Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

АТЕСТАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Алгоритми проходження лабіринту в умовах, що змінюються
(тема)

Виконав:
студент II курсу, групи СПм-19-1
Павленко О.С.
(прізвище, ініціали)

Спеціальність
123 – Комп'ютерна інженерія
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: доц. Іващенко Г.С.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 – Комп'ютерна інженерія
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20___ р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ

студентові Павленку Олексію Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Алгоритми проходження лабіринту в умовах, що змінюються

затверджена наказом по університету від " 30 " жовтня 2020 р. № 1486 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 14 грудня 2020 р.

3. Вхідні дані до роботи Відкритий ігровий світ, лабіринт, змінні умови проходження, теорія графів, пошук найкоротшого шляху, алгоритми побудови маршруту, ігровий штучний інтелект, середовище розробки Unity 2019.2.3f1, інтегроване середовище розробки Visual Studio 2019, джерела інформації з теорії графів та вирішення задачі пошуку найкоротшого шляху, документація середовища розробки Unity 3D

4. Перелік питань, що потрібно опрацювати в роботі

1. Аналіз предметної області
2. Алгоритми пошуку найкоротшого шляху
3. Вибір технології розробки та інструментальних засобів
4. Програмна реалізація тестового програмного застосунку
5. Аналіз результатів досліджень
6. Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Демонстраційні матеріали. Слайд-презентація – 14 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд існуючих методів вирішення задачі	03.11.20-09.11.20	
	пошуку найкоротшого шляху		
2	Вибір та обґрунтування методики дослідження	10.11.20-17.11.20	
3	Вибір інструментальних засобів	18.11.20-23.11.20	
4	Проведення експериментів	24.11.20-01.12.20	
5	Оформлення матеріалів атестаційної роботи	02.12.20-07.12.20	
6	Подання атестаційної роботи керівникові та	08.12.20-09.12.20	
	її попередній захист		
7	Подання атестаційної роботи на рецензування	10.12.20-11.12.20	

Дата видачі завдання 02 листопада 2020 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Іващенко Г.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка атестаційної роботи: 101 с., 14 рис., 10 табл., 2 дод., 43 джерел.

ЛАБІРИНТ, МАРШРУТ, НАЙКОРОТШНИЙ ШЛЯХ, ГРАФ, АЛГОРИТМ ДЕЙКСТРИ, АЛГОРИТМ A*, ГЕНЕТИЧНИЙ АЛГОРИТМ, ІГРОВИЙ ШТУЧНИЙ ІНТЕЛЕКТ, UNITY, NAVMESH.

Метою атестаційної роботи є дослідження ефективності використання алгоритмів пошуку найкоротшого шляху у графах, якими описуються лабіринти. Лабіринти є загальною абстракцією віртуальних ігрових світів, для яких необхідно програмувати ігровий штучний інтелект для неігрових персонажів, що є актуальним завданням у теперішній час.

У ході виконання атестаційної роботи досліджувалися алгоритми Дейкстри та A* та генетичні алгоритми, для графів великої розмірності. Особливістю дослідження є можливість врахування алгоритмами змінних умов під час проходження маршруту. У якості змінних умов розглянуто можливість зміни кінцевої точки маршруту під час його проходження. Створено тестовий програмний застосунок з графічним інтерфейсом користувача, що забезпечує можливість дослідження впливу параметрів роботи алгоритмів на ефективність пошуку маршрутів.

ABSTRACT

Master’s thesis: 101 pages, 14 figures, 10 tables, 2 appendices, 43 sources.

MAZE, ROUTE, SHORTEST WAY, GRAPH, DIJKSTRA ALGORITHM, ALGORITHM A*, GENETIC ALGORITHM, GAME ARTIFICIAL INTELLIGENCE, UNITY, NAVMESH.

The purpose of the certification work is to study the effectiveness of the use of algorithms for finding the shortest path in the graphs that describe the mazes. Mazes are a general abstraction of open virtual game worlds, for which it is necessary to program game artificial intelligence for non-player characters, which is an important task nowadays.

During the certification work, Dijkstra and A* algorithms and genetic algorithms for large graphs were developing. A feature of the investigation is the ability to take into account of variable conditions during the routing. The possibility of changing the end point of the route during its passage is considered as variable conditions. A test software application with a graphical user interface has been created, which provides an opportunity to study the influence of algorithm parameters on the efficiency of route search.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ГА – генетичний алгоритм

ГРАФ – сукупність об’єктів зі зв’язками між ними

КРОСОВЕР – це операція отримання нащадків шляхом схрещування батьківських особин

ЛАБІРИНТ – це будь-яка структура, яка складається з заплутаних шляхів до виходу, або які ведуть в тупик

МУТАЦІЯ – випадкова зміна отриманих в результаті схрещування хромосом нащадків

СКРИПТ – послідовність дій для автоматичного виконання визначених завдань

ХРОМОСОМА – у контексті генетичного алгоритму, це бітовий рядок, у якій закодовані параметри рішення завдання

BFS – алгоритм пошуку в ширину (англ., Breadth-First Search)

COLLIDER – компонент Unity, який визначає форму об’єкта для фізичних взаємодій

MESH – сітка полігонів, з яких складається будь-який 3D об’єкт у комп’ютерній графіці

NAVMESH – це компонент у Unity, абстрактна структура даних для імітації штучного інтелекту для неігрових персонажів (англ., navigation mesh)

NPC – персонаж у іграх, який не знаходиться під контролем гравця, скорочено – неігровий персонаж (англ., Non-Player Character)

PREFAB – шаблонний об’єкт у Unity, який може зберігати компоненти, що застосовані до нього

TRANSFORM – компонент Unity, який визначає позицію, обертання та масштаб кожного об’єкта на сцені

ВСТУП

Сучасна ігрова індустрія дуже швидко розвивається і має різноманітний спектр пристроїв, які дають змогу користувачеві спробувати ту чи іншу гру. Велике значення має візуальна складова ігор, і сучасні ігри розвинулись від 2D-ігор з простою піксельною графікою до 3D-ігор, що мають графіку, схожу на навколишній світ. Такі ігри використовують ігрову фізику, яка розроблена на основі існуючих законів фізики реального світу.

Ринок ігор зростає, через це актуальним є прискорення процесу розробки, для чого доцільним є використання спеціальних засобів розробки – ігрових движків, які містять у собі реалізацію основних законів фізики навколишнього світу. За весь час існування ігрової індустрії розробниками було створено багато ігрових движків. Не дивлячись на їх різноманіття, всі ці движки включають у себе однакові компоненти, такі, як фізика поведінки різноманітних об’єктів у грі, освітлення навколишнього середовища гри і багато іншого. Наявність подібних компонентів наштовхнуло компанії-розробники створити окремі платформи для розробки ігор.

Платформа для розробки ігор – це програмне забезпечення, яке придатне для повторного використання і розширення для розробки різних ігор без істотних змін [1]. Зазвичай ігрова платформа вже містить фізику поведінки ігрових об’єктів, редактор для налаштування ігрового світу, реалізація стандартного освітлення (яке можна також налаштовувати). У теперішній час існує багато платформ для розробки ігор, особливо популярні серед них – Unity, UnrealEngine, CryEngine.

Кількість різноманітних ігор поступово зростала, через це доцільним э поділ по жанрам. Ігровий жанр – це об’єднана спільними характеристиками ігрового процесу сукупність ігор [2]. Існує велика кількість жанрів ігор, серед яких основними є симулятори, стратегії, спортивні, пригодницькі, рольові ігри, головоломки (логічні пазли), традиційні та настільні.

Ігри передбачають взаємодію гравця з ігровими об’єктами. У більшості жанрів ігровими об’єктами можуть виступати як ненатхнені, так і натхнені об’єкти. З цього слідкує, що в натхнених ігрових об’єктів окрім фізики має бути імітація інтелекту. Такі об’єкти звуться або NPC (Non-Player Character), або ботами. Щоб створити атмосферу дійсності, просто статичних ботів недостатньо, треба, щоб вони чимось займалися, наприклад пересувалися по ігровому світу. Існує багато способів реалізації руху ботів [3, 11]. Дуже часто у іграх боти виступають ворогами чи суперниками гравця. Наприклад, у іграх, геймплей (ігровий процес) яких передбачає перегони, або пересування гравця та NPC по ігровому світу, де важливе значення має взаємне положення ігрових об’єктів. Щоб зробити бота конкурентоспроможним та цікавим для гравця, треба не непросто задати йому дві точки для маршруту, але й зробити так, щоб він пересувався раціональним найкоротшим шляхом, з урахування поточних умов – доступною боту інформацією про карту ігрового світу, перешкоди на неї та об’єкти, що пересуваються (інших ботів та гравців).

Якщо абстрагуватися та розглянути світи різних ігор, то можна зробити висновок, що зазвичай ігровий світ може бути описаний у вигляді лабіринту. NPC повинні пересуватися по ігровому світу без зупинок на перешкодах, тобто повинні проходити лабіринти. Лабіринти можна генерувати у коді, а можна використати можливості платформи для розробки ігор, та створити їх самому. Лабіринти можуть бути описані за допомогою теорії графів, що дозволяє використовувати для вирішення завдань побудови маршрутів типові алгоритми пошуку найкоротшого шляху у графах.

Задача пошуку найкоротшого шляху є однією з найважливіших задач теорії графів. З цього приводу існує багато досліджень, де виявлено переваги і недоліки кожного алгоритму [8]. Але, якщо ігровий світ нестатичний, тобто частини лабіринту, що описує ігровий світ, змінюються, то пошук шляху має бути динамічно змінний. Отже стає актуальною задача пошуку найкоротшого шляху у лабіринтах у змінних умовах.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Представлення ігрового світу як лабіринту

Існує багато ігор з різноманітними ігровими світами. Основне призначення ігрового світу – надання віртуального ігрового простору, у якому відбуваються дії гравця [1]. Вибір, яким саме буде ігровий світ, безпосередньо залежить від жанру гри. Ігровий світ присутній у кожній грі, він наповнює гру навколишнім середовищем відповідно до жанру. У кожного ігрового світу є свої стандарти – це система розмірів, побудована на габаритах ігрових персонажів, яка дозволяє забезпечити правильне функціонування всіх геймплейних механік [2].

Ігровим світом можна назвати будь-який віртуальний світ. Віртуальний світ являє собою штучно створений світ, побудований за допомогою програмування на основі комп’ютерних технологій [3]. Віртуальні світи призначені для проведення часу користувачів, і у теперішній час у більшості нагадують інтерактивні віртуальні 3D-середовища. Ігрові світи бувають побудованими під певний сюжет гри, або відкритими для гравця. Відкритий світ – це термін у комп’ютерних іграх, який визначає віртуальний світ, який гравець може досліджувати і вільно досягати в ньому власних цілей [2].

Геймдизайн ігор, що мають відкрите віртуальне ігрове середовище з можливістю вільного переміщення, як правило, не містить «невидимих стін» і екранів завантаження, які є звичайними при лінійному геймплею [4]. Відкритий світ можна охарактеризувати як рівень або гру, створені як нелінійні відкриті локації з великою кількістю способів досягнення мети, яку переслідує гравець [4]. Відкритий світ забезпечує можливість вільного, не обов’язково за сюжетом гри, дослідження гравцем ігрового світу, в порівнянні з серією невеликих рівнів або в порівнянні з рівнем, виконаним лінійно під сюжет гри. Найвідомішими представниками таких ігор є серії ігор

Grand Theft Auto, Mafia, Cyberpunk, Saints Row. У цих серіях ігор світ представляється не лише відкритим, а й живим, завдяки заповненню вулиць неігровими персонажами. Кількість таких проектів стрімко збільшується.

Незалежно від типу ігрового світу, лінійний чи відкритий, кожний з них має вільні частини, по яких можна рухатися, і частини, через які неможна пройти – перешкоди. Абстрагувавшись від зовнішнього виду ігрового світу, і враховуючи, що маються вільні і непрохідні частини, можна стверджувати про подібність кожного ігрового світу до лабіринту.

Лабіринт – це будь-яка структура (у двомірному або тривимірному просторі), яка складається з заплутаних шляхів до виходу, або які ведуть в тупик [5]. Прикладом випадкового створення лабіринтів є різноманітні шахти, рудники, каменоломні. Найчастіше під терміном «лабіринт» мається на увазі штучна, спеціально створена, надзвичайно складна споруда. Походження таких лабіринтів досить давнє. Під лабіринтом у древніх греків і римлян малося на увазі більш-менш широкий простір, що складається з численних залів, камер, дворів і переходів, розташованих по складному і заплутаному плану, з метою заплутати і не дати виходу недосвідченій в схемі лабіринту людині. Найпростіший спосіб проходження [4], який полягає у тому, що якщо проходити лабіринт, торкаючись тільки одного з країв його стінок, то лабіринт обов'язково буде пройдено – не є завжди вірним, бо лабіринти бувають з непов'язаними стінами.

Лабіринти бувають двох видів: двох або тривимірними, тобто, одноповерховим або багатоповерховими [4]. У контексті ігрового дизайну, використання двовимірної графіки гри не обов'язково пов’язане з використанням саме двовимірного лабіринту, і, навпаки, часто тривимірна графіка може використовуватися при відображенні двовимірних лабіринтів. При двовимірній графіці зображення лабіринту може бути горизонтальне або вертикальне. У контексті тривимірної графіки теж існують два способи зображення (зазвичай тільки частини лабіринту): ізометрія і фронтальна проекція.

Існують три базові конфігурації лабіринтів:

1) Перша конфігурація – однозв'язкові.

2) Друга конфігурація – багатозв'язкові без «петлі» навколо мети.

3) Третя конфігурація – багатозв'язкові із замкнутою «петлею» навколо мети.

Однозв'язковими називаються лабіринти, що не містять замкнутих маршрутів і не мають окремо розташованих стін [4].

Лабіринти з окремо розташованими стінками та з замкнутими маршрутами називаються багатозв'язковими [4]. При цьому, якщо замкнутий маршрут не проходить навколо мети пересування, то це лабіринт другої конфігурації; якщо ж мету проходження лабіринту можна обійти по замкнутому маршруту, то даний лабіринт третьої конфігурації.

У лабіринтах першої і другої конфігурації завжди можна досягнути мети, якщо рухатися, весь час торкаючись рукою стінки. Але, цей шлях не буде найкоротшим.

В лабіринті третьої конфігурації таким способом ніколи не досягти мети, бо мета буде заточена у колі, яке буде обходитись по найбільшому замкнутому маршруту.

Часто людина, що проходить лабіринт (у реальному чи віртуальному світі), взагалі не знає, з лабіринтом якої конфігурації має справу, з цього виникає проблема визначення універсального алгоритму проходження лабіринту.

Універсальним алгоритмом проходження лабіринтів у реальному світі є алгоритм Люка-Термо [4].

Етапи алгоритму Люка-Термо:

Крок 1. На виході тунелю до перехрестя і на вході з перехрестя в тунель завжди робиться відмітка з однієї і тієї ж сторони (наприклад, праворуч) по ходу руху.

Крок 2. Від першого перехрестя потрібно йти за яким завгодно шляхом, поки не буде досягнуто або нового перехрестя, або глухого кута. Тоді:

Крок 3. Якщо пересування зупинилось у глухому куті, то слід повернутися назад, а пройдений шлях відкинути (крок 1), зазначивши його двічі (вхід-вихід). від перехрестя йти в іншому направленні.

Крок 4. Якщо пересування дійшло до нового перехрестя, то потрібно рухатися в будь-якому напрямку, зазначивши шлях, по якому відбувалося пересування, і поточний шлях.

Крок 5. Якщо пересування до відомого перехрестя відбувається не тим шляхом, то негайно потрібно повертати назад (виконуючи пункт 1).

Крок 6. Забороняється пересуватись по шляху, зазначеному двічі.

Крок 7. Якщо більше немає шляхів без відміток, треба обрати шлях, зазначений однієї міткою.

Алгоритм Люка-Термо має практичне застосування у реальному житті, але у контексті розробки ігрових програмних застосунків цей алгоритм є не актуальним через його повільну швидкодію, знаходження неоптимальних шляхів та неспроможність знаходження шляху у лабіринтах третьої конфігурації [4, 5].

1.2 Задача пошуку найкоротшого шляху

Ігрові світи представлені у вигляді лабіринтів, які можуть бути описані за допомогою теорії графів, що дозволяє використовувати для вирішення завдань побудови маршрутів типові алгоритми пошуку найкоротшого шляху у графах. Властивості графів вивчає теорія графів [5].

Теорія графів – це область дискретної математики, особливістю якої є геометричний підхід до вивчення об'єктів [6]. Зазвичай її відносять до топології (тому що в багатьох випадках розглядаються лише топологічні властивості графів), однак вона перетинається з багатьма розділами теорії множин, комбінаторної математики, алгебри, геометрії, теорії матриць, теорії ігор, математичної логіки і багатьох інших математичних дисциплін. Основний об'єкт теорії графів – граф і його узагальнення [7].

Граф являє собою сукупність двох непорожніх множин представлених множиною точок, які звуться вершинами та множиною ребр відповідно. Лабіринт може бути описаний у вигляді графу [6].

Першою класичною задачею теорії графів є «задача з Кенігсберськими мостами» [8]. Задача полягала у тому, що потрібно було пройти всі сім мостів, але можна було пройти по кожному мосту лише один раз. Для створення математичної моделі для вирішення цієї задачі був використаний граф.

Одною з найбільш важливих оптимізаційних задач теорії графів є завдання знаходження ланцюга, що з'єднує два вузли графа [8]. Нехай заданий спрямований граф, у якому кожному ребру поставлено відповідно невід'ємне число, яке є довжиною ребра. Довжиною шляху такого графа є сума довжин ребр, що складають цей шлях. У багатьох застосуваннях цього завдання має значення довжина шляху обігу графа [9].

