**ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**
**НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ**
**ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**
**КАФЕДРА КОМП'ЮТЕРНИХ НАУК**

# КВАЛІФІКАЦІЙНА РОБОТА

на тему: ««Розробка програми генерації лабіринтів для комп'ютерних ігор на мові Java»»

на здобуття освітнього ступеня бакалавра
зі спеціальності 122 Комп'ютерні науки
*(код, найменування спеціальності)*
освітньо-професійної програми Комп'ютерні науки
*(назва)*

*Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело*

_____ Софія Лоскучерява
*(підпис)* *(Ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти Софія Лоскучерява
група КНД-41

Керівник: Ільїн Олег Олександрович
*науковий ступінь, вчене звання* д.т.н.,професор кафедри

Рецензент:
*науковий ступінь, вчене звання*
_____
*(Ім'я, ПРІЗВИЩЕ)*

**Київ 2024**

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

**Навчально-науковий інститут інформаційних технологій**

Кафедра Комп'ютерних наук

Ступінь вищої освіти Магістр

Спеціальність Комп'ютерні науки

Освітньо-професійна програма Комп'ютерні науки

**ЗАТВЕРДЖУЮ**
Завідувач кафедрою Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
«_____» _____ 2023 р.

# ЗАВДАННЯ
# НА КВАЛІФІКАЦІЙНУ РОБОТУ

Лоскучерявої Софії Сергіївни
*(прізвище, ім'я, по батькові здобувача)*

1. Тема кваліфікаційної роботи: Розробка програми генерації лабіринтів для комп'ютерних ігор на мові Java
керівник кваліфікаційної роботи Олег ІЛЬЇН д.т.н., професор кафедри,
*(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)*
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «27» 02.2024р. №36

2. Строк подання кваліфікаційної роботи «31» травня 2024р.

3. Вихідні дані до кваліфікаційної роботи: Література з питань,що пов'язані з розробкою лабирінтів на мові Java і побудови маршруту з початку до кінця

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Розгляд алгоритмів та методів пошуку шляху в генерації лабіринтів
Вибір програми в розробці генерації лабіринтів
Розробка програми генерації лабіринту
5. Перелік графічного матеріалу: *презентація*
1.Блок-схема
2.Результати роботи створенної програми
6. Дата видачі завдання«23» лютого 2024 р.

# КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів кваліфікаційної роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Аналіз наявної науково-технічної літератури | 23.02-11.04 | |
| 2 | Дослідження поставленої задачі,алгоритмів та методів пошуку | 11.03-14.03 | |
| 3 | Аналіз методів розв’язання задачі | 13.03-17.03 | |
| 4 | Застосування алгоритмів на практиці | 20.03-31.03 | |
| 5 | Аналіз отриманих результатів генерації лабіринтів | 01.04-04.04 | |
| 6 | Розробка програми генерації лабіринтів,а також на їх аналогах | 17.03-04.04 | |
| 7 | Написання вступу,висновків,реферату | 29.03-27.04 | |
| 8 | Розробка демонстраційних матеріалів | 20.03- 12.05 | |

Здобувач вищої освіти _____ Софія Лоскучерява
*(підпис)* *(Ім’я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Олег Ільїн
*(підпис)* *(Ім’я, ПРІЗВИЩЕ)*

# РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 69 стор.,6 табл.,16 рис.,12 джерел.

*Наукове завдання* – розробка на програмі мовою Java Eclipse генерацію лабіринтів ,а також на їх аналогах Ruby Proccesing,в одному із них буде продемонстровано алоритм пошуку шляху.

*Мета роботи* – проаналізувати алгоритм лабіринту та алгоритм у пошуку A* та реалізувати.

*Об'єкт дослідження* – процес генерації лабіринтів у програмуванні та розробка алгоритмів для автоматичної генерації різних типів лабіринтів у програмних середовищах.

*Предмет дослідження* – реалізація генерації лабіринтів та побудови пошуку шляху в лабіринті від початку до кінця мовою програмування Java.

*Короткий зміст роботи*: Проведено аналіз побудування лабіринту та їх використання.Розглянуті алгоритми генерації лабіринтів,а також методи пошуку шляху в лабіринтах.

Розглянуто застосування алгоритму Еллера як основу для генерації лабіринтів,а також метод пошуку A* в якості бонусу для пошуку шляху від початку до кінця

Розроблено програму для генерації лабіринтів на Eclipse, який будує ідеальний лабіринт,а також на аналогах програм мовою Java,таких як Ruby,в якому будується лабіринт на командній строці з командою зупинки за допомогою клавіш та Proccesing,де окрім генерації лабіринту з клітинок,генерує старт і фініш і будує шлях від початку до кінця.

КЛЮЧОВІ СЛОВА: ЛАБІРИНТ,АВТОМАТИЧНА ГЕНЕРАЦІЯ,ІДЕАЛЬНІ АЛГОРИТМИ СТВОРЕННЯ ЛАБІРИНТУ,АЛГОРИТМ ПОШУКУ A*

## ABSTRACT

Text part of the master's qualification work: 69 pages, 16 pictures, 6 tables, 12 sources.

The purpose of the work – creating the maze generation using the Java Eclipse program, as well as their analogues Ruby Proccesing, in one of them the search algorithm will be demonstrated.

Object of research – the process of generating mazes in programming and the development of algorithms for the automatic generation of various types of mazes in software environments..

Subject of research – implementation of maze generation and maze pathfinding construction from start to finish in the Java programming language.

Summary of the work: In the work, a program was created for generating mazes on Eclipse, which builds an perfect maze, as well as on analogues of programs in the Java language, such as Ruby, in which a maze is built on the command line with a stop command using keys and Proccesing, where, in addition to generating a maze from cells, it generates a start and finish and builds a path from start to finish.



KEY WORDS: MAZE, AUTOMATIC GENERATION, IDEAL ALGORITHMS FOR CREATING A MAZE, SEARCH ALGORITHM A*

**ЗМІСТ**

# ВСТУП

У сучасному інформаційному суспільстві, де обсяги даних постійно зростають, виникає необхідність у розробці ефективних методів обробки та аналізу цих даних. Одним із важливих завдань є генерація лабіринтів в програмуванні, яка відіграє ключову роль у різних галузях, від ігор до робототехніки. Завдяки генерації лабіринтів можна створювати складні інтерактивні сценарії та досліджувати різноманітні проблеми, що виникають у процесі навчання штучного інтелекту, проектування оптимальних маршрутів для роботів та багато іншого.

Напрямок досліджень генерації лабіринтів стає все більш актуальним у зв'язку із зростанням потреб у розвитку інтерактивних додатків та вирішенні складних задач у галузі штучного інтелекту. Сучасні технології дозволяють розробляти складні алгоритми для генерації різноманітних типів лабіринтів, що відкриває нові перспективи для розробки ігор, навчальних програм, інтерактивних систем та багатьох інших застосувань.

На сьогоднішній день існує безліч алгоритмів та методів для генерації лабіринтів. До найпоширеніших належать алгоритми: рекурсивний пошук у глибину (Depth-First Search, DFS), рекурсивний пошук у ширину (Breadth-First Search, BFS), алгоритм Прима (Prim's algorithm), алгоритм Крускала (Kruskal's algorithm), алгоритм Ейлера (Euler's algorithm) та інші.

Правильно розроблені алгоритми для генерації лабіринтів можуть значно полегшити процес створення програмних додатків, забезпечити нові можливості для дослідження та дозволити реалізувати інноваційні ідеї в різних галузях індустрії та науки. Тому розвиток цього напрямку досліджень має великий практичний та науковий потенціал, який потребує подальших досліджень та вдосконалення алгоритмів.

Крім теоретичних аспектів, важливо також розглянути практичну реалізацію генерації лабіринтів у програмному коді. Академічні дослідження дозволяють нам не лише розвинути теоретичний фундамент, а й створити реалізацію цих алгоритмів у вигляді програм, що можуть бути використані у різних проектах. Практична реалізація важлива для перевірки ефективності та функціональності алгоритмів у реальних умовах та може послужити основою для подальших досліджень та розробок.

Генерація лабіринтів в програмуванні стає все більш актуальною у зв'язку з розвитком ігрової індустрії, штучного інтелекту, а також вирішенням різних завдань у робототехніці. Із зростанням кількості ігор, де лабіринти виступають ключовим елементом, потрібно розробляти ефективні алгоритми їх генерації.

Об'єктом дослідження є процес генерації лабіринтів у програмуванні та розробка алгоритмів для автоматичної генерації різних типів лабіринтів у програмних середовищах.

Предметом дослідження є розробка та оптимізація алгоритмів для генерації лабіринтів з різними характеристиками, такими як розмір, складність, наявність перешкод та інші параметри.

Метою даної дипломної роботи є розробка та апробація ефективних алгоритмів для генерації лабіринтів в програмуванні. Основна увага буде зосереджена на розробці алгоритмів, що забезпечують створення лабіринтів з різними характеристиками, такими як розмір, складність, наявність перешкод та інші.

**Для досягнення мети роботи необхідно вирішити такі завдання:**

1)Розглянути основні поняття лабіринтів та їх алгоритми виконання

2)Реалізувати версію алгоритма мовою Java в Eclipse,Ruby та Processing.

3) Перевірити працездатність та коректність роботи програм

4) Виявити переваги з реалізації згенерированого лабіринту та пошуку шляху від початку до кінця

# 1 ОСНОВНІ ВИЗНАЧЕННЯ ТА КЛАСИФІКАЦІЇ ЛАБІРИНТІВ ТА ЇХ АЛГОРИТМІВ ТА МЕТОДІВ

## 1.1 Основні поняття створення лабіринту

Лабіринт - це складна структура, що складається з мережі коридорів, стін та можливих шляхів, які можуть бути розгалуженими та перешкодженими. Він представляє собою простір, в якому необхідно здійснити переміщення від початкової точки до кінцевої точки, обходячи або уникнувши перешкод (рис 1.1).

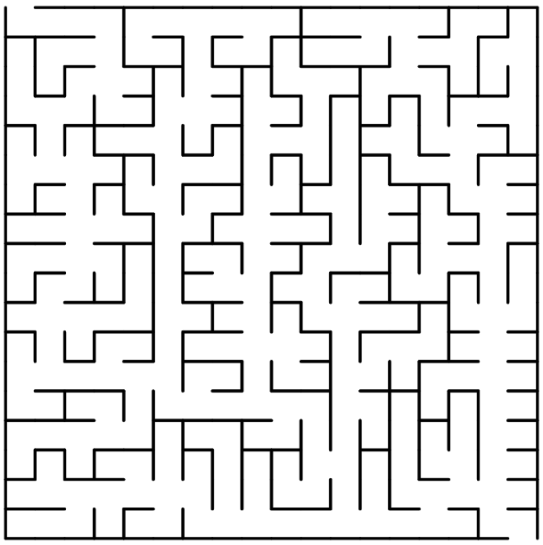


Рисунок 1.1-Приклад лабіринту модіфікованою версією алгоритму Прима

У програмуванні Java, лабіринт може бути використаний для розв'язання різних завдань, таких як ігри, алгоритми пошуку, штучний інтелект, візуалізація даних та інші. Лабіринти можуть бути представлені за допомогою структур даних, таких як двовимірні масиви або графи.У програмуванні Java, лабіринт може бути використаний для:

**1.Ігор:**Створення графічних ігор, де гравець повинен пройти через лабіринт, уникнувши перешкод та знаходячи шлях до мети.

**2.Алгоритмів пошуку:**Реалізація алгоритмів пошуку найкоротшого шляху або знаходження шляху через лабіринт.Використання лабіринту для тестування ефективності та правильності алгоритмів пошуку.

**3.Штучного інтелекту:**Використання лабіринту для навчання та тестування алгоритмів штучного інтелекту, таких як алгоритми навчання з підкріпленням або генетичні алгоритми.

**4.Візуалізації даних:**Використання лабіринту для візуалізації даних та їхньої обробки у вигляді графіків, діаграм, теплових карт та інших візуальних елементів.

У Java, лабіринти можуть бути представлені за допомогою різних структур даних, таких як двовимірні масиви, графи або власні класи для представлення вершин та ребер. Вони можуть бути оброблені за допомогою циклів, рекурсії, а також вбудованих або користувацьких алгоритмів для розв'язання конкретних завдань, пов'язаних з лабіринтом.

## 1.2 Методи теорії графів

Лабіринт можна створити, починаючи з визначеного розташування клітин (часто у формі прямокутної сітки, але можливі й інші конфігурації) зі стінками між ними. Це початкове розташування можна уявити як зв'язний граф, де ребра представляють потенційні стінки, а вузли – клітини. Мета алгоритму генерації лабіринту полягає в створенні підграфу, у якому важко знайти шлях між двома окремими вузлами.

Якщо підграф не є зв'язним графом, то деякі області будуть непотрібними, оскільки вони не сприятимуть простору пошуку. Якщо граф містить цикли, між деякими вузлами може існувати кілька шляхів. Тому для створення лабіринтів часто використовують методи генерації випадкового остовного дерева. Цикли, які можуть ускладнити розв'язання лабіринтів, можуть бути додані шляхом включення випадкових ребер під час виконання алгоритму (рис 1.2).

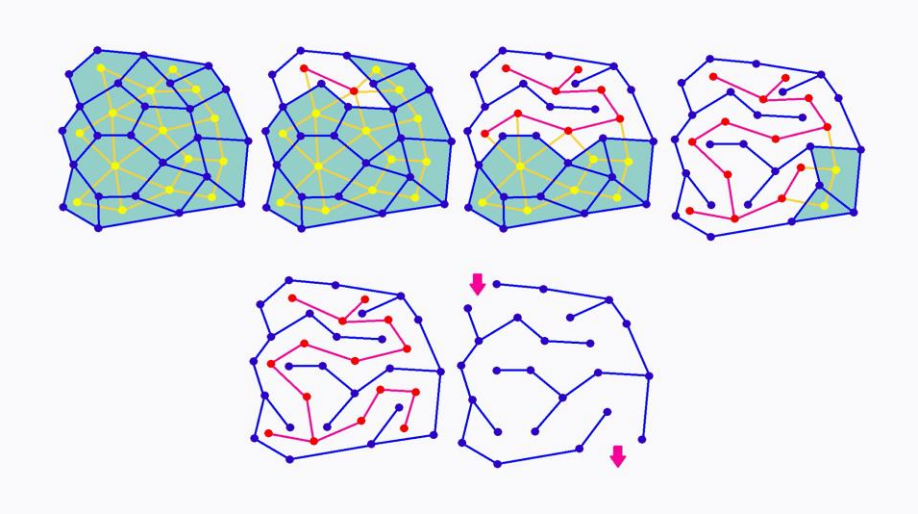


Рисунок.1.2-Анімація методу на основі теорії графів

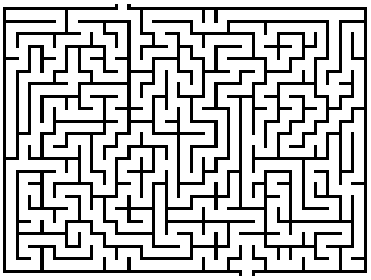
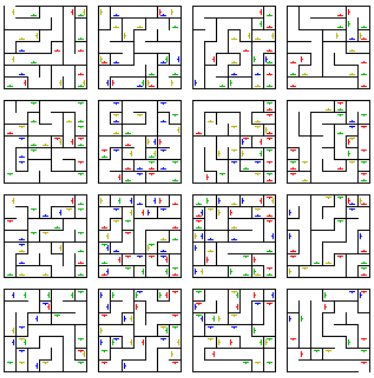
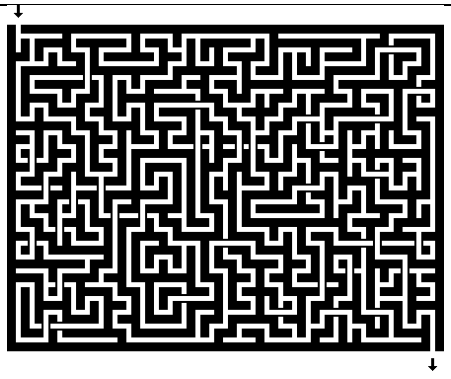
Анімація показує покрокове створення лабіринту на графі, який не розташований на прямокутній сітці. Спочатку комп'ютер генерує випадковий планарний граф G (синій), а його двозв'язний граф F – жовтим кольором. Потім комп'ютер обходить F за допомогою вибраного алгоритму, наприклад, пошуку в глибину, і позначає шлях червоним кольором. Під час обходу, коли червоне ребро перетинає синє, синє ребро видаляється. Після того як усі вершини F відвідані, F видаляється, як і два ребра графу G, що використовуються для входу та виходу.

