Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи генерації та проходження лабіринтів

(тема)

Виконав:

студент II курсу, групи СПм-23-1

Вітко В.О.
(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)

2025 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 КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Вітко Владиславу Олександровичу
(прізвище, ім’я, по батькові)

1. Тема роботи Методи генерації та проходження лабіринтів

затверджена наказом по університету від “ 22 ” листопада 2024 р. № 1236 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 20 січня 2025 р.

3. Вхідні дані до роботи лабіринт, теорія графів, пошук найкоротшого шляху, задача генерації лабіринтів, джерела інформації з проблем генерації лабіринтів та їх вирішення, інформація щодо представлення ігрового світу у вигляді математичної моделі, інтегроване середовище розробки Visual Studio 2022, мова програмування C#, документація середовища розробки Unity 3D, середовище розробки Unity 2021.3.22f1.

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз предметної області;
2) розгляд алгоритмів пошуку найкоротшого шляху;
3) розгляд методів генерації лабіринтів;
4) вибір технологій розробки та інструментальних засобів;
5) реалізація програмного застосунку;
6) аналіз результатів досліджень;
7) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Слайд-презентація – 12 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	26.11.24-30.11.24	
	генерації лабіринтів та їх обходу		
2	Вибір та обґрунтування методики дослідження	02.12.24-05.12.24	
3	Вибір інструментальних засобів	06.12.24-10.12.24	
4	Розробка програмного забезпечення	11.12.24-21.12.24	
5	Проведення експериментів	23.12.24-03.01.25	
6	Оформлення матеріалів кваліфікаційної роботи	04.01.25-07.01.25	
7	Подання кваліфікаційної роботи керівникові та	08.01.25-11.01.25	
	її попередній захист		
8	Подання кваліфікаційної роботи на рецензу-	13.01.25-17.01.25	
	вання		

Дата видачі завдання 25 листопада 2024 р.

Студент _____
(підпис)

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

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 84 с., 10 рис., 4 табл., 2 дод., 26 джерел.

ЛАБІРИНТ, КЛІТИНА, АЛГОРИТМ, ГЕНЕРАЦІЯ ЛАБІРИНТУ, ШЛЯХ, СТРУКТУРА, ПОШУК ШЛЯХУ, ГРАФ, ТЕОРІЯ ГРАФІВ.

Метою кваліфікаційної роботи є дослідження методів генерації лабіринтів та пошуку шляхів, а також їх порівняльний аналіз. Розглянуто підходи на основі алгоритму бінарного дерева, методу Олдоса-Бродера та алгоритму Еллера для генерації лабіринтів, та пошук у ширину, алгоритм Дейкстри та метод A* для вирішення задачі пошуку шляху. Передбачено створення застосунку, що дозволить користувачу генерувати лабіринт за одним із обраних алгоритмів і знаходити шлях за допомогою реалізованих методів. Для розробки програмного застосунку використовуються ігровий двигун Unity та мова програмування C#.

У ході виконання кваліфікаційної роботи реалізовано застосунок, який генерує лабіринт одним з обраних методів та виконує пошук найкоротшого шляху у лабіринті усіма реалізованими алгоритмами водночас. Лабіринт генерується з заданим параметром розмірності, а знайдений шлях аналізується за часом, витраченим на його пошук. Проведено порівняльний аналіз алгоритмів генерації та вирішення лабіринтів.

ABSTRACT

Master’s thesis: 84 pages, 10 figures, 4 tables, 2 appendices, 26 sources.

LABYRINTH, KLITINA, ALGORITHM, GENERATION OF THE LABYRINTH, WAY, STRUCTURE, SEARCH FOR WAY, GRAPH, GRAPH THEORY.

In order to achieve the goal of the qualification work, the research focuses on methods of maze generation and pathfinding, along with their comparative analysis. The study examines such methods as the binary tree algorithm, Aldous-Broder method, and Eller's algorithm for maze generation, as well as breadth-first search, Dijkstra's algorithm, and the A* method for pathfinding. The project involves the development of an application that allows users to generate mazes using selected algorithms and find paths using the implemented methods. The Unity game engine and C# programming language are utilized for application development.

As part of the qualification work, an application has been implemented that generates mazes using one of the selected methods and performs shortest pathfinding in the maze using all implemented algorithms simultaneously. The maze is generated with a specified dimension parameter, and the found paths are analyzed based on the time taken for their computation. A comparative analysis of the maze generation and pathfinding algorithms has been conducted.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Алгоритм – впорядкований набір інструкцій або правил, який визначає послідовність дій

Воксель – тривимірний аналог пікселя, елементарна одиниця об'єму в тривимірному просторі

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

Евклідовий простір – скінченновимірний дійсний векторний простір зі скалярним добутком

Інтерфейс – визначений набір методів та властивостей, які спадкоємець повинен реалізувати

Крос-валідація – метод оцінки ефективності моделей машинного навчання та їх здатності до узагальнення на нових даних

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

Рендеринг – процес отримання зображення з цифрової 3D-моделі

Скрипт – послідовність дій для автоматичного виконання визначених завдань

Тайлинг – метод створення великих зображень або текстур шляхом повторення та комбінування менших фрагментів

ВСТУП

У зв’язку з швидким розвитком ігрової індустрії задача пошуку шляху є актуальною через постійну необхідність розрахунків найкоротшого маршруту під час переміщення ігрових об’єктів [1]. Для спрощення цих розрахунків ігрову локацію можна уявити як лабіринт, де стінами будуть усі місця, куди гравцеві шлях заборонено [2].

Для створення локацій в ігровій індустрії витрачається велика кількість часу та ресурсів. Через це для автоматизації генерації лабіринтів доцільне використання відповідних алгоритмів [3].

Лабіринти також широко використовуються у робототехніці [2, 4], де вони служать тестовими середовищами для розробки алгоритмів навігації та пошуку шляху. Роботи навчені орієнтуватися в лабіринтах, які можуть бути представлені у вигляді графів [5, 6], що допомагає покращити їхню здатність до самоуправління та адаптації [4]. В інженерії лабіринти є моделями для оптимізації складних систем, таких як дорожні мережі, лінії електропередач та каналізації.

Генерація лабіринтів – це процес створення структури, що складається з шляхів і стін [3]. Кожен лабіринт можна описати такими властивостями, як розмір, форма, кількість розвилок, гілок та їх середня довжина [7, 8].

Метою кваліфікаційної роботи є аналіз методів генерації лабіринтів [9-12] та пошуку найкоротшого шляху у них [13-15]. Для виконання цього завдання реалізовано застосунок на основі мови програмування C# та технології Unity [1], який дозволяє здійснювати аналіз та порівняння шляхів, побудованих обраними алгоритмами, а також часу, витраченого на пошук.

У ході аналізу досліджено використання таких підходів генерації лабіринтів як метод бінарного дерева [16-18], алгоритм Олдоса-Бродера [19-21] та алгоритм Еллера [16, 22, 23], а також таких підходів пошуку шляху: алгоритм Дейкстри [24, 25], метод A* [26] та алгоритм пошуку в ширину.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Галузі застосування лабіринтів

Актуальність вивчення лабіринтів визначається їх різнобічним застосуванням в таких областях, як математика, ігри та робототехніка [3, 4]. Лабіринти можуть використовуватися для вивчення теорії графів та алгоритмів [2, 5, 6]. Ці структури можуть бути описані у термінах теорії графів, при такому поданні вершини відповідають вузлам, наприклад кімнатам, а ребра – можливим переходам між ними [7, 8]. Таке уявлення дозволяє використовувати методи теорії графів для аналізу структури лабіринту та пошуку шляхів [2, 5, 6, 8].

Лабіринти також можуть бути проаналізовані з точки зору їхньої зв’язності [3, 7]. Наприклад, якщо лабіринт складається з декількох незв’язаних частин, алгоритми пошуку можуть не знайти вихід, якщо він знаходиться в іншому компоненті лабіринту. Це призводить до необхідності враховувати структуру графу при розробці алгоритмів [3, 5, 8].

Рішення завдань з пошуку шляху в лабіринті дає можливість застосовувати математичні моделі для оптимізації процесів [2, 7], що робить їх важливими як в освітньому контексті, так і в наукових дослідженнях.

Методи теорії графів, які засновані на аналізі лабіринтів, знаходять застосування у різних галузях, таких як робототехніка [4], планування маршрутів і оптимізація транспортних систем [2, 6, 7]. Це дозволяє ефективно вирішувати завдання навігації та мінімізації витрат на переміщення [8].

У сфері робототехніки лабіринти служать тестовим середовищем для вдосконалення навігаційних алгоритмів. Дослідження в цій галузі сприяють розвитку технологій навігації та планування шляху автономних роботів, що має значення для створення ефективних автономних систем пересування [4].

Сучасні дослідження підтверджують, що роботи можуть навчатися орієнтуватися в лабіринтах за допомогою нейроморфних систем. Наприклад, навчання робота на власному досвіді, отримуючи коригувальні сигнали, при виниканні помилок, що дозволяло йому поступово знаходити правильний шлях [4]. Ця методологія використовує сенсорні дані для адаптації поведінки робота в режимі реального часу.

У сфері обчислювального інтелекту лабіринти використовуються як тестове середовище для розробки інтелектуальних агентів, які можуть розпізнавати структуру лабіринту [8] та планувати свої дії. Це важливо для створення більш складних навігаційних систем і взаємодії з навколишнім середовищем.

Таким чином, завдання дослідження лабіринтів залишається актуальним та багатогранним, а також охоплює освітні, розважальні та наукові аспекти.

1.2 Класифікація лабіринтів

Лабіринт – це структура, що складається із проходів та стін [7]. Його можна визначити за семи класифікаціями: розмірності, гіперрозмірності, топології, тесселяції, маршрутизації, текстури та пріоритету [3, 5]. Лабіринт може характеризуватися властивостями, що притаманні кожній з наведених класифікацій, в будь-якому поєднанні [9].

Властивість розмірності визначає кількість вимірів у просторі, які заповнює лабіринт:

- двовимірні лабіринти. Найрозповсюдженіша розмірність лабіринту, завдяки якій можна відобразити план на аркуші паперу та переміщатися ним, не перекриваючи інші проходи в лабіринті [3];

- тривимірні лабіринти. Лабіринти з кількома рівнями, які пов’язані між собою. Тривимірний лабіринт часто відображається як масив двовимірних рівнів із покажчиками сходів;

- лабіринти з вищими вимірами. Чотиривимірні лабіринти, та лабіринти з більшою кількістю вимірів, які іноді зображуються як 3D лабіринти зі спеціальними засобами переміщення між вимірами [9];

- лабіринти з переплетеннями. Двомірні лабіринти, в яких проходи можуть накладатися один на одного. При відображенні таких лабіринтів можна побачити, де знаходиться глухий кут і як один прохід знаходиться над іншим [8].

Класифікація за гіперрозмірністю відповідає розмірності об’єкта, що рухається через лабіринт. Більшість лабіринтів є не-гіперлабіринтами [5]. В них оброблюється об’єкт, який рухається від точки до точки, а прокладений маршрут утворює лінію [9]. У кожній точці існує обмежена кількість варіантів вибору, які є принципова можливість прорахувати. Гіперлабіринти це лабіринти, в яких є складний об’єкт, що пересувається лабіринтом. Стандартний гіперлабіринт складається з лінії, яка при переміщенні утворює поверхню. Гіперлабіринт може існувати тільки в 3D або середовищі з більшою розмірністю. Гіперлабіринт фундаментально відрізняється, через обробку декількох частин вздовж лінії водночас. У кожну мить часу існує практично нескінченна кількість станів лінії. Лінія рішення нескінченна, або її кінцеві точки знаходяться за межами гіперлабіринту [9].

Класифікація за топологією визначає геометрію простору, де існує лабіринт [3]. Стандартний лабіринт розташовано в евклідовому просторі. Будь-який лабіринт з аномальною топологією визначається як «planair» лабіринт. Зазвичай межи таких лабіринтів з’єднані нестандартно. Прикладами є лабіринти на поверхні куба та лабіринти на поверхні стрічки Мебіуса [9].

Теселяція визначає класифікацію геометрії окремих клітин, з яких складається лабіринт. Вона має наступні типи:

- ортогональний. Це стандартна прямокутна сітка, у якій клітини мають проходи, що перетинаються під прямими кутами. У контексті тесселяції їх можна назвати гамма-лабіринтами;

- дельта. Такі лабіринти складаються із з'єднаних трикутників, і в кожній клітині може бути до трьох з'єднаних з нею проходів;

- сигма. Лабіринти, які складені зі з'єднаних шестикутників. У кожного осередку може бути до шести проходів;

- тета. Лабіринти складаються з концентричних кіл проходів, в яких початок або кінець знаходиться в центрі, а інший – на зовнішньому краї [9]. Осередки зазвичай мають чотири можливі з'єднання, але їх може бути і більше завдяки більшій кількості осередків у зовнішніх кільцях проходів;

- іпсилон. Іпсилон-лабіринти складаються із з'єднаних восьмикутників або квадратів. У таких лабіринтах кожен осередок може мати до чотирьох або восьми проходів;

- дзета. Лабіринт, який розташовано на прямокутній сітці, але додатково до горизонтальних та вертикальних проходів допускаються діагональні проходи під кутом 45 градусів;

- crack. Це аморфний лабіринт без постійної тесселяції. В лабіринтах такого типу стіни та проходи розташовані під випадковими кутами;

- фрактальний. Лабіринт можна назвати фрактальним, якщо він складається із менших лабіринтів [9]. Процес вкладання лабіринтів усередину може повторюватися кілька разів. Нескінченно рекурсивний фрактальний лабіринт – це фрактал, де лабіринт містить копії самого себе і фактично є нескінченно великим лабіринтом.