Проблема пошуку найкоротшого шляху є однією з найважливіших задач класичної теорії графів, задача є дуже важливою і має багато різних практичних застосувань у реальному житті. Популярнішим прикладом є застосування цієї задачі у GPS-навігаторах для пошуку найкоротшого шляху між точкою відправлення і точкою призначення. Переміщення ігрових персонажів по ігровому світу можна розглядати у контексті вирішення проблеми навігації і пересування, для опису якої можна використовувати теорію графів, а для визначення маршруту ігрового персонажу – існуючі алгоритми обходу графів.

Для того, щоб NPC і боти долали перешкоди під час проходження лабіринту, є багато різних вирішень. Можна розробити власний алгоритм долання перешкод NPC і ботами, який буде працювати лише для розробленої гри, з урахуванням її особливостей, або використовувати існуючі готові алгоритми пошуку найкоротшого шляху, серед яких при розробці ігор найчастіше використовуються алгоритми Дейкстри та A* [10].

1.3 Аналіз існуючих методів вирішення задачі проходження лабіринту

У теперішній час існує багато досліджень з приводу вирішення задачі пошуку найкоротшого шляху у розробці ігрових програмних застосунків [11, 12, 18]. Поширення у геймдеві набули алгоритм Дейкстри та його модифікація, алгоритм A* [18]. Також у розробці ігор, через представлення об’єктів ігрового світу у термінах теорії графів, часто використовується алгоритм пошуку у ширину – BFS (Breadth First Search) [19].

У [18] дослідження алгоритмів Дейкстри, A* та BFS проводилися у три етапи. Тестові графи поділялися на блоки, та для кожного з використовуваних алгоритмів виконувався аналіз залежності обчислювального часу від розмірності задачі, тобто кількості блоків. Кожний наступний етап аналізу відрізнявся від попереднього кількістю перешкод, які були у лабіринті.

Дослідження ефективності алгоритмів Дейкстри, A* та алгоритму BFS [18], виявило, що при невеликій кількості перешкод алгоритм Дейкстри у статичному лабіринті має потребує менших часових витрат, ніж алгоритми A* та BFS, що показав найменшу швидкодію.

Окрім швидкодії у дослідженні [18] проводилося порівняння алгоритмів за кількістю застосовуваних блоків. Розбиття на однакову кількість блоків мали алгоритми Дейкстри та BFS, але найменшу кількість блоків (а через це – мали менші потреби до пам’яті) використовував для пошуку маршруту алгоритм A*. З цього зроблено висновок, що A* використовує найменше пам’яті для обчислювального процесу.

Другий етап дослідження показав, що алгоритмом з найменшим часом розрахунку був A*, який в свою чергу дослідив найменшу кількість блоків. Найгіршим з трьох алгоритмів по заміряним показникам став алгоритм BFS. Третій етап дослідження з найгіршим обчислювальним часом завершив алгоритм A*, але він залишився лідером у кількості досліджених блоків, бо дослідив менше за інші алгоритми.

Висновком розглянутого дослідження [18] стало твердження, що A* – кращий алгоритм пошуку шляху, особливо для використання у грі або лабіринті. Це підтримуються мінімальним необхідним обчислювальним процесом і відносно коротким пошуком час.

Поширеним є використання ГА для налаштування інших методів обчислювального інтелекту [13-17]. В дослідженні [20] зазначено, що кількість популяцій у алгоритмі пошуку найкоротшого шляху на основі генетичного алгоритму пропорційно залежить від розмірності лабіринту. Зокрема, у статті [21] генетичний алгоритм використовується для налаштування оптимальних параметрів проходження лабіринтів великої розмірності усіх трьох конфігурацій.

До недоліків, характерних для методів пошуку найкоротшого шляху на основі генетичних алгоритмів, можна віднести низьку швидкість (внаслідок багаторазового обчислення функції пристосованості) і можливість потрапляння в локальні екстремуми [21].

Розглянуті роботи не враховують можливість зміни умов під час проходження лабіринту, яким чином змінність умов впливає на роботу та налаштування алгоритмів, що можуть бути використані для вирішення задачі пошуку найкоротшого шляху.

1.4 Ігровий штучний інтелект

Сучасний світ наповнений різними технологіями, які допомагають людині у будь-якій її діяльності. У теперішній час одним із найважливіших помічників людини у повсякденному житті є штучний (обчислювальний) інтелект [22].

Засоби штучного інтелекту використовуються у багатьох сферах повсякденного життя людини. Найпопулярнішими сферами є навігація, системи пошуку, електрона пошта, автопілот у автомобілях, медицина, тематична реклама для користувача, комп’ютерні технології [23].

Штучний інтелект – це властивість програмних систем виконувати творчі функції, які традиційно вважають прерогативою людини [23, 24]. Однією із сфер використання штучного інтелекту у комп'ютерних технологіях є розробка ігор, де найчастіше штучний інтелект знаходить використання для опису поведінки персонажів, які керуються комп'ютером.

Штучний інтелект у розробці ігор звужує сферу діяльності і визначається саме як ігровий штучний інтелект, який представляє собою набір програмних процедур, які використовуються для створення ілюзії інтелекту в поведінці персонажів, що керуються комп'ютером [25]. Також реалізація ігрового штучного інтелекту сильно впливає на геймплей, бюджет гри і системні вимоги. Через це підхід до створення ігрового штучного інтелекту сильно відрізняється від методу реалізації традиційних методів штучного інтелекту. Щоб зберегти баланс, широко застосовуються різні спрощення, обмани та емуляції.

Існує три типи персонажів комп'ютерних ігор, які керуються ігровим штучним інтелектом [25]:

- NPC (Non-Player Charecter) – неігрові персонажі, які є дружніми або нейтральними до гравця;
- Bot – персонажі, які вороже ставляться до гравця. Вони наближені за ігровими можливостями до персонажу гравця;
- Mob – ворожі до гравця персонажі, які мають низький інтелект.

Кожний з перерахованих вище персонажів має свою роль у грі.

NPC потрібен у грі для заповнення і створення ефекту живого ігрового світу, та часткової взаємодії з персонажем гравця.

Mob відповідає за надання нескладної можливості гравцю накопичити ігровий досвід та ігрові ресурси при його знищенні, який не чинить сильного опору.

Bot потрібен для надання геймплею характеру змагання, він відрізняється ворожим ставленням до гравця і є складним суперник тому, що чинить сильний опір гравцю.

У кожного з різновидів персонажів комп’ютерних ігор є спільна властивість – можливість переміщення по ігровому світу. Ця властивість надає гравцю ілюзію інтелекту інших ігрових персонажів [3]. Як персонаж гравця, так і персонаж, який керується комп’ютером, мають змогу рухатись. Для забезпечення ілюзії реалістичної поведінки необхідно, щоб персонаж при своєму пересуванні не був строго детермінованим. Тобто його поведінка має бути не визначена заздалегідь сценарієм, а повинна змінюватися відповідно до поточних умов [1]. Для реалізації цього доцільне використання засобів обчислювального інтелекту.

Щоб розробити бота, який здатен пересуватися лабіринтом, що описує ігровий світ, потрібно керувати його поведінкою під час ігрової сесії. Для цього можна використовувати алгоритми пошуку найкоротшого шляху. Але ігрові світи бувають різними і майже всі світи мають у собі вільний простір для переміщення і об’єкти, які неможливо пройти наскрізь. Тому актуальною є задача пошуку шляху у лабіринту у змінних умовах.

1.5 Середовище розробки Unity 3D

Unity – це середовище розробки комп'ютерних ігор [26]. Але в Unity можна розробляти Desktop застосунки і мобільні застосунки. Серед переваг слід відмітити можливість кросплатформеної розробки на Unity. Середовище дозволяє створювати застосунки, що працюють під більш ніж 20 різними операційними системами, що підтримуються персональними комп'ютерами, ігровими консолями, мобільними пристроями, тощо.

Основними перевагами Unity є наявність візуального середовища розробки, підтримка багатьох платформ і модульної системи компонентів. Недоліками є труднощі при підключенні зовнішніх бібліотек і поява складнощів при роботі з багатокомпонентними схемами. На Unity написані тисячі ігор, симуляцій та застосунків, які охоплюють безліч платформ і жанрів.

Unity є дуже популярним середовищем розробки, особливо для ігор, яке використовується як великими розробниками, так і незалежними студіями. Редактор Unity має простий Drag&Drop інтерфейс, який легко налаштовувати [26]. Він складається з різних вікон, завдяки цьому можна проводити налагодження застосунка прямо в редакторі.

Проект в Unity поділяється на сцени (окремі файли), що містять свої ігрові світи із своїм набором об'єктів, сценаріїв, і налаштувань. Сцени можуть містити в собі як об'єкти (моделі), так і порожні ігрові об'єкти – об'єкти, які не мають моделі. Об'єкти, в свою чергу містять набори компонентів, з якими і взаємодіють скрипти. Також у об'єктів є назва (в Unity допускається наявність двох і більше об'єктів з однаковими назвами), може бути тег (мітка) і шар, на якому він повинен відображатися. Також, у будь-якого об'єкта на сцені обов'язково присутній компонент Transform. Цей компонент зберігає в собі координати місця розташування, повороту і розмірів об'єкта по всіх трьох осях. У об'єктів з видимої геометрією також за замовчуванням присутній компонент Mesh Renderer, що робить модель об'єкта видимої.

Також Unity підтримує фізику твердих тіл і тканини, а також фізику типу Ragdoll (потрібна для зберігання твердого тіла на сцені). У редакторі є система успадкування об'єктів, в якій дочірні об'єкти будуть повторювати всі зміни позиції, повороту і масштабу батьківського об'єкта. В редакторі скрипти прикріплюються до об'єктів у вигляді окремих компонентів.

При імпорті текстури в Unity можна згенерувати alpha-канал, mip-рівні, normal-map, light-map, карту відображень, проте безпосередньо на модель текстуру прикріпити не можна. Буде створено матеріал, за яким буде призначений шейдер, і потім матеріал прикріпиться до моделі. Редактор Unity підтримує написання і редагування шейдеров [26]. Редактор Unity має компонент для створення анімації, але також анімацію можна створити попередньо в 3D-редакторі (наприклад Blender 3D) та імпортувати разом з моделлю, а потім розбити на файли.

Розрахунки фізики виконує фізичний движок PhysX від NVIDIA. Unity 3D підтримує систему Level Of Detail (LOD) [26]. Задача системи LOD полягає в тому, що на далекій відстані від гравця високо деталізовані моделі замінюються на менш деталізовані, і навпаки. Також Unity підтримує систему Occlusion culling, задача якої в тому, що у об'єктів, що не потрапляють в поле зору камери не візуалізується геометрія і колізія, що знижує навантаження на центральний процесор і дозволяє оптимізувати проект.

Unity підтримує мову C#. При компіляції проекту створюється виконуваний (.exe) файл (для Windows), а в окремій папці – дані (включаючи всі рівні і спільні бібліотеки) [26].

Як правило, ігровий движок надає безліч функціональних можливостей, що дозволяють їх задіяти в різних програмних засобах, в які входять моделювання фізичних середовищ, карти нормалей, динамічні тіні і багато іншого. На відміну від багатьох ігрових движків, у Unity є дві основні переваги: наявність візуальної середовища розробки та міжплатформенна підтримка [26]. Перша перевага включає не тільки інструментарій візуального моделювання, а й інтегровану середу, лінію складання, що направлено на підвищення продуктивності розробників, зокрема, етапів створення прототипів і тестування.

Кросплатформеність визначає не тільки місця розгортання (установка на персональному комп'ютері, на мобільному пристрої, консолі і т.д.), але і наявність інструментарію розробки для різних операційних систем (інтегроване середовище може використовуватися під Windows и Mac OS).

Модульна система компонентів Unity дозволяє конструювання об'єктів, коли останні є комбінованими пакетами функціональних елементів [26]. На відміну від механізмів успадкування, об'єкти в Unity створюються за допомогою об'єднання функціональних блоків, а не додавання у вузли дерева успадкування. Такий підхід полегшує створення прототипів, що актуально при розробці ігор та інших додатків. Ще один недолік, пов'язаний з

використанням шаблонів примірників (Prefabs) [26]. З одного боку, ця концепція Unity пропонує гнучкий підхід візуального редагування об'єктів, але з іншого боку, редагування таких шаблонів є складним. Також, WebGL-версія движка, через специфіку своєї архітектури (трансляція коду з C# в C++ і далі в JavaScript), має ряд невирішених проблем з продуктивністю, споживанням пам'яті і працездатністю на мобільних пристроях [26].

1.6 Постановка задачі

Метою роботи є дослідження ефективності роботи алгоритмів пошуку найкоротшого шляху у змінних умовах, для застосування їх як імітацію інтелекту у неігрових персонажів в розробці ігрових програмних застосунків.

Для тестування алгоритмів пошуку найкоротшого шляху у змінних умовах потрібно створити ігровий світ, який може бути представлений у вигляді лабіринту, який, в свою чергу, описується за допомогою теорії графів.

Лабіринт, що э спрощеним відображенням будь-якого ігрового світу, повинен генеруватися засобами програми для тестування. Також для взаємодії користувача із лабіринтом потрібно забезпечити інтерактивний функціонал, щоб користувач міг створювати свій власний лабіринт для тесту алгоритмів пошуку найкоротшого шляху.

Для тестування обрані алгоритми Дейкстри, A* та генетичний алгоритм. Для реалізації тестового застосунку використовуються такі засоби розробки, як мова програмування C# та платформа для розробки ігор Unity 3D, що використовує дану мову програмування.

Створення лабіринту для аналізу роботи алгоритмів Дейкстри та A*, передбачає генерування координатної сітки, що буде заповнюватись префабом вузла (шаблонний об’єкт у Unity), який допоможе відобразити лабіринт. Стан вузла визначається булевою ознакою (вільний або перешкода). Відповідно до цього потрібно створити матеріали для префабів

Unity (білий – вільний для шляху, чорний – перешкода, зелений та синій – стартова та кінцева точки маршруту відповідно, жовтий – шлях, який було згенеровано алгоритмом).

Генерацію лабіринту для тестування генетичного алгоритму потрібно реалізувати за допомогою масиву цілих чисел, у якому 0 – вільний для шляху вузол, 1 – перешкода, 5 – стартова точка, 8 – кінцева точка маршруту. Для візуалізації також використовуються префаби Unity (білий – вільний для шляху, чорний – перешкода, зелений та синій – стартова і кінцева точки маршруту відповідно, червоний – шлях, який було згенеровано алгоритмом), які будуть розташовані у місцях на сцені Unity відповідно до їх значення. Щоб забезпечити ефективну роботу генетичного алгоритму потрібно використовувати повну матрицю відстаней, для отримання якої доцільно використати алгоритм Флойда.

Тестове програмне забезпечення повинно враховувати можливість зміни умов під час проходження маршруту у режимі реального часу. Можливі такі варіанти зміни умов: зміна положення і позиції стінок-перешкод у лабіринті; зміна області зору персонажу (що призводить до зміни доступної інформації, яка використовується для побудови маршруту пересування); зміна позиції кінцевою точки маршруту. При виборі інтерпретації змінних умов, що аналізуються у даній роботі, серед розглянутих варіантів обрана саме необхідність враховувати при побудові маршруту можливості зміни позиції кінцевої точки маршруту під час його проходження. Такі змінні умови зустрічаються у ігрових програмних застосунках найчастіше, а саме тому оптимізація їх урахування у роботі алгоритмів пошуку найкоротшого шляху буде найбільш ефективною для поліпшення швидкодії ігрових застосунків.

Потрібно розробити застосунок, який допоможе у тестуванні алгоритмів пошуку найкоротшого шляху у змінних умовах. Для доступності користування тестовим застосунком вирішено розробити його під операційну систему Windows. Інтерфейс користувача повинен забезпечувати вибір

алгоритмів для тестування. Велике значення має наочність аналізу роботи алгоритмів, для чого необхідна візуалізація як генерованого лабіринту, так і маршрутів його проходження, з урахуванням зміни кінцевої точки у режимі реального часу. Окрім відображення статичних перешкод, що притаманні лабіринту, застосунок підтримує роботу з динамічними перешкодами, що встановлюються користувачем.

У ході досліджень вимірюється час роботи алгоритму та точність запропонованого рішення – величина похибки відносно кращого варіанту маршруту. Аналіз роботи алгоритмів включає у себе запуск їх на тестових прикладах лабіринтів різної розмірності, щоб отримати результати про залежність часу роботи від обсягу даних. Такі виміри проводяться спочатку у статичних лабіринтах, а потім у змінних умовах. Також потрібно звернути увагу на тестування роботи алгоритмів при різній частоті змін позиції кінцевої точки під час пошуку шляху у реальному часі. Важливе значення має тестування роботи генетичного алгоритму у залежності від його початкових налаштувань, таких як обсяг популяції, стратегії відбору, вірогідність мутації та інші.

Аналіз передбачає використання однакових тестових прикладів, тобто запуск алгоритмів у рівних умовах, на однакових лабіринтах та з ідентичними сценаріями зміни позиції кінцевої точки. Для лабіринтів великої розмірності візуалізація не буде використовуватись через занадто дрібний масштаб при відображенні.

2 АЛГОРИТМИ ПОШУКУ НАЙКОРОТШОГО ШЛЯХУ

Задача про найкоротший шлях є однією з найважливіших класичних задач теорії графів [8]. Існує велика кількість різноманітних алгоритмів пошуку найкоротшого шляху. Найчастіше на практиці застосовуються різні модифікації таких методів, як метод використання жадібних алгоритмів, генетичні алгоритми, методи на основі гілок і меж, а також алгоритми мурашиної колонії [27, 28].

В даній роботі для дослідження можливих рішень обрано використання жадібних алгоритмів (Дейкстри та його модифікація, алгоритм A*), а також генетичний алгоритм.

2.1 Жадібні алгоритми

Жадібний алгоритм – алгоритм, що полягає в прийнятті локально оптимальних рішень на кожному етапі вирішення поставленої задачі, приймаючи, що кінцеве рішення також виявиться оптимальним [29]. Якщо глобальна оптимальність алгоритму має місце практично завжди, його зазвичай вважають за краще іншим методам оптимізації, таким як динамічне програмування.