### 1.3 Класифікація лабіринтів

Лабіринти та алгоритми для їх створення можна класифікувати за сімома різними критеріями: розмірністю, гіперрозмірністю, топологією, тесселяцією, маршрутизацією, текстурою та пріоритетом. Лабіринт може використовувати по одному елементу з кожної категорії в будь-якому поєднанні.

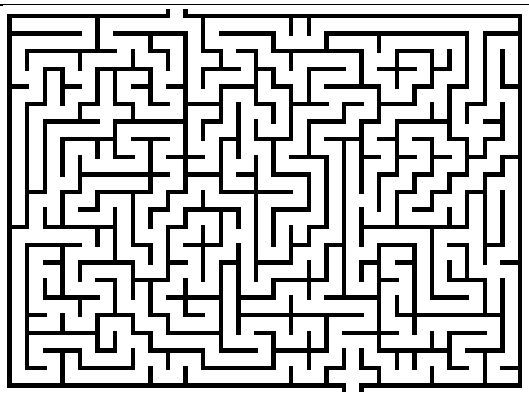
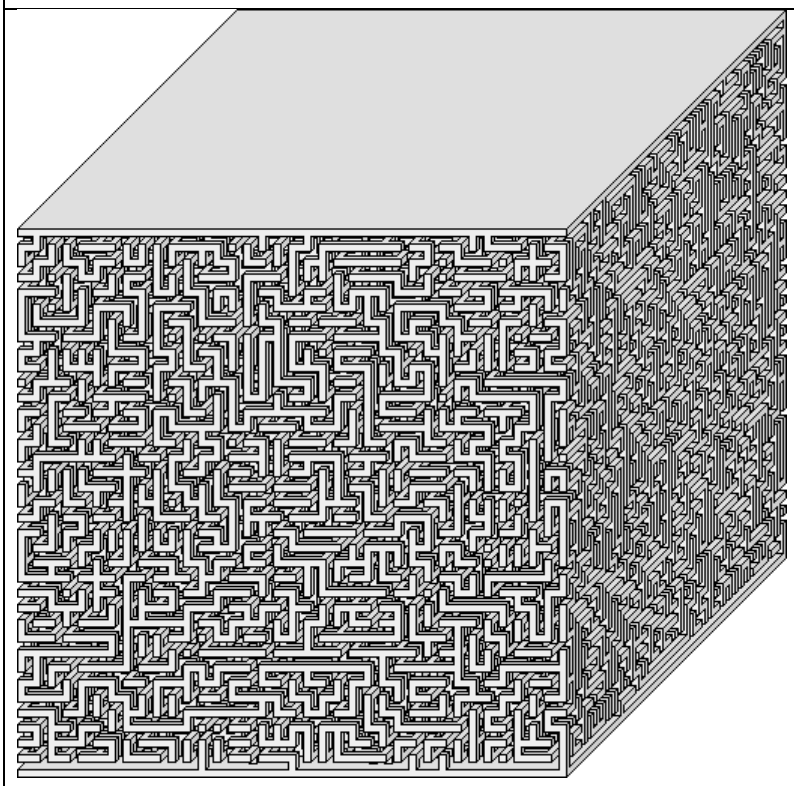
Розмірність: цей клас визначає, скільки вимірів у просторі займає лабіринт. Типи можуть бути наступними (табл 1.1):

Таблиця 1.1 Розмірна класифікація

| Вид | Опис |
|---|---|
|  | Двовимірний: більшість лабіринтів, як на папері, так і у фізичному світі, мають цю розмірність. Це означає, що план лабіринту можна зобразити на плоскому аркуші паперу, і переміщатися ним можна, не перетинаючи інші коридори. |
| | Трьохвимірний: такий лабіринт складається з кількох рівнів, де проходи можуть не тільки вести в чотири горизонтальні напрямки, але й підніматися вгору або спускатися вниз (в ортогональній версії). 3D-лабіринт часто уявляють як набір 2D-рівнів із позначками для сходів, що вказують напрямки "вгору" і "вниз". |
|  | Більш високі розмірності: можуть включати чотиривимірні лабіринти та навіть багатовимірніші конструкції. Іноді їх зображують як тривимірні лабіринти з «порталами», що проходять через четвертий вимір, наприклад, портали в «минуле» і «майбутнє». |
|  | Переплетення: це здебільшого двовимірні (або точніше 2,5-мірні) лабіринти, в яких проходи можуть накладатися один на одного. При візуалізації зазвичай чітко видно, де знаходиться глухий кут і як один прохід проходить над іншим. Лабіринти з реального світу, що мають мости, які з'єднують різні частини, є прикладами часткових переплетень. |

Гіпервимірність: клас гіпервимірності стосується розміру об’єкта, який ви переміщуєте лабіринтом, на відміну від розміру самого середовища лабіринту. Існують такі типи (табл 1.2):

Таблиця 1.2 Гіпервимірна класифікація

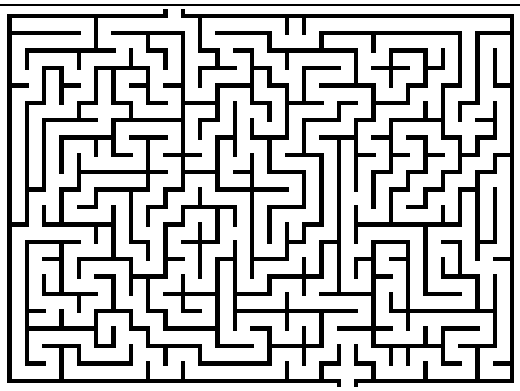
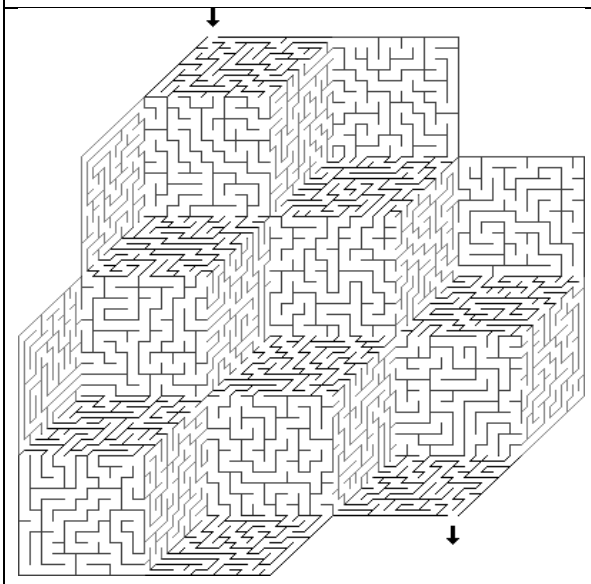
| Вид | Опис |
|---|---|
|  | Не-гіперлабіринти: майже всі лабіринти, навіть ті, що створені у високих розмірностях або з особливими правилами, зазвичай є не-гіперлабіринтами. В таких лабіринтах ми маємо справу з точкою або невеликим об'єктом, як-от кулькою чи самим гравцем, що рухаються від однієї точки до іншої, утворюючи лінійний маршрут. У кожній точці існує обмежена кількість варіантів вибору, які легко підрахувати. |
|  | Гіперлабіринти: це лабіринти, де об'єкт, який вирішується, не є просто точкою. Стандартний гіперлабіринт (або гіперлабіринт першого порядку) включає лінію, яка при русі утворює поверхню. Такі лабіринти можуть існувати тільки у тривимірному просторі або середовищах з більшою кількістю вимірів, і вхід до гіперлабіринту часто представлений лінією, а не точкою. Це принципово відрізняється тим, що необхідно враховувати і працювати з кількома частинами вздовж лінії одночасно. В будь-який момент існує практично нескінченна кількість можливих станів та варіантів дій з лінією. Лінія рішення або нескінченна, або її кінцеві точки розташовані за межами гіперлабіринту, щоб уникнути стиснення лінії до точки, що перетворило б його на звичайний лабіринт. |

Продовження таблиці 1.2

| | |
|---|---|
| | Гіпер-гіперлабіринти: можуть мати будь-яку кількість вимірів. Гіпер-гіперлабіринт (або гіперлабіринт другого порядку) підвищує розмірність об'єкта, що вирішується. У цьому випадку об'єктом є площина, яка при русі утворює об'ємну фігуру. Гіпер-гіперлабіринт можливий лише у чотиривимірному просторі або в середовищах з ще більшою кількістю вимірів. |

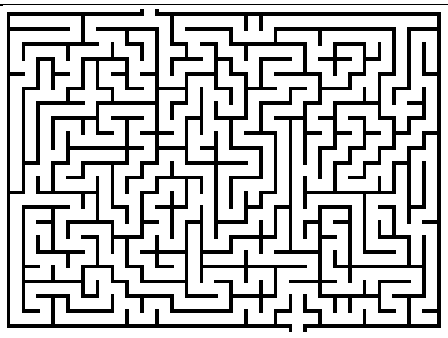
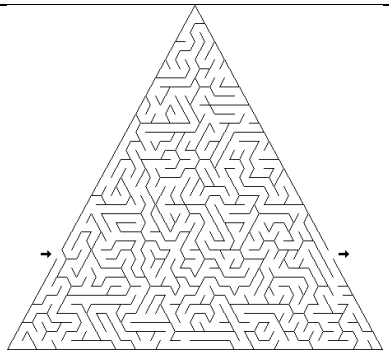
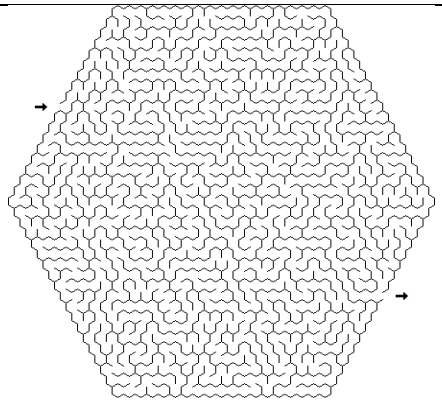
Топологія: визначає структуру простору лабіринту в цілому. Це відноситься до класифікації геометрії, яка характеризує, як об'єкти взаємодіють у лабіринті. Деякі типи топології включають (табл 1.3):

Таблиця 1.3 Топологічна класифікація

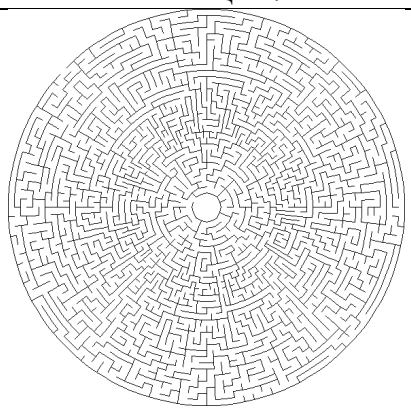
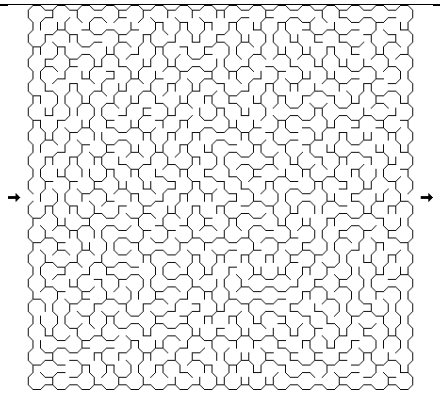
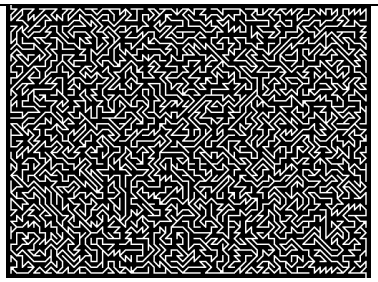
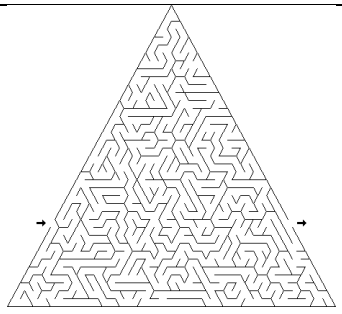
| Вид | Опис |
|---|---|
|  | Стандартний: це типовий лабіринт, який відповідає евклідовому простору. |
|  | Planair: термін, що використовується для опису лабіринтів з нетрадиційною топологією, де краї лабіринту з'єднані у цікавий спосіб. Наприклад, це може бути лабіринт на поверхні куба, лабіринт на поверхні стрічки Мебіуса або лабіринт, еквівалентний тору, де ліва і права сторони, а також верхня і нижня сторони з'єднані між собою. |

Тесселяція: класифікація визначає геометрічну структуру окремих клітинок або областей, включаючи лабіринти. Різноманітні типи тесселяції включають (табл 1.4):

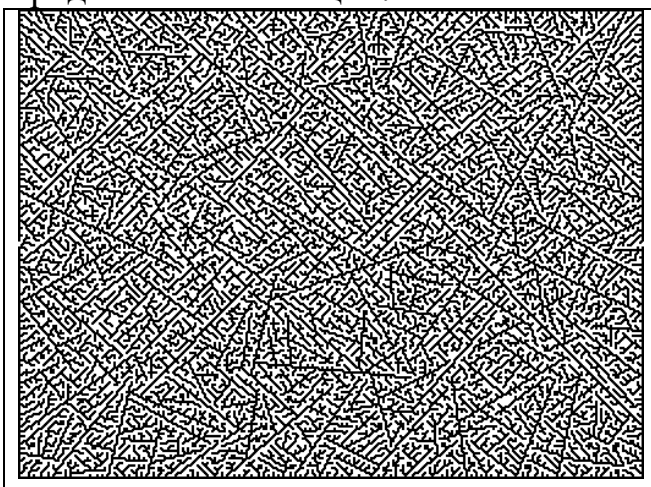
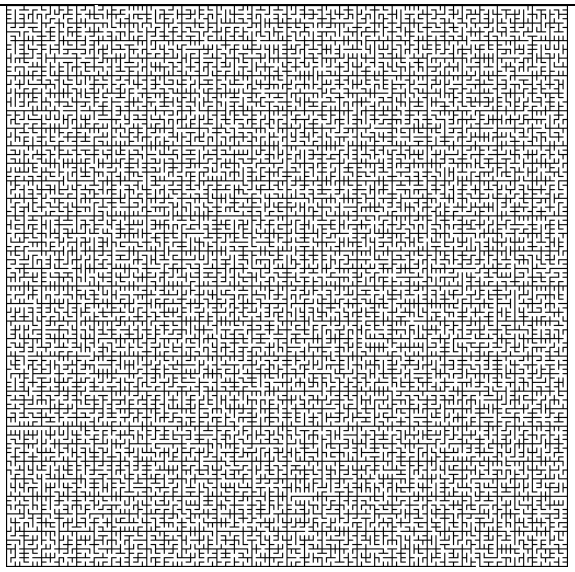
Таблиця 1.4 Тесселяційна класифікація

| Вид | Опис |
|---|---|
|  | Ортогональний: це типова сітка з прямокутними осередками, де проходи перетинаються під прямими кутами. У контексті тесселяції їх часто називають гамма-лабіринтами. |
|  | Дельта: складається зі з'єднаних трикутників, де кожен осередок може мати до трьох з'єднаних з ними проходів. |
|  | Сигма: включає в себе з'єднані шестикутники; у кожного осередка може бути до шести проходів. |

Продовження таблиці 1.4

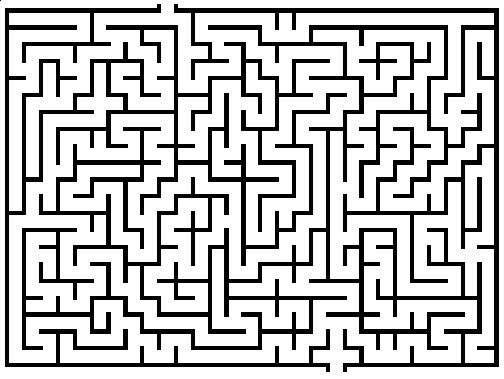
| | |
|---|---|
|  | Тета: це осередки з концентричними колами проходів, де один кінець або початок розташований у центрі, а інший - на зовнішньому краї. Зазвичай осередки мають чотири можливі з'єднання з проходами, але ця кількість може зростати через додаткові осередки у зовнішніх кільцях. |
|  | Іпсилон: складається зі з'єднаних восьмикутників або квадратів, де кожен осередок може мати до восьми або чотирьох проходів. |
|  | Дзета: це варіант на прямокутній сітці, де, крім горизонтальних та вертикальних проходів, дозволені діагональні під кутом 45 градусів. |
|  | Омега: термін «омега» описує майже будь-який лабіринт з нерегулярною тесселяцією. До цього типу відносяться, наприклад, дельта-, сигма-, тета- та іпсилон-лабіринти, а також багато інших схем, таких як лабіринти з парними прямокутними трикутниками. |

