УДК 004.4: 004.4`6 DOI https://doi.org/10.35546/kntu2078-4481.2024.3.29

Г. В. МАРЧУК
старший викладач кафедри комп'ютерних наук
Державний університет «Житомирська політехніка»
ORCID: 0000-0003-2954-1057

М. С. ГРАФ
доктор філософії з комп'ютерних наук,
завідувач кафедри комп'ютерних наук
Державний університет «Житомирська політехніка»
ORCID: 0000-0003-4873-548X

В. Л. ЛЕВКІВСЬКИЙ
доктор філософії з інженерії програмного забезпечення,
доцент кафедри комп'ютерних наук
Державний університет «Житомирська політехніка»
ORCID: 0000-0002-1643-0895

Ю. В. ВЕНГЛОВСЬКА
магістрантка
Державний університет «Житомирська політехніка»
ORCID: 0009-0003-1291-9284

АНАЛІЗ ТА ПОРІВНЯННЯ ІСНУЮЧИХ МЕТОДІВ ГЕНЕРАЦІЇ ЛАБІРИНТІВ В КОМП'ЮТЕРНИХ ІГРАХ

В епоху інформаційного суспільства, де візуалізація ігрових та симуляційних просторів набуває все більшого значення, розробка алгоритмів для генерування лабіринтів стає все більш актуальною. Інтерактивні ігри, віртуальна реальність, навчальні платформи та інші сфери потребують генерації лабіринтів різної складності та конфігурації, щоб забезпечити різноманіття та захоплюючий досвід для гравців та користувачів. Традиційні методи генерації лабіринтів, такі як алгоритм Вілсона або алгоритм Прима, можуть бути обмежені у своїй гнучкості та здатності створювати лабіринти з заданими характеристиками. Процедурні методи генерування лабіринтів пропонують більш гнучкий та потужний підхід. Ці методи дозволяють генерувати лабіринти з різними параметрами, такими як розмір, складність, тип розгалуження, наявність тупиків та інших елементів. Завдяки цьому процедурні методи стають все більш популярними для генерації лабіринтів у широкому спектрі застосувань. Це дослідження має на меті порівняти різні алгоритми генерації лабіринтів. В роботі досліджується та порівнюється п'ять алгоритмів: Recursive Backtracker, Kruskal's Algorithm, Aldous-Broder Algorithm, Hunt-and-Kill Algorithm та Binary Tree Algorith. Існує необхідність визначення ефективності різних методів з точки зору структурної складності та варіативності створених лабіринтів. Аналіз впливу різних методів на обчислювальні ресурси та час генерації також є ключовим аспектом, особливо у вимогливих застосунках, таких як відеоігри чи симуляції. Такий аналіз дозволить визначити оптимальні та ефективні підходи до генерації лабіринтів для різних застосувань. Результати дослідження можуть бути корисними для розробників комп'ютерних ігор, для дослідників штучного інтелекту, які хочуть розробити алгоритми для вирішення задач на маршрутизацію, тощо. Це дослідження допоможе розробникам краще зрозуміти можливості та обмеження кожного алгоритму, щоб зробити свідомий вибір.

Ключові слова: *алгоритм, лабіринт, Unity, гра, генерація лабіринту.*

G. V. MARCHUK
Senior Lecturer at the Department of Computer Sciences
Zhytomyr Polytechnic State University
ORCID: 0000-0003-2954-1057

M. S. GRAF
PhD in Computer Sciences,
Associate Professor at the Department of Computer Sciences
Zhytomyr Polytechnic State University
ORCID: 0000-0003-4873-548X

V. L. LEVKIVSKYI
PhD in Software Engineering,
Associate Professor at the Department of Computer Sciences
Zhytomyr Polytechnic State University
ORCID: 0000-0002-1643-0895

YU. V. VENHLOVSKA
Master's Student
Zhytomyr Polytechnic State University
ORCID: 0009-0003-1291-9284

ANALYSIS AND COMPARISON OF EXISTING MAZE GENERATION METHODS IN COMPUTER GAMES

In the era of the information society, where the visualization of game and simulation spaces is becoming increasingly important, the development of algorithms for generating mazes is becoming more and more relevant. Interactive games, virtual reality, educational platforms, and other areas require the generation of mazes of various complexities and configurations to provide variety and an engaging experience for players and users. Traditional maze generation methods, such as the Wilson algorithm or the Prim's algorithm, may be limited in their flexibility and ability to create mazes with specified characteristics. Procedural maze generation methods offer a more flexible and powerful approach. These methods allow generating mazes with different parameters, such as size, complexity, branching type, presence of dead ends, and other elements. As a result, procedural methods are becoming increasingly popular for maze generation in a wide range of applications.

