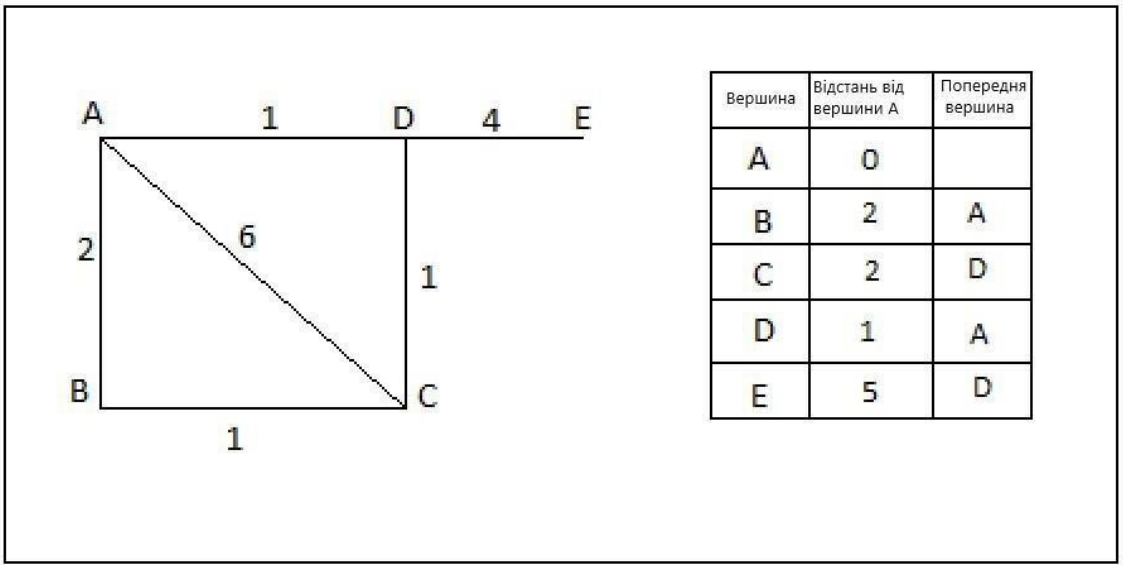
3.1.3 Depth-First Search

Пошук в глибину можна вважати доповненням до пошуку в ширину. Відмінність полягає в тому, щоб ми не одночасно розглядаємо всі суміжні вершини, а вибираємо їх по черзі. Стуктурою виступає стек. Проте даний алгоритм теж має свої недоліки. Ми не контролюємо глибину, на яку мизаходимо, тому навіть якщо шлях знайдено, нема гарантії, що він найкоротший.Для цього можна ввести обмеження глибини, і припиняти перевірку для певної вершини, якщо ми не досягли цілі, збільшити обмеження на 1 і повторити

алгоритм(називається ітеративним заглибленням), а якщо ціль досягнена, то алгоритм завершений і шлях знайдений успішно. Існує практика вибору сталого обмеження, проте, якщо вибрати надто високе значення, то витратимо зайвий час на обходження дерева і обов'язково потрібно слідкувати за відстанню, але це знову ж не дозволить забезпечити найкоротший шлях. Виправити цю проблему можна таким способом: присвоєння значення відстані кожній вершині. Тоді, якщо ми знайшли коротший шлях до цієї вершини, ми можемо оновити це значення і в кінці вибрати шлях, що має найменшу відстань. В пошуку в ширину вершини, що вже були відвідані, знову не відвідуються, в цьому ж алгоритмі не відвідуються вершини тоді, коли їхнє значення відстані менше ніж поточне [5].

3.1.4 Алгоритм Дейкстри

Цей алгоритм корисний тим, що він враховує ваги ребер графа.



Вершина	Відстань від вершини А	Попередня вершина
A	0	
B	2	A
C	2	D
D	1	A
E	5	D

Рисунок 3. 6 Приклад графу та структури значень для алгоритму Дейкстри

Для невідвіданих вершин використовується черга з мінімальнимпріоритетом, в якій пріоритет визначається за відстанню від стартової точки. Спочатку ми заповнюємо всі значення відстані, крім початкової вершини (тут значення 0), максимальним значенням. Тоді береться з черги перша вершина(в якої відстань від початкової найменша), на першому кроці це стартова вершина [12].

Тоді вибираються суміжні вершини і для кожної обраховується відстань як сума відстані від початкової до поточної і від поточної до наступної. Тоді порівнюється значення зі значенням, що вже записане в структурі, якщо нове значення менше, то воно вноситься в таблицю і поточна вершина записується в значення попередньої [12].

Таким чином алгоритм дає можливість як оцінити найкоротшу відстань, такі знайти список вершин пройшовши по значеннях попередньої вершини [12].

*3.1.5 Пошук шляху A**

Алгоритм A* вважається одним з найкращих алгоритмів для пошуку шляхів. Він об'єднує в собі алгоритм Дейкстри і евристичну оцінку залишку шляху. Кожна вершина має своє значення. Воно обраховується за такою формулою [13]:

$$f(n) = g(n) + h(n) \quad (3.1)$$

де:

n – вершина, f(n) – значення, що присвоюється вершині n, g(n) – відстань вершини n від початкової точки за алгоритмом Дейкстри, h(n) – евристична оцінка відстані від вершини n до цілі.

Для того, щоб алгоритм спрацював правильно, необхідно правильно обрахувати евристичну оцінку. Вона не повинна бути вищою ніж будь-яка реальна відстань до цілі. Зазвичай її обраховують як добуток мінімальної відстані від цілі і мінімальної ваги [13].

