Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Системотехніки
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Методи та засоби автоматичної генерації лабіринтів
і дизайну рівнів в ігрових додатках
(тема)

Виконав:
студент 2 курсу, групи СПРм-18-2
Сухоцька Г. В.
(прізвище, ініціали)

Спеціальність 122 – Комп’ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма Системне проектування
(повна назва освітньої програми)

Керівник к.т.н проф. Чайніков С.І.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ____________ ____________________
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет ______________________ Комп'ютерних наук ____________________
(повна назва)

Кафедра ______________________ Системотехніки ______________________
(повна назва)

Рівень вищої освіти ______________ другий (магістерський) _______________

Спеціальність _______________ 122 – Комп'ютерні науки _________________
(код і повна назва)

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

Освітня програма _____________ Системне проектування __________________
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«____»_____________ 2020 р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ

студентові __Сухоцькій Ганні Володимирівні___________________________
(прізвище, ім'я, по батькові)

1. Тема роботи _Методи та засоби автоматичної генерації лабіринтів і дизайну рівнів в ігрових додатках ____

затверджена наказом університету від _30_ __березня_2020 р. № 477СТ

2. Термін подання студентом роботи до екзаменаційної комісії 20 __травня_ 2020 р.

3. Вихідні дані до роботи методи автоматичної генерації лабіринтів; сучасні засоби; методи проходження лабіринтів ______________________________________

4. Перелік питань, що потрібно опрацювати в роботі Огляд і аналіз сучасного стану розглянутої проблеми; Розробка методу автоматичної генерації лабіринтів; Розробка методу проходження лабіринтів; Практична апробація отриманих наукових результатів_________

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) 1) Графік Співвідношення часу виконання алгоритму до розмірів лабіринту (1 аркуш А4), 2) Модифікація алгоритму Вілсона (1 аркуш А4), 3) Модифікація мурашиного алгоритму (1 аркуш А4), 4) Приклад реалізації лабіринту в середовищі Unity 3D (1 аркуш А4), 5) Таблиця 1 - Результати роботи алгоритму видалення по довжині шляху (1 аркуш А4), 6) Таблиця 2 - Результати роботи алгоритму видалення схожих шляхів (1 аркуш А4)

6. Консультанти розділів роботи (п.6 включається до завдання 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	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на дипломне проектування	30.03.2020	
2	Аналіз завдання, літератури та аналогів з теми дипломної роботи	01.04.2020	
3	Аналіз засобів оцінювання програмного забезпечення	04.04.2020	
4	Постановка задачі	08.04.2020	
	Детальний аналіз вхідної інформації	09.04.2020	
	Розробка алгоритмів реалізації вирішення задачі	15.04.2020	
	Розробка програмного засобу	19.04.2020	
	Проведення обчислювальних експериментів	01.05.2020	
	Оформлення пояснювальної записки	05.05.2020	
	Оформлення графічної частини та презентаційних матеріалів захисту	15.05.2020	
	Представлення на рецензування	18.05.2020	
	Представлення дипломного проекту в ДЕК	20.05.2020	

Дата видачі завдання _30 березня 2020 р.

Студент ____
(підпис)

Керівник роботи ______________________________ ________________________________
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Реферат атестаційної роботи магістра: 95 с., 42 рис., 9 табл., 19 формул, 33 джерел.

АВТОМАТИЧНА ГЕНЕРАЦІЯ, АЛГОРИТМИ, ЛАБІРИНТИ, ІГРОВІ ДОДАТКИ, UNITY 3D, ГРАФІЧНИЙ ДИЗАЙН, ГЕЙМПЛЕЙ, ПРОХОДЖЕННЯ ЛАБІРИНТУ, МОДИФІКАЦІЯ

Об'єкт дослідження – процеси автоматичної генерації лабіринтів і дизайну рівнів в ігрових додатках.

Предмет дослідження – методи та засоби автоматичної генерації лабіринтів і дизайну рівнів в ігрових додатках.

Мета роботи – дослідження та розробка методів та засобів автоматичної генерації лабіринтів і дизайну рівнів в ігрових додатках.

Методи дослідження – аналіз існуючих методів вирішення проблеми автоматизованого порівняння моделей вибору дизайну мобільних додатків, їх формальних описів, аналіз методів розробки інтерфейсів користувача, аналіз методів проектування дизайну інтерфейсів користувача мобільних додатків.

Визначено підхід до вирішення проблеми автоматичної генерації лабіринтів і дизайну рівнів в ігрових додатках.

Виконано порівняння існуючих алгоритмів генерації та проходження лабіринтів, проведено порівняльний аналіз їх показників, виділені найефективніші алгоритми.

Запропоновано модифікацію алгоритмів генерації для поставлених цілей та оптимізацію методів проходження лабіринтів, яка задовольняє поставленій задачі.

ABSTRACT

Abstract on master's thesis: 95 p., 42 pic., 9 tables, 19 formulas, 33 sources

AUTOMATIC GENERATION, ALGORITHMS, MAZE, GAME ACCESSORIES, UNITY 3D, GRAPHIC DESIGN, GAMEPLAY, LABYRINTH, MODIFICATION

The object of study – the processes of automatic generation of mazes and level design in game applications.

The subject of study - methods and tools for automatic generation of mazes and level design in game applications.

The purpose of the work is to study methods and develop tools for automatic maze generation and level design in game applications.

Research Methods – analysis of existing methods of solving the problem of automated comparison of models of choice of design of mobile applications, their formal descriptions, analysis of methods of development of user interfaces, analysis of methods of designing of design of user interfaces of mobile applications.

The approach to solving the problem of automatic maze generation and level design in game applications is defined.

The comparison of existing algorithms of generation and passing of labyrinths is made, the comparative analysis of their indicators is carried out, the most effective algorithms are allocated.

Modification of generation algorithms for the set purposes and optimization of methods for passing labyrinths which satisfies the set task are offered.

ЗМІСТ

Зміст…………………………………………………………………………… 6
Скорочення та умовні познаки……………………………………………… 8
Вступ………………………………………………...…………………….... 9
1 Огляд і аналіз сучасного стану розглянутої проблеми…………………… 10
1.1 Особливості процесу розробки ігрових додатків……………………… 10
1.2 Огляд та класифікація видів лабіринту…………………………… 19
1.3 Використання лабіринтів у реальному житті…………………………… 28
1.4Постановка задачі………………………………………………………..... 30
2 Розробка методу автоматичної генерації лабіринтів……………………… 31
2.1 Дослідження існуючих методів генерації лабіринтів 31
2.1.1 Алгоритм двійкового дерева……………………………………… 31
2.1.2 Алгоритм Еллера……………………………………………………... 32
2.1.3 Алгоритм Sidewinder……………………………………………… 34
2.1.4 Алгоритм Олдоса-Бродера…………………………………………... 36
2.1.5 Алгоритм Вілсона………………………………………………... 38
2.1.6 Алгоритм Прима……………………………………………………... 40
2.1.7 Алгоритм Краскала…………………………………………………... 42
2.2 Порівняння існуючих алгоритмів генерації лабіринтів 43
2.3 Розробка модифікованого методу генерації лабіринтів 48
3 Розробка методу проходження лабіринтів…………………………………... 53
3.1 Дослідження існуючих методів проходження лабіринтів……………... 53
3.1.1 Алгоритм А-стар……………………………………………………... 53
3.1.2 Алгоритм Лі…………………………………………………………... 55
3.1.3 Променевий алгоритм………………………………………………... 56
3.1.4 Алгоритм Дейкстри……………………………………………... 58
3.1.5 Пошук в глибину……………………………………………………... 60

3.1.6 Пошук в ширину…………………………………………………... 61
3.2 Порівняння існуючих методів проходження лабіринтів………………. 63
3.3 Розробка модифікованого методу пошуку шляхів в лабіринті………. 72
4 Практична апробація отриманих наукових результатів…………………...
4.1 Особливості реалізації у середовищі Unity 3D……………………….. 79
4.2 Програмна реалізація генерації лабіринту у середовищі Unity 3D….. 80
4.3 Експериментальна перевірка модифікованого методу пошуку шляхів в лабіринті…………………………………………………………………... 85
Висновки…………………………………………………………………….. 91
Перелік джерел посилання………………………………………………… 92
Додаток А. Відомість атестаційної роботи……………………………….. 96
Додаток Б Специфікація…………………………………………………… 98
Додаток В. Сертифікат учасника наукової конференції……………………… 100

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

A* – А-стар, інформований алгоритм пошуку;

ДРП – дискретне робоче поле;

МР – мобільний робот;

ОБ – алгоритм Олдоса-Бродера;

BFS – Breadth-first search, пошуку в ширину;

BSP – Binary Space Partitioning, алгоритм двійкового дерева;

CBT – Close Beta Testing, закрите бета-тестування;

DFS – Depth-first search, пошук в глибину;

IDDFS – Iterative deeping depth-first search, алгоритм послідовних наближень при пошуку в глибину;

HDFS – Heuristic Depth First Search, евристичний пошук в глибину;

OBT – Open Beta Testing, відкрите бета-тестування;

RW – Random Walk, випадковий пошук

ВСТУП

У сучасному світі все глибше проникають комп'ютерні технології, тому доводиться освоювати нові навички при роботі на персональному комп'ютері. Якщо ази комп'ютерної грамотності вже присутні в багажі знань, настає момент розвитку швидкості і оптимізації роботи з персональним комп'ютером.

Оскільки комп'ютер може вирішувати багато завдань швидше, ніж це робить людина, йому довіряють різного роду роботу, в тому числі створення і проходження лабіринтів.

В даний час значимість лабіринтів в розважальній сфері все більш зростає. Завдання їх проходження зустрічається у багатьох проектах різних жанрів.

Генерація лабіринтів і алгоритми пошуку шляху в лабіринті – класична задача в програмуванні. Створення випадкових лабіринтів може використовуватися при створенні локацій в іграх, а алгоритми пошуку шляху – при навігації по карті.

У міру збільшення потужностей обчислювальної техніки, розмірності і швидкість побудови лабіринтів також зростають.

Однак якщо лабіринти невеликий розмірності можна намалювати вручну, то складність генерації лабіринтів великої розмірності набагато зростає. Існують численні алгоритми генерації лабіринтів, зі своїми особливостями, обмеженнями, достоїнствами і недоліками. Також є алгоритми пошуку шляхів в лабіринті. Однак існують практичні завдання, які не вирішуються стандартними алгоритмами. Одна з таких завдань була розглянута в ході роботи, і для неї буде потрібно конструювання нових методів і підходів до вирішення такого завдання.

Магістерська атестаційна робота виконується відповідно до вимог методичних вказівок по організації і виконанню магістерської атестаційної роботи [1, 2].

1 ОГЛЯД І АНАЛІЗ СУЧАСНОГО СТАНУ РОЗГЛЯНУТОЇ ПРОБЛЕМИ

1.1 Особливості процесу розробки ігрових додатків

Розробка будь-якої гри ґрунтується на трьох базових елементах – ігровий концепції, технології та візуальному стилі. Першим елементом є ігрова концепція, яка визначає правила гри і механіку взаємодії гравця з віртуальним світом. Як швидко гравець бігає і як високо він стрибає? Які перешкоди і будь персонажів він зустріне на своєму шляху? Яким чином він буде з ними взаємодіяти? На всі ці запитання відповідає геймдизайнер (Game designer), саме він і займається створенням ігрової концепції.

Другий елемент – це технологія, яка представляє собою технічну базу для реалізації гри. Основний функціонал забезпечує ігровий движок (Game engine), а додаткові інструменти дозволяють дизайнерам і художникам створювати і редагувати вміст гри. Створенням технології, інструментарію, а так само реалізацією ігрового функціоналу і механіки зазвичай займаються програмісти.

Третій базовий елемент, що лежить в основі будь-якої гри, – це візуальний стиль, який визначає вигляд гри. Створенням візуальної частини займається широке коло художників, що працюють в самих різних областях комп'ютерної графіки. Наприклад, концепт-художники придумають унікальний вигляд ігрового світу. Віртуальні скульптори сліплять моделі персонажів, аніматори «пожвавлять» їх, надавши кожному руху свій неповторний характер. А художники по ігровому оточенню створять всі необхідні деталі для декорування рівнів і наповнять його спецефектами.

Всі ці три елементи в сукупності служать одній меті – створенню ігрового процесу, спрямованого на розвагу гравця.

Для позначення процесу взаємодії гравця з грою існує спеціальний термін «геймплей» (Gameplay).

Геймплей – це те, що відрізняє комп'ютерну гру від таких неінтерактивних видів розваг, як книги або кіно. Дивлячись кіно або читаючи книгу, людина є лише стороннім спостерігачем. У випадку з геймплеєм ми отримуємо можливість безпосередньо взаємодіяти з грою, вирішуючи поставлені геймдизайнером завдання за допомогою ігрових механік.

У будь-якій грі можна виділити два основних типи геймплея - геймплей, що становить ядро гри, і геймплей на рівнях. В англійській термінології ці два типи найчастіше називають «core gameplay» і «level gameplay».

Геймплей, що становить ядро гри (core gameplay) – це набір базових правил і механік, які доступні гравцеві за замовчуванням. Наприклад, цей тип геймплея визначає характеристики персонажів, їх здібності, а також те, як вони взаємодіють один з одним. Всі ці основні правила створюються Геймдизайнер і є відправною точкою для роботи дизайнера рівнів – геймдизайнера, який спеціалізується на проектуванні віртуального простору.

Хорошим прикладом рівня без геймплея є чисте поле або порожня кімната, де гравець може використовувати тільки дані йому за умовчанням здатності, тобто бігати, стрибати і стріляти. Отже, від дизайнера рівнів потрібно спроектувати таку локацію, яка забезпечить достатню кількість цікавих ігрових ситуацій для застосування кожної базової механіки. Таким чином, оперуючи простором, об'єктами і настройками віртуального світу, дизайнер створює геймплей на рівнях (level gameplay), що дозволяє подарувати новий унікальний досвід взаємодії з грою.

В іграх-стратегіях рівні називають картами (Map), де геймплей створюється за рахунок різних особливостей ланшафта, розташування гравців і цінних ресурсів. У гонках рівні представляють собою траси з великою різноманітністю маршрутів і перешкод на них. У пригодницьких іграх геймплей будується на тому, які перешкоди, пастки, головоломки і вороги зустрінуться гравцеві. Таким чином, в іграх різних жанрів рівні за своєю структурою і зовнішнім виглядом можуть кардинально відрізнятися один від одного і носити різні назви (карти, місії, зони, етапи, завдання), але разом з тим виконувати одну і ту ж функцію - доповнювати, розвивати і створювати варіативність базового геймплею.

Для того, щоб розробити гру слід пройти наступні етапи [3]:

1. Етап «Альфа». Продюсер проекту отримує завдання від керівництва (іноді - бере участь в її формуванні), розробляє концепцію нового проекту і збирає початкову (альфа-) команду для його реалізації;

2. Етап «Пре-продакшен». Альфа-команда виявляє потенційні ризики для проекту і або усуває їх, або демонструє здатність усунути їх в майбутньому. Продюсер становить план виробництва проекту, формує повну команду розробки. Пишеться проектна документація. Розробляється прототип проекту.

3. Етап «Софт-ланч». Прототип проекту розширюється командою розробки до повноцінної версії гри відповідно до складеного плану і проектною документацією. Етап закінчується випуском ранньої версії гри для обраного ринку. Зазвичай така версія містить 30-50% від загальної кількості ігрового контенту і відрізняється від попередніх технічних версій великою стабільністю роботи і малою кількістю помилок.

4. Етап «Хард-ланч». Збирається статистика по ранній версії гри, з урахуванням цієї статистики гра приводиться до пізньої версії, що випускається на всі вибрані ринки. Вибирається маркетингова стратегія, формується план подальшого поліпшення гри.

5. Етап «Підтримка». Формується команда підтримки (як правило, це частина команди розробки) для випуску послідовних оновлень з урахуванням статистики. Продюсер продовжує роботу на проекті в якості консультанта, перекладаючи основні завдання на гейм-дизайнера команди підтримки.

Графічне уявлення цих етапів наведено на рисунку 1.1.

Рисунок 1.1 – Графічне уявлення етапів розробки гри

Етап «Альфа». Початок етапу. Продюсер отримує від керівництва первинну задачу – тобто набір умов для старту проекту і список пропонованих до проекту вимог. В якості умов зазвичай виступають: бажаний масштаб проекту, необхідні терміни розробки, умови фінансування, заздалегідь обрані технології та властивості команди. Вимогами є показники проекту в питаннях утримання, залучення та монетизації трафіку.

Мета етапу. Продюсеру необхідно отримати і затвердити у керівництва концепцію проекту і складу альфа-команди.

Протягом етапу. Продюсер формує концепцію проекту – тобто будь-який опис, що дає інформацію про те, як в заданих умовах проект задовольнить пред'явленим до нього вимогам. Концепція зазвичай включає опис цільової аудиторії, сеттинга і жанру гри, ключових особливостей геймплея, методик розробки, інструментарію і технологічних рішень, ринків збуту, підтримуваних платформ, термінів і вартостей реалізації, вимог до команди розробки. Одним з ключових факторів успіху на даному етапі є аналіз поточної ситуації на ринку - відстеження існуючих трендів, огляд найбільш успішних проектів, вибір нових інструментів розробки для освоєння і тощо

Закінчення етапу. Як тільки концепція затверджена, вибирається альфа-команда – тобто набір ключових співробітників, які складуть основу майбутньої команди розробки. Продюсер і альфа-команда переходять до наступного етапу.

Тривалість етапу. Як правило, опрацювання першого варіанту концепції укладається в один або два тижні. У разі невідповідності концепції вимогам керівництва можливо ітераційне поліпшення концепції. При визначенні терміну цього етапу слід виходити з масштабності і складності пропонованого проекту.

Етап «Пре-продакшен». Початок етапу. Продюсер і альфа-команда виявляють найбільш суттєві потенційні ризики для проекту. Ризики можуть породжуватися грою, менеджментом або командою.

Мета етапу: мінімізація знайдених ризиків шляхом проектування прототипу гри, розробки і деталізації проектної документації, складання плану розробки на наступні етапи.

Протягом етапу. Ризики, що пред'являються до гри, мінімізуються за допомогою створення прототипу (початкової версії гри, що містить приклади функціоналу, графіки або інших ризикових аспектів). Ризики менеджменту мінімізуються шляхом складання продюсерського плану і вимог до команди розробки. Ризики команди мінімізуються вибором вже освоєних технологій, а також команд, що мають досвід випуску схожих проектів, або використання цих технологій.

Закінчення етапу. Коли прототип продемонстрований керівництву і схвалений як вирішальний завдання щодо мінімізації ризиків, коли затверджений продюсерський план, формується команда розробки і запускається наступний етап.

Тривалість етапу визначається в залежності від кількості виявлених ризиків. Як правильно пре-продакшн займає не більше однієї шостої від загального терміну розробки проекту.

Етап «Софт-ланч». Початок етапу. Продюсер і команда розробки вирішують питання, що залишилися по методикам розробки, організації робочого процесу, інструментарію тощо. Потім починається процес розробки за затвердженим продюсерському плану.

Мета етапу полягає в розробці першої повноцінної версії гри, яка буде випущена для невеликої початкової аудиторії, щоб зібрати статистику і побачити потенціал проекту.

Протягом етапу. Команда працює над отриманням так званої soft-launch-версії – першої версії гри, яка стане доступна обраному безлічі гравців.

Закінчення етапу. Коли статистична картина проекту після запуску вже видно, керівництво приймає рішення про далі доля проекту. У гіршому випадку це закриття проекту як не має потенціалу, а в кращому – перехід до наступного етапу.

Тривалість етапу: в залежності від критеріїв, які висуваються до open beta, етап може займати від 40 до 80 відсотків часу, що залишився з моменту закінчення пре-продакшна до релізу.

Етап «Хард-ланч». Початок етапу. Коли статистична картина soft-launch-версії задовольняє поставленим критеріям, команда розробки починає готувати більш повну версію гри для запуску на всіх можливих ринки. Список доповнень і змін у проекті складається як із запланованих заздалегідь робіт, так і з виявлених за результатами збору статистики.

Мета етапу: підготувати версію гри для світового релізу, тобто випуску на всі заплановані ринки і аудиторії.

Протягом етапу. Всі зміни, як правило, вносяться в гру поступово, поки вона продовжує бути доступною для скачування аудиторії софт-ланчу. Кожне таке зміна змінює статистику проекту, що може відбитися на списку завдань.

Закінчення етапу: Коли статистична картина поточної версії гри задовільна, відбувається т.зв. хард-ланч – тобто запуск повноцінної версії гри для всіх обраних ринків. Статистична картина проекту після цього визначає долю проекту. Як правило, етап «хард-ланч» проходить кілька однакових циклів, на кожному з яких поточна статистика визначає фронт робіт для команди розробки і впливає на вибір маркетингової стратегії.

Тривалість етапу: все, що залишився до релізу. У разі раннього софт-ланчу це – 60 відсотків від часу між закінченням пре-продакшна і світовим релізом, а в разі пізнього – 20 відсотків.

Етап «Підтримка». Початок етапу. Якщо статистика проекту стабілізується на задовільною позначки, і проект перестає вимагати істотних доробок і

розширень, він віддається «на підтримку». Перед цим вибирається команда підтримки, яка часто є частиною команди розробки.

Мета етапу: мінімальними засобами підтримувати прийнятний рівень доходу з проекту аж до моменту, коли такий рівень стає об'єктивно недосяжним.

Протягом етапу. Команда підтримки ітеративно покращує гру за допомогою невеликих доповнень, такі ітерації з поліпшеннями можуть тривати до тих пір, поки підтримка гри виправдана з точки зору керівництва.

Закінчення етапу. Якщо подальше обслуговування проекту стає з яких-небудь причин невиправданим, проект вважається закритим. При цьому він може залишатися доступним для кінцевого користувача, якщо це доцільно і не вимагає участі команди підтримки.

Тривалість етапу багато в чому визначається властивостями проекту. Прийнятний термін доцільною підтримки проекту дорівнює року, тоді як два-три роки можна оцінити як дуже хороший результат.