Продовження таблиці 1.4

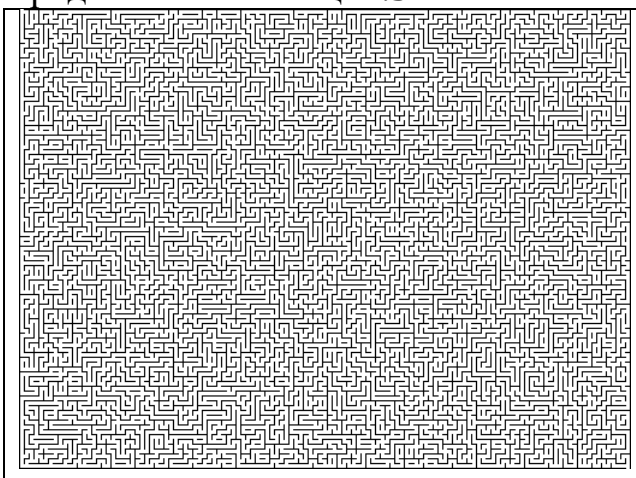
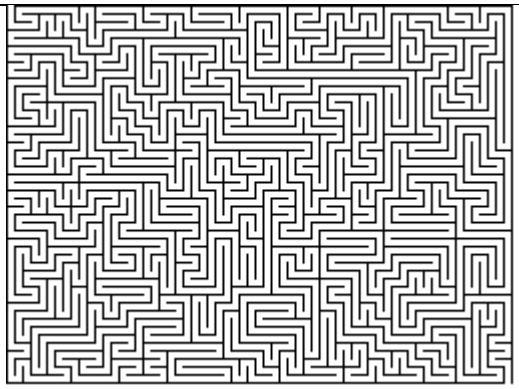
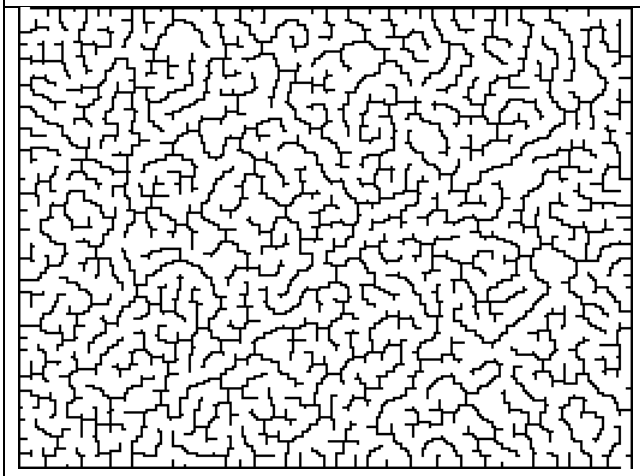
| | |
|---|---|
|  | Crack: це тип лабіринту без фіксованої тесселяції, де стіни та проходи мають випадкові кути. |
|  | Фрактальний: це лабіринт, що складається з менших лабіринтів. Фрактальний лабіринт з вкладеними осередками містить у собі лабіринти в кожному осередку, і цей процес може повторюватися кілька разів. Нескінченно рекурсивний фрактальний лабіринт — це справжній фрактал, у якому вміст лабіринту копіює себе, утворюючи насправді нескінченно великий лабіринт. |

Маршрутизація: класифікація маршрутизації – це, ймовірно, найцікавіший аспект у генерації лабіринтів. Від пов’язаний із типами проходів у межах геометрії, визначеної в описаних вище категоріях (табл 1.5).

Таблиця 1.5 Маршрутизаційна класифікація

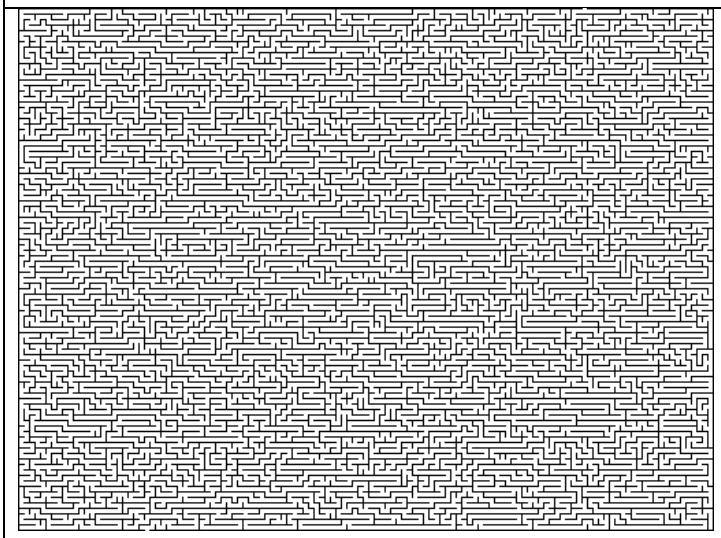
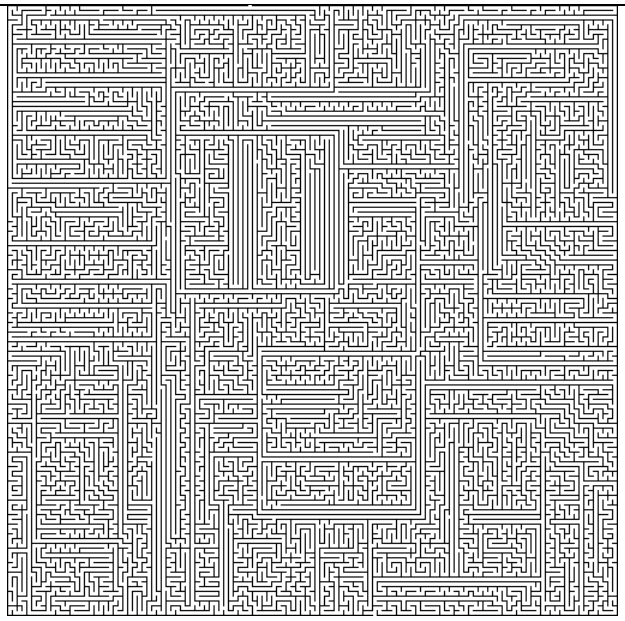
| Вид | Опис |
|---|---|
|  | Ідеальний:лабіринт без петель чи замкнутих ланцюгів та без недосяжних областей. Також він називається лабіринтом з одиночною сполукою (simply-connected Maze). З кожної точки існує рівно один шлях до будь-якої іншої точки. Лабіринт має лише одне рішення. З погляду програмування такий лабіринт можна описати як дерево, що сполучає безліч осередків чи вершин. |

Продовження таблиці 1.5

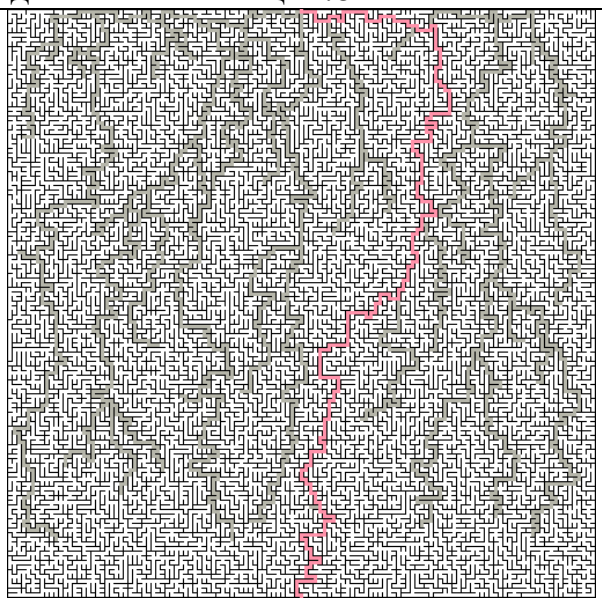
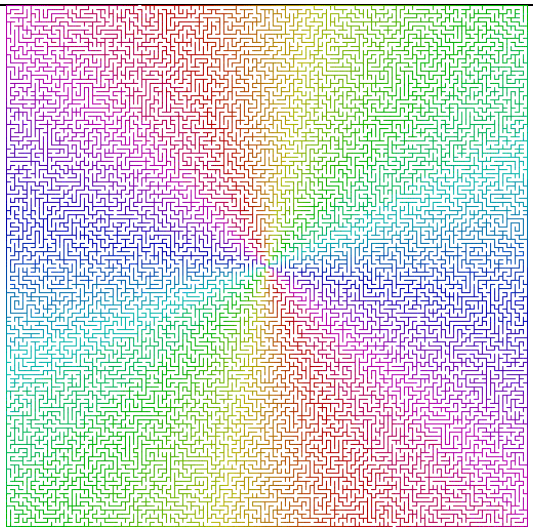
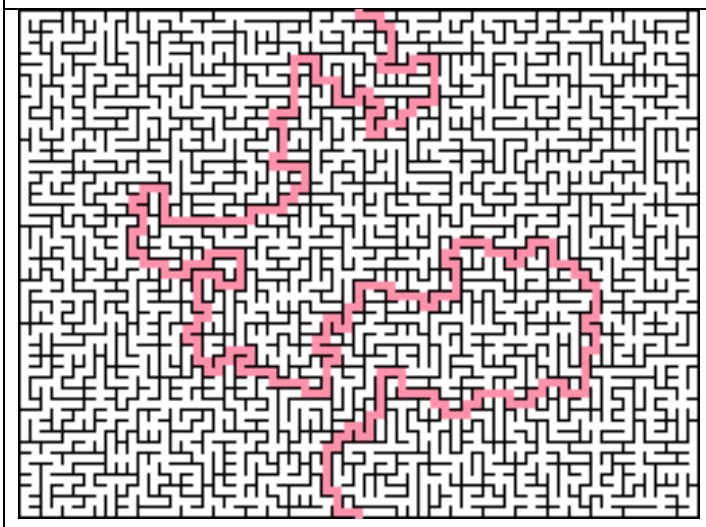
| | |
|---|---|
|  | Плетений (Braid): «плетений» означає, що в лабіринті немає глухих кутів. Також його називають лабіринтом із багаторазовими сполуками (purely multiply connected Maze). У такому лабіринті використовуються проходи, що замикаються та повертаються один до одного (звідси назва «плетений»), вони змушують витрачати більше часу на ходьбу колами замість попадання в глухий кут. Якісний плетений лабіринт може бути набагато складнішим за ідеальний лабіринт того ж розміру. |
|  | Одномаршрутний (Unicursal): під одномаршрутним мається на увазі лабіринт без розвилок. Одномаршрутний лабіринт містить один довгий прохід, що звивається, який змінює напрямок на всьому протязі лабіринту. Він не дуже складний, тільки якщо ви випадково не повернете назад на півдорозі і не повернетеся на початок. |
|  | Розріджений: розріджений лабіринт не прокладає проходи через кожну комірку, тобто деякі з них не створюються. Це має на увазі наявність недосяжних областей, тобто він у певному сенсі протилежний плетеному лабіринту. Подібну концепцію можна застосувати і при додаванні стін, завдяки чому можна отримати нерівномірний лабіринт із широкими проходами та кімнатами |
| | Частково плетений: частково плетений лабіринт - це змішаний лабіринт, в якому є і петлі, і глухий кут. Слово «плетений» можна використовувати для кількісної оцінки, тобто «лабіринт із сильним плетінням» - це лабіринт з безліччю петель або прибраних стін, а в «лабіринті зі слабким плетивом» їх лише кілька. |

Текстура: класифікація за текстурою описує стиль проходів за різної маршрутизації та геометрії. Текстура – це не просто параметри, які можна вмикати або вимикати. Ось кілька прикладів змінних (табл 1.6):

Таблиця 1.6.Текстурна класифікація

| Вид | Опис |
|---|---|
|  | Зміщення (Bias): у лабіринті зі зміщеними проходами прямі проходи схильні більше йти в одному напрямку, ніж в інших. Наприклад, у лабіринті з високою горизонтальною зміщеністю ми матимемо довгі проходи зліва направо, і лише короткі проходи зверху вниз, що з'єднують їх. Такий лабіринт зазвичай складніше проходити «поперек волокон». |
|  | Прольоти: показник «прольотів» визначає, як довго йдуть довгі проходи, перш ніж з'являться вимушені повороти. У лабіринті з низьким показником прольотів не буде прямих проходів довше трьох-чотирьох осередків, і він виглядатиме дуже випадковим. У лабіринті з високим показником прольотів протягом лабіринту буде великий відсоток довгих проходів, через що він буде схожим на мікрочіп. |

Продовження таблиці 1.6

| | |
|---|---|
|  | Елітність: показник елітності лабіринту визначає довжину рішення щодо розміру лабіринту. У елітних лабіринтів зазвичай є коротке пряме рішення, а в неелітних лабіринтах рішення проходить з великої частини площі лабіринту. Якісний елітний лабіринт може бути набагато складнішим, ніж неелітний. |
|  | Симетрія: симетричний лабіринт має симетричні проходи, наприклад, симетрією обертання щодо центру, або відбитий по горизонтальній або вертикальній осях. Лабіринт може бути частково або повністю симетричним, і може повторювати патерн будь-яку кількість разів. |
|  | Однорідність: однорідний алгоритм генерує всі можливі лабіринти з ймовірністю. Лабіринт можна назвати таким, що має однорідну текстуру, якщо він виглядає як типовий лабіринт, згенерований однорідним алгоритмом. Неоднорідний алгоритм теоретично також може згенерувати всі можливі лабіринти у будь-якому просторі, але з рівною ймовірністю. Неоднорідність може зайти ще далі — можливе існування лабіринтів, які алгоритм ніколи не згенерує. |

Продовження таблиці 1.6

| | |
|---|---|
| | Плинність (River): характеристика «плинність» означає, що при створенні лабіринту алгоритм шукатиме і очищатиме сусідні осередки (або стіни) до поточної, тобто буде текти (звідси і термін «плинність») у ще нестворені частини лабіринту, як вода. В ідеальному лабіринті з меншим показником плинності зазвичай буде безліч коротких глухих кутів, а в більш «плинному» лабіринті буде менше глухих кутів, але вони виявляться довшими. |

Пріоритет: ця класифікація показує, що процеси створення лабіринтів можна розділити на два основні типи: додають стіни та прорізи, що вирізають. Зазвичай при генерації це зводиться тільки до різниці в алгоритмах, а не до помітних відмінностей лабіринтів, але це корисно враховувати. Один і той же лабіринт часто генерується обома способами:

1.Створення стін: В алгоритмах, де основним завданням є розміщення стін, процес розпочинається з порожньої області або зовнішньої межі, і поступово додаються стіни. Реальні лабіринти, що складаються з огорож, перегородок або дерев'яних стін, використовують цей принцип, додаючи елементи стін для створення лабіринту.

2.Формування проходів: У алгоритмах, де пріоритетом є створення проходів, процес розпочинається з блоку або повної стіни, і проходи вирізаються в ньому. Цей підхід використовується у створенні лабіринтів, які можуть нагадувати тунелі шахт або лабіринти всередині труб.

3.Гібридний метод: Деякі комп'ютерні алгоритми одночасно створюють проходи та додають стіни, що відображається в шаблоні лабіринту. Це графічне зображення, яке не є лабіринтом, але може бути легко перетворене на нього, зберігаючи текстуру вихідного графічного шаблону. Складні стилі лабіринтів, такі як спіралі, часто краще реалізувати як шаблони, зберігаючи їхній стиль, замість того, щоб намагатися створити правильний лабіринт.

### 1.4 Ідеальні алгоритми створення лабіринту

Існує безліч способів створення ідеальних лабіринтів, і кожен з них має свої унікальні характеристики. Ось список алгоритмів по побудові ідеальних лабіринтів.У всіх них описано створення лабіринту вирізанням проходів, проте якщо не зазначено інше, кожен також можна реалізувати додаванням стін:

**1.Алгоритм Олдоса-Бродера** - Цікаве в тому, що він генерує лабіринти з однаковою ймовірністю для всіх можливих варіантів заданого розміру. Йому не потрібно додаткової пам'яті або стека.

**2. Лабіринти на основі двійкових дерев(Binary Tree)**- по суті, це найпростіші та найшвидші з можливих алгоритмів, проте створювані лабіринти мають текстуру з дуже високою зміщеністю

**3. Алгоритм Еллера** - це особливий алгоритм, тому що він не тільки найшвидший, але й не має очевидної зміщеності або недоліків; крім того, при створенні пам'ять використовується найбільш ефективно. Для нього навіть не потрібно, щоб у пам'яті знаходився весь лабіринт, він використовує об'єм, пропорційний розміру рядка.

**4. Алгоритм вирощування дерева (Growing tree algorithm)** - це узагальнений алгоритм, здатний створювати лабіринти із різною текстурою. Потрібна пам'ять може досягати розміру лабіринту.