Приклади жадібних алгоритмів:

- алгоритм Хаффмана;
- алгоритм Краскала;
- алгоритм Пріма;
- алгоритм Дейкстри [30].

Узагальненням жадібних алгоритмів є алгоритм Радо-Едмондс. Слід зазначити, що не кожен «жадібний» алгоритм дозволяє отримати оптимальний результат в цілому.

2.1.1 Алгоритм Дейкстри

Алгоритм Дейкстри названий на честь голландського вченого Едсгер Дейкстри (Edsger Dijkstra). Алгоритм був запропонований в 1959 році для знаходження найкоротших шляхів від однієї вершини до всіх інших в орієнтованому зваженому графі, за умови, що всі ребра в графі мають невід'ємні ваги [30] (рисунок 2.1). Алгоритм Дейкстри відноситься до так званих «жадібних» алгоритмів [29].

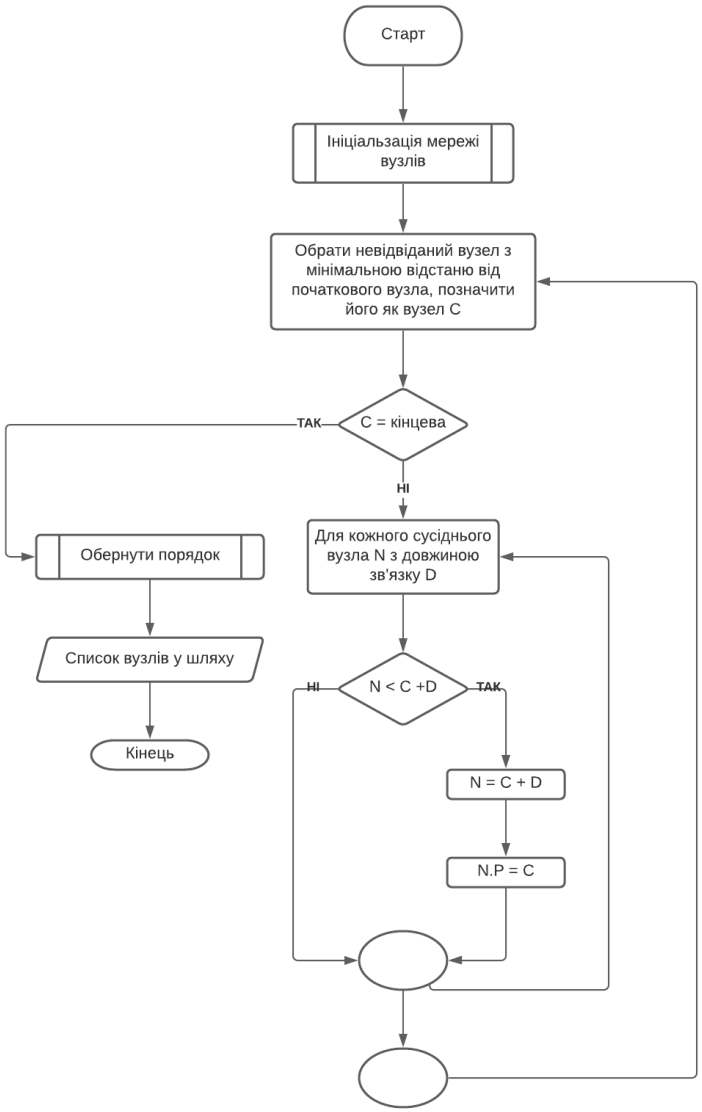

Рисунок 2.1 – Блок-схема алгоритму Дейкстри

Алгоритм пошуку найкоротшого шляху Дейкстри є «жадібним» в тому сенсі, що він завжди вибирає вершину, найближчу до джерела (початкової вершини графу), серед тих, найкоротший шлях до яких ще невідомий [31].

Алгоритм Дейкстри містить n ітерацій, на кожній з яких вибирається непомічена вершина з найменшою величиною $d[v]$, ця вершина позначається, і потім проглядаються всі ребра, що виходять з даної вершини. Вздовж кожного ребра робиться спроба поліпшити значення $d[v]$ на іншому кінці ребра. Час роботи алгоритму складається з наступних часових витрат:

- n раз виконується пошук вершини з найменшою величиною $d[v]$ серед всіх непомічених вершин, тобто серед $O(n)$ вершин;
- m раз проводиться спроба релаксації ребра.

При простій реалізації цих операцій на пошук вершини буде витрачатися $O(n)$ операцій, а на одну релаксацію ребра – $O(n)$ операцій, і підсумкова асимптотика алгоритму становить згідно (2.1) [32]:

$$O(n) = O(n^2 + m), \tag{2.1}$$

де $O(n)$ - підсумкова асимптотика алгоритму Дейкстри;

n – кількість ітерацій алгоритму Дейкстри;

m – кількість спроб релаксації ребра.

В орієнтованому зваженому графі $G = (V, E)$, вага ребр якого невід'ємна і визначається ваговою функцією $w: E \to R$, алгоритм Дейкстри знаходить довжини найкоротших шляхів із заданої вершини s до всіх інших.

В алгоритмі використовується множина вершин U, для яких уже обчислені довжини найкоротших шляхів до них з вершини s. На кожній ітерації основного циклу вибирається вершина u, якій на поточний момент відповідає мінімальна оцінка найкоротшого шляху. Вершина u додається в множину U і проводиться релаксація всіх ребр, що виходять з неї.

2.1.2 Алгоритм A*

Алгоритм A * – один з найпопулярніших методів вирішення завдань на пошук найкоротшого маршруту [33]. Його відносять до проінформованих алгоритмів пошуку, так як для вирішення завдань використовуються дані про вартість шляху і принципи евристики. Цей алгоритм набув широкого застосування при розробці ігрових програмних застосунків завдяки простоті та відносно невеликій обчислюівальний складності.

У 1964 році Нільс Нільсон винайшов евристичний підхід до збільшення швидкості алгоритму Дейкстри. Цей алгоритм був названий A1. У 1967 році Бертрам Рафаель зробив значні поліпшення до цього алгоритму, але йому не вдалося досягти оптимальності. Він назвав цей алгоритм A2.

В 1968 році Пітер Е. Харт представив аргументи, які доводили, що A2 був оптимальним при використанні послідовної евристики лише з незначними змінами. У його доказ алгоритму також включений розділ, який показував, що новий алгоритм A2 був можливо найкращим алгоритмом, враховуючи умови [34]. Таким чином, він позначив новий алгоритм в синтаксисі зірочкою, оскільки він починається на A і включав в себе всі можливі номери версій [35].

Порядок обходу вершин визначається евристичної функцією «відстань + вартість» (зазвичай позначається як $f(x)$. Ця функція є сумою двох інших: функції вартості досягнення даної вершини x з початковою (зазвичай позначається як $g(x)$ і може бути евристичною), і функції евристичної оцінки відстані від розглянутої вершини до кінцевої (позначається як $h(x)$).

Функція $h(x)$ повинна бути допустимою евристичною оцінкою, тобто не повинна переоцінювати відстані до цільової вершині [36]. Наприклад, для завдання маршрутизації $h(x)$ може являти собою відстань до цілі по прямій лінії, так як це фізично найменша можлива відстань між двома точками.

2.2 Генетичні алгоритми

Генетичний алгоритм (ГА) – це група алгоритмів, що імітують природню еволюцію. ГА який дозволяє виявити задовільне рішення до аналітично нерозв'язних проблем через послідовний підбір і комбінування шуканих параметрів з використанням механізмів, що схожі на процес біологічної еволюції [37]. Генетичний алгоритм є різновидом еволюційних обчислень, що засновані на понятті природного відбору.

Природний відбір – основний механізм еволюції. Суть відбору полягає в тому, що особини, які більш пристосовані, мають більше можливостей для виживання і розмноження і, отже, приносять більше потомства, ніж погано пристосовані особини. При цьому, завдяки передачі генетичної інформації (генетичному спадкуванню), нащадки успадковують від батьків основні їх якості (рисунок 2.2) [37].

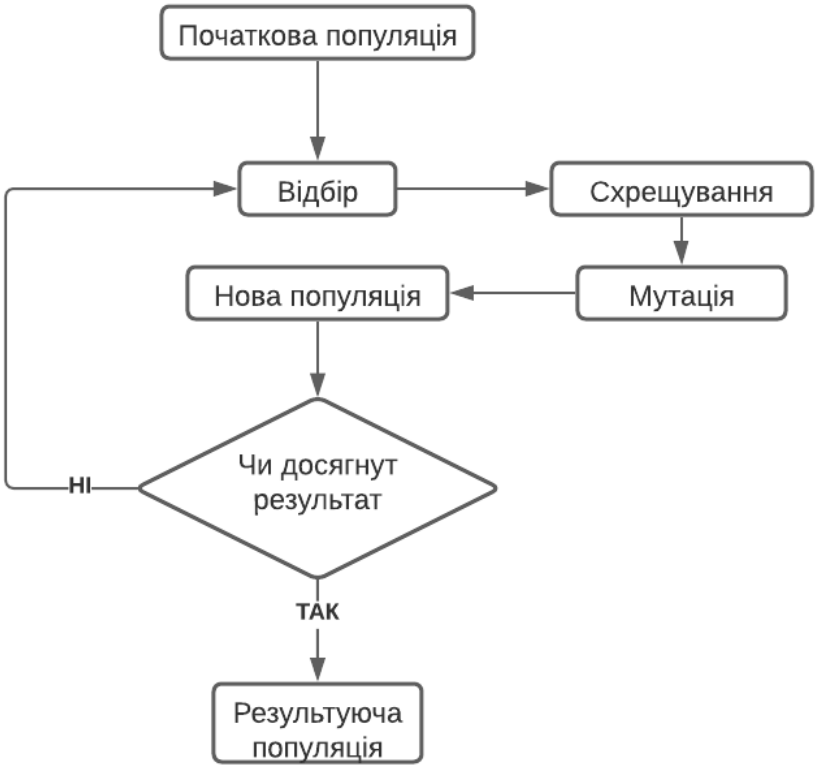

Рисунок 2.2 – Блок-схема генетичного алгоритму

Основоположником генетичних алгоритмів вважається Джон Холланд (John Holland), який є автором книги «Адаптація в природних і штучних системах» (Adaptation in Natural and Artificial Systems), яка є основоположною працею в галузі досліджень генетичних алгоритмів.

У генетичних алгоритмах застосовуються запозичений з генетики ряд термінів: гени, хромосоми, особина, аллель, генотип, фенотип, популяція.

Генетичні алгоритми застосовуються при розробці програмного забезпечення, в системах штучного інтелекту, оптимізації, штучних нейронних мережах і в інших галузях знань [37]. Слід зазначити, що завдання, для яких раніше використовувалися тільки нейронні мережі, у даний час можуть вирішуватись також за допомогою генетичних алгоритмів.

Генетичні алгоритми часто використовуються спільно з нейронними мережами [38]. Вони можуть підтримувати нейронні мережі або навпаки, або обидва методи взаємодіють у контексті гібридної системи, призначеної для вирішення конкретного завдання. Також генетичні алгоритми застосовуються спільно з нечіткими системами.

Існують багато різновидів моделей генетичних алгоритмів, які забезпечують застосовність ГА для вирішення завдань у конкретних сферах використання [38]. Правильний вибір моделі необхідний не тільки для можливості вирішення задачі, але й приведе до прискорення роботи розробленої системи. Найбільшого поширення набули такі моделі, як Canonical GA, Genitor, Hybrid algorithm, Island Model GA, CHC.

2.2.1 Принципи генетичного алгоритму

Генетичний алгоритм э евристичним алгоритмом пошуку, який використовується для вирішення завдань моделювання, шляхом послідовного підбору та оптимізації, комбінування і варіації шуканих параметрів з використанням механізмів, що імітують процеси біологічної еволюції [39].

Генетичні алгоритми виникли в результаті спостереження і спроб копіювання природних процесів, що відбуваються в світі живих організмів, зокрема, еволюції та пов'язаної з нею селекції (природного відбору) популяцій живих істот. Еволюційна теорія передбачає, що кожен біологічний вид цілеспрямовано розвивається і змінюється для того, щоб найкращим чином пристосуватися до навколишнього середовища. Можна сказати, що еволюція – це процес оптимізації всіх живих організмів [38].

У природі особини в популяції втягнуті у конкуренцію один з одним за різні ресурси, такі, наприклад, як їжа або вода. Найбільш пристосовані до навколишніх умов особини матимуть відносно більше шансів відтворити нащадків [39]. Особини, які слабо пристосовані, мають шанси зовсім не залишити потомства, або їх потомство буде дуже невеликим по кількості, відносно більш пристосованих особин. Це означає, що гени від високо адаптованих або пристосованих особин будуть поширюватися в збільшенні кількості нащадків на кожному наступному поколінні.

Комбінація оптимальних параметрів від різних батьків іноді може призводити до появи нащадку, чия пристосованість більше, ніж пристосованість будь-якого з його батьків [39]. Таким чином, вид розвивається, краще і краще пристосовуючись до середовища проживання з кожним наступним поколінням.

Наслідуючи природньому процесу, генетичні алгоритми здатні «розвивати» вирішення реальних завдань, якщо ті відповідним чином закодовані. Вони працюють з сукупністю «особин» – популяцією, кожна з особин представляє собою можливе рішення поставленої задачі.

Кожна особина оцінюється мірою її «пристосованості» згідно з тим, наскільки «добре» відповідне їй рішення задачі [40]. Найбільш пристосовані особини отримують можливість «відтворювати» потомство за допомогою «перехресного схрещування» з іншими особинами популяції. Це призводить до появи нових особин, які поєднують в собі деякі характеристики, успадковані ними від батьків. Найменш пристосовані особини з меншою

ймовірністю зможуть відтворити нащадків, так що ті властивості, якими вони володіли, будуть поступово зникати з популяції в процесі еволюції. Таким чином, з покоління в покоління, ті характеристики, що позитивно впливають на пристосовуваність особини, поширюються по всій популяції. Схрещування найбільш пристосованих особин призводить до того, що досліджуються найбільш перспективні фрагменти поверхні пошуку можливих значень характеристик.

Як і в біологічної еволюції, в генетичних алгоритмах використовується механізм мутацій, завдяки якому випадковим чином змінюються окремі особини. Мутації дозволяють досліджувати нові області поверхні рішень і не дають алгоритму затримуватись в точках локальних оптимумів. В процесі роботи генетичного алгоритму популяція буде сходитися до оптимального рішення задачі [39].

Спочатку для вирішення будь-якої задачі за допомогою генетичного алгоритму повинен бути розроблений метод кодування варіантів рішень [40]. Зазвичай рішення задачі представляються у вигляді рядків – стрінгів або хромосом. Кожен атомарний елемент в хромосомі (в разі бітових рядків – біт) називається геном. У разі, якщо рішення обмежене якимись межами, також складається алгоритм, який виключає хромосоми, що в процесі створення вийшли за ці межі. Для хромосом також є якась функція придатності або пристосованості (fitness-функція), яка визначає числове значення придатності хромосом для даного завдання, показує, наскільки близьким до найкращого є рішення, що відповідає цій хромосомі.

2.2.2 Етапи генетичного алгоритму

На початковому етапі генерується випадковим чином досить велика популяція особин, представлених множиною хромосом випадкового виду та обчислюється пристосованість особин в популяції [41]. Далі циклічно повторюється наступний процес: деяким чином з популяції відбираються

пари особин-рішень і схрещуються, дана операція називається кросовер (crossover). Рішення-нащадки додаються в популяцію, зазвичай замість найгірших рішень.

Для того, щоб не залежати від можливої обмеженості початкового набору побудованих хромосом і нездатності на їх основі побудувати оптимальну хромосому, що відповідає найкращому рішенню, використовуються мутації. Таким чином одержують зовсім нові хромосоми. Щоб мутації не заважали природним процесам, їх застосовують досить рідко. Далі цикл повторюється.

Критерієм зупинки в генетичному алгоритмі може служити знаходження точного рішення, якщо його можливо визначити заздалегідь іншими методами (наприклад, при пошуку коренів рівняння), виконання заданої кількості ітерацій або сталість найкращого знайденого рішення протягом певної кількості ітерацій алгоритму [42].

Генетичний алгоритм для вирішення завдання пошуку найкоротшого шляху не застосовується в чистому вигляді, він використовує специфіку завдання та різні евристики локального поліпшення у вигляді використання жадібних алгоритмів. У якості засобу кодування варіантів маршруту обігу графа найбільш природним буде перестановка з номерів вершин графа для кодування допустимого Гамильтонова циклу [43]. В цьому випадку, число в поданні позначає номер вершини, вершини з'являються в поданні відповідно до порядку їх обходу.

Інший спосіб кодування Гамильтонова циклу – позиційне уявлення. Формується еталонний список вершин, кожній з яких, присвоюється порядковий номер. При кодуванні рішення кожної вершини із списку ставиться у відповідність її порядковий номер в еталонному списку. Кодування вершини відбувається в порядку їх появи в обході, вершини, що закодовані, викреслюються зі списку і порядкові номери вершин, які залишилися, змінюються.

2.2.3 Основні операції генетичного алгоритму

Механізм простого генетичного алгоритму нескладний, він копіює послідовності і переставляє їх частини. Попередньо генетичний алгоритм випадково генерує популяцію послідовностей – стрінгів (хромосом). Потім алгоритм застосовує безліч простих операцій до початкової популяції і генерує нові популяції. Генетичний алгоритм, незалежно від обраної моделі, складається з декількох етапів [39].

Етап створення початкової популяції. Перед першим кроком потрібно випадковим чином створити якусь початкову популяцію; навіть якщо вона виявиться абсолютно неконкурентоспроможною, генетичний алгоритм все одно достатньо швидко переведе її в життєздатну популяцію.

Кросовер – це операція отримання нащадків шляхом схрещування батьківських особин. Щоб зробити особину-нащадку, необхідно декілька батьків; зазвичай потрібні рівно два [40].