На ефективність алгоритму впливає кількість вершин, якщо їх надто багато,то списки відкритих і закритих вершин(нижче наведено пояснення для чого вони використовуються) стають надто довгими, і може не вистачити ресурсів для обрахунку всіх значень. Також впливає значення евристичної оцінки. Якщовона надто низька, то кількість необхідних обрахунків стрімко збільшується. Якщо ж вона надто висока, можна вибрати не найкоротший шлях. Чим ближчавона до реального значення, тим ефективніше і точніше працює алгоритм [13].

Полерозбивається на клітинки, які і є вершинами.

Отже, кожна вершина повинна містити відстань від старту за Дейкстрою, евристичну оцінку, і указник на вершину, з якої на неї ми перейшли, щоб потім можна було відновити послідовність переходів [13].

В нас повинні бути списки відкритих і закритих вершин. Так як серед відкритих вершин необхідно вибирати ту, в якої найменше значення f(n), то варто використовувати Heap або MinPQ, назвемо її Open, а список закритих вершин - Closed. Якщо 2 відкриті вершини мають однакове значення f(n), то вибирають ту, в якої евристична оцінка менша. Починаємо алгоритм з додавання початкової точки в Open. Тоді починається цикл, який закінчується, коли ми відкриваємо вершину, яка ж нашою ціллю, або коли Open пустий. Цикл виглядає наступним чином [13]:

current_node = get min from Open delete min from Open add current_node to Closed if current_node isgoal return for Node neighbor in neighbor_list for current_node if neighbor is in Closed continue if new_cost_for_neighbor>=current_cost_for_neighbor continue else current_cost_for_neighbor = new_cost_for_neighbor previous_node_for_neighbor=current_node

if not Open contains neighboradd neighbor to Open

Обчислення значення g(n) за Дейкстрою було описане в розділі вище. Тепер визначимо як обчислити евристичну оцінку. Візьмемо за приклад поле, на якому можна ходити на будь-яку з восьми сусідніх клітинок, тобто рухи по діагоналі є допустимі і на вартість переходу залежить тільки від відстані. Перехід на вверх/вниз/в сторону має вартість 1. Тоді перехід по діагоналі становить ~1.4. Для зручності перетворимо значення в цілі числа – 10 і 14 відповідно [13].

Тоді оцінюємо позицію нашої вершини відносно цілі. Отримуємо значення x та y – відстань по кожній осі. Менше значення буде кількістю переходів по діагоналі. Модуль різниці – кількість переходів по паралельно доосі [8]. h(n) = min(x,y)*14+abs(x-y)*10.

3.2. Вибір поведінки

3.2.1 Скінченні автомати станів

Скінченні автомати станів можна знайти чи не в кожній грі, що існує на ринку. Свою популярність вони здобули з ранніх днів розробки ігор. Для цьогоє свої причини – а саме ряд переваг:

А) Скінченні автомати досить прості в реалізації. Їхня структура доволі проста і легко розширюється за потреби.

Б) Відладка скінченних автоматів станів легка. Так як в них відсутня ієрархічна архітектура і стани самі по собі незалежні, кожен з них можна змінювати незалежно один від іншого.

В) Вони економно використовують ресурси. Для кожного стану прописані чіткі умови переходу до інших станів. Тому кількість інформації, яку треба оцінити, а також об’єм обрахунків порівняно малий.

Проте остання перевага наштовхує на запитання: «Якщо агент змінює станиза чітко визначеними правилами, то чи дійсно автомат скінченних станів можна вважати штучним інтелектом?». І справді цей алгоритм швидше симулює наявність інтелекту у агента. Хоча, якщо взяти до уваги, що визначення штучного інтелекту включає в себе симуляцію людської поведінки, то розуміємо, що ця умова виконується [7].

Щоб побудувати алгоритм автомат скінченних станів необхідно спочатку проаналізувати очікувану поведінку агента. Треба виділити чіткі самостійні стани, в яких може знаходитися персонаж.

Тоді для кожного стану слід визначити список станів, в які агент може перейти з нього, а також умови, які повинні задовольнитися для реалізації такого переходу.

Приклад псевдокоду для гри тамагочі, в якій агент може мати такі стани: щасливий, сумний, голодний,

сонний switch(current_state)case happy

play

if time since last ate >30 minchange_state(hungry) break

if time since last sleep > 3 hchange_state(sleepy)break

case hungry

cry

if given food

change_state(happy)break;

case sleepy

yawn if sleep

change_state(happy)break;

case unhappy

cry

if entertained

change_state(happy)break

if time since last ate >30 min

change_state(hungry)break

Таким чином автомати скінченних станів – це простими словами оператор switch-case з набором операторів умови в кожному випадку. З однієї сторони, такий підхід виглядає легким і зрозумілим, проте при намаганні реалізувати складний поведінковий механізм, розробник зіштовхнеться з проблемою надто об’ємного коду, відхиленням від правил хорошого тону в програмуванні і

методами зміни станів, які займають таку кількість рядків, що перестають бути читабельними [5].

Для реалізації простого поведінкового паттерну алгоритм скінченних станів пропонує швидше і легке вирішення задачі, проте при більш розвинутих схемах поведінки агентів варто подумати над підходом, який може бути складнішим в плануванні, проте зробить красивішими як структуру самої програми, так і поведінку агентів [5].

3.2.2 Поведінкові дерева вибору

Особливістю поведінкових дерев вибору є ієрархічне розміщення вузлів. Вони контролюють перебіг станів і поведінки агента.

Кожен вузол може повернути один з трьох найчастіше використовуваних статусів: Success, Failure або Running. Є випадки, коли видів статусів є більше,але для більшості ігор цього достатньо. Перший статус підтверджує успіх виконання, другий – невдачу, а третій повідомляє, що результат ще не відомий,так як вузол ще виконується [4].