**5. Алгоритм Hunt and Kill** - Цей метод зручний тим, що не потребує додаткової пам'яті чи стека. Він підходить для створення великих лабіринтів або для використання на слабких комп'ютерах, оскільки не може вичерпати пам'ять

**6. Алгоритм Краскала**-створює мінімальне сполучне дерево, вирізаючи сегменти проходів по всьому лабіринту випадковим чином. Для його роботи потрібна пам'ять, пропорційна розміру лабіринту.Для його роботи потрібно об'єм пам'яті, пропорційний розміру лабіринту, а також можливість перерахування кожного ребра або стіни між осередками лабіринту у випадковому порядку (зазвичай для цього створюється список всіх ребер і перемішується випадковим чином)

**7. Алгоритм Прима (спрощений):** Цей метод створює мінімальне сполучне дерево, присвоюючи випадкові ваги ребрам. Потребує обсяг пам'яті, що залежить від розміру лабіринту.

**8. Алгоритм Прима (істинний):** Також створює мінімальне дерево, використовуючи випадкові ваги ребрам. Пам'ять, необхідна для роботи, пропорційна розміру лабіринту.

**9. Рекурсивний зворотний трекер(Recursive backtracker):** Цей метод схожий на рекурсивний пошук і вимагає стека, розмір якого залежить від розміру лабіринту. При вирізанні він завжди вирізає прохід у невідвіданій частині, якщо вона існує поруч з поточною осередком.

**10. Рекурсивний поділ(Recursive division)-** цей алгоритм схожий на рекурсивний зворотний трекер, але працює зі стінами замість проходів.

**11. Лабіринти Sidewinder -** цей простий метод схожий на алгоритм двійкового дерева, але трохи складніший.

**12. Алгоритм Вілсона -** це вдосконалена версія алгоритму Олдоса-Бродера, яка швидше створює лабіринти з такою самою текстурою. Пам'ять, необхідна для роботи, зростає з розміром лабіринту. Починається з випадково обраного початкового осередку лабіринту.

## 1.5 Алгоритм Еллера

Алгоритм Еллера дозволяє процедурно генерувати одно маршрутний ідеальний лабіринт який не містить у собі циклів. Серед інших алгоритмів, важається найоптимальнішим через швидкий процес генерації, та невеликий об’єм пам’яті для зберігання даних про лабіринт (пропорційну довжені рядка лабіринту). Через специфіку запам’ятання лише останнього рядка, дозволяє генерувати нескінченно великий у довжину лабіринт (рис 1.3).

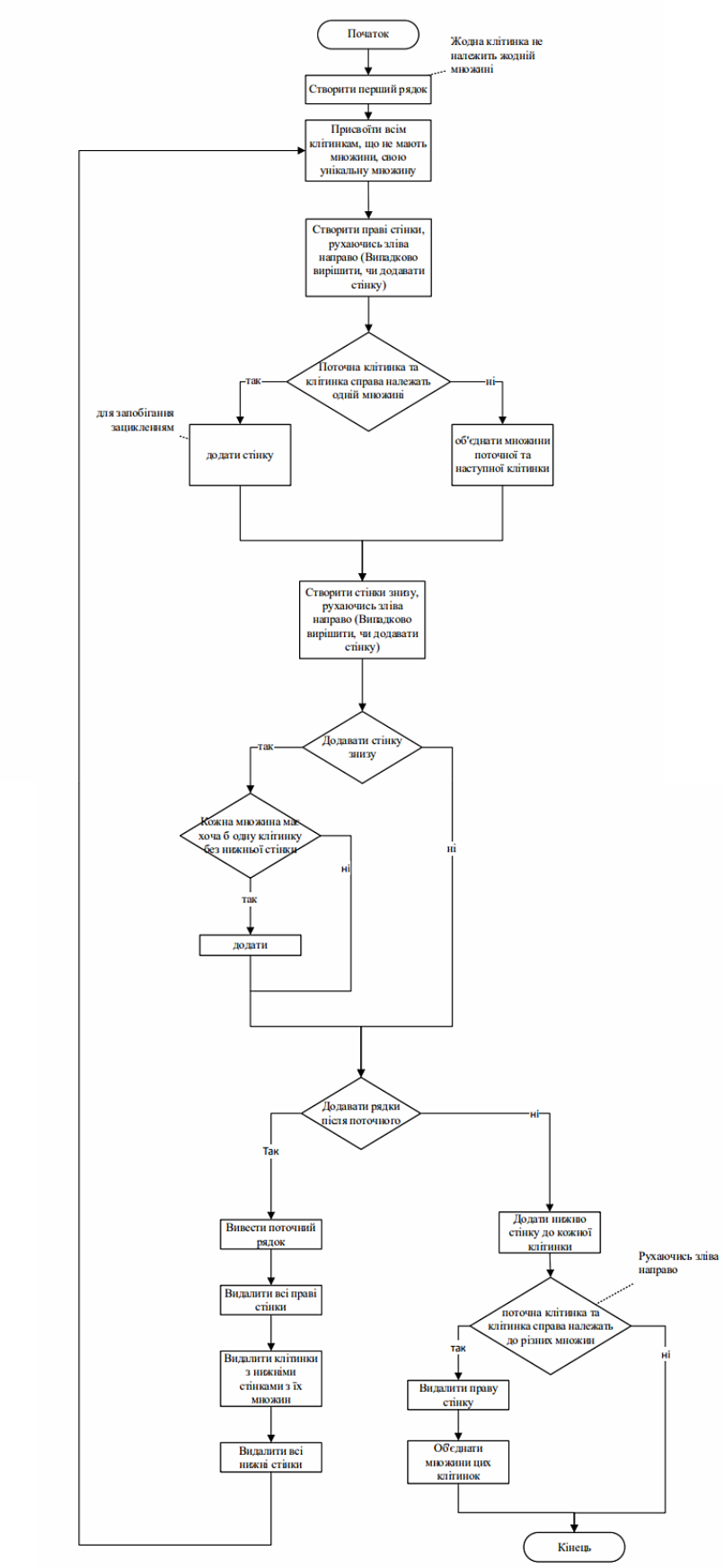



Рисунок 1.3-Блок-схема алгоритму Еллера

Алгоритм Еллера був винайдений Марліном Еллером у 1982 році. Він має деякі надзвичайні подібності з Sidewinder, але йому вдалося уникнути вражаючої упередженості цього алгоритму, включивши також деякі функції Крускала.Подібно до Sidewinder, він працює, розглядаючи лише один рядок за раз, створюючи набори (у стилі Крускала), щоб відстежувати, які комірки доступні з яких інших комірок.

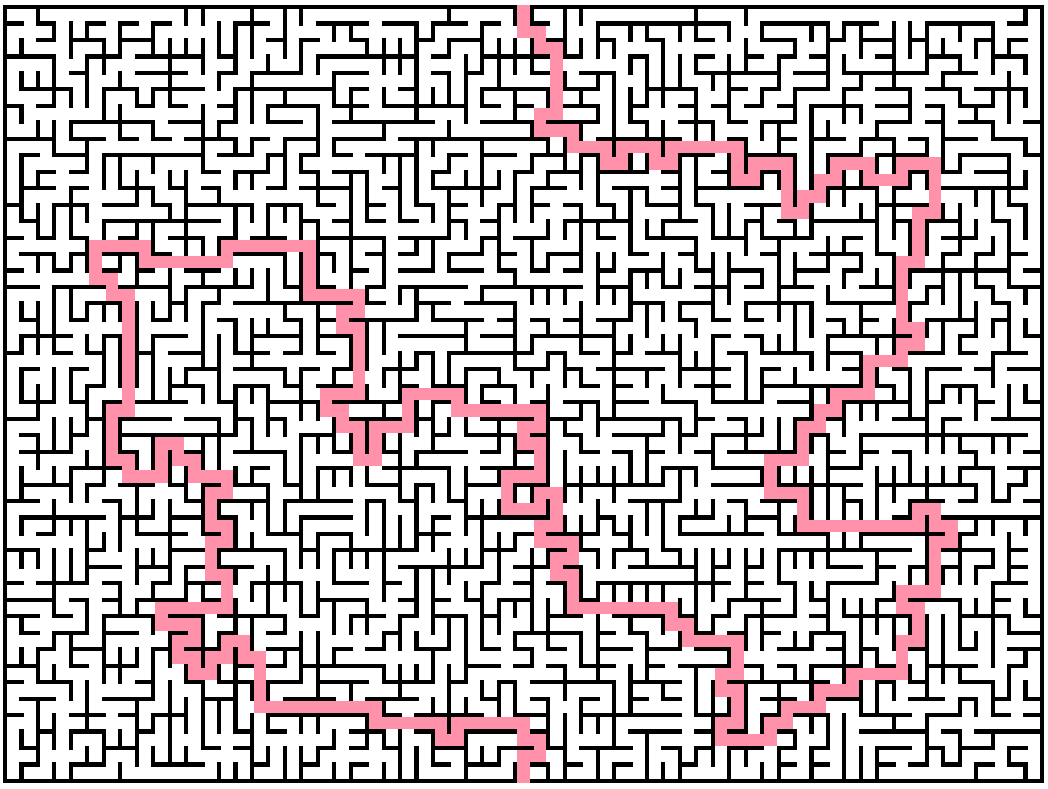


Рисунок 1.4-Лабіринт модіфікованою версією алгоритму Еллера

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 лабіринту. 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 (рис 1.4).

### 1.6 Методи пошуку шляху у лабіринті

Існує кілька методів пошуку шляху у лабіринті, кожен з яких має свої особливості. Ось детальний опис кількох з них:

**1.Пошук в ширину (Breadth-First Search, BFS)**:є простим алгоритмом пошуку, що працює на основі принципу "пошарового" просунення вглиб лабіринту.Починаючи з початкової точки, він спочатку досліджує всі можливі напрямки руху з цієї точки, а потім переходить до наступного "шару" точок.Цей процес продовжується, доки не буде знайдений шлях до цільової точки або поки всі доступні шляхи не будуть досліджені.

**2.Пошук у глибину (Depth-First Search, DFS)**:також працює на основі принципу "пошарового" просунення, але він просувається вглиб лабіринту наскрізь, поки не досягне кінцевої точки або поки не зустріне тупик.Він обирає один із доступних напрямків руху та йде до кінця, перевіряючи кожну наступну точку, поки не досягне тупика. Після цього він повертається на попередній рівень і обирає інший напрямок, продовжуючи процес.

**3.Алгоритм Дейкстри (Dijkstra's Algorithm)**: алгоритм на графах, винайдений нідерландським ученим Едсгер Дейкстрой в 1959 році. Знаходить найкоротші шляхи від однієї з вершин графа до решти. Алгоритм працює лише для графів без ребер негативної ваги. Алгоритм широко застосовується у програмуванні, наприклад, його використовують протоколи маршрутизації OSPF та IS-IS.

**4.Алгоритм A*(A-star Algorithm):** належить до евристичних алгоритмів пошуку. Використовується для пошуку найкоротшого шляху між двома вершинами графу з додатніми вагами ребер. Описаний 1968 р. Пітером Хартом, Нільсом Нільсоном та Бертрамом Рафаелєм.

**5. Алгоритм Лі:**алгоритм, що дозволяє знайти мінімальний шлях в графі з ребрами одиничної довжини. Заснований на алгоритмі пошуку в ширину. Застосовується для знаходження найкоротшого шляху в графі, в загальному випадку знаходить лише його довжину.Алгоритм також називають хвильовим алгоритмом

## 1.7.Висновки до розділу

У даному розділі було проаналізовано та описано основні аспекти лабіринтів, включаючи їх класифікації та структуру. Також було розглянуто метод теорії графів, який використовується для моделювання лабіринтів, що дозволяє ефективно вирішувати задачі пошуку шляху в них.

Основна увага була приділена алгоритмам і методам пошуку шляху, які широко використовуються для розв'язання завдань навігації в лабіринтах. Зокрема, були розглянуті такі алгоритми, як Пошук в ширину (BFS), Пошук у глибину (DFS), Алгоритм Дейкстри, Алгоритм A* та Алгоритм Лі.

За результатами аналізу можна зробити висновок, що кожен з наведених алгоритмів має свої переваги та обмеження, і вибір конкретного методу для конкретної задачі може залежати від різних факторів, таких як розмір лабіринту, кількість перешкод, точність шляху тощо.

Далі дослідження може бути спрямоване на розробку та оптимізацію алгоритмів для більш ефективного пошуку шляху в лабіринтах, а також на їх практичну реалізацію в різноманітних додаткових застосунках, таких як автономні роботи, відеоігри, системи навігації та інші.

# 2 ВИБІР МЕТОДУ ТА РОЗРОБКА МОДЕЛІ

## 2.1 Вибір алгоритму та методу вирішення задачі

Для досягнення мети роботи необхідно вибрати алгоритм,щоб побудувати генерацію лабіринту.З розглянутих у першому розділі алгоритмів з'ясувалося, що більшість з них є послідовними алгоритмами і лише алгоритм Еллера можливо використовувати в паралельних режимах обчислень завдяки використання динамічного програмування.Ось деякі додаткові фактори, які слід врахувати при виборі алгоритму:

**1.Розмір лабіринту:** Якщо ви хочете генерувати великі лабіринти, вам може знадобитися використовувати алгоритм, який є більш ефективним, наприклад, алгоритм Прима.

**2.Структура лабіринту:** Якщо ви хочете, щоб лабіринт мав багато розгалужень, вам може знадобитися використовувати алгоритм Глибинного пошуку.

**3.Особливі вимоги:** Якщо у вас є якісь особливі вимоги до лабіринту, наприклад, якщо він повинен мати певну форму або містити певні об'єкти, вам може знадобитися модифікувати існуючий алгоритм або розробити новий.

Ідея алгоритму Еллера в генерації лабіринтів полягає в тому, щоб розглядати лабіринт у вигляді рядків та магістралей і генерувати їх поетапно, об'єднуючи комірки лабіринту у такий спосіб, що буде забезпечена його цілісність, але при цьому він залишиться складним для проходження.

Основні принципи алгоритму Еллера включають в себе:

**1.Робота з рядками:**Лабіринт розглядається у вигляді рядків, де кожен рядок представляє собою горизонтальний ряд комірок.

**2.Об'єднання комірок у рядку:**Для кожного рядка створюються випадкові перегородки між його комірками, за винятком останньої комірки.Комірки об'єднуються випадковим чином у межах кожного рядка, створюючи "магістралі" для проходження.

**3.З'єднання рядків:**Після обробки кожного рядка з'єднуються деякі його комірки з комірками наступного рядка.Зв'язок між рядками створюється таким чином, щоб уникнути утворення циклів у лабіринті.

**4.Видалення перегородок:**Після з'єднання рядків деякі перегородки випадковим чином видаляються, забезпечуючи цілісність лабіринту та його зв'язність.

Цей підхід дозволяє генерувати лабіринти зі структурою, що має мало стін, але при цьому залишається складними для проходження. Крім того, він ефективно працює для автоматичної генерації лабіринтів у комп'ютерних програмах та іграх, де потрібно створювати нові рівні або генерувати випадкові структури.

Окрім вибору алгоритму  також потрібно буде вирішити, як представляти лабіринт у програмі. Один із поширених способів - використовувати двовимірний масив, де кожна комірка масиву представляє комірку в лабіринті. Стіни лабіринту можна потім представляти як значення в масиві.

Також можете використовувати інші структури даних, такі як списки або графи, для представлення лабіринту. Вибір структури даних залежатиме від ваших конкретних потреб і від того, як ви плануете використовувати лабіринт у своїй програмі.

Виходячи з вище сказаного, до нашого завдання більше підходить алгоритм Еллера завдяки своїй однорідній структурі розрахунків, Алгоритм Еллера є гарним вибором для задачі. Він простий у реалізації, може генерувати лабіринти різних форм і розмірів, і гарантує, що лабіринт буде мати єдиний шлях між двома точками.

## 2.2 Вибір інструментів для розробки програми

У наші часи існує широкий вибір технологій та інструментів розробки генерації лабіринтів.

Найпопулярнішими мовами є Java, Python, C++ та C#.Лише мова прогрмування Java залишається найбільш ефективним інструментом для

створення програм і генерація лабіринтів не є винятком.Ще одним інструментом розробки програми, з яким потрібно було визначитися. є інтегроване середовище розробки або IDE (Integrated Development Environment).Найпопулярнішим IDE для Java є на сьогодні-Eclipse, IntelliJ IDEA, NetBeans,Ruby та Processing.