Маршрутизація визначає спосіб, яким організовані проходи та з'єднання між різними частинами лабіринту. За маршрутизацією лабіринти можна поділити на наступні:

- ідеальний. Ідеальним називають лабіринт без петель чи замкнутих ланцюгів та без недосяжних областей [7, 21]. Також він називається лабіринтом з одиночним з'єднанням. З кожної точки існує рівно один шлях до будь-якої іншої точки [9]. Такий лабіринт має лише одне рішення. З погляду програмування такий лабіринт можна описати як дерево, що складається з множини вершин [3];

- плетений. Це такий лабіринт, в якому немає глухих кутів. Такий лабіринт також називають лабіринтом із багаторазовими сполуками. У такому лабіринті використовуються проходи, що замикаються та повертаються один до одного. Такий принцип генерації змушує витрачати більше часу для пересування колами замість попадання в глухий кут [7]. Плетений лабіринт може бути набагато складнішим за ідеальний лабіринт [21] того ж розміру [9];

- одномаршрутний. Лабіринт можна назвати одномаршрутним, якщо він не має розвилок. Одномаршрутний лабіринт містить один довгий прохід, який змінює напрямок на всьому протязі лабіринту;

- розріджений. Такий лабіринт не прокладає проходи через кожну клітину. Таку концепцію можна застосувати і при додаванні стін, завдяки чому можна отримати нерівномірний лабіринт із широкими проходами та кімнатами.

Класифікація за текстурою описує стиль проходів за різної маршрутизації та геометрії [9]. Розповсюджені приклади змінних, які використовуються при створенні лабіринту, наведені нижче:

- зміщення. У лабіринті зі зміщеними кількість проходів в одному напрямку більше ніж в інших. У лабіринті з високим коефіцієнтом горизонтального зміщення містяться довгі проходи зліва направо, і лише короткі проходи зверху вниз, що з’єднують їх [9];

- прольоти. Цей показник визначає, як довго йдуть довгі проходи, перш ніж з’являться вимушені повороти. У лабіринті з низьким показником прольотів не буде прямих проходів довше трьох-чотирьох клітин. У лабіринті з високим показником прольотів протягом лабіринту буде великий відсоток довгих проходів;

- елітність. Показник елітності визначає довжину рішення щодо розміру лабіринту [3]. У елітних лабіринтів зазвичай є коротке рішення, а в неелітних лабіринтах рішення проходить з великої частини площі лабіринту. Отже елітний лабіринт може бути набагато складнішим, ніж неелітний [9];

- симетрія. Симетричний лабіринт має симетричні проходи. Лабіринт може бути частково або повністю симетричним, а також повторювати патерн будь-яку кількість разів;

- плинність. Ця характеристика означає, що при створенні лабіринту алгоритм шукатиме і очищатиме сусідні осередки до поточної. В ідеальному лабіринті з меншим показником плинності зазвичай буде безліч коротких глухих кутів. При збільшені цього показнику буде зменшуватись кількість глухих кутів, але вони будуть довшими [9].

Класифікація за пріоритетом, за яким був створений лабіринт. Лабіринти за пріоритетом можна розділити на два основні типи: додавання стін та прорізів. При генерації це зводиться до різниці в алгоритмах, а не до відмінностей лабіринтів. Один і той же лабіринт часто генерується обома способами [9]. Алгоритми, для яких пріоритетом є стіни, починають із порожньої області, в процесі роботи додаючи стіни, а алгоритми, пріоритетом яких є проходи, починають із суцільного блоку та у процесі роботи вирізають у ньому проходи. У деяких алгоритмах можливо одночасне і вирізання проходів і додавання стін у лабіринті [9].

Шаблоном лабіринту називають графічне зображення, яке не є лабіринтом, але яке за найменшу кількість кроків перетворюється на лабіринт. Воно також все одно зберігає текстуру вихідного графічного шаблону. Складні стилі лабіринтів, наприклад, спіралі, що перетинаються, простіше реалізувати заздалегідь як шаблони, а не намагатися створити правильний лабіринт, в той же час зберігаючи його стиль [9].

1.3 Існуючі методи генерації лабіринтів

Під час генерації лабіринту спочатку створюється пуста (або повністю заповнена) структура, та завдяки обраному методу генерації будується лабіринт [3, 21]. Існує велика кількість методів генерації лабіринту, кожен з них має певні особливості та може бути більш або менш ефективним в

залежності від типу лабіринту та поставленого завдання [16]. Серед поширених алгоритмів генерації лабіринту можна виділити такі, як метод бінарного дерева [17, 18], алгоритм Олдоса-Бродера [19-21] та алгоритм Еллера [22].

1.3.1 Формування лабіринтів на основі фотозображень

Дослідження методу формування лабіринтних зображень з використанням згладжуючих фільтрів [10] розглядає різні методи нефотореалістичного рендерингу. Розроблена програма приймає зображення або тривимірні дані в якості вхідних даних і генерує нові вихідні зображення з вказаним художнім стилем.

Було запропоновано кілька методів нефотореалістичного рендерингу для генерації лабіринту [10], а саме, метод, що використовує диференціальні рівняння, метод, що використовує тайлинг та метод, що використовує мінімальні остовні дерева.

У цій статті також пропонується простий і швидкий метод нефотореалістичного рендерингу для створення лабіринту з фотографічних зображень. Запропонований метод реалізовано за допомогою ітеративної обробки з використанням двох згладжуючих фільтрів з різними розмірами вікон [10].

Хоча запропонований метод і звичайний метод використовують однакову ітераційну обробку, новий метод може досягати збігу зображення лабіринту з меншою кількістю ітерацій, ніж традиційний [10]. Крім того, запропонований метод здатний автоматично генерувати лабіринтові зображення шляхом зміни значень яскравості фотозображень, а також регулювати ширину та форму лабіринтів шляхом зміни параметрів.

У результаті експериментів запропонований метод продемонстрував високу швидкість, але у нього було виявлено певні недоліки. Для працездатності алгоритму потрібно використання досить великої кількості

параметрів, що значно ускладнює читання та модифікацію коду. Також запропонований метод не має розробленого функціоналу для застосування до кольорових фотографічних зображень і відео.

1.3.2 Гібрид алгоритмів Олдоса-Бродера та Вілсона

Генерація складних і випадкових лабіринтів застосовується в різних сферах, включаючи комп'ютерні науки, математику, ігри та симуляції. Дослідження [7, 11] представляє інноваційний підхід, інтегруючи два відомі алгоритми генерації ідеальних лабіринтів: алгоритм Олдоса-Бродера [20, 21] та алгоритм Вілсона [16, 19]. Перевагою обох алгоритмів є сильна випадковість та ефективність, але їх комбінація пропонує новий спосіб оптимізації генерації лабіринтів.

Проведено детальний аналіз взаємозв'язку між коефіцієнтом покриття та розміром карти. Сформовано механізм, який забезпечує перехід до алгоритму Вілсона [19] з метою мінімізації витрат часу. У рамках досліджень гібридного алгоритму досягнуто середньої економії часу на 34,124% [11].

З огляду на переваги алгоритмів Олдоса-Бродера [20] та Вілсона [19] у генерації рівномірних випадкових остовних дерев, у статті пропонується використати їх у розробці нових алгоритмів генерації лабіринтів. В результаті аналізу процесу генерації гілок за допомогою алгоритмів Олдоса-Бродера [20, 21] та Вілсона [16, 19], слід зазначити еквівалентність між ними в точці завершення гілки, що дозволяє розробити гібридний алгоритм. Цей гібридний алгоритм демонструє доцільність формування остовного дерева для повних графів [11].

Метою розглянутого дослідження є розробка нового гібридного алгоритму із знаходженням формули розрахунку покриття, для її подальшого використання при створенні лабіринтів різних розмірів. Під час дослідження були використані методи машинного навчання для автоматизації пошуку стратегії переходу між алгоритмами Олдоса-Бродера [20] та Вілсона [19] для

лабіринтів різних розмірів [11]. Застосовано різноманітні методи регресійного аналізу та крос-валідації для побудови регресійних моделей і спроби виділити більш узагальнені моделі.

Результати розглянутого дослідження можуть бути безпосередньо застосовані до ефективної генерації лабіринтів конкретних розмірів у ігрових програмних застосунках, тим самим вносячи оригінальні ідеї та рішення для генерації лабіринтів у іграх через інноваційне поєднання класичних алгоритмів [11].

Одним із недоліків запропонованого методу поєднання двох відомих алгоритмів генерації лабіринтів є залежність від параметрів, через те що ефективність методу може сильно залежати від вибору параметрів, таких як розмір лабіринту та значення покриття, що може ускладнити його використання в практичних застосуваннях. Крім того запропонований метод може не забезпечувати достатньо гнучкості для адаптації до різних стилів генерації лабіринтів, що може обмежити його застосування в різних контекстах.

Можна також відмітити складність налаштування, так як автоматизовані методи пошуку стратегії можуть вимагати значного часу на налаштування та тестування, що може бути недоцільним у випадках, коли потрібна швидка генерація лабіринтів.

1.3.3 Процедурна генерація у відеоіграх

За останнє десятиліття індустрія відеоігор зазнала безпрецедентних змін, які більш за все торкнулися процесу проектування. Розробники відеоігор змушені генерувати багато вмісту для однієї гри, що призводить до значного збільшення витрат на виробництво. Цей вміст має бути згенерований дизайнером, і в більшості випадків його створення також передбачає необхідність складніших завдань програмування для його реалізації.

Процедурне створення контенту полягає в тому, що відеогра або її частина генерується обчислювальним шляхом за допомогою чітко визначеної процедури. Використання таких методів процедурної генерації значно зросло за останнє десятиліття [12].

Завдання авторів дослідження [12] полягає в розробці чотирьох демонстрацій, де вони використовуватимуть різні методики процедурної генерації. У кожній демонстрації використовуватиметься один або кілька методів машинного навчання для створення певного типу вмісту [12]. Для створення цих демонстрацій використано відому інтегровану систему розробки Unity [1].

У статті [12] досліджено різні алгоритми штучного інтелекту, переважно на основі правил, які використовуються для генерації процедурного контенту. Ці алгоритми були застосовані для генерації лабіринту та генерації рельєфу.

Лабіринти, створені, дослідженими у статті алгоритмами, моделюються як двовимірні матриці, елементи яких представляють комірки лабіринту, кожна з яких має чотири стіни [12]. Створені лабіринти завжди відповідають визначенню ідеального лабіринту [21].

Під час дослідження алгоритм пошуку у глибину створював лабіринти з дуже довгими коридорами і короткими шляхами без виходу. Алгоритм Пріма [19] генерував лабіринти з дуже короткими коридорами, великою кількістю хрестів і кількома дорогами без виходу. Особливістю алгоритму рекурсивного поділу є забезпечення необхідного ступеню хаотичності згенерованого лабіринту [12], це сприяє складності знаходження спільних ознак між згенерованими лабіринтами, що сповільнює процес пошуку правильного шляху гравцем.

Розглянуті у роботі зазначені підходи порівнювалися за мінімальним, середнім та максимальним часом генерації у різних випадках. Як наслідок алгоритм Пріма [19] надає кращі результати і є доцільним, щоб уникнути максимального часу генерації. Пошук у глибину отримує найгірший

результат, оскільки занадто часто повертатися до попередніх кроків. А результати використання алгоритму рекурсивного поділу схожі у середній продуктивності з результатами алгоритмом Пріма, але з найгіршим максимальним часом генерації [12].

Під час тестування та дослідження алгоритмів генерації автори проводили тестування з використанням невеликої кількості досліджуваних параметрів. Рекомендацією є дослідження лабіринту також за показниками симетрії та плинності. Також тестування проводилося у лабіринтах досить малого розміру [12]. Для більш точних результатів, які можуть бути застосовані під час більш детального аналізу генерації лабіринтів, слід розглядати лабіринти більшого розміру.

1.4 Аналіз існуючих досліджень алгоритмів пошуку шляху у лабіринтах

1.4.1 Порівняння алгоритмів пошуку A*

Для порятунку людей, які опинилися в пастці в небезпечних ситуаціях в локаціях, які можна абстрагувати як лабіринт, можна використовувати алгоритми пошуку шляху [25]. Алгоритм A* надає прийнятні результати для його застосування у роботах [4], які можуть буди широко використані для виконання цього завдання.