Виділяють три основних архітипи: сполука(Composite), декоратор(Decorator) і листок(Leaf).

Сполука може містити одного або більше нащадка. Їхнє значення зазвичай визначається нащадками. Залежно від типу вузла, значення залежить від успіху виконання одного або всіх дітей[9].

Декоратор дає можливість додати рівень обробки інформації, що приходитьвід нащадка, який у декоратора може бути лише один. Декоратори можуть викликати виконання нащадка безперервно(нескінченний цикл), певну кількість разів або до отримання невдачі в якості результату, змінювати значення на протилежне[9].

Листок – найнижчий в ієрархії компонент. Він не має нащадків. До нього зазвичай прив’язана певна дія, яка повертає батьківському вузлу один зі статусів

виконання. Також в листку може знаходитися певна умова, невиконання якої призводить до повернення статусу невдачі[9].

Сполука може бути послідовністю(Sequence) або селектором(Selector). В першому випадку нащадки являють собою список подій, які мають справдитися для того, щоб батьківський вузол набув значення успіху. Для селектора достатньо одного успішного результату[9].

Побудова поведінкового дерева починається з аналізу агента і вибору базових його варіантів поведінки. Тоді поступово додаються нові гілки та нащадків, збільшуючи глибину дерева. Агенти в іграх зазвичай слідують дуже складним поведінковим деревам. Чим більша глибина дерева і чим більше в нього гілок, тим реалістичнішою буде поведінка об’єкта [9].

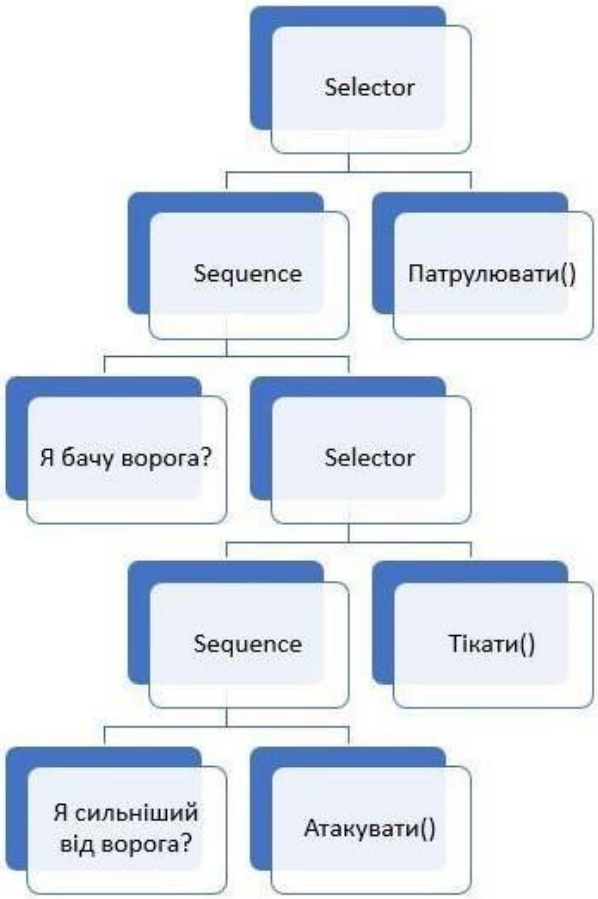


Рисунок 3. 7 Приклад схеми поведінкового дерева вибору

На рисунку 3.7 зображений приклад схеми поведінкового дерева вибору. Воно може застосовуватися для агента, що охороняє певну територію. Коренемдерева є селектор. Перший варіант – агент бачить ворога. Тоді йде селектор.

Якщо агент сильніший від ворога, то він атакує, інакше тікає. Другий варіант – агент ворога не бачить, тоді він просто патрулює територію. Це дерево демонструє просту поведінку агента, але його глибина – 5. До нього можна додати більше гілок, наприклад, для взаємодії з іншими агентами, які не є ворогами, чи пошуку слідів. Також можна додати дереву глибини визначивширізні варіанти розвитку атаки, вибору зброї і т. [6].

3.3. Прийняття рішення

3.3.1 Minimax

Minimax – це алгоритм, що популярний при реалізації ігор на два гравці. Він застосовується при необхідності вибору найкращого кроку.

Один гравець вважається мінімізатором, а інший – максимізатором. Саму жгру називають грою з нульовою сумою, так як на кожному кроці один графецьотримує очки, а в іншого вони віднімаються. Відповідно сумарне значення залишається сталим і дорівнює нулю[10].

При реалізації алгоритму ми повинні прорахувати всі потенційні варіанти розвитку ходу гри. Для цього будується дерево, листки якого мають певні значення. Кожен рівень дерева відповідає за хід певного гравця і ці рівні чергуються.

В корені дерева після завершення алгоритму в нас буде значення, що відповідає оптимальному кроку. Для обрахування значення кожного вузла, ми визначаємо, на рівні якого гравця вона знаходиться. Якщо це рівень мінімізатора, то ми вибираємо мінімальне значення нащадків цього вузла, длямаксимізатора навпаки.

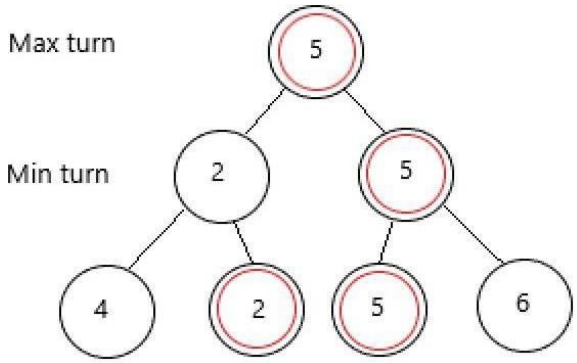


Рисунок 3.8 Вигляд дерева вибору

На рисунку 3.8 продемонстровано процес обрання найкращого кроку гравцем. На кожному рівні вибирається значення, що відповідає інтересам поточного гравця.