Eclipse — безкоштовне модульне інтегроване середовище розробки програмного забезпечення, яке розробляється та підтримується Eclipse Foundation. Воно включає проєкти, такі як платформа Eclipse, набір інструментів для програмування на Java, системи контролю версій, засоби для створення графічних інтерфейсів та інше. Основна мова програмування для Eclipse — Java, але за допомогою різних плагінів його можна використовувати для розробки застосунків на багатьох інших мовах, включаючи Ada, C, C++, C#, COBOL, Fortran, Groovy, Haskell, JavaScript, Julia, Lua, Perl, PHP, Python, R, Ruby (включаючи Ruby on Rails), Scala, Clojure та Scheme. Серед інтегрованих середовищ розробки є Eclipse ADT (Ada Development Toolkit) для Ada, Eclipse CDT для C/C++, Eclipse JDT для Java та Eclipse PDT для PHP.

Ruby — це інтерпретована, повністю об'єктно-орієнтована мова програмування з динамічною типізацією. Вона відзначається високою ефективністю розробки програм і об'єднує в собі найкращі риси таких мов, як Perl, Java, Python, Smalltalk, Eiffel, Ada та Lisp. Ruby поєднує Perl-подібний синтаксис з об'єктно-орієнтованим підходом Smalltalk, а також запозичує деякі особливості з Python, Lisp, Dylan і CLU. Серед ключових можливостей Ruby — обробка винятків у стилі Java та Python, підтримка переозначення операторів, які є методами, і наявність автоматичного збирання сміття, яке працює для всіх об'єктів, включаючи зовнішні бібліотеки.

Processing — це безкоштовна графічна бібліотека та інтегроване середовище розробки (IDE), створені для електронного мистецтва, нових медіа та візуального дизайну, з метою навчання основам комп'ютерного програмування у візуальному контексті людей без досвіду в програмуванні.Processing базується на мові програмування Java, але пропонує додаткові спрощення, такі як нові класи та псевдонімні математичні функції та операції. Середовище також має графічний інтерфейс, який полегшує процес компіляції та виконання програм.

## 2.3 Генерація та формальний опис алгоритму Еллера

**Як відбувається генерація?**Алгоритм полягає у поступовому додаванні нових рядків до лабіринту. Кожен рядок складається з фіксованої кількості клітинок, що задається спочатку. Клітинки об'єднуються в множини, які визначають можливість проходу між ними.

Під час генерації поточного рядка клітинки однієї множини з'єднані між собою, тоді як клітинки з різних множин залишаються ізольованими. Кожна клітинка може мати або не мати праву та нижню стінку. Стінки генеруються випадковим чином, але за певними правилами, щоб уникнути утворення циклів у лабіринті.

**Формальний опис алгоритму**

1.Створити перший рядок, де жодна клітинка ще не належить до жодної множини.

2.Призначити кожній клітинці, яка не належить до множини, свою унікальну множину.

3.Генерація правих стінок, рухаючись зліва направо:

3.1.Випадкове рішення про додавання стінки:

3.1.1.Якщо поточна клітинка і клітинка справа належать одній множині, додати стінку (щоб уникнути циклів).

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

4.Генерація нижніх стінок, рухаючись зліва направо:Випадкове рішення про додавання стінки знизу. Переконатися, що кожна множина має хоча б одну клітинку без нижньої стінки, щоб уникнути ізольованих областей.

5.Вирішити, чи додавати нові рядки після поточного:

5.1.Якщо додається новий рядок:

5.1.1.Вивести поточний рядок.

5.1.2.Видалити всі праві стінки.

5.1.3.Видалити клітинки з нижніми стінками зі своїх множин.

5.1.4.Видалити всі нижні стінки.

5.1.5.Продовжити з кроку 2.

5.2.Якщо лабіринт завершено:

5.2.1Додати нижню стінку до кожної клітинки.

5.2.2.Рухаючись зліва направо,Якщо поточна клітинка і клітинка справа належать до різних множин, видалити праву стінку і об'єднати множини цих клітинок.

Процес створення лабіринту можна проілюструвати покроково. Наприклад, оберемо довжину рядка рівною 5. Перший рядок буде мати верхні стінки, останній — нижні, всі перші клітинки — ліві стінки, а останні в кожному рядку — праві стінки, щоб лабіринт був оточений стінками з усіх боків (рис 2.1).

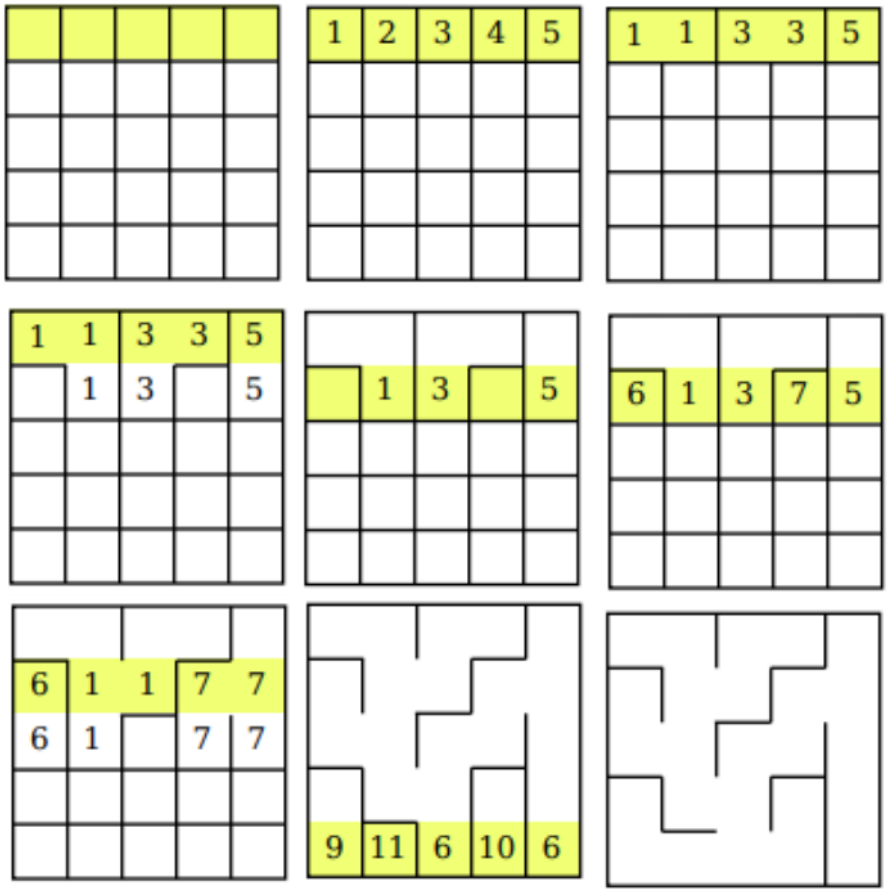



Рисунок 2.1-Побудова алгоритму Еллера

**Приклади лабіринтів**

Наступний лабіринт 15x15 було створено за допомогою цього алгоритму (рис 2.2):

Рисунок 2.2-Лабіринт розміром 15x15

Тут ми можемо змінити «текстуру» своїх лабіринтів, додавши зміщення до генератора випадкових чисел, щоб було більш чи менш імовірно, що ви створите праву стіну, ніж нижню. Наприклад, ви можете створити лабіринт із більш довгими горизонтальними проходами, створюючи праві стінки рідше та нижні стінки частіше.

Це дуже легко зробити. Коли ви вирішуєте додати стіну, зробіть щось на кшталт наступного: 1) згенеруйте випадкове число від 1 до 100. 2) якщо створюєте праву стіну, переконайтеся, що отримане число менше за наше значення зміщення, якщо створюєте нижню – стіни, перевірте, чи це число перевищує наше значення зсуву. У цьому випадку зміщення 50 створить рівну текстуру, як у прикладі лабіринту вище.

Наступні два приклади лабіринтів показують, як додавання зміщення впливає на кінцеву текстуру лабіринту (рис 2.3, рис 2.4).

Рисунок 2.3-Більш вертикальний(зліва) та білш горизонтальний(справа)лабіринти

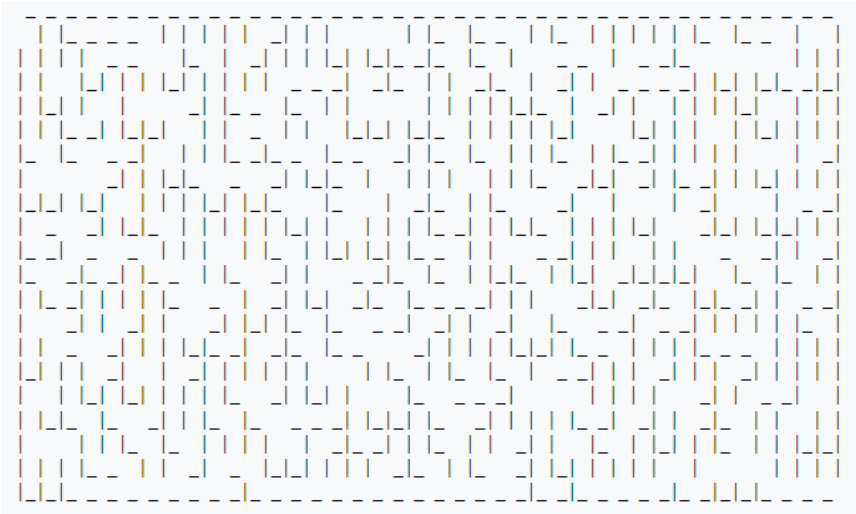


Рисунок 2.4-Приклад згенерованого лабіринту алгоритмом Еллера розміром 40x20 клітинок

### 2.4 Використання методу пошуку A*

Алгоритм A* (читається як «А зірочка» або англійською «A star») є евристичним алгоритмом пошуку, призначеним для знаходження найкоротшого шляху між двома вершинами графа з позитивними вагами ребер. Він був описаний у 1968 році Пітером Хартом, Нільсом Нільсоном та Бертрамом Рафаелем.

Цей алгоритм використовує допоміжну функцію (евристику) для спрямування пошуку, що дозволяє зменшити час пошуку. Алгоритм завжди знаходить оптимальне рішення, якщо воно існує.

A* пріоритетно обирає вершини, які ймовірніше ведуть до найкоротшого шляху до мети. Для кожної відомої вершини x обчислюється значення f(x), яке відповідає довжині найкоротшого шляху від початкової вершини до кінцевої, що проходить через вершину x. Вершини з найменшим значенням f обробляються першими.

Функція f(x) визначається як:

$f(x)=g(x)+h(x)$
де g(x) — це функція, яка визначає вартість шляху від початкової вершини до x, а h(x) — евристична функція, що оцінює вартість шляху від вершини x до кінцевої вершини.

Евристика повинна давати недооцінку вартості шляху. Прикладом може служити відстань по прямій: загальний шлях не може бути коротшим за пряму лінію.

## 2.5 Принцип роботи пошуку A*

Алгоритм поділяє вершини на три категорії:

**Невідомі вершини:** ці вершини ще не були знайдені, і шлях до них невідомий. На початку алгоритму всі вершини, крім початкової, належать до цього класу.

**Відомі вершини (OpenList):** до цих вершин вже відомий (але можливо не оптимальний) шлях. Всі відомі вершини разом зі значеннями f зберігаються в списку. З цього списку обираються найперспективніші вершини. Реалізація цього списку значно впливає на швидкодію алгоритму і зазвичай має вигляд черги з пріоритетом (наприклад, бінарна купа). На початку алгоритму до відомих вершин належить тільки початкова вершина.

**Повністю досліджені вершини (ClosedList):** для цих вершин вже відомий найкоротший шлях. Вони додаються до замкненого списку, щоб уникнути повторного дослідження. Спочатку цей список порожній.

Кожна відома або повністю досліджена вершина має вказівник на попередню вершину, що дозволяє відтворити шлях від кінцевої до початкової вершини.

Коли вершину x повністю досліджено, суміжні з нею вершини додаються до списку відомих, а сама вершина переходить до списку повністю досліджених. Вказівники на попередню вершину встановлюються на x. Суміжні вершини, що вже знаходяться у списку повністю досліджених, не додаються до списку відомих, а зворотні вказівники не змінюються. Якщо суміжні вершини вже є в списку відомих, їх значення f та вказівники оновлюються, якщо новий шлях коротший за вже відомий.

Алгоритм завершується, коли кінцева вершина потрапляє до списку повністю досліджених. Шлях відтворюється за допомогою вказівників на попередні вершини. Якщо список відомих вершин спорожніє до досягнення мети, алгоритм припиняє пошук, оскільки рішення не існує (рис 2.5).

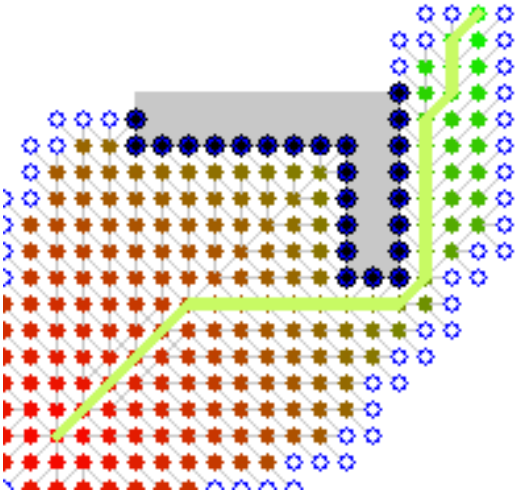


Рисунок 2.5-Ілюстрація пошуку A* на прикладі задачі планування шляху(https://ru.wikipedia.org/wiki/A*)

## 2.6 Висновки до розділу

У розділі були розглянути методи реалізації генерації лабіринтів та існуючі інструменти для виконання,як Eclipse,Ruby Proccessing.Обрано програми як найбільш відповідний розв'язуваному завданню, розглянуті його функції які можуть бути використанні в роботі. Та обрано інструменти розробки програми.

# 3 РОЗРОБКА ГЕНЕРАЦІЇ ЛАБІРИНТУ В ПРОГРАМАХ ТА ПОШУКУ ШЛЯХУ A*

## 3.1 Розробка функції алгоритму Еллера в лабіринті

Основна ідея алгоритму Еллера полягає у використанні "рядків" і "магістралей" для створення лабіринту. В процесі він обходить лабіринт вирішуючи, які комірки з'єднувати, і на якій стадії вони з'єднуються.

Цей алгоритм дозволяє створювати лабіринти з варіативною структурою, але з мінімальною кількістю стін, що робить їх ефективними для використання у відеоіграх, головоломках та інших сферах, де потрібно генерувати лабіринти (рис 3.1).

```
def step(state, finish=false)
  connected_sets = []
  connected_set = [0]
  # ---
  # create the set of horizontally connected corridors in this row
  # ---
  (state.width-1).times do |c|
    if state.same?(c, c+1) || (!finish && rand(2) > 0)
      # cells are not joined by a passage, so we start a new connected set
      connected_sets << connected_set
      connected_set = [c+1]
    else
      state.merge(c, c+1)
      connected_set << c+1
    end
  end
  connected_sets << connected_set
  # ---
  # create the set of vertically connected corridors from this row, but
  # only if this is not the last row
  # ---
  verticals = []
  next_state = state.next
  unless finish
    state.each_set do |id, set|
      cells_to_connect = set.sort_by { rand }[0, 1 + rand(set.length-1)]
      verticals.concat(cells_to_connect)
      cells_to_connect.each { |cell| next_state.add(cell, id) }
    end
  end
  # ---
  # translate the connected sets and verticals into a bitmap that can be
  # returned and displayed
  # ---
  row = []
  connected_sets.each do |connected_set|
    connected_set.each_with_index do |cell, index|
      last = (index+1 == connected_set.length)
      map = last ? 0 : E
      map |= S if verticals.include?(cell)
      row << map
    end
  end
  [next_state.populate, row]
end
```

Рисунок 3.1-Псевдокод алгоритму Еллера в Ruby

Алгоритм Еллера використовується для генерації лабіринтів у багатьох контекстах, таких як в іграх, де потрібно створювати нові рівні або у графічних програмах для візуалізації структур даних. Його особливість полягає у тому, що він генерує лабіринти зі структурою, що має мало стін, але при цьому

залишає достатньо складності для розв'язання головоломки або створення цікавого геймплею.

### 3.2 Розробка функції алгоритму пошуку A* в лабіринті

Знайдений шлях, відтворений за допомогою зворотних вказівників, йде від кінцевої вершини до початкової. Щоб одразу отримати шлях у правильному напрямку, від початкової вершини до кінцевої, можна поміняти місцями початок і кінець задачі. Якщо ж здійснювати пошук, починаючи з кінцевої вершини, отриманий список починатиметься з початкової вершини і прямуватиме до кінцевої.