Кросовер в різних алгоритмах визначається по-різному і сильно залежить від представлення даних. Головна вимога до кросоверу – щоб нащадок чи нащадки мали можливість успадкувати риси обох батьків, «змішавши» їх будь-яким досить раціональним способом (рисунок 2.3).

Рисунок 2.3 – Класичний одноточковий кросовер.

Існують різні варіанти кросовера, суть класичного одноточкового кросовера (рисунок 2.3) полягає в наступному: береться випадкова точка t на хромосомі [40]. Тепер, всі елементи масиву з індексами від 0 до t нової особини (нащадку) заповнюються елементами з тими ж індексами з масиву першої батьківської особини. Інші елементи заповнюються з масиву другої батьківської особини. Для другого нащадку робиться навпаки – елементи від 0 до t беруться від другого нащадку, а решта – від першого.

Серед стандартних видів кросовера найбільшу ефективність має порядковий кросовер. Цей тип кросовера працює наступним чином: в одному з батьків випадковим чином вибираються дві вершини, шлях між якими повністю копіюється в нащадку. Решта міста копіюються в нащадку в порядку їх перебування в маршруті, представленому у другому батьку генерованої особини.

Ще один поширений тип кросовера – «жадібний» кросовер. При його використанні в одному з батьків шукається найкоротший шлях, що містить задану кількість вузлів. Цей шлях повністю копіюється в нащадку, а решта міста копіюються в порядку їх слідування в другому батьку (аналогічно порядковому кросоверу).

Мутації – випадкові зміни отриманих в результаті схрещування хромосом нащадків [40]. Мутації застосовуються тільки до нащадків і здатні поліпшити або погіршити пристосованість особини. В результаті мутацій зміни може зазнати будь-який ген особини (рисунок 2.4).

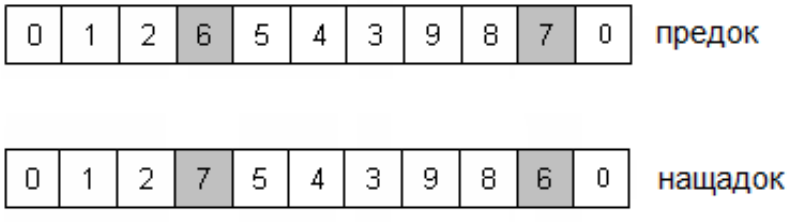

Рисунок 2.4 – Одноточкова мутація

На етапі відбору потрібно з усієї популяції вибрати певну її частку, яка залишиться і перейде в наступне покоління. Є різні способи проводити відбір. Імовірність виживання особини повинна залежати від значення функції пристосованості. Існує кілька підходів до вибору батьківської пари. Найбільш простий з них – панміксія [43]. Цей підхід передбачає випадковий вибір батьківської пари, коли обидві особини, які складуть батьківську пару, випадковим чином вибираються з всієї популяції. У цьому випадку будь-яка особина може стати членом кількох пар. Незважаючи на простоту, такий підхід є універсальним для вирішення різних класів задач. Однак він досить критичний до чисельності популяції, оскільки ефективність алгоритму, що реалізує такий підхід, знижується зі зростанням чисельності популяції.

Селективний спосіб вибору особин в батьківську пару полягає в тому, що «батьками» можуть стати тільки ті особини, значення пристосованості яких не менше середнього значення пристосованості по популяції, при рівній ймовірності таких кандидатів скласти шлюбну пару [41].

Такий підхід забезпечує більш швидку збіжність алгоритму. Однак через швидку збіжність селективний вибір батьківської пари не підходить у випадках, коли ставиться завдання визначення декількох екстремумів, оскільки для таких завдань алгоритм, як правило, надмірно швидко сходиться до одного з рішень, не завжди глобального. Крім того, для деякого класу задач зі складним ландшафтом функції пристосованості швидка збіжність може стати передчасною, до квазіоптимального рішення. Цей недолік може бути частково компенсований використанням відповідного механізму відбору, який би уповільнював занадто швидку збіжність алгоритму [42].

Інбридинг – це метод, при якому перший член пари вибирається випадково, а другим, з більшою ймовірністю, буде особина, яка максимально близька до нього [40]. При аутбридингу також використовують поняття схожості особин. Однак тепер з максимально далеких особин формують шлюбні пари.

Інбридинг та аутбридингу по-різному впливають на поведінку генетичного алгоритму. Інбридинг можна охарактеризувати властивістю концентрації пошуку в локальних вузлах, що фактично призводить до розбиття популяції на окремі локальні групи навколо вірогідних на екстремум ділянок ландшафту. Аутбридинг спрямований на попередження збіжності алгоритму до вже знайдених рішень, змушуючи алгоритм переглядати нові, недосліджені області.

2.2.4 Переваги і недоліки генетичних алгоритмів

Для вирішення різних завдань оптимізації добре зарекомендували себе адаптивні властивості генетичного алгоритму [37]. Головною метою завдань дискретної оптимізації є знаходження кращого, оптимального рішення з кінцевого, але досить великого числа їх можливих варіантів.

Задачі дискретної оптимізації є складними не тільки для класичних детермінованих алгоритмів, але і для генетичного алгоритму [41]. Крім того, область допустимих рішень може бути незв'язна і неопукла. Для визначення близькості рішень в цих завданнях використовуються різні техніки, що залежать, в основному, від виду завдання. Для подолання цих труднощів запропоновані різні модифікації генетичного алгоритму, які досить успішно справляються з вирішенням завдань цього класу.

Вибір параметрів генетичного алгоритму важко формалізується, та параметри встановлюються емпіричним шляхом [38], так як не існує об'єктивних доказів переваги тих чи інших налаштувань і операторів.

Для вирішення оптимізаційних задач комбінаторного типу використовуються специфічні алгоритми, серед них можна виділити декілька основних: методи відсікання, комбінаторні та наближені. Генетичні алгоритми відносяться до третього типу.

Генетичний алгоритм має властивість неявного паралелізму, що дозволяє йому динамічно шукати і досліджувати області, які містять у собі

локальні або глобальні оптимуми. Такі властивості повинні допомогти генетичному алгоритму при роботі в динамічно змінному середовищі.

Генетичний алгоритм не рекомендується використовувати [38]:

- в разі, коли задача знайти точний глобальний оптимум;
- якщо великий час виконання функції оцінки;
- коли необхідно знайти не одне, а всі можливі рішення задачі;
- складна конфігурація представлення рішення (кодування рішення).

Основними перевагами генетичного алгоритму є:

- не вимагають ніякої інформації про поверхні відповіді. Розриви, що існують на поверхні відповіді мають незначний ефект на ефективність використання, тож стійкість до потрапляння в локальний оптимум;
- добре працюють при вирішенні великомасштабних проблем оптимізації;
- можуть бути використані для широкого класу задач;
- прості і прозорі в реалізації;
- можуть бути використані в задачах з змінюється середовищем.

Генетичні алгоритми не гарантують виявлення глобального рішення за поліноміальний час [39], але дозволяють знайти квазіоптимальне, «раціональне» рішення. Під «раціональними» рішеннями розуміються рішення, які задовольняють дослідника. Адже в більшості реальних завдань немає необхідності знаходити саме глобальний оптимум. Найчастіше метою пошуків є рішення, які відповідають певним обмеженням.

2.3 Засоби пошуку найкоротшого маршруту у середовищі Unity

Середовище розробки Unity найчастіше використовують саме для розробки ігор, тому розробники часто зустрічаються із задачею навігації і пересування ботів або NPC. Для вирішення цієї задачі можна розробити систему навігації і пересування власноруч, або використати компонент середовища розробки Unity – NavMesh [26].

NavMesh – це скорочення від «навігаційна сітка», яка являє собою абстрактну структуру даних, що використовується в різних застосунках із використанням імітації штучного інтелекту для неігрових персонажів або для персонажу гравця (пересування якого відбувається за допомогою кліків миші по ігровому світу), щоб направляти агентів (об'єкти, для яких імітується штучний інтелект) в пошук шляхів від однієї точки до інший, що знаходяться на одній конкретній сцені (рисунок 2.5) [26]. NavMesh дозволяє неігровому персонажу проходити сцену в точних межах карти, не проходячи крізь стіни і не падаючи з карти. Такі помилки були дуже поширені в старих іграх, в яких навігаційна сітка не використовувалася.

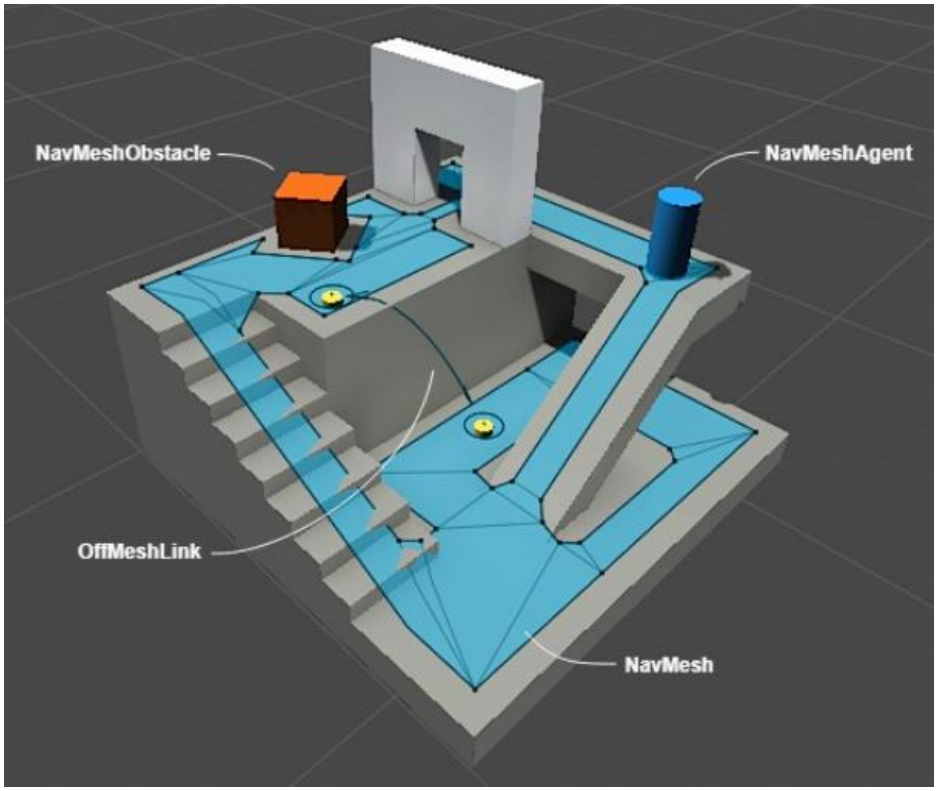

Рисунок 2.5 – Приклад дії компонента NavMesh на сцені

Основними поняттями і компонентами у NavMesh є сітка (Mesh) і агент (Agent). Mesh – це поле, утворене точками і лініями [26]. У контексті NavMesh поле представлене багатокутною сіткою, що складається з декількох ліній і багатокутників. Таким чином, сцену можна розглядати як

сітку, яка автоматично конвертована у граф. Agent – можна визначити як об'єкт, здатний виконувати дії самостійно, тобто неігровий персонаж[26]. Саме агент є об’єктом, у якого імітується присутність штучного інтелекту.

У компоненті NavMesh є можливість налаштовувати об’єкти ігрової сцени, чи є вони перешкодами, чи скрізь них можна пересуватися. Для сценарію, де у ігровому світі, створеному на сцені є перешкода у вигляді відсутності поверхні для пересування, передбачено функція стрибка. У NavMesh присутня функція підйому або спуску по поверхні, яка знаходиться під кутом [26]. Об’єкт agent має параметри налаштування швидкості переміщення і обертання, прискорення (визначає, наскільки швидко агент починає або змінює свій рух), відстань зупинки від мети, висоту та радіус циліндру, який визначає розміри об’єкту agent у фізиці компонента NavMesh (рисунок 2.6).

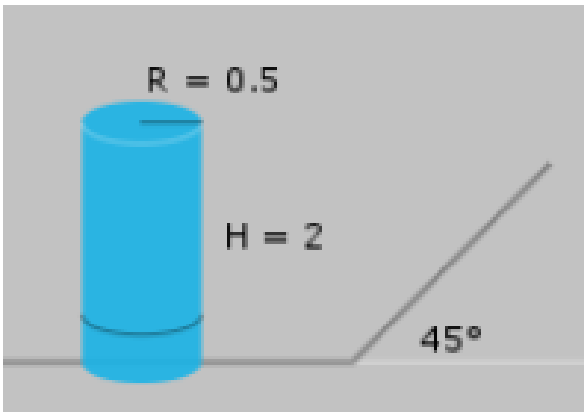

Рисунок 2.6 – Відображення параметрів об’єкту agent у Unity Editor

NavMesh входить в область дослідження комп'ютерних наук (а також математики), яка називається Pathfinding або Pathing. Це надзвичайно корисний інструмент для вирішення лабіринтів і пересування неігрових персонажів. Пошук шляху в значній мірі заснований на алгоритмі Дейкстри, який знаходить найкоротший шлях в графі.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Реалізація ігрового світу

Розробка тестового застосунку передбачає реалізацію декількох генерацій ігрового світу у зв’язку із специфікою роботи обраних алгоритмів пошуку найкоротшого шляху. Окремо розроблено засоби генерації ігрового світу для алгоритмів Дейкстри та A*, для генетичного алгоритму та для використання компоненту Unity – NavMesh.

3.1.1 Ігровий світ для тестування алгоритмів Дейкстри та A*

Для генерації ігрового світу для алгоритмів Дейкстри та A* створено шаблонний об’єкт (префаб) з геометричного примітива – куба, до якого додано компонент у вигляді скрипта Node.cs. У собі клас Node містить деякі характеристики, властивості та методи префаба-вузла, що передаватимуться у алгоритми Дейкстри та A*, такі як, вага вузла, батьківський вузол, список сусідніх вузлів, та змінну, що вказує на стан вузла (чи є він відкритим для шляху, чи перешкодою). За замовчуванням його забарвлено у білий колір.

Стартова та кінцеві точки маршруту помічаються зеленим і синім кольором відповідно, їх забарвлення відбувається, коли згенеровано площу для тестування при старті сцени за допомогою методу SetColor (приклад 3.1).

```
void SetColor(Transform node, Color color)
    {
        Renderer rend = node.GetComponent<Renderer>();

        rend.material.color = color;
    }
```

Приклад 3.1 – Метод для зміни кольору префабу (файл GridGenerator.cs)

Метод має вхідні параметри: node типу Transform (компонент у Unity, який у собі містить параметри розташування об’єкту у тривимірному просторі, його кут повороту у просторі сцени, та розміри об’єкту у тривимірному просторі).

В node передається префаб, який треба забарвити; color типу Color приймає у себе колір, яким потрібно забарвити, клас Color містить у собі основні кольори. У об’єкт типу Renderer передається Renderer існуючого префабу і через властивість material змінюється значення поля color.

Генерація сітки з префабів виконується від початку координат у тривимірному просторі. Для створення сітки створено поля rowInGrid та columnInGrid, які відповідають за кількість стовпців та колонок сітки. Налаштування значень змінних відкрите для користувача, тобто користувач може сам встановити розмірність сітки, на якій тестуються алгоритми. Сама генерація сітки відбувається у методі GenerateGrid (приклад 3.2).

```
        private void GenerateGrid()
        {
            int counter = 0;
            for (int i = 0; i < rowInGrid; i++)
            {
                for (int j = 0; j < columnInGrid; j++)
                {
                    Transform node = Instantiate(nodePrefab, new Vector3((j*
padding) + gameObject.transform.position.x, gameObject.transform.position.y,
(i * padding) + gameObject.transform.position.z), Quaternion.identity);
                    node.name = "node (" + counter + ")";
                    grid.Add(node);
                    counter++;
                }
            }
            startNode = grid[startNodeIndex];
            endNode = grid[endNodeIndex];
            if (startNode != null)
            {
                this.SetColor(startNode, Color.green);
            }
            if (endNode != null)
            {
                this.SetColor(endNode, Color.blue);
            }
        }
```

Приклад 3.2 – Метод генерації сітки (файл GridGenerator.cs)

У подвійному циклі, в межах стовпців і колонок, які задав користувач, будуються сітка та заповнюється префабом вузла. Ініціювання префаба вузла на сцені відбувається у функції Instantiate, яка приймає в себе параметри префаб вузла, позицію на сцені та кут повороту у просторі сцени. Метод встановлює за замовчуванням стартову та кінцеву точки маршруту. Якщо стартова та кінцева точки існують, то забарвлює їх за допомогою описаного вище метода SetColor.

Також за допомогою методу GenerateNeighbours при генерації тестової площини знаходяться сусідні вузли для кожного вузла і записуються кожному вузлу у його список сусідів, який реалізований у класі Node. Тобто кожен вузол зберігає інформацію про своїх сусідів незалежно один від одного. Цей метод потрібен для реалізації алгоритму Дейкстри.

Для ігрового світу додана функція інтерактивності, щоб користувач міг взаємодіяти з ігровим світом, встановлюючи на ньому перешкоди. Таким чином, користувач може будувати свої лабіринти. Функція реалізована у класі UserInput за допомогою метода EnableOrDisableOffObstracle. Метод приймає у якості параметрів вузол, на який натиснув користувач та компонент Renderer вузла, на який натиснув користувач.

```
      void EnableOrDisableOffObstracle(Node node, Renderer renderer)
          {
              if    (node    !=    gridGenerator.StartNode    &&    node    !=
gridGenerator.EndNode)
              {
                  if (node.IsWalkable() == true)
                  {
                      node.SetWalkable(false);
                      renderer.material.color = Color.black;
                  }
                  else if (node.IsWalkable() == false)
                  {
                      node.SetWalkable(true);

                      renderer.material.color = Color.white;
                  }
              }
          }
```

Приклад 3.3 – Метод встановлення або зняття перешкод (файл UserInput.cs)

У кожного вузла є метод SetWalkable та властивість IsWalkable. Метод SetWalkable приймає у себе параметр типу bool і встановлює стан вузла – true, якщо вузол є вільним для пересування, false – якщо вузол став перешкодою, тобто стінкою лабіринту. Властивість IsWalkable показує стан вузла true або false – вільний для пересування або перешкода відповідно.