У статті [13] розглянуто три типові варіації алгоритму A* та порівняно їхню ефективність пошуку в лабіринті. Оскільки алгоритм пошуку в глибину можна розглядати як алгоритм A* без евристики, тому всім рухам надається однакова вага. Також було використано алгоритм пошуку в глибину як еталон для оцінки корисності евристики функції алгоритмів A* [26].

Мета дослідження полягає в тому, щоб знайти варіант алгоритму A*, який перевершує інші в програмі пошуку в лабіринті, орієнтованій на використанні в задачах порятунку [13], і, як результат, який забезпечує

можливість використовувати цей алгоритм A* у фізичному роботі [4] для виконання завдання пошуку та порятунку.

Експерименти підтвердили ефективність використання евристичних функцій, оскільки результати показали, що алгоритми A* перевершують алгоритм пошуку в глибину в більшості випадків. Зокрема, алгоритм A* з використанням евклідової відстані між початковою та поточною точкою до цільової точки продемонстрував найкращі результати у проведених експериментах [13].

Таким чином, результати дослідження [13] підкреслюють важливість використання оптимізованих алгоритмів для вирішення складних завдань пошуку в лабіринтах, що відкриває нові можливості для покращення засобів рятування та навігації.

Проте алгоритм A* з використанням евклідової відстані між початковою та поточною точкою має кілька недоліків, які можуть негативно вплинути на його ефективність у задачах пошуку. Перш за все це ігнорування існуючих перешкод, оскільки алгоритм A* з евклідичною евристикою не враховує наявність стін або інших перешкод, що може призвести до того, що алгоритм вибере шлях, який насправді буде недоступним. Також при використанні евклідової відстані необхідно виконання великої кількості обчислень для оцінки відстаней, що може сповільнити роботу алгоритму в великих графах [2].

Для покращення результатів роботи алгоритму пошуку шляху A* запропоновано розглянути використання наступних альтернативних евристичних функцій. Застосування гібридної евристики, яка поєднує кілька метрик, таких як манхеттенська відстань для прямих шляхів і евклідова для діагональних, може підвищити точність оцінок і поліпшити загальну ефективність пошуку шляху. Крім того використання генетичних алгоритмів для автоматичного налаштування та оптимізації евристичних функцій може допомогти знайти найбільш ефективні підходи для конкретних задач пошуку шляху.

1.4.2 Застосування мультиагентного підходу

Мета дослідження [14] полягає у знаходженні кожним агентом свого шляху для досягнення цілі, як окремо, так і в групах. Слід врахувати те, що цей підхід дозволяє агентам рухатися, не стикаючись один з одним. Цей метод пошуку шляху реалізовано в багатьох сферах [2], де потрібне пересування різних агентів, таких як складські роботи, автономні автомобілі, відеоігри, контроль дорожнього руху, безпілотні літальні апарати, пошуково-рятувальні служби тощо [4, 14].

Використання агентів часто передбачає, що всі області, які потрібно дослідити, вільні від перешкод. Однак використання мультиагентного пошуку шляху для досягнення своїх цілей часто стикається зі статичними бар'єрами [14], і навіть інші агенти також можна вважати динамічними бар'єрами. Це вимагає певних обмежень у програмі, наприклад, заборона агентам стикатися один з одним.

Використання одного агенту може знайти найкоротший шлях, але багатоагентна система повинна скоротити час для пошуку цільового розташування, особливо якщо є більше ніж одна ціль [14].

У дослідженні здійснено огляд досліджень різноманітних багатоагентних алгоритмів пошуку [2]. Виявлено, що мультиагентне дослідження лабіринтів більш ефективно з причин, пов'язаних з особливостями взаємодії та координації між агентами. Мультиагентні системи дозволяють кільком агентам одночасно досліджувати різні частини лабіринту. Це значно прискорює процес пошуку шляху, оскільки кожен агент може зосередитися на своїй території, що призводить до швидшого виявлення шляхів та виходів [14]. Крім того, різні агенти можуть використовувати різні алгоритми та стратегії для вирішення однієї і тієї ж задачі.

Крім того мультіагентні системи можуть швидко адаптуватися до змін, що особливо важливо в динамічних середовищах або лабіринтах, що

змінюються. Якщо один із агентів стикається з несподіваною перешкодою, інші агенти можуть змінити свої маршрути на основі нової інформації [14].

До недоліків обраного авторами статті методу можна віднести складність розробки та налаштування мультиагентної системи. Також мультиагентні системи можуть вимагати значних обчислювальних ресурсів, при великої кількісті агентів. Це може призвести до проблеми із масштабованістю та збільшення витрат на обладнання.

1.4.3 Пошук шляху за допомогою тетраедральної сітки

У роботі [15] пропонується підхід до 3D-пошуку шляху в іграх, використовуючи тетраедральну сітку як структуру пошуку шляху. Розглядаються різні методи створення такої тетраедричної сітки, і вибирається один із них. Спочатку цей метод перетворює оточення на бінарну воксельну сітку, яка використовується для генерації тетраедричної навігаційної сітки. Наступним кроком виконується порівняння тетраедричної сітки з розрідженою воксельною октодеревою структурою у двох сценах [15].

Результати показали, що тетраедральна сітка може бути краще за охопленням і складністю, але за рахунок продуктивності. Цей підхід робить навігацію з використанням тетраедричної сітки непридатною для ігор. Через це були проведені дослідження щодо покращення продуктивності. Оскільки за результатами аналізу продуктивність тетраедричної сітки було збільшено , вона може стати альтернативою існуючим структурам 3d-навігації [15].

З урахуванням того, що метою дослідження було визначення можливості використання тетраедральної сітки для навігації в іграх, порівняння проводилося за трьома категоріями показників : покриття, складність і продуктивність [15]. Оскільки авторам не вдалося знайти стандартну методологію, яка б порівнювала дві структури даних за вибраними показниками, був розроблений спрощений метод для порівняння двох структур [15]. Цю методологію можна використовувати для порівняння

будь-яких двох структур даних, які використовуються для навігації, якщо вони працюють в одному вимірі.

Використання тетраедральної сітки для розрахунку маршруту переміщення лабіринтом може негативно вплинути на час пошуку через геометрію тетраедра. Створення тетраедральної сітки може вимагати значних обчислювальних ресурсів та ручного втручання.

1.5 Постановка задачі

Оскільки лабіринти використовуються як у ігрових застосунках, так і у робототехніці [4] та вивченні графів [2, 5], ефективність алгоритмів генерації може відрізнятись відносно особливостей лабіринту, таких як його розмірність, наявність окремих кімнат, довжини шляхів а також інших особливостей [3, 6, 7].

Під час пошуку шляху у лабіринтах, створених різними алгоритмами генерації, результати одних й тих самих алгоритмів можуть відрізнятись через особливості структури лабіринту [3, 8]. Це призводить до необхідності порівняння часу пошуку шляху та інших параметрів кожного з алгоритмів генерації до кожного окремого методу генерації [2].

Метою роботи є аналіз методів генерації лабіринтів [18-22] та методів пошуку шляху [25, 26], а також порівняння результатів. Крім того необхідною є розробка застосунку, у якому користувачеві буде надана можливість генерації лабіринту одним з обраних алгоритмів та пошук шляху обраним методом. Для розробки застосунку обрано двигун Unity [1] та мову програмування C#.

2 ВИКОРИСТОВУВАНІ ТЕХНОЛОГІЇ

На сьогодні велика кількість застосунків, що випускаються, розроблені за допомогою двигуна Unity, який використовує мову програмування C#, яка широко використовується у теперішній час [1]. В цьому двигуну вже реалізовано більшість необхідних функцій, що значно спрощує процес розробки програмного застосунку. Ще одна перевага Unity полягає у тому, що навколо нього сформувалася велика спільнота, яка готова ділитися досвідом, також наявна велика кількість докладних уроків [1].

Оскільки двигун Unity використовує мову C# [1], розробник має доступ до великої кількості сучасних синтаксичних можливостей, які спрощують процес написання коду. Наприклад є можливість використовувати готову конструкцію, а компілятор застосує усі необхідні програмісту інструкції. Можна зазначити, що деякі конструкції не є оптимальними з точки зору продуктивності, але підвищують зручність читання коду і організують високу швидкість розробки [1].

Однією з найважливіших переваг є наявність великої кількості бібліотек і шаблонів, що дозволяють зменшити час програмування за рахунок використання бібліотек готового коду [1]. Зокрема, наявна можливість використання та вдосконалення готових реалізацій класичних алгоритмів теорії графів, широко застосовуваних в контексті роботи з лабіринтами.

2.1 Алгоритми генерації лабіринтів

2.1.1 Алгоритм бінарного дерева

Метод бінарного дерева є одним з найпростіших методів генерації лабіринту [3, 16-18]. Він здатний створити ідеальний лабіринт, та не зберігає

ніякого стану під час генерації [7]. Алгоритм бінарного дерева може побудувати весь лабіринт, оброблюючи лише одну клітину за одну ітерацію алгоритму. Створюється лабіринт, що представляє кожну клітину як вузол дерева [16]. При генерації лабіринту для кожної клітини вибирається одне з двох можливих направлень для створення проходу, якщо це можливо, вгору або праворуч [3]. Напрямок залежить від обраного зміщення [17, 18].

Процес генерації починається з порожнього лабіринту, де всі клітини спочатку є стінами. Під час виконання алгоритму оброблюється кожна клітина у сітці лабіринту [18]. До кожної клітини що оброблюється застосовується два правила: якщо клітина не знаходиться на верхньому кордоні, можна створити прохід вгору, та якщо вона не знаходиться на правій межі, можна створити прохід праворуч [16]. Кожну ітерацію випадковим чином вибирається один із доступних напрямків і створюється прохід шляхом видалення стін між поточною клітиною та обраною сусідньою клітиною [17, 18].

Результатом роботи алгоритму бінарного дерева є випадкове двійкове дерево [16, 18], в якому з кожної клітини є тільки один шлях до батьківської вершини і тільки один шлях до будь-якої іншої клітини (рисунок 2.1). Через це будь-яка клітина має не більше трьох з’єднань з сусідніми клітинами [17]. Такі лабіринти також мають тенденцію до утворення діагональних проходів.

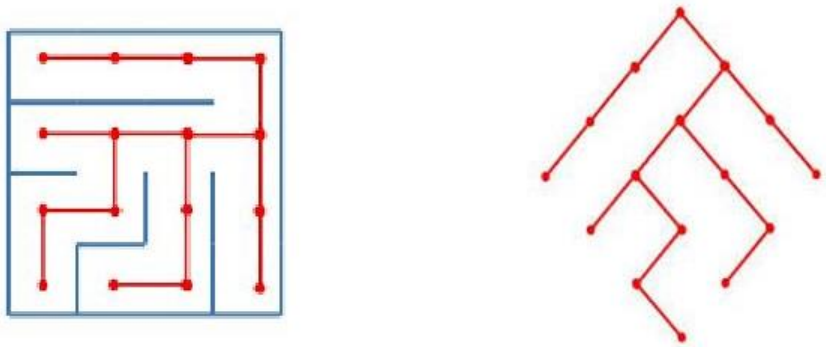

Рисунок 2.1 – Приклад виконання алгоритму бінарного дерева у вигляді лабіринту та у вигляді двійкового дерева

До переваг бінарного методу можна віднести просту реалізацію, високу швидкість виконання, можливість генерації нескінченних лабіринтів та відсутню необхідність зберігати додаткову інформацію під час виконання [3, 16]. Недоліками є діагональне зміщення та одноманітність згенерованих лабіринтів [17, 18].

2.1.2 Алгоритм Олдоса-Бродера

Алгоритм Олдоса-Бродера є сучасним методом для генерації рівномірно випадкових остовних дерев [5] у кінцевих зв'язних графах [2, 20]. Розроблений незалежно Девідом Олдосом і Алістером Бродером, цей алгоритм базується на концепції випадкових переміщень, що дозволяє досягти цілей у випадковій генерації структур [16, 19, 21]. Алгоритм генерації реалізується за наступними етапами [20]:

- випадковим чином обирається початкова вершина графа, яка позначається як відвідана;
- виконується випадкове переміщення по графу, де на кожному кроці обирається сусідня вершина з ймовірністю, пропорційною кількості сусідніх вершин [19];
- якщо обрана сусідня вершина не була відвідана, ребро, що з'єднує поточну вершину з цією сусідньою, додається до остовного дерева, а сусідня вершина позначається як відвідана [20];
- процес повторюється до тих пір, поки всі вершини графа не будуть відвідані, в результаті чого формується рівномірне остовне дерево.

Цей підхід забезпечує теоретичну рівномірність у виборі остовних дерев, гарантуючи, що кожне можливе остовне дерево обирається з однаковою ймовірністю [16, 19, 21].

Алгоритм Олдоса-Бродера надає можливість генерації ідеальних лабіринтів, розглядаючи кожну клітину як вершину, а стіни – як ребра. Отриманий лабіринт є повністю зв'язним і без циклів [3, 16, 19-21].