В корені дерева опинився хід зі значенням 5, отже вибрати слід його.

3.3.2 α-β відтинання

α-β відтинання – це удосконалений Minimax алгоритм. Головна його перевага – це зменшення кількості обчислень, що відповідно пришвидшує процес обрахунку оптимального кроку. α-β відтинання полягає в відтинанні гілок, які завідомо не містять значень, які нам пригодяться. α і β це додаткові параметри, які використовує функція пошуку найкращого кроку [5].

Значення α відповідає за найкраще значення для максимізатора, а β – для мінімізатора[11]. α надаємо значення MIN_VALUE, значення може оновлюватися на кроці максимізатора, β MAX_VALUE перед початком алгоритму, значення може оновлюватися на кроці мінімізатора.

Відтинання інших гілок здійснюється, якщо в отриманій гілці α>β.

Якщо в нас крок мінімізатора, ми визначаємо бету для всіх нод як мінімальне значення бети і значення цих нод, поки не дійдемо до кінця спискунод-нащадків або не отримаємо бету, яка менша за альфа.

Якщо ж в нас крок максимізатора, то визначаємо альфу як максимальне значення альфи і значення поточної ноди, поки не дійдемо до кінця списку ноднащадків або не отримаємо бету, яка менша за альфа.

Таким чином, алгоритм знаходить оптимальний шлях не витрачаючи надмірну кількість ресурсів на обчислення всіх значень, які завідомо не підходять

4. ПРИКЛАДИ РЕАЛІЗАЦІЇ ІГРОВОГО ШТУЧНОГО ІНТЕЛЕКТУ

4.1. Вибрані інструменти розробки

Для створення практичної частини курсової роботи був обраний Unity. Це кросплатформенний двигун для розробки ігор. Unity активно росте і розвивається, тепер за допомогою Unity можна розробляти 2D, 3D проекти, а також доповнену і віртуальну реальність.

Редактор Unity пропонує зручний інтерфейс для створення графічної частини застосунку, в ньому легко розміщувати елементи, організовувати ієрархії, визначати властивості об’єктів а також прив’язувати до них методи.

Основними структурними елементами розробленого проекту Unity можна вважати сцени, скрипти, спрайти і префаби.

Сцени – це певні локації, які зберігаються в окремих файлах. Вони можуть бути як великими, по яких можна переміщатися, так і представляти певне вікно(наприклад налаштувань).

Спрайти – це двовимірні зображення, які поміщаються на сцени і зазвичай дозволяють певний рівень взаємодії з ними.

Скрипти – це файли(.cs), в яких міститься код. Вони наслідують клас MonoBehaviour. Скрипти прив’язуються до об’єктів і стають активними тоді, коли існують ці об’єкти. Клас містить методи, які дуже корисні для управління об’єктами, наприклад Start() – метод, що викликається при першому оновленні фрейму після запуску скрипта. Update() викликається під час кожного оновлення.

Префаби – це попередньо зібрані елементи, або групи елементів з певними налаштуваннями, які використовуються в сценах кілька разів. Вони даютьзмогу уникати багаторазового копіювання, а також дозволяють редагувати всі екземпляри одного префабу за раз. Це особливо зручно, коли хочеться змінити стилі, чи розміри, або додати нащадка. Можна зробити це в одному місці, а всі екземпляри набудуть оновлених характеристик без додаткового втручання [5].

Unity надає всі необхідні інструменти для реалізації прикладів ігор з застосуванням штучного інтелекту, а тому був обраний як двигун для розробки.

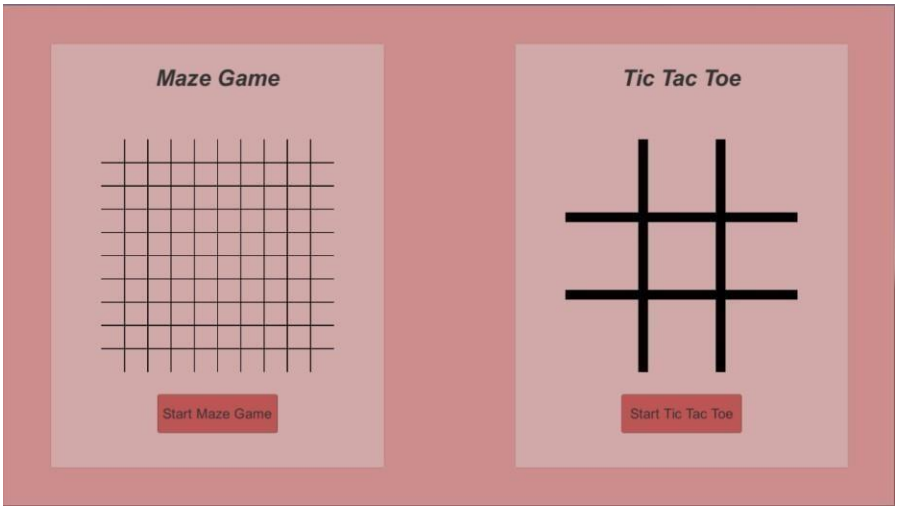


Рисунок 4. 1 Меню розробленого застосунку

У практичній частині було продемонстровано 2 алгоритми штучного інтелекту: A* для пошуку найкоротшого шляху та Minimax для гри хрестикинулики. Для початку гри необхідно натиснути на кнопку під обраною грою. На вікні кожної гри є кнопка «Restart Game», яка заново починає гру, а також кнопка «Go To Menu», яка повертає користувача на екранменю.