Алгоритм пошуку A* можна описати у вигляді псевдокоду (рис 3.2):

```
program a-star
    // Ініціалізація списку відомих вершин, список досліджених порожній
    // (f-значення початкової вершини відсутнє)
    openlist.enqueue(startknote, 0)
    // цей шлях буде пройдений доки:
    // - буде знайдено оптимальний розв'язок або
    // - встановлено, що розв'язків не існує
    repeat
        // Вилучити вершину з найменшим f-значенням
        currentNode := openlist.removeMin()
        // Досягнута кінцева вершина?
        if currentNode == endknote then
            return PathFound
        // Якщо кінцева вершина не досягнута: додати суміжні
        // до поточної вершини в список відомих
        expandNode(currentNode)
        // поточна вершина вже повністю досліджена
        closedlist.add(currentNode)
    until openlist.isEmpty()
    // список відомих порожній, розв'язків не існує
    return NoPathFound
end

// перевіряє суміжні вершини та додає до списку відомих якщо:
// - суміжні вершини зустрічаються вперше або
// - знайдений кращий шлях до цієї вершини
function expandNode(currentNode)
    foreach successor of currentNode
        // пропустити, якщо вершина вже є списку досліджених
        if closedlist.contains(successor) then
            continue
        // обчислити значення g нового шляху:
        // значення g попередньої вершини + вартість щойно пройденого ребра
        tentative_g = g(currentNode) + c(currentNode, successor)
        // якщо суміжна вершина вже в списку відомих,
        //  але знайдений шлях не кращий за вже відомий - пропустити
        if openlist.contains(successor) and tentative_g >= g[successor] then
            continue
        // встановити вказівник на попередню вершину та зберегти g
        successor.predecessor := currentNode
        g[successor] = tentative_g
        // оновити значення f вершини у списку відомих
        // або додати вершину до списку відомих
        f := tentative_g + h(successor)
        if openlist.contains(successor) then
            openlist.decreaseKey(successor, f)
        else
            openlist.enqueue(successor, f)
    end
end
```

Рисунок 3.2-Псевдокод алгоритму пошуку A*(https://ru.wikipedia.org/wiki/A*)

### 3.3 Отримані результати виконання програми

В результаті активації коду програми в Eclipse в пакеті check і активувати Main.java.,то з’явиться вікно програми,де відкривається вікно і генерує стінки лабіринту,де кожна відмічена клітинка позначається точкою і навколо неї робиться стінка і так для кожної клитини до тих пір,поки увесь простір не заповниться клітигкою і на цьому побудова закінчується (рис 3.3)

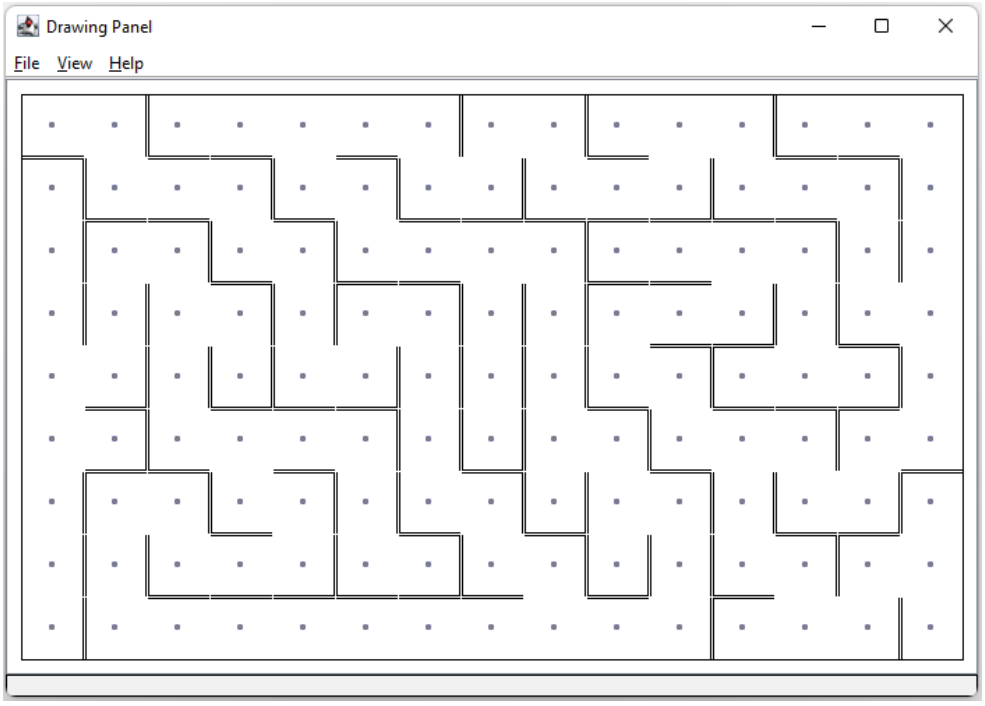



Рисунок 3.3-Згенерований лабіринт у Eclipse

В результаті активації коду програми на Ruby на командній строці буде генеруватися дуже довгий лабіринт з згенерованими стінками і буде будувати до тих пір,поки ти не натиснеш конкретну комбінацію клавіш,щоб це зупинити.В моєму випадку-це комбінація клавіш CTRL+C і головне-наптиснути якомога швидше,бо він будує лабіринт по вертикальній основі (рис 3.4).


Рисунок 3.4-Згенерований лабіринт ellersalgorithm.rb у командній строці Ruby

### 3.4.Результат генерації лабіринту з пошуком A*

В якості бонусу я зробила генерацію лабіринту в Proccesing ,який в результаті не тільки генерує лабіринт,але і після генерації автоматично ставить старт і фініш і автоматично за допомогою методом тику шукати шлях від початку до кінця.

В результаті виконання команди генерації лабіринту на Processing при запуску програми відкривається вікно,де квадратик генерує із кожної клітини стінку лабіринту від початку до кінця,залежно від розміру вікна,де генерується лабіринт,розміру клітинок,швидкості генерації лабіринту і так далі.Весь процес команди генерування лабіринту разом з блоком показується на консолі (рис 3.5)

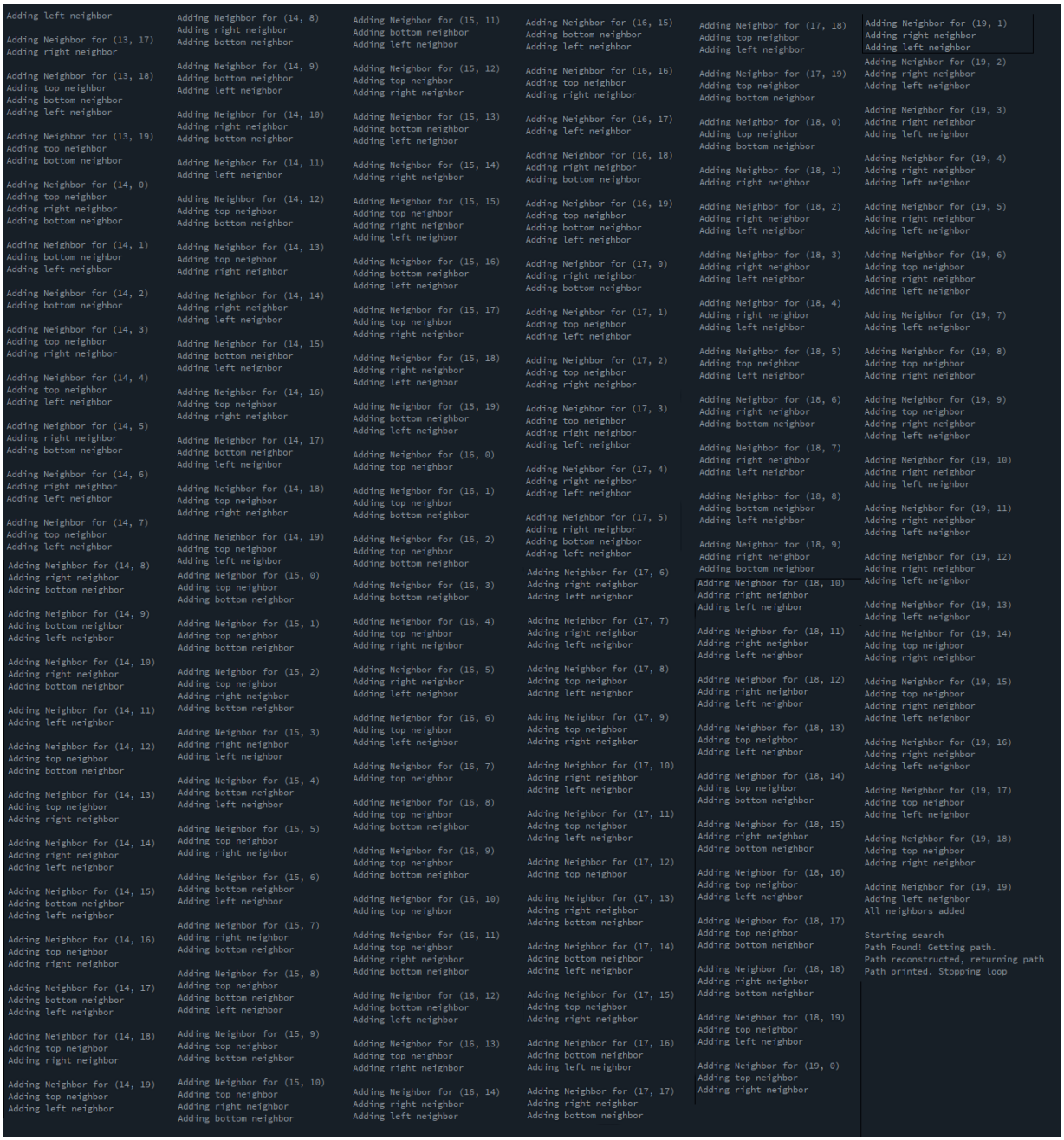



Рисунок 3.5-Консоль завершенного проектування лабіринту в Processing

Сам консоль в процесі,коли він генерує лабіринт в окремому вікні і активує алгоритм пошуку A*,він пише конкретну клітину,координати X та Y та повідомляє,що знайдений сусід(тобто конкретна клітинка) та вибрана клітинка,а коли блок(основна клітинка,який генерує стінки лабіринту) вже підмічений,то він повідомляє,що ви раніше були тут і треба знайти іншу клітинку для генерування стінки лабіринту (рис 3.6).

```
Choosing neighbors for (16, 7)
Found a neighbor!
Picked: (16, 16)
X Distance: 0
Y Distance: 1


Choosing neighbors for (16, 6)
Found a neighbor!
Picked: (16, 16)
This has been visited before, picking a new one.
Picked: (17, 17)
X Distance: -1
Y Distance: 0


Choosing neighbors for (17, 6)
Found a neighbor!
Picked: (17, 17)
This has been visited before, picking a new one.
Picked: (16, 16)
This has been visited before, picking a new one.
Picked: (18, 18)
X Distance: -1
Y Distance: 0


Choosing neighbors for (18, 6)
Found a neighbor!
Picked: (18, 18)
X Distance: 0
Y Distance: -1
```

Рисунок 3.6-Фрагмент консолі генерування частини лабіринту в процессі

І коли генерація усіх клітинок в стінки лабіринту завершується,блок повертається до початку шляху і включає цикл,який спочатку відмічає старт та фініш,а потім генерує шлях,який шукає шлях від початку до кінця,перевираючи варіанти.Коли шлях знайдено,результат в кінці позначається червоним шляхом і цикл закінчується (рис 3.7)

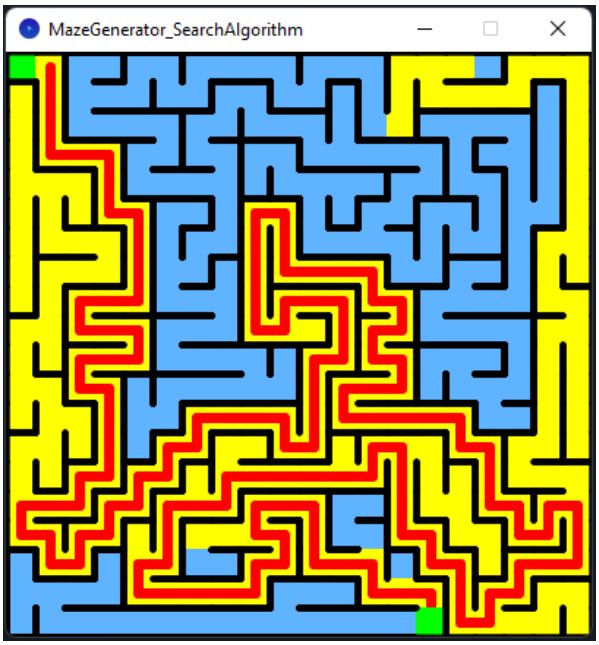



Рисунок 3.7-.Згенерований лабіринт та генерування пошуку A* в программі Proccesing

## 3.5 Висновки до розділу

В цьому розділі було показано розробка лабіринтів на трьох програмах на програмуванні Java-программу Eclipse,який при запуску генерує лабіринт де навколо точки генерує стінку для лабіринту і так до тих пір,поки не завершиться результат, програма Ruby з командами генерації лабіринту за допомогою додаткової програми Sublime Text і командою зупинки за допомогою комбінації клавіщ,щоб зупинити роботу,а також створення генерації лабіринту в програмі Processing,який не тільки самостійно генерує лабіринт,а ще й виставляє початок і кінець і будує шлях завдяки алгоритму пошуку A*.

# ВИСНОВКИ

1. В роботі проведено аналіз найпоширеніших алгоритмів генерації лабіринтів і алгоритмів пошуку шляху та визначено, що тільки один з них можна декомпозувати для використання у паралельному програмуванні.

2. Вивчено деякі елементи генерації алгоритму Еллера,а також алгоритм пошуку A* і вони є досить ефективними алгоритмами створення в программах генерації лабіринтів.Вони дозволяють не тільки створювати побудову ронадомної генерації на комп'ютері,а і в якості бонусу створювати пошуку входу і виходу методом алгоритму A*.

3. Розроблені програми показали успішність застосування динамічного алгоритму Еллера для генерації лабіринту і пошуку A*.

4. Проведене дослідження швидкості виконання паралельних обчислень показало, що приріст швидкості вирахувань на одній комп'ютерній системі не має прямої залежності від кількості паралельних процесів, тому що залежить від способів реалізації віртуалізації на конкретній операційній системі.

5. Загалом робота показала процес генерації лабіринтів.Але вимагає додаткових командних строк і бібліотек,щоб отримати кращі результати у генеруванні лабіринтів у різних програмах,а також пошуку шляху у дійсності.