2.1.3 Алгоритм Еллера

Алгоритм Еллера є ефективним методом генерації лабіринтів, що створює однозв’язні структури, гарантуючи існування між будь-якими двома точками єдиного шляху [16, 22]. Цей алгоритм відрізняється простотою реалізації та достатньо низькими вимогами до пам’яті, що робить його поширеним у теперішній час [7, 16]. Алгоритм працює за принципом рядкової генерації, це означає що кожен рядок лабіринту обробляється окремо [3]. Генерацію лабіринту алгоритмом Еллеру можна поділити на наступні етапи [22]:

- генерується сітка майбутнього лабіринту, усі клітини якого спочатку перебувають у окремих множинах;
- випадковим чином з’єднуються сусідні клітини, за умови, що вони не належать до однієї множини [22]. При з’єднанні сусідніх клітин необхідно об’єднати елементи обох множин в єдину, вказуючи на те, що всі клітини в цих множинах тепер є взаємопов’язаними, тобто існує шлях, що з’єднує будь-які дві клітини в множині [16];
- для кожного набору потрібно випадковим чином створити вертикальні зв’язки до наступного рядка. Кожен залишений набір повинен містити принаймні один вертикальний зв’язок [16]. Клітини в наступному рядку, з’єднані таким чином, повинні належати тому ж набору, що й клітина над ними [22];
- доповнити наступний ряд, розміщуючи залишені клітини в окремих наборах. Цей процес необхідно повторювати до досягнення алгоритмом останнього ряду;
- в останньому рядку з’єднати всі сусідні клітини, які не належать до одного набору, та виключити вертикальні зв’язки [16, 22]. Після цього процедура буде завершена.

Детальний огляд обробки строк під час генерації лабіринту методом Еллера зображено на рисунку 2.2

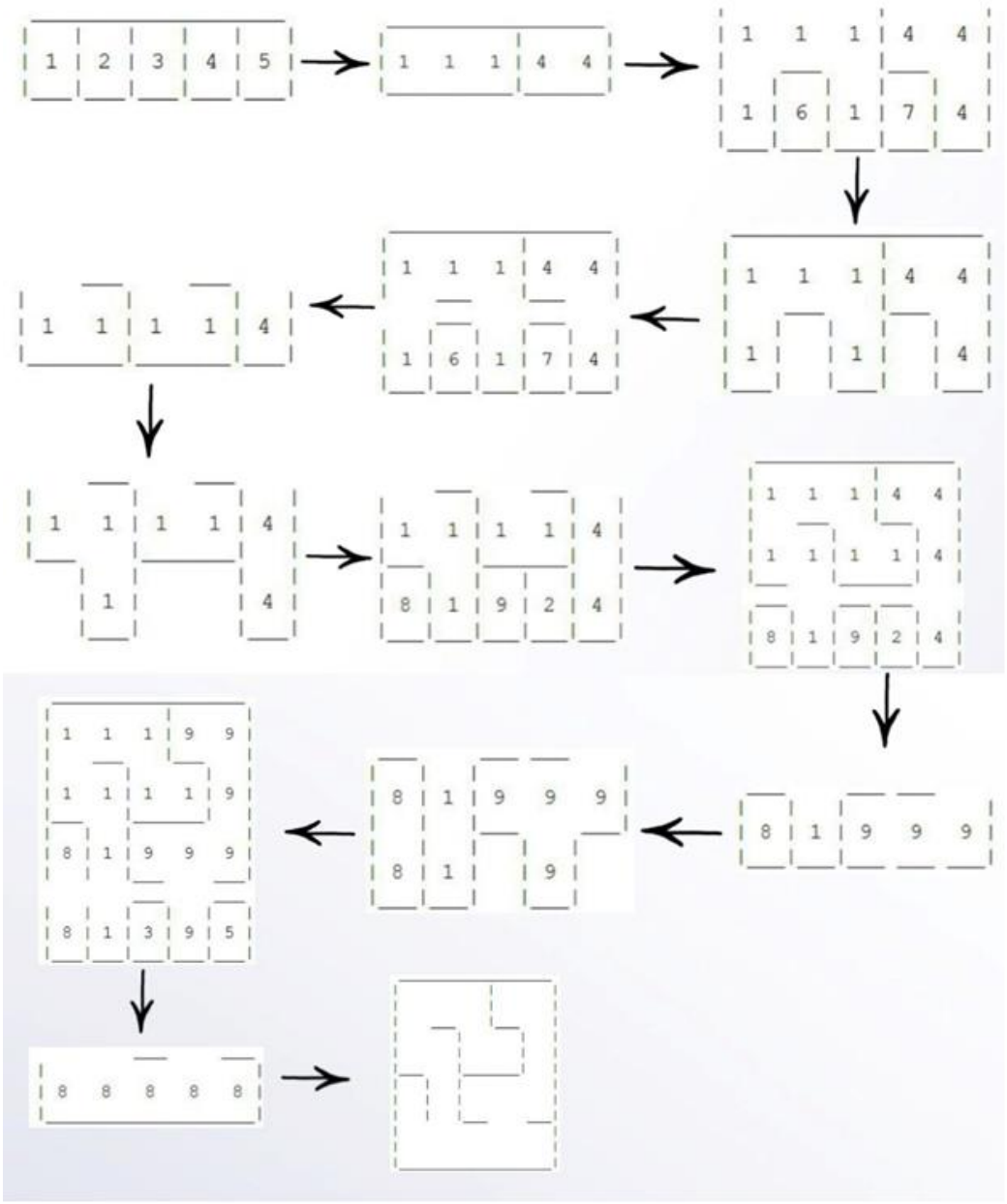

Рисунок 2.2 – Покроковий приклад обробки рядків алгоритмом Еллера

2.2 Методи пошуку шляху у лабіринтах

Визначення найкоротшого маршруту широко використовується при розробці ігрових програмних застосунків, в яких існують об’єкти, що вільно пересуваються.

Завдання пошуку шляху складається з двох етапів: адаптування ігрового світу до математичної моделі та пошук у цій моделі шляху між

двома точками. Для адаптування ігрового світу до математичної моделі необхідно описати ігровий світ у вигляді графу і вибрати набір ознак, якими буде визначатися, що між двома вершинами можна пройти. Вершинами зазвичай є мінімальні площі світу, в яких пересування від одного краю до другого відбувається миттєво, або за незначущий час.

Одна з найпростіших варіацій графа – ігрова сітка, де точками є клітини, а ребра проводяться між будь-якими сусідніми клітинами, вільними від перешкод. Клітини позначаються 0 або 1, де 1 означає, що пройти не можна, а 0 – що пройти можна.

Під час роботи з тривимірними світами можна використовувати двомірну карту проекції ландшафту із зазначенням висот у точках. При такому підході у кожній клітині вказана її висота, а прохідність визначається через різницю висот сусідніх клітин.

Найчастіше адаптація ігрового світу – це найскладніша частина розробки ігрового застосунку, після виконання якої можливе використання та модифікація вже існуючих ефективних алгоритмів пошуку маршрутів, у відповідності до особливостей поставленого завдання. При розробці зазвичай впроваджується один з алгоритмів і потім проводяться експерименти з різними моделями та варіаціями порівнянь двох шляхів. Перспективним рішенням є комбінація різних алгоритмів з метою досягти найкращого результату.

2.2.1 Метод пошуку у ширину

Існує велика кількість алгоритмів пошуку шляху [2]. Наприклад, пошук в ширину починає досліджувати шляхи від початкової точки відразу на всі боки. Спочатку переглядаються сусідні зі стартом точки, потім сусідні з ними і так далі, доки не знайде кінцеву точку або поле не закінчиться.

Виконання алгоритму пошуку у ширину можна поділити на наступні кроки:

- переміщення стартової точки до списку на перевірку;
- додавання усіх точок зі списку «на перевірку» до списку «досліджено»;
- знаходження всіх сусідніх точок для всіх точок зі списку на перевірку, на які можна перейти та додавання їх у список «поточні»;
- видалення зі списку «поточні» тих точок, які вже були досліджені та зберігаються у списку «досліджене»;
- очищення списку «на перевірку» та поміщення туди всіх точок зі списку «поточне»;
- очищення поточного списку.

Алгоритм повторюється з кроку 2, поки не буде знайдено кінцеву точку або всі доступні точки не будуть перевірені.

Такий алгоритм на кожному N циклі знаходить всі точки, яких можна дійти за N кроків. Щоб отримати шлях, потрібно запам’ятовувати як перевірені точки, так й послідовність точок до них. Послідовність перевірки вершин у графі лабіринту зображено на рисунку рисунок 2.3.

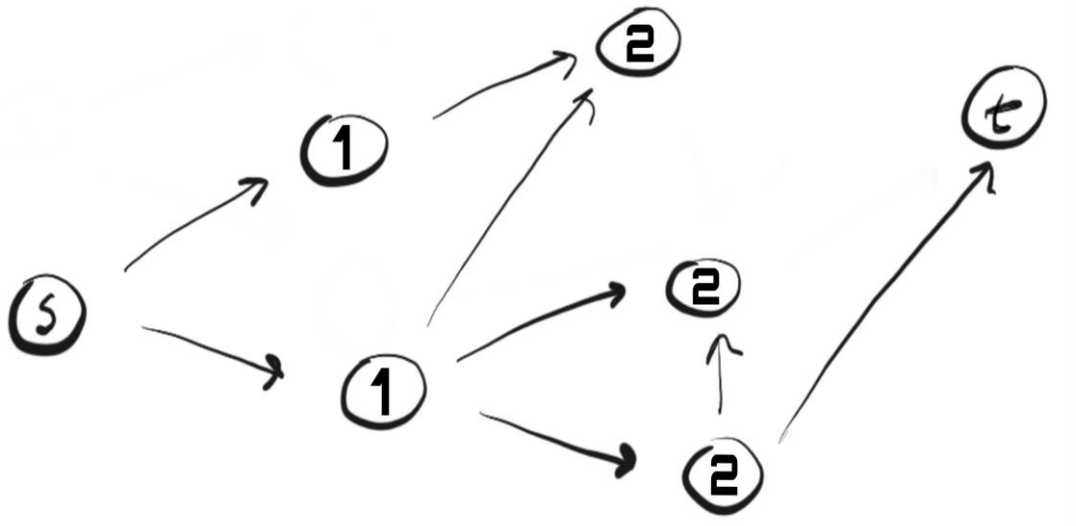

Рисунок 2.3 – Послідовність перевірки вершин графу під час виконання пошуку у ширину

Пошук у ширину ефективно працює, якщо перехід між сусідніми точками завжди займає однакову кількість часу, або іншого параметра, за яким необхідно мінімізувати шлях. У іншому разі цей алгоритм аналогічен алгоритму Дейкстри [25].

2.2.2 Алгоритм Дейкстри

Алгоритм Дейкстри працює з моделями, в яких відстані між точками можуть бути різними [25]. Від пошуку в ширину алгоритм Дейкстри відрізняє кілька особливостей. Крім збереження перевірених точок, зберігається ще й кількість кроків, витрачених на те, щоб дістатися до них. Точки виключаються з наступної перевірки лише у випадку, якщо попередній знайдений шлях до неї займав меншу кількість кроків. Точки у списку розглядаються не по порядку, спочатку вибираються ті, шлях до яких менший [25]. Алгоритм виглядає наступним чином:

- встановлення стартової точки та відстані до неї у вигляді «0» до списку «на перевірку» та до списку «досліджені»;
- вибір зі списку «на перевірку» точки з найменшою відстанню до неї та видалення її зі списку;
- знаходження всіх сусідніх доступних точок та підсумування відстані від обраної точки до них з відстанню до обраної точки. Додавання цієї інформації до списку «поточні»;
- видалення зі списку «поточні» точок зі списку «досліджені», відстань до яких більша, ніж значення відстані у списку «досліджені»;
- очищення списку «на перевірку» та поміщення туди всіх точок зі списку «поточні». Поміщення цих точок до списку «досліджені» та очищення поточного списку.

Алгоритм повторюється з кроку 2, поки не буде знайдено кінцеву точку або всі доступні точки не будуть перевірені [25] Блок схему роботи алгоритму Дейкстри наведено на рисунку 2.4.