Технічний погляд на розробку передбачає наступні етапи [4]:

Концептування (Concept) – на цьому кроці команда придумує концепцію гри, і проводить початкове опрацювання ігрового дизайну. Головна мета цього етапу – це геймдизайнерска документація, що включає Vision (розгорнутий документ, що описує гру, як кінцевий бізнес-продукт) і Concept Document (початкове опрацювання усіх аспектів гри). У документації геймдизайнер формулює і зберігає свої ідеї. Виконавцеві документація дозволяє правильно розуміти свої завдання по реалізації продукту. Тестувальник чітко бачить, що і як тестувати. Для Продюсера/ПМ ця документація надає матеріал для формування планів і контролю виконання завдань. Інвестор же (особливо на ранніх етапах) отримує розуміння, на що саме він виділяє кошти. Принципово важливо, щоб уся проектна і продуктова документація підтримувалася в актуальному стані на усіх етапах розвитку проекту. Для її ефективного використання і оновлення правильно використати спеціальні інструменти. Серед ключових принципів формування документації слід зазначити: структурованість, захищеність від різночитань, повний опис продукту, регулярну актуалізацію.

Прототипування (Prototyping) – важливий етап проектування будь-якої гри – це створення прототипу. Те, що добре виглядає «на папері», зовсім не обов'язково буде цікаво в реальності. Прототип реалізується для оцінки основного ігрового процесу, перевірки різних гіпотез, проведення тестів ігрових механік, для перевірки ключових технічних моментів. Дуже важливо на етапі створення прототипу реалізовувати тільки те, що потрібно перевірити і в стислі терміни. Прототип повинен бути простим в реалізації, тому що після досягнення поставлених перед ним цілей, він повинен бути «викинуть». 9 Серйозна помилка початківців розробників – нести тимчасову інфраструктуру і «милиці» реалізації коду в основний проект.

Вертикальний зріз (Vertical Slice) – отримання мінімально можливу повноцінну версію гри, що включає в себе повністю реалізований основний ігровий процес. При цьому висока якість опрацювання обов'язково потрібно втілити тільки для тих ігрових елементів, які суттєво впливають на сприйняття продукту. При цьому всі базові функції гри присутні як мінімум в чорновому якості. Реалізовано мінімальний, але достатній для втілення повноцінного ігрового процесу набір контенту (один рівень або одна локація).

Виробництво контенту (Content production) – на цьому етапі виробляється достатня кількість контенту для першого запуску на зовнішню аудиторію. Реалізуються всі фічі, заплановані до закритого бета-тестування. Це найбільш тривалий етап, який може займати, для великих клієнтських проектів рік і більше. На цьому етапі задіються найбільша кількість фахівців, які займаються виробництвом всього основного наповнення гри. Художники створюють всі графічні ресурси, Геймдизайнер налаштовують баланс і заповнюють конфіги, програмісти реалізують і полірують всі функції.

Friends & Family / CBT (закрите бета-тестування) – на етапі CBT продукт уперше демонструється досить широкій публіці, хоча і лояльній продукту або компанії. Серед найбільш важливих завдань на цьому етапі виступають: пошук і виправлення гейм-дизайнерских помилок, проблем ігрової логіки і усунення критичних багів (дефектів). На цьому етапі в грі є присутніми вже все ключові фічи

(функції), створений досить контенту для повноцінної гри тривалий час, налагоджені збір і аналіз статистики. Тестування йде по тест-плану, проводяться стрес-тести вже із залученням реальних гравців.

Soft Launch / OBT (відкритий бета-тест) – на цьому етапі триває тестування гри, але вже на широкій аудиторії. Йде оптимізація під великі навантаження. Гра має бути готова для прийому великого трафіку. У грі реалізований білинг і приймаються платежі. На цьому етапі повністю завершується розробка нових фічей. Відбувається feature freeze, програмісти перестають реалізовувати щось нове, а повністю перемикаються на відладку і тюнинг наявних фічей. Геймдизайнери, продюсер і аналітики роблять висновки із зібраної на CBT статистики і перевіряють ефективність монетизації. При 10 цьому, до початку етапу повинна повністю функціонувати інфраструктура проекту: сайт, групи соц. мережах, канали залучення (User Acquisition), підтримка користувачів.

Release – публікація гри. Мета – це отримання прибутку. Базовий вживаний для оцінки прибутковості критерій: кількість грошей, принесених в середньому одним гравцем за весь час (LTV aka lifetime value), повинна перевершувати витрати на залучення цього гравця (CPI aka cost per install). На цьому етапі має бути повністю відлагоджене оперування продукту (технічна підтримка, робота з ком'юніті), дотримуються маркетингові і фінансові плани, ведуться роботи по поліпшенню фінансових показників, активно відпрацьовуються канали по залученню трафіку. Команда розробки на цьому етапі займається виправленням технічних багів, що виявляються в процесі експлуатації і оптимізацією продукту. Геймдизайнери займаються тонким налаштуванням геймплея під реальну ситуацію у ігровому світі (особливо актуально для MMO проектів). Також реалізує різні внутрішньоігрові фічі, що підтримують нові монетизаційні схеми. І звичайно йде розробка і інтеграція в продукт нового контенту, що підтримує інтерес гравців.

Одним з ключових етапів створення гри є генерація її рівнів, в більшості ігор вони представлені ігровими картами, які вдають із себе лабіринти. Так як лабіринти дуже широко використовуються для створення локацій в багатьох іграх, гравець в будь-якому випадку буде стикатися з ними в процесі гри.

У зв'язку з цим, виникає питання про створення різних рівнів з різним розташуванням елементів в лабіринті. Для вирішення цієї проблеми дизайнери можуть створювати кожну нову локацію вручну, але в зв'язку з цим виникає ряд інших проблем. По-перше, це дуже трудомісткий процес, який вимагає часу і хорошої фантазії. По-друге, при створенні вручну великої кількості рівнів, неможливо придумати безліч різних розташувань. Все, так чи інакше, будуть схожі один на одного, і це не внесе ніякого різноманіття. Отже, побудова лабіринтів вручну – неефективне рішення проблеми. А тому перед розробниками ігор постає питання про процедурну генерації лабіринтів, тобто щоб кожне чергове проходження гри проходило на заново згенерованій території.

Для підвищення інтересу у користувачів, а також інтерактивності в іграх, які побудовані на лабіринтах, зазвичай використовують додаткові предмети, з якими може взаємодіяти гравець. Для ефективності та простоти розробки геймплея має сенс поєднати генерацію таких предметів з генерацією самого лабіринту.

Наступна проблема, з якою стикається розробник при створенні лабіринтів – це розміщення виходу з лабіринту. Це завдання теж було б доцільно автоматизувати, для того щоб вихід з'являвся щораз у випадковому місці, і відрізнявся від рівня до рівня. Оскільки створюємо весь лабіринт автоматично, вихід теж потрібно генерувати автоматично при створенні лабіринту. Для того, щоб користувачеві було найцікавіше, вихід з лабіринту повинен знаходитися на максимальному видаленні від входу.

Для вирішення цих проблем перш за все необхідно визначати, що таке лабіринт та яких видив він буває.

1.2 Огляд та класифікація видів лабіринту

Лабіринт – будь-яка структура (зазвичай в двомірному або тривимірному просторі), що складається із заплутаних шляхів до виходу (і / або шляхів, що ведуть

в тупик). 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 планом, з метою заплутати і не дати виходу недосвідченому в плані лабіринту людині [5,24].

Лабіринти в цілому можуть бути класифіковані за сімома різними параметрами [6]. Це: мірність, топологія, мозаїка, маршрутизація, текстура і фокус. Розглянемо деякі види лабіринтів різних класів [7]. Мірність: клас відповідає за кількість вимірів в просторі, які лабіринт покриває. Типи:

– 2D (двовимірний) – більшість лабіринтів, на папері або в реальному житті, є двовимірними, тобто завжди можна відобразити їх схему на аркуші паперу і переміщатися по ній, не перекриваючи ніяких інших проходів в лабіринті (рисунок 1.2).

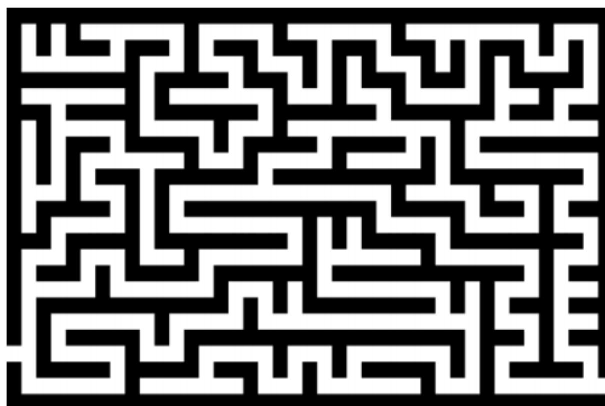

Рисунок 1.2 – 2D лабіринт

– 3D (тривимірний лабіринт) – це лабіринт, в якому декілька рівнів, де проходи можуть йти вгору і вниз, в доповненні до чотирьох сторін світу. 3d лабіринтів зазвичай відображаються на екрані як масив 2d рівнів, зі сходами вгору і вниз (рисунок 1.3).

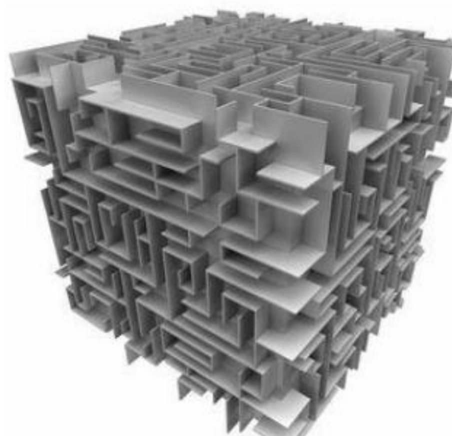

Рисунок 1.3 – 3D лабіринт

– Higher-dimensions (багатовимірні): можливі 4d і лабіринти з великою кількістю вимірів. Вони представляються як 3d лабіринтів із спеціальними " порталами", через які можна переміститися в 4-й вимір (рисунок 1.4).

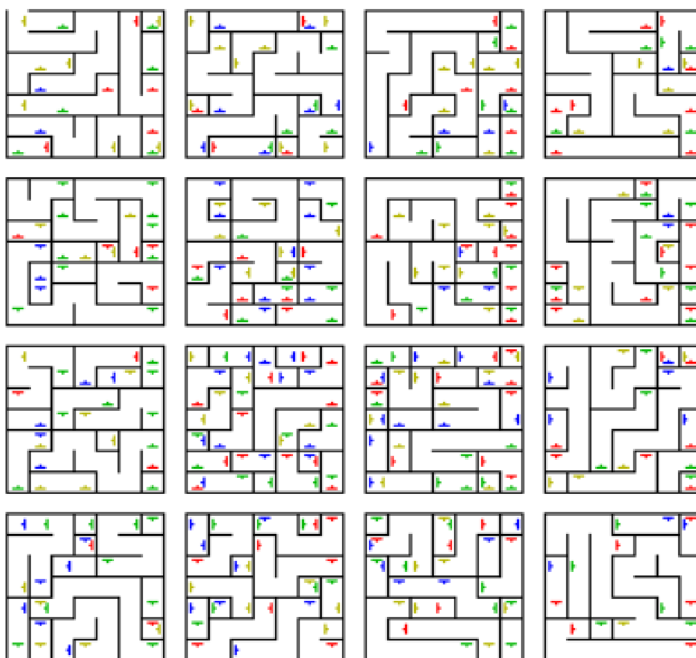

Рисунок 1.4 – 4D лабіринт

– Weave (переплетений) : переплетеними називаються в основному 2d (чи точніше 2.5d) лабіринтів, де проходи можуть перекривати один одного. Реальні

лабіринти, в яких є мости, що сполучають одну частину лабіринту з іншою, – частково переплетені (рисунок 1.5).

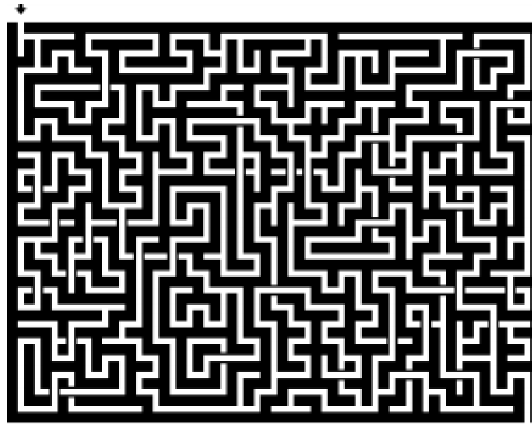

Рисунок 1.5 – Weave лабіринт

Топологія: клас топології описує геометрію простору, в якому в цілому існує лабіринт:

– Planair – термін відноситься до лабіринту з неправильною топологією. Це означає, що краї лабіринту сполучені незвичайними способами. Приклад, лабіринт на поверхні куба (рисунок 1.6).

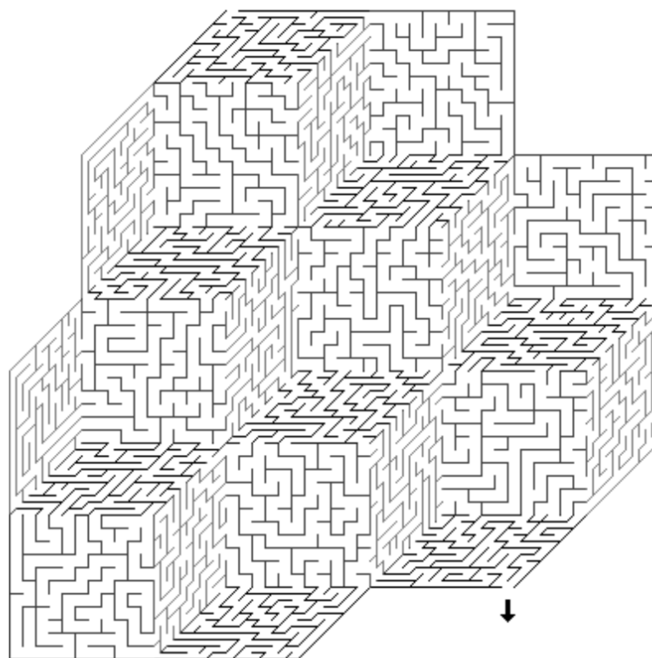

Рисунок 1.6 – Planair лабіринт

Мозаїка: клас складання мозаїки відповідає за геометрію окремих клітин, які складають лабіринт :

– Ортогональний – стандартна прямокутна сітка, де у клітин є проходи, що перетинаються під прямим кутом. У контексті складань мозаїки це можна також викликати Гамма-лабиринтом.

– Delta. Лабіринт Delta складається зі взаємозв'язаних трикутників, де у кожної клітини може бути до трьох проходів, сполучених з нею (рисунок 1.7).

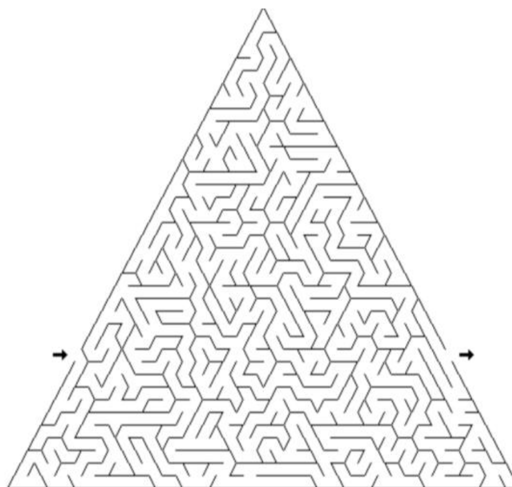

Рисунок 1.7 – Лабіринт Delta

– Sigma. Лабіринт Sigma складається зі взаємозв'язаних шестикутників, де у кожної клітини може бути до шести проходів, сполучених з нею (рисунок 1.8).

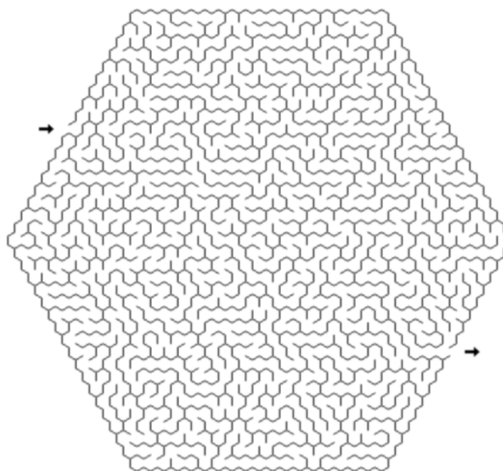

Рисунок 1.8 – Лабіринт Sigma

– Theta. Лабіринти Theta складається з концентричних кругів, де початок або кінець знаходиться в центрі, а інший на зовнішньому краю. Клітини зазвичай мають по чотири можливих прохідних з’єднаннь, але можуть мати і більше, якщо в зовнішніх прохідних кільцях буде більше клітин (рисунок 1.9).

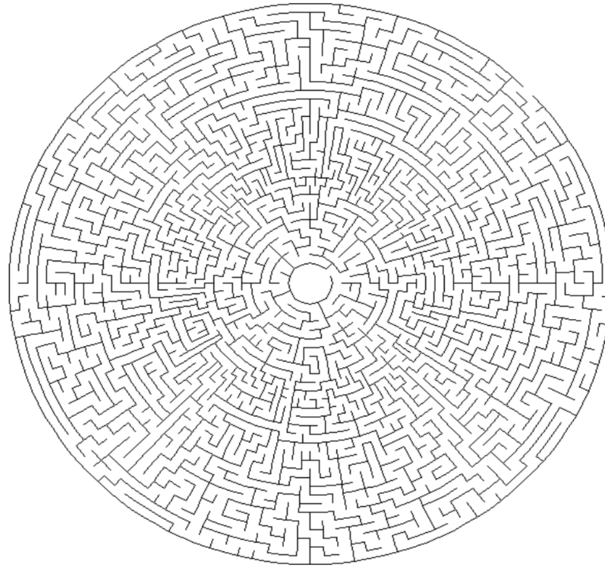

Рисунок 1.9 – Лабіринт Theta

– Upsilon. Лабіринти Upsilon складається зі взаємозв'язаних восьмикутників і квадратів, де у кожної клітини може бути до восьми або чотири можливі проходи, сполучених з нею (рисунок 1.10).

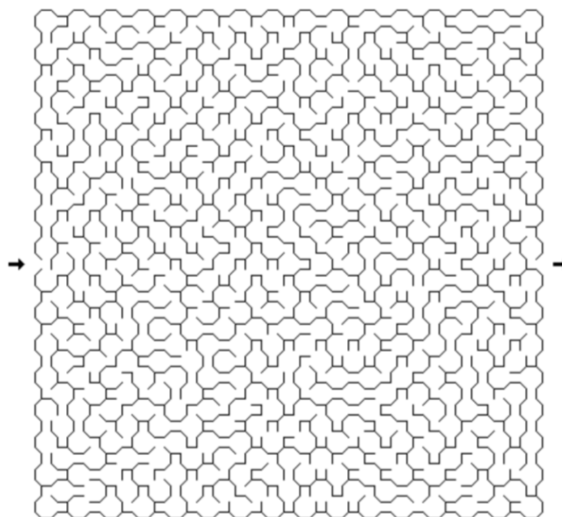

Рисунок 1.10 – Лабіринт Upsilon

– Zeta. Лабіринт Zeta знаходиться на прямокутній сітці, але в доповненні до прямих кутів, можуть бути проходи між клітинами по діагоналі на 45 градусів (рисунок 1.11).

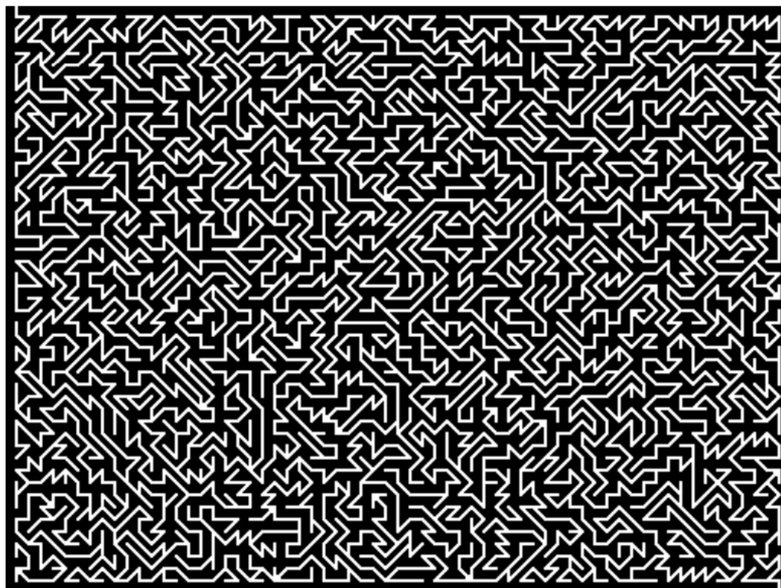

Рисунок 1.11 – Лабіринт Zeta

– Crack. Це безформний Лабіринт без постійної мозаїки, але що має стіни, або проходи у випадкових кутах (рисунок 1.12).

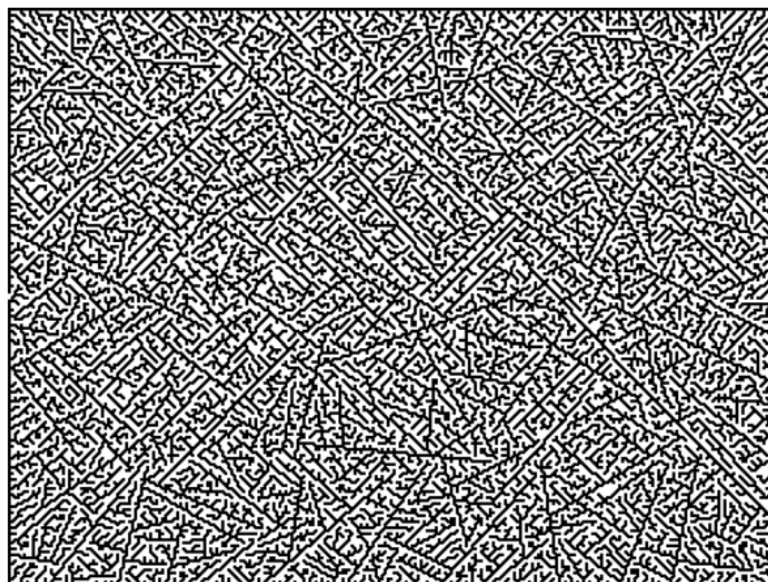

Рисунок 1.12 – Лабіринт Crack

– Fractal. Фрактальний Лабіринт – лабіринт, що складається з лабіринтів меншого розміру. Вкладена клітина фрактального лабіринту – це лабіринт з іншими лабіринтами, які складають мозаїчну структуру між кожною клітиною, в якій процес може повторюватися багаторазово. Нескінченний рекурсивний фрактальний лабіринт – це справжній фрактал, в якому лабіринт містить копії себе і в результаті є нескінченно великим лабіринтом (рисунок 1.13).