Метод EnableOrDisableOffObstracle містить декілька етапів перевірок. Перший етап перевірки, чи є обраний вузол початковим або кінцевим. Потім перевіряється, чи є вузол вільним для пересування, якщо так, то встановлює його стан у false, тобто вузол стає перешкодою і перефарбовує у чорний колір. І також, якщо вузол є перешкодою, то встановлює true і вузол стає вільним для пересування і перефарбовується у білий колір.

3.1.2 Ігровий світ для тестування генетичного алгоритму

У ігровому світі відображення лабіринту починається з початкової точки координат у тривимірному просторі. Ігровий світ, який містить у собі тестове поле, генерується за допомогою матриці чисел. У матриці цілими числами описана карта лабіринту, 1 – стінка лабіринту, 0 – вільно для шляху, 5 – стартова позиція та 8 – кінцева позиція. Кожному числу присвоєний префаб за допомогою методу PrefabByTile (приклад 3.4).

```
public GameObject PrefabByTile(int tile)
{
      if (tile == 1) return wallPrefab;
      if (tile == 5) return startPrefab;
      if (tile == 8)
            return exitPrefab;
      if (tile == 0)
            return nodePrefab;

      return null;
}
```

Приклад 3.4 – Метод, що встановлює відповідність між числом та префабом, який ілюструє число у ігровому світі (файл MazeController.cs)

Метод PrefabByTile приймає у себе один параметр tile типу int, в нього передається матриця і встановлюється відповідність між числами у матриці і префабами, які їх ілюструють.

Для відображення карти лабіринту, що описана у матриці, створено чотири префабу з геометричного примітива – куба: білий куб – вільний для шляху простір; чорний куб – перешкода, або стінка лабіринту; червоний куб – стартова позиція; зелений куб – кінцева позиція. Ініціалізація префабів на сцені відбувається за допомогою методу Populate (приклад 3.5).

```
public void Populate() {

      for (int y = 0; y < map.GetLength(0); y++) {
            for (int x = 0; x < map.GetLength(1); x++) {
                  GameObject prefab = PrefabByTile (map [y, x]);

                  if (prefab != null) {
                        GameObject wall = Instantiate (prefab);
                        wall.transform.position = new Vector3 (x, 0, -
y);
                  }
            }
      }
```

Приклад 3.5 – Метод ініціювання на сцені префабів (файл MazeController.cs)

За допомогою подвійного циклу, у якому встановлено межі у виді висоти і ширини матриці генеруються префаби на сцені. Спочатку створюється об’єкт типу GameObject, яким описуються усі об’єкти на сцені і у нього подається відповідність префабу до числа. Потім перевіряється, чи об’єкт містить у собі префаб.

Якщо об’єкт не пустий, то за допомогою методу Instantiate, який є стандартним методом у Unity і приймає у якості параметра створений об’єкт, створює отриманий префаб у об’єкті на сцені. Всі описані вище дії відбуваються у методі Start – стандартний метод Unity. Метод Start – один з основних методів життєвого циклу програмного застосунку у Unity, виконується при старті сцени.

3.1.3 Ігровий світ для тестування компоненту NavMesh

Компонент NavMesh підтримує роботу з заздалегідь створеними ігровими світами. Опираючись на цю особливість компоненту NavMesh створено ігровий світ, який описується лабіринтом. Створено префаб з геометричного примітиву куба. Для префабу зроблено чотири матеріали. Білий колір відповідає за вільний шлях, чорний колір означає перешкоду, синій колір позначає стартову точку, червоний колір – кінцеву точку. Створення карти зроблено у редакторі сцени власноруч, без програмного коду (рисунок 3.1).

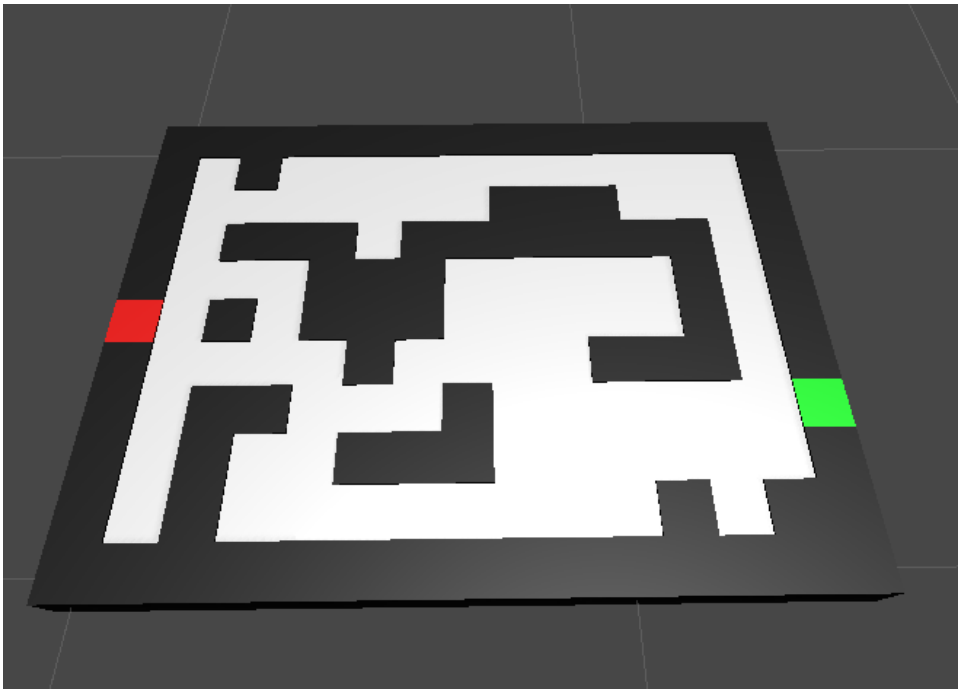

Рисунок 3.1 – Лабіринт для компоненту NavMesh

Потім завантажено компонент NavMesh і імпортовано його у проект. Створено префаб гравця, який пересуватиметься по лабіринту, та додано на нього компонент NavMeshAgent і встановлено шар «player». У цьому компоненті регулюється швидкість, прискорення агенту та його тип. Також компонент містить налаштування радіусу зупинки від цілі, яку було задано.

У даній роботі майже всі налаштування NavMeshAgent залишені за замовчуванням, окрім радіусу зупинки, його встановлено на 0, тобто відстані між ціллю та агентом не буде, коли агент її досягне. На префаби перешкод додано компонент NavMeshObstacle, які містять налаштування типу, розміру і обрізу на сітці NavMesh і статичного та нестатичного положення. На всі перешкоди встановлено статичне положення, налагоджені розміри під розміри самих префабів, обріз не встановлено, бо перешкоди кубічної форми.

Після встановлення компонентів агенту і перешкод на сцені створюється пустий об’єкт, на який додано компонент NavMeshSurface, який генерує сітку можливих пересувань. За допомогою NavMeshSurface запікається сітка пересувань (зберігаються вільні місця для пересування), і карта стає готовою до використання. Компонент містить налаштування агенту та перешкод, шарів, які можна виключити з побудування сітки.

3.2 Реалізація алгоритмів пошуку найкоротшого шляху

У даній роботі реалізовано декілька алгоритмів пошуку найкоротшого шляху на графі. Досліджується використання алгоритмів Дейкстри, A* та генетичних алгоритмів. Алгоритм A* найчастіше використовується у розробці ігрових програмних застосунків, тому його було обрано для тестування. Алгоритм Дейкстри являє собою попередника алгоритму A*, який є модифікованим алгоритмом Дейкстри. Генетичні алгоритми обрані через можливість використання у змінних умовах, на задачах великої розмірності, але ці алгоритми дуже чутливі до налаштувань.

3.2.1 Реалізація алгоритму Дейкстри

Для реалізації алгоритму Дейкстри реалізовано клас Node (вузол). Скрипт цього класу додано компонентом на префаб вузла. Клас Node містить у собі поля ваги вузла, батьківського вузла, список сусідніх вузлів та поле

стану вузла (чи є він перешкодою, чи вільним для пересування). Поле ваги за замовченням встановлено у максимальне значення типу int. Поле батьківського вузла за замовченням дорівнює null, і має тип Transform (компонент у Unity, який у собі містить розташування об'єкту у тривимірному просторі, його кут повороту у просторі сцени, та розміри об'єкту у тривимірному просторі). Список сусідніх вузлів, типу Transform, за замовченням пустий. Поле walkable за замовченням встановлене у true, тобто кожен вузол є вільним для пересування. Також префаб отримує тег «Node».

Алгоритм Дейкстри реалізовано у методі DijkstrasAlgo. Метод приймає у себе два параметри – початковий вузол і кінцевий вузол. У методі DijkstrasAlgo є замір часу роботи методу, тобто самого алгоритму. Поле endTime, яке записує у себе час роботи алгоритму, обчислюється за допомогою віднімання від змінної realtimeSinceStartup класу Time, яка містить у собі час з моменту запуску додатку, змінної startTime, яка містить у собі час, коли почав працювати алгоритм Дейкстри.

Перед початком роботи алгоритму встановлюється масив nodes типу GameObject, потім у нього записується усі об'єкти з тегом «Node» та створюється список result типу Transform, який зберігатиме у собі вузли шляху.

На початку роботи алгоритму записується конкретний час, коли він почав працювати, та створюється список unexplored типу Transform, який потрібен, щоб зберігати у собі відкриті для пересування, але ще не досліджені вузли. При переборі списку nodes створюється об'єкт типу Node, який заповнюється вузлом, який міститься у списку nodes. Потім перевіряється, чи є вузол вільним для пересування. При умові, шо вузол є відкритим для шляху, значення ваги та батьківського вузла встановлюються за замовченням (приклад 3.6).

```
foreach (GameObject obj in nodes)
     {
         Node n = obj.GetComponent<Node>();
         if (n.IsWalkable())  {
```

```
                n.ResetNode();
                unexplored.Add(obj.transform);
            }
        }
```

Приклад 3.6 – Додання усіх прохідних вузлів до списку недосліджених

У список unexplored додаються вузли, які пройшли описані вище дії, тобто вони призначені для підготовки для побудови маршруту. Після цього створюється об’єкт, який містить у собі початковий вузол, і для початкового вузла встановлюється вага, яка дорівнює 0.

Поки кількість недосліджених вузлів перевищує нуль, список unexplored сортується. Початковий елемент списку недосліджених вузлів записується у об’єкт current типу Transform, та видаляється зі списку недосліджених вузлів. Потім створюється об’єкт currentNode типу Node, і в нього передається компонент Node об’єкту current (приклад 3.7).

```
      while (unexplored.Count > 0)
              {
                  unexplored.Sort((x,                    y)                    =>
x.GetComponent<Node>().GetWeight().CompareTo(y.GetComponent<Node>().GetWeight
()));
                  Transform current = unexplored[0];
                  unexplored.Remove(current);
                  currentNode = current.GetComponent<Node>();
                  List<Transform> neighbours = currentNode.GetNeighbourNode();
                  foreach (Transform neighNode in neighbours)
                  {
                      Node node = neighNode.GetComponent<Node>();
                      if (unexplored.Contains(neighNode) && node.IsWalkable())
                      {
                          float                   distance                   =
Vector3.Distance(neighNode.position, current.position);
                         distance = currentNode.GetWeight() + distance;
                          if (distance < node.GetWeight())
                          {
                              node.SetWeight(distance);
                              node.SetParentNode(current);
                          }
                      }
                  }
```

Приклад 3.7 – Пошук вузла, який створюватиме маршрут (файл DijkstraPathfinder.cs)

Створюється список neighbours типу Transform, у який додається список сусідніх вузлів поточного вузла. Перебираючи список сусідніх вузлів поточного вузла, створюється об’єкт node типу Node, у який почергово, при переборі, записується компонент Node вузла-сусіда. Потім перевіряється, чи містить список недосліджених вузлів у собі поточний вузол-сусід, та чи є цей вузол-сусід вільним для побудування маршруту. Завдяки цьому вимірюється частина дистанції від поточного вузла-сусіда до поточного вузла за допомогою методу Distance стандартної структури Unity – Vector3.

Обчислюється дистанція від поточного вузла-сусіда до поточного вузла як сума ваги поточного вузла і значення частини дистанції. Якщо дистанція менша за вагу поточного вузла-сусіда, то вага поточного вузла-сусіда встановлюється значенням дистанції та його батьківським вузлом встановлюється поточний вузол. Після пройдених дій отримується вузол шляху і додається до списку result.

Відображення шляху на лабіринті відбувається у методі Update (стандартний метод життєвого циклу програмного застосунку на Unity, який опрацьовує кожний фрейм). При переборі вузлів, що склали маршрут, почергово зафарбовуються у жовтий колір вузли. Процес фарбування реалізовано за допомогою компоненту, який має кожний вузол, Renderer. Почергово отримується компонент Renderer кожного вузла, що склав маршрут і через властивість material змінюється значення поля color на потрібний колір. Клас Color містить у собі десять стандартних кольорів.

3.2.2 Реалізація алгоритму A*

Для реалізації алгоритму A* створено клас Node (вузол). Клас містить у собі поля-характеристики, властивості, метод порівняння вартостей вузла. Також міститься конструктор класу вузла (приклад 3.8). Конструктор у якості параметрів приймає: змінну _walkable типу bool, яка встановлює стан вузлу (true – вільний для маршруту, false – перешкода); змінну _worldPos типу

Vector3 (стандартна структура у Unity, яка містить у собі координати у трьох вимірах), яка встановлює позицію вузлу на сцені у тривимірному просторі; змінні _gridX та _gridY типу int, які містять інформацію про розмірність генерованого лабіринту.

```
public Node(bool _walkable, Vector3 _worldPos, int _gridX, int _gridY) {
      walkable = _walkable;
      worldPosition = _worldPos;
      gridX = _gridX;
      gridY = _gridY;
}
```

Приклад 3.8 – Конструктор класу Node (файл Node.cs)

Конструктор встановлює поля класу Node згідно переданим параметрам. Однією із властивостей, що містить клас Node, є властивість, у якій обчислюється довжина шляху до цілі (формула 3.1).

$$f(v) = g(v) + h(v), \tag{3.1}$$

Алгоритм A* реалізовано у методі FindPath. Метод приймає у якості параметрів змінні startPos і targetPos типу Vector3, які визначають початкову та кінцеву точки маршруту відповідно. Спочатку створюється масив waypoints типу Vector3, який буде зберігати у собі шлях для його обернення. Створюється змінна pathSuccess типу bool, яка показує стан шляху (true – маршрут побудовано, false – маршрут не побудовано) і за замовченням встановлюється у false. Наступними створюються об’єкти startNode і targetNode типу Node, які заповнюються значеннями початкової та кінцевої точками маршруту.

Після визначення основних потрібних об’єктів перевіряється умова – чи вільні для побудування маршруту початкова та кінцева точки маршруту. Якщо умова дотримана, створюються дві колекції openSet та closedSet типу Node. Колекція openSet створена для вузлів, які не досліджені. У свою чергу

колекція closedSet буде зберігати у собі досліджені вузли. Після створення колекцій у колекцію openSet додається початковий вузол. У наступному кроці перевіряється лічильник колекції openSet на умову його порожнечі (приклад 3.9).

```
      while (openSet.Count > 0) {
      Node currentNode = openSet.RemoveFirst();
      closedSet.Add(currentNode);

      if (currentNode == targetNode) {
            pathSuccess = true;
            break;
      }
      foreach (Node neighbour in grid.GetNeighbours(currentNode)) {
            if (!neighbour.walkable || closedSet.Contains(neighbour)) {
                                    continue;
                              }
            int     newMovementCostToNeighbour     =     currentNode.gCost     +
GetDistance(currentNode, neighbour);
            if     (newMovementCostToNeighbour     <     neighbour.gCost     ||
!openSet.Contains(neighbour)) {
                  neighbour.gCost = newMovementCostToNeighbour;
                  neighbour.hCost = GetDistance(neighbour, targetNode);
                  neighbour.parent = currentNode;

                  if (!openSet.Contains(neighbour))
                        openSet.Add(neighbour);
                  }
       }
```

Приклад 3.9 – Фрагмент метода FindPath (файл Pathfinding.cs)

Колекція openSet вже містить у собі початковий вузол, тому перевірка завершується із результатом true. Після проходження перевірки створюється об’єкт currentNode типу Node, що містить у собі вузол для дослідження. Вузол currentNode видаляється з колекції недосліджених вузлів openSet і додається у колекцію досліджених вузлів closedSet. Далі перевіряється, чи є досліджуваний вузол кінцевим вузлом маршруту.

Якщо досліджуваний вузол є кінцевим вузлом маршруту, алгоритм завершує роботу і стан pathSuccess встановлюється у true. Якщо умова не виконана виконується перебір сусідів досліджуваного вузлу та перевіряється, чи є перешкодою сусідній вузол, та чи міститься у колекції closedSet. Після перевірки створюється змінна newMovementCostToNeighbour типу int, що

дорівнює перерахованій вартості переміщення до сусіднього вузлу. Далі перевіряється, чи не міститься сусідній вузол у колекції openSet, та чи перерахована вага сусіднього вузлу newMovementCostToNeighbour менша за вартість шляху від сусіднього вузла до кінцевої точки маршруту. Якщо умова виконана, то у вартість шляху від сусіднього вузла до кінцевої точки маршруту записується перерахована вага newMovementCostToNeighbour. Далі у евристичне приближення вартості шляху від сусіднього вузлу до кінцевої точки записується дистанція від сусіднього вузлу до кінцевої точки маршруту за допомогою методу GetDistance.

У наступному кроці поле parent (батьківский вузол) заповнюється досліджуваним вузлом. Якщо сусідній вузол не міститься у колекції openSet, то він додається до колекції. Далі перевіряється, чи завершив свою роботу метод FindPath. Якщо умова виконана, то у масив waypoints записується обернений шлях. Сигналізується про завершення роботи алгоритму.