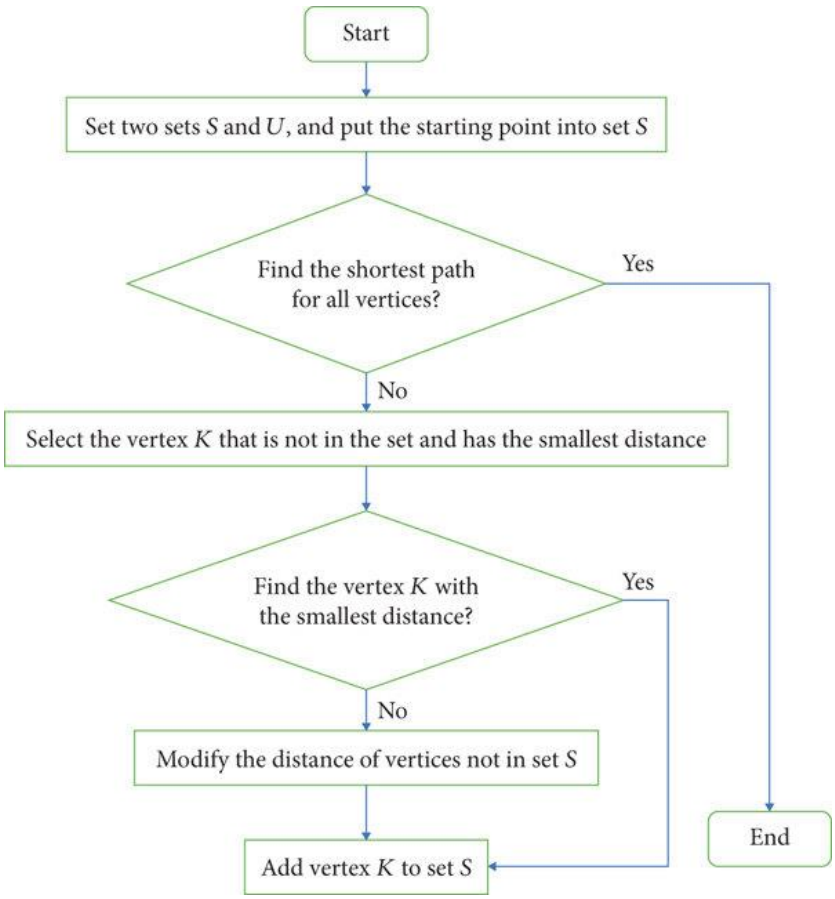

Рисунок 2.4 – Блок схема роботи алгоритму Дейкстри

Недоліком як алгоритму пошуку в ширину, так і алгоритму Дейкстри є те, що пошук шляху відбувається на всі боки відразу та витрачається час на безперспективні шляхи, які віддаляють від необхідної точки, а не наближають до неї [2, 25].

2.2.3 Алгоритм пошуку A*

Недоліки алгоритмів пошуку в ширину і Дейкстри можуть бути усунені завдяки використанню евристик. Евристичний алгоритм працює так само, як і алгоритм Дейкстри, але з однією зміною. При виборі наступної точки на розгляд першою вибирається не та точка, яка ближче до початку шляху, а та, що ближче до кінця [26]. Відстань до кінця розраховується приблизно, наприклад, як відстань між двома точками на карті. Недолік алгоритму в

тому, що знайдений шлях необов’язково буде найкоротшим. Натомість його знаходження вимагатиме менше часу [25].

Алгоритм A* містить у собі властивості обох алгоритмів одразу, через що він знаходить найкоротшу відстань, але робить це за менший час, ніж алгоритм Дейкстри [26]. Це здійснюється завдяки тому, що наступна точка на розгляд вибирається за мінімальною сумою відстаней до початку та до кінця шляху (рисунок 2.5).

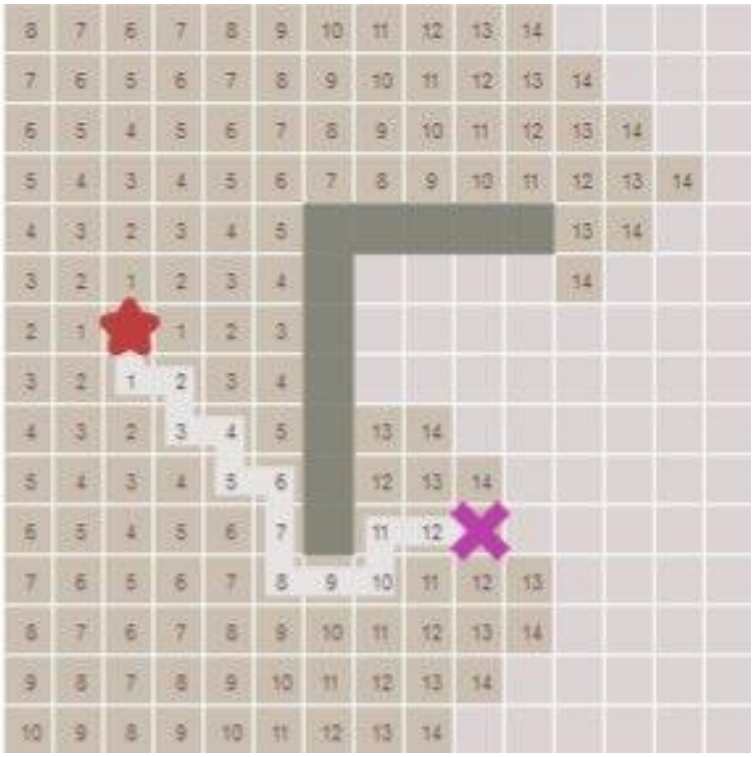

Рисунок 2.5 – Приклад оцінки клітин під час виконання алгоритму A*

Існують інші алгоритми, які мають переваги за різних умов. У іграх є багато аспектів, які можуть ускладнити алгоритм [25]. Наприклад, може бути кілька варіантів переміщень, повороти, перешкоди можуть рухатись, можуть бути вузькі місця. Кожна така умова уповільнює роботу алгоритму. Через це доводиться спрощувати алгоритм або змінювати його.

3 ПРОГРАМНАЯ РЕАЛІЗАЦІЯ

3.1 Процес генерації лабіринту

На першому етапі генерації лабіринту потрібно перш за все визначити його розмір та форму. Під час ініціалізації цієї структури обирається початкова точка для генерації. Генерація проходів виконується різними методами, вибір яких залежить від типу лабіринту, його розміру та інших параметрів. Якщо в процесі генерації виникає блокування, алгоритм повинен опрацювати цю ситуацію, повертаючись до попередньої клітини і намагаючись знайти інший маршрут. Після створення всіх проходів лабіринт вважається завершеним.

Описаний процес генерації лабіринту є універсальним і може бути адаптований під різні алгоритми генерації лабіринтів, такі як наївний алгоритм, алгоритм Еллера, рекурсивний метод, клітинний автомат та інші.

В процесі розробки тестового програмного середовища для генерації лабіринтів описано інтерфейс, який наслідують усі реалізовані методи генерації. Генерацією керує окремий компонент MazeGenerator, у якому визначаються розмірності лабіринту, а також координати старту та фінішу у лабіринті. Перерахування (enum) GenerationMethod містить у собі реалізовані методи генерації, серед яких можна обрати необхідний для тестування. Спочатку виконується генерація лабіринту обраним методом та виводиться у консоль попередження, якщо обраний метод не реалізовано. Потім алгоритм встановлює стіни навколо лабіринту та розставляє точки старту та фінішу. За необхідністю аналізу згенерованого лабіринту є можливість його виводу у консоль, що доцільно для огляду невеликих лабіринтів.

Статичний метод GetMaze() надає можливість іншим скриптам отримати доступ до згенерованого лабіринту та використовується для передачі лабіринту скриптам, які відповідають за пошук шляху.

3.1.1 Генерація лабіринту методом бінарного дерева

Для реалізації методу бінарного дерева розроблено скрипт BinaryTreeMazeGenerator, котрий реалізує інтерфейс IMazeGenerator, приймає параметри довжини та ширини лабіринту та повертає згенерований лабіринт. Через те, що вирішено генерувати лабіринти, у яких прохід та стіна мають однакові розміри, кількість клітин лабіринту повинна бути непарна. При спробі зазначити розмірності лабіринту парними значеннями, вони будуть збільшені на одиницю. Після цього створюється масив, у якому буде зберігатися згенерований лабіринт, та заповнюється стінами.

Алгоритм перевіряє кожну клітину у лабіринті та випадковим чином видаляє правий або верхній кордони. Після цього видаляється усі стіни справа та зверху (лістинг 3.1).

Лістинг 3.1 – Процес генерації лабіринту методом бінарного дерева

```
for (int x = 2; x < height; x+=2)
{
     for (int y = 0; y < width-1; y+=2)
     {
          maze[x, y] = 0;
          if (Random.Range(0, 2) == 0)
            maze[x, y + 1] = 0; // Open the passage on the right
          else
            maze[x - 1, y] = 0; // Open the passage on the top
     }
}
// Removing the walls at the edges
for (int i = 0; i < height; i++)
{
    maze[0, i] = 0;
}
for (int i = 1; i < width; i++)
{
    maze[i, width - 1] = 0;
}
```

Після виконання усіх обчислювальних операцій скрипт передає згенерований алгоритм у компонент MazeGenerator.

3.1.2 Реалізація методу Олдоса-Бродера

Алгоритм Олдоса-Бродера передбачає випадкове блукання під час генерації лабіринту. Для цього створено скрипт AldousBroderMazeGenerator, який реалізує інтерфейс IMazeGenerator.

На початку виконання алгоритм додає одиницю до параметрів розмірності лабіринту, якщо вона є парною. Після цього він створює масив для зберігання лабіринту та заповнює його стінами. Також розраховується загальна кількість можливих клітин у лабіринті. Початкова точка генерації є випадковою та визначається за допомогою функції вбудованої бібліотеки UnityEngine Random.Range(int MinInclusive, int MaxExlusive), яка повертає випадкове значення, зазначене у заданому діапазоні. Для визначення напрямів випадкового блукання створюється двовимірний масив directions (лістинг 3.2), який містить у собі усі доступні напрямки.

Лістинг 3.2 – Двовимірний масив, що визначає напрями

```
// A collection that contains possible directions of movement
int[][] directions = new int[][] {
    new int[] { 0, -2 }, // up
    new int[] { 0, 2 },  // down
    new int[] { -2, 0 }, // left
    new int[] { 2, 0 }   // right
};
```

Далі обирається випадковий напрямок з створеного раніше масиву напрямків та обирається наступна клітина. Якщо наступна клітина не знаходиться поза межами та не є відвіданою, стіна між поточної та наступною клітиною видаляється, до загальної кількості відвіданих клітин додається одиниця, а координати клітини, що розглядалася стають поточними координатами, які будуть далі розглядатися під час виконання алгоритму. Цей процес відбуватиметься доки усі клітини у лабіринті не будуть відвідані (лістинг 3.3).

Лістинг 3.3 – Основний цикл генерації лабіринту

```
while (visitedCells < totalCells) {
    int[] direction = directions[UnityEngine.Random.Range(0,
directions.Length)];
    int nextX = currentX + direction[0];
    int nextY = currentY + direction[1];
    if (IsInBounds(nextX, nextY, width, height))
    {
        if (maze[nextX, nextY] == 1)
        {
            maze[currentX + direction[0] / 2, currentY +
direction[1] / 2] = 0;
            maze[nextX, nextY] = 0;
            visitedCells++;
        }
        currentX = nextX;
        currentY = nextY;
    }
}
```

Для визначення чи знаходиться обрана клітина у межах лабіринту створено окрему функцію private bool IsInBounds (int x, int y, int width, int height). Функція приймає у якості вхідних параметрів координати клітини, що розглядається, а також довжину та ширину лабіринту. Після виконання вона повертає значення true, якщо клітина знаходиться у межах лабіринту, або значення false, якщо координати клітини, що розглядається, знаходяться по за межами лабіринту.

Після виконання усіх дій лабіринт буде згенерований методом Олдоса-Бродера та зберігатиметься у змінній maze. Ця змінна передає своє значення у метод, з якого був зроблений виклик.

3.1.3 Використання методу Еллера для генерації лабіринту

Метод Еллера генерує лабіринт коригуючи по черзі кожну строку та об’єднуючи клітини у групи. Це дозволяє зменшити час виконання при генерації великих лабіринтів та використовувати меншу кількість пам’яті під час генерації.

Для реалізації алгоритму Еллера створено клас EllerMazeGenerator. Він наслідує інтерфейс IMazeGenerator та повертає згенерований лабіринт у скрипт, з якого здійснено виклик. Для перевірки на необхідність об’єднання множин створено змінну isConnection типу System.Random.

Після запуску скрипту розмірності будуть збільшені за необхідністю, а також буде створено двовимірний масив для зберігання лабіринту.

Для покрокової обробки кожної строки використовується одновимірний масив sets. Кількість елементів у масиві дорівнює кількості елементів у кожній строчці. Також додано змінну nextSet типу int, яка відповідає за встановлення відповідних множин відповідним елементам. Значення цієї змінної під час запуску скрипту дорівнює одиниці та збільшується після створення кожної нової множини (лістинг 3.4).