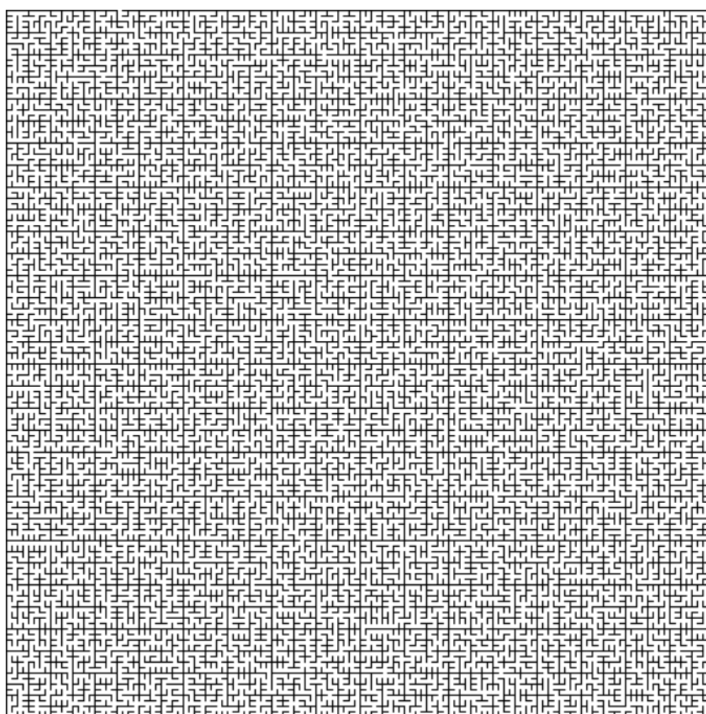

Рисунок 1.13 – Лабіринт Fractal

Маршрутизація: клас маршрутизації відноситься до самої генерації лабіринту. Лабіринти відрізняються по типах проходів :

– Досконалий – це лабіринт без витків, замкнутих циклів, або недоступних частин. Від кожної точки точно є хоч би один шлях до будь-якої іншої точки. У лабіринту точне є хоч би одно рішення. У комп'ютерних науках такий лабіринт може бути описаний як остовне дерево з набором клітин і вершин.

– Коса – це лабіринт без тупіків. У такому лабіринті проходи намотуються навкруги і заплітаються один в одного (від сюди термін " коса") і примушують вас витрачати час, ходячи по кругу замість того, щоб врізатися у тупік. Добре

розроблений лабіринт-коса може бути набагато важча, ніж досконалий лабіринт того ж розміру (рисунок 1.14).

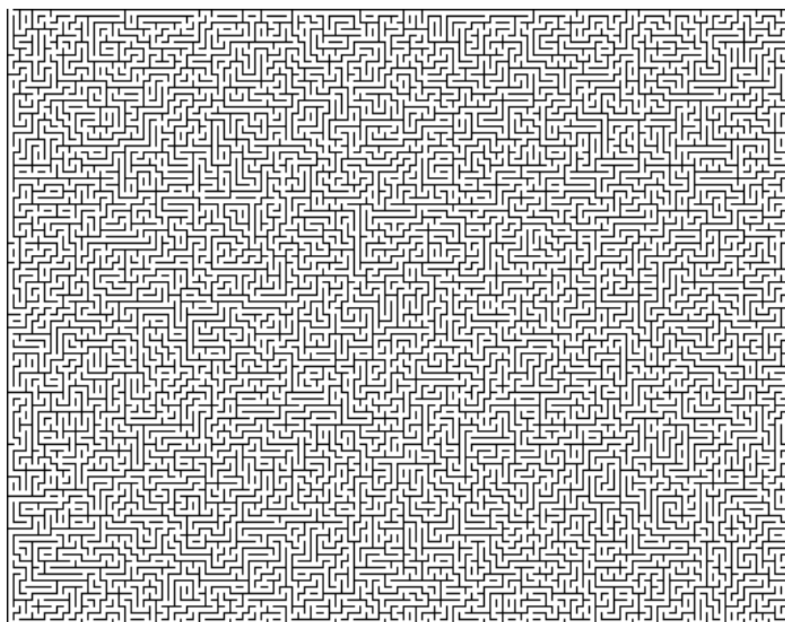

Рисунок 1.14 – Лабіринт-коса

– Унікурсальний – це лабіринт без перехресть. У унікурсального Лабіринту є всього один довгий змієподібний прохід. Пройти такий лабіринт не дуже важко, якщо тільки випадково не заблукати на половині шляху через, і не доведеться повертатися назад (рисунок 1.15).

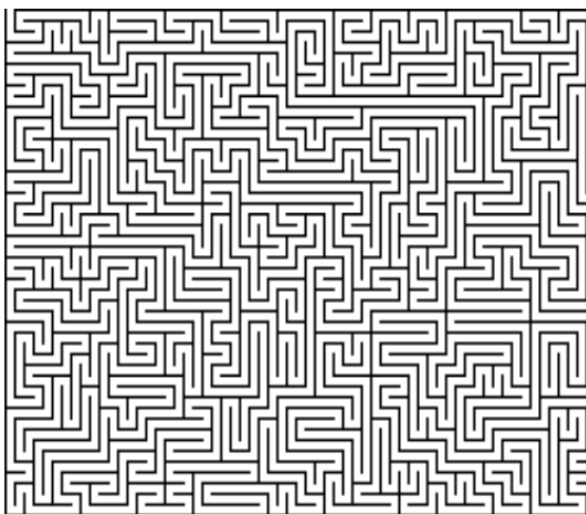

Рисунок 1.15 – Унікурсальний лабіринт

– Розріджений – в цьому лабіринті немає проходів через кожну клітину, тобто деякі проходи не створені (рисунок 1.16). Тут можуть бути присутніми недоступні області. Цей лабіринт можна назвати зворотним лабіринту-косі.

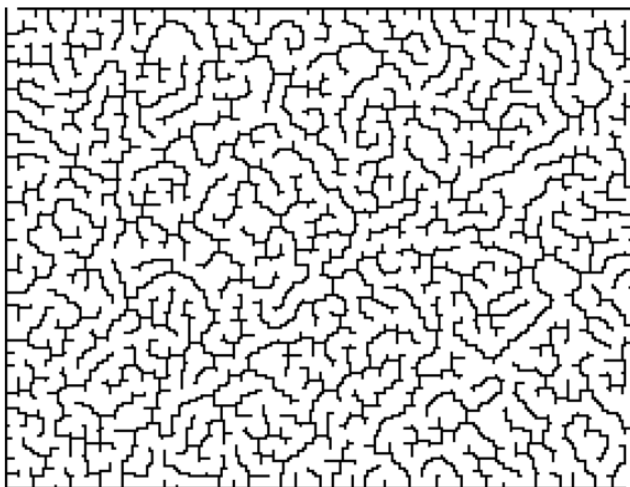

Рисунок 1.16 – Розріджений лабіринт

1.3 Використання лабіринтів у реальному житті

Крім основи для комп'ютерних ігор, лабіринти використовуються як штучне випробувальне середовище для експериментального дослідження алгоритмів керування рухом автономних мобільних роботів (МР), що широко використовуються в робототехніці.

Область застосування МР дуже широка. Це пошук та знешкодження небезпечних об'єктів, задача радіаційної та хімічної розвідки, робота в зоні техногенних і природних катастроф. Такі робототехнічні системи знаходять застосування і в цивільній сфері в якості сервісної робототехніки. Сервісні роботи вже з'явилися і успішно виконують функції обслуговування відвідувачів в музеях, аеропортах, магазинах. Особливо важливо застосування сервісних роботів в медичних установах, в тому числі, в якості засобу реабілітації пацієнтів. Активно

використовуються сервісні роботи телеприсутності, що дозволяють віддалено перебувати в приміщенні і переміщатися по ньому, спостерігаючи те, що відбувається навколо відеокамерою робота.

Розглядається робота мобільного робота в приміщенні, план якого заздалегідь невідомий. У приміщенні є як статичні перешкоди (стіни, столи, стільці), так і рухливі (люди, інші роботи).

Класична задача проходження лабіринту МР полягає в наступному. Є виконавець – автономний робот, він повинен пройти з вихідної (стартової) позиції до цільової (фінішної). Під рішенням лабіринту мається на увазі пошук такого маршруту. Якщо конфігурація лабіринту (тобто його план, карта) відома, то повинна вирішуватися завдання знаходження найкоротшого маршруту.

МР виконує функції орієнтації і руху. Він має набір сенсорів, що дозволяють визначати наявність стіни і контролювати відстань до неї, а також виявляти перешкоди.

Для виконання своїх завдань робот повинен рухатися по заданому маршруту і при цьому дотримуватися заходів безпеки, в тому числі, при наявності рухомих об'єктів в робочій зоні. Таким чином, робот переміщується автономно за допомогою навігаційної системи, при цьому оператор виконує тільки функцію постановки завдання. Не виключається і напівавтоматичний режим, наприклад, режим телеприсутності, при якому завдання оператора істотно спрощується.

Функціональна структура навігаційної системи мобільного робота [8] показана на рисунку 1.17.

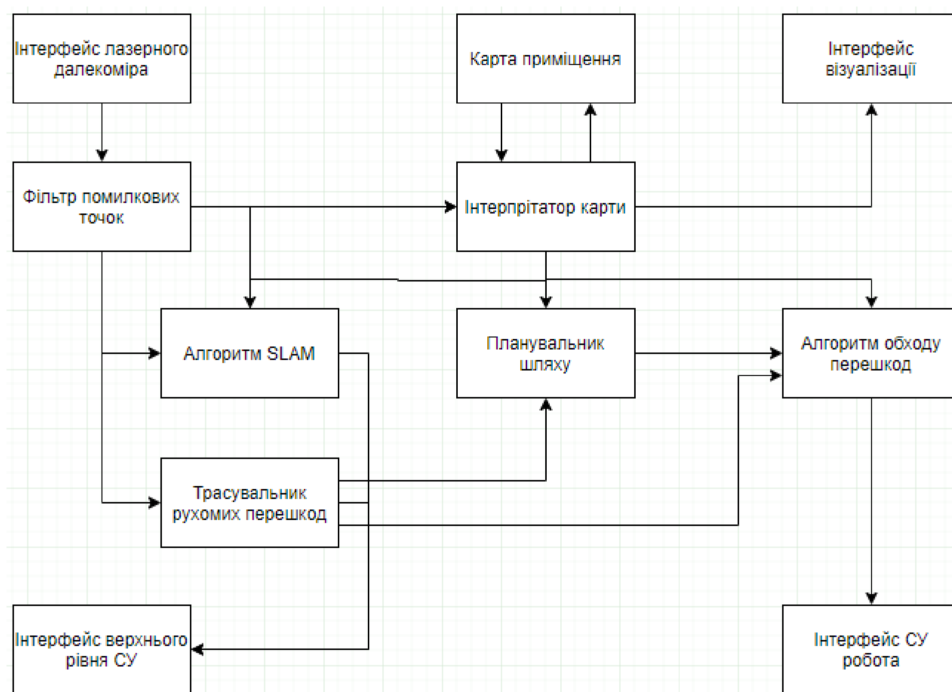

Рисунок 1.17 – Функціональна структура навігаційної системи мобільного робота

Найбільш важливим, для даної роботи є блок планування шляху, для чого необхідно дослідити не тільки алгоритми генерації лабіринтів але й їхнього проходження.

1.4 Постановка задачі

На підставі перерахованих проблем з розробкою ігрових систем, що включають генерацію рівнів, завданням даної дипломної роботи буде розробка методу автоматичної генерації лабіринтів та методу проходження лабіринтів, що є ефективним для розробки лабіринтів будь-якої складності та не потребує зайвих ресурсів.

2 РОЗРОБКА МЕТОДУ АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ ЛАБІРИНТІВ

2.1 Дослідження існуючих методів генерації лабіринтів

2.1.1 Алгоритм двійкового дерева

Алгоритм двійкового дерева (BSP –Binary Space Partitioning) – найпростіший алгоритм в розумінні, його суть полягає в тому, щоб прокласти шлях у випадковому напрямку з кожної клітини поля, що залежить від обраного зміщення. Після чого обробляється тільки одна клітина за одиницю часу, отже, з'являється можливість генерувати лабіринти нескінченного розміру, зберігаючи лише кінцевий результат (лабіринт) без необхідності зберігати будь-яку побічну інформацію [9-10].

Такий спосіб генерації має два побічних ефекти:

– лабіринти володіють сильним діагональним зміщенням і відсутністю тупиків в його напрямку;

– два порожніх коридору по сторонам лабіринту. Коли алгоритм «прокопується» до кінця рядка / стовпця, йому не залишається вибору, окрім як продовжити шлях в одному єдиному напрямку, створюючи порожні «кордони».

Результатом роботи алгоритму є випадкове двійкове дерево, в якому з кожної клітини (вершини) є рівно один шлях у напрямку до кореня (батьківської вершині), і, відповідно, рівно один шлях до будь-якої іншої клітці. Як наслідок, будь-яка клітина має не більше трьох з'єднань зі своїми сусідами.

Формальний алгоритм (для північно-східного зміщення) має наступний вигляд:

1. Вибрати початкову клітку.
2. Вибрати випадковий напрям для прокладання шляху. Якщо сусідня клітина в цьому напрямку виходить за межі поля, прокопати клітку в єдино можливому напрямку.
3. Перейти до наступної клітці.

4. Повторювати кроки 2-3 до тих пір, поки не будуть оброблені всі клітини.

Перевагами алгоритму є:

– проста реалізація;
– висока швидкість роботи;
– можливість генерувати нескінченні лабіринти.

Серед недоліків можливо відзначити:

– низька складність лабіринту;
– сильне зміщення по діагоналі;
– відсутність тупиків за зміщенням;
– одноманітність згенерованих лабіринтів.

Приклад роботи алгоритму двійкового дерева наведено на рисунку 2.1.

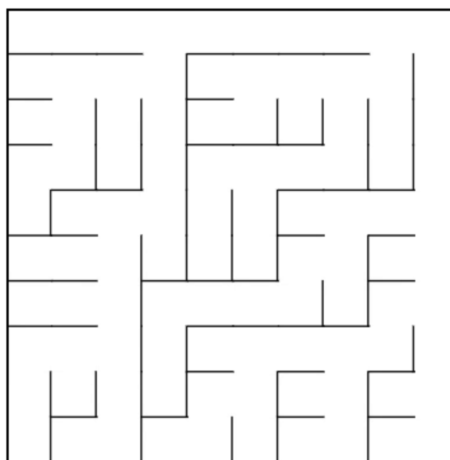

Рисунок 2.1 – Приклад роботи алгоритму BSP

2.1.2 Алгоритм Еллера

Алгоритм Еллера – це математичний генератор, що дозволяє створювати лабіринти, у яких між кожними двома точками існує єдиний шлях, тобто лабіринти не містять циклів. У порівнянні з іншими генераторами, даний алгоритм є одним з

найбільш швидких та потребує незначну кількість пам'яті – пропорційну довжині рядка лабіринта. Потрібно зберігати в пам'яті лише останній створений рядок – це дозволяє генерувати лабіринти з необмеженою кількістю рядків.

Алгоритм являє собою цикл додавання нових рядків. Рядок містить одну і ту саму кількість клітинок, яка довільно задається на початку. Клітинки належать до множин, що слугують для контролю можливості проходу між клітинками. На момент генерації поточного рядка клітини однієї множини з'єднані між собою, водночас клітини з різних множин знаходяться в ізольованих між собою частинах лабіринта. У кожної клітинки може бути або не бути права та нижня стінка. Загалом, стінки генеруються випадковим чином, але при дотриманні певних правил, які гарантують відсутність циклів у лабіринті [11].

Алгоритм Еллера включає наступні етапи:

1. Створити перший рядок. Жодна клітинка не належить жодній множині.

2. Присвоїти всім клітинкам, що не мають множини, свою унікальну множину.

3. Створити праві стінки, рухаючись зліва направо:

3.1. Випадково вирішити, чи додавати стінку:

3.1.1. Якщо поточна клітинка та клітинка справа належать одній множині, додати стінку(для попередження зациклень).

3.1.2. Якщо вирішено не додавати стінку, об'єднати множини поточної та наступної клітинки.

4. Створити стінки знизу, рухаючись зліва направо:

4.1. Випадково вирішити, чи додавати стінку знизу. При додаванні переконатися, що кожна множина має хоча б одну клітинку без нижньої стінки — це гарантуватиме відсутність ізольованих областей.

5. Вирішити, чи додавати рядки після поточного.

5.1. Якщо вирішено додати рядок, то:

5.1.1. Вивести поточний рядок.

5.1.2. Видалити всі праві стінки.

5.1.3. Видалити клітинки з нижніми стінками з їх множин.

5.1.4. Видалити всі нижні стінки.

5.1.5. Продовжити з кроку 5.1.2.

5.2. Якщо вирішено закінчити лабіринт, то:

5.2.1. Додати нижню стінку до кожної клітинки.

5.2.2. Рухаючись зліва направо:

5.2.2.1. Якшо поточна кліткинка та клітинка справа належать до різних множин, то:

5.2.2.1.1. Видалити праву стінку.

5.2.2.1.2. Об'єднати множини цих клітинок.

Приклад згенерованого алгоритму наведено на рисунку 2.2.

Рисунок 2.2 – Приклад згенерованого лабіринту

2.1.3 Алгоритм Sidewinder

Алгоритм Sidewinder дуже схожий на алгоритм двійкового дерева, але має більшу складний. Лабіринт генерується по одному рядку за раз: для кожного осередку випадковим чином вибирається, чи потрібно вирізати прохід, що веде

вправо. Якщо прохід не вирізаний, то вважається, що тільки що завершився горизонтальний прохід, утворений поточної осередком і всіма осередками зліва, що вирізали проходи, що ведуть в неї. Випадковим чином вибирається одна клітинка уздовж цього проходу і вирізається прохід, що веде з неї вгору [12].

У той час як лабіринт двійкового дерева завжди піднімається вгору від самої лівої комірки горизонтального проходу, лабіринт Sidewinder піднімається вгору від випадкової осередки. У двійковому дереві у лабіринту в верхньому і лівому краї є один довгий прохід, а в лабіринті Sidewinder є тільки один довгий прохід по верхньому краю.

Як і лабіринт двійкового дерева, лабіринт Sidewinder можна без помилок і детермінованою вирішити знизу вгору, тому що в кожному рядку завжди буде рівно один прохід, що веде вгору. Рішення лабіринту Sidewinder ніколи не робить петель і не відвідує один рядок більше одного разу, проте воно «звивається з боку в бік». Єдиний тип осередків, який не може існувати в лабіринті Sidewinder – це глухий кут з провідним вниз проходом, тому що це буде суперечити правилу, що кожен прохід, що веде вгору, повертає нас до початку.

Лабіринти Sidewinder схильні до появи елітного рішення, при якому правильний шлях виявляється дуже прямим, але поруч з ним є безліч довгих помилкових шляхів, що ведуть зверху вниз [13].

Формальний алгоритм (для стандартного зміщення) має наступний вигляд:

1. Вибрати початковий ряд.
2. Вибрати початкову клітку ряду і зробити її поточно.
3. Ініціалізувати порожню множину.
4. Додати поточну клітку до множини.
5. Вирішити, прокладати чи шлях направо.
6. Якщо проклали, то перейти до нової клітці і зробити її поточною. Повторити кроки 3-6.
7. Якщо не проклали, вибираємо випадкову клітку безлічі і прокладаємо звідти шлях наверх. Переходимо на наступний ряд і повторюємо 2-7.
8. Продовжувати, поки не буде оброблений кожен ряд.

Перевагами алгоритму є:

– можливість генерувати нескінченні лабіринти;

– тільки один порожній коридор;

– більш складний малюнок, на відміну від алгоритму двійкового дерева.

Серед недоліків можливо відзначити:

– більш заплутана реалізація;

– відсутність тупиків за зміщенням;

– значне вертикальне зміщення.

Приклад роботи алгоритму Sidewinder наведено на рисунку 2.3.

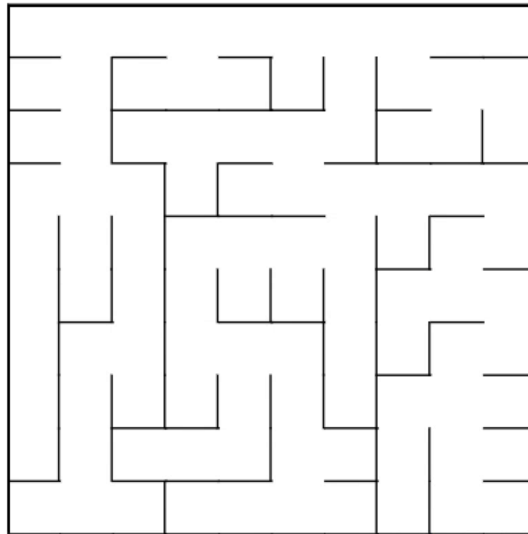

Рисунок 2.3 – Приклад роботи алгоритму Sidewinder

2.1.4 Алгоритм Олдоса-Бродера

Алгоритм Олдоса-Бродера (ОБ) – однорідний алгоритмі, тобто він з однаковою ймовірністю створює всі можливі лабіринти заданого розміру. Крім того, йому не потрібно додаткової пам'яті або стека. Цей алгоритм створює лабіринти з низьким показником плинності (це означає, що для заданого розміну

існує більше лабіринтів з низьким показником плинності, ніж з високим, тому що лабіринт із середньою однаковою ймовірністю має низький показник).

Погано в ОБ те, що він дуже повільний, тому що не виконує інтелектуального пошуку останніх осередків, тобто по суті не має гарантій завершення. Однак через свою простоту він може швидко проходити по безлічі осередків. В середньому його виконання займає в сім разів більше часу, ніж у стандартних алгоритмів, хоча в поганих випадках воно може бути набагато більше, якщо генератор випадкових чисел постійно уникає останніх кількох осередків.

Він може бути реалізований як алгоритм, що додає стіни, якщо стіну кордону вважати єдиної вершиною, тобто, наприклад, якщо хід переміщує до стіни кордону, то відбувається телепортація до випадкової точки вздовж кордону, а вже потім продовжується рух. У разі додавання стін він працює майже в два рази швидше, тому що телепортація уздовж стіни кордону дозволяє швидше отримувати доступ до далеких частин лабіринту [14].