This research aims to compare different maze generation algorithms. The paper investigates and compares five algorithms: Recursive Backtracker, Kruskal's Algorithm, Aldous-Broder Algorithm, Hunt-and-Kill Algorithm, and Binary Tree Algorithm. There is a need to determine the efficiency of different methods in terms of structural complexity and variability of the generated mazes. Analyzing the impact of different methods on computational resources and generation time is also a key aspect, especially in demanding applications such as video games or simulations. Such analysis will help identify optimal and efficient approaches to maze generation for different applications.The research results may be useful for game developers, for researchers in artificial intelligence who want to develop algorithms for solving routing problems, and so on. This research will help developers better understand the capabilities and limitations of each algorithm to make an informed choice.

Key words: *algorithm, maze, Unity, game, maze generation.*

Постановка проблеми

У сучасному інформаційному суспільстві, де візуалізація ігрових та симуляційних середовищ відіграє важливу роль, розробка процедурних методів генерації лабіринтів стає актуальною задачею. Інтерактивні ігри, віртуальна реальність, навчальні платформи та інші області вимагають генерації різних типів лабіринтів для забезпечення різноманітності та виклику гравців або користувачів.

Аналіз останніх досліджень і публікацій

На сьогоднішній день лабіринт вже не просто головоломка. У галузі комп'ютерних ігор він може служити основною структурою для рівнів, у робототехніці – як платформу для демонстрації навчальних здібностей роботів. В області архітектури його візерунок можна використовувати для прикраси будівлі. Дослідники, які використовують лабіринт у різних галузях, можуть мати різні цілі та потребувати різних властивостей лабіринту. Методів генерації лабіринтів створено та проаналізовано безліч (алгоритм Прима, алгоритм Крускала, бінарне дерево тощо), тож часто виникають труднощі з вибором. Дослідження [1] зосереджено на створенні алгоритму, що вибирає найкращий метод генерації для введених бажаних властивостей проекту. Інноваційний підхід, який успішно створили розробники, добре справляється із задачею, але все ще є деякі випадки, коли складно обрати правильний варіант.

Лабіринти використовуються в розважальних іграх, і це робить їх хорошим прикладом для інтегрування автоматизації генерації [2]. Вони забезпечують створення різних непередбачуваних ігрових середовищ як для аркадних ігор, такі як Ms. Pac-Man [3], так і для 3D програм віртуальної реальності [4], які можуть максимально збільшити занурення, коли вони відчувають оточення лабіринту.

Від текстових пригод Zork до коридорів Doom і міських вулиць Grand Theft Auto, лабіринт часто використовувався як простір для захоплення та заплутування гравців у їхній навігація ігровими світами [5]. Шлях у відеоіграх пов'язаний зі знаходженням об'єктів в ігровому світі. Шлях не можна розглядати окремо від цих об'єктів і їх використання, оскільки реакція гравця на ці об'єкти є одним із способів формування шляху.

Незважаючи на те, що екранне відображення в деяких аспектах допускає «подвійність лабіринту», питання про те, чи пропонують відеоігри і дизайн, і досвід одночасно залишається відкритим для інтерпретації. Ігри

можна обговорювати через відкриття ігрового простору. Граючи в ігровому світі і/та відчуваючи його, гравець багато в чому розкриває більшу частину його дизайну. Це двосторонній процес, головним у якому є гра та дослідження, а початок розуміння дизайну допомагає у подальшому вирішенні головоломок [6].

Тепер зрозуміло, наприклад, що шляхи реального лабіринту, і шляхи ігрового простору часто схожі. Проте видно, що комп'ютерні технології розробляють шляхи-лабіринти по-новому та виразно.

З новими підходами до змішування генерації процедурного контенту з динамічним налаштуванням складності, можна не тільки розробити гру з наявністю нескінченного рівня, використовуючи найменшу кількість дискового простору, але також можна адаптувати його до потужності гравця, гарантуючи, що кожен рівень унікальний не лише за дизайном, але й за досвідом гравця [7-9].