Лістинг 3.4 – Створення необхідних для виконання алгоритму змінних та присвоєння власних множин елементам першої строки

```
// Initial structure of sets for rows
int[] sets = new int[(width / 2) + 1];
int nextSet = 1;

// Create new sets if needed
for (int x = 0; x < sets.Length; x++)
{
    if (sets[x] == 0) sets[x] = nextSet++;
}
```

Після присвоєння елементам першої строки відповідних множин виникає необхідність у горизонтальних зв’язках між множинами. Поступово оброблюється кожна клітина у кожній строчці. Рішення щодо об’єднання множин є випадковим, але якщо множини поточної та наступної клітини співпадають, то між ними потрібно встановити стіну. Це необхідно для уникнення створення більше ніж одного проходу до однієї і тієї самої клітини. Якщо вирішено об’єднати множини, необхідно кожному елементу однієї множини присвоїти іншу. Для цього використовується допоміжна

змінна targetSet типу int, Вона зберігає номер однієї з множин, які об’єднуються. У випадку якщо вирішено не об’єднувати множини, між ними встановлюється стіна (лістинг 3.5).

Лістинг 3.5 – Створення горизонтальних з’єднань

```
for (int x = 0; x < sets.Length - 1; x++){
    if (isConnection.Next(2) == 1 || sets[x] == sets[x + 1]) {
        // Put up a wall
        maze[y, (x * 2) + 1] = 1;
    }
    else {
        // Combine sets
        int targetSet = sets[x + 1];
        for (int i = 0; i < sets.Length; i++) {
            if (sets[i] == targetSet) sets[i] = sets[x];
        }
    }
}
```

Перед початком створення вертикальних зв’язків потрібно перевірити, чи не є поточний рядок останнім. Якщо рядок є останнім то потрібно видалити стіни між усіма множинами для уникнення клітин, до яких неможливо дістатись, та забезпечити гарантований зв’язок між усіма клітинами у лабіринті. Для виконання цієї дії потрібно розглянути кожен елемент останньої строки, та якщо множини елементів різняться, видалити стіну між ними та об’єднати множини. Після виконання цих дій потрібно завершити цикл генерації та повернути згенерований лабіринт до компоненту, з якого було здійснено виклик (лістинг 3.6).

Лістинг 3.6 – Обробка останнього рядка

```
// If this is the last line, join all sets
if (y + 2 >= height)
{
    for (int x = 0; x < sets.Length - 1; x++)
    {
        if (sets[x] != sets[x + 1]) {
```

```
                maze[y, (x * 2) + 1] = 0; // Remove the wall
                int targetSet = sets[x + 1];
                for (int i = 0; i < sets.Length; i++)
                {
                    if (sets[i] == targetSet) sets[i] = sets[x];
                }
            }
        }
        break; // End the row processing loop
    }
```

Якщо поточний рядок не є останнім, необхідно додати вертикальні зв’язки. Для цього використовується колекція HashSet<int>, яка зберігає у собі множини, які вже мають вертикальний зв’язок.

Якщо було вирішено не додавати зв’язок або множина, до якої належить клітина, вже має зв’язок між клітинами встановлюється стіна. В іншому випадку стіна не встановлюється (лістинг 3.7). Після виконання усіх дій, клітинам наступної строки, які мають верхні границі отримують присвоюється нові множини.

Лістинг 3.7 – Створення вертикальних з’єднань

```
var connectedSets = new HashSet<int>();
for (int x = 0; x < sets.Length; x++)
{
    if (isConnection.Next(2) == 0 ||
connectedSets.Contains(sets[x]))
    {
        // Put up a wall
        maze[y + 1, x * 2] = 1;
    }
    else
    {
        // Guarantee connection for a set
        connectedSets.Add(sets[x]);
    }
}
```

Генерація продовжується доки усі строки лабіринту не будуть оброблені. Після цього згенерований лабіринт передається компоненту, з якого був зроблений виклик.

3.2 Реалізація алгоритмів пошуку шляху

На першому етапі пошуку шляху необхідно визначити стартову та кінцеву точки на заданій карті, графу або лабіринту. Під час ініціалізації цієї структури обирається підхід для пошуку. Він може бути заснований на певному алгоритмі, такому як пошук у глибину, пошук у ширину, алгоритм Дейкстри або A*. Пошук виконується шляхом перевірки можливих сусідніх вузлів або клітин. Якщо шлях не знайдено через поточний маршрут, алгоритм повинен повернутися до попередньої точки й обрати інший варіант. Після знаходження найкоротшого шляху, або шляху який задовольняє заданим умовам, процес вважається завершеним.

Описаний підхід до пошуку шляху є універсальним і може бути адаптований для різних задач, таких як навігація в лабіринтах, розв’язання задачі комівояжера, планування маршрутів та інші сценарії.

Для реалізації алгоритмів пошуку шляху створено інтерфейс IPathfindingAlgorithm, який наслідуватимуть усі реалізовані алгоритми пошуку шляху.

Для керуванням процесом пошуку шляху у лабіринтах створено окремий скрипт PathfindingController, з якого виконується виклик усіх розроблених методів пошуку шляху. У цьому скрипту міститься дві колекції Dictionary, в одному з яких зберігається посилання на алгоритми пошуку, а в іншому найдені шляхи відповідними алгоритмами.

Після запуску скрипту словник з алгоритмами заповнюється посиланнями на розроблені алгоритми та ініціалізується змінна для зберігання найдених шляхів. Двовимірний масив отримує та зберігає лабіринт завдяки методу GetMaze() компоненту MazeGenerator.

Якщо лабіринт ще не було згенеровано, то у консоль буде виведено відповідне попередження, у іншому випадку буде викликана функція RunPathfinding(int[,] maze), яка запускає пошук шляху у лабіринті. У якості вхідного параметру функція приймає лабіринт, у якому буде здійснюватися

пошук. Вона по черзі визиває кожен з методів пошуку та виводить у консоль результати пошуку (лістинг 3.8).

Лістинг 3.8 – Функція RunPathfinding

```
private void RunPathfinding(int[,] maze)
{
    foreach (var algorithm in algorithms)
    {
        Debug.Log($"Start the algorithm: {algorithm.Key}");
        List<Vector2Int> path = algorithm.Value.FindPath(maze);
        paths[algorithm.Key] = path;
        if (path.Count > 0)
        {
            Debug.Log($"Path found using {algorithm.Key}:");
            foreach (var point in path)
            {
                Debug.Log(point);
            }
        }
        else
        {
            Debug.Log($"Path not found using {algorithm.Key}.");
        }
    }
}
```

3.2.1 Реалізація методу пошуку шляху у ширину

Реалізацію алгоритму пошуку шляху у ширину описано у окремому класі BreadthFirstSearch, який наслідує інтерфейс IpathfindingAlgorithm. Перед початком пошуку шляху скрипт отримує довжину та ширину лабіринту, оголошує змінні для зберігання точок старту та фінішу у лабіринті, а також присвоює їм відповідні координати. Якщо точка старту або точка фінішу не буда знайдена у лабіринті, у консоль буде виведено відповідне попередження.

Далі визначається функція для перевірки меж лабіринту, яка визначає, чи знаходиться задана точка всередині лабіринту. Це важливо для того, щоб уникнути виходу за межі масиву під час перевірки сусідніх клітин. Також

створюється масив, який містить чотири можливі напрями руху, які використовуються для перевірки сусідніх клітинок кожної поточної точки.

Для ініціалізації пошуку у ширину створюється черга, до якої додається стартова точка, та словник cameFrom, який зберігає, клітину, з якої клітинки існує шлях до кожної іншої, для відновлення шляху. Стартова точка має значення null, оскільки це початок маршруту. Основний цикл генерації виконується поки черга не буде порожня. Обробляється поточна клітинка, якщо вона є цільовою точкою, відновлюється шлях, використовуючи словник cameFrom, додаючи кожну точку до списку path і повертаючи його після виконання усіх дій (лістинг 3.9).

Лістинг 3.9 – Основний цикл генерації методом пошуку у ширину

```
Queue<Vector2Int> queue = new Queue<Vector2Int>();
queue.Enqueue(start);
Dictionary<Vector2Int, Vector2Int?> cameFrom = new
Dictionary<Vector2Int, Vector2Int?>
{
    [start] = null
};
Vector2Int current = queue.Dequeue();
// If the goal is reached, restore the path
if (current == goal)
{
    List<Vector2Int> path = new List<Vector2Int>();
    while (current != null && cameFrom.ContainsKey(current))
    {
        path.Add(current);
        current = cameFrom[current].GetValueOrDefault();
    }
    path.Reverse();
    return path;
}
```

Для кожної сусідньої клітинки перевіряється, чи знаходиться вона в межах лабіринту та чи не є ця клітина стіною. Також перевіряється чи була вже ця клітина відвідана. Якщо всі умови виконані, клітинка додається до черги, а в словнику зберігається інформація про те, що шлях до неї веде з поточної точки.

Після виконання усіх дій у консоль буде виведено інформацію що шлях знайдено, а відповідний маршрут буде передано до компоненту, з якого було здійснено виклик.

3.2.2 Алгоритм Дейкстри

На відміну від черги Queue<Vector2Int>, яка використовувалась під час реалізації пошуку у ширину, алгоритм Дейкстри використовує пріоритетну чергу SortedSet<(int distance, Vector2Int)>, де вузли обробляються відповідно до найменшої відстані. Для зберігання мінімальної відстані від початкової точки до кожної клітинки використовується словник distances, початкова дистанція дорівнює нулю. Під час виконання алгоритму кожне оновлення відстані до сусідів порівнює нову відстань із уже відомою. Якщо нова відстань менша, вона оновлюється (лістинг 3.10).

Лістинг 3.10 – Пошук шляху методом Дейкстри

```
foreach (Vector2Int direction in directions)
{
  Vector2Int neighbor = current + direction;
  if (IsInBounds(neighbor) && maze[neighbor.x, neighbor.y] != 1)
    {
      int newDistance = currentDistance + 1;
      if (!distances.ContainsKey(neighbor) || newDistance < 
distances[neighbor])
      {
          // Update the distance to the neighbor
          if (distances.ContainsKey(neighbor))
          {
              // Remove the old entry from the queue
              priorityQueue.Remove((distances[neighbor], 
neighbor));
          }
          distances[neighbor] = newDistance;
          cameFrom[neighbor] = current;
          priorityQueue.Add((newDistance, neighbor));
      }
    }
}
```

Для сортування елементів у колекції SortedSet за їх відстанню або за іх координатами, якщо значення відстані є однаковим розроблено приватний клас DistanceComparer який реалізує інтерфейс IComparer<(int distance, Vector2Int point)>. Цей клас забезпечує правильний порядок елементів у колекції SortedSet, для швидкого знаходження та обробки вузла з мінімальною відстанню від початкової точки при виконанні алгоритму Дейкстри.

Метод Compare порівнює два елементи типу (int distance, Vector2Int point), щоб визначити їх порядок у SortedSet. Спочатку відбувається порівняння за відстанню та якщо відстань у першого елемента менша, метод повертає від`ємне значення, якщо більша – додатне. Якщо відстані однакові, порівняння відбувається за координат точки. Спочатку порівнюються значення координати x, та якщо вони різні, результат порівняння визначається на основі цих координат. Якщо координати x однакові, порівнюються координати y. Це забезпечує унікальність елементів навіть за однакових відстаней (лістинг 3.11).

Лістинг 3.11 – Компаратор для колекції SortedSet

```
// Comparator for SortedSet
private class DistanceComparer : IComparer<(int distance,
Vector2Int point)>
{
    public int Compare((int distance, Vector2Int point) a, (int
distance, Vector2Int point) b)
    {
      int distanceComparison = a.distance.CompareTo(b.distance);
      if (distanceComparison == 0)
      // If the distances are equal, compare by point for
uniqueness
      {
          return a.point.x == b.point.x ?
a.point.y.CompareTo(b.point.y) : a.point.x.CompareTo(b.point.x);
      }
      return distanceComparison;
    }
}
```

3.2.3 Програмна реалізація алгоритму A*

Програмний код, який реалізує алгоритм A* наведено у файлі AStarSearch, відповідний клас у якому є спадкоємцем інтерфейсу IPathfindingAlgorithm. Після його виконання знайдений шлях буде повернуто до методу, з якого зроблено виклик. На початку скрипту розраховуються розмірності лабіринту, оголошуються змінні, для зберігання координат клітин старту та фінішу у лабіринті, та присвоюється їхні значення. Для перевірки знаходження клітин, що розглядаються, у межах лабіринту реалізована функція bool IsInBounds(Vector2Int pos), а для зберігання напрямків руху – словник directions.

Для покращення ефективності пошуку шляху алгоритмом A* використовується Манхеттенська евристика, яка є методом оцінки відстані між двома точками на сітці. Вона є особливо ефективною для сіткових карт, де рух обмежений горизонтальними та вертикальними напрямками. Використання Манхеттенської евристики в A* значно зменшує кількість вузлів, які потрібно дослідити, порівняно з іншими методами. Це призводить до швидшого знаходження шляху у великих лабіринтах.

На початку кожної ітерації основного цикл пошуку шляху, обирається вузол з найнижчим пріоритетом. Якщо цей вузол є цільовим, алгоритм відновлює шлях від початку до цілі, використовуючи словник cameFrom, і повертає цей шлях.