# ПЕРЕЛІК ПОСИЛАНЬ


**1.Алгоритм створення лабіринту-Вікіпедія[Електронний ресурс]:** https://uk.wikipedia.org/wiki/Алгоритм_створення_лабіринту

**2.Класичні методи створення лабіринтів. Частина 1:Вступ(російською) [Електронний ресурс]:** https://habr.com/ru/articles/320140/

**3.Класичні алгоритми створення лабіринтів. Частина 2:Занурення у випадковість(російською) [Електронний ресурс]:** https://habr.com/ru/articles/321210/

**4.Генерація лабіринтів: алгоритм Еллера(російською) [Електронний ресурс]:**https://habr.com/ru/articles/667576/

**5. Введення у алгоритм A*(російською) [Електронний ресурс]:** https://habr.com/ru/articles/331192/

**6.Maze Generator and A* Search Algorithm using Java(Частина 1):** https://www.youtube.com/watch?v=zsG2ceOlY6I

**7.Path Finding Using A* Search Algorithm(Частина 2):** https://www.youtube.com/watch?v=_KEzW50Nhyw

**8.Алгоритм Еллера-Вікіпедія**[Електронний ресурс]**:** https://uk.wikipedia.org/wiki/Алгоритм_Еллера_генерації_лабіринтів

**9.Алгоритм пошуку A*(Російською та українською)-Вікіпедія[Електронний ресурс]:**
https://uk.wikipedia.org/wiki/Алгоритм_пошуку_A*
https://ru.wikipedia.org/wiki/A*

**10."Mazes for Programmers: Code Your Own Twisty Little Passages" by Jamis Buck:**
Buck, Jamis. Mazes for Programmers: Code Your Own Twisty Little Passages. The Pragmatic Programmers, LLC,2015

**11.Think Labyrinth: Maze Algorithms[Електронний ресурс]:**https://www.astrolog.org/labyrnth/algrithm.htm

**12.Лабіринти: класифікація, генерування, пошук рішень**[Електронний ресурс]**:**https://habr.com/ru/articles/445378/

# Додаток А

## Cell.java

```
package cells;
public class Cell {
	//WALLS
	public boolean top;
	public boolean bot;
	public boolean left;
	public boolean right;
	//IS CELLVISITED
	public boolean visited;
	//DEFAULT INIT OF CELL
	public Cell() {
		this(false, false);
	}

	//THIS CONSTRUCTOR CREATE ONLY FILLED OR ONLY EMPTY MAZE
	public Cell(boolean initState, boolean visited) {
		this(initState, initState, initState, initState, visited);
	}

	//THIS CONSTRUCTOR ALLOW TO CUSTOMIZE YOUR OWN RULES
	public Cell(boolean top, boolean bot, boolean left, boolean right,
boolean visited) {
		this.top = top;
		this.bot = bot;
		this.left = left;
		this.right = right;
		this.visited = visited;
}

	@Override
	public String toString() {
		return String.format(" cell has %b, %b, %b, %b. ", this.top,
this.bot, this.left, this.right);
	}
}
```

## Main.java

```
package check;

import mazes.Maze;

public class Main {

	public static void main(String[] args) {

		Maze maze = new Maze(15, 9);

		maze.buildMaze(0, 0, 0, 0); // Start building maze from (0, 0)
coords

	}

}
```

## Maze.java

```
package mazes;
import java.awt.Color;
import java.awt.Graphics2D;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
import cells.Cell;
import draw.DrawingPanel;

public class Maze {
	//MAZE
	public Cell[][] maze;
	//SIZE OF MAZE
	public int rows;
	public int cols;
	//IS MAZE COMPLETED
	public boolean completed;
	//RANDOMIZER FOR SIDES
	Random r = new Random();
	//SHIFT FROM BORDER
	int padding = 10;
	//SIZE OF CELL
	int side = 45;
	//GENERAL SIZE OF MAZE
	int tableX, tableY;
	//PANEL AND GRAPHICS FOR IT
	DrawingPanel panel;
	Graphics2D g;
	//CREATE DEFAULT 7X7 MAZE
	public Maze() {
		this(7, 7);
	}
	//CREATE YOUR OWN MAZE
	public Maze(int rows, int cols) {
		this.rows = rows;
		this.cols = cols;
		maze = new Cell[rows][cols];//Creates new maze with given size

		for (int i = 0; i < rows; i++) {
			for (int j = 0; j < cols; j++) {
				maze[i][j] = new Cell(true, false);//Leave maze empty
and unvisited
			}
		}

		buildBorders();//Build outer borders

		tableX = rows * side;
		tableY = cols * side;
		panel = new DrawingPanel(tableX + padding * 2, tableY + padding
* 2);//Create canvas with given size
		g = panel.getGraphics();//Get painter tool of canvas for
painting
		drawBorders();//Draw borders on canvas
	}
	//CREATE BORDERS
```

```
	void buildBorders() {
		for (int i = 0; i < rows; i++) {
			for (int j = 0; j < cols; j++) {
				if (j == 0)
					maze[i][j].top = false;
				if (j == cols - 1)
					maze[i][j].bot = false;
				if (i == 0)
					maze[i][j].left = false;
				if (i == rows - 1)
					maze[i][j].right = false;
			}
		}
	}

	//RECURSIVE BUILDER OF MAZE
	public void buildMaze(int x, int y, int px, int py) {
		int l = padding + x * side + 1;                  //left side of
cell

		int t = padding + y * side + 1;                  //top side of
cell
		int r = padding + x * side + (side - 1);   //right side of cell
		int b = padding + y * side + (side - 1);   //bottom side of cell
		g.setColor(Color.blue);
		g.fillOval(padding + x * side + side / 2 - 2, padding + y * side
+ side / 2 - 2, 4, 4);//Marking current cell
		g.setColor(Color.black);
		maze[x][y].visited = true;//Visit current cell
		Integer[] dirs = generateRandomDirections();//Choose all
directions in random order
		for (Integer dir : dirs) {

			switch (dir) {//Check direction
			case 1:                                  //Top direction
				if (y - 1 < 0)              //If border
					continue;

				if (x==px && y-1==py) //If previous cell
					continue;

				if (maze[x][y - 1].visited == false) {//If next cell
is unvisited
					buildMaze(x, y - 1, x, y);
				} else {
					maze[x][y].top = false;
					g.drawLine(l, t, r, t);
					maze[x][y - 1].bot = false;
					g.drawLine(l, t - 2, r, t - 2);

				}
				panel.sleep(100);
				break;
			case 2:                                  //Bottom direction
				if (y + 1 >= cols)          //If border
					continue;

				if (x==px && y+1==py) //If previous cell
```

```
                    continue;

                if (maze[x][y + 1].visited == false) {//If next cell
is unvisited
                    buildMaze(x, y + 1, x, y);
                } else {
                    maze[x][y].bot = false;
                    g.drawLine(r, b, l, b);
                    maze[x][y + 1].top = false;
                    g.drawLine(r, b + 2, l, b + 2);

                }
                panel.sleep(100);
                break;
            case 3:                                 //Left direction
                if (x - 1 < 0)              //If border
                    continue;
                if (x-1==px && y==py) //If previous cell
                    continue;

                if (maze[x - 1][y].visited == false) {//If next cell
is unvisited
                    buildMaze(x - 1, y, x, y);
                } else {
                    maze[x][y].left = false;
                    g.drawLine(l, b, l, t);
                    maze[x - 1][y].right = false;
                    g.drawLine(l - 2, b, l - 2, t);

                }
                panel.sleep(100);
                break;
            case 4:                                 //Right direction
                if (x + 1 >= rows)          //If border
                    continue;

                if (x+1==px && y==py) //If previous cell
                    continue;

                if (maze[x + 1][y].visited == false) {//If next cell
is unvisited
                    buildMaze(x + 1, y, x, y);
                } else {
                    maze[x][y].right = false;
                    g.drawLine(r, t, r, b);
                    maze[x + 1][y].left = false;
                    g.drawLine(r + 2, t, r + 2, b);
                }
                panel.sleep(100);
                break;
            }
        }



        g.setColor(Color.gray);
        g.fillOval(padding + x * side + side / 2 - 2, padding + y * side
+ side / 2 - 2, 4, 4);
```

```
            g.setColor(Color.black);
      }
      //ALL FOUR DIRECTIONS SHUFFLES IN RANDOM ORDER
      public Integer[] generateRandomDirections() {
            ArrayList<Integer> randoms = new ArrayList<Integer>();
            for (int i = 0; i < 4; i++)
                  randoms.add(i + 1);
            Collections.shuffle(randoms);

            return randoms.toArray(new Integer[4]);
      }
      //SHOWS MAZE STATS IN CONSOLE
      public void showMaze() {
            for (int i = 0; i < rows; i++) {
                  for (int j = 0; j < cols; j++) {
                        System.out.print("Number [" + i + " " + j + "]" +
maze[i][j].toString());
                  }
                  System.out.println();
            }
      }
      //DRAW BORDERS ON CANVAS
      public void drawBorders() {
            int l, t, r, b;
            for(int x = 0; x < rows; x++) {
                  for(int y = 0; y < cols; y++) {
                        l = padding + x * side;
                        t = padding + y * side;
                        r = padding + x * side + side;
                        b = padding + y * side + side;
                        if(!maze[x][y].top)
                              g.drawLine(l, t, r, t);
                        if(!maze[x][y].bot)
                              g.drawLine(l, b, r, b);
                        if(!maze[x][y].left)
                              g.drawLine(l, t, l, b);
                        if(!maze[x][y].right)
                              g.drawLine(r, t, r, b);
                  }
            }
      }
}
```

## Додаток Б

### Ellers.rb

```
class Ellers
  class RowState
    def initialize(starting_set=0)
      @cells_in_set = {}
      @set_for_cell = []
      @next_set = starting_set
    end

    def record(set, cell)
      @set_for_cell[cell.column] = set
```

```
    @cells_in_set[set] = [] if !@cells_in_set[set]
    @cells_in_set[set].push cell
  end

  def set_for(cell)
    if !@set_for_cell[cell.column]
      record(@next_set, cell)
      @next_set += 1
    end

    @set_for_cell[cell.column]
  end

  def merge(winner, loser)
    @cells_in_set[loser].each do |cell|
      @set_for_cell[cell.column] = winner
      @cells_in_set[winner].push cell
    end

    @cells_in_set.delete(loser)
  end

  def next
    RowState.new(@next_set)
  end

  def each_set
    @cells_in_set.each { |set, cells| yield set, cells }
    Self
  end
end

def self.on(grid)
  row_state = RowState.new

  grid.each_row do |row|
    row.each do |cell|
      next unless cell.west

      set = row_state.set_for(cell)
      prior_set = row_state.set_for(cell.west)

      should_link = set != prior_set &&
                    (cell.south.nil? || rand(2) == 0)

      if should_link
        cell.link(cell.west)
        row_state.merge(prior_set, set)
      end
    end

    if row[0].south
      next_row = row_state.next

      row_state.each_set do |set, list|
        list.shuffle.each_with_index do |cell, index|
          if index == 0 || rand(3) == 0
            cell.link(cell.south)
```

```
              next_row.record(row_state.set_for(cell), cell.south)
            end
          end
        end

        row_state = next_row
      end
    end
  end

end
```

ellers_demo.rb

```
require 'ellers'
require 'grid'

grid = Grid.new(20, 20)
Ellers.on(grid)

filename = "ellers.png"
grid.to_png.save(filename)
puts "saved to #{filename}"
```

## ellersalgorithm.rb

```
# ------------------------------------------------------------------------
# 1. Allow the maze to be customized via command-line parameters
# ------------------------------------------------------------------------
width  = (ARGV[0] || 10).to_i
height = (ARGV[1] || "-") == "-" ? nil : ARGV[1].to_i
seed   = (ARGV[2] || rand(0xFFFF_FFFF)).to_i

srand(seed)

# ------------------------------------------------------------------------
# 2. Set up constants to aid with describing the passage directions
# ------------------------------------------------------------------------

S, E = 1, 2

# ------------------------------------------------------------------------
# 3. Data structures to assist the algorithm
# ------------------------------------------------------------------------

class State
  attr_reader :width

  def initialize(width, next_set=-1)
    @width = width
    @next_set = next_set
    @sets = Hash.new { |h,k| h[k] = [] }
    @cells = {}
  end

  def next
    State.new(width, @next_set)
  end
```

```
  def populate
    width.times do |cell|
      unless @cells[cell]
        set = @next_set += 1
        @sets[set] << cell
        @cells[cell] = set
      end
    end

    self
  end

  def merge(sink_cell, target_cell)
    sink, target = @cells[sink_cell], @cells[target_cell]

    @sets[sink].concat(@sets[target])
    @sets[target].each { |cell| @cells[cell] = sink }
    @sets.delete(target)
  end

  def same?(cell1, cell2)
    @cells[cell1] == @cells[cell2]
  end

  def add(cell, set)
    @cells[cell] = set
    @sets[set] << cell
    self
  end

  def each_set
    @sets.each do |id, set|
      yield id, set
    end
  end
end

def row2str(row, last=false)
  # the \r makes sure we always start at the beginning of the line, even
after
  # ctrl-C (which prints "^C" to the console)
  s = "\r|"

  row.each_with_index do |cell, index|
    south = (cell & S != 0)
    next_south = (row[index+1] && row[index+1] & S != 0)
    east = (cell & E != 0)

    s << (south ? " " : "_")

    if east
      s << ((south || next_south) ? " " : "_")
    else
      s << "|"
    end
  end

  return s
```

```
end

state = State.new(width).populate
row_count = 0

# --------------------------------------------------------------------
# 4. Eller's algorithm
# --------------------------------------------------------------------

def step(state, finish=false)
  connected_sets = []
  connected_set = [0]

  # ---
  # create the set of horizontally connected corridors in this row
  # ---

  (state.width-1).times do |c|
    if state.same?(c, c+1) || (!finish && rand(2) > 0)
      # cells are not joined by a passage, so we start a new connected set
      connected_sets << connected_set
      connected_set = [c+1]
    else
      state.merge(c, c+1)
      connected_set << c+1
    end
  end

  connected_sets << connected_set

  # ---
  # create the set of vertically connected corridors from this row, but
  # only if this is not the last row
  # ---

  verticals = []
  next_state = state.next

  unless finish
    state.each_set do |id, set|
      cells_to_connect = set.sort_by { rand }[0, 1 + rand(set.length-1)]
      verticals.concat(cells_to_connect)
      cells_to_connect.each { |cell| next_state.add(cell, id) }
    end
  end

  # ---
  # translate the connected sets and verticals into a bitmap that can be
  # returned and displayed
  # ---

  row = []
  connected_sets.each do |connected_set|
    connected_set.each_with_index do |cell, index|
      last = (index+1 == connected_set.length)
      map = last ? 0 : E
      map |= S if verticals.include?(cell)
      row << map
```

```
    end
  end

  [next_state.populate, row]
end

# ---
# allow ctrl-c to stop the program gracefully, letting the final row
# be generated and displayed before aborting.
# ---

spinning = true
trap("INT") { spinning = false }

puts " " + "_" * (width * 2 - 1)

while spinning
  state, row = step(state)
  row_count += 1
  puts row2str(row)
  spinning = row_count+1 < height if height
end

state, row = step(state, true)
row_count += 1

puts row2str(row)

# ---------------------------------------------------------------------
# 5. Show the parameters used to build this maze, for repeatability
# ---------------------------------------------------------------------

puts "#{$0} #{width} #{row_count} #{seed}"
```

## Додаток В

## MazeGenerator_SearchAlgorithm.java

```
//global variables
int cols;
int rows;
int size = 25;
Block[][] blocks;
//global variables for maze
Block currentMazeBlock;
ArrayList<Block> mazeStack = new ArrayList<Block>();
boolean isMazeFinished = false;
//global variables for search algorithm
Block currentSearchBlock;
Block startSearchBlock;
Block finishSearchBlock;
ArrayList<Block> actualPath = new ArrayList<Block>();
ArrayList<Block> searchedPath = new ArrayList<Block>();
boolean searchNeighborsAdded = false;
ArrayList<Block> openSet = new ArrayList<Block>();
boolean pathFound = false;
```

```
void setup() {
  size(400, 400);
  rows = floor(height / size);
  cols = floor(width / size);
  blocks = new Block[rows][cols];

  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      blocks[i][j] = new Block(i, j);
    }
  }

  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      blocks[i][j].addNeighbors();
    }
  }
  currentMazeBlock = blocks[0][0];
  currentMazeBlock.visitedByMaze = true;

  startSearchBlock = blocks[0][0];
  finishSearchBlock = blocks[parseInt(random(0, rows - 1))][cols - 1];

  frameRate(60);
}

void draw() {
  if (!isMazeFinished) { //Maze Generator Here
    background(0, 255, 255);
    //background(0);
    strokeWeight(4);
    stroke(255, 255, 0);
    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
        blocks[i][j].show();
      }
    }
    fill(193, 50, 193);
    rect(currentMazeBlock.x, currentMazeBlock.y, size, size);

    if (currentMazeBlock.hasUnvisitedNeightbors()) {
      Block nextCurrent = currentMazeBlock.pickRandomNeighbor();
      mazeStack.add(currentMazeBlock);
      removeWalls(currentMazeBlock, nextCurrent);
      currentMazeBlock = nextCurrent;
    } else if (mazeStack.size() > 0) {
      Block nextCurrent = mazeStack.get(mazeStack.size() - 1);
      mazeStack.remove(nextCurrent);
      currentMazeBlock = nextCurrent;
    } else {
      print("\n\n************************\n");
      print("Maze Finsihed!");
      isMazeFinished = true;
    }
  } else {
    startSearchBlock.makeRect(255, 0, 0);
    finishSearchBlock.makeRect(255, 0, 0);
    //A* Search Algorithm Here
```

```
    if (!searchNeighborsAdded) { //Add neighbors only once
      print("\n\n\n************* Entring A* Search Algorithm
*************\n");
      for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
          print("\nAdding Neighbor for (" + i + ", " + j + ")\n");
          blocks[i][j].addMazeNeighbors();
        }
      }
      searchNeighborsAdded = true;
      //frameRate(1);
      print("All neighbors added\n\n");
      //Any inital setup before starting search algorithm.
      startSearchBlock.g = 0;
      startSearchBlock.f = heuristic(startSearchBlock, finishSearchBlock);
      openSet.add(startSearchBlock);
      print("Starting search\n");
    }
    if (openSet.size() > 0) { //runs until reaches a decision
      currentSearchBlock = lowestFinOpenSet();
      //currentSearchBlock.makeRect(255, 255, 255);
      if (currentSearchBlock == finishSearchBlock) {
        print("Path Found! Getting path.\n");
        pathFound = true;
        reconstructActualPath();
        for (int i = 0; i < actualPath.size() - 1; i++) {
          actualPath.get(i).makeLine(actualPath.get(i + 1), 255, 0, 0);
        }
        startSearchBlock.makeRect(0, 255, 0);
        finishSearchBlock.makeRect(0, 255, 0);
        print("Path printed. Stopping loop\n");
        noLoop();
      }
      if (!pathFound) {
        openSet.remove(currentSearchBlock);
        for (Block ngbr : currentSearchBlock.mazeNeighbors) {
          float tent_gScore = currentSearchBlock.g + 1;
          if (tent_gScore < ngbr.g) {
            ngbr.prev = currentSearchBlock;
            ngbr.g = tent_gScore;
            ngbr.f = ngbr.g + heuristic(ngbr, finishSearchBlock);
            if (!openSet.contains(ngbr)) {
              openSet.add(ngbr);
            }
          }
        }
        reconstructSearchPath();
        for (Block block : searchedPath) {
          block.makeRect(255, 255, 0);
        }
        startSearchBlock.makeRect(0, 255, 0);
        finishSearchBlock.makeRect(0, 255, 0);
      }
      for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
          blocks[i][j].show();
        }
      }
```

```
    } else if (pathFound) {
      print("No Path Found");
      noLoop();
    }
  }
}
void removeWalls(Block current, Block next) {
  int xDistance = current.thisRow - next.thisRow;
  int yDistance = current.thisCol - next.thisCol;

  print("X Distance: " + xDistance + "\n");
  print("Y Distance: " + yDistance + "\n");

  if (xDistance == -1) {
    current.walls[1] = false;
    next.walls[3] = false;
  } else if (xDistance == 1) {
    current.walls[3] = false;
    next.walls[1] = false;
  }
  if (yDistance == -1) {
    current.walls[2] = false;
    next.walls[0] = false;
  } else if (yDistance == 1) {
    current.walls[0] = false;
    next.walls[2] = false;
  }
}
ArrayList reconstructSearchPath() {
  Block current = currentSearchBlock;
  searchedPath.add(current);
  while (current != startSearchBlock) {
    searchedPath.add(current);
    current = current.prev;
  }
  return searchedPath;
}
ArrayList reconstructActualPath() {
  Block current = currentSearchBlock;
  actualPath.add(current);
  while (current != startSearchBlock) {
    actualPath.add(current);
    current = current.prev;
  }
  print("Path reconstructed, returning path\n");
  return actualPath;
}
float heuristic(Block from, Block to) {
  float distance = 0.0;
  distance = dist(from.thisRow, from.thisCol, to.thisRow, to.thisCol);
  return distance;
}
Block lowestFinOpenSet() {
  Block lowestFScore = openSet.get(0);
  for (Block block : openSet) {
    if (block.f < lowestFScore.f) {
      lowestFScore = block;
    }
```

```
  }
  return lowestFScore;
}
```