4.2. Пошук шляху в лабіринті за допомогою a*

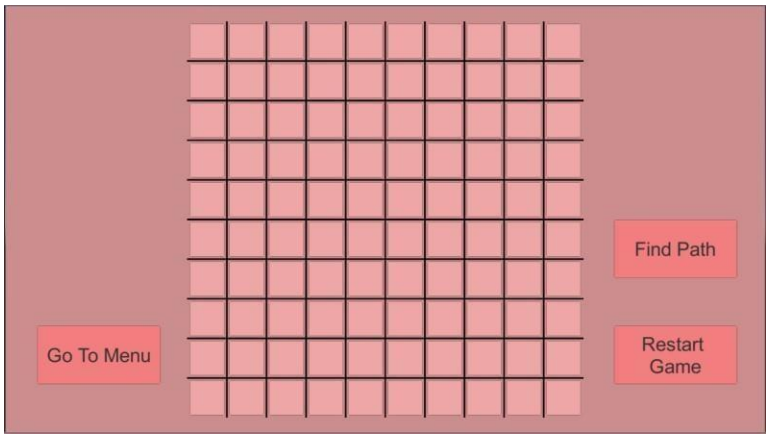


Рисунок 4. 2 Початковий вигляд гри "Maze"(лабіринт)

Суть розробленого застосунку полягає в тому, щоб знайти шлях між двома точками. Користувач має доступ до поля 10x10. Він сам обирає розташування елементів на екрані. Є 3 типи елементів: стартова точка, ціль і точкиперешкоди. Для розташування цих елементів, користувач натискає на вільні клітинки. При першому натисканні клітинка набуває значення старту. При другому натисканні обрана клітинка стає ціллю. Всі наступні натискання на клітинки утворюють перешкоди. Коли користувач утворив поле таке, яке його задовольняє, то тоді потрібно натиснути кнопку «Find Path», яка відобразить найкоротший шлях. Клітинки шляху стануть зеленими.

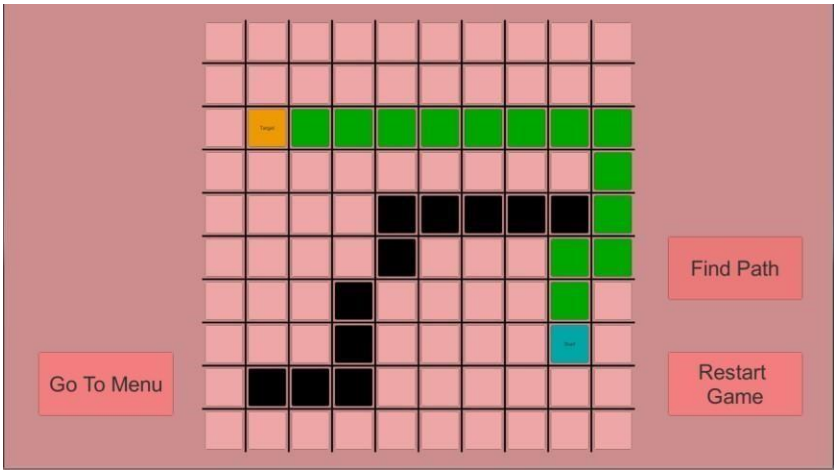


Рисунок 4. 3 Вигляд застосунку після пошуку шляху

На рисунку 4.3 зображений вигляд програми після відпрацювання. Голуба клітинка з написом «Start» - це точка початку пошуку шляху. Жовта з написом «Target» - це ціль, до якої необхідно знайти найкоротший шлях. Чорніклітинки – виставлені користувачем перешкоди. А клітинки зеленого кольору – це шлях, побудований алгоритмом. Якщо шляху немає, то з’являється відповідне повідомленння зліва.

Структурною одиницею, що використовується при пошуку, є об'єкт класу Node. Він містить інформацію про координати вершини, указник на попередню вершину, з якої була відкрита ця вершина, її значення за Дейкстрою та евристичною оцінкою і булеве значення, що відображає чи є цявершина перешкодою.

Для швидшої роботи алгоритму була реалізована така структура як Heap (купа). Вона дозволяє скоротити час обходу всіх відкритих верших, так як можна постійно вибирати першу з купи. Для цього клас Node повинен реалізувати IComparable, а для цього в ньому доданий метод CompareTo, що порівнює 2 об’єкти і повертає той, в якого оцінка менша.

Метод FindPath в класі PathFinder відповідає за пошук найкоротшого шляхуі повертає список вершин, що утворюють цей шлях.

```
public List<Node> Find(int sx, int sy, int tx, int ty, Node[,] grid)
                Node start = grid[sx, sy]; Node target = grid[tx, ty];
                 MinHeap open = new MinHeap(100);open.Insert(start);
                HashSet<Node> closed = new HashSet<Node>();while (!open.isEmpty)
                {
                        Node current = open.Pop();closed.Add(current);
                       if (current == target)
                       {
                        List<Node> res = new List<Node>();Node curr = current;
                        while (curr != start)
                              {
                                     res.Add(curr);
                                     curr = curr.PrevNode;
                              }
                        res.Add(curr);res.Reverse();return res;
                        }

                       {
                 else

                              foreach (Node n in Neighbors(grid,current))
                              {
                                     if (n.isPath && !closed.Contains(n))
                                     {


                                      int newg = current.gVal + 1;if (!open.Contains(n))
                                            {
                                                   n.gVal = newg;
                                                   n.hVal = Heuristic(n, target);n.PrevNode =
                                                   current; open.Insert(n);
                                            }
                                            else if (newg < n.gVal)
                                            {
                                                   n.gVal = newg; n.PrevNode = current;
                                                   open.UpdateNode(n);
                                            }
                                     }
                              }
                       }
                }
                return new List<Node>();
        }
```

Метод приймає на вхід координати стартової точки і цілі, а також масиву вершин. Точки, які були відкриті алгоритмом(для яких обраховане f- значення), зберігають в купі open, після того, як вершину взяли з купи і

опрацювали її сусідів, її поміщають в список closed, він потрібен лише для того, щоб перевіряти чи вершина вже була опрацьована. В купі вершини відсортовані за відстанню до цілі, вершина з найменшою відстаннюзнаходиться на першому місці.