3.2.3 Реалізація генетичного алгоритму

Генетичний алгоритм має багато початкових налаштувань, тому першим кроком реалізації є створення усіх потрібних параметрів для налаштування роботи алгоритму (приклад 3.10).

```
public int populationSize = 140;
public double crossoverRate = 5f;
public double mutationRate = 0.001f;
public int chromosomeLength = 70;
public int geneLength = 2;

public int fittestGenome;
public double bestFitnessScore;
public double totalFitnessScore;
public int generation;
public bool busy;
```

Приклад 3.10 – Параметри налаштування і поля генетичного алгоритму (файл GeneticAlgorithm.cs)

Параметр populationSize відповідає за максимальну кількість популяцій у генетичному алгоритмі. За швидкість кросоверінгу відповідає параметр crossoverRate типу double, а швидкість мутації встановлюється параметром mutationRate типу double. Довжина хромосоми регулюється параметром chromosomeLength. Параметр geneLength типу int відповідає за довжину гену. Переліченим вище параметрам встановлені значення за замовчуванням. Наступний блок змінних потрібен у ході роботи алгоритму. Змінна fittestGenome типу int зберігатиме у собі найбільш пристосований геном. Кращий показник пристосованості зберігатиметься у bestFitnessScore, та загальний показник пристосованості зберігатиметься у totalFitnessScore. Лічильник поколінь generation типу int. Створено змінну busy типу bool, яка відповідає за стан роботи алгоритму (false – алгоритм не виконує роботу, true – алгоритм працює над пошуком найкоротшого шляху).

Також реалізований клас Genome, який є моделлю геному. У його склад входять список бітів, та змінна fitness, яка відповідає пристосованості того чи іншого геному. Конструктор класу у якому визивається метод Initialize. У методі встановлюються значення змінної fitness за замовчуванням у 0, та створюється пустий список бітів.

У класі GeneticAlgorithm створено конструктор, який встановлює значення зайнятості алгоритму у стан вільного від обчислень, створює список геномів, та список останніх генерованих геномів. Наступним реалізовано метод Mutate, який приймає у себе список бітів (приклад 3.11).

```
public void Mutate(List<int> bits) {

      for (int i = 0; i < bits.Count; i++) {

            if (UnityEngine.Random.value < mutationRate) {
                  bits [i] = bits [i] == 0 ? 1 : 0;
            }
      }
}
```

Приклад 3.11 Метод мутації (файл GeneticAlgorithm.cs)

У циклі, обмеженому лічильником бітів та за умови що число генероване UnityEngine.Random менше за параметр швидкості мутації, проходить обернення бітів, тобто мутація.

Реалізовано кросоверінг у методі Crossover, який приймає чотири параметри: два списки батьківських елементів, та два списки, для нащадків (приклад 3.12).

```
      public void Crossover(List<int> mom, List<int> dad, List<int> baby1, 
List<int> baby2) {
            if (UnityEngine.Random.value > crossoverRate || mom == dad) {
                  baby1.AddRange(mom);
                  baby2.AddRange(dad);

                  return;
            }
            System.Random rnd = new System.Random ();
            int crossoverPoint = rnd.Next (0, chromosomeLength - 1);
            for (int i = 0; i < crossoverPoint; i++) {
                  baby1.Add (mom [i]);
                  baby2.Add (dad [i]);
            }
            for (int i = crossoverPoint; i < mom.Count; i++) {
                  baby1.Add (dad [i]);
                  baby2.Add (mom [i]);
            }
                  }
```

Приклад 3.12 Метод кросоверу (файл GeneticAlgorithm.cs)

Проходить перевірка, чи генероване UnityEngine.Random більше швидкості кросоверінгу, та чи рівні між собою батьківські колекції. Якщо так, то батьківські колекції додаються у списки нащадків. Якщо ні, то створюється випадкове число і встановлюється точка кросоверінгу у хромосомі. Далі за допомогою циклів йде обмін генами, тобто елементами списків батьківських бітів. Метод RouletteWheelSelection містить у собі функціонал отримання найбільш пристосованих геномів і додання їх до списку genomes.

Для створення початкової популяції реалізовано метод CreateStartPopulation (приклад 3.13).

```
public void CreateStartPopulation() {
      genomes.Clear ();
      for (int i = 0; i < populationSize; i++) {
            Genome baby = new Genome (chromosomeLength);
            genomes.Add (baby);
      }
}
```

Приклад 3.13 Метод створення початкової популяції (файл GeneticAlgorithm.cs)

Спочатку очищується популяції, які генерувалися при запуску алгоритму раніше. У наступному кроці через цикл, обмежений кількістю популяцій, створюється геном із нульовою пристосованістю та пустим списком бітів, і додається до списку геномів.

Реалізовано метод UpdateFitnessScores для оновлення пристосованості хромосом, у якому встановлюються усі значення пристосованості у значення нуль, та перераховуються згідно декодованих бітів. Якщо пристосованість хромосоми дорівнює одиниці, то шлях знайдено, і робота алгоритму зупиняється і його стан встановлюється у вільний від роботи.

Відображення побудованого маршруту реалізовано у методі RenderFittestChromosomePath. Відображення відбувається завдяки створенню на сцені префабів червоного кольору, які створені для ілюстрації маршруту, та видаленню вільних для шляху префабів білого кольору, які знаходились на місці розміщеня префабів шляху.

3.3 Реалізація змінних умов

Насамперед розглянуто можливість зміни умов під час проходження маршруту у режимі реального часу. Вибір змінних умов включав у себе так варіанти, як зміна положення і позиції стінок-перешкод у лабіринті; зміна області зору персонажу (що призводить до зміни доступної інформації, яка використовується для побудови маршруту пересування); зміна позиції кінцевої точки маршруту.

При виборі можливої інтерпретації змінних умов проходження лабіринту, що аналізуються у даній роботі, серед розглянутих варіантів обрана необхідність враховувати при побудові маршруту можливості зміни позиції кінцевої точки маршруту під час його проходження.

Змінні умови такого типу найчастіше зустрічаються у ігрових програмних застосунках, а саме тому оптимізація їх урахування у роботі алгоритмів пошуку найкоротшого шляху буде найбільш ефективною для поліпшення швидкодії ігрових застосунків.

Для реалізації змінних умов під час проходження лабіринту, що представляє відкритий ігровий світ, у кожному з алгоритмів створено змінні, що відповідають кінцевій точці маршруту. Через різну реалізацію ігрових світів та різний принцип роботи алгоритмів, підхід до реалізації змінних умов у кожному з алгоритмів відрізняється.

Змінні умови для роботи застосунку, що досліджує використання алгоритму Дейкстри, реалізовано за допомогою створення змінних, які відповідають індексам у списку вузлів, які складають лабіринт. Тобто користувач має змогу задати індекси стартової і кінцевої точки маршруту і під час виконання алгоритму Дейкстри змінювати індекс кінцевої точки лабіринту.

Для алгоритму A* можливість зміни умов проходження лабіринту реалізовані схожим способом, різними є тільки данні, які відповідають за спосіб описання кінцевої точки маршруту, що будуються алгоритмом. Описання кінцевої точки маршруту у алгоритмі A* відбувається через осі ординат і абсцис.

Змінні умови для генетичного алгоритму створено завдяки реалізації зміни позиції кінцевої точки користувачем. Перед початком роботи алгоритму, коли згенеровано лабіринт, користувач може встановити початкову і кінцеву точки маршруту. Реалізовано можливість змінити кінцеву точку маршруту користувачем, записавши у поля типу Vector2 значення по осям ординат і абсцис.

Для компоненту NavMesh реалізовано зміну позиції під час виконання роботи пересування маршрутом за допомогою зчитування кліку лівої кнопки миші (приклад 3.14).

```
      void Update()
          {
              if (Input.GetMouseButtonDown(0))
              {
                  RaycastHit raycastHit;
                  if
(Physics.Raycast(mainCamera.ScreenPointToRay(Input.mousePosition),         out
raycastHit))
                  {
                      agent.SetDestination(raycastHit.point);
                  }
              }
          }
```

Приклад 3.14 Переміщення агенту NavMesh (файл PlayerControl.cs)

У методі Update (метод запускається кожний фрейм) перевіряється, чи нажата ліва кнопка миші, якщо так пускається промінь. Координати миші на екрані конвертуються у піксельні координати позиції за допомогою метода ScreenPointToRay та перевіряє, чи промінь перетинає колайдер (компонент Collider визначає форму об’єкту для фізичних взаємодій). Якщо промінь перетинає колайдер, спрацьовує метод SetDestination, який відповідає за рух агенту. У метод SetDestination передається точка, у котрій промінь перетнув колайдер і агент пересувається до неї.

4 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕНЬ

4.1 Дослідження роботи генетичного алгоритму

Генетичні алгоритми мають багато параметрів для налаштувань. Через це, перед початком дослідження та порівняльного аналізу роботи алгоритмів пошуку найкоротшого шляху у різних умовах, потрібно налагодити роботу генетичного алгоритму, для знаходження саме таких параметрів, що забезпечують можливість використання для вирішення задання та ефективну роботу алгоритму.

У даній роботі розроблене тестове програмне забезпечення надає змогу налагоджувати та аналізувати вплив наступних параметрів роботи генетичного алгоритму:

- розмір популяції;
- швидкість зміни популяції, яка налагоджується через встановлення вірогідності кросоверінгу для тих особин популяції, що приймають участь у схрещуванні;
- вірогідність мутації, при цьому ступінь впливу мутації на хромосому не корегується;
- довжина хромосоми.

4.1.1 Дослідження впливу розміру популяції

Для аналізу обрано випадковий лабіринт (рисунок 4.1), що описується графом із 165 вершин. Обчислено найкоротший маршрут, який містить у собі 19 вершин графу, від якого рахується похибка. Досліджувалися час роботи та величина похибки отриманого рішення при заданому числі поколінь для популяцій різного розміру.

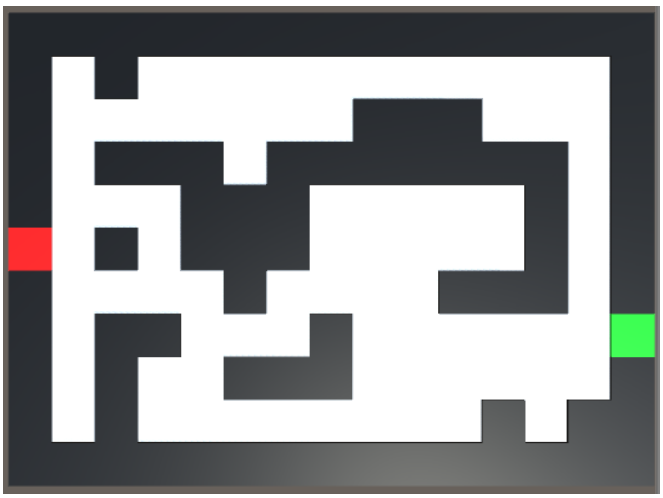

Рисунок 4.1 – Лабіринт для дослідження ефективності та візуалізації впливу налаштувань генетичного алгоритму

Тестування проводилося з генетичними алгоритмами, популяції обсягом від 100 до 1000 особин з кроком у 100 особин (таблиця 4.1).

Таблиця 4.1 – Залежність часу роботи генетичного алгоритму від кількості особин у популяції

Кількість особин у популяції	Час роботи алгоритму, с	Кількість вершин у маршруті	Величина похибки у вершинах
100	1,083	21	2
200	1,417	21	2
300	1,968	23	4
400	2,348	23	4
500	2,167	25	6
600	3,123	19	0
700	3,634	21	2
800	3,823	25	6
900	5,538	21	2
1000	4,730	19	0

Дослідження показали (рисунок 4.2), що при розмірі популяції менше 100 особин алгоритм потребує збільшеного часу на збіжність, при цьому вірогідна збіжність популяції до локального мінімуму. Для отримання доцільного результату, необхідна популяція від 100 особин.

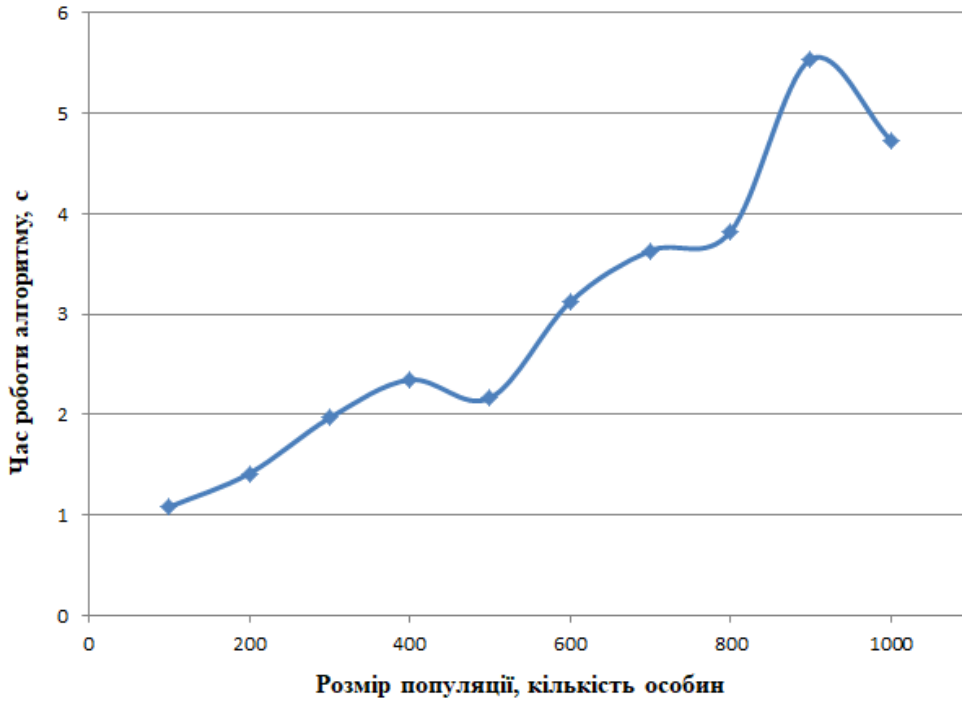

Рисунок 4.2 – Графік залежності часу роботи алгоритму від кількості особин у популяції

Результати дослідження показали, що час роботи алгоритму лінійно залежить від кількості особин у популяції, за деякими виключеннями. Також при більшості досліджених кількостях особин у популяції, генетичний алгоритм припускав похибку від 9,5% до 17,39%. Однак при популяції, яка складала 600 та 1000 особин, маршрут будувався без похибок, досягаючи результатів, аналогічних еталонному маршруту. Тому для подальшого аналізу роботи генетичного алгоритму було обране налаштування значення обсягу популяції у 600 особин, через менший час роботи пошуку найкоротшого шляху алгоритму ніж при обсязі популяції у 1000 особин.

4.1.1 Дослідження впливу вірогідності кросоверінгу

Кросовер alternating edges будує нащадків, випадково вибравши ребро від першого батька, потім наступне ребро від другого батька, потім знову наступне від першого і т.д. Якщо нове ребро являє замкнутий цикл, береться випадковий кут з того ж батька, який ще не вибирався та не утворює замкненого циклу.

Для прикладу, один з нащадків батьків:

P1 = (2 3 8 7 9 4 1 5 6)
P2 = (7 5 1 6 9 2 8 4 3)

Може бути П1 = (2 5 8 7 9 1 6 4 3), де процес починався від вузла (1,2) батька P1, продовжуючи до кута (7,8), замість якого обраний вузла (7,6), оскільки тут вершина 8 вже була включена в маршрут.

Кросовер subtour chunks створює нащадків, вибираючи випадково фрагмент маршруту від одного з батьків, потім фрагмент маршруту випадкової довжини від іншого з батьків, і т.д. Якщо на якомусь етапі утворюється замкнутий цикл, він вирішується аналогічним чином.

Heuristic crossover будує нащадків, вибираючи випадкове місце як стартову точку для маршруту-нащадка. Потім він порівнює два відповідних ребра від кожного з батьків і вибирає більш коротке. Потім кінцевий вузол вибирається як початковий для вибору наступного більш короткого ребра з цього вузла. Якщо на якомусь етапі виходить замкнутий маршрут, побудова шляху триває будь-яким випадковим вузлом, який ще ніколи не відвідували.

Недоліки розглянутих кросоверів для даного методу подання наступні: кросовер alternating edges часто руйнує хороші тури, які були у обох батьків до застосування цієї операції. Кросовер subtour chunks має кращі характеристики, ніж перший, завдяки тому, що його руйнівні властивості менше, але все одно його експлуатаційні якості досить низькі. Кросовер

heuristic crossover – найкращий для даного подання, так як він враховує довжину ребр, але, з іншого боку, може вийти так, що подальший маршрут буде неможливий і доведеться вибирати ребро, довжина якого невиправдано велика – така ситуація є наслідком застосування «жадібної» евристики.

Вплив вірогідності кросоверінгу на час роботи пошуку найкоротшого шляху алгоритмом досліджувався шляхом перевірки роботи алгоритму при при значенні вірогідності кросоверінгу у діапазоні від 0.1 до 1 (таблиця 4.2).

Таблиця 4.2 – Залежність часу роботи генетичного алгоритму від вірогідності кросоверінгу

Вірогідність кросоверінгу	Час роботи алгоритму, с	Кількість вершин у маршруті	Похибка (у вершинах)
0,1	3,218	21	2
0,2	3,315	21	2
0,3	3,170	21	2
0,4	3,506	19	0
0,5	3,258	23	4
0,6	3,352	21	2
0,7	3,105	21	2
0,8	3,233	21	2
0,9	3,144	21	2
1,0	3,325	21	2

Дослідження впливу вірогідності кросоверінгу на час роботи генетичного алгоритму показали, що вірогідність кросоверінгу незначно впливає на час збіжності (рисунок 4.3), при цьому забезпечує невелику похибку майже в усіх побудованих маршрутах при досліджуваних значеннях.