Разом з тим практичне значення таких додатків полягає не тільки у розвагах. Дослідження підтверджують нещодавні припущення про те, що ігри з лабіринтами є багатообіцяючими для фіксації змін у зорово-рухових, зорово-конструктивних і виконавчих функціях, пов'язаних зі старінням і нейродегенерацією. Використовуючи попередньо згенеровані рівні складності головоломки з цього дослідження, можна краще виявляти тонкі зміни в виконавчих і моторних функціях у ширшому діапазоні складності головоломок, схожих на лабіринт, і уникнути ефектів як підлоги, так і стелі. Це дослідження є перехресним, і учасники проходять певний лабіринт лише один раз і вперше [10].

Формулювання мети дослідження

Дослідити різні методи генерації лабіринтів, визначити їхню структурну складність та різноманітність результатів. Оцінка ресурсозатратності кожного алгоритму в термінах обчислювального часу та використання пам'яті. Дослідження ефективності різних методів в контексті відеоігор та симуляцій, де важлива якість та економія ресурсів.

Для досягнення мети дослідження необхідно провести дослідження алгоритмів Recursive Backtracker, Kruskal's Algorithm, Aldous-Broder Algorithm, Hunt-and-Kill Algorithm та Binary Tree Algorithm за наступними категоріями:

- ефективність та складність;
- оптимальність використання для Unity;
- швидкість генерації.

Викладення основного матеріалу дослідження

Лабіринти в іграх є популярним елементом геймплею, який викликає інтригу, викликає емоції та сприяє розвитку логічного мислення у гравців. Зазвичай лабіринти складаються з вузьких коридорів та перешкод, що утворюють заплутану мережу шляхів. Гравець зазвичай має завдання пройти через лабіринт, знаходячи правильний шлях до виходу або до цільового об'єкта, уникаючи пасток, монстрів або інших небезпек.

Ідеальний лабіринт – це лабіринт, в якому відсутні петлі, і всі вузли доступні. Таким чином, гравець може перейти з одного вузла до будь-якого іншого в ідеальному лабіринті, і є лише один маршрут між цими вузлами. Побудова, або ж генерація, можлива за рахунок використання одного з алгоритмів.

Recursive backtracker. Процес починається з будь-якого вузла сітки лабіринту та переміщується до одного із сусідніх вузлів, який ще не було відвідано. У сітці кожен вузол має не більше чотирьох сусідніх вузлів, з'єднаних ребрами. Після того, як процес переходить до поточного вузла, він рекурсивно переміщується до невідвіданого сусіднього вузла, поки не відвідає вузол, який відвідував раніше. Коли вузол u не має невідвіданих сусідів, він повертається до вузла v, який відвідував безпосередньо перед вузлом u, і ребро між u і v стає ребром охоплюючого дерева. Потім процес рекурсивно переміщується до інших невідвіданих сусідніх вузлів від вузла v, доки він не відвідає вузол, який відвідував раніше. Процес триває, доки не будуть відвідані всі вузли на графі [1].

Для реалізації алгоритму у Unity за допомогою мови C#, можна спочатку згенерувати сітку зі стінами, а потім видаляти проходи. Для цього слугує метод RemoveWalls, який в якості аргументів приймає масив клітинок та розміри сітки. Основна робота відбувається зі списком сусідніх клітинок:

```
List<RecursiveBacktrackerCell> unvisitedNeighbours = new List<RecursiveBacktrackerCell>();
```

Перевірка сусідніх клітинок на те, чи є серед них не відвідана клітинка:

```
//check all 4 neighbours
if (x > 0 && !maze[x - 1, y].visited)
    unvisitedNeighbours.Add(maze[x - 1, y]);
if (y > 0 && !maze[x, y - 1].visited)
    unvisitedNeighbours.Add(maze[x, y - 1]);
if (x < width - 2 && !maze[x + 1, y].visited)
    unvisitedNeighbours.Add(maze[x + 1, y]);
if (y < height - 2 && !maze[x, y + 1].visited)
    unvisitedNeighbours.Add(maze[x, y + 1]);
```

Якщо така знаходиться, то обирається одна навмання та шлях переходить до неї. Інакше повертається до минулого значення зі стеку:

```
if (unvisitedNeighbours.Count > 0){
    RecursiveBacktrackerCell chosen =
        unvisitedNeighbours[Random.Range(0, unvisitedNeighbours.Count)];
    RemoveWall(current, chosen);
    chosen.visited = true;
    current = chosen;
    stack.Push(current);
    current.DistanceFromStart = stack.Count;
}
else{
    current = stack.Pop();
}
```

На рисунку 1 представлено лабіринт 10x10, який згенеровано за алгоритмом Recursive backtracker.