Поки в купі є елементи, отримуємо з неї перший елемент, якщо цейелемент є нашою ціллю, то шлях знайдено, проходимося у зворотньомупорядку від цілі по попередніх першинах, поки не прийдемо до стартової вершини. Інакше проходимося по всіх незакритих вершинах-сусідах, які не єперешкодами і для них визначаємо значення f. Якщо вершина ще не була в open, то поміщаємо її туди, попередньо вказавши поточну вершину як попередню. Якщо вершина була в купі, але значення f більше за нове, то оновлюємо значення f, оновлюємо значення попередньої вершини і позиціювершини в купі. В інших випадках переходимо на наступну ітерацію.

Сусідів по діагоналі алгоритм не розглядає.

Після багаторазових тестувань було перевірено, що алгоритм дійсно знаходить найкоротший шлях, якщо такий існує, а швидкість пошуку задовольняє вимоги до ігор. Звичайно, якщо поле за розмірами буде 100*100 або й більше, то час виконання зросте, проте цей алгоритм має шляхи удосконалення, які при потребі можуть бути додані до реалізації.

4.3 Хрестики-нулики з використанням Minimax

Хрестики-нулики – це відома гра для двох, суть якої – отримати на полі 3х3ряд з 3 однакових символів(для одного гравця це X, для іншого O). Ряди можна утворювати по вертикалі, по горизонталі та по діалгоналях.

Так як гравці ходять по черзі і успіх кожного залежить саме від його ходу, то ця гра є хорошим прикладом для демонстрації роботи алгоритмів штучного інтелекту, що використовуються для прийняття рішення. В даному практичному завданні був обраний алгоритм minimax. Як вже описувалося в розділі 3.3.1, його головна ідея полягає в тому, щоб на кожному кроці будувати дерево, в якому містяться всі можливі комбінації розвитку гри, тоді для кожного термінального стану (в якому можна визначити переможця, або оголосити нічию), призначається певне значення (цінність такого результату). А тоді для кожного батька визначається значення, як максимальне/мінімальне значення серед усіх нащадків, залежно від того, чи є гравець, чий крок прораховують, максимізатором чи мінімізатором.

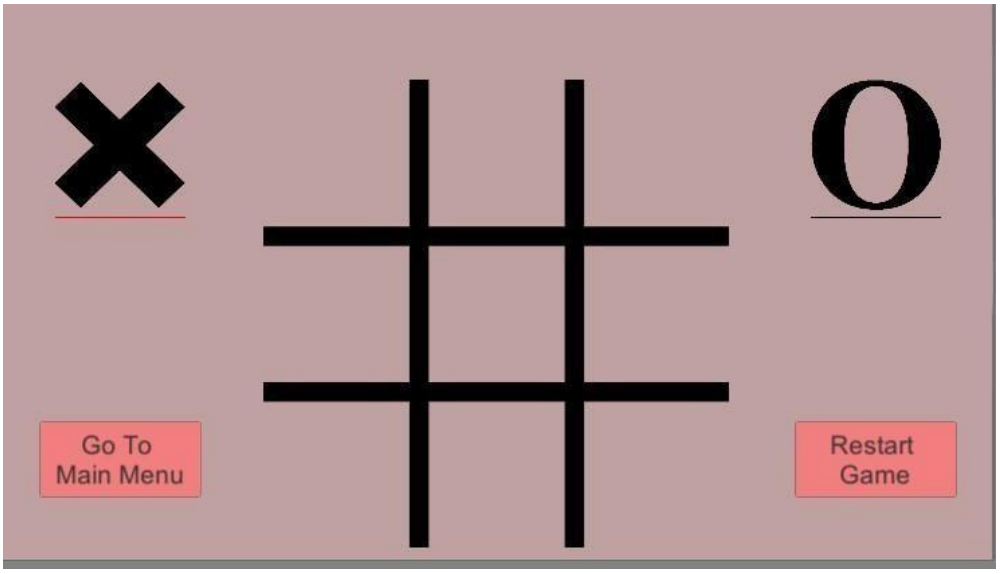


Рисунок 4. 4 Початкове вікно гри

Максимізатором був обраний штучний інтелект, так як при реалізації була зосереджена увага на отриманні якомога кращих результатів саме для нього.

```
int ChooseMove()
   {
          int best = -10000;
          int cell = -1;
          for (int i = 0; i < 9; i++)
          {
                 if (grid[i] < 0)
                 {
                        grid[i] = 2;
                        int current = Minimax(grid, false);
                        grid[i] = -100;
                        if (current > best)



                         {
                                best = current;
                                cell = i;
                         }
                  }
           }
     return cell;
}
```

ChooseMove() – це метод, який для кожного можливого кроку на даному етапі порівнює значення minimax та обирає найбільш вигідну для комп'ютерапозицію та повертає її в метод MakeChoice(int no), що приймає номер клітинки, визначає чий зараз крок і за цими даними здійснює відповідні змінина сцені.