Алгоритм ОБ є максимально простим і створює максимально заплутані лабіринти. Пояснюється це тим, що в алгоритм складається трьох дій:

1. Вибрати випадкову клітку на полі.
2. Перейти до будь-якої сусідньої клітинки.
3. Якщо дана клітинка не була раніше переглянуло, то зруйнувати між клітинами стінку і перейти до кроку 2, інакше просто перейти до кроку 2.

Таким чином створюється максимально непередбачуваний лабіринт.

Найбільшим мінусом даного алгоритму є час побудови лабіринту. Навіть не дуже великий лабіринт може будуватися досить довго. Також він не дозволяє генерувати нескінченні лабіринти та має сильне падіння ефективності під кінець генерації [15].

Найголовнішими перевагами є:

– відсутність будь-яких зміщень;
– лабіринти є абсолютно випадковими, тому неможливо створити певний алгоритм їх вирішення;

– складність рішення для людини;

– проста реалізація.

Приклад згенерованого алгоритму наведено на рисунку 2.4.

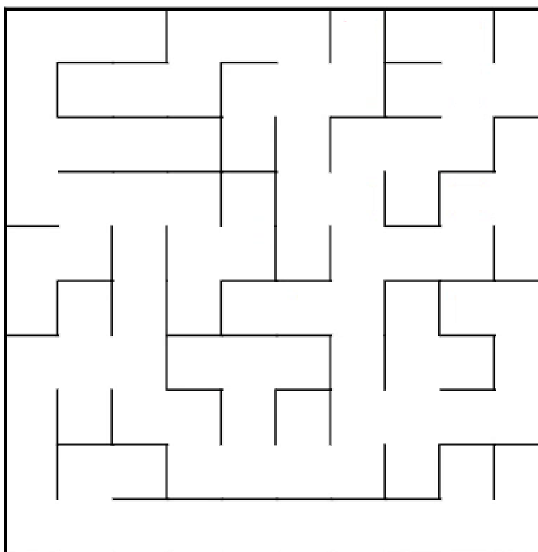

Рисунок 2.4 – Приклад згенерованого лабіринту

2.1.5 Алгоритм Вілсона

Алгоритм Вілсона – це вдосконалена версія алгоритму Олдоса-Бродера, він створює лабіринти точно з такою ж текстурою (алгоритми однорідні, тобто всі можливі лабіринти генеруються з однаковою ймовірністю), проте алгоритм Вілсона виконується набагато швидше.

Він займає пам'ять аж до розмірів лабіринту. Алгоритм має ті ж проблеми зі швидкістю, що і ОБ, тому що може піти багато часу на знаходження першого випадкового шляху до початкової осередку, однак після розміщення декількох шляхів інша частина лабіринту вирізається досить швидко. В середньому він виконується в п'ять разів швидше Олдоса-Бродера, і менш ніж в два рази повільніше кращих алгоритмів. Варто врахувати, що в разі додавання стін він

працює в два рази швидше, тому що вся стіна кордону спочатку є частиною лабіринту, тому перші стіни приєднуються набагато швидше.

Алгоритм Вілсона, як і ОБ, генерує абсолютно випадкові лабіринти без будь-якого зсуву. Алгоритм не має переваг за спрямованістю, заплутаності або ще якимось характеристикам. Для отримання найкращих результатів слід використовувати апаратні генератори випадкових чисел, які не мають переваг в числах [16].

Алгоритм Вілсона включає наступні етапи:

1. Вибрати випадкову вершину, що не належить кістяку і додати її в дерево.

2. Вибрати випадкову вершину, що не належить до кістяку і почати обхід графа (лабіринту), поки не прийдемо в уже додану вершину дерева.

2.1. Якщо утворюється цикл, видалити його.

3. Додати всі вершини отриманого підграфа в кістяк.

4. Повторювати кроки 2-3, поки все вершини не будуть додані до кістяку.

Перевагами алгоритму є:

– відсутність будь-яких зміщень;

– лабіринти є абсолютно випадковими, тому неможливо створити певний алгоритм їх вирішення;

– складність рішення для людини;

– немає безглуздого блукання;

– в порівнянні з ОБ в рази швидшіший.

Серед недоліків можливо відзначити:

– непроста реалізація;

– падіння швидкості на початку генерації;

– великі вимоги до пам'яті, ніж у Олдос-Бродера.

Приклад роботи алгоритму Вілсона наведено на рисунку 2.5.

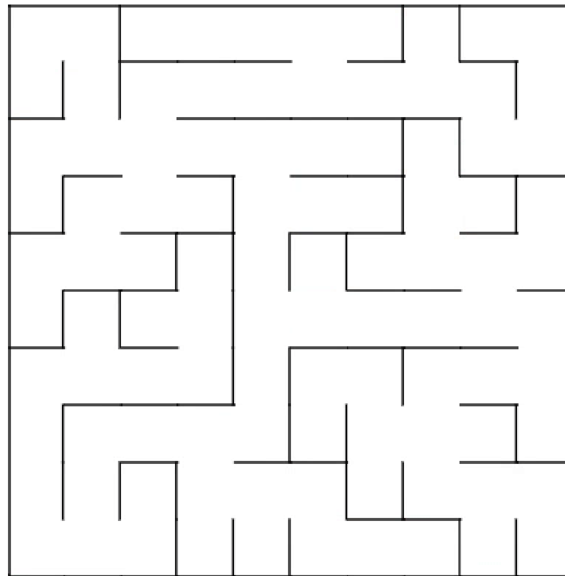

Рисунок 2.5 – Приклад роботи алгоритму Вілсона

2.1.6 Алгоритм Прима

Алгоритм Прима, який використовується для генерації лабіринтів – це рандомізована версія алгоритму Прима: методу створення мінімального остовного дерева з неорієнтованого зваженого графа. Алгоритм Прима створює дерево, збираючи сусідні осередки і знаходячи найкращій для переходу до наступного.

Для створення лабіринтів з використанням алгоритму Прима необхідно замість найкращого обрати випадковий осередок, щоб перейти до наступного. Алгоритм Прима зберігає список сусідніх осередків та вимагає зберігання, пропорційного розміру лабіринту.

Алгоритм Прима починається з будь-якої клітини і зростає з цієї клітини назовні. Алгоритм зберігає набір можливих осередків, на які може бути розширений лабіринт. На кожному кроці лабіринт розширюється у випадковому напрямку, якщо при цьому не возз'єднується з іншою частиною лабіринту.

Рандомізований алгоритм змінює з'єднання з осередком, так що замість того, щоб витягати осередок з найменшою вагою, осередок підключається довільно [17].

Алгоритм Прима включає наступні етапи:

1. Почати з сіткою повною стін.

2. Вибрати клітинку та відзначити її як частину лабіринту. Додати стінки комірки в список стіни.

3. У той час як є стіни в списку:

3.1. Вибрати випадкову стіну зі списку. Якщо тільки один з двох осередків, що стіна ділить відвідується, то:

3.1.1. Зробити стіни проходу і відзначити осередок, що не був відвіданий як частину лабіринту.

3.1.2. Додати сусідні стіни осередку в список стіни.

3.2. Видалити стіну зі списку.

Лабіринт створений за алгоритмом Прима наведено на рисунку 2.6.

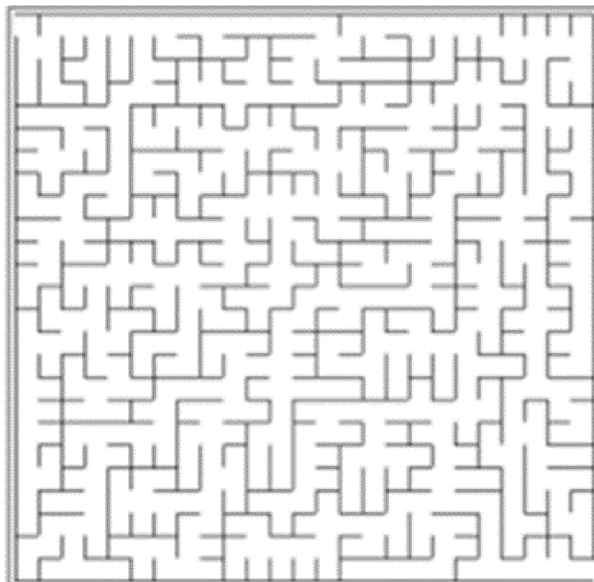

Рисунок 2.6 – Лабіринт створений за алгоритмом Прима

2.1.7 Алгоритм Краскала

Алгоритм Краскала – це рандомізована версія алгоритму Краскала: метод створення мінімального остовного дерева для зваженого графа.

Алгоритм Краскала цікавий тим, що він не «вирощує» лабіринт як дерево, а замість цього вирізає сегменти проходу по всьому лабіринту випадковим чином. Проте, в результаті виходить ідеальний лабіринт.

Рандомізований алгоритм змінює перший крок циклу так, що замість витягування ребра з найменшою вагою видаляється ребро з набору випадковим чином [18].

Аналог повинен вимагати зберігання, пропорційного розміру лабіринту, поряд з можливістю перераховувати кожне ребро між осередками у випадковому порядку (використовуючи набір ребер і вибираючи їх випадковим чином).

Як тільки всі ребра зібрані в великий набір і знайдений унікальний ідентифікатор піднабора, пов'язаний з кожною клітинкою; все, що потрібно, це вибрати ребро випадковим чином, перевірити, чи належать сусідні осередки іншої підмножині, і об'єднати їх, встановивши один і той же ідентифікатор для всіх осередків обох підмножин.

Формальний алгоритм має наступний вигляд:

1. Створити список всіх стін та набір для кожного осередку, кожна з яких містить лише одну клітину.

2. Для кожної стіни, в випадковому порядку:

2.1. Якщо клітини, розділені на цій стіні належать різним множинам:

1. Видалити поточну стіну.

2. Реєстрація множини раніше розділених клітин.

Приклад роботи алгоритму Краскала наведено на рисунку 2.7.

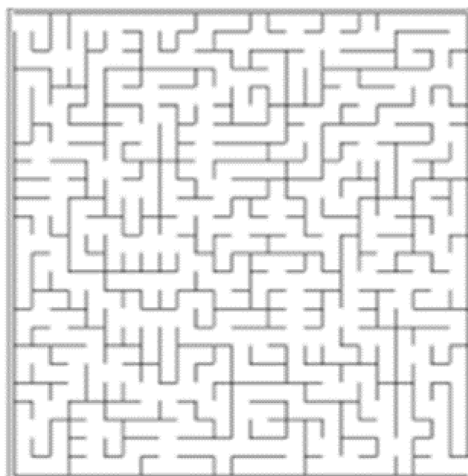

Рисунок 2.7 – Приклад роботи алгоритму Краскала

2.2 Порівняння існуючих алгоритмів генерації лабіринтів

Для порівняння існуючих алгоритмів генерації лабіринтів необхідно проаналізувати складність лабіринтів, побудованих вищезгаданими алгоритмами. Для цього були проаналізовані властивості лабіринтів і їх тимчасові характеристики. Також необхідною умовою аналізу є використання агентів пошуку.

Агенти пошуку допомагають зрозуміти, наскільки складно вирішити тий чи інший лабіринт. Агенти оцінюють різні атрибути складності лабіринтів, які будуть проаналізовані, серед них:

– кількість кроків необхідна для проходження лабіринту;
– кількість відвіданих осередків;
– кількість відвіданих перетинів;
– кількість відвіданих тупиків.

Були проаналізовані наступні агенти пошуку:

– агент випадкового пошуку (Random Walk – RW) – агент випадково вибирає шлях від вершини до випадкового сусіда, поки не доберется до кінця лабіринту;

– агент пошуку в глибину (Depth First Search – DFS) – цей агент не закінчує шлях, поки не зайде в глухий кут. Потім агент повертається до першого вузла з невідвідуванними сусідами. Він повторює пошук, поки не дійде до кінця лабіринту. Пріоритет поворотів агента на перетинах попередньо визначається вручну: схід, південь, захід, північ;

– агент евристичного пошуку в глибину (Heuristic Depth First Search – HDFS) – аналогічний агенту DFS, але вибирає кращі напрямки з використанням простої евристики. Зокрема, перевага віддається сусідам з більш низькою Манхеттенською відстанню до кінця лабіринту.

– агент пошуку в ширину (Breadth First Search – BFS) – цей агент використовує ідею BFS [19] для вирішення лабіринту, але замість того, щоб спочатку відвідати початкові вузли, агент заходить в кінець лабіринту. Цей агент нагадує людський вирішувач, який може вільно стрибати з одного шляху до іншого.

Було проведено експериментальний аналіз тимчасової складності алгоритмів, метою якого була оцінка якості їх результатів. Було проаналізовано ставлення часу виконання алгоритмів генерації лабіринту до розмірів лабіринту. Результат представлений на рисунку 2.8.

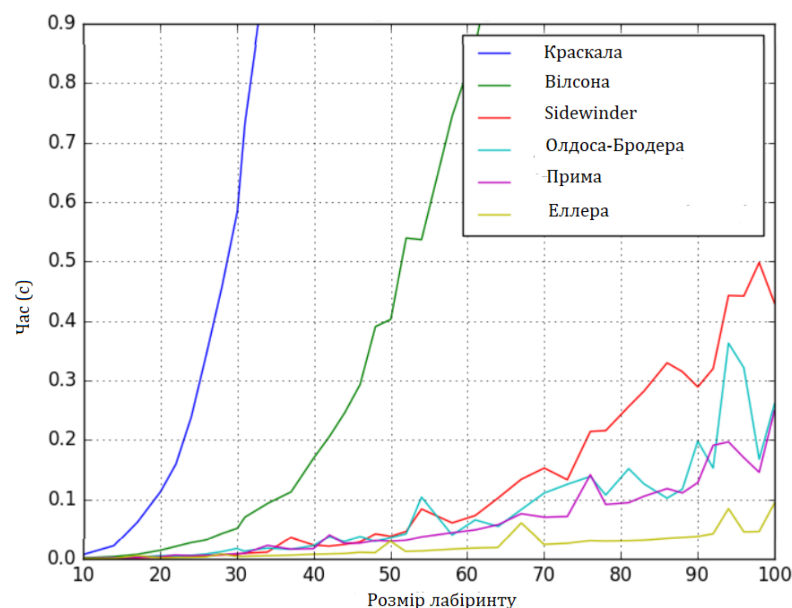

Рисунок 2.8 – Співвідношення часу виконання алгоритму до розмірів лабіринту

Щоб проаналізувати складність лабіринту, необхідно розглянути його наступні властивості:

– розмір s_i;
– кількість перетинів n_i;
– кількість тупиків n_{de}.

Були створені 1000 лабіринтів (розміром 100 × 100) кожного типу, за якими були розраховані кількість перетинів і тупиків. Середні результати наведені в таблиці 2.1.

Таблиця 2.1 – Середня кількість перетинів і тупиків в лабіринтах

Алгоритм	n_i	n_{de}	Ранг
Прима	2946	3559	1
Краскала	2654	3058	2
Олдоса-Бродера	2577	2933	3
Вілсона	2576	2932	4
Sidewinder	920	939	5
Еллера	869	898	6

Наведені вище дані показують, що певні лабіринти, створені за допомогою аналогічних алгоритмів, поводяться однаково. Зокрема, алгоритми Олдоса-Бродера і Вілсона, які походять з алгоритмів знаходження однорідного остовного дерева. мають практично однакову кількість перетинів і тупиків. Алгоритми Еллера і Sidewinder виділяються тим, що вони мають значно менші значення n_i і n_{de} ніж інші лабіринти, ці алгоритми походять з DFS. Інша пара алгоритмів – Краскала і Прима походять з алгоритмів пошуку на графі, але не мають таких чітких значень властивостей, як і дві інші групи.

Основним показником складності лабіринту є кількість кроків, які агент здійснює для його проходження. Крок визначається як перехід від поточного вузла до сусіднього (таблиця 2.2).

Таблиця 2.2 – Середня кількість кроків необхідна для проходження лабіринту

Алгоритм	RW	DFS	HDFS	BFS	Ранг
Sidewinder	7,3M	5,3k	13,3k	3,5k	1
Олдоса-Бродера	4,3M	7,1k	12,6k	3,1k	2
Вілсона	4,3M	7,2k	12,4k	3,0k	3
Еллера	9,1M	2,2k	12,7k	2,7k	4
Краскала	4,0M	6,8k	12,5k	2,8k	5
Прима	2,3M	5,6k	15,4k	1,7k	6

Ранжування алгоритмів генерації лабіринтів було проведено відповідно до характеристик рішення лабіринту агентами. Для цього був розроблений простий метод. Для кожного алгоритму i був розрахований показник s_i за формулою:

$$s_i = \sum_{a \in A} \frac{a_i}{\max(a)} \quad (2.1)$$

де a_i – значення балу агенту a для алгоритму i, $\max(a)$ – максимальне значення, яке агент набрав серед всіх генерацій алгоритмів.

Відповідно до цієї оцінки, алгоритм Sidewinder показує кращі результати. Цікаво те, що результати схожого з ним алгоритму Еллера значно гірші.

Далі необхідно проаналізувати, як часто агенти відвідують перетини. Чим більше перетинів відвідує агент, тим більше шансів пропустити правильний шлях. Отже, алгоритм генерації є більш складним (таблиця 2.3).

Таблиця 2.3 – Середня кількість відвіданих перетинів для кожного агента

Алгоритм	RW	DFS	HDFS	BFS	Ранг
Олдоса-Бродера	1.8M	2.9k	5.1k	850	1
Вілсона	1,7M	2,9k	5,0k	844	2
Краскала	1,7M	2,9k	5,2k	800	3
Прима	1,1M	2,7k	7,3k	567	4

Кінець таблиці 2.3

Алгоритм	RW	DFS	HDFS	BFS	Ранг
Sidewinder	1,0M	750	1.8k	337	5
Еллера	1.2M	232	2,0k	211	6

Була використана та ж техніка ранжирування, що і для кількості кроків. На відміну від них, тут алгоритм Sidewinder виступає погано. кращими виявилися алгоритми Олдоса-Бродера і Вілсона.

Остання властивість, яка аналізується, це число тупиків, які агенти відвідують в середньому (таблиця 2.4).

Таблиця 2.4 – Середня кількість відвіданих тупиків для кожного агента

Алгоритм	RW	DFS	HDFS	BFS	Ранг
Олдоса-Бродера	0,6M	1023	1863	823	1
Вілсона	0,6M	1035	1840	816	2
Краскала	0,6M	1030	1927	769	3
Прима	0,4M	974	2759	535	4
Еллера	0,4M	75	681	190	5
Sidewinder	0,4M	249	626	307	6

Ранг кількості відвіданих тупиків приблизно схожий з рангом кількості відвіданих перетинів.

Далі було розраховано середнє значення всіх рангів, яке дає остаточний ранг алгоритмів генерації лабіринтів за рівнем складності:

1. Алгоритм Олдоса-Бродера.
2. Алгоритм Вілсона.
3. Алгоритм Краскала.
4. Алгоритм Прима.
5. Алгоритм Sidewinder.

6. Алгоритм Еллера.

З'ясувавши ранг алгоритмів, з'являється можливість визначення характеристик, які розрізняють різні рівні складності алгоритмів.

Кількість перетинів корелює зі складністю лабіринтів. Більша кількість перетинів означає, що лабіринт складніше, і що є більше шансів пропустити правильний шлях. Значення рівня складності в залежності від типу алгоритму:

1. Кращі результати були досягнуті за допомогою алгоритмів Олдоса-Бродера та Вілсона, які відбуваються з алгоритмів рівномірного охоплення дерев пошукати графах. Агенти пошуку з працею переміщаються по лабіринту, тому що шляхи неупереджені в будь-якому напрямку.

2. Наступною парою по рейтингу є алгоритми Краскала і Прима. Вони походять з алгоритмів знаходження мінімальних остовних дерев в графах. У порівнянні з першою групою шляху тут не так рівномірно розподілені, що робить лабіринти менш складними.

3. Найгірша пара –алгоритми Еллера та Sidewinder. Вони походять з алгоритмів пошуку на графі. З іншого боку більшість агентів пошуку використовують ті ж самі методи, що дозволяє їм вирішувати лабіринт легко. Тому лабіринти генерується таким чином, щоб задовольнити рішення агентів.

2.3 Розробка модифікованого методу генерації лабіринтів

Виходячи із аналізу, що був проведений в попередньому підрозділі найбільш доцільними для використання є алгоритми Олдоса-Бродера та Вілсона. Тому потребується більш детально розглянути ці алгоритми.

Ідея алгоритму Олдоса-Бродера полягає в побудові лабіринту з випадково вибраних невідвіданих клітин. Відповідно, клітини можуть бути двох видів – невідвідані і додані в лабіринт.

Алгоритм Олдоса-Бродера включає наступні кроки:

0. Ініціалізувати лабіринт, вибравши в ньому випадкову клітку.

1. Вибрати будь-яку сусідню (для поточної клітки) клітку. Сусідніх клітин може бути 2, 3 або 4, в залежності від розташування клітини.

2. Якщо сусідня клітина не є частиною лабіринту – додати її в лабіринт, тобто прибрати стінку, що сполучає дві клітини. Інакше перехід до кроку 3.

3. Якщо не залишилося невідвіданих клітин – останов. Інакше перехід до кроку 1.

Алгоритм використовує тільки сусідні клітини. Це означає, що якщо на нульовому кроці вибрати, наприклад, клітину (2,2), то сусідній для неї буде наприклад, клітина (4,2). Відповідно до алгоритму, клітина (4,2) буде додана в лабіринт, після чого алгоритм знову перейде на крок 1, на якому буде знову вибирати сусідню клітку, але вже для поточної, тобто для клітини (4,2). Але внаслідок того, що в алгоритмі не передбачено використання пам'яті для відвіданих і невідвіданих клітин, сусідній для клітини (4,2) може знову опинитися клітина (2,2). Такого роду «блукання» по вже відвіданих клітинам і є головним недоліком алгоритму, і при збільшенні розмірності лабіринту час його побудови прагне до нескінченності.

Наведемо приклад: на поле розміром 101 × 101 залишилися дві невідвідані клітини. Імовірність випадковим чином потрапити хоча б в одну з них близька до нуля.

Алгоритм Вілсона усуває цей мінус – все невідвідані клітини заносяться в список. Також даний алгоритм видаляє цикли, якщо такі утворюються. Циклом називається ситуація, коли в лабіринті присутній замкнутий шлях з деякої клітини в саму себе.

Алгоритм Вілсона містить наступні етапи:

1. Занести всі клітини лабіринту в список невідвіданих клітин.

2. Вибрати будь-яку початкову клітку, додати її в лабіринт, прибрати з списку невідвіданих.