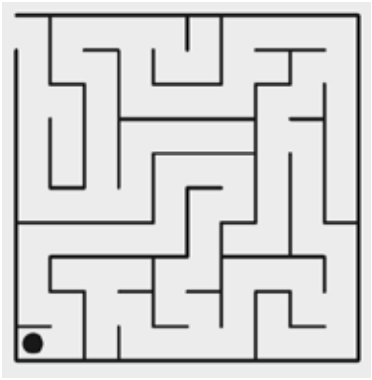

Рис. 1. Лабіринт 10x10, згенерований за алгоритмом Recursive backtracker

Kruskal's algorithm. На початку кожен вузол міститься у власному наборі. Припустимо, що є група всіх ребер сітки. Грань вибирається випадковим чином і видаляється в групі. Якщо обидва суміжні вузли вибраного ребра знаходяться в різних наборах, ребро стає ребром охоплюючого дерева, і ці набори об'єднуються в один набір. Якщо обидва суміжні вузли вибраного ребра знаходяться в однакових наборах, оскільки ребро збирається утворити петлю, ребро просто видаляється з групи ребер. Цей алгоритм зупиняється, коли всі вузли знаходяться в одному наборі [1].

Для реалізації методу у Unity можна написати метод RemoveWalls, який прорізатиме потрібні проходи. Спочатку кожній клітинці потрібно дати унікальній ідентифікатор:

```
//give unique id
int tmp = 1;
for (int x = 0; x < width; x++){
    for (int y = 0; y < height; y++){
        maze[x, y].id = tmp;
        tmp++;
    }
}
```

Далі потрібно оголосити словники для лівих та нижніх стін (у клітинки є лише дві стіни, щоб не виникало накладок):

```
List<KruskalCell> wallsLeft = new List<KruskalCell>();
List<KruskalCell> wallsBottom = new List<KruskalCell>();
```

Для видалення ж стіни із словника з лівими стінами можна використати код:

```
int randomId = Random.Range(0, wallsLeft.Count);
KruskalCell cell = wallsLeft[randomId];
wallsLeft.RemoveAt(randomId);
if (cell.X != 0 && cell.X != width-1) //if the cell is not in first or last column{
 if (cell.id != maze[cell.X - 1, cell.Y].id)    //if id differ, delete the wall
    {
     cell.wallLeft = false;

     //change id
     int idToChange = cell.id;
     foreach (KruskalCell mazeCell in maze){
       if (mazeCell.id == idToChange){
          mazeCell.id = maze[cell.X - 1, cell.Y].id;
       }
     }
   }
 }
```

На рисунку 2 представлено лабіринт 10x10, який згенеровано за алгоритмом Kruskal's algorithm.

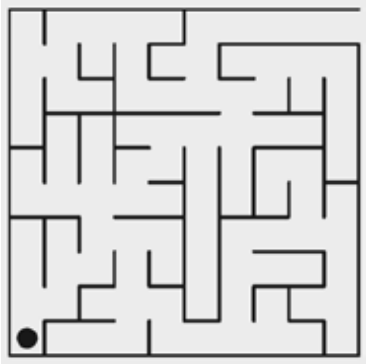

Рис. 2. Лабіринт 10x10, згенерований за алгоритмом Kruskal's algorithm

Aldous-Broder algorithm. Алгоритм Aldous-Broder – це імовірнісний алгоритм для генерації рівномірного кістяка на неорієнтованому графі G = (V, E). Алгоритм починається з будь-якого вузла. Потім випадковим чином відвідує одного зі своїх сусідів, і наступна дія визначається на основі статусу відвідування сусіда. Якщо сусідній вузол ще не відвідано, цей вузол і попередній вузол з'єднані ребром остовного дерева. Якщо сусідній вузол було відвідано, процес просто переходить до наступного сусіда. Потім він знову переходить до випадкового сусіда. Процес триває, доки не будуть відвідані всі вузли. Зауважте, що є випадки, коли цей алгоритм ніколи не закінчується, оскільки останні кілька вузлів не знайдені випадковим чином. Коли всі вузли відвідано, процес завершується створенням остовного дерева [1].