Якщо поточний вузол не є ціллю, алгоритм досліджує його сусідів. Для кожного допустимого сусіда в межах лабіринту обчислюється нове значення вартості шляху. Якщо це значення краще за попереднє або якщо сусід ще не був досліджений, оновлюється інформація про цей вузол. Новий показник оцінювальної функції обчислюється як сума вартості шляху та евристичної оцінки відстані до цілі. Після цього вузол додається до списку для подальшого дослідження (лістинг 3.12).

Лістинг 3.12 – Обробка сусідніх клітин під час пошуку шляху методом A*

```
// Processing neighbors
foreach (Vector2Int direction in directions)
{
   Vector2Int neighbor = current + direction;
   if (IsInBounds(neighbor) && maze[neighbor.x, neighbor.y] !=
1)
   {
      int tentativeGScore = gScore[current] + 1;
      if (!gScore.ContainsKey(neighbor) || tentativeGScore <
gScore[neighbor])
      {
         gScore[neighbor] = tentativeGScore;
         int fScore = tentativeGScore + Heuristic(neighbor,
goal);
         openList.Add((neighbor, fScore));
         cameFrom[neighbor] = current;
      }
   }
}
```

Описаний процес продовжується, доки не буде знайдено шлях до цілі або не буде вичерпано всі можливі шляхи. Алгоритм ефективно знаходить оптимальний шлях, уникаючи непотрібного дослідження менш перспективних напрямків.

4 АНАЛІЗ ПРОВЕДЕНИХ ДОСЛІДЖЕНЬ

4.1 Використання методів генерації лабіринтів

Для аналізу лабіринтів, згенерованих різними методами генерації, реалізовано застосунок за допомогою середовища розробки Unity та мови програмування C#. Розроблене програмне забезпечення дозволяє виконувати генерацію лабіринту заданого розміру обраним алгоритмом та здійснювати пошук шляху у згенерованому лабіринті методом бінарного дерева, алгоритмом A*, а також методом Дейкстри.

Наявна можливість виводу у консоль згенерованого лабіринту (рисунок 4.1), часу, витраченого на його генерацію та часу, витраченого алгоритмами пошуку шляху для знаходження найкоротшого маршруту.

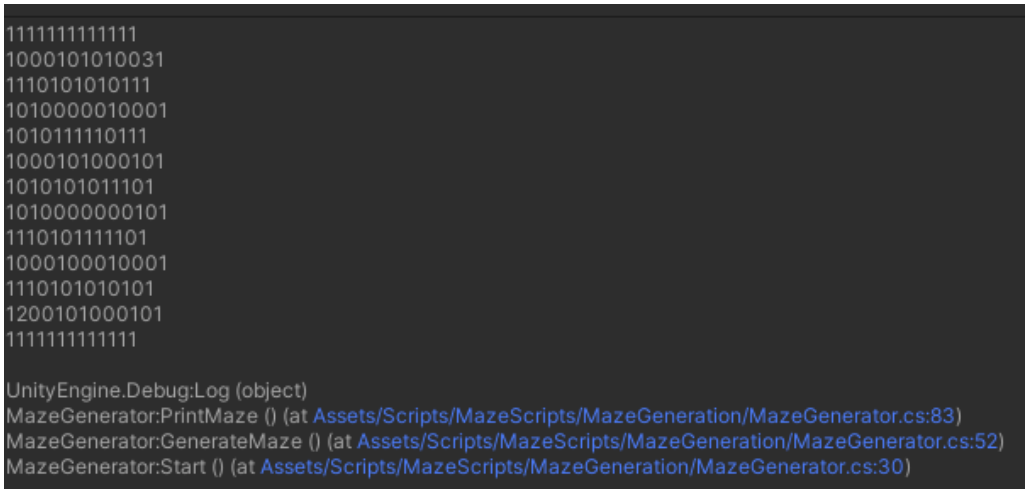

Рисунок 4.1 – Відображення у консолі згенерованого лабіринту

Для відстеження часу, який витрачають алгоритми під час виконання експериментів використана бібліотека System.Diagnostics, в якій передбачені класи, що дозволяють взаємодіяти з системними процесами, журналами подій та виконувати діагностику програмних застосунків.

Основні тести були виконані у лабіринтах розміром 1000x1000, 10000x10000 та 50000x50000 одиниць. Результати проведених експериментів наведено у таблиці 4.1

Таблиця 4.1 – Витрачений час на генерацію лабіринтів розміром 1000x1000, 10000x10000 та 50000x50000 одиниць

Назва алгоритму	Час генерації лабіринтів різної розмірності, мс		
	1000x1000	10000x10000	50000x50000
Метод бінарного дерева	13	1386	24256
Алгоритм Еллера	150	61527	1055954
Алгоритм Олдоса-Бродера	397	80767	2068884

Як показали результати тестів, метод бінарного дерева генерує лабіринти швидше за інші завдяки простоті алгоритму та відсутності необхідності у додаткових розрахунках під час генерації. Також лабіринти, згенеровані алгоритмом бінарного дерева мали помітне діагональне зміщення, що призводить до спрощення вирішення такого лабіринту.

Ілюстрація порівняльного аналізу часу роботи методу бінарного дерева, алгоритмів Еллера та Олдоса-Бродера наведена на рисунку 4.2.

Рисунок 4.2 – Діаграми витраченого алгоритмами часу на генерацію лабіринтів відповідних розмірів

Результатом застосування алгоритму Еллера є генерація випадкових лабіринтів, але за більший час. Метод Олдоса-Бродера генерував лабіринти високої складності, але витрачав досить велику кількість часу через перевірки клітин, які вже були застосовані для генерації.

4.2 Порівняльний аналіз методів пошуку шляху

Для порівняння алгоритмів пошуку шляху потрібно забезпечити однакові умови пошуку через те, що складність лабіринтів, згенерованих різними алгоритмами генерації, може відрізнятись. Для достовірності аналізу вирішено порівнювати методи пошуку окремо для лабіринтів, згенерованих кожним методом генерації.

Оскільки всі згенеровані лабіринти є ідеальними, то шлях рішення (обходу лабіринту) можливий тільки один, таким чином знайдений маршрут для різних методів пошуку буде однаковим. Через це аналіз знайдених шляхів доцільно оцінювати тільки за витраченим часом на виконання, який вимірюється у мілісекундах, не враховуючи довжину маршруту, яка буде однаковою для всіх методів пошуку шляху. Результати для лабіринтів розміром 1000x1000 одиниць наведено у таблиці 4.2.

Таблиця 4.2 – Час пошуку шляху для лабіринтів створених різними методами

	Час пошуку при використанні різних алгоритмів генерації лабіринту, мс		
Алгоритм пошуку шляху	Алгоритм бінарного дерева	Алгоритм Еллера	Алгоритм Олдоса-Бродера
Метод пошуку у ширину	21	1579	852
Алгоритм Дейкстри	43	5077	2947
Метод A*	9	7149	2186

У лабіринті розміром 1000*1000 одиниць, згенерованим методом бінарного дерева на пошук шляху було витрачено найменшу кількість часу. Це пов’язано з низькою складністю згенерованого лабіринту. Алгоритм A* показав найліпші результати, завдяки наявності у створеному за алгоритмом бінарного дерева лабіринті діагонального зміщення, завдяки якому евристика методу є достатньо ефективною.

У лабіринті, згенерованим алгоритмом Еллера, для знаходження шляху знадобилося значно більше часу. Для пошуку шляху методом A* була витрачена найбільша кількість часу, це пов’язано з особливістю реалізації алгоритму генерації.

Як у лабіринті, згенерованим методом Еллера, так і у лабіринті, створеним алгоритмом Олдоса-Бродера, метод пошуку у ширину продемонстрував найбільшу ефективність. Результати алгоритму Дейкстри є гіршими за відсутності вартості переходу між клітинами лабіринту.

Діаграми даних щодо часу, витраченого на пошук шляху у лабіринті 1000x1000 одиниць, створеному різними методами, наведено на рисунку 4.3.

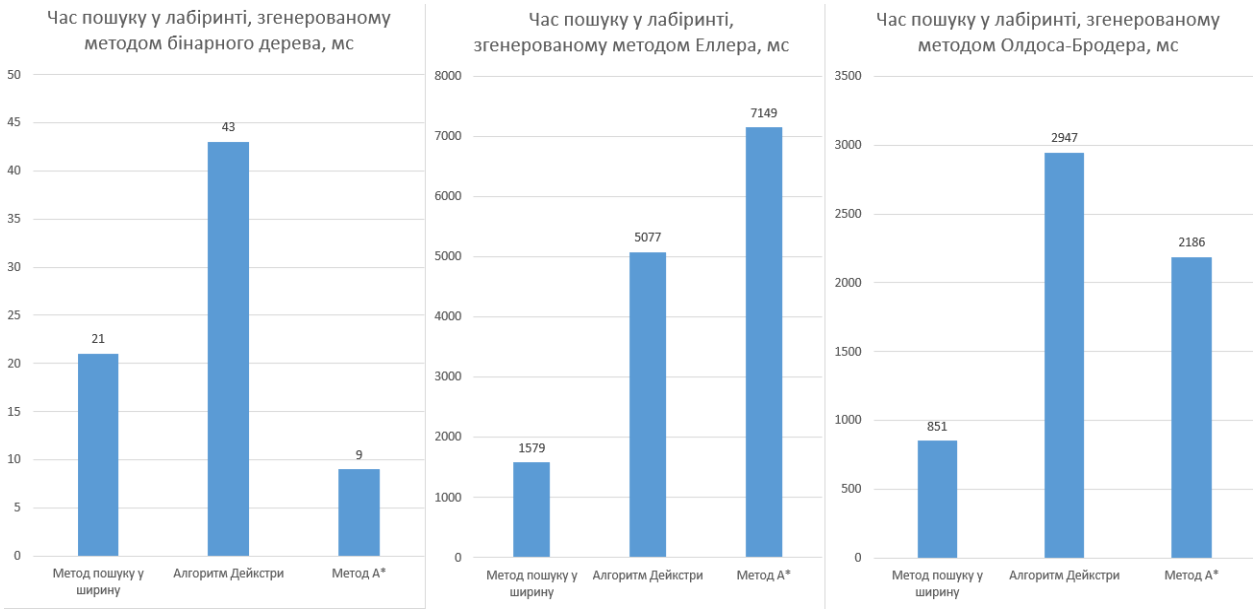

Рисунок 4.3 – Діаграми роботи алгоритмів пошуку шляху у лабіринтах розміром 1000x1000 одиниць

Проведене дослідження ефективності роботи зазначених алгоритмів пошуку шляху за витраченим часом у лабіринтах розмірністю 10000x10000 одиниць.

При використанні лабіринту, створеного методом бінарного дерева, алгоритм A* показав найкоротший час, але у інших лабіринтах його результати є найгіршими. Для знаходження шляху у лабіринті, створеному методом Олдоса-Бродера, алгоритму знадобилось 267,5 хвилин. Результати тестів наведені у таблиці 4.3.

Таблиця 4.3 – Час пошуку шляху для лабіринтів створених різними методами

	Час пошуку при використанні різних алгоритмів генерації лабіринту, мс		
Алгоритм пошуку шляху	Алгоритм бінарного дерева	Алгоритм Еллера	Алгоритм Олдоса-Бродера
Метод пошуку у ширину	2841	3738884	3940872
Алгоритм Дейкстри	5913	7852283	8754335
Метод A*	979	15753157	16053608

Як і у випадку з лабіринтом розмірністю 1000x1000, алгоритм бінарного дерева генерує низькоякісні лабіринти з наявністю діагонального зміщення, для вирішення яких потрібна найменша кількість витраченого часу. Для більш складних лабіринтів метод пошуку у ширину демонструє найліпші результати з малою кількістю витраченого часу за умови, що лабіринти є ідеальними та вартість переходу між вершинами дорівнює нулю. Алгоритм Олдоса-Бродера продемонстрував найвищу складність згенерованих лабіринтів, для проходження яких алгоритми пошуку шляху витрачають найбільшу кількість часу.

Графічне відображення впливу методів, за якими були створені лабіринти, на необхідну для пошуку шляху кількість часу наведено на рисунку 4.4.