3. Вибрати випадковим чином клітку зі списку невідвіданих.

4. Вибрати випадкову сусідню клітку для поточної клітини.

5. Якщо сусідня клітина є частиною лабіринту – додати пройдені на поточній ітерації клітини в лабіринт, прибрати стінки між ними, прибрати зі списку невідвіданих клітин.

5.1.Еслі утворився цикл -- видалити його, тобто знову занести в список невідвіданих ті клітини, які утворили цикл, продовживши рух від клітини, з якої цикл починався.

5.2.Іначе перейти в сусідню клітку і продовжувати виконувати крок 4 до тих пір, поки не виконається крок 5.

6. Якщо список невідвіданих клітин порожній – останов. Інакше перейти до кроку 3.

Необхідно пояснити метод видалення циклів. Припустимо, що рух розпочато з клітки (3,3). Потім алгоритм Вілсона знайде множину *A* невідвіданих клітин (їх число залежить тільки від вибору випадкових клітин на кроках 3 і 4), остання з яких на черговому кроці 4 алгоритму привела знову до клітки (3,3). Це означає, що необхідно зробити все клітини з знайденого набору *A* знову невідвідуваними. Далі необхідно продовжити виконання алгоритму з точки (3,3).

Варто відзначити, що дані дії не гарантують, що при подальшій роботі алгоритму з точки (3,3) він не знайде множину клітин *A* (можливо, що складається з тих же самих клітин), яка знову призведе до утворення циклу.

Можна зробити висновок, що алгоритм Уїлсона працює набагато ефективніше, але вимагає додаткових витрат пам'яті для зберігання списку невідвіданих клітин. Також варто відзначити і наявність у даного алгоритму засобів видалення циклів, яке уповільнює час його роботи.

Алгоритм Вілсона, завдяки своїм особливостям, є найбільш відповідним для поставленого завдання. Але його базовий варіант генерує лабіринт тільки з одним проходом. Тому пропонується виконати його модифікацію.

Таку модифікацію дозволяє зробити ще одна особливість: для алгоритму не важлива форма поля лабіринту, тому можна генерувати лабіринт частинами. Іншими словами, можна задати будь-які обмеження на індекси двовимірного

масиву. Наприклад, будувати лабіринт алгоритмом Вілсона в разі, коли індекси задовольняють деяким обмеженням, наприклад, $5 < i < 21$, а $j > i / 2$, де i відповідає за рядки лабіринту, а j – за стовпці.

Примітно, що у лабіринті, побудованому в обмеженому нерозривному просторі, також буде прохід з будь-якої клітини в будь-яку.

Тому було введено поняття циклової частини лабіринту – це частина лабіринту, яка утворює замкнутий простір. Відповідно, крім самої циклової частини, позначеної як 1, поле розбивається ще на дві частини – частина всередині циклу і поза ним.

Циклічну частину можна задати наступним чином:

$$A = \begin{cases} \left|\frac{N}{8}\right| < i < \left|N - \frac{N}{8}\right| \\ \left|\frac{N}{8}\right| < j < \left|N - \frac{N}{8}\right| \end{cases} \quad (2.2)$$

$$B = \begin{cases} \left|\frac{N}{2} - 3\right| < i < \left|\frac{N}{2} + 3\right| \\ \left|\frac{N}{2} - 3\right| < j < \left|\frac{N}{2} + 3\right| \end{cases} \quad (2.3)$$

І взяти простір A/B.

Тоді, модифікований алгоритм Вілсона включає наступні етапи:

1. У початковому лабіринті (рисунок 2.9 а) виділити циклічну частину.

2. Базовим алгоритмом Вілсона побудувати лабіринт в циклічній частині (рисунок 2.9 б).

3. Базовим алгоритмом Вілсона побудувати лабіринти всередині циклу (рисунок 2.9 в) і поза ним (рисунок 2.9 г).

4. Зробити виріз на кордоні циклової і зовнішньої деяке заздалегідь задане число проходів (рисунок 2.9 д).

Потребується пояснити крок 4. Принцип вирізання може бути наступним: якщо межа області є прямокутником, то можна зробити деяке число вирізів k на двох суміжних сторонах цього прямокутника, після чого зробити на двох інших

сторонах k діаметрально протилежних вирізів. Таким чином, буде завжди парне число вирізів.

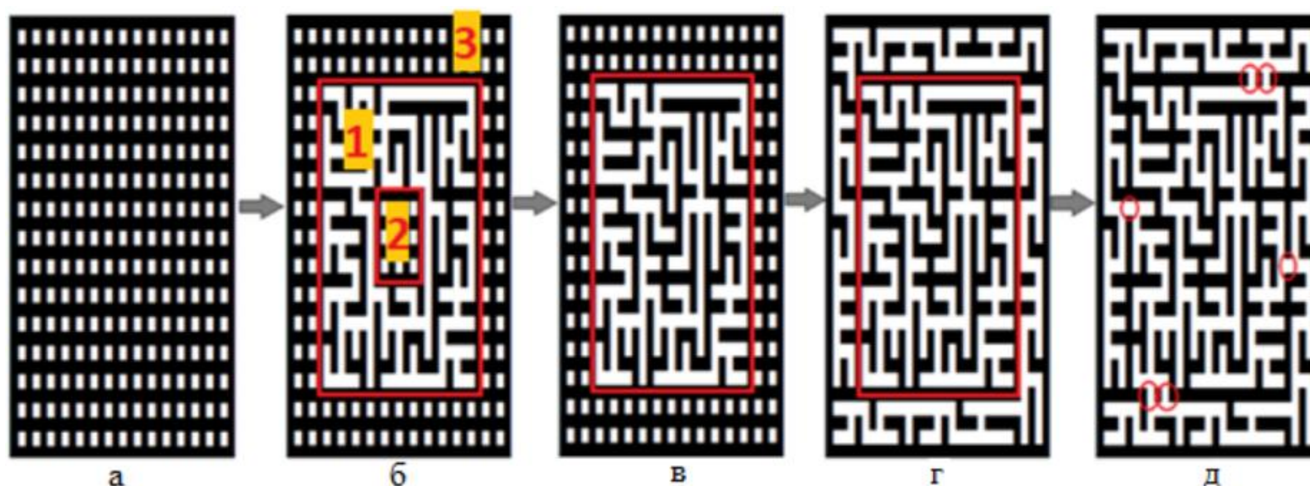

Рисунок 2.9 – Модифікований алгоритм Вілсона

Виконано модифікацію алгоритму Вілсона, який працює набагато ефективніше тому, що включає наявність засобів видалення циклів, яке уповільнює час його. Цей алгоритм не вимагає додаткових витрат пам'яті для зберігання списку невідвіданих клітин. Тепер треба розглянути методи проходження сгенерованних лабіринтів.

3 РОЗРОБКА МЕТОДУ ПРОХОДЖЕННЯ ЛАБІРИНТІВ

3.1 Дослідження існуючих методів проходження лабіринтів

3.1.1 Алгоритм А-стар

А-стар (A*) – популярний інформований алгоритм пошуку. «Інформований» означає, що замість тупого перебору варіантів алгоритм досліджує найбільш багатообіцяючі з них. Критерієм для визначення перспективності є евристична функція [20].

Як і багато інших алгоритми пошуку, A* працює з простором станів. Простір станів – це граф, в якому:

– вузли – можливі стану системи.

– ребра – дозволені переходи між станами.

Якщо мета пошуку – знайти шлях, то в якості вхідних даних задаються початковий і кінцевий стан. Якщо мета – знайти рішення, то кінцевий стан невідомо. В такому випадку повинен бути заданий спосіб визначити, чи є стан шуканим.

Алгоритм починає рух по простору станів з початкового стану. Для кожного вузла розраховується кілька величин:

– g – фактична вартість шляху від початку до поточного вузла;

– h – оціночна вартість шляху від поточного вузла до мети;

– f – повна вартість шляху , є сумою фактичної та оціночної вартостей: f = g + h.

Вузли-кандидати містяться в так званому «відкритому списку». Вузли в цьому списку відсортовані за повною вартістю. Першим з відкритого списку завжди береться вузол з найменшою вартістю. Пройдені вузли поміщаються в «закритий» список і більше ніколи не відвідуються. Це дозволяє прискорити алгоритм і знизити споживання пам'яті.

Ефективність алгоритму А* сильно залежить від того, наскільки точно евристична функція оцінює відстань до цілі.

А* є «повним» алгоритмом. Він завжди знаходить шлях до вирішення, якщо такий шлях існує.

Схема алгоритму А* наведено на рисунку 3.1.

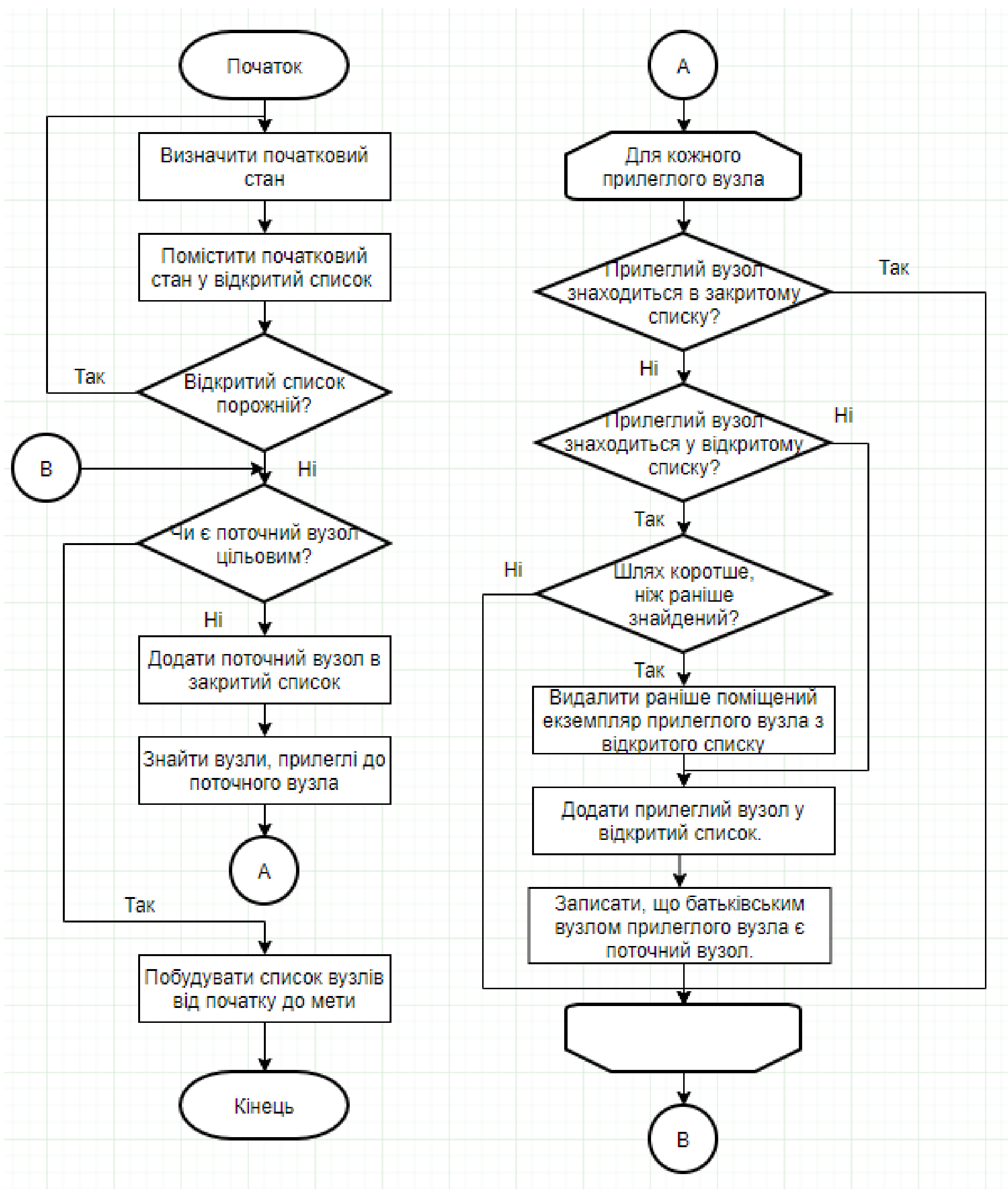

Рисунок 3.1 – Схема алгоритму А*

3.1.2 Алгоритм Лі

Хвильовий алгоритм (Алгоритм Лі) – алгоритм пошуку найкоротшого шляху на планарном графі. Належить до алгоритмів, заснованим на методах пошуку в ширину, він дозволяє побудувати шлях між двома елементами в будь-якому лабіринті [21]. Сам процес можна розділити на 2 етапи:

1. З початкового елемента поширюється в 4-х напрямках хвиля (рисунок 3.2). Елемент в який прийшла хвиля утворює фронт хвилі. На рисунку цифрами позначені номери фронтів хвилі.

Кожен елемент першого фронту хвилі є джерелом вторинної хвилі. Елементи другого фронту хвилі генерують хвилю третього фронту тощо. Процес триває до тих пір поки не буде досягнутий кінцевий елемент. Ну, або поки не стане ясно, що його не досягти.

2. Будується сама траса. Її побудова здійснюється від кінцевого елемента до початкового.

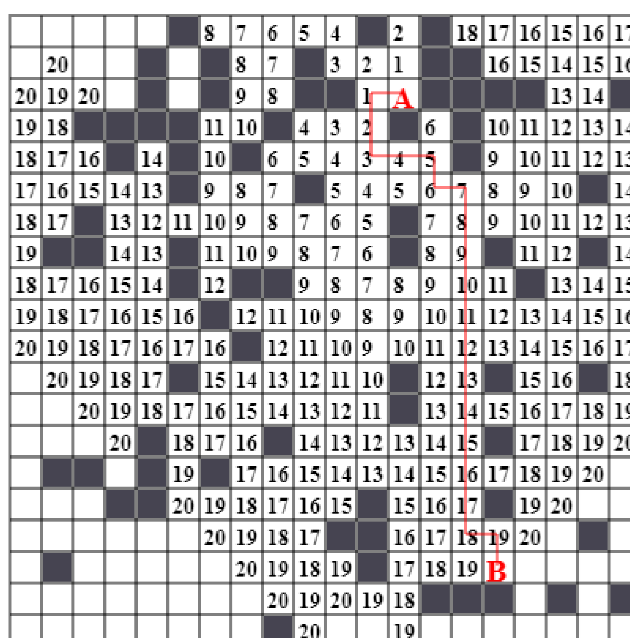

Рисунок 3.2 – Приклад роботи алгоритму Лі

Плюс алгоритму в тому, що він дозволяє знайти шлях в будь-якому лабіринті (за умови, що шлях існує). Головний недоліком даного алгоритму є те, що при побудові траси потрібен великий обсяг пам'яті.

Існує два способи визначення кількості сусідніх клітин: вважати сусідніми клітини, доступні через загальні межі і вважати сусідніми клітини, доступні через загальні межі і загальні кути [22].

3.1.3 Променевий алгоритм

Хвильовий алгоритм характеризується високою ефективністю знаходження шляху за рахунок дослідження всіх вільних осередків дискретного робочого поля (ДРП), але вимагає значного часу на поширення хвилі, Тому використовують різні методи прискорення виконання першого етапу алгоритму. Один з них - вибір початкової точки пуску хвилі.

Променевий алгоритм – це алгоритм побудови ортогонального шляху, він забезпечує проведення ламаної лінії, що обходить перешкоди і сполучає джерело і приймач. Шуканий шлях не повинен перетинати сам себе.

Лінія (промінь) будується покроково. На кожному кроці виконується переміщення з поточної комірки лінії в вільну сусідню клітинку, найбільш близьку до приймача, що дозволяє віднести променевої алгоритм до різновиду жадібного алгоритму.

У випадку досягнення приймач або всі сусіди поточної осередки зайняті, то процес пошуку шляху припиняється.

Ідея променевого алгоритму полягає в дослідженні не всіх вільних осередків ДРП, а лише по заданих напрямках. Для осередків A і B задають кількість розповсюджуваних променів і дозвіл напрямки їх руху. При проходженні променя через осередок їй присвоюють шляхову координату [23].

На рисунку 3.3, а показаний приклад проведення шляху двопроменевим алгоритмом, причому променю A_1 дозволено рух вправо і вниз, променю A_2 – вниз і вправо, променю B_1 – вгору і вліво, променю B_2 – вліво і вгору.

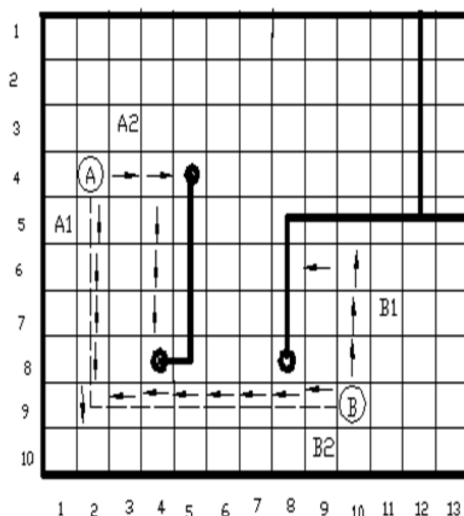

Рисунок 3.3 – Приклад проведення шляху двопроменевим алгоритмом

Зазвичай за допомогою променевого алгоритму вдається проведення 80% трас, інші проводяться хвильовим алгоритмом або конструктором.

Схема алгоритму променевого алгоритму наведено на рисунку 3.4.

Рисунок 3.4 – Схема променевого алгоритму

3.1.4 Алгоритм Дейкстри

Алгоритм Дейкстри – алгоритм на графах для знаходження найкоротшої відстані від однієї з вершин графа до всіх інших. Алгоритм працює тільки для графів без ребер з негативною вагою.

У загальному випадку цей алгоритм заснований на приписуванні вершин тимчасових позначок, причому позначка вершини дає верхню межу довжини від *s* до цієї вершини. Ці позначки (їх величини) поступово зменшуються за допомогою деякої ітераційної процедури, і на кожному кроці ітерації одна з тимчасових позначок стає постійною [24].

Алгоритм Дейкстри складається з n ітерацій. на черговій ітерації вибирається вершина *v* з найменшою довжиною найкоротшого шляху серед ще не помічених (на першій ітерації буде обрана стартова вершина *s*). Обрана таким чином вершина v вважається «поміченої». Далі, на поточній ітерації, з вершини *v* виробляються релаксації: проглядаються всі ребра, що виходять з вершини *v*, і для кожної вершини алгоритм намагається поліпшити значення найкоротшого шляху.

На цьому поточна ітерація закінчується, алгоритм переходить до наступної ітерації (знову вибирається вершина з найменшою довжиною найкоротшого шляху, з неї виробляються релаксації тощо). При цьому після n ітерацій все вершини графа стануть поміченими і алгоритм свою роботу завершує. Стверджується, що знайдені значення найкоротшого шляху і є шукані довжини найкоротших шляхів з *s* в *v*.

Слід зазначити, що якщо не всі вершини графа досяжні з вершини s, то значення найкоротшого шляху для них так і залишаться нескінченними.

Зрозуміло, що кілька останніх ітерацій алгоритму будуть якраз вибирати ці вершини, але ніякої корисної роботи проводити не будуть (оскільки нескінченна відстань не зможе прорелаксувати інші). Тому алгоритм можна зупиняти, як тільки буде обрано вершина з нескінченним відстанню.

Зазвичай потрібно знати не тільки довжини найкоротших шляхів, а й отримати самі шляхи. Для цього достатньо так званого масиву «предків»: масиву p, в якому для кожної вершини v зберігається номер вершини, що є передостанньою в найкоротшому шляху до вершини v.

Масив «предків» будується просто: при кожній успішній релаксації, тобто коли з обраної вершини v_0 відбувається поліпшення відстані до деякої вершини v, записується, що «предком» вершини v є вершина v_0.

Алгоритм Дейкстри враховує ваги при просуванні по графу, до того ж він оновлює дані в раніше досягнутих вузлах шляху. Таким чином, гарантовано буде знайдений найкоротший шлях, якщо він існує. Однак у цього алгоритму є слабка сторона – пошук в ширину: він ігнорує напрям до мети.

Графічне уявлення алгоритму Дейкстри наведено на рисунку 3.5.

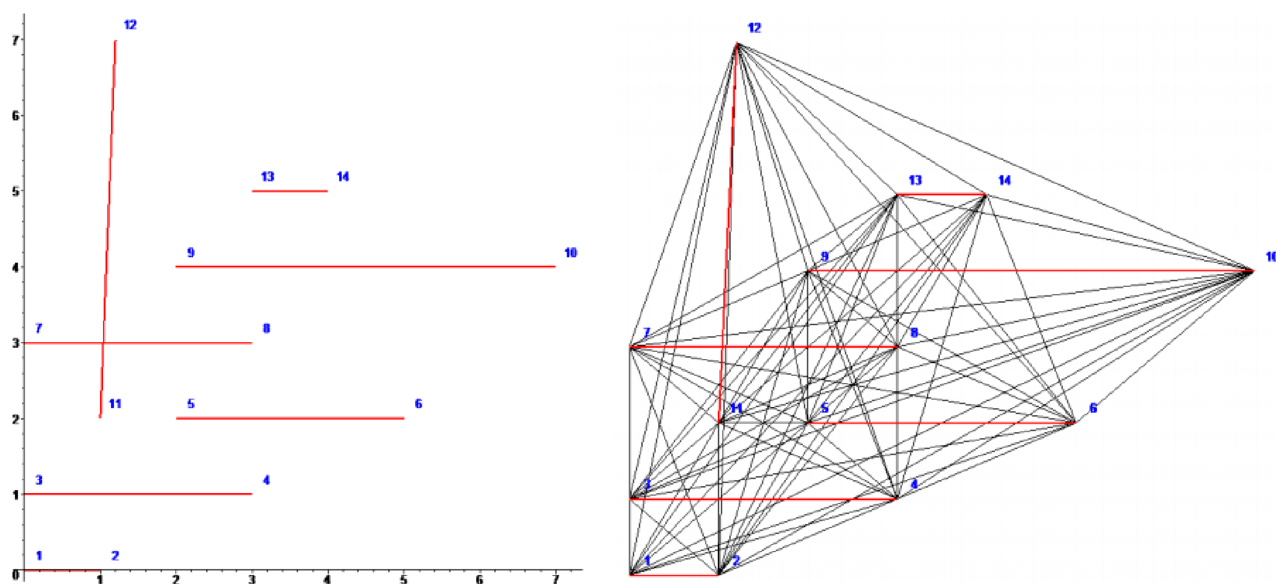

Рисунок 3.5 – Графічне уявлення алгоритму Дейкстри

3.1.5 Пошук в глибину

Пошук в глибину (Depth-first search, DFS) перевіряє кожен можливий шлях цілком, перш ніж перейти до іншого можливого маршруту. При обході дерева маршрутів кожен раз вибирається ліва гілка, поки не буде досягнутий кінцевий вузол або знайдена мета. Якщо поточний вузол – кінцевий, відбувається підйом на один рівень вгору і вибирається права гілка дерева, а з неї триває рух по лівим гілкам до тих пір, поки не зустрінеться мета або кінцевий вузол. ця процедура повторюється, поки не буде виявлена мета або пройдений останній вузол простору.