Основним методом в коді можна зробити RemoveWalls, що видалятиме потрібні стіни. Спочатку обирається випадкова клітинка та позначається як відвідана:

```
        //choose random cell and mark it as visited
        int cellX = Random.Range(1, width - 1);
        int cellY = Random.Range(1, height - 1);
        AldousBroderCell currentCell = maze[cellX, cellY];
        currentCell.visited = true;
```

Далі у циклі випадковим чином обирається один із чотирьох сусідів. Якщо він не був відвіданий, то стінка між ним та попередньою клітинкою видаляється:

```
if (!cellNeighbour.visited){
                        currentCell.wallLeft = false;
                        cellNeighbour.visited = true;
}
         currentCell = cellNeighbour;
```

Процес повторюється, поки існують невідвідані клітинки. На рисунку 3 представлено лабіринт 10x10, який згенеровано за алгоритмом Aldous-Broder algorithm.

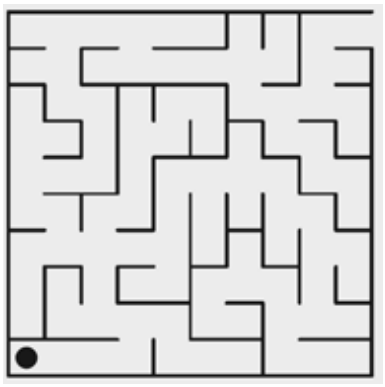

Рис. 3. Лабіринт 10x10, згенерований за алгоритмом Aldous-Broder algorithm

Hunt-and-Kill algorithm. Процес починається з режиму «знищення». Він починається в будь-якому вузлі u сітки. Потім процес переміщується до одного з невідвіданих сусідніх вузлів, вузла v, випадковим чином. Ребро між вузлами u і v стає ребром остовного дерева. Процес послідовно переміщується до невідвіданих сусідів, поки не досягне вузла, де немає невідвіданого сусіда. Це режим вбивства. Після цього процес переходить у режим пошуку. У режимі пошуку процес сканує сітку, щоб знайти невідвіданий вузол, w, який є сусіднім з одним із відвіданих вузлів, вузлом x. Процес відвідує вузол w, і ребро між вузлами w і x стає ребром остовного дерева. Потім він переходить у режим «знищення» та знову переміщується до одного зі своїх невідвіданих сусідів. Процес зупиняється, коли відвідуються всі вузли в сітці [1].

Метод Kill(HuntAndKillCell[,] maze, int currentX, int currentY, int width, int height) приймає в якості аргументів масив клітинок лабіринту, ідентифікатори поточної клітинки та розміри лабіринту. Роль методу – видалити стіну між поточною клітинкою та однією із сусідніх, вибір якої проходить шляхом перевірки, чи вона ще не відвідана.

```
if (neighbourCell.X == currentX){
                    if (neighbourCell.Y > currentY && neighbourCell.Y < height - 1){
                        neighbourCell.wallBottom = false;
                    }
                    else{
                        currentCell.wallBottom = false;
                    }
                }
                else{
                    if (neighbourCell.X > currentX && neighbourCell.X < width - 1){
                        neighbourCell.wallLeft = false;
                    }
                    else{
                        currentCell.wallLeft = false;
                    }
                }
                currentX = neighbourCell.X;
                currentY = neighbourCell.Y;
```

На рисунку 4 представлено лабіринт 10x10, який згенеровано за алгоритмом Hunt-and-Kill algorithm.

Binary Tree Algorithm. Для кожної комірки в сітці випадковим чином виділяється прохід на північ або на захід, також можна вибрати між іншими наборами діагоналей: північ/схід, південь/захід або південь/схід, але будь-який набір діагоналей має використовуватися постійно, весь лабіринт. Недоліком цього алгоритму є те, що він має сильний діагональний ухил, тобто коридори охоплюють межі вибраного напрямку. Крім того, дві з чотирьох сторін лабіринту будуть охоплені одним проходом.

Метод для даного алгоритму доволі простий. У циклі кожній клітинці, за умови, що та не знаходиться з крайнього нижнього або лівого краю, видаляється нижня або ліва стіна.