Для подальшого налаштування генетичного алгоритму обрано величину вірогідності кросоверінгу у 0,4, через те, що при такій величині

генетичний алгоритм будує маршрути без похибок та не збігається до квазіоптимального рішення.

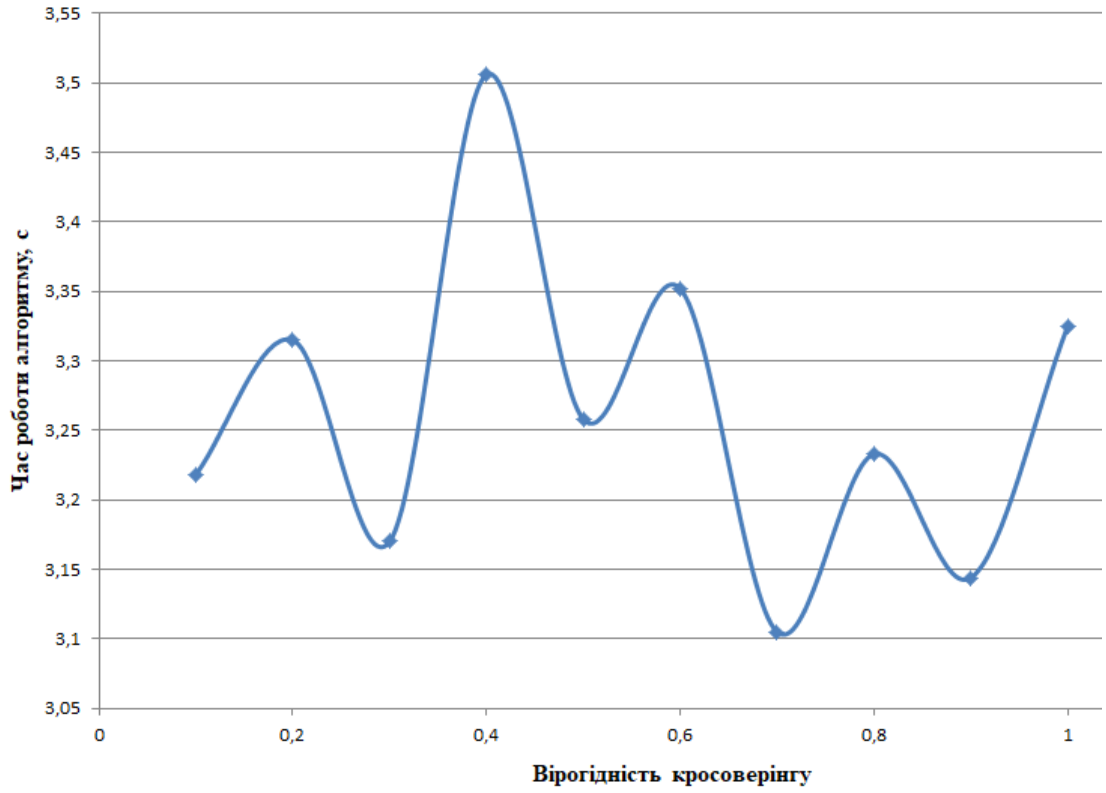

Рисунок 4.3 – Графік залежності часу роботи алгоритму від вірогідності кросоверінгу

Також експеримент з налаштуванням вірогідності кросоверінгу проводився при значеннях від 1 до 10. Експеримент продемонстрував незначний вплив на час роботи алгоритму при обраних значеннях.

4.1.2 Дослідження впливу вірогідності мутації

Одноточкова мутація, яка застосовується в класичній моделі генетичного алгоритму, при певних умовах виявляється неефективною. Наприклад, при великій кількості вершин графа.

У одноточковій мутації випадковим чином вибираються два гена, які

потім міняються місцями. Запропоновано спрощений варіант цієї мутації – випадковим чином вибирається один ген (що сприяє збільшенню швидкодії), і змінюється місцем з сусіднім геном.

Вплив вірогідності мутації на час роботи алгоритму та величину похибки знайденого маршруту досліджувався у діапазоні значень вірогідності мутації від 0.01 до 0.09 (таблиця 4.3).

Таблиця 4.3 – Залежність часу роботи генетичного алгоритму від вірогідності мутації

Вірогідність мутації	Час роботи алгоритму, с	Кількість вершин у маршруті	Похибка (вершин)
0,01	3,332	20	1
0,02	4,391	21	2
0,03	3,216	23	4
0,04	3,207	23	4
0,05	3,048	21	2
0,06	3,140	21	2
0,07	3,482	20	1
0,08	3,298	21	2
0,09	3,239	21	2
1,0	4,018	19	0

Як показали результати випробувань, при низькій вірогідності мутації наявна велика похибка, а у деяких випадках – результат не знайдені зовсім, тобто алгоритм не зійшовся до екстремума поверхні рішень.

Переміщення двох сусідніх генів при високій ймовірності мутації призводить до погіршення популяції і повільної збіжності алгоритму. Таким чином, даний метод вкрай неефективний, хоча і володіє швидкодією більшим, ніж у однокрапкової мутації.

У подальшому досліджувалась робота алгоритму для пошуку

найкоротшого маршруту при використанні гіпермутації – якщо вірогідність приймає значення 1, тобто мутації підлягають усі особини в популяції.

4.1.3 Дослідження впливу довжини хромосоми

Довжина хромосоми вказує, яка максимальна кількість вузлів можлива у знайденому маршруті. Таким чином, для невеликих графів доцільне більш обмежене значення, ніж для графів великої розмірності. Досліджувався вплив довжини в інтервалі від 10 до 120 вузлів (таблиця 4.4).

Таблиця 4.4 – Залежність часу роботи генетичного алгоритму від вірогідності мутації

Довжина хромосоми	Час роботи алгоритму, с
50	3,9944
60	4,0972
70	3,7754
80	4,02
90	4,4402
100	4,8696
110	5,1983
120	5,8019

В межах довжини хромосоми від 10 до 40 рішення не знаходяться, через неможливість виходу з локального екстремума, тобто алгоритм зовсім не прокладає маршрут. Обмеження від 50 до 90 вузлів дозволяє досягати низької вірогідності знайти найкоротший шлях, тобто рішення або не буде знайдено, або потребує значного числа поколінь популяції для виходу з локального екструмума та досягнення оптимального маршруту, що проходить через мінімально можливу кількість вершин. ля подальшого дослідження роботи алгоритму пошуку найкоротшого маршруту у

лабіринтах у стаціонарних та змінних умовах обмеженням максимальної довжини хромосоми прийнято для налаштування від 120 вузлів.

4.2 Дослідження роботи у статичних умовах

Для аналізу обрано лабіринти (рисунок 4.4), які генерувалися випадковим чином, що описуються графами розмірністю від 100 вершин до 1000. Експерименти проводилися на графах різної розмірності з кроком у 100 вершин. Дослідження роботи алгоритму Дейкстри (таблиця 4.5) показали, що для розрахунку маршруту потрібно від 0,0017 секунд до 1,18 секунд.

Таблиця 4.5 – Залежність часу роботи від розмірності графу у алгоритмі Дейкстри

Розмірність графу	Час роботи алгоритму, с
100	0,0017
200	0,0168
300	0,0404
400	0,0534
500	0,2218
600	0,3138
700	0,4554
800	0,6498
900	0,9696
1000	1,1878

Експеримент показав лінійну залежність між часом пошуку маршруту за допомогою алгоритму Дейкстри та величиною лабіринтів, які описувався графами різної розмірності (рисунок 4.4). Також слід відмітити, що починаючи з графів розмірністю вище 300 вершин, алгоритм Дейкстри демонстрував у знайдених маршрутах значну похибку, а починаючи з

кількості і від 600 вершин на графах, не завжди мав спроможність знаходити маршрути.

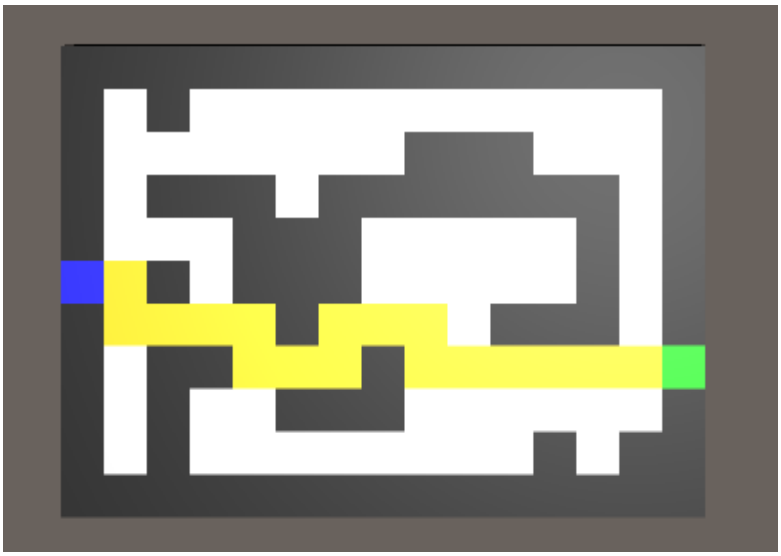

Рисунок 4.4 – Відображення роботи алгоритму на тестовому лабіринті

Таблиця 4.6 – Залежність часу роботи від розмірності графу у алгоритмі A*

Розмірність графу	Час роботи алгоритму, с
100	0,8759
200	1,6466
300	3,2335
400	4,6378
500	6,3562
600	7,9114
700	9,6091
800	11,0409
900	12,7789
1000	14,8183

Дослідження роботи алгоритму A* (таблиця 4.6) показали, що для розрахунку найкоротшого маршруту, час роботи складає від 0,87 секунд до

14,81 секунд відповідно до розмірності графів, які описують тестові лабіринти.

Не дивлячись на значно більший час на розрахунок маршруту, можна також помітити лінійну залежність між часом пошуку маршруту за допомогою алгоритму A* та розмірністю лабіринту (рисунок 4.5).

Для роботи генетичного алгоритму потрібне проходження декількох поколінь, що значно збільшує час на пошук маршруту (рисунок 4.5), але дозволяє найти маршрут з меншою похибкою (таблиця 4.7).

Таблиця 4.7 – Залежність часу роботи від розмірності графу у генетичному алгоритмі

Розмірність графу	Час роботи алгоритму, с
100	1,1768
200	12,00
300	13,907
400	11,438
500	13,524
600	12,544
700	20,775
800	26,217
900	32,259
1000	114,681

Згідно графіку залежності часу роботи алгоритмів від розміру лабіринту (рисунок 4.5), найгірші показники швидкодії показав генетичний алгоритм. Але при цьому на лабіринтах, що описуються графами від 400 вершин, незважаючи на більший час роботи, генетичний алгоритм знайшов найкоротші маршрути з мінімальною кількістю похибок, на відміну від алгоритмів Дейкстри та A*.

Проведено додаткове дослідження на лабіринті, що описується графом з 10000 вершин (рисунок 4.6).

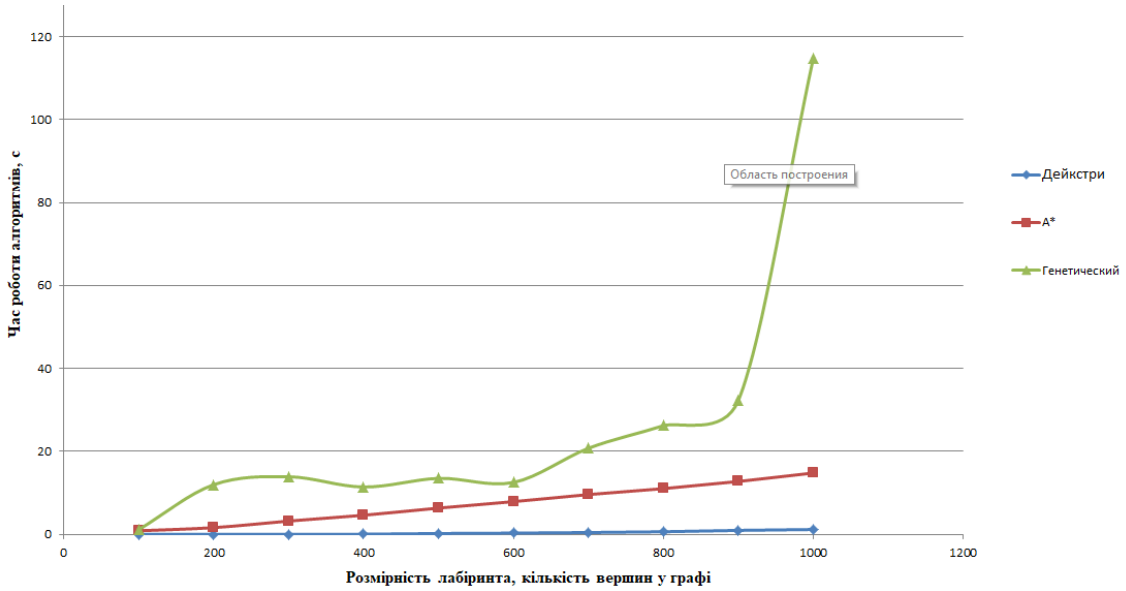

Рисунок 4.5 – Графік залежності часу роботи алгоритмів від розмірності графу у статичних умовах

В результаті роботи генетичного алгоритму знайдено найкоротший маршрут, тобто кінцеве рішення, яке використовувалось у якості еталону при розрахунку похибки (рисунок 4.6).

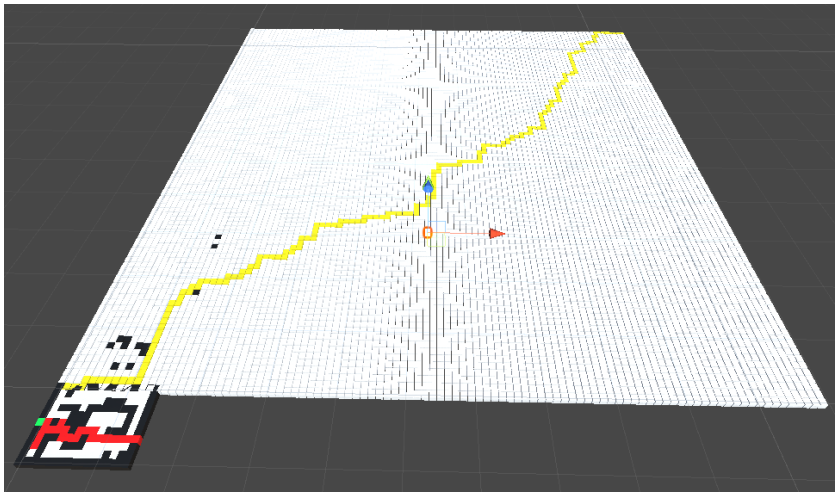

Рисунок 4.6 – Візуалізація роботи генетичного алгоритму

4.3 Дослідження роботи у змінних умовах

Змінні умови реалізовані у вигляді зміни кінцевої точки маршруту. Зміна кінцевої точки реалізована у програмному коді, а не шляхом встановлення користувачем. Зміна позиції відбувається кожні 0,2 секунди. Дослідження проводилися з однією зміною позиції. Позиції для зміни в усіх тестових лабіринтах однакові. Кількість секунд, при яких відбувається зміна кінцевої точки, незалежна. Виконання цих умов дозволяє досліджувати алгоритми, не втручаючись у їх швидкість роботи, у статичних умовах. Тобто при зміні кінцевої точки маршруту останні вершини у маршрутах обходу, що побудовані за допомогою різних алгоритмів можуть бути на різній відстані від попередньої цілі у один момент часу. Через це, агенти, що керуються алгоритми з менш швидким пошуком, можуть продовжувати рухатись до іншої цілі, майже не міняючи траєкторію. В свою чергу, більш швидкодіючі алгоритми можуть опинитися у ситуації, коли потрібно повертатися назад.

Таблиця 4.8 – Залежність часу роботи від розмірності графу у алгоритмі Дейкстри у змінних умовах

Розмірність графу	Час роботи алгоритму, с
100	0,4117
200	0,4268
300	0,4504
400	0,4634
500	0,6318
600	0,7238
700	0,8654
800	1,0598
900	1,3796
1000	1,6066

У результаті досліджень роботи алгоритму Дейкстри для пошуку найкоротшого шляху у змінних умовах отримано лінійну залежність часу пошуку маршруту алгоритмом від кількості вершин, що складають граф (таблиця 4.8).

Тестування пошуку найкоротшого шляху у змінних умовах за допомогою алгоритму Дейкстри показали, що на графах, розмірність яких перевищує 300 вершин, побудовані маршрути мають похибки у діапазоні від 15,4% до 67,5% відповідно величині кількості вершин у графі. Це вказує на недоцільність використання алгоритму Дейкстри у змінних умовах на лабіринтах, які описуються графами кількістю від 300 вершин.

Таблиця 4.9 – Залежність часу роботи від розмірності графу у алгоритмі A* у змінних умовах

Розмірність графу	Час роботи алгоритму, с
100	1,4157
200	3,4978
300	4,4054
400	6,4784
500	7,2647
600	9,2375
700	11,3478
800	13,1752
900	16,2736
1000	19,3494

Дослідження роботи пошуку найкоротшого шляху у змінних умовах алгоритмом A* показало лінійну залежність часу пошуку маршруту алгоритмом від кількості вершин, що складають графи (таблиця 4.9). При тестуванні роботи пошуку найкоротшого шляху алгоритму A* відмічається більш тривалий час на пошук найкоротшого маршруту ніж у алгоритму

Дейкстри. Але на графах, розмірність яких перевищує 300 вершин, побудовані алгоритмом A* маршрути мають меншу кількість похибки ніж маршрути знайдені за допомогою алгоритму Дейкстри. Діапазон похибки склав від 3,3% до 39,6% відповідно величині кількості вершин у графі.

Таблиця 4.10 – Залежність часу роботи від розмірності графу у генетичному алгоритмі у змінних умовах

Розмірність графу	Час роботи алгоритму, с
100	5,7531
200	21,3754
300	27,3417
400	21,5584
500	27,1794
600	22,1432
700	39,7562
800	53,7809
900	63,9376
1000	234,3567

При дослідженні роботи генетичного алгоритму (таблиця 4.10), виявлено, що залежність часу роботи алгоритму від кількості вершин у розмірності графу є нелінійною.