Однією з можливих оптимізацій алгоритму DFS є алгоритм послідовних наближень при пошуку в глибину (Iterative deeping depth-first search, IDDFS).

Насправді в алгоритмі пошуку в глибину існує ще одна проблема – вибір правильної глибини зупинки. Якщо вона буде дуже маленькою, то шлях не буде знайдений, якщо занадто великий, то можна витратити багато часу і грошей даремно. Ці проблеми вирішуються ітеративним поглибленням – прийомом, при якому виконується DFS зі збільшенням глибини до тих пір, поки шлях не буде знайдений. При пошуку шляху можна не починати з глибини, що дорівнює одиниці, а відразу почати з глибини, яка дорівнює відстані по прямій від старту до мети. Цей пошук є асимптотично оптимальним серед усіх переборних алгоритмів по часу і пам'яті [25].

Формальний алгоритм DFS для проходження лабіринту включає наступні етапи [26]:

1. Зробити початкову клітку поточною і відзначити її як відвідану.
2. Поки є невідвідані клітини:

2.1. Якщо поточна клітина має невідвіданих «сусідів»:

2.1.1. Додати поточну клітку в стек.

2.1.2. Вибрати випадкову клітку з сусідніх.

2.1.3. Прибрати стінку між поточною кліткою і обраною.

2.1.4. Зробити обрану клітку поточною і відзначити її як відвідану.

2.2. Інакше якщо стік не порожній:

2.2.1. Видалити клітку з стека.

2.2.2. Зробити її поточною.

3. Інакше:

3.1. Вибрати випадкову невідвідування клітку, зробити її поточною і відзначити її як відвідану.

Графічне уявлення алгоритму пошуку в ширину наведено на рисунку 3.6.

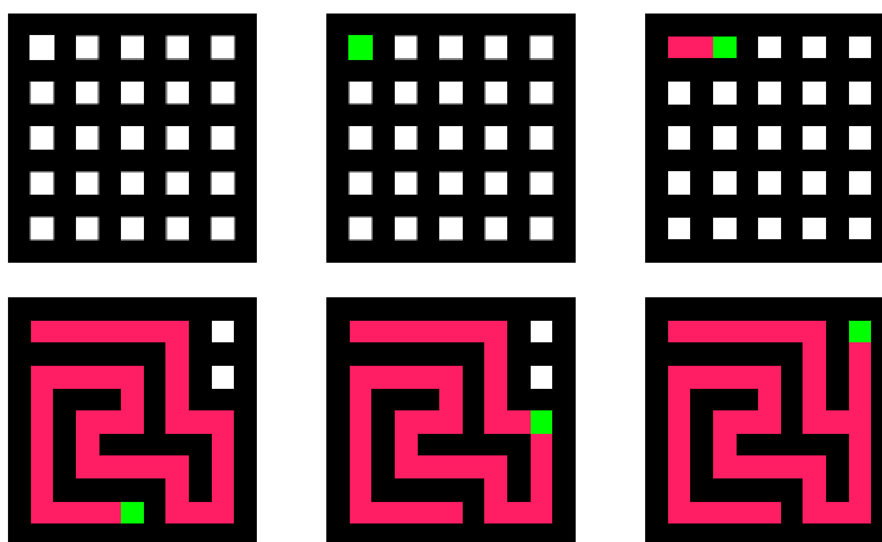

Рисунок 3.6 – Графічне уявлення роботи алгоритму пошуку в ширину

3.1.6 Пошук в ширину

При роботі алгоритму пошуку в ширину (Breadh-first search, BFS) сітка представляється у вигляді зваженого графа, де з кожної вершини можливий перехід в одну з 8 сусідніх вершин (крім граничних). Ваги вершин – 0 або ∞ (для випадку повної непрохідності) [27].

Для роботи алгоритму потрібно організувати чергу, в яку будуть послідовно добавкою додаватися вершини. Крім цього, щоб мати можливість відновити пройдений шлях, знадобиться масив «предків», де буде вказано, з якої вершини

веде шлях в дану. необхідний також номер стартовою вершини *s* (координати стартової осередки). Сам алгоритм можна розуміти як процес «підпалювання» графа: на нульовому кроці «підпалюємо» тільки вершину *s*. На кожному наступному кроці «вогонь» з уже «палаючої» вершини перекидається на всіх її сусідів. Таким чином, за одну ітерацію алгоритму відбувається розширення «кільця вогню» в ширину на одиницю (звідси і назва алгоритму).

Також можна порахувати довжини найкоротших шляхів (для чого просто треба завести масив довжин шляхів) і компактно зберегти інформацію, достатню для відновлення всіх цих найкоротших шляхів (завести масив «предків», в якому для кожної вершини необхідно зберігати номер вершини, з якої ми в неї потрапили). При досягненні кінцевої вершини алгоритм припиняє свою роботу і відновлюється загальний шлях до кінцевої вершини.

Одна з можливих оптимізацій алгоритму BFS полягає в додаванні пошуку спрямованості руху. Найчастіше шлях між двома вершинами виявляється ув'язненим в відповідний прямий кут, тому додавати вершину в чергу можна, не обходячи послідовно всі сусідні з нею, а починаючи від бісектриси відповідного прямого кута і розходячись в сторони, при цьому беручи по черзі вершини з кожної зі сторін по відношенню до бісектрисі.

Інша можливість оптимізації – двонаправлений пошук в ширину. Це покращує простий пошук BFS тим, що запускаються два одночасних пошуку в ширину з стартового і кінцевого вузлів, і процес зупиняється, коли вузол з одного фронту пошуку знаходить сусідній вузол з іншого фронту. Це може поліпшити простий пошук в ширину, який залишається при цьому не дуже ефективним.

Одним з головних переваг алгоритму BFS є те, що він гарантовано знаходить найкоротший шлях з початкової вершини в кінцеву. Однак недоліком є необхідність дослідження великого числа сторонніх точок, що вимагає зайвих витрат пам'яті для зберігання інформації, а також додаткового часу.

Графічне уявлення алгоритму пошуку в ширину наведено на рисунку 3.7.

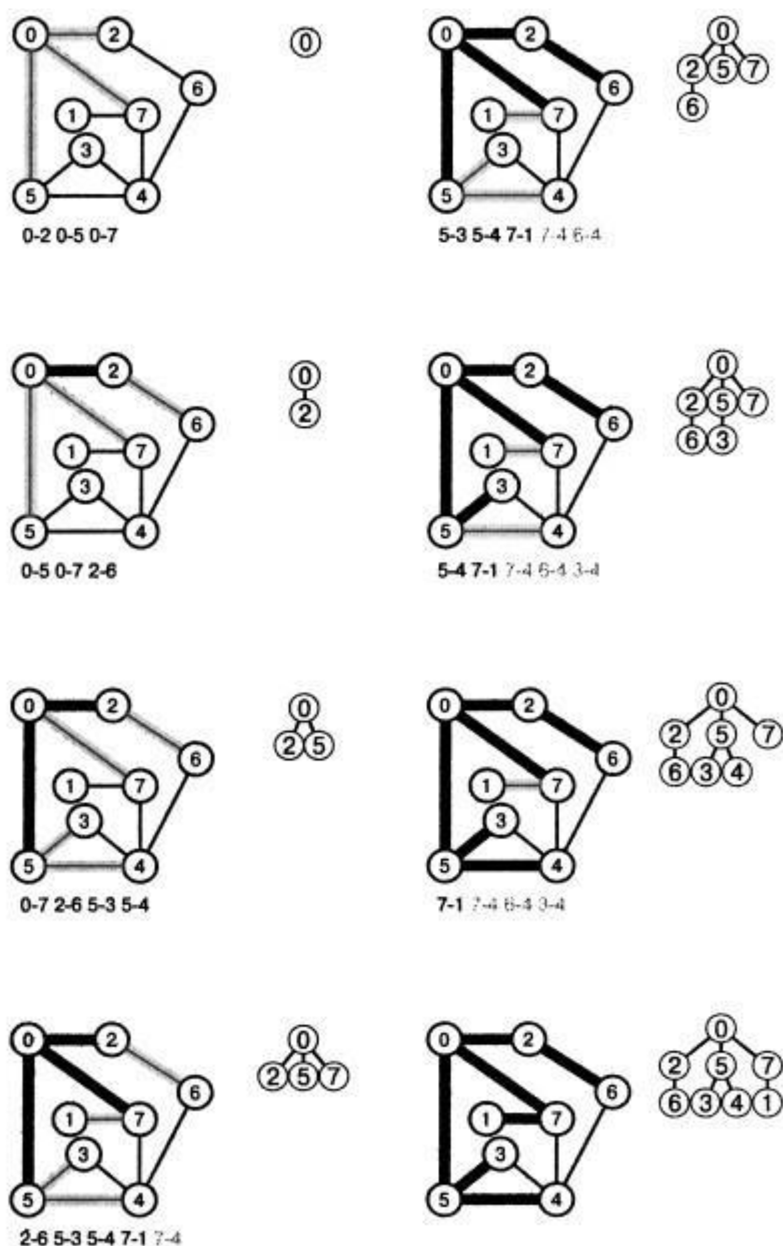

Рисунок 3.7 – Графічне уявлення роботи алгоритму пошуку в ширину

3.2 Порівняння існуючих методів проходження лабіринтів

Лабіринти, в першу чергу, можуть відрізнятися за розмірами. Нижче представлено чотири основних типорозміру лабіринтів [28]:

– малі (розміром близько 20x20 блоків);

– середні (розміром від 40х40 до 70х70 блоків);

– великі (розміром близько 100х100 блоків);

– гігантські (розміром близько 200х200 блоків і більше).

Лабіринти менших і більших розмірів не розглядаються.

Інша класифікація лабіринтів – за кількістю перешкод:

– вільні (менше 10% заповнення перешкодами);

– середні (від 10 до 40% заповнення перешкодами);

– тісні (більше 50% заповнення перешкодами).

Приклад карти середнього розміру (40х34) і середньої заповнювання перешкодами (268/1360 ~ 0,19 = 19) представлений на рисунку 3.8.

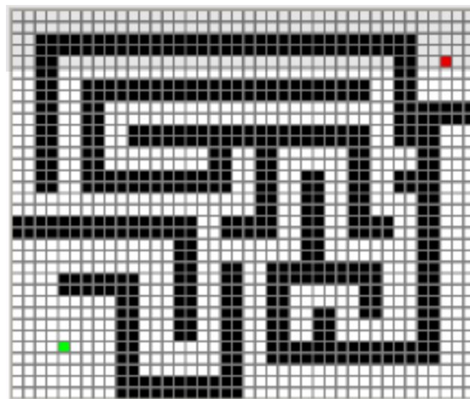

Рисунок 3.8 – Приклад лабіринту

У попередньому підрозділі були розглянуті наступні алгоритми:

– A* (так званий А-стар);

– хвильової алгоритм (алгоритм Лі);

– хвильової алгоритм (восьминаправлена модифікація);

– променевий алгоритм;

– алгоритм Дейкстри;

– пошук в глибину;

– пошук в ширину;

– алгоритм правої руки;

– алгоритм лівої руки.

Для вивчення описаних вище алгоритмів і їх оцінці за методом, який буде описаний нижче, була зібрана статистика, яка представляє собою результуючі характеристики роботи кожного з алгоритмів пошуку шляху. Детальний аналіз цих даних може показати, який з алгоритмів поводиться краще в різних типових випадках. Список цих характеристик виглядає наступним чином [29]:

– час роботи алгоритму (час обчислень) – ;
– довжина маршруту;
– обсяг пам'яті для обчислень;
– небезпека маршруту;
– гладкість маршруту.

Далі слід надати детальні пояснення щодо кожної з характеристик. Нижче наведені більш суворі математичні формулювання кожної:

Час роботи алгоритму (час обчислень) – одна з найбільш значущих характеристик. Очевидно, що менше значення однозначно говорить про кращу роботу алгоритму, однак для отримання достовірних даних необхідна достатня вибірка, за якої обчислюється середнє значення. Дана величина залежить від розмірів карти.

Довжина маршруту - величина, також важлива для розгляду, тому що занадто довгий маршрут призводить до поганого результату. Довжина маршруту вимірюється один раз для кожного алгоритму.

Позначимо довжину маршруту як L_R, а кількість точок як N_R при цьому не обов'язково, що $L_R = N_R$. Однак кількісна міра для даної величини малоінформативна, і для збереженої статистики обчислюється відносна величина

$$L = \frac{L_R}{L_B} \tag{3.1}$$

де L_B – умовний ідеальний шлях з стартової точки в фінішну.

Якщо порівняти величину L для двох алгоритмів за допомогою відносини, то виключиться вплив величини L_R, тому що вона скоротиться. Також для кожного блоку задана прохідність і, таким чином, величина L_R, що залежить від N_R, буде обчислюватися за формулою:

$$L_R = \sum_{i=0}^{N_{R-1}} k_i \quad (3.2)$$

де k_i – коефіцієнт прохідності для кожного блоку, що входить в маршрут (для простого вільного блоку дорівнює одиниці). Величина L не залежить від розмірів карти.

Якщо алгоритм працює швидко, але споживає багато пам'яті, його використання може бути небажаним, а іноді навіть і неприйнятним. Тому обсяг пам'яті, використаний алгоритмом для обчислень, також показовий і повинен бути визначений (один раз для кожного алгоритму). Дана величина позначена як M і залежить від розмірів карти.

Короткий маршрут в багатьох випадках може вважатися поганим, якщо він пролягає поблизу перешкод. Кількісна міра цієї характеристики - число точок маршруту, що є сусідами з перешкодами. З огляду на розбіжності розмірів карт, цю величину слід вимірювати в процентному відношенні від довжини маршруту в точках за формулою:

$$H = \frac{n_H}{N_R} \quad (3.3)$$

де n_H – число точок маршруту, що є сусідами з перешкодами. Дана величина не залежить від розмірів карти.

Для вимірювання гладкості маршруту необхідно попарно розглянути всі точки маршруту як відрізків і виміряти кут між кожною парою відрізків. Можливі наступні варіанти:

– немає повороту (позначимо кількість таких стиків як n_0);

- малий поворот – 45 (кількість таких стиків – n_1)
- крутий поворот – 90 (кількість таких стиків – n_2);
- малий розворот – 135 (кількість таких стиків – n_3);
- повний розворот – 180 (кількість таких стиків – n_4).

При цьому зручніше використовувати відносне число поворотів кожного типу, виражене у відсотках від загального числа поворотів. Величина буде обчислюватися за формулою:

$$A_i = \frac{n_i}{N-2} \tag{3.4}$$

При цьому очевидно, що справедливе співвідношення:

$$\sum_{i=0}^{4} A_i = 1 \tag{3.5}$$

Далі необхідно ввести таку величину A, яка при повній відсутності поворотів буде дорівнювати одиниці, а при наявності переважної більшості повних розворотів буде прагнути до нуля. При цьому відсутність повороту не змінюватиме величину A, а більш різкі повороти будуть знижувати значення величини сильніше, ніж малі повороти. Значення величини A буде обчислюватися за формулою:

$$A = 1 - 0.25A_1 - 0.5A_2 - 0.75A_3 - A_4 \tag{3.6}$$

Виходячи з формули отримуємо, що при переважанні малих поворотів величина A буде прагнути до значення 0,75; при переважанні крутих поворотів – до 0.5 тощо. При цьому неможливо ніяке поєднання величин A_i, що дає A < 0.

Отримана величина гладкості A інваріантна до довжини маршруту.

Для отримання достатньої кількості статистичних даних необхідно провести безліч тестів. Вони повинні бути розділені на 12 груп за такими критеріями: розмір карти і її наповненість перешкодами. На основі такого поділу за допомогою аналізу статистичних даних можна простежити, як змінюються характеристики кожного

алгоритму для різних типів карт, а також порівнювати з ним різні алгоритми на одному типі карт. Тести всередині кожної групи проводяться з варіюванням параметрів карти в заданих для даного типу межах і з різною структурою самого лабіринту.

Жоден з алгоритмів не є універсальним для всіх лабіринтів, тому потребується виробити стратегію, яка буде полягати у виборі відповідного алгоритму в залежності від параметрів карти, ситуації та обчислювальної потужності комп'ютера.

Для того щоб зробити комплексну оцінку кожного алгоритму за отриманими характеристиками, необхідно модифікувати деякі величини з абсолютних у відносні.

Для величини часу очевидно, що час кожного алгоритму буде не більше сумарного часу всіх алгоритмів. Отже, для того щоб укласти величину часу в діапазон 0 ... 1, необхідно використовувати нормалізовану характеристику часу:

$$T_N = 1 - \frac{T_i}{\sum_{i=1}^{m} T_i} \tag{3.7}$$

де m – число алгоритмів.

Аналогічні міркування можна застосувати і для обчислення величин, нормалізованих довжини маршруту та обсягу:

$$L_N = 1 - \frac{L_i}{\sum_{i=1}^{m} L_i} \tag{3.8}$$

$$M_N = 1 - \frac{M_i}{\sum_{i=1}^{m} M_i} \tag{3.9}$$

Для характеристики небезпеки маршруту нормалізоване значення буде обчислюватися за формулою:

$$H_N = 1 - H \tag{3.10}$$

Характеристика гладкості вже нормалізована, тому $A_N = A$. На основі нормалізованих характеристик можна обчислити величину F_{alg}, що показує, наскільки алгоритм є вдалим:

$$F_{alg} = k_T * T_N + k_L * L_N + k_M * M_N + k_N * H_N + k_A * A_N \qquad (3.11)$$

де k_i – вагові коефіцієнти кожної з характеристик.

При всіх вагових коефіцієнтах, рівних одиниці, значення функції лежить в межах від 0 до 5, а в загальному випадку – від 0 до $\sum_{i=0}^{4} k_i$.

F_{alg} не залежить від розмірів карти і дозволяє порівнювати різні алгоритми як по абсолютному, так і у відносному значенні.

Далі представлені загальні результати проведеного експерименту. Вся зібрана статистика приведена не буде, так як є дуже об'ємною і недостатньо наочної. Результати представлені в таблиці 3.1, в якій є 12 блоків, що відповідають кількості груп тестів. В кожному блоці наведені середні значення величини F.

Також таблиці введено наступні позначення:

- A* – алгоритм А-Стар;
- Wave – хвильової алгоритм (Лі);
- Wave8 – хвильової алгоритм (8 напрямків);
- Beam – променевої алгоритм;
- D – алгоритм Дейкстри;
- Depth – алгоритм пошуку в глибину;
- Width – алгоритм пошуку в ширину;
- R – алгоритм правої руки;
- L – алгоритм лівої руки.

Таблиця 3.1 – Результати основної серії тестів

	Малі		Середні		Великі		Гігантські	
Вільні	A*	6,82	A*	6,88	A*	6,22	A*	6,51
	Wave	8,16	Wave	8,68	Wave	8,14	Wave	8,04
	Wave8	8,27	Wave8	8,59	Wave8	8,13	Wave8	8,57
	Beam	1,53	Beam	2,34	Beam	3,65	Beam	1,03
	D	6,54	D	7,35	D	6,69	D	7,04
	Depth	4,32	Depth	8,12	Depth	1,23	Depth	0
	Width	8,17	Width	8,41	Width	8,04	Width	8,31
	R	5,62	R	6,2	R	5,65	R	5,41
	L	5,78	L	6,44	L	5,74	L	5,50
Середні	A*	6,41	A*	6,71	A*	6,18	A*	6,31
	Wave	8,01	Wave	7,71	Wave	8,06	Wave	8,02
	Wave8	8,16	Wave8	7,86	Wave8	8,06	Wave8	8,15
	Beam	0,97	Beam	0,32	Beam	0,11	Beam	0,05
	D	6,46	D	5,99	D	6,58	D	6,58
	Depth	4,16	Depth	0,92	Depth	0,42	Depth	0
	Width	8,02	Width	7,7	Width	8,02	Width	8,11
	R	3,71	R	5,69	R	5,8	R	5,43
	L	3,72	L	5,81	L	5,76	L	5,49
Тісні	A*	6,05	A*	6,5	A*	6,06	A*	6,2
	Wave	7,53	Wave	7,67	Wave	7,98	Wave	7,98
	Wave8	7,72	Wave8	7,84	Wave8	8,05	Wave8	8,14
	Beam	0,53	Beam	0	Beam	0	Beam	0
	D	6,44	D	6,05	D	6,33	D	6,48
	Depth	3,05	Depth	0,56	Depth	0	Depth	0
	Width	7,57	Width	7,68	Width	7,97	Width	8,09
	R	7,2	R	5,55	R	5,78	R	5,45
	L	7,24	L	5,49	L	5,81	L	5,7

Далі наведені результати у вигляді таблиці, для окремого порівняння алгоритмів правої і лівої руки (таблиця 3.2).

Таблиця 3.2 – Результати додаткової серії тестів

	Малі		Середні		Великі		Гігантські	
Вільні	R	7,26	R	7,35	R	7,46	R	7,33
	L	7,84	L	7,22	L	7,7	L	7,44
Серед.	R	7,81	R	7,69	R	7,62	R	7,4
	L	7,83	L	7,81	L	7,56	L	7,32
Тісні	R	7,31	R	7,64	R	7,88	R	7,68
	L	7,26	L	7,42	L	7,73	L	7,59

За результатами дослідження можна зробити наступні висновки по кожному з алгоритмів:

1. Алгоритми А-Старт (А*) та Дейкстри – найменш продуктивні і найбільш вимогливі до пам'яті, проте дають, найчастіше, найкоротший маршрут.

2. Хвильовий алгоритм (Лі) перевершує по швидкодії восьмінаправленний хвильової алгоритм приблизно на 30-60%, однак, з огляду на їх високу продуктивність, цей факт має малий вплив. При цьому, класичний хвильової алгоритм дає погано згладжений шлях, тоді як восьмінаправленний дає мінімальне число поворотів і короткий шлях, який можна порівняти за довжині з результатами А* і алгоритму Дейкстри.