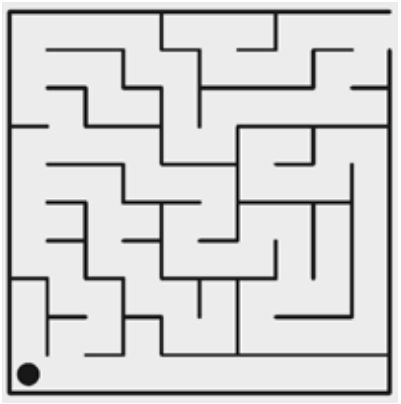

Рис. 4. Лабіринт 10x10, згенерований за алгоритмом Hunt-and-Kill algorithm

```
private void RemoveWalls(BinaryTreeCell[,] maze, int width, int height){
        for (int x = 0; x < width - 1; x++){
            for (int y = 0; y < height - 1; y++){
                //carve a passage either south or west
                bool deleteLeftWall = Random.RandomRange(0, 2) == 0;
                if (y < 1 && x < 1){
                    continue;
                }
                else if (x < 1){
                    deleteLeftWall = false;
                }
                else if (y < 1){
                    deleteLeftWall = true;
                }

                if (deleteLeftWall){
                    maze[x, y].wallLeft = false;
                }
                else{
                    maze[x, y].wallBottom = false;
                }
            }
        }
    }
```

На рисунку 5 представлено лабіринт 10x10, який згенеровано за алгоритмом Binary Tree algorithm.

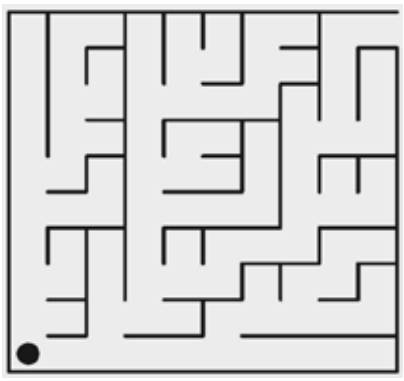

Рис. 5. Лабіринт 10x10, згенерований за алгоритмом Binary Tree algorithm

Результати дослідження

В результаті дослідження були визначені переваги і недоліки алгоритмів різних методів створення лабіринтів. У таблиці 1 представлені переваги і недоліки досліджуваних алгоритмів.

Таблиця 1

Переваги і недоліки досліджуваних алгоритмів

Алгоритм	Переваги	Недоліки
Recursive backtracker	Простий та легко реалізований	Може створювати лабіринти з великою кількістю довгих коридорів
Kruskal's algorithm	Генерує лабіринти з рівномірно розподіленими шляхами	Обчислювально витратний для великих лабіринтів
Aldous-Broder algorithm	Простий та безупинний процес	Велика варіативність форми лабіринту. Не завжди ефективний для випадкової ходьби
Hunt-and-Kill algorithm	Може створити лабіринти з відкритими просторами. Збалансована випадковість та структура	Випадковість може призводити до менш передбачуваних лабіринтів
Binary Tree Algorithm	Простий та легкий для реалізації. Швидкий та ефективний	Один напрямок утворення коридорів. Обмежена варіативність форми лабіринту

У таблиці 2 представлені результати роботи алгоритмів відносно використаних ресурсів.

Таблиця 2

Порівняння алгоритмів відносно використання ресурсів

Алгоритм	Використання ресурсів	Час виконання генерації лабіринту розміром (у мілісекундах)			
		10x10	50x50	100x100	250x250
Recursive backtracker	Використовує пам'ять для стеку рекурсії	8	136	754	6985
Kruskal's algorithm	Використовує пам'ять для зберігання наборів	15	163	1508	21725
Aldous-Broder algorithm	Вимагає додаткової пам'яті для зберігання стану лабіринту	30	864	12953	50009
Hunt-and-Kill algorithm	Використовує пам'ять для зберігання стану та поточних координат	15	230	1028	7460
Binary Tree Algorithm	Використовує пам'ять для зберігання стану та поточних координат	6	189	623	4412

На основі даних, отриманих в результаті дослідження, побудовано графік (рис. 6), на якому зображується залежність часу побудови від розміру лабіринту для кожного з алгоритмів. Найбільш стрімко зростає графік алгоритму Aldous-Broder algorithm (синього кольору). Найменше часу займає побудова за допомогою Binary Tree Algorithm (фіолетовий колір графіку). Між ними розташовані: Kruskal's algorithm – червоний колір, Hunt-and-Kill algorithm – помаранчевий колір та Recursive backtracker – зелений колір.