Сам метод Minimax(int[] grid, bool computer) має такий вигляд:

```
int Minimax(int[] grid, bool computer)
{
       int res = Victory();
       if (res == 0)
       {
              return -1;
       }

       if (res == 1)
       {
              return 2;
       }

       if (res == 2)
       {
              return 1;
       }
       int best;
       if (computer)
```

```
{
        best = -10000;
        for (int i = 0; i < 9; i++)
        {
                if (grid[i] < 0)
                {
                        grid[i] = 2;
                        int current = Minimax(grid, false);
                        grid[i] = -100;
                        best = Math.Max(current, best);
                }

        }
}
else
{
        best = 10000;
        for (int i = 0; i < 9; i++)
        {
                if (grid[i ] < 0)
                {
                        grid[i ] = 1;
                        int current = Minimax(grid, true);
                        grid[i] = -100;




                 {
                        best = current;
                        cell = i;
                 }
         }
 }
 return cell;
}
```

В параметрах передається масив, що відповідає стану таблиці на поточному етапі та булеве значення, яке визначає чий зараз крок. Тоді обраховується найкраще значення для відповідного гравця і повертається у метод ChooseMove(). Даний алгоритм можна оптимізувати, якщо взяти до уваги глибину рекурсії. Адже на даному етапі реалізації, виграш на наступному кроці і виграш через 2 кроки є рівнозначними. Але хрестики- нулики – це гра, що при правильному алгоритмі вибору кроків не має програшу, а отже, якщо додати ще й цю перевірку, то в гравця не буде шансівна виграш. Задля інтересу з боку користувачів у ігор повинен бути шлях до виграшу, інакше гра стає нецікавою, тому дана реалізація є достатньою для виконання цієї задачі.

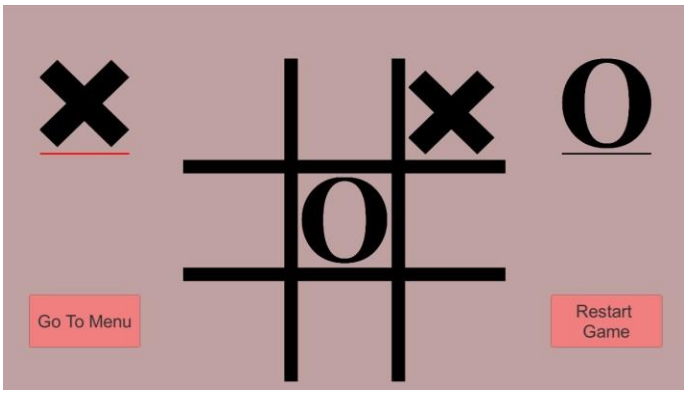


Рисунок 4. 5 Гра після першого кроку з обох сторін

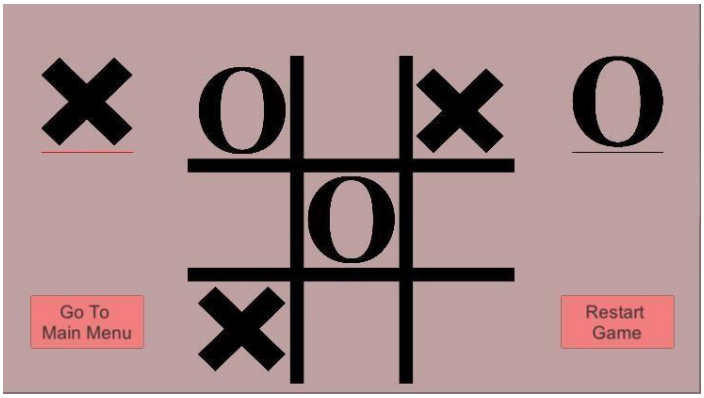


Рисунок 4. 6 Гра після другого кроку

На другому кроці видно, що комп’ютер (O) блокує шлях гравця до перемоги.

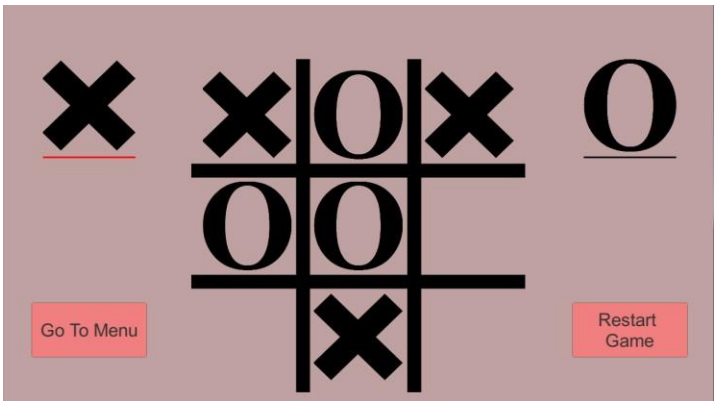


Рисунок 4. 7 Стан гри після третього кроку

Знову видно, що комп’ютер заблокував хід.