3. Променевий алгоритм має високий відсоток невдалих запусків, внаслідок чого можна зробити висновок, що він застосовується лише для дуже обмеженого кола завдань при максимально простій топології навколишнього середовища, однак має високу швидкодію. Застосування алгоритму в реальних задачах недоцільно.

4. Пошук в глибину погано справляється з великими лабіринтами, з огляду на те, що перевищується глибина вкладеності рекурсії ще до знаходження шляху.

Має високу продуктивність (вище хвильових алгоритмів), однак непридатний на великих лабіринтах зважаючи на зазначені причин.

5. Пошук в ширину можна порівняти з восьминаправленним хвильовим алгоритмом за загальними результатами, хоча результуючий маршрут довший. В цілому трохи перевершує класичний хвильової алгоритм.

6. Алгоритми правої і лівої руки – найбільш швидкі і найменш вимогливі до пам'яті, проте мають приблизно п'ятдесятивідсоткову статистику неспрацьовування. В ході додаткової серії тестів виявити істотної переваги одного алгоритму над іншим не вдалося.

За результатами тестів найефективнішим алгоритмом є восьмінаправленний хвильової алгоритм – він найбільш продуктивний в більшості тестів. Так само перевагами даного алгоритму є: незначне використання пам'яті, короткий результуючий шлях, який має середню ступінь безпеки і високу ступінь гладкості.

3.3 Розробка модифікованого методу пошуку шляхів в лабіринті

Для пошуку шляхів в лабіринті існує безліч алгоритмів, проте всі вони мають свою спеціалізацію. Є алгоритми, які можуть як пропустити певну кількість проходів, так і додати проходи, які після закінчення роботи алгоритму не будуть являтися рішенням лабіринту. Інша частина алгоритмів має інший принцип роботи: замість пошуку шляхів алгоритм знаходить тільки множину клітин, які не ведуть в тупик. А це означає, що для отримання множини проходів необхідно додати ще один алгоритм, який буде з множини клітин отримувати безліч проходів. Іншими словами, для отримання множини проходів необхідно вдруге пройти лабіринт. Отже, серед усіх алгоритмів немає такого, який знаходив би всі існуючі шляхи. Рішенням цієї проблеми є модифікований мурашиний алгоритм.

Для початку необхідно розглянути базовий мурашиний алгоритм [30].

Для вирішення будь-якої задачі мурашиним алгоритмом необхідно представити її у вигляді набору вершин і ребер. Якщо спроектувати таке подання на лабіринт, то вершина – це будь-яка клітина лабіринту, а ребро – прохід між двома будь-якими клітинами.

Загальні етапи мурашиного алгоритму:

1. Створити мурах на стартовій точці, яка залежить від обмежень, що накладаються на задачу.

2. Задати початкове число феромонів.

3. Пошук шляху відбувається на основі ймовірності переходу з вершини i в вершину j, яка визначається за формулою:

$$p_{ij}^{t} = \frac{(\tau_{ij}^{t})^{\alpha}\left(\frac{1}{c_{ij}}\right)^{\beta}}{\sum_{l\in J}(\tau_{il}^{t})^{\alpha}\left(\frac{1}{c_{il}}\right)^{\beta}} \tag{3.12}$$

де t – номер ітерації, τ_{ij}^{t} – рівень феромонів на ребрі (i, j) на ітерації t, c_{il} – відстань між i і j, α, β – деякі константи, J – множина вершин, доступних на ітерації t з вершини i.

4. Поки не закінчаться мурахи, повторювати крок 3.

5. Оновити феромони відповідно до формули:

$$\tau_{ij}^{t+1} = (1-\rho)\tau_{ij}^{t} + \sum_{k\in A_{ij}} \frac{Q}{L_k} \tag{3.13}$$

де ρ – інтенсивність випаровування феромонів, L_k – довжина шляху, пройденого k-м мурахою, A_{ij} – множина мурашок, що проходили по (i, j), Q – загальна кількість феромонів у мурашки, $\frac{Q}{L_k}$ – кількість феромонів, що відкладаються k-м мурахою на ребро (i, j).

6. Якщо не виконана умова зупинки, то повторювати крок 3.

Відмінність розробленого алгоритму знаходження всіх шляхів лабіринту від базового полягає в наступному:

1. Відстані між вершинами не повинні впливати на ймовірність переходу, тому покладемо, що довжина шляху, пройденого мурахою, не важлива і формула (3.12) ймовірності переходу з вершини i в j на ітерації t змінюється наступним чином:

$$p_{ij}^{t} = \frac{\left(\tau_{ij}^{t}\right)^{\alpha}}{\sum_{l \in J}\left(\tau_{il}^{t}\right)^{\alpha}} \quad (3.14)$$

2. Мурахи відкладають фіксовану кількість феромонів, яка не залежить від кількості відвіданих клітин. Тоді формула (3.13) поновлення феромонів спрощується:

$$\tau_{ij}^{t+1} = (1-\rho)\tau_{ij}^{t} + L \quad (3.15)$$

де $L = const$ – кількість феромонів, які відкладає мураха.

Введемо також в алгоритм додаткові параметри на основі поставленої задачі:

1. Матриця навігації – матриця, кожен елемент якої позначає число подальших можливих напрямків з поточної клітки. Очевидно, що матриця буде складатися тільки з чисел 0 (стіна), 1 (тупик або вхід / вихід), 2, 3 і 4. Розмірність матриці збігається з розмірністю лабіринту.

2. У лабіринт вводяться позначки, тобто місця можливих змін в ньому. Найпростіший для реалізації варіант – залишати позначки в вигляді порядкового номера мурашки в клітці, в яку він переходить після кожної розвилки. Таким чином, в кожній позначці буде міститися масив з порядкових номерів мурах, які відвідали дану клітку.

Таким чином, модифікований мурашиний алгоритм виглядає наступним чином:

0. Побудувати матрицю навігації P_Crit.

1. Поставити число мурах N_Ant.

2. Поставити однакову кількість феромонів для кожного направлення на кожній розвилці.

3. $k = 1$.

4. Запустити k-го мурашку.

5. Внести в поточну клітку позначку.

6. Якщо $P_Crit_{ij} = 1$ і клітина є входом в лабіринт – перейти в єдиному можливому напрямку до наступної розвилки. Перейти до кроку 5.

7. Якщо $P_Crit_{ij} = 1$ і клітина є виходом з лабіринту – k-й шлях знайдений. Перевірити шлях на унікальність і перейти до кроку 18.

8. Якщо 6 і 7 не виконані і $P_Crit_{ij} = 1$ – знайдений тупик. Повернутися назад на попередню розвилку, обнулити феромони в поточний напрямок. Прибрати позначки, залишені в пройденому напрямку. Перейти до кроку 5.

9. Якщо $P_Crit_{ij} = 2$ і клітина є входом в лабіринт – перейти до кроку 13.

10. Якщо $P_Crit_{ij} = 2$ і клітина є виходом з лабіринту – k-й шлях знайдений. Перевірити шлях на унікальність і перейти до кроку 18.

11. Якщо 9 і 10 не виконано, то продовжити рух в єдино можливому напрямку до найближчої розвилки. Перейти до кроку 5.

12. Якщо $P_Crit_{ij} = 3$ або $P_Crit_{ij} = 4$ – перейти до кроку 13.

13. Обчислити ймовірність p_{ij}^t за формулою (3.14).

14. Якщо $p_{ij}^t = 0$ для всіх напрямків, то повернутися на попередню розвилку. Обнулити феромон для поточного напрямку. Прибрати позначки, залишені в поточному напрямі. Перейти до кроку 5. Інакше перейти до кроку 15.

15. Якщо знайдена розвилка, в якій мураха вже був – знайдений цикл. Повернутися назад на попередню розвилку і обнулити «фіктивні» феромони на даний напрямок. Прибрати позначки, залишені в циклі. Перейти до кроку 5. Інакше перейти до кроку 16.

16. Здійснити імовірнісний перехід в наступну клітку.

17. Оновити феромони за формулою (3.15). Перейти до кроку 5.

18. Якщо $k = N_Ant$ – останов. Інакше $k = k + 1$; перейти до кроку 4.

Необхідно пояснити 15-й крок. Він має на увазі введення «фіктивних» феромонів, тобто феромонів, які діють тільки на поточного мурашку, і яким повертається вихідне значення після запуску наступного мурашки. Це необхідно в тому випадку, якщо один мураха обнулить феромони в одному з напрямків внаслідок циклу.

Тоді решта мурашок через обнулення феромону можуть втратити один або кілька шляхів в лабіринті. «Фіктивні» феромони вирішують дану проблему, однак жертвують часом пошуку шляху.

Також можна оптимізувати кроки, в яких залишаються позначки. Наприклад, можна не залишати позначки в першій і останній розвилці. Більш того, можна не залишати позначки на розвилці або розвилках, наступних за початковою точкою/, а також не залишати позначки на розвилках, що передують виходу з лабіринту. Причина полягає в тому, що вони не будуть задіяні в алгоритми видалення шляхів, іншими словами, в цих клітинах ніколи не з'являться стіни, так як якщо б вони з'явилися, лабіринт перестав би мати можливість вирішення.

Далі необхідно провести редагування лабіринту. Під редагуванням розуміється видалення деяких шляхів, тобто додавання в лабіринт стін. Для цієї мети пропонується використовувати залишені мурахами позначки. Для реалізації даного завдання були розроблені два алгоритму редагування: алгоритм видалення по довжині і алгоритм видалення схожих шляхів.

Алгоритм видалення по довжині:

1. Поставити інтервал довжин шляхів, які необхідно залишити.

2. Знайти одну або кілька позначок, що містять якомога більше шляхів, довжини яких не входять в заданий інтервал, і якомога менше шляхів, що входять в заданий інтервал.

3. Поставити в знайдених позначках стіни.

Для наступного алгоритму необхідно ввести параметр k – даний параметр задає відсоток збігу шляхів. Наприклад, k = 80 означає, що алгоритм знайде кілька

множин шляхів, які покоординатно збігаються не менше ніж на 80% і залишить з кожної множини тільки один шлях, інші ж шляху необхідно закрити. Принцип відбору шляху з кожної множини можна задати довільно. Наприклад, можна залишати тільки найкоротший шлях в кожній множини.

Алгоритм видалення схожих шляхів.

1. Задати параметр k.

2. Знайти множину шляхів, які збігаються не менше ніж на k%.

3. Вибрати з кожної множини один шлях, інші шляхи помістити в список шляхів, які необхідно закрити.

4. Знайти позначку з максимальною кількістю шляхів, які необхідно видалити, і мінімальним числом шляхів, які необхідно залишити.

5. Поставити в знайдену позначку стіну.

1) Для першого шляху обчислюється величина:

$$c = \frac{S_i}{l} \quad i = 2 \cdots N \tag{3.16}$$

де S_i – кількість координат, які збігаються у першого і i-го шляхів, $l = \max(l_1, l_i)$ – максимум з довжин двох шляхів, N – загальна кількість шляхів.

2) Якщо $c > k$, то i-й шлях збігається з першим на k% або більше.

6. Після знаходження всіх шляхів, які збігаються з першим на k%, з усього набору співпадаючих шляхів можна залишити, наприклад, найкоротший.

7. Якщо ще залишилися необроблені шляхи, повторити кроки 1-4 для них. Інакше – останов.

Схема модифікованого мурашиного алгоритму наведено на рисунку 3.9.

Запропоновано оптимізацію мурашиного алгоритму, який дозволяє з множини клітин отримувати безліч проходів, що будо потрібно для вирішення поставленої задачі.

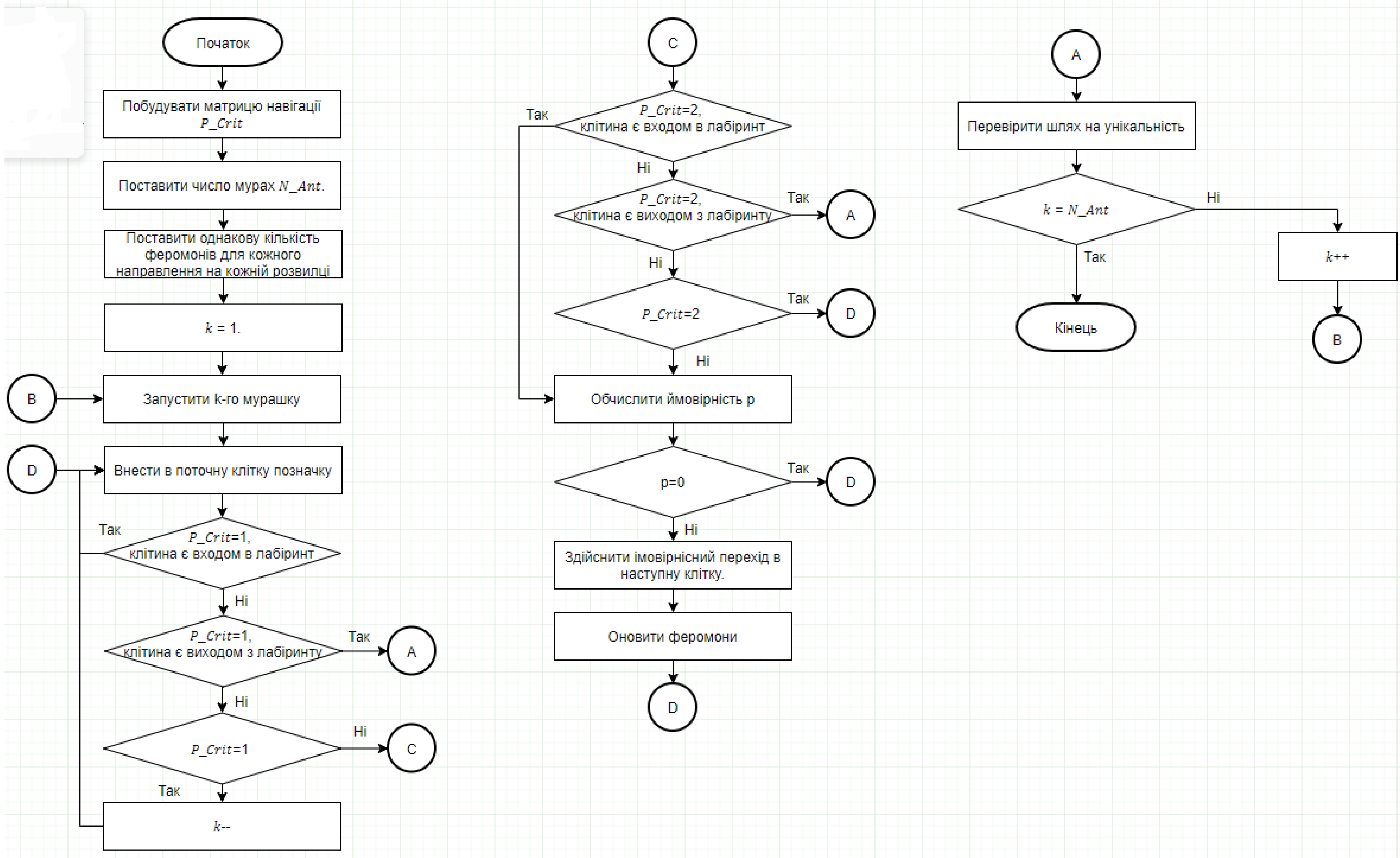

Рисунок 3.9 – Схема модифікованого мурашиного алгоритму

4 ПРАКТИЧНА АПРОБАЦІЯ ОТРИМАНИХ НАУКОВИХ РЕЗУЛЬТАТІВ

4.1 Особливості реалізації у середовищі Unity 3D

Основним засобом реалізації дипломної роботи є ігровий движок Unity 3D. Unity [31] – це мультиплатформений інструмент для розробки дву- і тривимірних застосувань і ігор, працюючий під операційними системами Windows і OS X. Створені за допомогою Unity додатки працюють під операційними системами Windows, OS X, Windows Phone, Android, Apple IOS, Linux, а також на ігрових приставках Wii, PlayStation 3 і Xbox 360. Є можливість створювати інтернет-додатки за допомогою спеціального модуля, що підключається, до браузеру Unity. Додатки, створені за допомогою Unity, підтримують DirectX і OpenGL.

Основні характеристики Unity:

– сценарії на C#, JavaScript (модифікація) і Boo (діалект мови Python із строгою типізацією для платформи.NET). У кожній з мов внесені модифікації для підтримки повного функціонала внутрішньої скриптової мови UnityScript;

– ігровий движок повністю пов'язаний з середовищем розробки. Це дозволяє прямо в редакторові випробовувати гру;

– робота з ресурсами можлива через простий Drag&Drop. Інтерфейс редактора настроюється;

– здійснена система спадкоємства об'єктів; підтримка імпорту з дуже великої кількості форматів;

– вбудована підтримка мережі;

– є рішення для спільної розробки – Version Control.

Основною мовою програмування є C# з платформою .NET 4.5 [15]. C# [12] відноситься до мов програмування з C-подібним синтаксисом, з яких його синтаксис більш близький до C++ і Java. Успадкувавши багато що від своїх попередників – мов C++, Java та ін. – C#, спирається на практичне їх використання

і виключає деякі моделі, що визначили себе як проблематичні при розробці програмного продукту.

Для написання і відладки кода використовується середовище розробки під ОС Windows – Microsoft Visual Studio 2017. Це середовище повністю інтегроване з Unity3D і операційною системою Windows. Для створення графічних об'єктів, які використовуються в генерації лабіринту, – використовується графічний редактор Blender. Він надається для розробки моделей у форматі, відповідному для Unity 3D на безкоштовній основі.

4.2 Програмна реалізація генерації лабіринту у середовищі Unity 3D

Алгоритм генерації лабіринту базується на тому, що необхідно об'єдинити комірки лабіринту в різні множини, а в останньому кроці алгоритму об'єдинити їх в одну загальну множину. Осередки знаходяться в одній множині – якщо від однієї комірки можна дійти до іншої. Спочатку всі стіни лабіринту підняті, так що все осередки знаходяться в різних множинах.

Сам алгоритм складається з 3-ох основних кроків. У циклі:

1. Обробляємо рядок, випадковим чином об'єднуючи осередки в рядку належать різним множинам (рисунок 4.1).

```
private void CreateRow(W4Maze maze, int rowNum)
   {
      for(int i = 0; i < maze.ColumnCount - 1; i++)
      {
         var cell = maze.GetCell(i, rowNum);
         var nextCell = maze.GetCell(i + 1, rowNum); if (cell.Set != nextCell.Set)
         {
            if (UnityEngine.Random.Range(0, 2) > 0)
            {
               RemoveHorizonWallBetweenCells(
                  maze,
                  cell,
                  nextCell,
                  rowNum);
            }
         }
      }
   }
```

Рисунок 4.1 – Реалізація першого кроку на мові C#

2. Проходимо по тій же рядку і випадковим чином об'єднуємо комірки з рядка вище з поточним рядком (з деякими обмеженнями) (рисунок 4.2).

```
private void CreateVerticalConnections( W4Maze maze, int rowNum)
  {
    bool removeVertical = false;
    bool isAddedVertical = false;
    for (int i = 0; i < maze.ColumnCount - 1; i++)
    {
      W4Cell cell = maze.GetCell(i, rowNum);
      W4Cell nextCell = maze.GetCell(i + 1, rowNum);
      W4Cell topCell = maze.GetCell(i, rowNum + 1);
      if (cell.Set != nextCell.Set)
      {
        if (!isAddedVertical)
          {
            RemoveVerticalWall(cell, topCell);
          }
          isAddedVertical = false;
        }
        else
        {
          removeVertical = Random.Range(0, 2) > 0;
          if (removeVertical)
        {
          RemoveVerticalWall(cell, topCell);
          isAddedVertical = true;
        }
      }
    }
    CheckLastVertical(maze, rowNum, isAddedVertical);
  }
```

Рисунок 4.2 – Реалізація другого кроку на мові C#

3. Обробляємо останній рядок об'єднуючи осередки з різних множин (рисунок 4.3).

```
private void CreateLastRow(W4Maze maze)
  {
    int y = maze.RowCount - 1;
    for(int i = 0; i < maze.ColumnCount - 1; i++)
    {
      var cell = maze.GetCell(i, y);
      var nextCell = maze.GetCell(i + 1, y);
      if(cell.Set != nextCell.Set)
      {
        RemoveHorizonWallBetweenCells(maze, cell, nextCell, y);
      }
    }
  }
```

Рисунок 4.3 – Реалізація третього кроку на мові C#

Лабіринт потрібно в чомусь зберігати. У цьому рішенні є 2 структури для зберігання лабіринтів. W4Maze – який по суті являє з себе масив осередків 16-ти типів за кількістю піднятих стін. Сам осередок W4Cell зберігає в собі просто 4 bool поля, які говорять про те, які стіни підняті. Ця структура зручна для генерації лабіринту алгоритмом Вілсона і зручна для серіалізації.

Але з іншого боку вона абсолютно незручна для генерації стін. ідея в тому, що у лабіринту стіни повинні володіти товщиною, для того щоб реалізувати подібний алгоритм зручні саме графові структури. Для цього була заведена структура MazeGraph і реалізовано перетворення з W4Maze у нього. Крім того, для зручності, в ньому було створено 2 списки – шляхів і стін. Після написавши візуалізацію за допомогою Gizmos був отриманий настапний лабіринт (рисунок 4.4). Жовтий – це стіни, а зелений – шляхи.

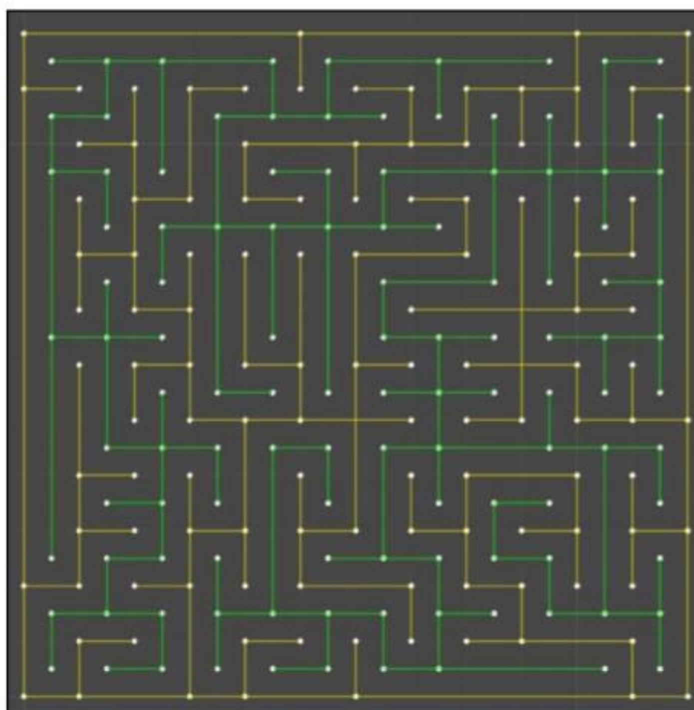

Рисунок 4.4 – Підсумок написаної візуалізації

Залишилося тільки додати клас для генерації площин, але в цілому можна використовувати клас Unity Quad, але в будь-якому випадком необхідно

перерахувати uv координати для того, щоб тайлін матеріалу на всіх сторонах був однаковий і можна було використовувати один матеріал (рисунок 4.5).