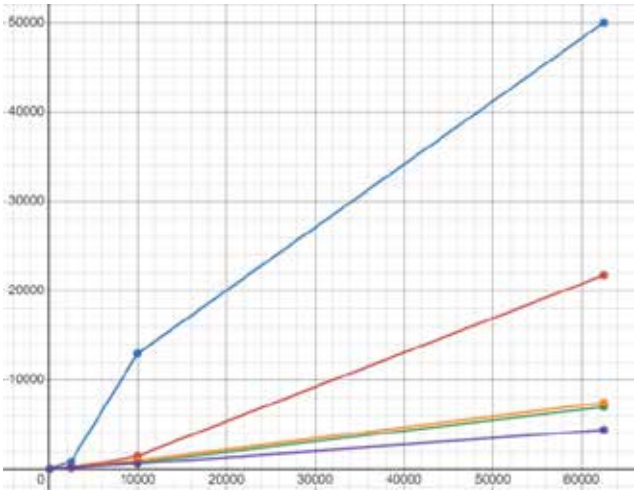

Рис. 6. Графік залежності часу побудови лабіринту від розміру лабіринту для різних алгоритмів

Висновки

В результаті дослідження було проаналізовано та порівняно алгоритми генерації лабіринтів. Отже, можна скласти заключну характеристику методів.

За ефективністю та складністю:

– Recursive Backtracker простий та ефективний, але може призводити до створення довгих та вузьких коридорів.

– Kruskal's Algorithm, ефективний, особливо для великих лабіринтів. Застосовує об'єднання наборів ребер.

– Aldous-Broder Algorithm не завжди ефективний через випадковий вибір шляхів, може мати велику кількість зайвих рухів.

– Hunt-and-Kill Algorithm варіативний, але може вести до створення лабіринтів з нерівномірним розподілом шляхів.

– Binary Tree Algorithm простий та швидкий, але створює лабіринти з сильним діагональним ухилом.

За оптимальністю використання для Unity:

– Recursive Backtracker легко реалізовується в Unity, підходить для основних лабіринтів.

– Kruskal's Algorithm вимагає управління структурою даних, але добре підходить для більших лабіринтів.

– Aldous-Broder Algorithm може бути менш ефективним через випадковий вибір шляхів, але реалізується в Unity.

– Hunt-and-Kill Algorithm складніший у реалізації в Unity, може бути менш оптимальним для деяких проектів.

– Binary Tree Algorithm швидкий та простий у реалізації в Unity, але має свої обмеження у структурі лабіринту.

За швидкістю генерації:

– У Recursive Backtracker швидка генерація, для великих лабіринтів час збільшується поступово.

– Kruskal's Algorithm невеликі лабіринти генеруються досить швидко, але зі збільшенням розмірів час різко збільшується.

– У Aldous-Broder Algorithm генерація займає дуже багато часу, особливо для великих лабіринтів.

– У Hunt-and-Kill Algorithm генерація досить швидка, час росте досить плавно зі збільшенням розмірів лабіринту.

– У Binary Tree Algorithm генерація відбувається дуже швидко, навіть для великих лабіринтів.

Отже, у кожного з алгоритмів є свої переваги та недоліки, але, враховуючи час та вигляд лабіринту, оптимальний вибір – Recursive backtracker. Даний метод легко реалізувати, він швидкий та не потребує багато ресурсів. Його можна обрати для проекту, який не має особливих вимог. Але не потрібно нехтувати іншими методами, так як вони можуть гарантувати різноманітність форм лабіринту, навіть якщо генерація займе більше часу.

Список використаної літератури

1. Kim P. H. Intelligent Maze Generation. 2019. URL: http://rave.ohiolink.edu/etdc/view?acc_num=osu1563286393237089.

2. Bontchev B., Panayotova R. Towards Automatic Generation of Serious Maze Games for Education. Serdica Journal of Computing. 2018. Vol. 11, no. 3-4. P. 249–278. URL: https://doi.org/10.55630/sjc.2017.11.249-278.

3. Safak A. B., Bostanci E., Soylucicek A. E. Automated Maze Generation for Ms. Pac-Man Using Genetic Algorithms. International Journal of Machine Learning and Computing. 2016. Vol. 6, no. 4. P. 226–230. URL: https://doi.org/10.18178/ijmlc.2016.6.4.602.