Отримані при дослідженні пошуку найкоротшого шляху у лабіринті у змінних умовах за допомогою генетичного алгоритму, результати демонструють повільну швидкодію роботи генетичного алгоритму в порівнянні з алгоритмами A* та Дейкстри (рисунок 4.7). Перевага генетичного алгоритму демонструється у величині похибки при побудуванні найкоротшого шляху.

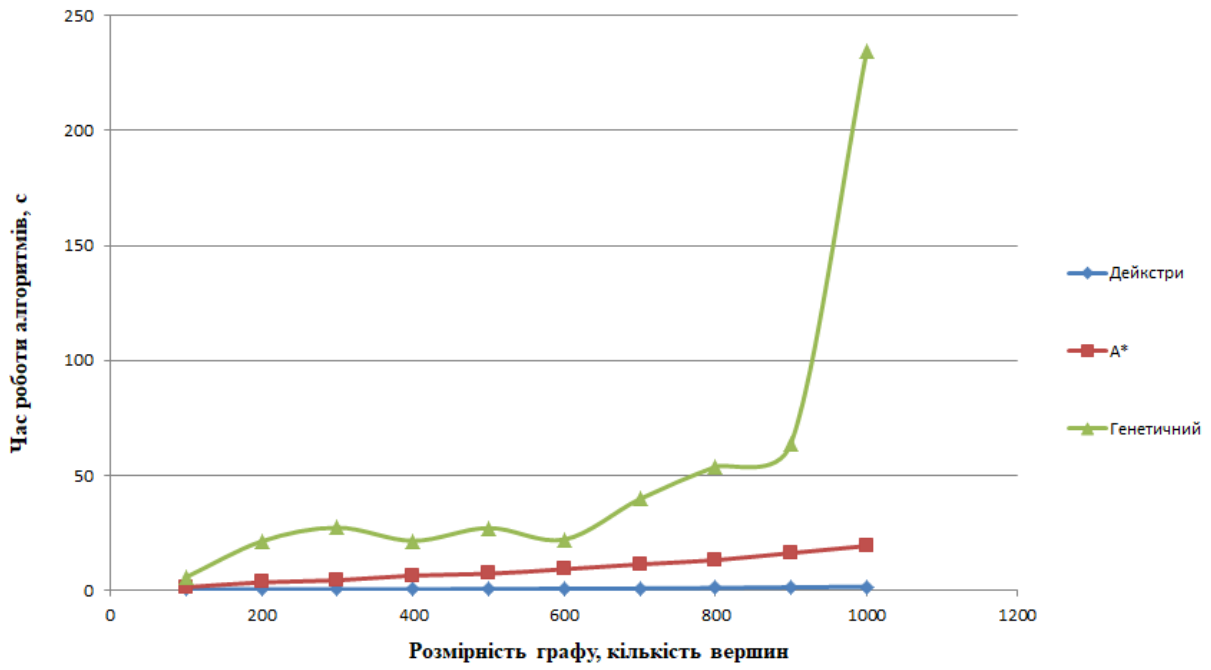

Рисунок 4.7 – Графік залежності часу роботи алгоритмів від розмірності графу у змінних умовах

На відміну від алгоритмів Дейкстри та A*, генетичний алгоритм містить невелику похибку у побудованому маршруті на графах великої розмірності, або похибка зовсім відсутня на графах з кількістю вершин до 600.

При проведені досліджень на розмірності графів від 3000 до 10000 вершин генетичний алгоритм упорався із задачею пошуку найкоротшого шляху з невеликою похибкою на відміну від алгоритмів Дейкстри та A*, які знаходили найкоротший маршрут з великою похибкою, або зовсім його не знаходили. Згідно результатів досліджень генетичного алгоритму у змінних умовах, доцільним є використання генетичного алгоритму на великих ігрових світах, які описуються графами великої розмірності.

ВИСНОВКИ

В даній роботі досліджено ефективність алгоритмів пошуку найкоротшого шляху у змінних умовах у лабіринтах, що описуються графами різної розмірності, для застосування при розробці ігрового штучного інтелекту неігрових персонажів. Розроблено застосунок для тестування алгоритмів пошуку найкоротшого шляху на платформі розробки ігор Unity 3D з використанням компоненту Navmesh. У дослідженні розглядалися такі алгоритми пошуку найкоротшого шляху, як алгоритм Дейкстри, алгоритм A* та генетичний алгоритм.

Проведено аналіз сучасного стану проблеми та огляд існуючих рішень реалізації штучного інтелекту для неігрових персонажів для вирішення задачі переміщення їх по ігровому світу, що описуються у вигляді лабіринтів, за допомогою алгоритмів пошуку найкоротшого шляху на графах різної розмірності.

Наукова новизна полягає у застосуванні алгоритмів для вирішення задачі пошуку найкоротшого шляху у змінних умовах, які ускладнюють задачу знаходження найкоротшого маршруту. При цьому реалізація роботи алгоритмів у змінних умовах поліпшує імітацію інтелекту у неігрових персонажів, що сприяє зануренню користувача у ігровий процес.

Для тестування та налагодження алгоритмів реалізований графічний користувальницький інтерфейс, що дозволяє формувати тестові вхідні дані і оцінювати ефективність застосування вищеописаних алгоритмів.

Налаштування генетичного алгоритму для пошуку найкоротшого шляху показали його невелику швидкодію, але спроможність знаходити маршрути у графах великої розмірності (3000 вершин і більше). Під час експерименту проводилося налаштування таких характеристик, як кількість особин у популяції, вірогідність кросоверінгу, вірогідність мутації та розмір хромосом.

Жадібні алгоритми є найбільш швидким, проте мають велику похибку у знайдених маршрутах. При тестуванні жадібних алгоритмів у статичних та змінних умовах, алгоритм Дейкстри показав кращу швидкодію, але при використанні його на великих лабіринтах, описаних графами великої розмірності, маршрути мали велику похибку, або зовсім не були знайдені. Майже аналогічні результати були показані алгоритмом A*, але на відміну від алгоритму Дейкстри він демонстрував меншу швидкодію при меншій похибці у знайдених маршрутах. Алгоритми продемонстрували лінійну залежність часу роботи від розмірності лабіринтів, які описувались графами з різної кількістю вершин. Тому використання алгоритмів Дейкстри та A* є доцільним при невеликих ігрових світах, що описані графами невеликої розмірності. Якщо притримуватись цього, буде досягнена швидкодія пошуку найкоротшого шляху при відсутності похибки, або невеликій похибці. Через це неігрові персонажі будуть мати більш природну поведінку.

Генетичний алгоритм у дослідженні продемонстрував невелику швидкодію, але забезпечував можливість знаходити маршрути у лабіринтах, що описуються графами різної розмірності. При тестуванні алгоритмів на графах розмірністю більше трьох тисяч величин спроможним знаходити маршрути був лише генетичний алгоритм, який у побудованих маршрутах мав невелику похибку. При розробці ігрового програмного застосунку генетичний алгоритм слід застосовувати на великих ігрових світах, або при імітації штучного інтелекту неігрового персонажу, який знаходиться у пошуках шляху у реальному часі у змінних умовах.

Практичне значення роботи полягає в розробці елементів ігрового штучного інтелекту для неігрових персонажів, що забезпечує пересування по ігровим світам з урахуванням зміни умов у реальному часі, через це поведінка персонажів виглядає більш природньою для гравця.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Шелл, Дж. Геймдизайн. Как создать игру, в которую будут играть все [Текст] / Дж. Шелл : Геймдизайню – 2019. – 640 с.

2. Хокинг, Дж. Unity в действии [Текст] / Дж. Хокинг – П. : Мультиплатформенная разработка на C#. – 2016. – 336 с. – ISBN 978-5-4461-0816-9.

3. Гибсон, Б. Дж. Unity и C#. Геймдев от идеи до реализации [Текст] / Б. Дж. Гибсон. – П. : Unity 3D. – 2016. – 928 с. – ISBN 978-5-4461-0715-5.

4. Aswathi, N. Maze solving automatons for self-healing of open interconnects: Modular add-on for circuit boards [Текст] / N. Aswathi, K. Raghunandan, S. S. Pillai, S. Sanijv // Applied Physics Letters. – 2015. – Vol. 106, №12. – Pp. 34-39.

5. Уилсон, Р. Введение в теорию графов [Текст] / Р. Уилсон. – Диалектика : Теория графов. – 2019. – 240 с. – ISBN 978-617-7812-39-4.

6. Акімов, О. Е. Дискретна математика. Логіка, групи, графи [Текст] / – М. : Лабораторія базових знань. – 2003. – 376 с.

7. Макаровских, Т. А. Комбинаторика и теория графов [Текст] / Т. А. Макаровских – Ленанд : Теория графов. – 2016. – 216 с. – ISBN 978-5-9710-3793-4.

8. Лекція 9. Пошук найкоротшого шляху у графі [Електронний ресурс] – Режим доступу : www/ URL: https://www.slideshare.net/mkurnosov/9-33206496 – 12.11.2020 г. – Загол. з екрану.

9. BolloBas, B. Modern Graph Theory [Текст] / B. BolloBas : Graduate Texts in Mathematics. – 1998. – 387 с. – ISBN 978-0-387-98488-9

10. Посібник з теорії графів [Електроний ресурс] – Режим доступу : www/ URL: https://infourok.ru/posobie-po-teorii-grafov-3550951.html – 26.10.2020 г. – Загол. з екрану.

11. Танянский, С. С. Об одном решении задачи нахождения оптимального маршрута перевозок разнородного продукта с учетом

ограничения времени доставки [Текст] / С. С. Танянский, Г. С. Иващенко, Е. В. Леонова // Управління розвитком. Харківський національний економічний університет. Міжнародна науково-практична конференція «Стратегії ІТ-технологій в освіті, економіці та екології». – 2007. – Т. 1, №7. – С. 18-20.

12. Аксак, Н. Г. Алгоритм нахождения минимального пути с временными ограничениями [Текст] / Н. Г. Аксак, Г. С. Иващенко, Е. В. Леонова // Комп'ютерні інтелектуальні системи та мережі. Матеріали Всеукраїнської студентської конференції. – Кривий Ріг: Криворізький технічний університет. – 2009. – С. 64-66.

13. Кораблев, Н. М. Применение гибридного иммунного алгоритма для решения задачи коммивояжера [Текст] / Н. М. Кораблев, Г. С. Иващенко // 2-я международная научно-техническая конференция «Информационные технологии в навигации и управлении: состояние и перспективы развития». – 2011. – С. 38.

14. Кораблев, Н. М. Использование агентно-ориентированного подхода для решения задачи коммивояжера [Текст] / Н. М. Кораблев, Г. С. Иващенко, М. В. Кушнарев // 9-я международная конференция «Математическое и программное обеспечение информационных систем». – 2011. – С. 137-138.

15. Кораблев, Н. М. Агентно-ориентированный подход на основе искусственных иммунных систем для решения задачи коммивояжера [Текст] / Н. М. Кораблев, Г. С. Иващенко, М. В. Кушнарев // Бионика интеллекта. – 2012. – Вып. 2 (79). – С. 33-37.

16. Кораблев, Н. М. Периодичность характеристик агентов искусственной иммунной системы, используемой для решения задачи коммивояжера [Текст] / Н. М. Кораблев, Г. С. Иващенко // 16-й международный молодежный форум «Радиоэлектроника и молодежь в XXI ст.». – 2012. – Т. 10. – С. 37-38.

17. Іващенко, Г.С Проблема проходження лабіриннту в умовах, що змінюються [Текст] / Г.С. Іващенко, О.С. Павленко // Управління розвитком. Харківський національний економічний університет. Проблеми

інформатизації: Матеріали восьмої міжнародної науково-технічної конференції. – 2020 – С. 15.

18. Permana, S. D. H. Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, and BFS on Maze Runner Game [Текст] / Silvester Dian Handy Permana, Ketut Bayu Yogha Bintoro, Budi Arifitama, Ade Syahputra // International Journal Of Information System & Technology. – 2018. – Vol. 1, №2. – Pp. 1–8.

19. Wengrow, J. A Common-Sense Guide to Data Structures and Algorithms [Текст] / J. Wengrow. – Pragmatic Bookshelf. – 2020. – 508 с. – ISBN 978-1680607225.

20. Rohringer, W. Stochastic optimization of a cold atom experiment using a genetic algorithm Game [Текст] / W. Rohringer, R. Bucker, S. Manz, T. Betz, A. Perrin // Applied Physics Letters. – 2008. – Vol. 93, №26. – Pp. 93-116.

21. Sangdani, M. H. Genetic algorithm-based optimal computed torque control of a vision-based tracker robot: Simulation and experiment [Текст] / M. H. Sangdani, A. R. Tavalopour-Saleh // Engineering Applications of Artificial Intelligence. – 2018. – Vol. 67, №2. – Pp. 24-38.

22. Buckland, M. Programming game AI by example 1st Edition [Текст] / M. Buckland. – M. : Game AI. – 2004. – 495 с. – ISBN 978-1556220784.

23. Millington, I. Artificial Intelligence for Games [Текст] / I. Millington, J. Funge – M Artificial Intelligence for Games, 2006. – 845с.

24. Искусственный интеллект: современный подход [Текст] / П. Норвиг, Рассел С. – М. : Искусственный интеллект, 2007. – 1408 с.

25. Паласіос, Х. UNITY 5 Х. Програмування штучного інтелекту у іграх [Текст] / Х. Паласіос – М. : UNITY 5 Х, 2017. – 272 с.

26. Unity Documentation [Електронний ресурс] – Режим доступу : www/ URL: https://docs.unity3d.com/Manual/index.html/ - 9.11.2020 г. – Загол. з екрану.

27. Алгоритми та методи [Електроний ресурс] – Режим доступу : www/ URL: http://algolist.manual.ru – 26.10.2020 г. – Загол. з екрану.

28. Needham, M. Graph Algorithms: Practical Examples in Apache Sparks and Neo4j [Текст] / M. Needham. - O'Reilly Media : Graph Algorithms, 2019. – 268 с. – ISBN 978-1492047681.

29. Алгоритми – частина 1. Жадібні алгоритми, алгоритм Дейкстри [Електроний ресурс] – Режим доступу : www/ URL: https://webdevblog.ru/zhadnye-algoritmy-chast-1-algoritm-dejkstry/ – 12.11.2020 г. – Загол. з екрану.

30. Рассел, Дж. Алгоритм Дейкстры [Текст] / Дж. Рассел. – Bookvika : Алгоритм Дейкстры, 2012. – 112 с. – ISBN 978-5-5128-6896-6.

31. Hwang-il, K. Path Planning Algorithm Using the Particle Swarm Optimization and the Improved Dijkstra Algorithm [Текст] / K. Hwang-il, B. Lee, K. Kim. // 2008 IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application. – 2008. – Pp. 20. – ISBN 978-0-7695-3490-9.

32. Swathinka, O. V. G. Prims-Aided Dijkstra Algorithm for Adaptive Protection in Microgrids [Текст] / O. V. G. Swathinka. // IEEE Journal of Emerging and Selected Topics in Power Electronics. – 2016. – Pp 1279-1286.

33. Bell, M. G. H. Hyperstar: A multi-path Astar algorithm for risk averse vehicle navigation [Текст] / M. G. H. Bell. // Transportation Research Part B: Methodological. – 2019. – Vol. 43, №1. – Pp. 97–107.

34. Xiang, L. A comparative study of A-star algorithms for search and rescue in perfect maze [Текст] / L. Xiang, D. Gong // 2011 International Conference on Electric Information and Control Engineering. – 2011. – Pp. 134-142. – ISBN 978-1-4244-8036-4.

35. Pratyaksa, O. N. S. Optimized A-Star algorithm in hexagon-based environment using parallel bidirectional search [Текст] / O. N. S. Pratyaksa // 2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE). – 2016. – Pp. 87-103. – ISBN 978-1-5090-4140-4.

36. Chaudhari A. M. Improved A-star Algorithm with Least Turn for Robotic Rescue Operations [Текст] / Ashok M. Chaudhari, Minal R. Apsangi, Akshay B. Kudale // International Conference on Computational Intelligence,

Communications, and Business Analytics. – 2017. – Pp. 45-56. – ISBN 978-981-10-6429-6.

37. Гладков, Л. Генетические алгоритмы [Текст] / Л. Гладков, В. Курейчик. – М. : ФИЗМАЛИТ, 2010. – 368 с. – ISBN 978-5-9221-0510-1.

38. Вирсански, Э. Генетические алгоритмы на Python [Текст] / Э. Вирсански. – ДМК Пресс : Искуственный интелект, 2020. – 286 с. – ISBN 978-5-97060-857-9.

39. Mathew, T V. Genetic Algoritm [Текст] / T. V. Mathew // Department of Civil Engineering, Indian Institute of Technology Bombay. – 2017. – №4. – Pp. 1-15.

40. Бібліотека для реалізації генетичних алгоритмів у .NET Framework 2.0 [Електронний ресурс] – Режим доступу : www/ URL: http://jenyay.net/index.php?n=Programming.Genetic#theory. – 05.12.2020 г. – Загол. з екрану.

41. Choubey, N. S. A-Mazer with Genetic Algorithm [Текст] / N. S. Choubey // International Journal of Computer Applications. – 2018. – Vol. 58, №17. – Pp. 48-54.

42. Costa, A. A. A computational model for exploratory activity of rats with different anxiety levels in elevated plus-maze [Текст] / A. Costa, S. Morato, A. C. Roque, R. Tinos // Journal of Neuroscience Methods. – 2014. – Vol. 236, №1. – Pp. 44-50.

43. Vose, M. D. The Simple Genetic Algorithm: Foundations and Theory [Текст] / M. D. Vose. – The MIT Press. : Genetic Algorithm, 1999. – 220 с. – ISBN 978-0262220583.