## Block.java

```
class Block {
  //general
  int x;
  int y;
  int thisRow;
  int thisCol;
  boolean[] walls = {true, true, true, true};
  //Maze variables
  boolean visitedByMaze = false;
  ArrayList<Block> neighbors = new ArrayList<Block>();
  //Search algorithm variables
  boolean visitedBySearchAlgo = false;
  ArrayList<Block> mazeNeighbors = new ArrayList<Block>();
  float g = 10000000000.0;
  float f = 10000000000.0;
  float h;
  Block prev;

  Block(int row, int col) {
    x = row * size;
    y = col * size;
    thisRow = row;
    thisCol = col;
  }
  void show() {
    if (walls[0]) { //draw top line
      line(x, y, x + size, y);
    }
    if (walls[1]) { //draw right line
      line(x + size, y, x + size, y + size);
    }
    if (walls[2]) { //draw bottom line
      line(x + size, y + size, x, y + size);
    }
    if (walls[3]) { //draw left line
      line(x, y + size, x, y);
    }
    if (visitedByMaze && !isMazeFinished) {
      noStroke();
      fill(255, 50, 255, 95);
      //fill(0, 255, 255);
      rect(x, y, size, size);
      stroke(0);
    }
  }
  void addMazeNeighbors() {
    if (!walls[3]) { //we are not in top row. Add top neighbor.
      print("Adding top neighbor\n");
      mazeNeighbors.add(blocks[thisRow - 1][thisCol]); //top neighbor
    }
    if (!walls[2]) { //we are not in rightmost column. Add right column.
      print("Adding right neighbor\n");
      mazeNeighbors.add(blocks[thisRow][thisCol + 1]); //right neighbor
```

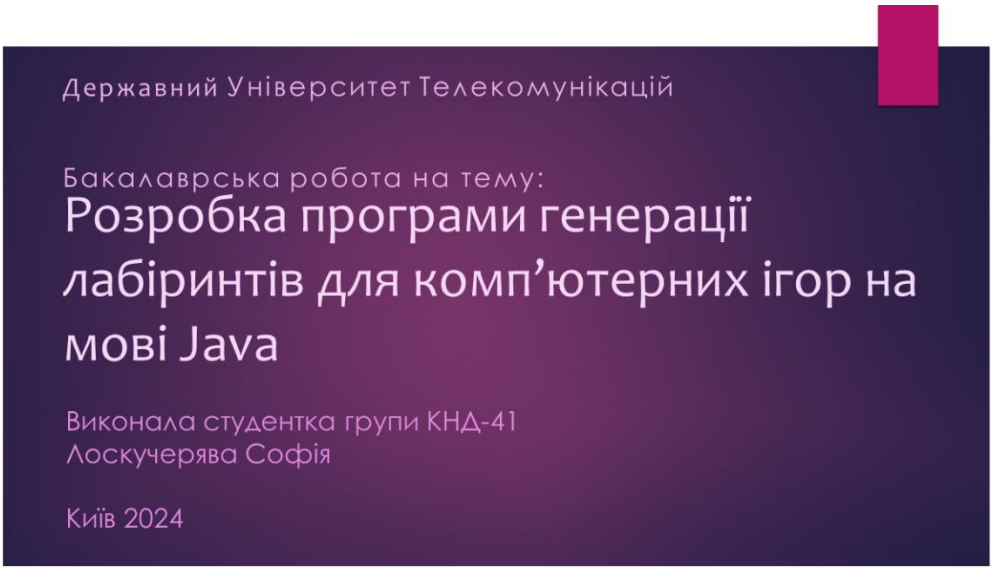
```
    }
    if (!walls[1]) { //we are not in bottom row. Add bottom neighbor.
      print("Adding bottom neighbor\n");
      mazeNeighbors.add(blocks[thisRow + 1][thisCol]); //bottom neighbor
    }
    if (!walls[0]) { //we are not in leftmost column. Add left column.
      print("Adding left neighbor\n");
      mazeNeighbors.add(blocks[thisRow][thisCol - 1]); //right neighbor
    }
  };

  void addNeighbors() {
    if (thisRow > 0) { //we are not in top row. Add top neighbor.
      neighbors.add(blocks[thisRow - 1][thisCol]); //top neighbor
    }
    if (thisCol < cols - 1) { //we are not in rightmost column. Add right
column.
      neighbors.add(blocks[thisRow][thisCol + 1]); //right neighbor
    }
    if (thisRow < rows - 1) { //we are not in bottom row. Add bottom
neighbor.
      neighbors.add(blocks[thisRow + 1][thisCol]); //bottom neighbor
    }
    if (thisCol > 0) { //we are not in leftmost column. Add left column.
      neighbors.add(blocks[thisRow][thisCol - 1]); //right neighbor
    }
  };
  boolean hasUnvisitedNeightbors() {
    print("\n\nChoosing neighbors for (" + thisRow + ", " + thisCol +
")\n");
    for (Block neighbor : neighbors) {
      if (!neighbor.visitedByMaze) {
        print("Found a neighbor!\n");
        return true;
      }
    }
    print("No neighbor found! Going back.\n");
    return false;
  }
  Block pickRandomNeighbor() {
    Block ngbr = neighbors.get(floor(random(0, neighbors.size())));
    print("Picked: (" + ngbr.thisRow + ", " + ngbr.thisRow + ")\n");
    while (ngbr.visitedByMaze) {
      print("This has been visited before, picking a new one.\n");
      neighbors.remove(ngbr);
      ngbr = neighbors.get(floor(random(0, neighbors.size())));
      print("Picked: (" + ngbr.thisRow + ", " + ngbr.thisRow + ")\n");
    }
    ngbr.visitedByMaze = true;
    neighbors.remove(ngbr);
    return ngbr;
  }
  void makeRect(int r, int g, int b) {
    noStroke();
    fill(r, g, b);
    rect(x, y, size, size);
    stroke(0);
  }
```

```
  void makeLine(Block to, int r, int g, int b) {
    strokeWeight(7);
    stroke(r, g, b);
    int x1 = x + size / 2;
    int y1 = y + size / 2;
    int x2 = to.x + size / 2;
    int y2 = to.y + size / 2;
    line (x1, y1, x2, y2);
    strokeWeight(4);
    stroke(0);
  }
}
```

# ДОДАТОК Г

## Слайди презентації




## Основні завдання роботи

- Проаналізувати існуючі алгоритми генерації лабіринтів та методи пошуку шляху та обрати потрібний для реалізації
- Дослідити особливості обраного алгоритму і методу пошуку,що його реалізують
- Розробити програму для генерації лабіринтів,який згенерирує не тільки сам лабіринт,але й в якості бонусу побудує шлях від початку до кінця

## Актуальність

- З кожним роком генерація лабіринтів в програмуванні стає все більш актуальною у зв'язку з розвитком ігрової індустрії, штучного інтелекту, а також вирішенням різних завдань у робототехніці. Із зростанням кількості ігор, де лабіринти виступають ключовим елементом, потрібно розробляти ефективні алгоритми їх генерації.

## Проблема

- 1.Ефективність: Необхідність створення алгоритмів, які можуть швидко та ефективно генерувати лабіринти різної складності та розміру, без значного впливу на продуктивність гри.
- 2.Реалізація: Розробка алгоритмів генерації, які забезпечують створення лабіринтів з різними характеристиками, такими як різноманітність шляхів, відсутність петель, ефективне використання пам'яті тощо.
- 3.Налагодження: Важливо забезпечити, щоб генеровані лабіринти були іграбельними та стимулювали гравця до активного дослідження, уникання створення нудних або нерозв'язних ситуацій.
- 4.Сумісність: Генератор лабіринтів повинен бути легко інтегрований з іншими компонентами гри, забезпечуючи зручну роботу з генерованими об'єктами лабіринту.
- Для подолання цих викликів потрібна ретельна розробка та тестування алгоритмів генерації, а також вдосконалення їх шляхом аналізу та впровадження змін відповідно до вимог ігрового процесу.Це не принижає їх ефективність, але зменшує їх доступність для загалу і вимагає більших економічних витрат

## Концепція

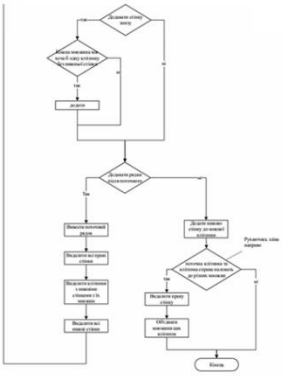
- Повинно бути створені конкретний програмний додаток на окремих вибраних програмах з мовою Java,де при запуску коду буде рандомно генерувати із клітинок повноційний лабіринт,а також в якості бонусу буде продемостровано в дії метод пошуку шляху в генерації лабіринту від початку до кінця.

## Ідея

- Ми пропонуємо розробити програму,що складається з рандомної генерації стінок лабіринтів на кожній клітинці за допомогою стандартних ресурсів мови програмування Java.
- У якості основи генерації лабіринту ми пропонуємо використати алгоритм Еллера,оскільки він простий у реалізації, може генерувати лабіринти різних форм і розмірів, і гарантує, що лабіринт буде мати єдиний шлях між двома точками

### Архітектура системи

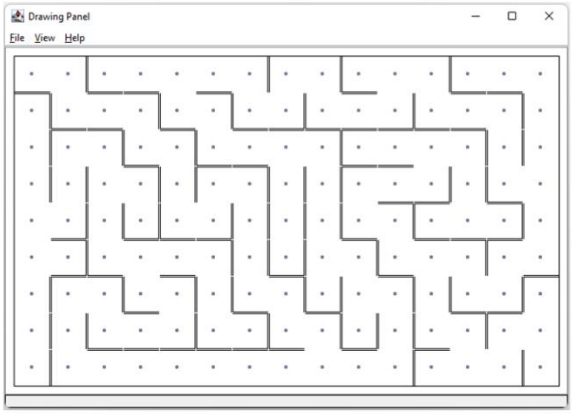
На малюнку зображено блок-схему алгоритму Еллера з кожною командою створення стінок і дозволяє процедурно генерувати одно маршрутний ідеальний лабіринт який не містить у собі циклів. Через специфіку запам'ятання лише останнього рядка, дозволяє генерувати нескінченно великий у довжину лабіринт.



## Результати тестування системи

В Eclipse розроблено стандартний варіант генерації лабіринту.

При запуску програми з'являється вікно,де буде генеруватися лабіринт кожної клітинки,де посередині стоїть крапка,навколо неї будується стінка і так для кожної клітини Якщо навести




# Коди,які були використані в Eclipse

```
package cells;

public class Cell {
    //WALLS
    public boolean top;
    public boolean bot;
    public boolean left;
    public boolean right;
    //IS CELLVISITED
    public boolean visited;
    //DEFAULT INIT OF CELL
    public Cell() {
        this(false, false);
    }
    //THIS CONSTRUCTOR CREATE ONLY FILLED OR ONLY EMPTY MAZE
    public Cell(boolean initState, boolean visited) {
        this(initState, initState, initState, initState, visited);
    }
    //THIS CONSTRUCTOR ALLOW TO CUSTOMIZE YOUR OWN RULES
    public Cell(boolean top, boolean bot, boolean left, boolean right, boolean visited) {
        this.top = top;
        this.bot = bot;
        this.left = left;
        this.right = right;
        this.visited = visited;
    }

    @Override
    public String toString() {
        return String.format(" cell has %b, %b, %b, %b. ", this.top, this.bot, this.left, this.right);
    }
}
```

**Cell.java**

```
//CREATE YOUR OWN MAZE
public Maze(int rows, int cols) {
    this.rows = rows;
    this.cols = cols;
    maze = new Cell[rows][cols];//Creates new maze with given size

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            maze[i][j] = new Cell(true, false);//Leave maze empty and unvisited
        }
    }
    buildBorders();//Build outer borders

    tableX = rows * side;
    tableY = cols * side;
    panel = new DrawingPanel(tableX + padding * 2, tableY + padding * 2);//Create canvas with given size
    g = panel.getGraphics();//Get painter tool of canvas for painting

    drawBorders();//Draw borders on canvas

}
//CREATE BORDERS
void buildBorders() {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (i == 0)
                maze[i][j].top = false;
            if (i == cols - 1)
                maze[i][j].bot = false;
            if (j == 0)
                maze[i][j].left = false;
            if (j == rows - 1)
                maze[i][j].right = false;
        }
    }
}
```

**Maze.java**

## Результати тестування системи

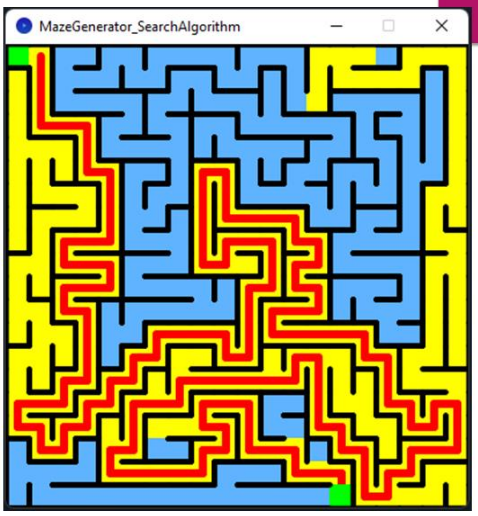
За допомогою програми Ruby та Sublime Text розроблено генерацію лабіринту в командній строці з командою зупинення генерації.

При запусканні програми він генерує дуже довгий лабіринт і буде генеруватися до тих пір,якщо не натиснеш конкретні клавіші,які зупиняє команду генерації(в нашому випадку-CTRL+C)

## Результати тестування системи

В якості бонусу в програмі Proccesing було розроблено генерацію лабіринту,який не тільки будує лабіринт,але й шукає шлях від початку до кінця.

Після генерації лабіринту зеленими квадратиками відмічається старт і фініш. Жовтими відмітками методом тику шукається шлях і коли шлях знайдено,червоною смугою відмічено шлях від старту до фінішу




# Висновки

- В бакалаврській роботі проведений аналіз існуючих алгоритмів та методів створення генерації лабіринтів
- Проведено дослідження особливостей алгоритму і методу,який дає можливість побудувати генерацію лабіринтів а також в якості бонусу знайти шлях від початку до кінця
- Розроблено програми на Eclipse,Ruby та Proccesing на основі запропонованих алгоритмів, яке повністю задовольняє мету роботи