4. Jeong K., Kim J. Event-Centered Maze Generation Method for Mobile Virtual Reality Applications. Symmetry. 2016. Vol. 8, no. 11. P. 120. URL: https://doi.org/10.3390/sym8110120.

5. Mazes in videogames: meaning, metaphor and design. Choice Reviews Online. 2013. Vol. 51, no. 02. P. 51–0692–51–0692. URL: https://doi.org/10.5860/choice.51-0692.

6. Gazzard A. Paths, players, places : towards an understanding of mazes and spaces in videogames : thesis. 2010. URL: http://hdl.handle.net/2299/4804.

7. Nwankwo G., Mohammed S., Fiaidhi J. Procedural Content Generation for Dynamic Level Design and Difficulty in a 2D Game Using UNITY. International Journal of Multimedia and Ubiquitous Engineering. 2017. Vol. 12, no. 9. P. 41–52. URL: https://doi.org/10.14257/ijmue.2017.12.9.04.

8. Watkins R. Procedural Content Generation for Unity Game Development. Packt Publishing, Limited, 2016.

9. Марчук Г.В., Любченко Д.В. Генерація лабіринтів за допомогою алгоритму Hunt and Kil. «Вчені записки ТНУ імені В.І. Вернадського». Серія: технічні науки, Том 35(74) № 3 Ч.1. 2024. С. 130-135. URL: https://doi.org/10.32782/2663-5941/2024.3.1/20

10. Development and Evaluation of Maze-Like Puzzle Games to Assess Cognitive and Motor Function in Aging and Neurodegenerative Diseases / T. Nef et al. Frontiers in Aging Neuroscience. 2020. Vol. 12. URL: https://doi.org/10.3389/fnagi.2020.00087.

References

1. Kim P. H. (2019) Intelligent Maze Generation. URL: http://rave.ohiolink.edu/etdc/view?acc_num=osu1563286393237089.

2. Bontchev B., Panayotova R. (2018) Towards Automatic Generation of Serious Maze Games for Education. Serdica Journal of Computing. Vol. 11, no. 3-4. P. 249–278. URL: https://doi.org/10.55630/sjc.2017.11.249-278.

3. Safak A. B., Bostanci E., Soylucicek A. E. (2016) Automated Maze Generation for Ms. Pac-Man Using Genetic Algorithms. International Journal of Machine Learning and Computing. Vol. 6, no. 4. P. 226–230. URL: https://doi.org/10.18178/ijmlc.2016.6.4.602.

4. Jeong K., Kim J. (2016) Event-Centered Maze Generation Method for Mobile Virtual Reality Applications. Symmetry. Vol. 8, no. 11. P. 120. URL: https://doi.org/10.3390/sym8110120.

5. Mazes in videogames: meaning, metaphor and design. Choice Reviews Online. 2013. Vol. 51, no. 02. P. 51–0692–51–0692. URL: https://doi.org/10.5860/choice.51-0692.

6. Gazzard A. (2010) Paths, players, places : towards an understanding of mazes and spaces in videogames : thesis. URL: http://hdl.handle.net/2299/4804.

7. Nwankwo G., Mohammed S., Fiaidhi J. (2017) Procedural Content Generation for Dynamic Level Design and Difficulty in a 2D Game Using UNITY. International Journal of Multimedia and Ubiquitous Engineering. Vol. 12, no. 9. P. 41–52. URL: https://doi.org/10.14257/ijmue.2017.12.9.04.

8. Watkins R. (2016) Procedural Content Generation for Unity Game Development. Packt Publishing, Limited.

9. Marchuk G.V., Liubchenko D.V. (2024) Heneratsiia labiryntiv za dopomohoiu alhorytmu Hunt and Kil. «Vcheni zapysky TNU imeni V.I. Vernadskoho». Seriia: tekhnichni nauky, Tom 35(74) № 3 Ch.1. S.130-135. URL: https://doi.org/10.32782/2663-5941/2024.3.1/20 [in Ukrainian].

10. Development and Evaluation of Maze-Like Puzzle Games to Assess Cognitive and Motor Function in Aging and Neurodegenerative Diseases / T. Nef et al. Frontiers in Aging Neuroscience. 2020. Vol. 12. URL: https://doi.org/10.3389/fnagi.2020.00087.