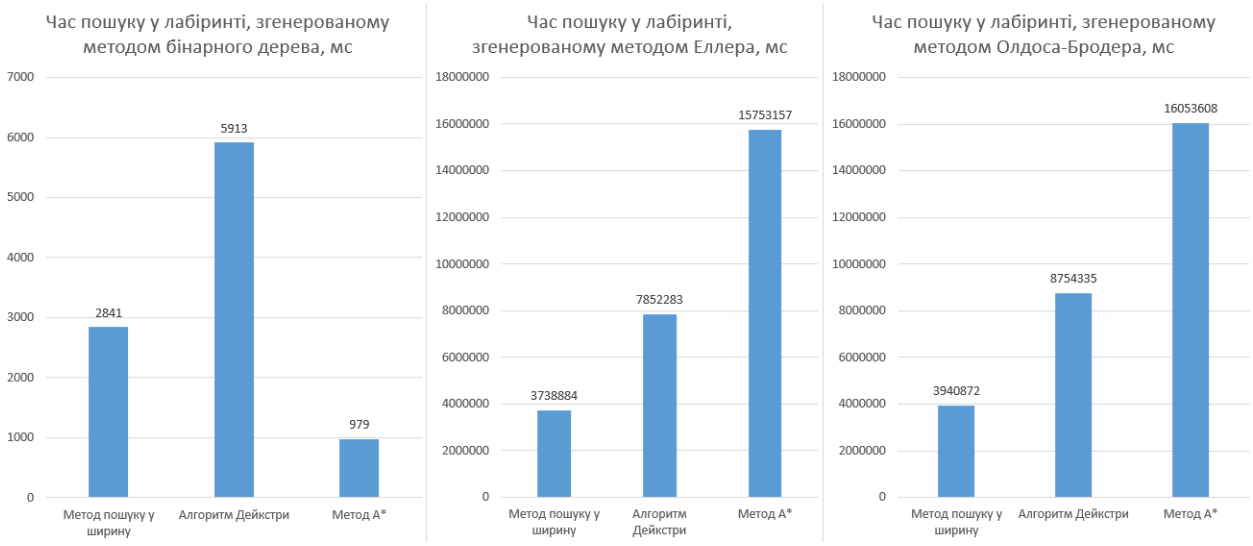

Рисунок 4.4 – Залежність витрачаємого алгоритмами пошуку шляху часу від методів, за якими були створені лабіринти

На представленій діаграмі показано, що для складних лабіринтів великого розміру алгоритми пошуку демонструють майже однаковий час знаходження найкоротшого шляху. Однак цей показник може варіюватися внаслідок випадкового характеру згенерованих лабіринтів.

Порівняння алгоритмів пошуку шляху у лабіринті розміром 50000x50000 одиниць проводилось тільки у лабіринті, згенерованим методом бінарного дерева. Таке рішення було прийнято через занадто велику кількість необхідного часу для тестування, який за розрахунками може складати більш одного місяця. Результати тестування алгоритмів пошуку шляху наведені у таблиці 4.4.

Таблиця 4.4 – Результати дослідження у лабіринті розміром 50000x50000 одиниць, створеного за допомогою метода бінарного дерева

Алгоритм пошуку шляху	Алгоритм бінарного дерева
Метод пошуку у ширину	19645 мс
Алгоритм Дейкстри	40627 мс
Метод А*	16052 мс

Оскільки генерація методом бінарного дерева займає достатньо малий час та генерує лабіринти, пошук найкоротшого шляху у яких здійснюється швидко, наявна можливість їх дослідження. Аналіз результатів роботи алгоритмів пошуку шляху у зазначеному лабіринті підтверджує низький рівень складності лабіринтів, згенерованих методом бінарного дерева та доводить ефективність застосування методу A* для обходу лабіринтів з візуально наявним діагональним зміщенням. Також експеримент свідчить про необхідність більшої кількості часу для генерації лабіринтів більшого розміру. Діаграму результатів дослідження зображено на рисунку 4.5.

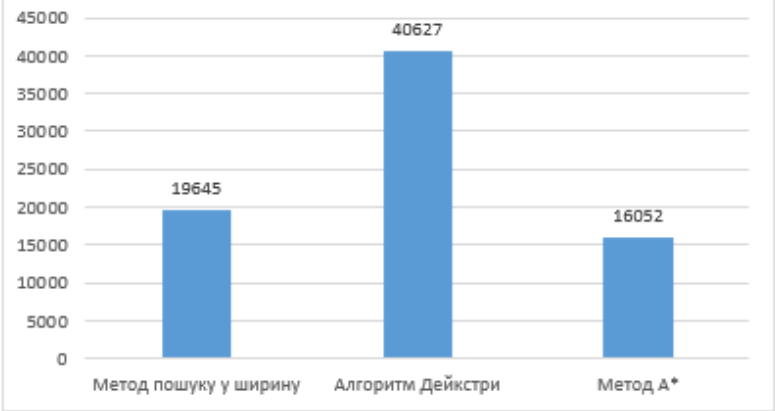

Рисунок 4.5 – Витрачений алгоритмами пошуку шляху час (мс) на знаходження маршруту у лабіринті розміром 50000x50000 одиниць

За результатами дослідження можна зазначити, що генерація методом бінарного дерева створює лабіринти низької складності, але витрачає малу кількість часу, а алгоритми Олдоса-Бродера та Еллера генерують складні лабіринти та потребують більше часових витрат. Для пошуку шляху у ідеальних лабіринтах без вартості переходу між вершинами доцільно використовувати алгоритм пошуку у ширину через відсутність складних перевірок під час виконання. Алгоритм A* може бути доцільним використовувати у лабіринтах, у яких існує більш ніж один можливий шлях.

ВИСНОВКИ

Лабіринти, як структури, що складаються з проходів і стін, мають широке застосування в різних сферах, включаючи ігрову індустрію, робототехніку та інженерію [4]. Вони можуть бути використані для створення ігрових локацій та виступати тестовими середовищами для розробки навігаційних алгоритмів. У результаті дослідження проведено детальний аналіз як методів генерації лабіринтів [23], так і алгоритмів пошуку найкоротшого шляху в них [24].

Аналіз показав, що існує велика кількість алгоритмів генерації лабіринтів, серед яких найбільш поширеними є метод бінарного дерева, алгоритм Олдоса-Бродера та алгоритм Еллера [23]. Кожен з цих методів має свої переваги та недоліки, які впливають на ефективність генерації і те, наскільки складним буде створений лабіринт.

Математично лабіринти можуть бути представлені у вигляді графів, де вершини відповідають кімнатам, а ребра – можливим переходам між ними. Це дозволяє використовувати алгоритми пошуку, такі як алгоритм Дейкстри та A*, для знаходження оптимальних маршрутів [24]. Проведений аналіз актуальних наукових досліджень показує, що комбінування різних підходів може значно зменшити час на пошук рішення.

В процесі роботи реалізоване тестове програмне середовище за допомогою мови програмування C# та технології Unity, що дозволило провести аналіз і порівняння різних алгоритмів генерації та пошуку шляхів.

Результати дослідження відкривають нові перспективи для подальших досліджень у цій галузі, зокрема в контексті розвитку автономних систем пересування та вдосконалення навігаційних технологій. У майбутньому можна очікувати подальшого вдосконалення алгоритмів генерації лабіринтів з використанням новітніх технологій машинного навчання та штучного інтелекту.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Хокінг, Д. Unity в дії. Мультиплатформенна розробка на C# [Текст] / Д. М. Хокінг. – Print2print, 2016. – 336 с. – ISBN: 978-5-4461-2266-0.

2. John, S. George, T. Path Problems in Networks [Текст] / S. B. John, T. George. – Morgan & Claypool Publishers, 2010. – 77 с. – ISBN: 978-1-5982-9924-3.

3. Buck, J. Mazes for Programmers: Code Your Own Twisty Little Passages [Текст] / Jamis Buck. – Pragmatic Bookshelf, 2015. – 288 p. – ISBN-13: 978-1680500554.

4. Autonomous Maze Solving Robot [Електронний ресурс] – Режим доступу : www/ URL: https://ieeexplore.ieee.org/document/9129943. – 03.11.2024 р. – Загол. з екрану.

5. Crafting Mazes with Graph Theory [Електронний ресурс] – Режим доступу : www/ URL: https://dev.to/optiklab/crafting-mazes-2dia. – 03.11.2024 р. – Загол. з екрану.

6. From Mazes to Complex Networks [Електронний ресурс] – Режим доступу : www/ URL: https://www.learninternetgrow.com/learning-graph-theory-mazes-complex-networks. – 03.11.2024 р. – Загол. з екрану.

7. Applications of Random Mazes and Graphs Generated via Markov Chain Monte Carlo Methods [Електронний ресурс] – Режим доступу : www/ URL: https://www.math.wustl.edu/Math350Fall2012/Projects/mathproj11.pdf. – 18.10.2024 р. – Загол. з екрану.

8. Martín-Nieto, M. Solving Mazes: A New Approach Based on Spectral Graph Theory graphs [Текст] / M. Martín-Nieto, D. Castaño, S. Horta Muñoz, D. Ruiz // Mathematics. – 2024. – Vol. 12, № 15. – С. 1–13. https://doi.org/10.3390/math12152305

9. Maze Classification [Електронний ресурс] – Режим доступу : https://puzzleu.com/class/mazes.html. – 20.10.2024 р. – Загол. з екрану.

10. Hiraoka, T. A high-speed method for generating labyrinth images using smoothing filters with different window sizes [Текст] / T. Hiraoka, H. Nonaka, Y. Tsurunari // ICIC Express Letters. – 2019. – Vol. 13, № 8. – С. 711–717.

11. Yang, K. Optimization and comparative analysis of maze generation algorithm hybrid [Текст] / K. Yang, S. Lin, Y. Dai, W. Li // Applied and Computational Engineering. – 2024. – Vol. 79, № 1. – С. 20–33. https://doi.org/10.54254/2755-2721/79/20241082

12. Study of Artificial Intelligent Algorithms Applied in Procedural Content Generation in Video Games [Електронний ресурс] – Режим доступу : https://eludamos.org/index.php/eludamos/article/view/vol10no1-4. – 04.11.2024 р. – Загол. з екрану.

13. A comparative study of A-star algorithms for search and rescue in perfect maze [Електронний ресурс] – Режим доступу : https://ieeexplore.ieee.org/document/5777723. – 04.11.2024 р. – Загол. з екрану.

14. Tjiharjadi, S. A Systematic Literature Review of Multi-agent Pathfinding for Maze Research [Текст] / S. Tjiharjadi, S. Razali, H. Asyrani Sulaiman // Journal of Advances in Information Technology. – 2022. – Vol. 13, № 4. – С. 358–367. DOI:10.12720/jait.13.4.358-367

15. Pathfinding for agents in games using a tetrahedron mesh [Електронний ресурс] – Режим доступу : https://www.researchgate.net/publication/362801751_Pathfinding_for_agents_in_games_using_a_tetrahedron_mesh. – 04.11.2024 р. – Загол. з екрану.

16. Maze Generation Algorithms - An Exploration [Електронний ресурс] – Режим доступу : www/ URL: https://professor-l.github.io/mazes/. – 04.11.2024 р. – Загол. з екрану.

17. Maze Generation: Binary Tree algorithm [Електронний ресурс] – Режим доступу : www/ URL: https://weblog.jamisbuck.org/2011/2/1/maze-generation-binary-tree-algorithm. – 20.10.2024 р. – Загол. з екрану.

18. Procedural Maze Generation using Binary Tree Algorithm [Електронний ресурс] – Режим доступу : www/ URL:

https://medium.com/@teemarveel/procedural-maze-generation-using-binary-tree-algorithm-5f17cb52a221. – 04.11.2024 р. – Загол. з екрану.

19. Peter, G. Analysis of Maze Generating Algorithms [Текст] / G. Peter // IPSI Transactions on Internet Research. – 2019/1. – Vol. 15. – p. 23–30.

20. A reverse Aldous–Broder algorithm [Електронний ресурс] – Режим доступу : https://projecteuclid.org/journals/annales-de-linstitut-henri-poincare-probabilites-et-statistiques/volume-57/issue-2/A-reverse-AldousBroder-algorithm/10.1214/20-AIHP1101.short. – 03.11.2024 р. – Загол. з екрану.

21. Generating Mazes [Електронний ресурс] – Режим доступу : www/ URL: https://healeycodes.com/generating-mazes. – 04.11.2024 р. – Загол. з екрану.

22. Maze Generation: Eller's Algorithm [Електронний ресурс] – Режим доступу : www/ URL: https://weblog.jamisbuck.org/2010/12/29/maze-generation-eller-s-algorithm. – 20.10.2024 р. – Загол. з екрану.

23. Вітко, В.О. Аналіз алгоритмів генерації лабіринтів [Текст] / В. О. Вітко, Г. С. Іващенко // Проблеми інформатизації: Тези доповідей дванадцятої міжнародної науково-технічної конференції. – 21-22 листопада 2024. – Том 2, С. 81.

24. Іващенко, Г.С. Аналіз методів пошуку шляху у лабіринтах [Текст] / Г. С. Іващенко, В. О. Вітко // Проблеми інформатизації: Тези доповідей одинадцятої міжнародної науково-технічної конференції. Черкаси – Баку – Харків – Бельсько-Бяла. – 16-17 листопада 2023. – Том 1, С. 65.

25. The 5 Most Powerful Pathfinding Algorithms [Електронний ресурс] – Режим доступу : https://www.graphable.ai/blog/pathfinding-algorithms/#:~:text=Top%205%20Pathfinding%20Algorithms%201%20Cycle%20detection%20The,algorithm%20...%205%20Minimum%20Spanning%20Trees%20algorithm%20. – 05.11.2024 р. – Загол. з екрану.

26. Easy A* (star) Pathfinding [Електронний ресурс] – Режим доступу : www/ URL: https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2. – 04.11.2024 р. – Загол. з екрану.