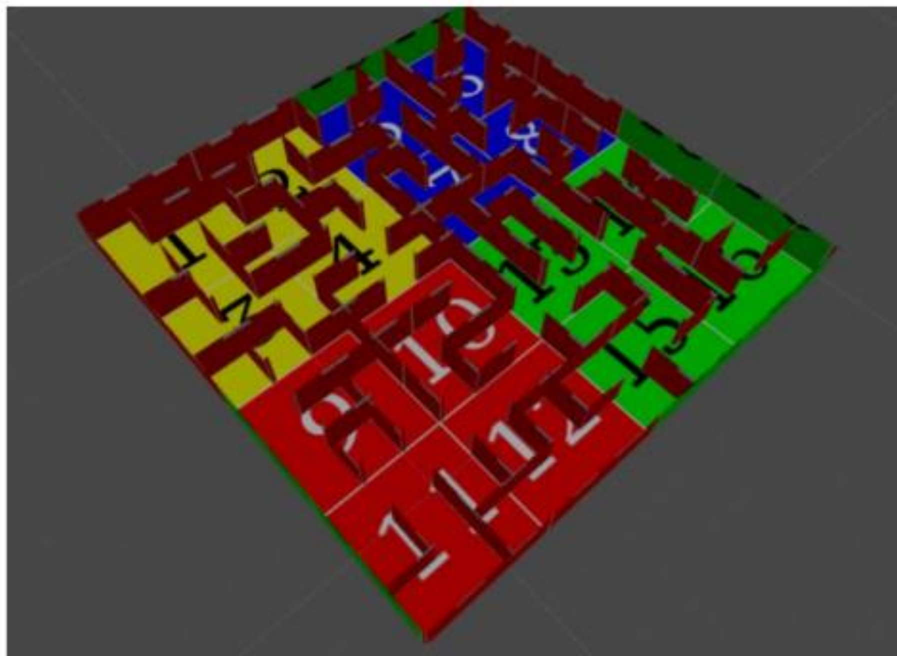

Рисунок 4.5 – Згенерований лабіринт

Тепер необхідно показати найпростіший приклад можливості пост процесингу на прикладі автоматичної розстановки світла. Для цього був створив клас LightPlacer і використаний лабіринт W4Maze. Алгоритм пост обробки досить простий. Необхідно задати рівні освітленості, перевести тип осередку в int і прийняти рішення, ставити джерело світла чи ні (рисунки 4.6-4.7).

```
public GameObject SetUpLights( W4Maze maze, float height)
  {
    GameObject lights = new GameObject();
    for(int i = 0; i < maze.ColumnCount; i++)
    {
      for(int j = 0; j < maze.RowCount; j++)
      {
        var cell = maze.GetCell(i, j);
        if(cell.ToInt() < (int) _LightLevel)
        {
          var lightGo = CreatePointLight();
          lightGo.transform.position = new Vector3( i , height * 0.9f, j);
        }
      }
    }
    return lights;
  }
  public enum LightLevel : byte
  {
    Low = 3,
    Medium = 5,
    High = 10,
    VeryHigh = 16
  }
```

Рисунок 4.6 – Приклад пост-процесингу на мові C#

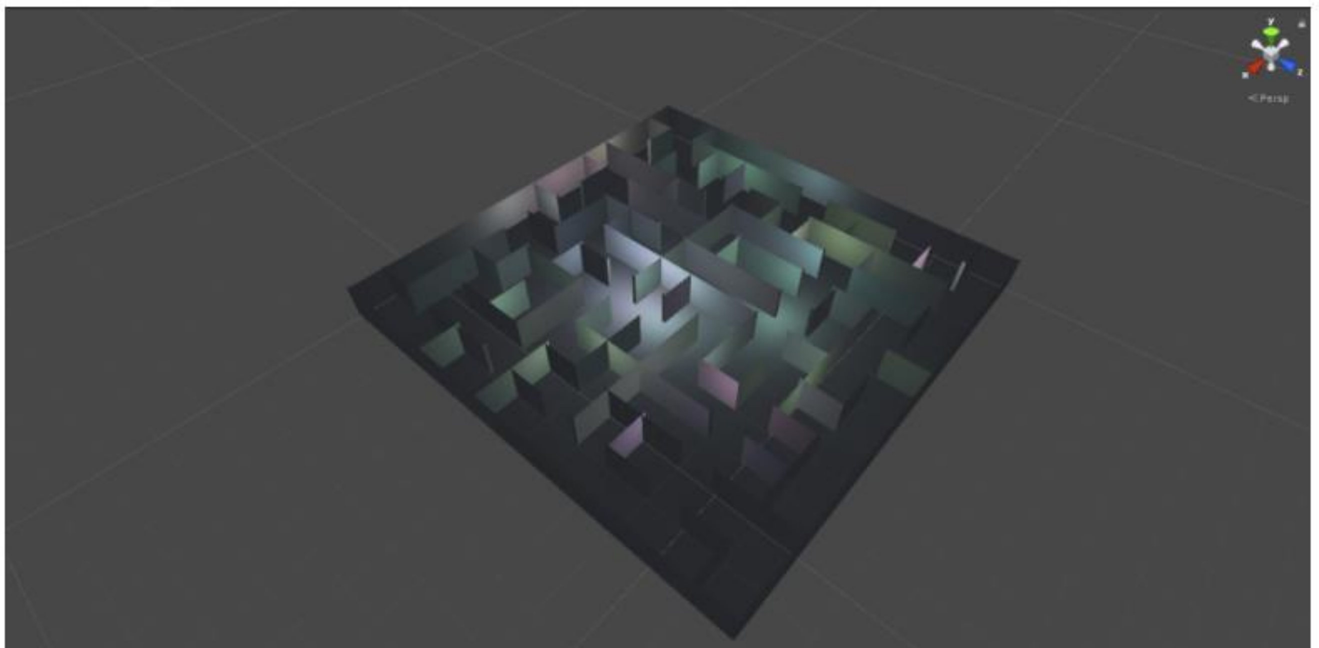

Рисунок 4.7 – Фінальна версія лабіринту

4.3 Експериментальна перевірка модифікованого методу пошуку шляхів в лабіринті

В ході обчислювального експерименту були поставлені наступні цілі:

1. З'ясувати, від яких вхідних параметрів залежить кількість одержуваних проходів в лабіринті.

2. Знайти оптимальне кількість мурах для мурашиного алгоритму при різних вхідних даних.

3. Оцінити ефективність роботи алгоритмів видалення шляхів.

Вхідними даними є:

1. Розмірність лабіринту.

2. Число вирізів, в алгоритмі модифікованого методу пошуку шляхів.

3. Інтервал довжин шляхів для алгоритму видалення по довжині.

У таблиці 4.1 показано, що і кількість шляхів в лабіринті, і оптимальне число мурах залежить від вирізів, зроблених на 4 кроці модифікованого алгоритму Вілсона, і не залежить від розмірності лабіринтів. Для знаходження закономірностей було взято 3 лабіринти різних розмірностей і з різною кількістю додаткових вирізів.

Таблиця 4.1 – Знаходження оптимальної кількості мурах

Розмірність лабіринту	Кількість вирізів	Кількість отриманих шляхів	№ останнього неповторюваного шляху	Можлива кількість мурах
19 × 19	4	4	12	100
		4	7	
		8	20	
		8	49	
		25	25	
	6	14	40	200
		14	54	
		14	20	
		14	38	
		10	32	
	8	48	421	700
		28	87	
		40	419	
		32	158	
		64	499	
55 × 55	4	8	9	100
		8	20	
		8	23	
		8	22	
		4	13	
	6	12	41	200
		32	98	
		14	65	
		28	72	
		12	58	

Кінець таблиці 4.1

<table>
<tr><th>Розмірність лабіринту</th><th>Кількість вирізів</th><th>Кількість отриманих шляхів</th><th>№ останнього неповторюваного шляху</th><th>Можлива кількість мурах</th></tr>
<tr><td rowspan="5">55 × 55</td><td rowspan="5">8</td><td>28</td><td>259</td><td rowspan="5">700</td></tr>
<tr><td>40</td><td>168</td></tr>
<tr><td>55</td><td>432</td></tr>
<tr><td>48</td><td>240</td></tr>
<tr><td>20</td><td>118</td></tr>
<tr><td rowspan="15">179 × 179</td><td rowspan="5">4</td><td>4</td><td>7</td><td rowspan="5">100</td></tr>
<tr><td>8</td><td>39</td></tr>
<tr><td>7</td><td>23</td></tr>
<tr><td>8</td><td>14</td></tr>
<tr><td>4</td><td>16</td></tr>
<tr><td rowspan="5">6</td><td>12</td><td>45</td><td rowspan="5">200</td></tr>
<tr><td>22</td><td>125</td></tr>
<tr><td>24</td><td>63</td></tr>
<tr><td>12</td><td>33</td></tr>
<tr><td>20</td><td>89</td></tr>
<tr><td rowspan="5">8</td><td>30</td><td>250</td><td rowspan="5">700</td></tr>
<tr><td>56</td><td>179</td></tr>
<tr><td>21</td><td>354</td></tr>
<tr><td>78</td><td>181</td></tr>
<tr><td>60</td><td>550</td></tr>
</table>

Число вирізів дорівнює 4, 6 і 8, число мурах задано завідомо велике (більше 2000). Якщо подивитися на всі рядки з кількістю вирізів рівним 4, то можна побачити, що при всіх трьох розмірностях, кількість отриманих шляхів лежить в інтервалі [3, 8]. Максимальний номер останнього унікального шляху можна

побачити при розмірності 19 × 19 - він дорівнює 49, крім того, всі номери також потрапляють в інтервал [7, 49] для всіх розмірностей.

Аналогічно, якщо розглянути число додаткових вирізів 4 і 6, можна виділити деякі інтервали кількості шляхів і номера останнього унікального шляху, також не залежать від розмірності лабіринту: в разі 6 вирізів це число 154, а в разі 8 вирізів– 635. Звідси можна зробити висновок, що число отриманих шляхів і оптимальна кількість мурах не залежить від розмірності лабіринту, але залежать від кількості додаткових вирізів.

Також, виходячи з отриманих результатів, можна зробити висновок про те, що для 4 вирізів необхідно взяти 49 мурах, для 6 вирізів – 154 мурашки і 635 мурах для 8 вирізів. Однак рекомендується брати певний «запас», наприклад, на 50 мурах більше. Таким чином, отримуємо 99, 204 і 685 мурах для 4, 6 і 8 вирізів відповідно. У таблиці 4.1 число мурах округлено до 100, 200 і 700. Варто відзначити, що при збільшенні числа вирізів з 6 до 8, можливу кількість мурах зростає більш ніж в три рази, що збільшує час пошуку всіх шляхів.

У таблиці 4.2 наведені результати роботи алгоритму видалення по довжині шляху. Було побудовано 15 лабіринтів для трьох розмірностей, після чого для кожного були введені різні інтервали довжин. При оцінці ефективності роботи алгоритму було з'ясовано, що майже в половині випадків вдалося видалити 50% і більше шляхів. Наприклад, в лабіринті розмірності 19 × 19 при інтервалі [38, 45] алгоритм видалив абсолютно всі шляхи, довжини яких опинилися поза введеного інтервалу, але також в деяких випадках алгоритм не вилучив жодного шляху, наприклад, при розмірності 55 × 55 і інтервалі [240, 290] . Таким чином, при грамотно підібраному інтервалі довжин ефективність алгоритму становить понад 50%.

Таблиця 4.2 – Результати роботи алгоритму видалення по довжині шляху

Розмірність лабіринту	Кількість шляхів	Інтервал довжин шляхів	Введений інтервал	Кількість шляхів, що підлягають видаленню	Кількість віддалених шляхів
19x19	9	[38,74]	[38,55]	5	3
			[38,45]	6	6
			[50,65]	6	3
			[55,74]	4	3
			[65,74]	7	6
55x55	21	[162,350]	[162,190]	14	9
			[162,250]	11	6
			[200,300]	10	0
			[240,290]	14	0
			[300,350]	18	7
179x179	28	[590,1698]	[590,1000]	22	8
			[590,1400]	6	0
			[950,1300]	22	8
			[1200,1698]	10	0
			[1400,1698]	22	7

Таблиця 4.3 показує результати роботи алгоритму видалення схожих шляхів при k = 80. Для трьох розмірностей було побудовано по 5 різних лабіринтів, до яких був застосований алгоритм видалення схожих шляхів. У таблиці 4.3 найбільший інтерес представляють три останніх стовпчика.

Таблиця 4.3 – Результати роботи алгоритму видалення схожих шляхів

Розмірність лабіринту	Кількість шляхів	Кількість шляхів, що підлягають видаленню	Кількість віддалених шляхів	Кількість шляхів-«жертв»
19 × 19	24	4	0	0
	22	4	0	0
	14	1	0	0
	14	6	4	3
	14	2	0	0
55 × 55	14	3	2	1
	24	8	4	4
	10	2	2	1
	21	10	6	3
	12	3	1	1
179 × 179	7	2	1	1
	14	2	0	0
	24	12	8	4
	21	7	4	5
	8	6	4	4

В трьох перших рядках можна побачити, що з усіх шляхів, які необхідно видалити, алгоритм не вилучив жодного. У четвертій же рядку інша ситуація: замість належних 6 шляхів алгоритм зміг видалити тільки 4, однак навіть для цього довелося видалити 3 шляхи-жертви. І лише в одному з випадків розмірності 55 × 55 з кількістю шляхів, різних 10, алгоритм видалив все 2 необхідних шляху, але до того ж прибрав один шлях-жертву.

ВИСНОВКИ

У даній атестаційній роботі магістра виконано наступні роботи у межах дослідження обраної теми.

Виконано загальний огляд і аналіз сучасного стану розглянутої проблеми. Зокрема приділено увагу особливостям процесу розробки ігрових додатків, загальному огляду та класифікації видів лабіринту та огляду сфери використання лабіринтів у реальному житті.

Проведено дослідження існуючих методів генерації лабіринтів серед яких були розглянуті алгоритми двійкового дерева, Прима, Краскала, Олдоса-Бродера, Вілсона, Еллера, Sidewinder. Основну увагу приділено порівнянню даних алгоритмів, за результатами якого був розроблений модифікований метод генерації лабіринтів основою якого є алгоритм Вілсона.

Проведено дослідження існуючих методів проходження лабіринтів серед яких були розглянуті алгоритм A* (так званий A-стар), хвильової алгоритм (алгоритм Лі), його восьминаправлена модифікація, променевий алгоритм, алгоритм Дейкстри, пошук в глибину та в ширину, алгорити правої та лівої руки. Основну увагу приділено порівнянню даних алгоритмів, за результатами якого був розроблений модифікований метод проходження лабіринтів основою якого є мурашиний алгоритм.

Проведено апробацію запропонованих рішень у ході застосування методів генерації та проходження лабіринтів. Проведено загальний огляд особливостей реалізації у середовищі Unity 3D, виконана програмна реалізація генерації лабіринту у середовищі Unity 3D.

Проведено обчислювальний експеримент, в ході якого вдалося встановити залежність кількості шляхів від параметрів генерації лабіринту, оптимальна кількість мурах для знаходження всіх шляхів в лабіринті. Для алгоритму редагування шляхів в лабіринті були протестовані і досліджені на ефективність, були виявлені їх недоліки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Методичні вказівки до організації виконання та захисту атестаційної роботи на здобуття другого (магістерського) рівня вищої освіти для студентів усіх форм навчання спеціальності 122 – «Комп'ютерні науки» за освітньою програмою «Системне проектування» / Упорядники: І.В. Гребеннік, В.Г. Іванов, Н.І. Калита, А.І. Коваленко, Д.Е.Ситніков, І.А. Урняєва – Харків: ХНУРЕ, 2018. – 52 с.

2. ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура і правила оформлювання. . – Чинний від 22.06.2015. – Київ: ДП «УкрНДНЦ», 2016. – 31 с.Федотова, Д. Э. CASE-технологии: Практикум [Текст] / Д. Э. Федотова, Ю. Д. Семенов, К. Н. Чижик — М. : "Горячая Линия – Телеком", 2005. — 160 с.

3. App2Top. Этапы разработки игры для мобильных платформ. URL: https://app2top.ru/columns/e-tapy-razrabotki-igry-dlya-mobil-ny-h-p-40118.html (дата звернення: 17.02.2020).

4. Сахнов К. Семь этапов создания игры: от концепта до релиза. URL: https://habrahabr.ru/company/miip/blog/308286/ (дата звернення: 17.03.2020).

5. Википедия. Лабиринт. URL: http://ru.wikipedia.org/wiki/Лабиринт (дата звернення: 28.02.2020).

6. Германн, Керн. Лабиринт: основные принципы, гипотезы, интерпретации // Керн Г. Лабиринты мира. – СПб.: Азбука-классика, 2007 – с. 7-33/

7. Lee, C.Y., An Algorithm for Path Connections and Its Applications, IRE Transactions on Electronic Computers, vol. EC-10, number 2, 1961 – P. 364-365.

8. Герасимов В.Н., Михайлов Б.Б Решение задачи управления движением мобильного робота при наличии динамических препятствий // Вестник МГТУ им. Н.Э. Баумана. Приборостроение. Спецвыпуск "Робототехнические системы". - 2012. - № 6. - С. 83-92.

9. Habr. Классические алгоритмы генерации лабиринтов. Часть 1: вступление URL: https://habr.com/ru/post/320140/ (дата звернення: 10.03.2020).

10. The Buckblog. Maze Generation: Binary Tree algorithm URL: http://weblog.jamisbuck.org/2011/2/1/maze-generation-binary-tree-algorithm.html (дата звернення: 10.03.2020).

11. Wikipedia. Алгоритм Еллера генерації лабіринтів URL: https://uk.wikipedia.org/wiki/Алгоритм_Еллера_генерації_лабіринтів (дата звернення: 10.03.2020).

12. Лабиринты: классификация, генерирование, поиск решений URL: https://ai-news.ru/2019/03/labirinty_klassifikaciya_generirovanie_poisk_reshenij.html (дата звернення: 10.03.2020).

13. The Buckblog. Maze Generation: Sidewinder algorithm URL: http://weblog.jamisbuck.org/2011/2/3/maze-generation-sidewinder-algorithm.html (дата звернення: 10.03.2020).

14. David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pages 296–303. ACM, 1996.

15. The Buckblog. Maze Generation: Aldous-Broder algorithm URL: http://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm.html (дата звернення: 10.03.2020).

16. The Buckblog. Maze Generation: Wilson's algorithm URL: http://weblog.jamisbuck.org/2011/1/20/maze-generation-wilson-s-algorithm.html (дата звернення: 10.03.2020).

17. Maze generations: Algorithms and Visualizations. URL: https://medium.com/analytics-vidhya/maze-generations-algorithms-and-visualizations-9f5e88a3ae37 (дата звернення: 10.03.2020).

18. The Buckblog. Maze Generation: Kruskal's Algorithm. URL: http://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm.html (дата звернення: 10.03.2020).

19. M Tim Jones. Artificial Intelligence: A Systems Approach: A Systems Approach. Jones & Bartlett Learning, 2015.

20. Визуальный язык ДРАКОН. Алгоритм A*. URL: https://drakon.su/algoritmy/a-star (дата звернення: 20.03.2020).

21. Suvitruf's Blog :: Gamedev suffering. Обход препятствий: волновой алгоритм (Алгоритм Ли). URL: https://suvitruf.ru/2012/05/13/1176/volnovoj-algoritm-algoritm-li/ (дата звернення: 20.03.2020).

22. И. Соловьёв. Выделение контуров зданий и распознавание служебных символов для трехмерной реконструкции объектов городской обстановки по топографическому плану. URL: https://www.graphicon.ru/html/2013/papers/290-293.pdf (дата звернення: 20.03.2020).

23. Studfiles. Модификация волнового алгоритма. URL: https://studfile.net/preview/1382785/page:25/ (дата звернення: 20.03.2020).

24. Habr. Алгоритм Дейкстры. Поиск оптимальных маршрутов на графе. URL: https://habr.com/ru/post/111361/ (дата звернення: 20.03.2020).

25. Басараб М.А., Домрачева А.Б., Купляков В.М. Алгоритмы решения задачи быстрого поиска пути на географических картах. Инженерный журнал: наука и инновации, 2013, вып. № 11. URL: http://engjournal.ru/catalog/it/hidden/1054.html

26. Habr. Генерация и решение лабиринта с помощью метода поиска в глубину по графу. URL: https://habr.com/ru/post/262345/ (дата звернення: 20.03.2020).

27. Studfiles. Поиск в ширину. URL: https://studfile.net/preview/1399243/ (дата звернення: 20.03.2020).

28. Князев Б.А., Черкасский В.С. Начала обработки экспериментальных данных. Измерительный практикум: методическое пособие. Новосибирск: Изд-во Новосиб. ун-та, 2011.

29. . Краснов Е.С. Методика оценки алгоритмов поиска пути в лабиринте для выбора мобильным роботом стратегии перемещения // Известия тульского государственного университета. Технические науки, выпуск 11, часть 2. С. 179–187.

30. Штовба С. Д. Муравьиные алгоритмы / С. Д. Штовба // Exponenta Pro. – 2003. – №4. – С. 70-75.

31. Will Goldstone. «Unity Game Development Essentials» – October 2009 – 316 p.

32. Кристиан Нейгел. «C# 5.0 и платформа .NET 4.5 для профессионалов – Professional C# 5.0 and .NET 4.5.» – М.: «Диалектика», 2013. – 1440 с. – ISBN 978-5-8459-1850-5

33. Хокинг Дж. «Unity в действии. Мультиплатформенная разработка на C#» – СПб.: Питер, 2016. – 336 с.: ил. – (Серия «Для профессионалов»)

34. 24-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті». Зб. матеріалів форуму. Т. 6. – Харків: ХНУРЕ. 2020. – 422 с.