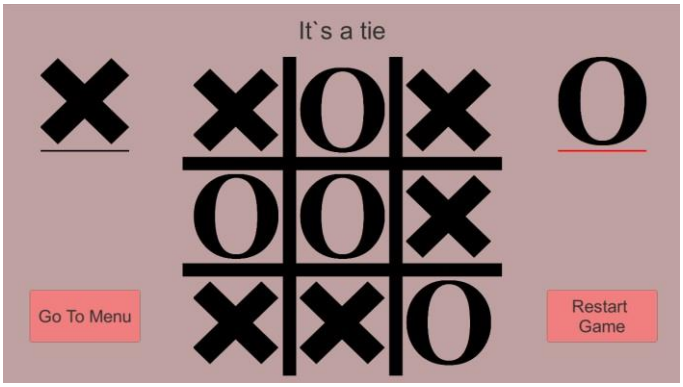


Рисунок 4. 8 Завершена гра. Результат – нічия

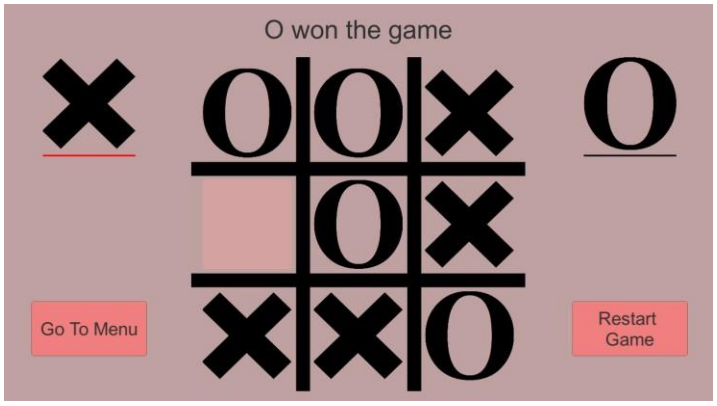


Рисунок 4. 9 Результат - виграш комп'ютера

Таким чином, реалізація гри хрестики-нулики за допомогою minimax демонструє те, що це хороший підхід до побудови штучного інтелекту в іграх,де необхідно приймати рішення. Особливо корисними такі підходи є для ігор, в яких гравці «ходять» по черзі та в яких є обмежена кількість комбінацій та термінальних станів.

ВИСНОВКИ

У даній роботі було розглянуте поняття штучного інтелекту та підходи до його побудови. Також для успішної реалізації практичної частини були освоєні інструменти для розробки відеоігор.

Для кожного розглянутого підходу до побудови штучного інтелекту було визначено, для яких типів задач кожен підхід найбільш пристосований. Серед задач, що вирішуються ігровим штучним інтелектом, були виділені такі категорії: practical path finding, вибір поведінки та прийняття рішення. Були розглянуті вхідні дані, які вимагають підходи, описані процеси їхнього виконання та отримання результатів, а також для деяких надано схеми та псевдокод для кращого сприйняття.

У практичній частині було реалізовано 2 алгоритми з різних категорій.

Перший алгоритм – це A* для пошуку шляху в лабіринті. Користувач вибирає на екрані 2 точки, після чого алгоритм знаходить найкращий шлях між цими точками. Алгоритм A* - це один з найпопулярніших алгоритмів для знаходження шляху, адже він не простим перебором комбінацій підбирає шлях,а використовує чергу з пріоритетом, а також для кожної клітинки обчислює її значення беручи до уваги як реальну відстань від стартової точки, так і евристичну оцінку – припущення залишку шляху до цілі. Ще одною перевагою даного алгоритму є можливість легко відновити послідовність відвіданих клітинок. Недоліком алгоритму є часозатратність, проте при експериментах з обчисленням евристичної оцінки можна знайти баланс між довжиною пройденого шляху і тривалістю обчислень.

Другий алгоритм був обраний серед алгоритмів для прийняття рішення. Minimax був використаний для реалізації гри хрестики-нулики. Цей алгоритм проходиться по всіх можливих комбінаціях ходів, для термінальних комбінацій

призначає певне значення, а потім шляхом оцінки найвигідніших ходів рівні дерева для відповідного гравця, отримує значення в корені дерева, яке є найбільш вигідним. Даний алгоритм можна покращити, використовуючи значення глибини дерева для більш точного визначення значень, обмеження максимального значення глибини для швидшого обрахунку, або й повністю перейшовши на альфа-бета відтинання, яке є оптимізованим варіантом minimax.

В цілому, в курсовій роботі розглянуто різні підходи до побудови ігрового штучного інтелекту, які надають можливість реалізації поведінки агента гри максимально наближено до поведінки опонента, що вміє міркувати, створювати розумове навантаження на гравця, та робить продукт цікавішим серед користувачів.

Список використаних джерел

1. What is Artificial Intelligence? [Електронний ресурс] -- URL https://builtin.com/artificial-intelligence
2. A. M. Turing, Computing Machinery and Intelligence [Електронний ресурс] URL https://www.csee.umbc.edu/courses/471/papers/turing.pdf
3. Pathfinding Demystified (Part I): Generic Search Algorithm [Електронний ресурс]- URL https://www.gabrielgambetta.com/generic-search.html
4. The University of Western Australia, Practical Path Finding [Електроннийресурс]- URL https://teaching.csse.uwa.edu.au/units/CITS4242/17-paths.pdf
5. Bryan Stout, Smart Move: Intelligent Path-Finding [Електронний ресурс]- URL https://www.gamasutra.com/view/feature/3317/smart_move_intelligent?print=1
6. Mat Buckland, Programming Game AI by Example [Електронний ресурс]- URL https://app.box.com/s/y4gvcrknxfmkefxbhlxt9ox5pxotks68
7. Kylotan, The Total Beginner's Guide to Game AI [Електронний ресурс]- URL https://www.gamedev.net/articles/programming/artificial-intelligence/thetotalbeginners-guide-to-game-ai-r4942/
8. Sebastian Lague, A* Pathfinding (E01: algorithm explanation) [Електронний ресурс]- URL https://www.youtube.com/watch?v=-L-WgKMFuhE

9. Chris Simpson, Behaviour trees for AI: How they work [Електронний ресурс] URL https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

10. Introduction to Minimax Algorithm [Електронний ресурс]- URL https://www.baeldung.com/java-minimax-algorithm

11. Akshay L Aradhya, Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning) [Електронний ресурс]- URL https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alphabetapruning/

12. Dijkstra's Algorithm [Електронний ресурс]- URL https://www.programiz.com/dsa/dijkstra-algorithm

13. a* search algorithm [Електронний ресурс]- URL https://www.geeksforgeeks.org/a-search-algorithm/